Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning

Aurick Qiao[1,2], Willie Neiswanger[1,2], Qirong Ho[2], Hao Zhang[1,2], Gregory R. Ganger[1], and Eric P. Xing[1,2]

[1]Carnegie Mellon University
[2]Petuum, Inc.

Abstract

Pollux improves scheduling performance in deep learning (DL) clusters by adaptively co-optimizing inter-dependent factors both at the per-job level and at the cluster-wide level. Most existing schedulers will assign each job a number of resources requested by the user, which can allow jobs to use those resources inefficiently. Some recent schedulers choose job resources for users, but do so without awareness of how DL training can be re-optimized to better utilize those resources.

Pollux simultaneously considers both aspects. By observing each job during training, Pollux models how their *goodput* (system throughput combined with statistical efficiency) would change by adding or removing resources. Leveraging these models, Pollux dynamically (re-)assigns resources to maximize cluster-wide goodput, while continually optimizing each DL job to better utilize those resources.

In experiments with real DL training jobs and with trace-driven simulations, Pollux reduces average job completion time by $25\% - 50\%$ relative to state-of-the-art DL schedulers, even when all jobs are submitted with ideal resource and training configurations. Based on the observation that the statistical efficiency of DL training can change over time, we also show that Pollux can reduce the cost of training large models in cloud environments by 25%.

1 Introduction

Deep learning (DL) training has rapidly become a dominant workload in many shared resource environments such as datacenters and the cloud. DL jobs are resource-intensive and long-running, demanding distributed execution using expensive hardware devices (eg. GPUs or TPUs) in order to complete within reasonable amounts of time. To meet this resource demand, dedicated clusters are often provisioned for deep learning alone [24], with a scheduler that mediates resource sharing between many competing DL jobs.

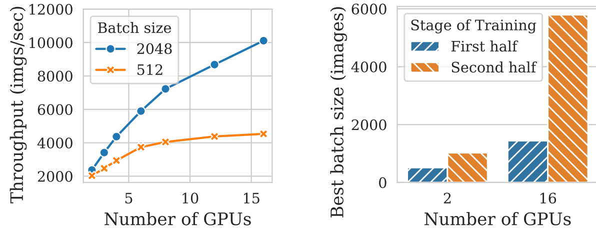
However, existing schedulers require users submitting jobs to also specify training parameters that, if set incorrectly, can greatly degrade job performance and resource efficiency. Of these training parameters, the *batch size* and *learning rate* of a DL job are strongly dependent on its allocation of resources, making them particularly difficult to decide in advance in shared-resource environments. Furthermore, an allocation of resources that can be efficiently utilized by a DL job not only depends on the structure of the model being trained, but also on the batch size and learning rate. This co-dependence between the resources, batch size, and learning rate creates a complex web of considerations a user must make in order to configure their job for efficient execution and resource utilization.

Fundamentally, an efficiently configured DL job strikes a balance between two often opposing desires: (1) *system throughput*, the number of training examples processed per wall-clock time, and (2) *statistical efficiency*, the amount of progress made per training example processed.

System throughput can be increased by increasing the batch size, as illustrated in Fig. 1a. A larger batch size enables higher utilization of more resources (eg. larger number of GPUs). However, when the batch size is increased, the learning rate must be re-tuned. Otherwise, statistical efficiency will decrease so that the total training time will not be any shorter, wasting the additionally allocated GPUs.

Even with an optimally-tuned learning rate, increasing the batch size results in faster-decreasing statistical efficiency [31, 43]. For every distinct allocation of GPUs, there is potentially a different batch size that best balances increasing system throughput with decreasing statistical efficiency, as illustrated in Fig. 1b. Furthermore, how quickly the statistical efficiency decreases with respect to the batch size depends on the current training progress. A job in a later stage of training can potentially tolerate 10x or larger batch sizes without degrading statistical efficiency, than earlier during training [31].

Thus, the best choice of batch size and learning rate depends on the resource allocation, which in turn depends on competition from other jobs sharing the cluster. In turn, the best choice of resource allocation depends on the chosen batch size. The batch size, learning rate, and therefore the best resource allocation, all depend on the current training progress of the



(a) Job scalability (and therefore resource utilization) depends on the batch size.

(b) The most efficient batch size can depend on the allocated resources and the stage of training.

Figure 1: Trade-offs between the batch size, resource scalability, and stage of training (ResNet18 on CIFAR-10). The learning rate is separately tuned for each batch size.

job. Therefore, we argue that the choice of resource allocations, batch sizes, and learning rates are best made collectively and dynamically by a knowledgeable cluster scheduler.

This paper presents *Pollux*, a hybrid resource scheduler that *co-adaptively* allocates resources while tuning the batch size and learning rate for every DL job in a shared cluster.

★ We provide a formulation of *goodput* for DL jobs, which is a performance metric that takes into account both system throughput and statistical efficiency. A model of goodput can be learned by observing the throughput and statistical behavior during training, and used for predicting the performance of DL jobs given different resource allocations and batch sizes.

★ We design and implement a scheduling architecture that locally tunes the batch size and learning rate for each DL job, and globally optimizes cluster-wide resource allocations using a genetic algorithm. Both components actively cooperate with each other, and operate based on a common goal of goodput maximization.

★ Pollux not only adapts to the changes in statistical efficiency over time, but can leverage it to reduce the cost of training large models. In cloud environments, Pollux can provision the right amount of resources *at the right time*, based on job training progress, to maximize statistical efficiency across the entire lifetime of a large DL job.

When compared with state-of-the-art DL schedulers using realistic DL job traces from Microsoft, we show that Pollux reduces the average job completion time by up to 70%. Even when all jobs are manually tuned beforehand, Pollux reduces the average job completion time by $25-50\%$. In cloud environments, Pollux reduces the cost of training ImageNet by 25%.

2 Background: Distributed DL Training

Training a deep learning model typically involves minimizing a *loss function* of the form

$$\mathcal{L}(w) = \frac{1}{|X|}\sum_{x_i \in X} \ell(w, x_i), \tag{1}$$

where $w \in \mathbb{R}^d$ are the model parameters to be optimized, X is the training dataset, x_i are the individual samples in the training data, and ℓ is the loss evaluated at a single sample.

The loss function can be minimized using *stochastic gradient descent* (SGD), which repeatedly applies the following update until the loss converges to a stable value:

$$w^{(t+1)} = w^{(t)} - \eta \hat{g}^{(t)}. \tag{2}$$

η is known as the learning rate, which is a scalar that controls the magnitude of each update, and $\hat{g}^{(t)}$ is a stochastic gradient estimate of the loss function $\mathcal{L}$, evaluated using a random *mini-batch* $M^{(t)} \subset X$ of the training data, as follows:

$$\hat{g}^{(t)} = \frac{1}{|M^{(t)}|}\sum_{x_i \in M^{(t)}} \nabla \ell(w^{(t)}, x_i). \tag{3}$$

The learning rate η and batch size $m = |M^{(t)}|$ are training parameters which are typically chosen by the user.

2.1 System Throughput

The *system throughput* of DL training can be defined as the number of training samples processed per unit of wall-clock time. When a DL job is distributed across several nodes, its system throughput is determined by several factors, including (1) the allocation and placement of resources assigned to the job, (2) the method of distributed execution and synchronization, and (3) the batch size used by the SGD algorithm.

Data-parallel execution. *Data-parallelism* is a popular method of distributed execution for DL training. The model parameters $w^{(t)}$ are replicated across a set of distributed GPUs $1, \ldots, K$, and each mini-batch $M^{(t)}$ is divided into equal-sized partitions per node, $M_1^{(t)}, \ldots, M_K^{(t)}$. Each GPU k computes a local gradient estimate $\hat{g}_k^{(t)}$ using its own partition, as follows:

$$\hat{g}_k^{(t)} = \frac{1}{|M_k^{(t)}|}\sum_{x_i \in M_k^{(t)}} \nabla \ell(w^{(t)}, x_i). \tag{4}$$

These local gradient estimates are then averaged across all replicas to obtain the desired $\hat{g}^{(t)}$, as defined by Eqn. 3. Finally, each node applies the same update using $\hat{g}^{(t)}$ to obtain the new model parameters $w^{(t+1)}$, as defined by Eqn. 2.

The run-time of each training iteration is determined by two main components. *First*, the time spent computing the local gradient estimates $\hat{g}_k^{(t)}$, which we denote by T_{grad}. For neural networks, this computation is done using backpropagation [41]. *Second*, the time spent averaging the local gradients and synchronizing the model parameters across all job replicas, which we denote by T_{sync}. This synchronization is typically done using a collective all-reduce operation [36, 42], or a set of parameter servers [7, 8, 21]. T_{sync} can be influenced by the placement of replicas, and is typically smaller when the replicas are co-located within the same physical node or rack, rather than spread across different nodes or racks.

