RubberBand: Cloud-based Hyperparameter Tuning

Ujval Misra*
UC Berkeley

Richard Liaw*
UC Berkeley

Lisa Dunlap
UC Berkeley

Romil Bhardwaj
UC Berkeley

Kirthevasan Kandasamy
UC Berkeley

Joseph E. Gonzalez
UC Berkeley

Ion Stoica
UC Berkeley

Alexey Tumanov
Georgia Institute of Technology

Abstract

Hyperparameter tuning is essential to achieving state-of-the-art accuracy in machine learning (ML), but requires substantial compute resources to perform. Existing systems primarily focus on effectively allocating resources for a hyperparameter tuning job under fixed resource constraints. We show that the available parallelism in such jobs changes dynamically over the course of execution and, therefore, presents an opportunity to leverage the elasticity of the cloud.

In particular, we address the problem of minimizing the financial cost of executing a hyperparameter tuning job, subject to a time constraint. We present RubberBand—the first framework for cost-efficient, elastic execution of hyperparameter tuning jobs in the cloud. RubberBand utilizes performance instrumentation and cloud pricing to model job completion time and cost prior to runtime, and generate a cost-efficient, elastic resource allocation plan. RubberBand is able to efficiently execute this plan and realize a cost reduction of up to 2x in comparison to static allocation baselines.

CCS Concepts • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**.

Keywords Hyperparameter Optimization, Distributed Machine Learning

ACM Reference Format:
Ujval Misra, Richard Liaw, Lisa Dunlap, Romil Bhardwaj, Kirthevasan Kandasamy, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2021. RubberBand: Cloud-based Hyperparameter Tuning. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456245

*Equal contribution.

This work is licensed under a Creative Commons Attribution International 4.0 License

EuroSys '21, April 26–28, 2021, Online, United Kingdom
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456245

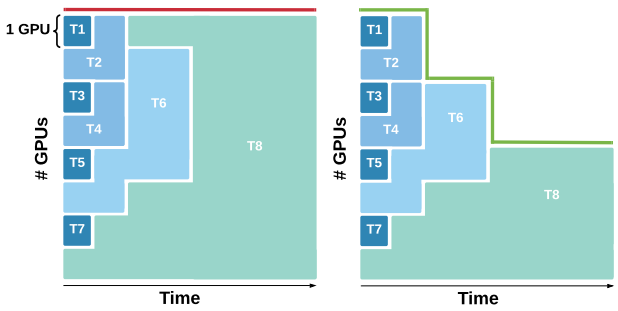


Figure 1. Elastic Hyperparameter Search. A comparison of a static allocation (left) and an elastic allocation (right) for the same tuning job. Each box represents the compute allocation to a trial for a number of epochs. The highest quality trial (green) is eventually allocated the entire cluster in the static allocation despite needing fewer resources to complete training within constraints.

1 Introduction

As state-of-the-art deep learning (DL) models have grown larger in the number of parameters [1, 2], so too have their price tags [3]. Because the dominant cost of developing these models lies in the process of *hyperparameter tuning* [4], it has become critical to reduce the cost of commonly employed hyperparameter tuning methods.

Hyperparameter tuning aims to determine the optimal configuration of hyperparameters of a model by repeatedly training it with different candidate configurations, and selecting the one that yields the highest accuracy. Despite its significant cost, hyperparameter tuning is essential to maximizing the performance of a DL model, since the accuracy of such models depends strongly on the choice of hyperparameters used [5, 6]. In this work, we therefore focus on the fundamental problem of *minimizing the cost of executing a hyperparameter tuning job, subject to a time constraint.*

To accelerate hyperparameter tuning jobs, state-of-the-art *early-stopping* techniques reduce the amount of work performed by terminating evaluation tasks expected to yield inferior quality configurations prior to their completion [7–10] or by dynamically deallocating compute resources (e.g., GPUs) from poorly performing hyperparameter configurations before they finish [11]. These techniques exploit the

available parallelism in hyperparameter tuning workloads to further reduce job completion time (JCT) through concurrent training and evaluation. However, such techniques thus far [9–12] have focused exclusively on exploiting parallelism over a *fixed* pool of compute resources.

In this paper, we demonstrate that early-stopping techniques exhibit poor utilization when running over fixed resource pools due to the changing available parallelism of hyperparameter tuning workloads. These techniques reallocate resources to a decreasing number of running candidates by terminating the worst performers. This leads to a *degradation in cluster resource utilization* since common parallel training methods incur significant communication overheads, and as such scale sub-linearly with compute [13–15].

Building on this observation, we show that it is possible to increase cluster utilization and reduce execution cost of hyperparameter tuning, while preserving JCT and model accuracy, by running it in the cloud. Leveraging the intrinsic compute elasticity of cloud platforms enables dynamically resizing of the provisioned cluster to adapt to the changing parallelism available in the job. By front-loading expenditure, we can allocate more resources to the earlier, highly parallel stages of a job and fewer to subsequent ones. This results in improved cluster utilization and a reduction in overall cost.

However, efficiently leveraging elastic cloud resources for hyperparameter tuning presents three key challenges. The first is to accurately model the execution of the job in the cloud, as a function of a *resource allocation plan*, which defines the number of parallel compute resources to allocate to each stage of the job. The second is to navigate the large search space of such plans to find a feasible, low-cost solution. The third is to cost-efficiently schedule and place workers over an elastically provisioned cluster at runtime.

First, the JCT and cost of a job are functions of its allocation plan. However, modeling this relationship can be challenging due to a variety of other factors that impact performance and spending. In particular, JCT is also a function of (i) the DL model's training latency and scalability over cloud hardware, (ii) cloud provider overheads, such as provisioning latency, and (iii) the overall computation structure of the job.

Similarly, execution cost is a function of a variety of factors, including the price of each resource provisioned, price of any ingress or egress data movement, and the billing model applied by the provider (e.g. per-instance or per-function) [16–19]. These factors vary not only across the different cloud providers (e.g. AWS, GCP and Azure), but also across the range of offerings within a single provider's ecosystem (spanning instance types, resource pools, regions etc.). It is therefore important to develop a comprehensive cost and runtime model that appropriately parameterizes these characteristics, to enable us to accurately evaluate the utility of an elastic allocation plan for a particular job.

Second, the system must be able to generate a feasible, low-cost allocation plan for a given job. Tuning a DL model that scales sublinearly with compute will likely require the solution to concentrate its spending and resources in the earlier stages of the job with greater available parallelism. Conversely, if the DL model being tuned scales relatively well with compute, the optimal solution may indeed be a static allocation, or even require late scale-up. The optimal solution therefore depends on a number of factors, including critically, the scalability of the model [15, 20]. The system must sufficiently explore the search space to ensure a feasible, cost-efficient solution is found.

Third, naive worker placement can negatively affect training performance [21–25], resulting in higher job execution costs. Parallel workers of a training job require co-location to avoid incurring unnecessary network overheads. Worker scheduling will therefore need to *optimize placement co-location* and maximize cluster utilization.

To address these challenges, we designed RubberBand, the first framework for cost-efficient, elastic execution of hyperparameter tuning jobs in the cloud. We focus on optimizing early-termination algorithms such as Successive Halving [7] and Hyperband [8] due to their strong theoretical foundations and relative popularity. Furthermore, because they are declarative in nature, we can plan their execution offline.

RubberBand introduces a DAG-based execution model that captures the various latencies and costs associated with executing a job in the cloud, over a given resource allocation plan. This model is parameterized by profiling DL model training latency and resource provisioning overheads prior to execution. RubberBand then searches for a feasible, cost-efficient resource allocation plan, utilizing the execution model to predict the end-to-end JCT and cost of the input job for each candidate plan explored. During execution, RubberBand elastically scales the underlying cluster according to the plan, dynamically allocates provisioned resources fairly among trials and manages the placement of individual workers for each trial. In summary, we make the following contributions:

1. We develop a cost and latency model that synthesizes profiling information about the input job and target cloud resources to evaluate the benefit of running hyperparameter tuning jobs over elastic resources.
2. We design and implement an elastic cost-minimizing resource allocation and placement policy for time-constrained hyperparameter tuning workloads.
3. We build a full system stack for hyperparameter tuning in the cloud and evaluate it with state-of-the-art DL models.

