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

Aurick Qiao[1,2] Sang Keun Choe[2] Suhas Jayaram Subramanya[2] Willie Neiswanger[1,2]
Qirong Ho[1] Hao Zhang[1,3] Gregory R. Ganger[2] Eric P. Xing[4,1,2]

[1]*Petuum, Inc.* [2]*Carnegie Mellon University* [3]*UC Berkeley* [4]*MBZUAI*

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 expect users to specify the number of resources for each job, often leading to inefficient resource use. Some recent schedulers choose job resources for users, but do so without awareness of how DL training can be re-optimized to better utilize the provided resources.

Pollux simultaneously considers both aspects. By monitoring the status of each job during training, Pollux models how their *goodput* (a metric we introduce to combine system throughput with statistical efficiency) would change by adding or removing resources. Pollux dynamically (re-)assigns resources to improve cluster-wide goodput, while respecting fairness and continually optimizing each DL job to better utilize those resources.

In experiments with real DL jobs and with trace-driven simulations, Pollux reduces average job completion times by 37–50% relative to state-of-the-art DL schedulers, even when they are provided with ideal resource and training configurations for every job. Pollux promotes fairness among DL jobs competing for resources, based on a more meaningful measure of *useful* job progress, and reveals a new opportunity for reducing DL cost in cloud environments. Pollux is implemented and publicly available as part of an open-source project at `https://github.com/petuum/adaptdl`.

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, often 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 [31, 67], with a scheduler that mediates resource sharing between many competing DL jobs.

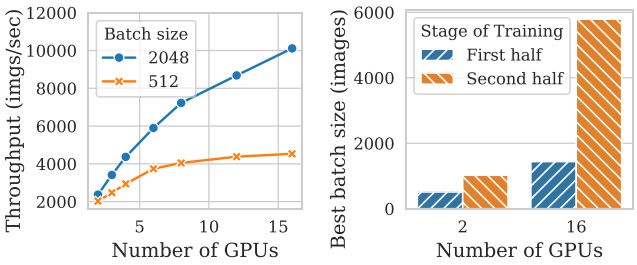
Existing schedulers require users to manually configure their jobs, which if done improperly, can greatly degrade training performance and resource efficiency. For example, allocating too many GPUs may result in long queuing times and inefficient resource usage, while allocating too few GPUs may result in long runtimes and unused resources. Such decisions are especially difficult to make in a shared-cluster setting, since optimal choices are dynamic and depend on the cluster load while a job is running.

Even though recent *elastic* schedulers can automatically select an appropriate amount of resources for each job, they do so blindly to inter-dependent training-related configurations that are just as important. For example, the *batch size* and *learning rate* of a DL job influence the amount of computation needed to train its model. Their optimal choices vary between different DL tasks and model architectures, and they have strong dependence on the job's allocation of resources.

The amount of resources, batch size, and learning rate are difficult to configure appropriately without expert knowledge about both the cluster hardware performance and DL model architecture. Due to the inter-dependence between their optimal values, they should be configured jointly with each other. Due to the dynamic nature of shared clusters, their optimal values may change over time. This creates a complex web of considerations a user must make in order to configure their job for efficient execution and resource utilization.

How can a cluster scheduler help to automatically configure user-submitted DL jobs? Fundamentally, a properly-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 compute resources (e.g., more GPUs). But, even with an optimally-retuned learning rate, increasing the batch size often results in a decreased statistical efficiency [46, 57]. For every distinct allocation of GPUs, there is potentially a different batch size that best balances increasing



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

(b) The most efficient batch size depends on the allocated resources and 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.

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 [46].

Guided by these insights, this paper presents *Pollux*, a hybrid resource scheduler that *co-adaptively* allocates resources and tunes the batch size and learning rate for all DL jobs in a shared cluster. Pollux achieves this by jointly managing several system-level and training-related parameters, including the number of GPUs, co-location of workers, per-GPU batch size, gradient accumulation, and learning rate scaling. In particular:

★ We propose a formulation of *goodput* for DL jobs, which is a holistic measure of training performance that takes into account both system throughput and statistical efficiency.

★ We show that a model of a DL job's goodput can be learned by observing its throughput and statistical behavior during training, and used for predicting the performance given different resource allocations and batch sizes.