Limitations due to the batch size. The batch size used during training determines the upper limit on the system throughput. When the number of replicas is increased, each replica will process a smaller partition of the overall mini-batch, resulting in a proportionally smaller T_{grad}. On the other hand, T_{sync} is typically dependent on the size of the gradients and model parameters, rather than the batch size or number of replicas. Due to Amdahl's Law, no matter how many replicas are used, the run-time of each training iteration is lower bounded by T_{sync}.

To overcome this scalability limitation, it is desirable to increase the batch size, which allows a larger proportion of time to be spent computing the local gradient estimates as opposed to synchronizing gradients and parameters over the network. Thus, using a larger batch size enables higher system throughput when scaling to more data-parallel replicas.

2.2 Statistical Efficiency

The *statistical efficiency* of DL training can be defined as the amount of training progress made per unit of data processed. Parameters such as batch size and learning rate influence this quantity. For example, when the batch size is increased, the efficiency will decrease (by an amount that depends on how the learning rate is comparatively scaled). Here, we give background on previous work that expresses the statistical efficiency in terms of a quantity called the *gradient noise scale*. We then describe adaptive scaling rules that have been developed to set the learning rate with respect to the batch size, to achieve high statistical efficiency.

In Pollux, the gradient noise scale will be used to define a measure of statistical efficiency for a given choice of batch size, which will allow us to evaluate and optimize the goodput. Relatedly, adaptive scaling rules will be used to dynamically choose an appropriate learning rate with respect to the current batch size during this optimization.

Gradient Noise Scale. Previous work [25, 31], has aimed to characterize the statistical efficiency of DL training in terms of the gradient noise scale φ_t, which can be intuitively viewed as a measure of the signal-to-noise ratio of gradient across training examples at iteration t [31]. A larger φ_t means that training parameters such as the batch size and learning rate can be increased to higher values with relatively less reduction of the statistical efficiency. The gradient noise scale can vary greatly between different DL models [13]. It is also non-constant and tends to gradually increase during training, by up to 10$\times$ or more [31]. Thus, it is possible to attain significantly better statistical efficiency for a large batch size later on in training. A knowledgeable scheduler like Pollux can adaptively scale training parameters for the right DL jobs at the right times, to better accommodate for their model-dependent and time-varying levels of efficiency.

Learning Rate Scaling and Adascale SGD. If the batch size is increased, the learning rate must also be scaled up to maintain a high statistical efficiency. One way to adjust the learning rate η with respect to the batch size m is by using simple scaling rules. For example, the linear scaling rule [30] prescribes that η be scaled proportionally with m, while the square-root scaling rule prescribes that η be scaled proportionally with $\sqrt{m}$. Correctly scaling the learning rate can improve statistical efficiency when training with large batch sizes, and result in orders-of-magnitude improvements to DL scalability and job completion time [15, 37, 47].

However, simple learning rate scaling rules cannot be used for predicting the statistical efficiency of a particular batch size and learning rate ahead of time. Thus, they are unable to provide the knowledge required by a cluster scheduler to jointly optimize resource allocations with batch sizes. Instead of scaling the learning rate by a constant factor for each batch size, *AdaScale* [25] scales the learning rate adaptively based on φ_t. Suppose a DL training job is run with batch size m_0. If the same job is run using a larger batch size $m > m_0$, then at iteration t, AdaScale scales the learning rate by the factor

$$r_t = (\varphi_t/m_0+1)/(\varphi_t/m+1), \tag{5}$$

where we describe how φ_t is computed in Sec. 3.1. AdaScale has been shown to outperform the linear scaling rule for a wide variety of DL models and batch sizes.

For resource scheduling, the most important characteristic of AdaScale is its *predictability*. One iteration of AdaScale training with batch size m is approximately equivalent to r_t iterations of training with the original m_0. This property can be leveraged to measure statistical efficiency during training, and to predict its value before scaling to different batch sizes, as described in Sec. 3.1.

2.3 Existing DL Schedulers

We broadly group existing DL schedulers into two categories. First, *non-resource-adaptive* schedulers are agnostic to the throughput scalability of DL jobs with respect the the amount of allocated resources. For example, Tiresias [16] does not require any knowledge about the throughput of each job if allocated different numbers of GPUs. Instead, users specify the number of GPUs at the time of job submission, which will be fixed for the lifetime of the job. Tiresias may preempt jobs to prevent head-of-line blocking, and co-locate job replicas for more efficient synchronization. Gandiva [46] is another DL scheduler which requires user-specified number of GPUs, but is able to optimize resource utilization through fine-grained time sharing and job packing. Although Gandiva is able to dynamically grow or shrink the number of GPUs used by a job, it only does so opportunistically, and not based on knowledge of job scalability.

Second, *only-resource-adaptive* schedulers automatically decide the amount of resources allocated to each job based on how well they can be utilized to speed up the job. For example, Optimus [38] learns a predictive model for the system throughput of each job given various amounts of resources, and optimizes cluster-wide resource allocations to minimize the

average job completion time. SLAQ [49], which was not evaluated on deep learning, uses a similar technique to minimize the average loss values for training general ML models.

All existing schedulers are agnostic to the statistical efficiency of DL training. The batch size and learning rate are left for the users to tune at the application-level, which as previously discussed, can be difficult to do efficiently.

3 The Goodput of DL Training

We present how *goodput*[1] can be defined, measured, and predicted for DL jobs, taking into account both system throughput and statistical efficiency. Goodput predictions are leveraged by Pollux to jointly optimize cluster-wide resource allocations and batch sizes.

Definition 3.1. (Goodput) The *goodput* of a DL job at iteration t is the product between its system throughput and its statistical efficiency at iteration t.

$$\texttt{GOODPUT}_t(a,m) = \texttt{THROUGHPUT}(a,m) \times \texttt{EFFICIENCY}_t(m) \quad (6)$$

$a \in \mathbb{R}^N$ is an *allocation vector*, where a_n is the number of GPUs allocated from node n, and m is the batch size.

An initial batch size m_0 and learning rate η_0 are selected by the user when submitting their job. Pollux will start each job using a single GPU, $m = m_0$, and $\eta = \eta_0$. As the job runs, Pollux profiles its execution to learn and refine predictive models for both THROUGHPUT (Sec. 3.2) and EFFICIENCY (Sec. 3.1). Using these predictive models, Pollux periodically re-tunes a and m for each job, according to cluster-wide resource availability and performance (Sec. 4.2). The learning rate η is re-tuned using AdaScale (Sec. 2.2).

Goodput and statistical efficiency are measured *relative to* the initial batch size m_0 and learning rate η_0, and Pollux only considers batch sizes which are at least the initial batch size, ie. $m \geq m_0$. In this scenario, statistical efficiency is always between 0 and 1, and can be interpreted as a percentage relative to the initial batch size m_0. Therefore, goodput is always less than or equal to the throughput, being equal only if perfect statistical efficiency is achieved.

3.1 Modeling Statistical Efficiency

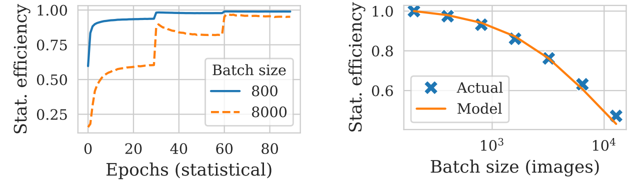
The statistical efficiency of a DL job using batch size $m \geq m_0$ is the amount of progress made per training example using m, relative to using m_0. This quantity can be framed in terms of the gradient noise scale φ_t. Specifically, [31] formulates the amount of training progress that can be made in a single update using a batch of size m relative to the amount of progress made with a batch of size m_0. Similarly, Adascale's [25] r_t (Sec. 2.2) is also the relative training progress made using batch size m versus m_0. In Appendix A, we show an equivalence between the formulations of efficiency given by both papers, and we can use these to write a concrete measure of statistical efficiency as

$$\texttt{EFFICIENCY}_t(m) = r_t m_0 / m = (\varphi_t + m_0)/(\varphi_t + m). \quad (7)$$

Here, we can write the gradient noise scale as $\varphi_t = m_0 \sigma_t^2 / \mu_t^2$, where $\sigma_t^2 = \mathrm{Var}[\hat{g}^{(t)}]$ is the variance and $\mu_t^2 = |\mathbb{E}[\hat{g}^{(t)}]|^2$ the squared norm of the gradient at iteration t using batch size m_0.