2 Background

The accuracy of a DL model can depend heavily on the choice of its *hyperparameters*. Hyperparameters are values that affect the learning dynamics of a model but are not directly optimized by the training procedure. We refer to a set of

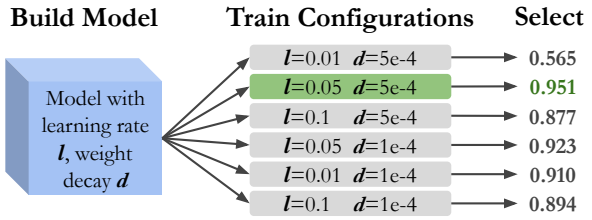


Figure 2. A basic hyperparameter grid search. Each trial initializes a model on a sampled hyperparameter configuration and trains it until its performance has converged. The configuration corresponding to the highest accuracy is selected.

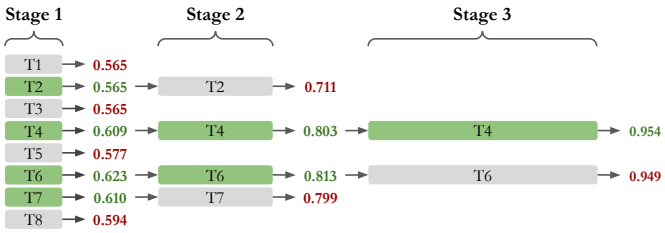


Figure 3. Successive Halving Procedure. An example of an early-stopping job produced by Successive Halving (with a reduction factor of 2). Each rectangle represents a trial, and its width indicates how many iterations it executes for in a stage.

hyperparameter values as a *hyperparameter configuration*, and the training/evaluation procedure associated with one such configuration as a *trial*. To improve DL model accuracy, practitioners run hyperparameter tuning *experiments* or *jobs*, which are composed of a collection of trials—each parameterized by a configuration sampled from a search space—with the objective of identifying the configuration that yields the best validation accuracy or loss (Figure 2).

The hyperparameter search space itself is a multi-dimensional distribution from which sample values of each hyperparameter (e.g., learning rate, momentum) are drawn. We note that both choosing the right search space and the optimal process of sampling from it are ongoing areas of research. For the scope of this work, we are not concerned with the methods used to design or navigate this space. Rather, we expect the user to provide a search space and sampling method (e.g. random search) for selecting hyperparameters.

During training, one can observe the progress and quality of the model via intermediate training metrics, such as accuracy or loss. These metrics generally exhibit diminishing returns, meaning that their rate of improvement generally decrease as training progresses. When evaluating multiple configurations, candidates that do not initially perform well can often be discarded after minimal training. However, because intermediate training metrics can be imperfect predictors of final model quality at convergence, identifying and training the best configuration from the top tier candidates can require an order of magnitude more resources [8].

Leveraging this insight, bandit-based hyperparameter tuning algorithms [7, 8, 10] have been developed to aggressively terminate poorly performing trials and reallocate available resources to better-performing ones that promise higher accuracy (Figure 3). These algorithms execute in sequential stages, where each stage constitutes a number of trials running independently, followed by synchronous evaluation and termination of the bottom-performing fraction. While a parallelization scheme is not explicitly prescribed, parallel execution is necessary in the context of real-world objectives and constraints such as cost and time.[1]

2.1 Distributed, Parallel Hyperparameter Tuning

Executing a hyperparameter tuning job cost-efficiently within constraints necessitates the efficient utilization of parallel and distributed resources. This parallelization can occur across two (nested) levels: distributed parallel evaluation of multiple independent trials at the stage-level, and distributed training at the trial-level.

Stage-level parallelism. Trials can be executed independently in parallel within each stage. Two phenomena cause the available parallelism at this level to change dynamically over the course of execution. First, as trials are progressively terminated at the end of each stage, fewer trials remain to be executed, fundamentally decreasing the parallelism available to be exploited. Second, the presence of stage-end synchronization barriers introduces a potential source for stragglers [9]. The scale of this effect changes in the reverse direction; earlier stages maintain a larger set of running trials and are therefore more prone to stragglers. Overall, however, the impact of the first phenomenon tends to dominate: aggressive termination of trials results in the available stage-level parallelism declining by up to an order of magnitude with each subsequent stage.

Trial-level parallelism. Deep learning models are frequently trained using a distributed, data parallel strategy. Due to the communication-heavy nature of this scheme, training performance does not scale linearly with compute allocated (Figure 4). In practice, performance scalability depends on several variables. Those specific to the training procedure itself include the model architecture, training batch size and the communication strategy used between data parallel workers (e.g. all-reduce, parameter server) [15, 20]. On the other hand, hardware variables include the type of the underlying compute resources utilized (CPUs, GPUs, TPUs, etc.), the quantity of these resources co-located on a single machine, the topology of the network and the communication bandwidth between machines .

[1]We focus our discussion on minimizing cost subject to a time constraint, but note that many of the techniques presented extend naturally to the related problem of minimizing job completion time subject to cost.

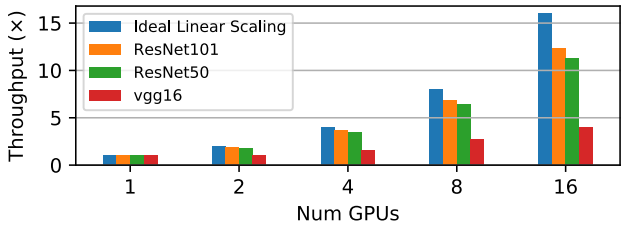


Figure 4. Scaling of deep learning models with increasing GPUs. Deep learning models exhibit sub-linear scaling as more resources are allocated. Throughput is normalized to throughput with a single GPU.

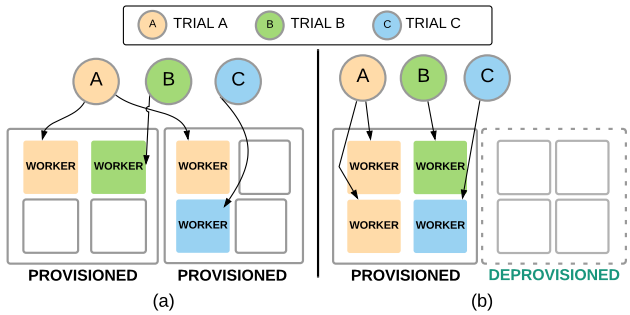


Figure 5. Worker colocation during hyperparameter tuning. A scattered placement of training workers across cluster resources can lead to low training throughput. By colocating workers and constraining the worker placement to use a minimal feasible set of nodes, nodes can be easily deprovisioned without interrupting the experiment, leading to more efficient spending.

Thus, existing techniques which naively reallocate *all* freed resources to running trials [11] can suffer from poor cluster utilization, as the available stage-level parallelism decreases. We are therefore interested in capturing the performance implications of these variables in an execution model. Doing so would allow us to evaluate the utility of an elastic allocation accurately and make better informed resource allocation decisions, given the specific characteristics of individual jobs.

While modeling job execution is necessary to determine the cost-optimal resource quantities to allocate, in practice the realized utilization of a particular allocation is not guaranteed to be efficient. Due to the structure of peer-to-peer communication between workers in data parallel training, placing workers on physically separate machines results in cross-node network transfers, which has been comprehensively shown to result in resource under-utilization [21–25]. Thus, a trial's workers must be co-located together when feasible, to ensure cluster resources are efficiently utilized.

Thus, the effective utilization of cluster resources by a distributed parallel hyperparameter tuning system is dependent on its ability to accurately predict the performance scalability of an input job's model training procedure, utilize the information to explore the allocation plan search space, and lastly, account for the co-location requirement of trial workers during execution.

2.2 Cloud Challenges and Opportunities

The cloud offers an elastic resource pool which can be dynamically scaled to adjust to the dynamically changing resource requirements of a job. Major cloud platforms charge for compute at a per-second billing granularity [16–18], and are able to service provisioning requests on the order of seconds across their various offerings. These properties enable cloud applications to make fine-grained provisioning decisions as their resource requirements vary, in order to optimize expenditure.

The key challenge that arises from leveraging the cloud to run hyperparameter tuning jobs is then integrating the pricing characteristics and overheads of the cloud provider into an execution model to enable accurate analysis of elastic resource allocation plans.

Cloud provider characteristics. Cloud providers maintain a diverse selection of compute offerings [26] that together produce a rich trade-off space between price and performance. The choice of offerings not only has impact an individual trial's training performance (as discussed in Section 2.1), but also on resource allocation planning.

The space of pricing options further extends the cloud cluster design space. Compute resources can be billed at per-instance [16] and per-function call [19] granularities. Per-instance pricing itself can be classified into the cheaper, pre-emptible spot and more expensive, uniterruptible on-demand sub-categories.