★ We design and implement a scheduling architecture that uses such models to configure the right combination of resource allocation and training parameters for each pending and running DL job. This includes locally tuning system-level and training-related parameters for each DL job, and globally optimizing cluster-wide resource allocations. The local and global components actively communicate and cooperate with each other, operating based on the common goal of goodput maximization.

★ We evaluate Pollux on a cluster testbed using a workload derived from a Microsoft cluster trace. Compared with recent DL schedulers, Tiresias [22] and Optimus [52], Pollux reduces the average job completion time by up to 73%. Even when all jobs are manually tuned beforehand, Pollux reduces the average job completion time by 37%–50%. At the same time, Pollux improves finish-time fairness [43] by 1.5×–5.4×.

★ We show that, in cloud environments, using goodput-driven auto-scaling based on Pollux can potentially reduce the cost of training large models 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 is an individual sample in X, and ℓ is the loss evaluated at a single sample.

The loss function can be minimized using stochastic gradient descent (SGD) or its variants like AdaGrad [15] and Adam [36]. For the purpose of explaining system throughput and statistical efficiency, we will use SGD as the running example. SGD repeatedly applies the following update until the loss converges to a stable value: $w^{(t+1)} = w^{(t)} - \eta \hat{g}^{(t)}$. 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* $\mathcal{M}^{(t)} \subset X$ of the training data:

$$\hat{g}^{(t)} = \frac{1}{M}\sum_{x_i \in \mathcal{M}^{(t)}} \nabla \ell(w^{(t)}, x_i). \tag{2}$$

The learning rate η and batch size $M = |\mathcal{M}^{(t)}|$ 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 (e.g. GPUs) assigned to the job, (2) the method of distributed execution and synchronization, and (3) the batch size.

Data-parallel execution. *Synchronous 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 $\mathcal{M}^{(t)}$ is divided into equal-sized partitions per node, $\mathcal{M}_1^{(t)}, \ldots, \mathcal{M}_K^{(t)}$. Each GPU k computes a local gradient estimate $\hat{g}_k^{(t)}$ using its own partition:

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

where $m = |\mathcal{M}_k^{(t)}|$ is the per-GPU batch size. These local gradient estimates are then averaged across all GPUs to obtain the desired $\hat{g}^{(t)}$. Finally, each node applies the same update using $\hat{g}^{(t)}$ to obtain the new model parameters $w^{(t+1)}$.

The run-time of each training iteration is determined by two main components. *First*, the time spent computing each $\hat{g}_k^{(t)}$, which we denote by T_{grad}. *Second*, the time spent

averaging $\hat{g}_k^{(t)}$ (e.g. using collective all-reduce [51, 56]) and/or synchronizing $w^{(t)}$ (e.g. using parameter servers [8, 11, 26, 53]) across all GPUs, which we denote by T_{sync}. T_{sync} is influenced by the size of the gradients, performance of the network, and is typically shorter when the GPUs are co-located within the same physical node or rack.

Limitations due to the batch size. When the number of GPUs is increased, T_{grad} decreases due to a smaller per-GPU batch size. On the other hand, T_{sync}, which is typically independent of the batch size, remains unchanged. By Amdahl's Law, no matter how many GPUs are used, the run-time of each training iteration is lower bounded by T_{sync}. To overcome this scalability limitation, a common strategy is to increase the batch size. Doing so causes the local gradient estimates to be computed over more training examples and thereby increasing the ratio of T_{grad} to T_{sync}. As a result, using a larger batch size enables higher system throughput when scaling to more GPUs in the synchronous data-parallel setting.

2.2 Statistical Efficiency

The *statistical efficiency* of DL training can be defined as the amount of training progress made per unit of training data processed, influenced by parameters such as *batch size* or *learning rate*; for example, a larger batch size normally decreases the statistical efficiency. The ability to predict statistical efficiency is key to improving said statistical efficiency, because we can use the predictions to better adapt the batch sizes and learning rates.

Gradient noise scale. Previous work [32, 46] relate the statistical efficiency of DL training to the *gradient noise scale* (GNS), which measures the noise-to-signal ratio of the stochastic gradient. A larger GNS 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 GNS can vary greatly between different DL models [19]. It is also non-constant and tends to gradually increase during training, by up to $10\times$ or more [46]. Thus, it is possible to attain significantly better statistical efficiency for large batch sizes later on during training.