Intuitively, Eqn. 7 measures the contribution from each training example to the overall progress. If $\texttt{EFFICIENCY}_t(m) = E$, then (1) $0 < E \leq 1$, and (2) training using batch size m will need to process $1/E$ times as many training examples to make the same progress as using batch size m_0.

During training, Pollux estimates the values of σ_t and μ_t to compute φ_t, then uses Eqn 7 to predict the statistical efficiency at different batch sizes. The true values of σ_t and μ_t vary according to the training progress at iteration t, thus $\texttt{EFFICIENCY}_t(m)$ reflects the lifetime-dependent trends exhibited by the true statistical efficiency.



(a) Efficiency vs training progress using different batch sizes.

(b) Actual efficiency vs predicted efficiency using Eqn. 7.

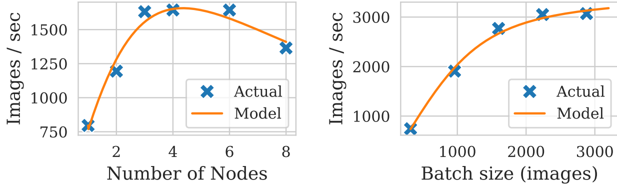
Figure 2: Statistical efficiency for training ResNet-50 on ImageNet. In Fig. 2a, each "statistical epoch" measures the same training progress across different batch sizes. In Fig. 2b, predictions are based on the value of φ_t measured using a batch size of 4000 images at epoch 15.

Fig. 2a shows an example of statistical efficiency measured for ImageNet training. First, a larger batch size results in lower statistical efficiency, but the gap narrows during the later phases of training. Second, how the statistical efficiency changes over time depends on details of the training procedure. In this case, the statistical efficiency increases dramatically when the learning rate is decayed by a factor of 10 at epochs 30 and 60, following the standard configuration for training ImageNet.

Fig. 2b shows the measured statistical efficiency at different batch sizes, compared with the statistical efficiency predicted by Eqn. 7 using φ_t measured at a fixed batch size. We find close agreement between the measured and predicted values across a range of different batch sizes, which indicates that $\texttt{EFFICIENCY}_t(m)$ can be used by Pollux to predict the statistical efficiency at a different batch size without needing to train using that batch size ahead of time.

Estimating σ_t and μ_t. The standard way of estimating σ_t and μ_t involves calculating the sample variance of the local gradient estimates $\hat{g}_k^{(t)}$ at each iteration [25, 31]. This

[1] Our notion of goodput for DL is analogous to the traditional definition of goodput in computer networking, ie. the fraction of *useful* throughput.



(a) Throughput vs. nodes. (b) Throughput vs batch size.

Figure 3: Our throughput model (Eqn. 8) fit to measured throughput values for ImageNet training.

can be done efficiently when there are multiple data-parallel processes, by using the different values of $\hat{g}_k^{(t)}$ already available on each process. However, this method doesn't work when there is only a single process. In this particular situation, Pollux switches to a differenced variance estimator [45] which uses consecutive gradient estimates $\hat{g}^{(t-1)}$ and $\hat{g}^{(t)}$.

3.2 Modeling System Throughput

To model and predict the system throughput for data-parallel DL, we aim to predict the time spent per training iteration, T_{iter}, given an allocation vector a and batch size m, and then calculate the throughput as

$$\texttt{THROUGHPUT}(a,m) = m/T_{iter}(a,m). \tag{8}$$

We start by separately modeling T_{grad}, the time in each iteration spent computing local gradient estimates, and T_{sync}, the time in each iteration spent averaging gradient estimates and synchronizing model parameters across all GPUs.

Modeling T_{grad}. The local gradient estimates are computed using back-propagation, whose run-time scales linearly with the local batch size in each process. Thus, we model T_{grad} as

$$T_{grad}(a,m) = \alpha_{grad} + \beta_{grad} \cdot m/K, \tag{9}$$

where m is the overall batch size, $K = \sum_n a_n$ is the number of allocated GPUs, and α_{grad} and β_{grad} are learnable parameters.

Modeling T_{sync}. When $K = 1$, then no synchronization is needed and $T_{sync} = 0$. Otherwise, we model T_{sync} as a linear function of K. We choose choose a linear model because in strict data-parallelism, the amount of data sent and received from each replica is typically constant with respect to the size of the gradients and/or parameters. We include a linear factor to account for performance retrogressions associated with larger K, such as increasing likelihood of stragglers or network delays.

Since synchronization requires network communication between replicas, co-location of those replicas on the same node can significantly improve T_{sync}. Thus, we use different parameters depending on the placement.

$$T_{sync}(a,m) = \begin{cases} 0 & \text{if } K=1 \\ \alpha_{sync}^{local} + \beta_{sync}^{local} \cdot (K-2) & \text{if } N=1, K \geq 2 \\ \alpha_{sync}^{node} + \beta_{sync}^{node} \cdot (K-2) & \text{otherwise,} \end{cases} \tag{10}$$

where N is the number of physical nodes occupied by at least one replica. α_{sync}^{local} and β_{sync}^{local} are the constant and retrogression parameters for when all processes are co-located onto the same node. α_{sync}^{node} and β_{sync}^{node} are the analogous parameters for when at least two process are located on different nodes. Note that our model for T_{sync} can be extended to account for rack-level locality by adding a third pair of parameters.

Combining T_{grad} and T_{sync}. Modern DL frameworks can partially overlap T_{grad} and T_{sync} by overlapping gradient computation with network communication [48]. The degree of this overlap depends on structures in the specific DL model being trained, like the ordering and sizes of its layers.

Assuming no overlap, then $T_{iter} = T_{grad} + T_{sync}$. Assuming perfect overlap, then $T_{iter} = \max(T_{grad}, T_{sync})$. A realistic value of T_{iter} is somewhere in between these two extremes. To capture the overlap between T_{grad} and T_{sync}, we model T_{iter} as

$$T_{iter}(a,m) = \left(T_{grad}(a,m)^{\gamma} + T_{sync}(a)^{\gamma}\right)^{1/\gamma}, \tag{11}$$

where $\gamma \geq 1$ is a learnable parameter. Eqn. 11 has the property that $T_{iter} = T_{grad} + T_{sync}$ when $\gamma = 1$, and smoothly transitions towards $T_{iter} = \max(T_{grad}, T_{sync})$ as $\gamma \to \infty$.

Fig. 3 shows an example of our `THROUGHPUT` function fit to measured throughput values for a range of resource allocations and batch sizes. We find that our model can represent the observed data closely, while varying both the amount of resources as well as the batch size. In Sec. 4.1, we describe how we fit our throughput model to values measured during training, and then used to predict training throughput for different resource allocations and batch sizes.

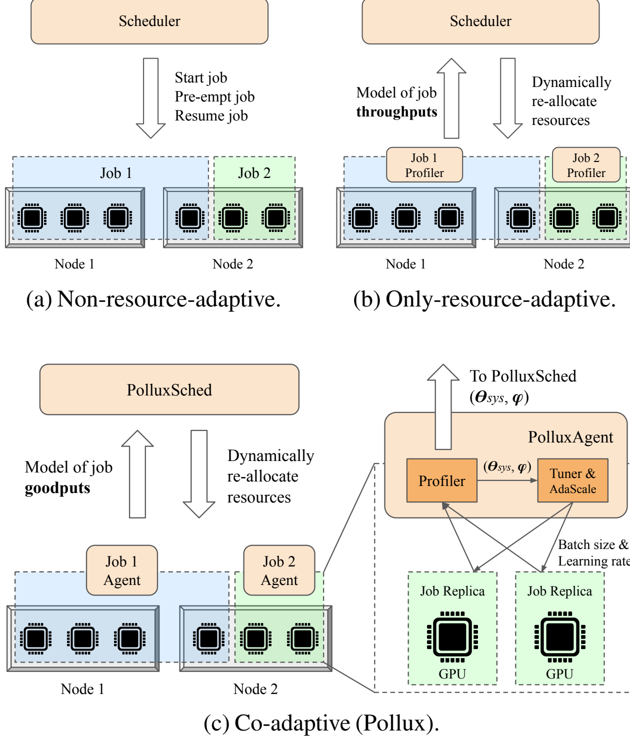
4 Pollux Design and Architecture

Compared with existing DL schedulers, Pollux performs adaptation at two distinct granularities. First, at a job-level granularity, Pollux dynamically tunes the batch size and learning rate for best utilization of the allocated resources. Second, at the cluster-wide granularity, Pollux dynamically re-allocates resources, driven by the goodput of all jobs sharing the cluster. To achieve this co-adaptivity in a scalable way, Pollux's design consists of two primary components.

First, a *PolluxAgent* runs together with each job. It measures the gradient noise scale and system throughput for that job, and tunes its batch size and learning rate for efficient utilization of its current allocated resources. PolluxAgent periodically reports the goodput function of its job to the PolluxSched.