Both the pricing characteristics and overheads attributed to different compute offerings can impact the value in provisioning or deprovisioning resources elastically. These variables must therefore be accounted for *in conjunction with* the performance variables in Section 2.1, to produce a feasible, cost-efficient resource allocation plan for a given job.

By addressing these challenges, hyperparameter tuning methods that leverage the elasticity provided by the cloud can reduce execution costs without compromising on JCT or model performance.

3 Cost-efficient Hyperparameter Tuning

We are interested in optimizing a cost objective, subject to a time constraint. In this section, we will overview basic assumptions of our workload and discuss characteristics of existing naive solutions that can be improved upon.

Training assumptions. We assume that each trial executes an iterative training procedure, optimizing a common objective (e.g. validation accuracy or loss). Each iteration of this procedure returns intermediate training metrics and yields control to the underlying trial scheduler. Between

```python
class ExperimentSpec:

    def get_stage(self, stage_index: int) -> Tuple[int, int]:
        """Returns number of trials and iterations per-trial."""

    def num_stages(self) -> int:
        """Returns number of stages."""
```

Figure 6. Experiment specification API. Simple API between early-stopping algorithms and Rubberband. A single specification can express a successive halving job, whereas a collection of them can specify Hyperband-based methods as a multi-job.

iterations, the model and training procedure can be checkpointed, allowing the scheduler to pause, resume or terminate trials. Trials themselves are capable of leveraging distributed, parallel resources (i.e. accelerators such as GPUs) to reduce model training latency.

Additionally, we assume that all trials share the same scaling function. In other words, we assume hyperparameters do not have a substantial impact on training latency, and how training performance scales with compute. We note that hyperparameters do not affect throughput in many important real-world workloads, particularly those involving a fixed model architecture (e.g., fine-tuning). There are notable exceptions, including batch size and model architecture, which we defer to future work.

Unlike prior work which may change effective batch size with resource allocation [11], we assume that the batch size of the trial is held constant across execution (aka *strong scaling*) to a user-configured value. This ensures that the learning dynamics of the model are not affected by the system's resource allocation decisions [27]. To address GPU memory limitations when using a large effective batch size over a small resource allocation, we assume the job can leverage gradient accumulation.

Provider assumptions. For simplicity, instance type is provided by the user. We assume that compute instances of the specified type can be acquired from the cloud provider in sufficient quantities (i.e. provisioning requests are always served). Cloud provider prices and SLAs are treated as parameters and studied in Section 4.1.

We assume that the price of an instance is held constant throughout an experiment. While this may not generally be the case for cheaper instance types, prices of GPU-based instances experience negligible variance over long time periods [28, 29]. We also assume that billing occurs at a per-second granularity (with a 60-second minimum charge), as is the case for all major cloud providers [16–18].

3.1 Early-stopping Algorithms

In this work, we are interested in optimizing the cost of executing *early-stopping* hyperparameter tuning methods. One such method is Successive Halving [7], which is a commonly used algorithm in a variety of production machine learning systems [25, 30, 31].

Successive Halving (SHA) is a bandit-based approach which terminates running trials in a stage-wise iterative fashion, as shown in Figure 3. SHA assigns quantities of work (e.g. training iterations, epochs, data samples etc.) to trials fairly within a stage. The best $\frac{1}{\eta}$ trials are retained after each stage, while the per-trial work assignment is simultaneously increased by a factor of η. This exponential reduction in the number of trials, coupled with the exponential increase in the work per-trial results in a sharp decline in the overall available parallelism.

We note that for declaratively-defined early-stopping algorithms such as SHA, the structural characteristics of the experiment are known *prior* to runtime. This presents an opportunity to separate a declarative specification from the control flow, and optimize the resource allocation to every stage offline. Using the simple, yet sufficiently general experiment specification API outlined in Figure 6, we are able to acquire the number of stages, the number of trials per-stage and the amount of work (iterations) to be assigned per-trial in each stage upfront.

3.2 Resource Allocation

Early-stopping methods specify the amount of work to be done by each trial, but not how this work should be parallelized. A naive method to minimize cost within the limitations of using a fixed-size cluster, is to provision the smallest static cluster such that the expected JCT of the input job fits within the time constraint. Parallel resources can then be fairly allocated within each stage across running trials (following the approach in previous systems [9, 25]). However, inelastic provisioning can result in under-utilization of compute resources for two reasons.

Sublinear scaling. As illustrated in Section 2.1, allocating additional resources to trials leads to reduced efficiency because of sublinear scaling. In the initial stages, a large number of trials fairly share a relatively small number of parallel resources, thus resulting in efficient utilization. However, as trials are subsequently pruned, resources are relinquished. Since the cluster is fixed in size, the relinquished resources cannot be deprovisioned and must be reallocated to the current set of running trials. While this increases the throughput of the beneficiary trials, it can be more cost-effective to simply deprovision resources and operate at lower but more efficient throughput.

Synchronization. Secondly, jobs are executed in a synchronous-parallel fashion. At the end of each stage, all trials are compared in order to promote top performers and terminate bottom performers. As a result of this synchronization step, stages are prone to stragglers. With a static allocation, several resources—held by completed trials—may be left idle at the end of each stage.

4 Cloud-based Hyperparameter Tuning

A finer-grained, elastic allocation plan is capable of alleviating the resource efficiency concerns associated with using a static cluster allocation (Section 3.2). Therefore, given a job specification and time constraint, we want to find a feasible, cost-efficient resource allocation plan over cloud resources and execute the job without incurring unforeseen costs or overheads. In this section we outline our solution to the following three key challenges:

First, it is necessary to develop an execution model that allows us to accurately evaluate the quality of a candidate allocation plan. Such a model must synthesize (i) the DL model's training latency and scalability over cloud hardware, (ii) cloud provider overheads, (iii) job specification, (iv) resource allocation plan, (v) and cloud provider pricing characteristics. RubberBand models the execution of a job as a directed acyclic graph $G = (V, E)$ of tasks V (with associated latencies and costs) and task dependencies E. This provides an end-to-end representation of a job's execution, and can be used to predict its JCT and total cost.

Second, the search space of feasible allocation plans is exponentially large in the number of allocation decisions to be made (i.e. the number of stages in the job). RubberBand therefore relies on a greedy heuristic planner, which leverages the simulator as a black box to guide its search. The planner outputs an allocation plan as a vector $\vec{a} \in \mathbb{N}^{|E|}$ where $|E|$ is the number of stages defined in the specification and $\vec{a}_i$ is the number of resources to be allocated to the job during the execution of stage i (shared fairly between running trials).

Finally, the performance of the workload is sensitive to the placement of trial workers on physical resources and optimizing solely for locality can be expensive. RubberBand compiles the allocation plan generated by the planner in terms of resource quantities into a utilization-maximizing placement plan, mapping trial workers to devices during execution.

4.1 Modeling Variables

The performance of the DL model, overheads in the cloud and provider pricing characteristics can impact the optimal resource allocation plan. We therefore parameterize our model with various cost and performance variables to ensure RubberBand can accurately and flexibly model a job's execution. We later utilize this model during allocation planning to predict job completion time and cost.

Cost modeling. There are three primary modeling parameters that impact the total incurred job execution cost, leading to differences in the optimal allocation plan.

Compute price determines the price of each allocable unit of compute, per-unit time. Pricing is typically commensurate with the number of co-located resources in the allocated unit [16]. For example, one on-demand EC2 p3.2xlarge instance with 1 GPU may cost ~$3 per-instance hour, whereas a p3.16xlarge instance with 8 GPUs may cost ~$24 per-instance hour.

While this billing model is the most common [16–18], recent systems have introduced alternative models with finer-grained pricing granularity [19]. In such models users pay *per-function* rather than *per-instance*, for only the specific resources requested on the underlying hardware per-unit time their function runs for. We therefore consider *billing granularity* as a parameter to study the impact of recent cloud trends on tuning workloads. In particular, we consider the traditional *per-instance billing* granularity, along with *per-function granularity*, which approximates newer models that offer finer-grained pricing and compute elasticity.

We also consider *data price*, which determines how much users pay per-GB of ingress data movement, such as for reading training data from cloud storage. We note that data movement is often free of cost (e.g. within an EC2 region). However, users do not always have operational permission to co-locate compute with data, and in such cases data access can incur high costs. We treat it as a parameter for this reason.

Performance modeling. There are three modeled sources of latency: *training latency*, *provider queuing delay*, and *instance initialization latency*.