The gradient noise scale mathematically captures an intuitive explanation of how the batch size affects statistical efficiency. When the stochastic gradient has low noise, adding more training examples to each mini-batch does not significantly improve each gradient estimate, which lowers statistical efficiency. When the stochastic gradient has high noise, adding more training examples to each mini-batch reduces the noise of each gradient estimate, which maintains high statistical efficiency. Near convergence, the stochastic gradients have relatively lower signal than noise, and so larger batch sizes can be more useful later in training.

Learning rate scaling. When training with an increased total batch size M, the learning rate η should also be increased, otherwise the final trained model quality/accuracy can be significantly worse [57]. How to increase the learning rate varies between different models and training algorithms (e.g. SGD, Adam [36], AdamW [42]), and several well-established scaling rules may be used. For example, the linear scaling rule [21], which prescribes that η be scaled proportionally with M, or the square-root scaling rule [40, 69] (commonly used with Adam), which prescribes that η be scaled proportionally with $\sqrt{M}$. More recent scaling rules such as AdaScale [32] may scale the learning rate adaptively during training.

In addition to decreasing statistical efficiency, using large batch sizes may also degrade the final model quality in terms of validation performance [19, 35, 60], although the reasons behind this effect are not completely understood at the time of this paper. However, for each of the learning rate scaling rules mentioned above, there is usually a problem-dependent range of batch sizes that achieve similar validation performances. Within these ranges, the batch size may be chosen more freely without significantly degrading the final model quality.

2.3 Existing DL Schedulers

We broadly group existing DL schedulers into two categories, to put Pollux in context. First, *non-scale-adaptive* schedulers are agnostic to the performance scalability of DL jobs with respect to the amount of allocated resources. For example, Tiresias [22] requires users to specify the number of GPUs at the time of job submission, which will be fixed for the lifetime of the job. Gandiva [66] also requires users to specify number of GPUs, but enhances resource utilization through fine-grained time sharing and job packing. Although Gandiva may dynamically change the number of GPUs used by a job, it does so opportunistically and not based on knowledge of job scalability.

Second, *scale-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 [52] 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 [71], which was not evaluated on DL, uses a similar technique to minimize the average loss values for training general ML models. Gavel [48] goes further by scheduling based on a throughput metric that is comparable across different accelerator types.[1] AntMan [67] uses dynamic scaling and fine-grained GPU sharing to improve cluster utilization, resource fairness, and job completion times. Themis [43] introduces the notion of finish-time fairness, and promotes fairness between multiple DL applications with a two-level scheduling architecture.

Crucially, existing schedulers are agnostic to the statistical efficiency of DL training and the inter-dependence of resource decisions and training parameters. Pollux explicitly co-adapts these inter-dependent values to improve goodput for DL jobs.

[1] Pollux's current throughput model does not consider accelerator heterogeneity. We believe that extending with Gavel's metric would allow Pollux to co-adapt for goodput in heterogeneous DL clusters.

没看懂

3 The Goodput of DL Training and Pollux

In this section, we define the *goodput*[2] of DL jobs, which is a measure of training performance that takes into account both system throughput and statistical efficiency. We then describe how the goodput can be measured during training and used as a predictive model, which is leveraged by Pollux to jointly optimize cluster-wide resource allocations and batch sizes.

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

$$\texttt{GOODPUT}_t(\star) = \texttt{THROUGHPUT}(\star) \times \texttt{EFFICIENCY}_t(M(\star)), \quad (4)$$

where $\star$ represents any configuration parameters that jointly influence the throughput and batch size during training, and M is the total batch size summed across all allocated GPUs.

While the above definition is general across many training systems, we focus on three configuration parameters of particular impact in the context of efficient resource scheduling, i.e. $\star = (a,m,s)$, where:

- $a \in \mathbb{Z}^N$: the *allocation vector*, where a_n is the number of GPUs allocated from node n.
- $m \in \mathbb{Z}$: the *per-GPU batch size*.
- $s \in \mathbb{Z}$: number of *gradient accumulation steps* (§3.2).