Second, the *PolluxSched* periodically optimizes the resource allocations for all jobs in the cluster, taking into account the current statistical efficiency for each job. It uses the goodput function to predict a job's training performance when allocated different resources.

PolluxAgent and PolluxSched *co-adapt* to each other. While PolluxAgent adapts each training job to make efficient use of its allocated resources, PolluxSched dynamically



(a) Non-resource-adaptive. (b) Only-resource-adaptive.

(c) Co-adaptive (Pollux).

Figure 4: Architecture of Pollux (Fig. 4c), compared with existing schedulers which are either non-resource-adaptive (Fig. 4a) or only-resource-adaptive (Fig. 4b).

re-allocates each job's resources, taking into account the PolluxAgent's ability to tune its job.

Fig. 4 illustrates Pollux's co-adaptive architecture (Fig. 4c), compared with existing schedulers which are either non-adaptive (Fig. 4a), or resource-adaptive without being involved with statistical efficiency (Fig. 4b).

4.1 PolluxAgent: Job-level Optimization

An instance of PolluxAgent is started with each training job. During training, it continually measures the job's gradient noise scale and system throughput, and reports them to PolluxSched at a fixed interval. It also uses this information to determine the most efficient batch size for its job given its current resource allocations, and adapts its job's learning rate to this batch size using AdaScale.

Online model fitting. In Sec. 3.2, we defined the system throughput parameters of a training job as the 7-tuple

$$\theta_{sys} = \left(\alpha_{grad}, \beta_{grad}, \alpha_{sync}^{local}, \beta_{sync}^{local}, \alpha_{sync}^{node}, \beta_{sync}^{node}, \gamma\right), \quad (12)$$

which are required to construct the THROUGHPUT function. Together with the gradient noise scale φ_t and initial batch size m_0, the triple $(\theta_{sys}, \varphi_t, m_0)$ specifies the GOODPUT function. While m_0 is a constant configuration provided by the user, and φ_t can be computed according to Sec. 3.1, θ_{sys} is estimated by fitting the THROUGHPUT function to observed throughput values collected about the job during training.

PolluxAgent measures the time taken per iteration, T_{iter}, and records the triple (a, m, T_{iter}) for all combinations of resource allocations a and batch size m encountered during its lifetime. Periodically, PolluxAgent fits the parameters θ_{sys} to all of the throughput data collected so far. Specifically, we minimize the root mean squared logarithmic error (RMSLE) between Eqn. 11 and the collected data triples, using L-BFGS-B [50]. We set constraints for each α and β parameter to be non-negative, and γ to be in the range $[1,10]$. PolluxAgent then reports the updated values of θ_{sys} and φ_t to PolluxSched.

Prior-driven exploration. At the beginning of each job, throughput values have not yet been collected for many different resource allocations. To ensure that Pollux finds efficient resource allocations through systematic exploration, we impose several priors which bias θ_{sys} towards the belief that throughput scales perfectly with more resources, until such resource configurations are explored.

In particular, we set $\alpha_{sync}^{local} = 0$ while the job had not used more than one GPU, $\alpha_{sync}^{local} = \beta_{sync}^{local} = 0$ while the job had not used more than one node, and $\beta_{sync}^{local} = \beta_{sync}^{node} = 0$ while the job had not used more than two GPUs. This creates the following behavior: each job starts with a single GPU, and is initially assumed to scale perfectly to more GPUs. PolluxSched is then encouraged to allocate more GPUs and/or nodes to the job, naturally as part of its resource optimization (Sec. 4.2), until the PolluxAgent can estimate θ_{sys} more accurately. Finally, to prevent a job from being immediately scaled out to arbitrarily many GPUs, we restrict the maximum number of GPUs which can be allocated to at most twice the maximum number of GPUs the job has been allocated in its lifetime.

Training job tuning. With θ_{sys}, m_{gns}, and m_0, which fully specify the DL job's GOODPUT function at its current training progress, PolluxAgent determines the most efficient batch size,

$$m^* = \underset{m}{\operatorname{argmax}}\, \text{GOODPUT}(a, m), \quad (13)$$

where a is the job's current resource allocation. To perform this maximization efficiently, we observe that GOODPUT(a,m) is a unimodal function of m, and use golden-section search [27] to find its maximum.

Once a new batch size is found, the job will use it for its subsequent training iterations, using AdaScale to adapt its learning rate appropriately. As the job's statistical efficiency changes over time, PolluxAgent will periodically re-evaluate the most efficient batch size and learning rate.

4.2 PolluxSched: Cluster-wide Optimization

The PolluxSched periodically allocates (or re-allocates) resources for every job in the cluster. To determine a set of efficient cluster-wide resource allocations, it uses a genetic algorithm to maximize a *fitness function* which is defined as a weighted mean across speedups for each job:

$$\texttt{FITNESS}(A) = \frac{\sum_j w_j \cdot \texttt{SPEEDUP}_j(A_j)}{\sum_j w_j}. \qquad (14)$$

A is an *allocation matrix* with each row A_j being the placement vector for a job j, thus A_{jn} is the number of GPUs on node n allocated to job j. Intuitively, maximizing FITNESS causes PolluxSched to allocate more resources to jobs that achieve a high SPEEDUP when provided with many GPUs (i.e. jobs that scale well). We define the speedup of each job as the factor of goodput improvement using the given placement vector over using a single process with the optimal batch size, ie.

$$\texttt{SPEEDUP}_j(A_j) = \frac{\max_m \texttt{GOODPUT}_j(A_j, m)}{\max_m \texttt{GOODPUT}_j(1, m)}, \qquad (15)$$

where $\texttt{GOODPUT}_j$ is the goodput of job j at its current training iteration. Our definition of $\texttt{SPEEDUP}_j$ has the property that allocating a single GPU always results in a speedup of 1, and scales sub-linearly with increasing numbers of allocated GPUs. Similar to the PolluxAgent, PolluxSched performs the maximizations in the numerator and denominator using golden-section search [27].

Job weights. Although PolluxSched can work well with all weights w_j set to 1, we provide the ability to re-weight jobs as an additional tuning knob for cluster operators. Since large jobs may take up a significant fraction of cluster resources for extended amounts of time, smaller jobs may be left with fewer resources available. Depending on the preference between large jobs versus small jobs, a cluster operator can address this issue by decreasing the weight of jobs according to the amount of GPU-time already spent on them. In particular, we define the weight w_j of job j as

$$w_j = \min\left(1, \frac{\texttt{GPUTIME_THRES}}{\texttt{GPUTIME}(j)}\right)^{\lambda}. \qquad (16)$$

The weight of a job is 1 if its current total GPU-time is at most GPUTIME_THRES, and decays gradually thereafter. The parameter λ controls the rate of decay, with $\lambda = 0$ being no decay, and larger values being faster decay. We further study the effect of job weights in Sec. 5.3.2.

4.2.1 Genetic Algorithm

Our genetic algorithm operates on a population of distinct allocation matrices (see Fig. 5). During each generation of the algorithm, existing allocation matrices are first randomly mutated, then crossed over to produce offspring allocation matrices, and finally modified to satisfy node resource constraints. A constant population size is maintained after each generation by discarding the allocation matrices with the lowest objective values (according to Eqn. 14). After several generations of the genetic algorithm, the allocation matrix with the highest fitness score is applied to the jobs running in the cluster.

We describe the genetic algorithm operations in detail below, which are illustrated in Fig. 5.

Mutation. Each element A_{jn} is mutated with probability $1/N$, where N is the total number of nodes. Thus, each job suffers on average one mutation in each generation. When A_{jn} is mutated, it is set to a random integer between 0 and the total number of GPUs on node n.

Crossover. When two allocation matrices are crossed over, their rows are randomly mixed. In other words, the offspring allocation matrix consists of job allocations which are randomly selected between its two parent allocation matrices. In each generation, the allocation matrices which participate in crossover are picked using tournament selection [32].

Repair. After the mutation and crossover operations, the resultant allocation matrices may no longer satisfy resource constraints, and try to request more GPUs than are available on a node. To address this issue, random elements are decremented within columns of the allocation matrix that correspond to over-capacity nodes, until the GPU resource constraints are satisfied.