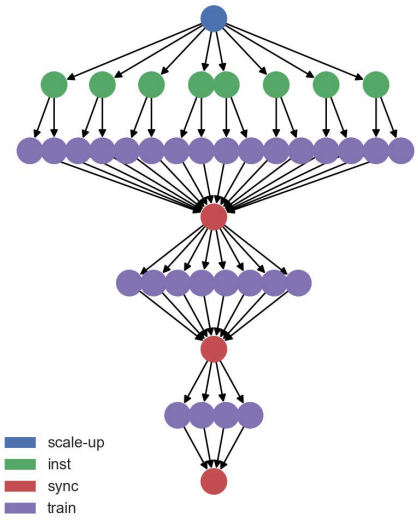
Training latency reflects (1) the initial latency of loading checkpoints and establishing connections between peer-to-peer training workers, and (2) the time it takes per step of all-reduce SGD to occur. The causes of variance for this latency measure can include, for example, variance in network latency and variance in data loading. This value depends on the batch size per update step. It also depends on the number of parallel training workers—we therefore capture this latency as a function of the number of resources allocated.

Scaling latency captures queuing delay at the provider. More specifically, it is the the time it takes from a job to request a resource to it being provisioned and made ready to access. *Instance initialization latency* reflects additional time for a provisioned resource to become ready for use by the job—this includes time to install dependencies and initialize cluster state. If these overheads are large in the context of the time constraint, increasing the cluster size of a job mid-execution is unlikely to be cost-efficient. Under such circumstances, an ideal allocation plan would minimize these latent costs by avoiding mid-job scale-up.

4.2 Execution Model and Simulator

RubberBand's simulator is parameterized by the modeling variables discussed in Section 4.1. The instantiated simulator is then responsible for synthesizing a DAG-based execution model from a given input experiment specification and resource allocation plan.

Each node in the DAG represents a task carried out by either the job (e.g. training the DL model for a number of




```
spec = rb.EmptyExperimentSpec() \
            .add_stage(num_trials=16, iters=16) \
            .add_stage(num_trials=8, iters=32) \
            .add_stage(num_trials=4, iters=64)
plan = rb.compile_plan(spec, model_profile, cloud_profile, deadline=4)
rb.execute(plan, trainer, search_space)
```

Figure 7. Example DAG representation. The plan generated is represented by the simulator as a DAG of system and job tasks. Each task has an associated cost and latency. This DAG is used to simulate the latency and cost of executing a particular plan. DAG node types are explained in Section 4.2.

iterations) or RubberBand's execution layer itself. We enumerate the types of these nodes below. Edges between nodes represent task dependencies.

1. **SCALE**: Represents a system task to provision a resource from the cloud provider.
2. **INIT_INSTANCE**: Represents a system task to initialize the instance after provisioning (e.g. installing dependencies).
3. **TRAIN**: Represents a trial task to train a model for the specified number of iterations, given the specified number of resources. Each stage in the DAG consists of one or more *TRAIN* nodes.
4. **SYNC**: Represents a synchronization barrier to evaluate trial quality. This indicates the end of a stage.

DAG construction. The simulator constructs the DAG by parsing the specification and allocation plan together stage-by-stage, extending dependency edges from the frontier (i.e. nodes with out-degree of zero) in each step. For each stage, cluster scaling nodes are first added if provisioning new nodes is necessary. This is followed by adding parallel training nodes and a synchronization node to end the stage.

If the resources provisioned preceding the stage being parsed are insufficient to meet the stage's allocation requirement, a blocking *SCALE* node is added to extend the previous frontier, followed by the necessary number of parallel *INIT_INSTANCE* nodes to sufficiently grow the cluster.

A parallel *TRAIN* node is then added for each trial that can run in parallel. If the cluster is too small to run all trials in parallel, each queued trial is represented by a *TRAIN* node with a serial dependency on a previously run trial. For example, a stage allocation of 1 resource would cause trials to run serially, and would be appropriately represented as sequentially linked nodes.

Stages are always concluded with a *SYNC* node, with all nodes on the frontier (i.e. training nodes) added as dependencies. An example DAG construction is illustrated in Figure Figure 7. In this example, the cluster was sufficiently scaled in the first stage. Note that low latency events with no cost (e.g. deprovisioning) are unrepresented. [2]

Simulation. Each node type of the graph has a probability distribution associated with its latency and a derived distribution associated with its cost. These distributions are parameterized by the aforementioned modeling variables (Section 4.1).

To predict JCT using this DAG, we sample latencies from the distribution of each node and compute the critical path (shown in Algorithm 1). We take a fixed number of samples and output an average.

The estimated cost of a given plan depends on the billing method. Per-function billing requires simply sampling the cost distribution for each billable node and summing them together. In per-instance billing we first sum non-compute related costs. We then add the cost of compute separately by determining the critical path within each stage. This is necessary because compute resources can be held past the lifetime of a trial (i.e. *TRAIN* node) due to stragglers.

4.3 Resource allocation planner

RubberBand's resource allocation planner is responsible for generating a feasible resource allocation plan that minimizes the predicted execution cost.

To do so, the planner warm-starts its optimizer with a *feasible* solution. This plan is then improved in a iterative-greedy fashion by (1) generating a set of new candidates from the current best, (2) predicting each of their JCT and cost using RubberBand's simulator, (3) selecting the best candidate, and (4) iterating until the best candidate generated is no longer predicted to improve cost, or violates the time-constraint. We outline these steps in Algorithm 2 and elaborate below.

Candidate generation. In each greedy step, candidate allocation vectors $\vec{a}^1 ... \vec{a}^{|E|}$ are generated from the current

[2]Node types can be easily added to capture additional latencies and costs.

best solution, $\vec{a}^*$. Each candidate $\vec{a}^i$ is equivalent to $\vec{a}^*$ at every index, except at i, where the allocation from $\vec{a}^*$ is decremented by a step size. The step size is set to be the smallest integer value such that the new stage allocation is either a factor or multiple of the number of trials, to ensure that resources can always be fairly divided.

Greedy selection. The planner selects the allocation with the largest predicted cost-marginal benefit. This marginal benefit is calculated by:

$$m_{\text{i}} = \frac{C(\vec{a}^*) - C(\vec{a}^i)}{T(\vec{a}^i) - T(\vec{a}^*)} \tag{1}$$

where $\vec{a'}$ is a candidate vector, and T and C predict cost and JCT respectively. We normalize cost reduction by the corresponding increase in JCT to ensure a fair comparison across candidates (since the step size varies to maintain invariants).

JCT and cost predictions are produced by the simulator, as described in the previous section. Simulator invocations are abstracted away in `select_best_candidate` (Algorithm 2).

Warm start. The plan optimizer must be warm-started with a feasible solution. In practice, one can be easily found by first solving the simpler problem of finding the cost-optimal static allocation. Since the search space is reduced to a single dimension, we can enumerate candidate static allocations from 1 to a large n, predict their costs and JCT, and return the cheapest feasible choice.

Because the optimizer never generates a candidate by *increasing* allocations, the initial solution constrains the maximum allocation to each stage. While this is often sufficient, we expand the search space to evaluate more candidates by invoking the optimizer on different warm-start solutions (e.g. 1x, 2x, 3x the optimal static allocation size). Of the final set, we choose the plan with minimum predicted cost.

Solution guarantees. Although this algorithm does not guarantee finding a cost-optimal solution, it does, however, guarantee that the solution found is predicted to do no worse than the warm-start solution. Since the optimizer can be warm-started with the optimal static cluster allocation, it can therefore be expected to always do just as well or better. As we show in Section 6, in practice it improves upon the cost of such allocations by an order of magnitude.

4.4 Placement Controller

During the execution of the workload, the *placement controller* will realize the resource allocation plan by managing the scheduling and coordinating the placement of trial workers across provisioned resources. Specifically, the placement controller will convert the resource quantity allocated to each trial into physical resource assignments for its workers. The output of this controller is *a placement plan* mapping trial workers to the host addresses and worker GPUs.

The algorithm underlying the placement controller is designed to maximize *spatial locality* given the available cluster resources. Parallel workers of a trial should be either colocated on a single machine or packed onto a minimal set of nodes. By colocating workers, the distributed training algorithm will avoid incurring unnecessary network overheads.