The total batch size is then defined as

$$M(a,m,s) = \texttt{SUM}(a) \times m \times (s+1).$$

Pollux's approach. An initial batch size M_0 and learning rate (LR) η_0 are selected by the user when submitting their job. Pollux will start each job using a single GPU, $m = M = M_0$, $s = 0$, and $\eta = \eta_0$. As the job runs, Pollux profiles its execution to learn and refine predictive models for both THROUGHPUT (§3.2) and EFFICIENCY (§3.1). Using these predictive models, Pollux periodically re-tunes (a,m,s) for each job, according to cluster-wide resource availability and performance (§4.2).

$\texttt{EFFICIENCY}_t$ is measured *relative to* the initial batch size M_0 and learning rate η_0, and Pollux only considers batch sizes that are at least the initial batch size, ie. $M \geq M_0$. In this scenario, $\texttt{EFFICIENCY}_t(M)$ is a fraction (between 0 and 1) relative to $\texttt{EFFICIENCY}_t(M_0)$. Therefore, goodput can be interpreted as the portion of the throughput that is useful for training progress, being equal to the throughput if and only if perfect statistical efficiency is achieved.

Plug-in Learning Rate Scaling. Recall from §2.2 that different training jobs may require different learning rate scaling rules to adjust η in response to changes in M. In order to support a wide variety of LR scaling rules, including state-of-the-art rules such as AdaScale [32], Pollux provides a plug-in interface that can be implemented using a function signature

$$\texttt{SCALE\_LR}(M_0, M) \longrightarrow \lambda.$$

SCALE_LR is called before every model update step, and λ is used by Pollux to scale the learning rate. The implementation of SCALE_LR can utilize metrics collected during training, such as the gradient noise scale. Using this interface, one can implement rules including AdaScale, square-root scaling [40], linear scaling [21] and LEGW [69].

3.1 Modeling Statistical Efficiency

We model $\texttt{EFFICIENCY}_t(M)$ as the amount of progress made per training example using M, relative to using M_0. For SGD-based training, this quantity can be expressed in terms of the gradient noise scale (GNS) [46]. To support popular adaptive variants of SGD like Adam [36] and AdaGrad [64], we use the *pre-conditioned gradient noise scale* (PGNS), derived by closely following the original derivation of the GNS ("simple" noise scale in [46]) starting from pre-conditioned SGD[3] rather than vanilla SGD. The PGNS, which we denote by φ_t, is expressed as

$$\varphi_t = \frac{\mathrm{tr}(P\Sigma P^T)}{|Pg|^2}, \quad (5)$$

where g is the true gradient, P is the pre-conditioning matrix of the adaptive SGD algorithm, and Σ is the covariance matrix of per-example stochastic gradients. The PGNS is a generalization of the GNS and is mathematically equivalent to the GNS for the special case of vanilla SGD.

Similar to the GNS (Appendix D of [46]), it takes $1+\varphi_t/M$ training iterations to make a similar amount of training progress across different batch sizes M. Therefore, we can use the PGNS φ_t to define a concrete expression for $\texttt{EFFICIENCY}_t(M)$ as