Penalty. Each time a job is re-allocated to a different set of GPUs, it will need to save a checkpoint of its current model parameters, and restart using its new allocation of GPUs. This process can typically take 30s-60s depending on the size of the model being trained. To prevent an excessive number of re-allocations, when PolluxSched evaluates the fitness function for a given allocation matrix, it applies a penalty for every job that needs to restart, i.e.

$$\texttt{SPEEDUP}_j(A_j) \longleftarrow \texttt{SPEEDUP}_j(A_j) - \texttt{RESTART_PENALTY}.$$

Interference avoidance. When multiple distributed DL jobs share a single node, their network usage while synchronizing gradients and model parameters may interfere with each other, causing both jobs to slow down [24]; Xiao et al. [46] report up to 50% slowdown for DL jobs which compete with each other for network resources. PolluxSched mitigates this issue by disallowing different distributed jobs (each using GPUs across multiple nodes) from sharing the same node.

This non-interference constraint is implemented as part of the repair step of the genetic algorithm (Sec. 4.2.1), by also removing distributed jobs from shared nodes until at most one distributed job is allocated to each node. We study the effects of interference avoidance in Sec. 5.3.2.

4.2.2 Cloud Auto-scaling

In cloud computing environments, GPU nodes can be dynamically provisioned during periods of high demand to decrease job completion time, as well as released during periods of low demand to decrease cost of cluster resources. Beyond adapting to the number and sizes of DL jobs, co-adaptive scheduling presents an unique opportunity for cluster auto-scaling. Since many distributed DL jobs tend to increase in statistical efficiency as training progresses, it may be more cost-effective to provision more cloud resources during the later iterations of a large training job, rather then earlier on.

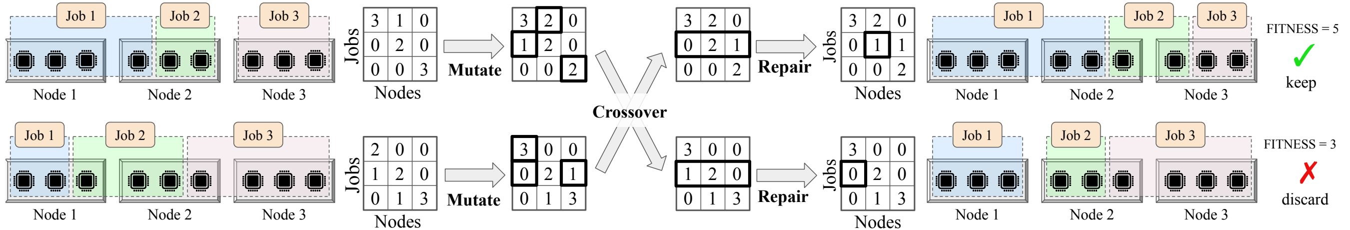


Figure 5: The mutation, crossover, and repair operations performed by PolluxSched during each generation of its genetic algorithm.

In order to decide when to request or release nodes in the cloud, we first define a measure of cluster resource utility for a given allocation matrix as

$$\texttt{UTILITY}(A) = \frac{\sum_j \texttt{SPEEDUP}_j(A_j)}{\texttt{TOTAL_GPUS}} \tag{17}$$

Since each $\texttt{SPEEDUP}_j$ is at most equal to the number of GPUs allocated to job j, then $\sum_j \texttt{SPEEDUP}_j(A_j) \leq \texttt{TOTAL_GPUS}$, and thus $0 \leq \texttt{UTILITY}(A) \leq 1$.

A cluster operator defines a LOW_UTIL_THRES and a HIGH_UTIL_THRES. PolluxSched will request additional nodes (when $\texttt{UTILITY}(A)$ is high) or release existing nodes (when $\texttt{UTILITY}(A)$ is low) until $\texttt{UTILITY}(A)$ falls within this range, where A is the current allocations applied in the cluster. The cluster operator also defines a MIN_NODES and MAX_NODES, which limit the size of the cluster.

PolluxSched performs a binary search for the desired number of nodes, with the assumption that UTILITY decreases with increasing numbers of nodes. At each step of the binary search, PolluxSched runs its genetic algorithm to evaluate the UTILITY of the cluster size being queried. In the end, the cluster size with a UTILITY value closest to $(\texttt{LOW_UTIL_THRES} + \texttt{HIGH_UTIL_THRES})/2$ is selected.

4.3 Implementation

PolluxAgent is implemented as a Python library which is imported into DL training code. We integrated PolluxAgent with PyTorch [36], which uses all-reduce as its gradient synchronization algorithm. PolluxAgent inserts performance profiling code which measures the time taken for each iteration of training, as well as calculating the gradient noise scale. At a fixed time interval, PolluxAgent fits the system throughput model (Eqn. 11) to the profiled metrics collected so far, and reports the fitted system throughput parameters, along with the latest gradient statistics, to PolluxSched. After reporting to PolluxSched, PolluxAgent updates the job's batch size, by optimizing its now up-to-date goodput function (Eqn. 6) with its currently allocated resources.

PolluxSched is implemented as a service in Kubernetes [2]. At a fixed time interval, PolluxSched runs a fixed number of generations of its genetic algorithm. It then applies the best allocation matrix to the cluster, by creating and terminating Kubernetes Pods which run the job replicas. Although only the allocation matrix with the highest fitness score is applied to the cluster, the entire population is saved and used to bootstrap the genetic algorithm in the next scheduling interval. The genetic algorithm is implemented using pymoo [6].

5 Evaluation

The primary value of Pollux is automatically choosing the right resource allocations, and adapting the batch size and learning rate of each job to best utilize those resources. In comparison, existing schedulers rely on users to specify these training parameters with well-configured values, which is unrealistic and often requires expert knowledge about the DL model structure and statistical behavior during training.

However, the ability of users to configure their jobs for efficient training is difficult to quantify. Thus, we compare Pollux with state-of-the-art DL schedulers from two perspectives. First, in Sec. 5.2, we construct a workload with ideally configured jobs, and show that Pollux still outperforms the baseline DL schedulers in this scenario. In Sec. 5.3.1, we show how the performance of Pollux and the baseline DL schedulers change when realistically configured jobs are added to the workload.

5.1 Methodology

Unless stated otherwise, we configured PolluxSched to use a 60s scheduling interval. During each scheduling interval, we run the genetic algorithm for 100 generations using a population of 100 allocation matrices. We set GPUTIME_THRES to 4 GPU-hours with $\lambda = 0.5$, and RESTART_PENALTY to 0.25. We configured PolluxAgent to report up-to-date system throughput parameters and gradient statistics every 30s.

Workload. We constructed our synthetic workload by sampling jobs from the deep learning cluster traces published by Microsoft [24]. Each job has information on its submission time, number of GPUs, and duration. However, no information is provided on the model architectures being trained or dataset characteristics. Instead, our synthetic workload consists of the models and datasets described in Table 1.

We categorized each job in the trace and in Table 1 based on their total GPU-time, calculated as the product between their duration and number of GPUs, into four categories — Small (0 to 1 GPU-hours), Medium (1 to 10 GPU-hours), Large (10 to 100 GPU-hours), and XLarge (100 to 1000 GPU-hours).

Task	Dataset	Model(s)	Validation Metric	Category	Frac. of Workload
Image Classification	ImageNet [9]	ResNet-50 [19]	75% top-1 accuracy	X-Large	2%
Object Detection	PASCAL-VOC [11]	YOLOv3 [40]	82% mAP score	Large	5%
Speech Recognition	CMU-ARCTIC [28]	DeepSpeech2 [3]	25% word error	Medium	17%
Image Classification	Cifar10 [29]	ResNet18 [19]	94% top-1 accuracy	Small	38%
Collaborative Filtering	MovieLens [18]	NeuMF [20]	71.5% hit rate	Small	38%

Table 1: Models and datasets used in our evaluation workload. Each model was trained until the provided validation metric. The fraction of jobs from each category are chosen according to the public Microsoft cluster traces.

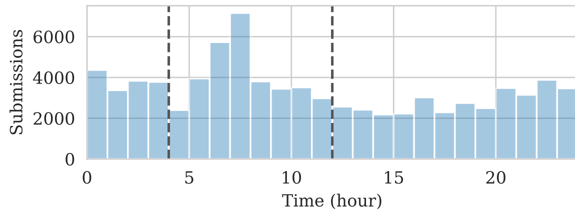


Figure 6: Number of job submissions during each hour of the day in the Microsoft trace. Our primary synthetic workload is sampled from the interval between the dashed lines.

We then picked a model and dataset from Table 1 which is in the same category as the corresponding job in the trace.

The primary workload used in our evaluation consists of 160 job submissions randomly sampled from an 8-hour period which includes the peak rate of job submissions during an average 24-hour day. Job submissions peak during the fourth hour, at 3× the rate of the first hour (Fig. 6).

Testbed. We conduct experiments using a cluster consisting of 16 nodes and 64 GPUs. Each node is an AWS EC2 `g4dn.12xlarge` instance with 4 Tesla T4 GPUs, 48 vCPUs, 192GB memory, and a 900GB SSD. All instances are launched within the same placement group. We deployed Kubernetes 1.18.2 on this cluster, along with CephFS 14.2.8 to store checkpoints for checkpoint-restart elasticity.