Algorithm 1: Simulate total plan duration

Input: Experiment spec E, allocation plan $\vec{a}$, scaling profile S, cloud profile C, number of samples n
Output: Simulated duration of plan
Function `simulate_duration` $(E, \vec{a}, S, C, n)$
  $G \leftarrow \text{to\_dag}_{S,C}(E, \vec{a})$;
  return $\frac{1}{n} \sum_{i=1}^{n} \text{sample\_duration}(G)$
Function `sample_duration` (G)
  $V \leftarrow \text{topological\_sort}(G)$;
  $l_1 \leftarrow \text{sample\_latency}(v_0)$;
  for $i \leftarrow 2 \ldots n$ **do**
    $P_i \leftarrow \text{predecessors}(G, v_i)$;
    $l_i \leftarrow \text{sample\_latency}(v_i) + \max_{v_j \in P_i}(l_j)$;
  return l_n

Algorithm 2: Compile resource allocation plan

Input: experiment specification E, model scaling profile S, cloud profile C, time-constraint t, warm-start plan $\vec{a}^0$
Output: resource allocation vector $\vec{a}^*$
Function `optimize_plan`$(E, S, C, t, \vec{a}^0)$
  $\vec{a}^* \leftarrow \vec{a}^0$;
  while *true* **do**
    $A \leftarrow \text{generate\_candidates}(\vec{a}^*)$;
    $\vec{a} \leftarrow \text{select\_best\_candidate}_{E,S,C}(A)$;
    if cost improvement $< \delta$ or violating t **then**
      break;
    $\vec{a}^* \leftarrow \vec{a}$;
  return $\vec{a}^*$

4.4.1 Placement Controller Algorithm

We assume that the total sum of allocations in an allocation vector is less than or equal to the size of the cluster, meaning that the provisioned resources will be anticipated and acquired ahead of time. We also assume a homogeneous instance pool—all worker instances have the same number of GPU resources and same type of GPUs. Under these assumptions, the placement algorithm is given the following parameters:

1. List of trials and their resource allocations
2. Previously generated physical placement plan
3. Current cluster state.

The algorithm first checks if the resource allocations are the same as the current placement plan. If not, it will attempt to place the trials with new resource allocations on the cluster. For each trial to place, it will first try to assign the trial to nodes that can fit the new allocation. If this does not suffice, trials with smaller allocations will be *displaced* to fit the trial.

Algorithm 3: Placement controller algorithm.

Input: Trial Allocation mapping A, the last placement plan *lastPlan*, list of nodes *nodes*, list of trials that have not confirmed placement *reserved*
Output: A new placement plan *placement*
Function *PlacementController*

```
if A satisfied by placement then
    return placement
trialsToMove = trials in A not satisfied by placement;
placement.remove_discrepancies(A);
for t ← trialsToMove.sort_by_alloc(descending) do
    if placement.full() then
        break
    alloc = A[t];
    while nodes.canFit(t.unit) do
        node = nodes.get_best_fit(t.unit);
        node.allocate(trial, resources=t.unit);
        alloc -= t.unit() ;
        if alloc == 0 then
            break
    for node ← nodes.sort(by=availSpace) do
        displaced = node.tryMakeSpace(trial, A);
        if displaced then
            node.allocate(trial, resources=trial.unit);
            alloc -= trial.unit ;
            trialsToMove += displaced;
        if alloc == 0 then
            break
return placement
```

Each of these trials will have an opportunity to be placed; placed trials cannot be displaced.

The algorithm then compares the proposed new allocation plan to the current placement plan, identifying trials that do not need to change in resource allocation. Then, upon a best-effort basis, it will preserve those resource assignments across scheduling epochs.

To achieve *spatial locality*, for each trial, the placement scheduler ensures that trials that have a resource allocation less than the node size is fully placed on the node. Otherwise, the trial will acquire one or more nodes to itself. To avoid resource over-subscription, the controller also maintains a list of trial placements that have been reassigned but have not been acquired yet. These resources are "locked" and cannot be perturbed in this scheduling epoch.

5 Implementation

We implement RubberBand on top of a recent version of Ray [32] (0.9.0), leveraging the framework's actor API to orchestrate distributed training jobs, and its autoscaling API to interact with the cloud provider. RubberBand runs a driver process on the head node, consisting of a scheduler, placement controller and cluster manager (Figure 8). This process is responsible for coordinating the experiment's execution.

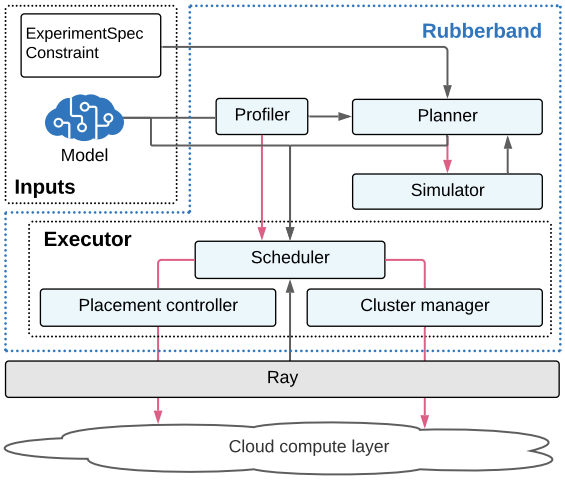


Figure 8. RubberBand system architecture. Pink arrows represent invocations, gray arrows represent object dependencies. The planner compiles the experiment specification to a resource allocation plan, based on the constraint and profiling information. The produced allocation plan is executed by the executor.

Planning. Prior to execution, RubberBand runs a profiling step for a configurable period of time to collect instrumentation data associated with model training and resource management. Because DL model training involves extremely repetitive operations with predictable performance, this can be done on the order of minutes. This involves iteratively scaling up the resource allocation to a trial by powers of two and measuring training latencies for each allocation. The data is aggregated to interpolate an estimated training latency scaling function of the model, and either utilize the mean or fit latency distributions for other various operations.

This information is then utilized in planning to simulate the latency and cost of executing candidate resource allocation plans. The simulator is initialized with the scaling function and fitted latency distributions. The number of samples drawn for each plan simulation request is configured to be small by default to ensure plans are generated quickly, but can be increased to trade-off planning performance for better simulation results.

Trial life-cycle. A trial consists of one or more gang-scheduled workers, each of which is responsible for managing one model replica. One worker from this group is additionally responsible for coordinating the trial's training procedure and reporting intermediate progress metrics back to the driver. Each worker is a Ray Actor, running in its own Python process. Workers are initialized by invoking user-provided functions that instantiate a model, optimizer, learning rate scheduler, dataset, and loss function, based on the trial's associated hyperparameter configuration.

Model training is performed in iterations. The driver invokes the trial's training API method, which trains the model on a fixed number of data samples and evaluates it on a validation dataset, returning the progress metrics produced. The trial distributes training work to its workers by leveraging PyTorch's native distributed data parallel training module [33].

Between training iterations, trials can be checkpointed, migrated and restored. Due to the symmetric nature of synchronous distributed data parallel training, only one worker needs to save its state. The checkpoint is constructed with the state of the model, optimizer, learning rate schedule, and other training metadata. It is then serialized and persisted in Ray's shared-memory object store, a reference to which is returned to the driver. During trial migration, newly instantiated workers use this reference to fetch the checkpoint from the store and restore their state.

Scheduling and placement. RubberBand's executor comprises of a control loop that makes scheduling decisions based on the experiment specification, resource allocation plan and the current state of running trials. Every iteration of this loop, the scheduler has an opportunity to start, continue running, pause or terminate a trial. The scheduler is also able to request the cluster manager to provision new resources or deprovision existing ones.

During execution of a stage, if the cluster size is greater than or equal to the number of planned trials, the scheduler runs all of them in parallel, allocating stage resources fairly among them. However, if the cluster size is to small to do so, each resource is assigned to a single trial until it is completed, queuing unscheduled trials until resources are freed.

RubberBand uses Ray's custom resource labels for actors to control the placement of workers. To resize or reassign resources to a trial, RubberBand checkpoints the state of the trial, destroys all of its workers and creates new ones with updated resource labels. The trial is then reassigned to the new workers, where the training procedure is restored from the checkpoint.

Cluster management. We extend Ray's autoscaler to implement a cluster manager which supports ad-hoc requests to scale the cluster size and tracks the total cost of provisioned compute during its lifetime . We use the `boto` [34] API under-the-hood to provision and deprovision instances from AWS EC2. We also use boto to fetch the price of an instance prior to execution, and assume that this price holds constant for the entire job duration.