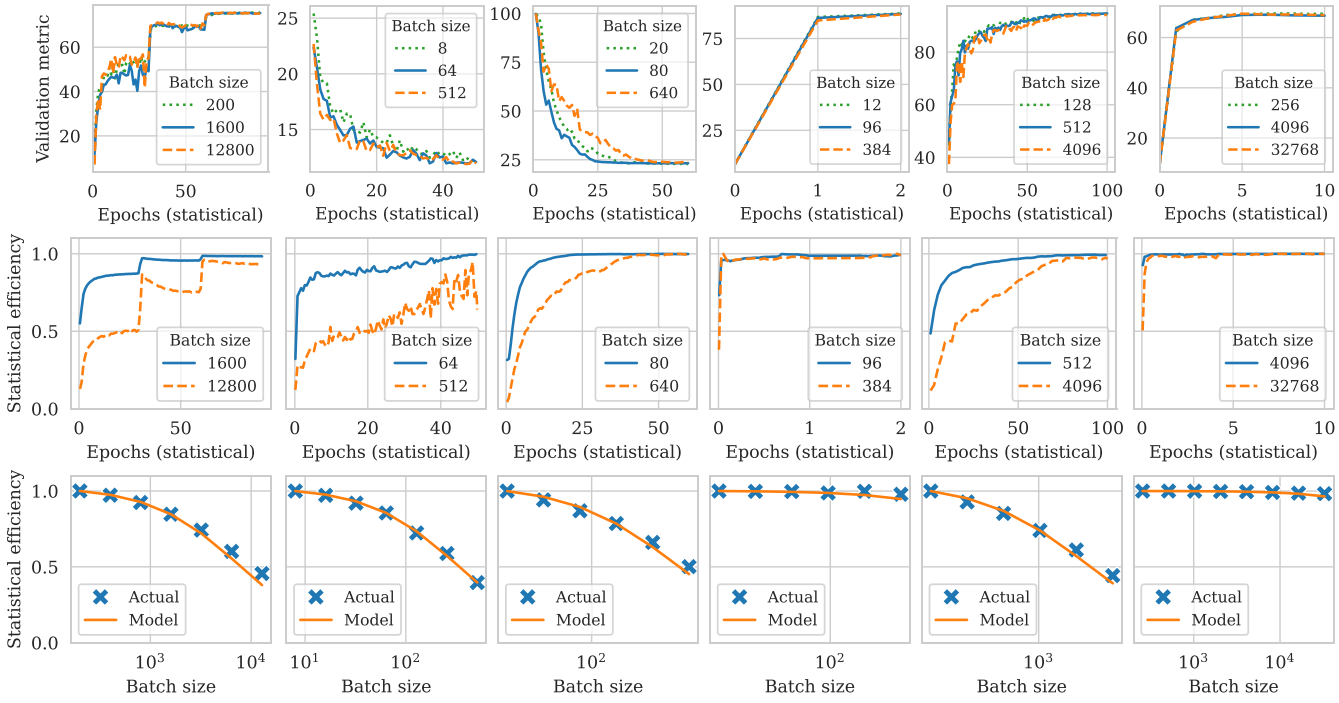
$$\texttt{EFFICIENCY}_t(M) = \frac{\varphi_t + M_0}{\varphi_t + M}. \quad (6)$$

Intuitively, Eqn. 6 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 value of φ_t, then uses Eqn 6 to predict the $\texttt{EFFICIENCY}_t$ at different batch sizes. The measured value of φ_t varies according to the training progress at iteration t, thus $\texttt{EFFICIENCY}_t(M)$ reflects the lifetime-dependent trends exhibited by the true statistical efficiency.

[2] Our notion of goodput for DL is analogous to the traditional definition of goodput in computer networks, ie. the *useful* portion of throughput as benchmarked by training progress per unit of wall-clock time.

[3] Pre-conditioned SGD optimizes $\mathcal{L}(Pw)$ instead of $\mathcal{L}(w)$, where P is known as a pre-conditioning matrix. Adaptive variants of SGD such as Adam and AdaGrad may be viewed as vanilla SGD (with momentum) applied together with a particular pre-conditioning matrix P.



(a) ImageNet (b) YoloV3 (c) DeepSpeech2 (d) BERT (fine-tune) (e) CIFAR10 (f) Recommendation

Figure 2: Statistical efficiency for all models described in Table 1. TOP: validation metric vs training progress for three different batch sizes: M_0, an intermediate batch size, and the max batch size limit we set for each DL task. Metrics are as defined in Table 1 except for YoloV3 for which validation loss is shown. MIDDLE: measured statistical efficiency vs. training progress for two different batch sizes. Training progress (x-axis) in the top two rows is shown in terms of "statistical epochs", defined as $\frac{M}{|X|}\sum_t \texttt{EFFICIENCY}_t(M)$ where $|X|$ is the size of the training dataset. BOTTOM: measured $\texttt{EFFICIENCY}_t$ vs. predicted $\texttt{EFFICIENCY}_t$ for a range of batch sizes (log-scaled), using φ_t measured using the median batch size from each range, during an early-training epoch (roughly 1/8th of the way through training).