Simulator. We built a discrete-time cluster simulator to study the effects of the techniques proposed in Pollux. Our simulator reproduces the system throughput of the jobs in our workload, using different batch sizes, numbers of GPUs, and different placements of GPUs across nodes. It also reproduces the gradient noise scale across the lifetime of each job, enabling statistical efficiency to be calculated using the simulator. More details of our simulator can be found in Sec. 5.3.

5.2 Testbed Experiments

We compare Pollux to idealized versions of two state-of-the-art deep learning schedulers, Tiresias [16] and Optimus [38], as described in Sec. 2.3. Tiresias is non-resource-adaptive, while Optimus is only-resource-adaptive. Whereas Pollux dynamically adapts the number of GPUs and batch sizes of DL jobs, Optimus only adapts the number of GPUs, and Tiresias adapts neither. To establish a fair baseline for comparison, we tune the learning rate using AdaScale for all three schedulers.

Tiresias+TunedJobs. Tiresias requires both the number of GPUs and batch size be set by the user at the time of job submission. We manually tuned the number of GPUs and batch sizes for each job in our synthetic workload, as follows. We measured the time per training iteration for each model in Table 1 using a range of GPU allocations and batch sizes, and fully trained each model using a range of different batch sizes (see Sec. 5.3 for details). We considered a number of GPUs *valid* if using the optimal batch size for that number of GPUs achieves 50% – 80% of the ideal speedup versus using the optimal batch size on a single GPU, where the ideal speedup is defined to be equal to the number of GPUs. For each job in our workload, we then selected the number of GPUs and batch size randomly from its set of valid configurations.

Our job configurations essentially assume that the users are highly rational and knowledgeable about the scalability of the models they are training. A speedup of less than 50% of the ideal would lead to under-utilization of resources, and a speedup of more than 80% means the job can still be further parallelized efficiently. We emphasize that this assumption of uniformly sophisticated users is unrealistic in practice and only serves for evaluating the performance of Tiresias in an ideal world.

Optimus+Oracle. For Optimus, we use the same batch sizes for each job as for Tiresias. However, since Optimus automatically decides resource allocations, the number of GPUs set for each job is ignored. If a job's batch size does not fit on a single GPU, we additionally enforce a minimum number of GPUs, such that the total batch size will fit on that many GPUs.

Like Pollux, Optimus leverages a performance model to predict the throughput of each job given different amounts of cluster resources. However, its performance model is specific to the parameter server architecture for synchronizing gradients. To account for this difference, our implementation of Optimus uses our own throughput model as described in Sec. 3.2.

Furthermore, Optimus leverages a prediction of the number of training iterations until convergence, by fitting a simple function to the model's convergence curve. However, training realistic models to an acceptable quality, including the ones in our workload, requires learning rate decay schedules that make the convergence curve non-smooth and difficult to extrapolate from. To eliminate any mispredictions, and for the purposes of our evaluation, we run each job ahead of time, and provide Optimus with the exact number of iterations until completion.

Policy	Job Completion Time		Makespan
	Average	99%tile	
Pollux	1.2h	8.8h	20h
Optimus+Oracle	1.6h	11h	24h
Tiresias+TunedJobs	2.4h	16h	33h

Table 2: Summary of testbed experiments. Even in the unrealistic scenario where each job is ideally-tuned before being submitted, Pollux still outperforms baseline DL schedulers.

5.2.1 Results

Even when every job is pre-configured with an ideal number of GPUs and batch size, Pollux outperforms both Optimus+Oracle and Tiresias. Table 2 shows the detailed results. Pollux achieves 25% lower average JCT and 17% shorter makespan than Optimus+Oracle, and 50% lower average JCT and 39% shorter makespan than Tiresias. We observe similar differences for tail JCTs—Pollux achieves a 20% and 45% lower 99th percentile JCT than Optimus+Oracle and Tiresias+TunedJobs, respectively.

On average, we measured that Pollux maintained $\approx 91\%$ statistical efficiency across all jobs running in the cluster at any given time, while Optimus+Oracle and Tiresias could only maintain $\approx 74\%$. Since the jobs in our workload already use the most efficient *fixed* batch size, this result indicates that Pollux is able to adaptively tune the batch size to achieve better statistical efficiency.

Furthermore, we find that over the lifetime of each job, Pollux achieves on average $1.2\times$ and $1.5\times$ higher throughput than Optimus+Oracle and Tiresias+TunedJobs, respectively. This is due to the fact that Pollux may increases the batch size to take advantage of more GPUs when allocated. In terms of goodput, the improvement for Pollux is wider, being on average $1.4\times$ and $2\times$ higher than Optimus+Oracle and Tiresias+TunedJobs, respectively.

5.3 Simulator Experiments

We use a cluster simulator in order to evaluate a broader set of workloads and settings. Our simulator is constructed by measuring the performance and gradient statistics of each model in Table 1, under many different resource and batch size configurations, and re-playing them for each simulated job. This way, we are able to simulate both the system throughput and statistical efficiency of the jobs in our workload.

Unless stated otherwise, each experiment in this section is repeated 8 times on different traces generated using the same duration, number of jobs, and job size distributions as in Sec. 5.2, and we report the average results across all 8 traces.

Simulating system throughput. We measured the time per training iteration for all allocations of GPUs in a 4-node cluster with 4 GPUs each, removing symmetric placements. For each allocation, we used a range of batch sizes, spaced geometrically by factors of $\approx \sqrt{2}$, up to the largest batch size which fit into the total GPU memory. We did the same for all valid combinations of 4 to 16 nodes and 1 to 4 GPUs per node, where the same number of GPUs is allocated from each node. Finally, to simulate the throughput for a job, we perform a multi-dimensional linear interpolation between the nearest configurations we measured.

Simulating statistical efficiency. We measured the gradient noise scale during training using a range of batch sizes, spaced geometrically by factors of $\approx \sqrt{2}$. To simulate the statistical efficiency for a job using a certain batch size, we linearly interpolated its value of the gradient noise scale between the two nearest batch sizes we measured.

Simulator fidelity. The data we collected about each job enables our simulator to reproduce several system effects, including the performance impact of different GPU placements. We also simulate the overhead of checkpoint-restarts by injecting a 30-second delay for each job which has its resources reallocated. Unless stated otherwise, we do not simulate any network interference between different jobs. Therefore, each distributed job runs at its full speed, ignoring other distributed jobs. We study the effects of interference in more detail in Sec. 5.3.2.

Compared with our testbed experiments in Sec. 5.2, we find that our simulator is able to reproduce similar factors of improvement. Our simulated results show that Pollux reduces the average JCT by 26% and 40% over Optimus+Oracle and Tiresias+TunedJobs, respectively, compared with 25% and 50% in our testbed experiments.

5.3.1 Workloads with Realistic Jobs

In Sec. 5.2, we compared Pollux with state-of-the-art DL schedulers in the scenario that every submitted job is already configured with an ideal number of GPUs and batch size. Without assistance from a system like Pollux, users likely need to try many different configurations for each job, before finding one that is reasonable. Each trial run by the user is yet another job configured with a potentially poor choice of GPUs and batch size.

In this section, we compare Pollux with Optimus+Oracle and Tiresias using realistic job configurations. For each job, we use the number of GPUs as specified in the Microsoft traces, which were decided by real users of a DL cluster. We then select a random batch size which is within a factor of 2 from the most efficient batch size for that number of GPUs, according to our simulator data. We constructed several workload traces using various mixtures of ideally-tuned jobs (Sec. 5.2), and user-configured jobs from the Microsoft cluster trace.

Fig. 7 shows the results. First, the performance of Pollux is unaffected by the inclusion of the user-configured jobs. This is simply because both the number of GPUs and the batch size are decided automatically by Pollux, and co-adapted during training, rather than being set by users at job submission time. On the other hand, the performance of Tiresias degrades quickly as more user-configured jobs are included in the workload. We find that many users requested a small number

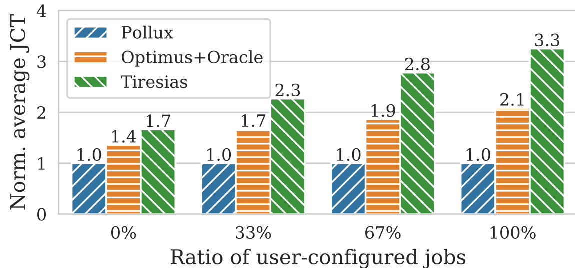


Figure 7: Average JCT (relative to Pollux) for workloads with increasing ratios of realistic, user-configured jobs. 100% corresponds to jobs exactly as configured in the Microsoft trace. 0% corresponds to the trace used in Sec. 5.2.