The cluster manager utilizes a configuration file provided by the user to service provisioning requests. This file specifies the instance types and machine images (AMI) to be used for both the driver and worker instances, as well as any dependency installation scripts to run upon machine initialization. Once a provisioned instance becomes available, these scripts are automatically run by the manager. The node is subsequently added to the active Ray cluster.

6 Evaluation

In this section, we (1) evaluate static and elastic policies over various DL models and compute platform characteristics, in simulation; (2) perform an ablation over the optimizations made by components of RubberBand; and (3) demonstrate up to 2x improvement in cost over static baselines end-to-end, with negligible deviaion in cost or JCT from simulated predictions.

We refer to RubberBand with no changes, as the elastic policy. The static policy is implemented by replacing RubberBand's elastic resource allocation planner (described in Section 4.3) with a static allocation baseline planner (described in Section 3.2). This baseline method finds the *cost-optimal* static allocation that can execute the job within constraints.

All experiments which evaluate the static and elastic resource allocation policies are performed on SHA-generated [7] experiment specifications. SHA parameters of note are (1) the number of trials n, (2) the minimum number of training iterations to assign each trial r, (3) the maximum number of training iterations to assign R (to at least 1 trial) and (4) the termination rate η (fixed to 2, unless otherwise specified).

6.1 Simulated Experiments

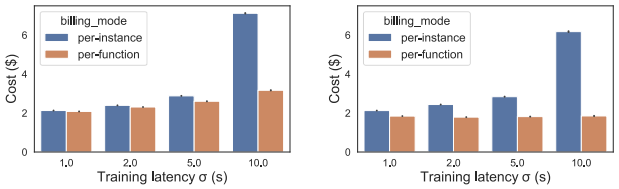
In this section, we perform simulated experiments to evaluate the cost-efficacy of the two aforementioned policies on SHA-based workloads. We leverage the parameterized cloud model to investigate the effects of different cloud characteristics upon using each policy. Specifically, we aim to evaluate the following factors on policy performance:

1. The impact of stragglers
2. Impact of the *job size*
3. Impact of the *data I/O pricing*
4. Impact of *instance initialization latency* on algorithm performance

Each experiment has two sets of parameters: *job parameters*, which include parameters to SHA, scaling, time constraint, and *modeling parameters*, which include cloud provider characteristics and profiling latencies.

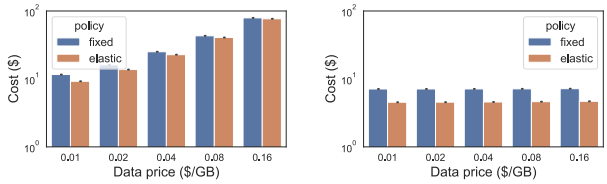
6.1.1 Impact of Stragglers

In this experiment, we measure the impact of stragglers on the cost objective for the two billing models. We compare a *per-instance* billing model (account for running time cost of each started instance) to a *per-function* billing model (only accounts for the cost of a utilized resource). We generate a specification using SHA($n = 64, r = 4, R = 508$). We use the scaling performance of a ResNet50 model [35] with a batch size of 512 using a cluster of EC2 p3.8xlarge workers. We generate stragglers by sampling training latency (per-iteration) from a normal distribution with $\mu = 4$ seconds, and vary σ from 1 to 10. Instance initialization latency is set to a constant 0 seconds. Results are shown in Figure 9.



(a) Fixed-cluster policy **(b)** Elastic policy

Figure 9. Impact of stragglers on simulated cost under different billing regimes. We simulate a SHA job with 64 ResNet-50 models running over p3.8xlarge instances. Stragglers are simulated by increasing variance of the training function latency distribution. Execution under a pay-per-instance regime exhibits higher costs due to under-utilization of resources at synchronization points, regardless of allocation policy.



(a) ImageNet **(b)** CIFAR-10

Figure 10. Impact of data I/O pricing on overall experiment cost for small and large datasets. The cost associated with I/O dominates experiment cost when the dataset is large or when the I/O pricing is high, resulting in little improvement relative to a fixed-cluster baseline.

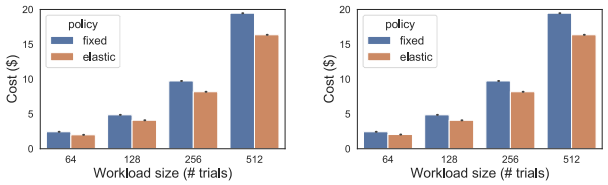
As seen in Figure 9, per-instance billing is an order of magnitude more expensive than per-function billing for both policies when there is large variance in training time. This is stragglers lead to RubberBand having to hold under-utilized resources at synchronization points. With per-function billing this is not the case because resources are released as soon as a trial completes its work.

6.1.2 Impact of data I/O pricing

In this experiment, we measure the impact of data I/O pricing on the overall execution cost. We assume that the only data transfer charged is that of downloading the dataset from an external store (e.g. S3), and occurs once per-instance. We compare the policies on both small and large datasets.

We generate a specification again using SHA($n = 64, r = 4, R = 508$). We also use the scaling performance of a ResNet50 model with a batch size of 512 using a cluster of AWS p3.8xlarge workers.

As illustrated in Figure 10, the benefit of elastic allocation is limited to jobs where the price of compute dominates the total experiment cost. Downloading ImageNet, a dataset of size 150GB, from S3 to a single EC2 instance at a price of just \$0.01 per GB costs \$1.50. This cost multiplies in a distributed environment where instances require local copies of data.



(a) Pay-per-instance **(b)** Pay-per-function

Figure 11. Simulated cost of executing SHA workload while increasing the number of trials. The elastic cluster allocation/algorithm is able to reduce costs across different trial counts, regardless of the billing model.

As a result, when the majority of expenditure is on data movement, the benefit of elastic resource allocation is diminished. However, we note that the elastic policy never does worse than the static one. In comparison, with CIFAR-10—a dataset of size only 150MB—data movement cost is negligible. We therefore observe a modest ~1.5x improvement in cost even at the relatively expensive data price of \$0.16 per-GB.

It is important to note that in typical cloud workloads, data transfer is free of cost. However this is not always the case—for example, in the case when transferring data between physically separate datacenter regions [36].

6.1.3 Impact of Job Size

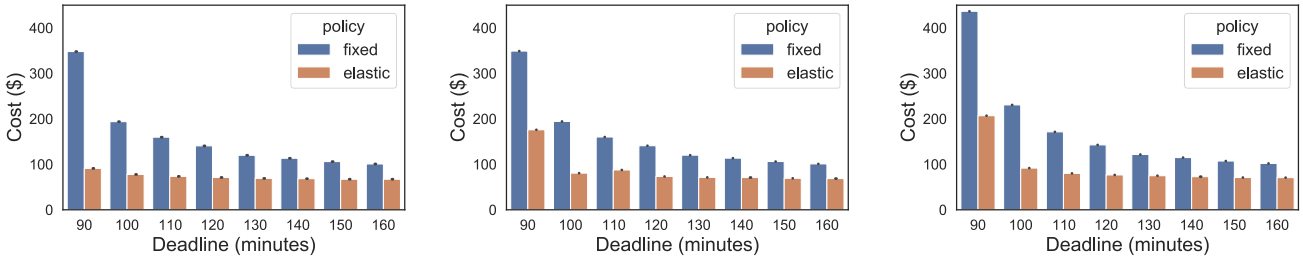
In this experiment, we measure how the cost-efficiency of the policies scale with job size. Specifically, we sweep the *number of trials* being evaluated by the job. We use SHA($n = k, r = 4, R = 508$) where k is varied as illustrated in Figure 11. We use the scaling performance of a ResNet50 model with a batch size of 512, using a cluster of EC2 p3.8xlarge workers. The time constraint is set to 20 minutes.

The results of this experiment are illustrated in Figure 11. We see that across the various workloads, the elastic cluster allocation policy always outperforms the fixed cluster baseline. Furthermore, the difference in the total execution cost increases as the number of trials increase. This is because an increase in the number of trials to evaluate is directly correlated with an increase in the overall available parallelism to exploit. However, for the static policy to leverage this increased parallelism in the earlier stages, it would have to incur worse utilization in the later stages.

6.1.4 Impact of Initialization Latency

In this experiment, we measure the impact of different initialization latencies on the cost objective, across different workload time constraints. Results are shown in Figure 12.