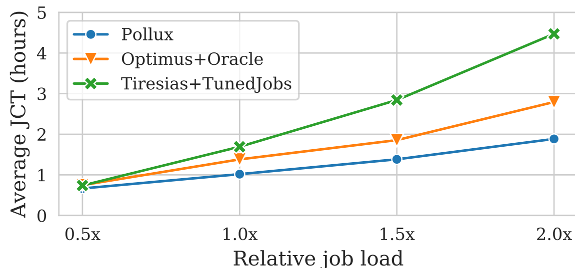


Figure 8: Average JCT for various number of jobs submitted within an 8-hour period.

of GPUs, when they could still have efficiently utilized more GPUs—especially in the later stages of each job, when the statistical efficiency of using larger batch sizes is high.

The performance of Optimus+Oracle also degraded, but not by as much as Tiresias. We found that even though Optimus+Oracle is able to allocate more GPUs to each job, the fact that it does not also adapt the batch size means those extra GPUs are under-utilized in terms of system throughput. Overall, when all job configurations are derived from the Microsoft cluster traces (corresponding to the 100% group in Fig. 7), Pollux reduces the average JCT by 50% relative to Optimus+Oracle, and by 70% relative to Tiresias.

5.3.2 Other Effects on Scheduling

Sensitivity to load. We compare the performace of Pollux with Optimus+Oracle and Tiresias+TunedJobs for increasing load, in terms of the rate of job submissions. Fig. 8 shows the results. As expected, all three scheduling policies suffer longer job completion times as the load is increased. However, the advantage of Pollux is even more noticeable at higher loads. At 2× the load, the average job completion time of Pollux increased by 1.8×, while that of Optimus+Oracle and Tiresias+TunedJobs increased by 2.0× and 2.6×, respectively.
Impact of job weights. We investigate the impact of job weights (Eqn 16) by comparing different values for the decay parameter λ. In particular, $\lambda = 0$ represents our baseline, ie. when each job is given a weight of 1. Table 3 summarizes the results. We find that increasing λ significantly improves

Decay (λ)	Avg. JCT	50%tile JCT	99%tile JCT
0.0	1	1	1
0.5	0.95	0.77	1.05
1.0	0.98	0.68	1.20

Table 3: Job completion time for various degrees of job weight decay λ. Results are shown relative to $\lambda = 0$, which always assigns a weight of 1 to every job.

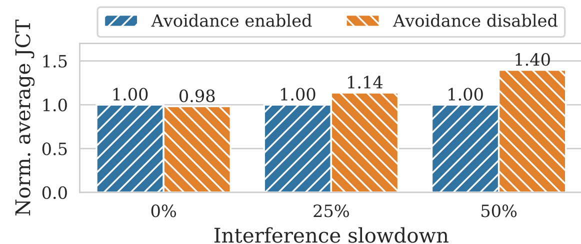


Figure 9: Average JCT for various degrees of artificial network contention, interference avoidance enabled vs. disabled.

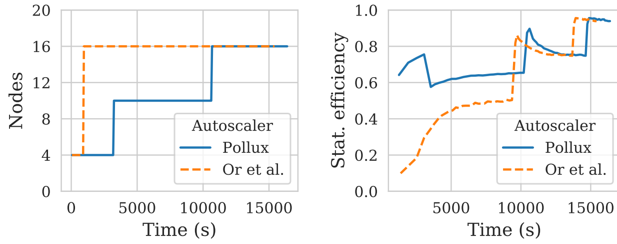
the 50th percentile JCT, while moderately degrading the 99th percentile JCT. These results show that job weighting effectively prioritizes smaller jobs to finish quickly ahead of larger jobs. On the other hand, we find only a small impact on the average JCT, which improves by 5% at $\lambda = 0.5$.
Impact of interference avoidance. PolluxSched attempts to avoid interference between DL jobs by disallowing cluster allocations which include two distributed jobs that share the same node. Although this strategy guarantees that network resources on each node are reserved for at most one job, it also constrains the set of feasible cluster allocations. The genetic algorithm may have a more difficult time finding an efficient cluster allocation. On the other hand, Xiao et al. [46] report up to 50% slowdown for DL jobs which compete with each other for network resources.

To evaluate the impact of PolluxSched's interference avoidance constraint, we artificially inject various degrees of slowdown for distributed jobs sharing the same node. Fig. 9 shows the results. With interference avoidance enabled, job completion time is unaffected by more severe slowdowns, because network contention is completely mitigated. However, without interference avoidance, job completion times increase significantly, up to 1.4× when interference slowdown is 50%.

On the other hand, in the ideal scenario when there is zero slowdown due to interference, PolluxSched with avoidance disabled only performs 2% better. This result indicates that PolluxSched is still able to find efficient cluster allocations while obeying the interference avoidance constraint.

5.3.3 Cloud Auto-scaling

In cloud environments, computing resources can be obtained and released as required, and users pay for the duration they hold onto those resources. In this scenario, it is less likely that



(a) Number of nodes over time. (b) Statistical efficiency over time.

Figure 10: Goodput-based auto-scaling (Pollux) vs throughput-based auto-scaling (Or et al.) for ImageNet training.

a DL training job is competing with other jobs for resources, but rather with the cost of renting resources. Thus, although Pollux is capable of auto-scaling a cluster in the cloud which runs many concurrent jobs (Sec. 4.2.2), we focus this section on training a single large model in the cloud.

Auto-scaling ImageNet training. Since the statistical efficiency of DL training typically increases over time, larger batch sizes can be utilized more effectively later on in training. Pollux measures the statistical efficiency during training, and is able to provision more resources to accelerate large-batch training when it is the most impactful, and fewer resources to save cost while statistical efficiency is low.

We compare Pollux with the cloud auto-scaling strategy proposed by Or et al. [35], which allows the batch size to be increased during training, but models job performance using the system throughput rather than the goodput.

Fig. 10 shows the results. Since the system throughput is a measure which does not change according to the training progress, throughput-based autoscaling (Or et al.) quickly scales out to more nodes and a larger batch size (Fig. 10a), which remains constant thereafter. On the other hand, Pollux knows that the statistical efficiency near the beginning of the job is low for large batch sizes, and gradually increases the number of nodes as the effectiveness of larger batch sizes improves over time. Fig. 10b shows that Pollux maintains a high statistical efficiency throughout training.

Overall, we found that Pollux can train ImageNet for 25% cheaper cost, with only a 6% longer completion time than auto-scaling policies based on throughput only.

6 Related Work

Resource scheduling for DL. Several recent work have explored specialized scheduling strategies for DL training. Gandiva [46] profiles the performance of DL training jobs during runtime, in order to dynamically migrate jobs for better resource placement. Tiresias [16] exploits the characteristics of large-scale DL workload traces to improve scheduling performance while requiring little information about the run-time predictability of each individual job. Compared with Pollux, Gandiva and Tiresias are agnostic to the resource-scalability of each job, and do not prioritize more resources for jobs that can utilize them more effectively.

Elasticity for ML and DL training [17, 22, 33, 39] has been leveraged for improving scheduling performance in shared clusters. SLAQ [49] aims to improve the average quality of all jobs in the cluster, by re-directing more resources towards jobs whose quality is improving the fastest. Optimus [38] optimizes for the average job completion time, by re-directing more resources towards jobs that are expected to improve the average JCT the most. Unlike Pollux, SLAQ and Optimus only adapt resource allocations, without also adapting batch sizes and learning rates for each job.

Adaptive batch size training. Recent work on DL training algorithms have explored dynamically adapting batch sizes for better efficiency and parallelization. AdaBatch [10] increases the batch size at pre-determined iterations during training, while linearly scaling the learning rate. Smith et al. [44] suggest that instead of decaying the learning rate during training, the batch size should be increased instead. CABS [4] adaptively tunes the batch size and learning rate during training using similar gradient statistics as Pollux.

These works have a common assumption that extra computing resources are available to parallelize larger batch sizes whenever desired, which is rarely true inside shared-resource environments. Pollux complements existing adaptive batch size strategies by adapting the batch size and learning rate in conjunction with the amount of resources currently available.

Hyper-parameter tuning. A large body of work focuses on tuning the hyper-parameters for machine learning and deep learning models [5, 12, 23, 26, 34], which typically involves managing shared resources between many training jobs [1, 14]. Although batch size and learning rate are within the space of hyper-parameters optimized by these systems, Pollux's goal is fundamentally different. Whereas hyper-parameter tuning systems search for the highest model quality, Pollux adapts the batch size and learning rate for the most efficient execution for each job, while not degrading model quality. Pollux may complement hyper-parameter tuning systems by efficiently scheduling the many training jobs spawned to run each trial.