We use a *per-instance* billing model and use SHA($n = 512, r = 4, R = 4096$) to generate the experiment specification. We also use the scaling performance of a ResNet50 model with a batch size of 2048 using a cluster of AWS p3.8xlarge workers. The mean training latency (per iteration) is set to



(a) 1s init latency **(b)** 10s init latency **(c)** 100s init latency

Figure 12. Simulated cost of executing SHA on 512 ResNet-50 models over p3.8xlarge instances, with static and elastic policies. In the presence of minimal initialization overheads, we observe up to a 3.5x reduction in cost with the elastic policy. When overheads exist, the policy prices in the cost of scaling up and generates a different plan, leading to higher costs.

12 seconds, while we sweep initialization latencies of 1, 10, and 100 seconds. For each latency setting, we sweep the job's time constraint from 90 to 160 minutes.

In Figure 12, we see that as the initialization latency increases, the cost advantage of the elastic policy relatively decreases. For example, increasing from a 1s latency to 10s has no impact on the static policy, but increases the cost of the elastic policy by almost 2x (at the 90 minute time constraint). This is because such latencies reduce the efficacy of scaling to large cluster allocations.

Nevertheless, the elastic policy is still predicted to perform an order of magnitude better at the tightest time constraints.

6.2 Ablation Experiment: Placement Controller

As seen in Table 1, we evaluate the benefit of the placement controller, which is designed to (1) co-locate parallel workers of each trial on as few machines as possible, and (2) deprovisioning resources safely by bin-packing trials prior to cluster scale-down events.

All experiments in this section are run on EC2 using p3.16xlarge instances, each of which provides 8 NVIDIA Tesla V100 GPUs at the hourly price of $7.50-per-instance—the price at time-of-writing. We utilize a single r5.4xlarge instance to coordinate trials and host model checkpoints. [3]

We use a fixed batch size of 2048 throughout the entire experiment. At smaller resource scales, the model is unable to process the entire batch size at once. Therefore, we use *gradient accumulation* to ensure that the batch size does not change during learning.

We benchmark the sample throughput of different trials using RubberBand on an end-to-end workload with and without the placement controller. Without the placement controller, RubberBand delegates placement of workers to the underlying scheduler without indicating location preferences. As seen in Table 1, placement allows throughput to scale almost linearly across resources (~3.8x), while the placement-unaware baseline achieves only a ~1.8x speedup.

[3] In practice, the price of CPU instances is negligible in comparison to that of GPU instances, and is therefore ignored for the purposes of this evaluation.

# GPUs	Placement	No Placement
1	749.58 ± 23.97	673.76 ± 13.92
2	1480.07 ± 152.23	947.76 ± 397.19
4	2773.04 ± 349.72	1209.51 ± 357.58

Table 1. Placement Controller sample throughput. We measure the sample throughput (samples per second) for a ResNet50 model with a batch size of 1024 across different worker sizes on a cluster of p3.16xlarge instances. We see that without placement control sample throughput scales poorly, and is over 2x slower.

6.3 End-to-end Experiments

In a series of end-to-end experiments, we now evaluate (1) how well RubberBand performs across various time constraints, and (2) how well RubberBand performs across various deep learning models.

6.3.1 Adaptability across time constraints

In this experiment, we evaluate the cost-efficiency of RubberBand across various time constraints. We train a ResNet101 model on CIFAR10 using a batch size of 1024, and use SHA($n = 32, r = 1, R = 50, \eta = 3$) to generate the specification. We set our instance initialization and node scale-up latency to 15 seconds (using a warm pool of instances). Across the board, RubberBand is able to reduce costs across a variety of time constraints.

We compared the job execution costs between RubberBand, the static cluster policy, and a naive elastic baseline across various time constraints. The naive elastic baseline used finds the cost-optimal allocation plan within the constrained space of *fixed allocations per-trial*. That is, although the cluster size is elastically adjusted, the number of resources allocated to each trial remains constant across stages. This is similar to a strategy described in prior work [9].

	Max Time (min)	JCT (sim)	Cost (sim)	JCT (real)	Cost (real)	Acc (%)
Static	20	19:24 ± 00:00	$34.00 ± $0.00	19:15 ± 00:26	$33.21 ± $0.84	91.9 ± 0.7
Naive elastic	20	14:39 ± 00.00	$27.50 ± $0.00	*	*	*
RubberBand	20	18:56 ± 00:02	$15.68 ± $0.02	18:47 ± 00:22	**$15.67 ± $0.41**	89.9 ± 1.3
Static	30	29:09 ± 00:02	$17.73 ± $0.00	29:37 ± 00:20	$17.99 ± $0.52	83.2 ± 6.4
Naive elastic	30	26:25 ± 00:00	$19.83 ± $0.00	*	*	*
RubberBand	30	29:14 ± 00:00	$14.70 ± $0.00	27:32 ± 00:03	**$14.06 ± $0.02**	87.6 ± 1.1
Static	40	36:53 ± 00:02	$14.99 ± $0.02	35:45 ± 00:49	$14.59 ± $0.33	89.2 ± 0.8
Naive elastic	40	26:25 ± 00:00	$19.83 ± $0.00	*	*	*
RubberBand	40	37:08 ± 00:02	$14.68 ± $0.02	36:05 ± 00:15	**$14.16 ± $0.11**	88.4 ± 0.6

Table 2. Cost to complete workload across various time constraints. We measure the cost of an end-to-end benchmark tuning ResNet101 on an elastic cluster of on-demand p3.8xlarges with a variety of time constraints. Each experiment is run across 3 seeds. RubberBand performance is able to match simulation and provide significant cost improvements. Naive elastic experiments were not run due to the prohibitively large number of resources required (e.g. 512 GPUs in the first stage of the 20-minute experiment).

Epoch range	trials	GPUs/trial	Cluster Size
0-1	32	1	8
1-4	10	2	5
4-13	3	4	4
13-50	1	8	2

Table 3. Example cluster schedule for elastic training. RubberBand will leverage a given allocation plan to create a cluster resource schedule. Through this schedule, RubberBand is able to dynamically adjust the number of resources allocated per trial, leveraging information about the model's scaling profile and the underlying cloud environment.

Results are shown in Table 2. We also include the allocation plan generated for the 20 minute time-constraint in Table 3 (the optimal static cluster size was 6 instances—a total of 24 GPUs). We make four observations regarding these results.

First, the error rate in both JCT and cost is low, showing high fidelity between simulation and reality. Second, across all deadlines, all baselines demonstrated a statistically insignificant difference in accuracy.[4] Any discrepancies are expected to be a result of improper seeding.

Third, the naive elastic policy is outperformed by both RubberBand and the static policy (at 30 minutes) in simulation. This shows that elasticity applied naively, as proposed in previous work [9], is ineffective at reducing cost.

Last, we see that tighter constraints lead to a larger cost deviation between RubberBand and baselines, as was demonstrated in simulated experiments. Specifically, RubberBand demonstrates a cost reduction of 53% on the 20 minute deadline over the fixed cluster allocation baseline, while demonstrating a negligible difference at the 40 minute time constraint. This is expected, as the necessity of parallelization decreases with a more lax constraint.

[4]While we do not attain state-of-the-art accuracy (94%), this is orthogonal to the problem RubberBand solves, and can be remedied by simply applying standard (compatible) techniques such as using an lr-schedule.

Model	Time	Fixed	RubberBand
Resnet101	0:20:00	$33.21 ± $0.84	**$15.67 ± $0.41**
Resnet152	1:00:00	$35.53± $0.30	**$26.54 ± $0.33**
BERT	0:20:00	$36.43± $0.16	**$29.17± $0.23**

Table 4. Cost to complete workload across various models. We compared the cost of executing a fixed cluster plan to RubberBand across a variety of popular deep learning models/datasets: Resnet101 on CIFAR10, Resnet152 on CIFAR100, and BERT on RTE. Results are averaged across 3 runs. In each case, we are able to leverage RubberBand to reduce costs on the given baseline.

6.3.2 Adaptability across DL models

We evaluate the behavior of RubberBand on a variety of deep learning models and datasets. Specifically, we evaluate:

1. ResNet101 on CIFAR10
2. ResNet152 on CIFAR100
3. BERT on RTE

For each of these models, we measured their scaling profile on p3.8xlarge instances and compared the performance of RubberBand against a fixed cluster baseline. We expect that RubberBand will be able to reduce execution costs by better utilizing resources. As seen in Table 4, RubberBand is able to provide cost savings across all models by leveraging information about the cloud platform and DL model.

7 Discussion and Related Work

RubberBand addresses the challenges of minimizing the cost of a hyperparameter tuning job by leveraging elastic cloud resources. In this section, we review literature related to this problem and some potential avenues for future work.

Hyperparameter tuning algorithms. There has been related work in both theory and practice for resource-constrained hyperparameter tuning. On the theoretical front, [7, 8, 37] address the problem of budgeted hyperparameter tuning, but do so for an abstract notion of a "work budget",

which requires translation to space-time dimensions. RubberBand supports such methods.

ASHA [9] and HyperSched [11] similarly attempt to practically adapt Successive Halving to the parallel setting, but are primarily designed around the limitations of using a fixed-size cluster. Namely, ASHA always samples new configurations when resources are freed during trial termination, whereas HyperSched employs a heuristic to determine whether to sample a new configuration or reallocate all freed resources to remaining trials. Sampling new configurations has been shown to be an ineffective use of resources under a time-constraint [11]. On the other hand, reallocating *all* resources results in resource under-utilization.

Systems for hyperparameter tuning. Several systems aim to scale hyperparameter tuning workloads [9, 30, 38, 39]. Vizier [30] is a scalable black-box optimization service used to generate hyperparameter configurations, but is completely agnostic to the execution layer. RubberBand has an execution layer and is therefore synergistic with Vizier.

Determined AI's production system [9, 39] supports the elastic execution of ASHA. Similarly to RubberBand, the allocation to a job is reduced over time, but the mechanism for doing so is naive—the *maximum* size of the allocation is ensured to be a *fixed* multiple of the number of remaining trials from the originally spawned set. This can easily result in sub-optimal utilization, as demonstrated in the evaluation.

Other systems similarly study the problem of scheduling hyperparameter tuning jobs in the on-premise, multi-tenant setting. Gandiva [23, 40] is a GPU cluster scheduling framework that introduces new scheduling primitives for DL, and aims to maximize global cluster efficiency. Philly [24] is another GPU cluster scheduler that similarly optimizes for global cluster utilization. Tiresias [22] and Themis [25] aim to address multi-tenant issue of fairness. Unlike RubberBand, these systems are primarily designed for the on-premise, multi-tenant GPU cluster setting.

Elasticity in DL. Recent work [41–44] has studied how to leverage compute elasticity in related workloads. NumPyWren [41] identifies and exploits dynamic parallelism in linear algebra algorithms, including matrix multiplication (key to DL) to increase compute efficiency. One layer higher, [44] and EDL [42] introduce elasticity in DL training workloads, the latter in the context of multi-tenant clusters. TorchElastic [43] similarly provides a interface for defining and executing elastic jobs in a fault-tolerant manner. Cirrus [45] leverages elasticity to scale end-to-end ML workflows, but primarily addresses interactive workloads.

These systems are related but leverage elasticity at a lower level (i.e. matrix multiplication and single-model training), rather than at the *hyperparameter tuning* layer.

DL performance modeling. Prior work has studied how to model neural network training performance, based on neural architecture and hardware utilized [15, 20]. However, such approaches are limited to specific architecture types. In order to handle arbitrary DL model architectures, we instead measure training performance empirically, since collection of instrumentation data can be done in low time and cost, relative to actual JCT and cost. However, prior performance modeling work becomes more pertinent when it is infeasible to collect sufficient data. This includes, for example, extensions to use-cases where performance is a function of hyperparameters, or when the training procedure leverages a batch size schedule.

Cloud autoscaling. There have been numerous efforts towards minimizing the cost of executing generic workloads by dynamically autoscaling the underlying cloud cluster. Several autoscaling systems observe and predict the resource utilization of jobs [46–48], but do not leverage workload-specific properties. Recent work [48] has explored autoscaling jobs in datacenters to meet time constraints, but aims to maximize *overall* datacenter utilization by reducing interference between co-located jobs. Stratus [49] autoscales cluster size and bin-packs tasks based on predicted JCT to minimize the cost of executing tasks; however, resource requirement determination is left to the user.

Other prior work [50] autoscales cluster size to optimize the cost of executing several tasks with different deadlines. Hotspot [29] exploits pricing differences across regions to migrate jobs when beneficial, but such migrations are ineffective for expensive GPU instance types. Resource type selection in the cloud has also attracted recent attention. Ernest [51] profiles a sub-sample of the workload to build a performance model of each job, while CherryPick [52] uses bayesian optimization to learn from historical performance.

While some of these systems are able to leverage elastic compute pools to execute jobs, they are unable to accommodate all of the requirements of distributed DL workloads (such as supporting distributed job execution and co-location), and do not exploit workload-specific characteristics to optimize cost.

8 Conclusion

RubberBand utilizes performance instrumentation and cloud pricing data to model job completion time and cost prior to runtime, in order to generate a feasible, cost-efficient, elastic resource allocation plan. RubberBand is then able to efficiently execute the plan it generates, dynamically scaling and managing the cluster as required. We demonstrate a reduction in cost of up to 2x on real-world hyperparameter optimization jobs against fixed-cluster baselines, while preserving JCT and model accuracy.

Acknowledgements

In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[3] Or Sharir, Barak Peleg, and Yoav Shoham. The cost of training nlp models: A concise overview, 2020.

[4] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019.

[5] Philipp Probst, Bernd Bischl, and Anne-Laure Boulesteix. Tunability: Importance of hyperparameters of machine learning algorithms, 2018.

[6] Y. Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, M. Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692, 2019.

[7] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[8] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. pages 1–52, 2018.

[9] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. In *Proceedings of Workshop on ML Systems in The Thirty-second Annual Conference on Neural Information Processing Systems (NIPS)*, 2018.

[10] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: robust and efficient hyperparameter optimization at scale. *CoRR*, abs/1807.01774, 2018.

[11] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 61–73, New York, NY, USA, 2019. ACM.

[12] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta. *A Generalized Framework for Population Based Training*, page 1791–1799. Association for Computing Machinery, New York, NY, USA, 2019.

[13] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, July 2019.

[14] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[16] Amazon ec2 pricing - amazon web services. https://aws.amazon.com/ec2/pricing/. (Accessed on 03/11/2021).

[17] All pricing | compute engine documentation | google cloud. https://cloud.google.com/compute/all-pricing. (Accessed on 03/11/2021).

[18] Pricing - windows virtual machine | microsoft azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/. (Accessed on 03/11/2021).

[19] Aws lambda — pricing. https://aws.amazon.com/lambda/pricing/. (Accessed on 03/11/2021).

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

[21] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys '18: Thirteenth EuroSys Conference*, page 14, 2018.

[22] Yun Shen, Enrico Mariconti, Pierre-Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, page 14, 2018.

[23] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610, 2018.

[24] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[25] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. volume abs/1907.01484, 2019.

[26] Amazon ec2 instance types - amazon web services. https://aws.amazon.com/ec2/instance-types/. (Accessed on 03/11/2021).

[27] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2017.

[28] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, , and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *VLDB DISPA Workshop 2020*.

[29] Supreeth Shastri and David Irwin. *HotSpot: Automated Server Hopping in Cloud Spot Markets*, page 493–505. Association for Computing Machinery, New York, NY, USA, 2017.

[30] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[31] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI 18), pages 561–577, 2018.

[33] Distributeddataparallel — pytorch 1.6.0 documentation. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html. (Accessed on 10/09/2020).

[34] Boto3 documentation — boto3 docs 1.15.15 documentation. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html. (Accessed on 10/09/2020).

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[36] Amazon s3 simple storage service pricing - amazon web services. https://aws.amazon.com/s3/pricing/. (Accessed on 03/11/2021).

[37] Ruben Martinez-Cantin. Bayesopt: A bayesian optimization library for nonlinear optimization, experimental design and bandits. In *Journal of Machine Learning Research*, pages 3735–3739, 2014.

[38] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 2623–2631, New York, NY, USA, 2019. Association for Computing Machinery.

[39] Distributed deep learning and hyperparameter tuning platform | determined ai. https://determined.ai/. (Accessed on 10/08/2020).

[40] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, and James Cheng. Elastic deep learning in multi-tenant gpu cluster. *arXiv preprint arXiv:1909.11985*, 2019.

[43] Torchelastic — pytorch/elastic master documentation. https://pytorch.org/elastic/0.2.1/index.html. (Accessed on 10/08/2020).

[44] Andrew Or, Haoyu Zhang, and Michael J Freedman. Resource elasticity in distributed deep learning.

[45] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: a serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.

[46] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[48] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[49] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 121–134, New York, NY, USA, 2018. Association for Computing Machinery.

[50] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[51] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 363–378, USA, 2016. USENIX Association.

[52] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 469–482, USA, 2017. USENIX Association.