7 Conclusion

Pollux is a DL cluster scheduler that co-adaptively allocates resources, while at the same time tuning each training job to best utilize those resources. We present a formulation of goodput that combines system throughput and statistical efficiency for distributed DL training. Based on the principle of goodput maximization, Pollux automatically and jointly tunes the resource allocations, batch sizes, and learning rates for DL jobs, which can be particularly difficult for users to configure manually. Pollux outperforms state-of-the-art DL schedulers even if users can configure their jobs well, and reduces the cost of training large models in the cloud.

References

[1] Introduction to katib | kubeflow. https://www.kubeflow.org/docs/components/hyperparameter-tuning/overview/. Accessed: 2020-05-18.

[2] Production-grade container orchestration - kubernetes. https://kubernetes.io/. Accessed: 2020-05-18.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Vaino Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 173–182. JMLR.org, 2016.

[4] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling adaptive batch sizes with learning rates. *CoRR*, abs/1612.05086, 2016.

[5] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[6] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, pages 1–1, 2020.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[10] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *CoRR*, abs/1712.02029, 2017.

[11] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[12] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[13] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W. Mahoney, and Joseph Gonzalez. On the computational inefficiency of large batch sizes for stochastic gradient descent. *CoRR*, abs/1811.12941, 2018.

[14] 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*, KDD '17, page 1487–1495, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[16] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

[17] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17,

page 589–604, New York, NY, USA, 2017. Association for Computing Machinery.

[18] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015.

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

[20] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 173–182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[21] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[22] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[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] Tyler B. Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. Adascale {sgd}: A scale-invariant algorithm for distributed training, 2020.

[26] Kirthevasan Kandasamy, Karun Raju Vysyaraju, Willie Neiswanger, Biswajit Paria, Christopher R Collins, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Tuning hyperparameters without grad students: Scalable and robust bayesian optimisation with dragonfly. *arXiv preprint arXiv:1903.06694*, 2019.

[27] J. Kiefer. Sequential minimax search for a maximum. *Proceedings of the American Mathematical Society*, 4(3):502–506, 1953.

[28] John Kominek and Alan Black. The cmu arctic speech databases. *SSW5-2004*, 01 2004.

[29] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[30] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[31] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *CoRR*, abs/1812.06162, 2018.

[32] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[33] Shravan Narayanamurthy, Markus Weimer, Dhruv Mahajan, Tyson Condie, Sundararajan Sellamanickam, and S. Sathiya Keerthi. Towards resource-elastic machine learning, 2013.

[34] Willie Neiswanger, Kirthevasan Kandasamy, Barnabas Poczos, Jeff Schneider, and Eric Xing. Probo: a framework for using probabilistic programming in bayesian optimization. *arXiv preprint arXiv:1901.11515*, 2019.

[35] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. In *Proceedings of Machine Learning and Systems 2020*, pages 400–411. 2020.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[37] Chao Peng, Tete Xiao, Zeming Li, Yuning Jiang, Xiangyu Zhang, Kai Jia, Gang Yu, and Jian Sun. Megdet: A large mini-batch object detector. *CoRR*, abs/1711.07240, 2017.

[38] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic

resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 631–644, Boston, MA, July 2018. USENIX Association.

[40] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[41] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*, page 318–362. MIT Press, Cambridge, MA, USA, 1986.

[42] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[43] Christopher J. Shallue, Jaehoon Lee, Joseph M. Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. *CoRR*, abs/1811.03600, 2018.

[44] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[45] WenWu Wang and Ping Yu. Asymptotically optimal differenced estimators of error variance in nonparametric regression. *Computational Statistics & Data Analysis*, 105:125 – 143, 2017.

[46] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[47] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-batch training for LSTM and beyond. *CoRR*, abs/1901.08256, 2019.

[48] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, July 2017. USENIX Association.

[49] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 390–404, New York, NY, USA, 2017. Association for Computing Machinery.

[50] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, December 1997.

A Relation between Adascale [25] and Gradient Noise Scale [31]

In Sec. 3, we've defined the statistical efficiency of a DL job running with batch size $m > m_0$ to be the amount of training progress made per unit of training data processed using m relative to using m_0. Previous work has aimed to analyze this quantity. Specifically, [31] formulates the amount of training progress that can be made in a single update using a batch of size m relative to the amount of progress made with a batch of size m_0. Similarly, Adascale's [25] r_t (Sec. 2.2) can be framed as the relative training progress made using batch size m versus m_0. We can express the statistical efficiency of both works in terms of the gradient noise scale φ_t (Sec. 2.2). Here, we show an equivalence between the formulations of efficiency given by both papers

In [31], the gradient noise scale is shown to be the largest value of the batch size m at which the optimal learning rate is under 50% of the optimal learning rate for the full (non-batch) gradient. Scaling the batch size m to below φ_t can allow for linear scaling of the learning rate, while scaling m to above φ_t does not allow for proportional scaling of the learning rate and thus yields diminishing returns.

Given the gradient $g^{(t)}$ and Hessian $H^{(t)}$ for the model parameters at a given iteration t, [31] approximates the optimal (non-batch) learning rate as

$$\eta_{\max} = \frac{|g^{(t)}|^2}{{g^{(t)}}^T H^{(t)} g^{(t)}} = \frac{\mu_t^2}{{g^{(t)}}^T H^{(t)} g^{(t)}}.$$

Given a gradient estimate $\hat{g}^{(t)}$ formed using a batch of size m_0, [31] shows that the optimal learning rate $\eta_{\text{opt}} \leq \eta_{\max}$ can be written

$$\eta_{\text{opt}}(m) = \frac{\eta_{\max}}{1 + \varphi_t / m},$$

where φ_t is defined to be

$$\varphi_t = \frac{\operatorname{tr}\left(H^{(t)} \Sigma^{(t)}\right)}{{g^{(t)}}^T H^{(t)} g^{(t)}}.$$

and where the single-example covariance matrix $\Sigma^{(t)}$ is defined to be

$$\Sigma^{(t)} = \operatorname{cov}_{x \in X}\left(\nabla \ell(w^{(t)}, x)\right).$$

The gradient noise scale is difficult to compute in practice, so [31] describes a simplification. If we assume that $H^{(t)} = cI$ for some constant c, then we can write the gradient noise scale as

$$\varphi_t = \frac{\operatorname{tr}\left(\Sigma^{(t)}\right)}{|g^{(t)}|^2} = \frac{m_0 \sigma_t^2}{\mu_t^2},$$

where $\sigma_t^2 = \operatorname{Var}[\hat{g}^{(t)}]$ is the variance and $\mu_t^2 = |\mathbb{E}[\hat{g}^{(t)}]|^2$ the squared norm of the gradient at iteration t using batch size m_0.

Under these assumptions we can also write the optimal learning rate for a batch of size m as

$$\eta_{\text{opt}}(m) = \frac{\eta_{\max}}{1 + \varphi_t / m} = \frac{k \mu_t^2}{\mu_t^2 + \frac{m_0}{m} \sigma_t^2}$$

for some k. The relative training progress, written as the ratio of optimal learning rates at batch size m versus m_0 is then

$$\frac{\eta_{\text{opt}}(m)}{\eta_{\text{opt}}(m_0)} = \frac{\mu_t^2 + \sigma_t^2}{\mu_t^2 + \frac{m_0}{m} \sigma_t^2} = \frac{1 + \varphi_t / m_0}{1 + \varphi / m},$$

and the statistical efficiency can therefore be written

$$\texttt{EFFICIENCY}_t(m) = \frac{\eta_{\text{opt}}(m)}{\eta_{\text{opt}}(m_0)} \frac{m_0}{m} = \frac{\varphi_t + m_0}{\varphi + m}.$$

Alternatively, in Adascale, [25] r_t (Sec. 2.2) can be framed as the relative training progress made using batch size m versus m_0. Adascale writes r_t as

$$r_t = \frac{\sigma_t^2 + \mu_t^2}{\frac{m_0}{m} \sigma_t^2 + \mu_t^2} \tag{18}$$

where σ_t^2 and μ_t^2 are again the variance and squared norm of the gradient at iteration t using batch size m_0. Since $\varphi = m_0 \sigma_t^2 / \mu_t^2$, we can rewrite r_t as

$$r_t = \frac{\varphi_t / m_0 + 1}{\varphi_t / m + 1}, \tag{19}$$

and can thus write the efficiency as

$$\texttt{EFFICIENCY}_t(m) = r_t \frac{m_0}{m} = \frac{\varphi_t + m_0}{\varphi + m}.$$