Fig. 2 (TOP) shows the validation metrics on a held-out dataset for a variety of DL training tasks (details in Table 1) versus their training progress. "Statistical epochs"[4] is the number of training iterations normalized by $\texttt{EFFICIENCY}_t$ so that each statistical epoch makes theoretically, as projected by our model, the same training progress across different batch sizes. Thus, the degree of similarity between validation curves at different batch sizes is an indicator for the accuracy of $\texttt{EFFICIENCY}_t$ as a predictor of actual training progress.

Although there are differences in the validation curves for several DL tasks (especially in earlier epochs), they achieve similar best values across the different batch sizes we evaluated ($\pm 1\%$ relative difference for all tasks except DeepSpeech2 at $\pm 4\%$). We note that these margins are within the plateau of high-quality models expected from large-batch training [45].

Fig. 2 (MIDDLE and BOTTOM) show the measured and predicted $\texttt{EFFICIENCY}_t$ during training and for a range of different batch sizes. In general, larger batch sizes have lower $\texttt{EFFICIENCY}_t$ early in training, but close the gap later on in training. The exceptions being BERT, which is a fine-tuning task starting from an already pre-trained model, and recommendation, which uses a much smaller and shallower model architecture than the others. How $\texttt{EFFICIENCY}_t$ changes during training varies from task to task, and depends on specific properties like the learning rate schedule. For example, $\texttt{EFFICIENCY}_t$ for ImageNet, which uses step-based learning rate annealing, experiences sharp increases whenever the learning rate is annealed.

Finally, we note that the $\texttt{EFFICIENCY}_t$ function (which is supplied with estimates of φ_t by Pollux) is able to accurately model observed values at a range of different batch sizes. This means that φ_t measured using batch size M can be used by Pollux to predict the value of $\texttt{EFFICIENCY}_t$ at a different batch size M' without needing to train using M' ahead of time.

Upper batch size limit. In some cases, as the batch size increases, the chosen LR scaling rule may break down before the statistical efficiency decreases, which degrades the final model quality. To address these cases, the application may define a maximum batch size limit that will be respected by Pollux. Nevertheless, we find that a batch size up to $32\times$ larger works

[4] Similar to the notion of "scale-invariant iterations" defined in [32].

well in most cases. Furthermore, limits for common models are well-studied for popular LR scaling rules [21, 32, 57, 69]. As better LR scaling rules are developed, they may be incorporated into Pollux using its plug-in interface (§3).

Estimating φ_t. The PGNS φ_t can be estimated in a similar fashion as the GNS by following Appendix A.1 of [46], except using the pre-conditioned gradient Pg instead of the gradient g. This 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 GPU k. However, this method doesn't work when there is only a single GPU (and gradient accumulation is off, i.e. $s = 0$). In this particular situation, Pollux switches to a differenced variance estimator [63] 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}, and then calculate the throughput as

$$\texttt{THROUGHPUT}(a,m,s) = M(a,m,s)/T_{iter}(a,m,s). \quad (7)$$

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. We also start by assuming no gradient accumulation, i.e. $s = 0$.

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

$$T_{grad}(m) = \alpha_{grad} + \beta_{grad} \cdot m, \quad (8)$$

where $\alpha_{grad}, \beta_{grad}$ are fittable parameters.

Modeling T_{sync}. When allocated a single GPU, no synchronization is needed and $T_{sync} = 0$. Otherwise, we model T_{sync} as a linear function of the number of GPUs since in data-parallelism, the amount of data sent and received from each replica is typically only dependent on the size of the gradients and/or parameters. We include a linear factor to account for performance retrogressions associated with using three or more GPUs, such as increasing likelihood of stragglers or network delays.

Co-location of GPUs on the same node reduces network communication, which can improve T_{sync}. Thus, we use different parameters depending on GPU placement. Letting $K = \texttt{SUM}(a)$ be the number of allocated GPUs,

$$T_{sync}(a,m) = \begin{cases} 0 & \text{if } K = 1 \\ \alpha_{sync}^{local} + \beta_{sync}^{local} \cdot (K-2) & \text{if } N = 1, K \geq 2 \\ \alpha_{sync}^{node} + \beta_{sync}^{node} \cdot (K-2) & \text{otherwise,} \end{cases} \quad (9)$$

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 [70]. 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

$$T_{iter}(a,m,0) = \left(T_{grad}(a,m)^\gamma + T_{sync}(a)^\gamma\right)^{1/\gamma}, \quad (10)$$

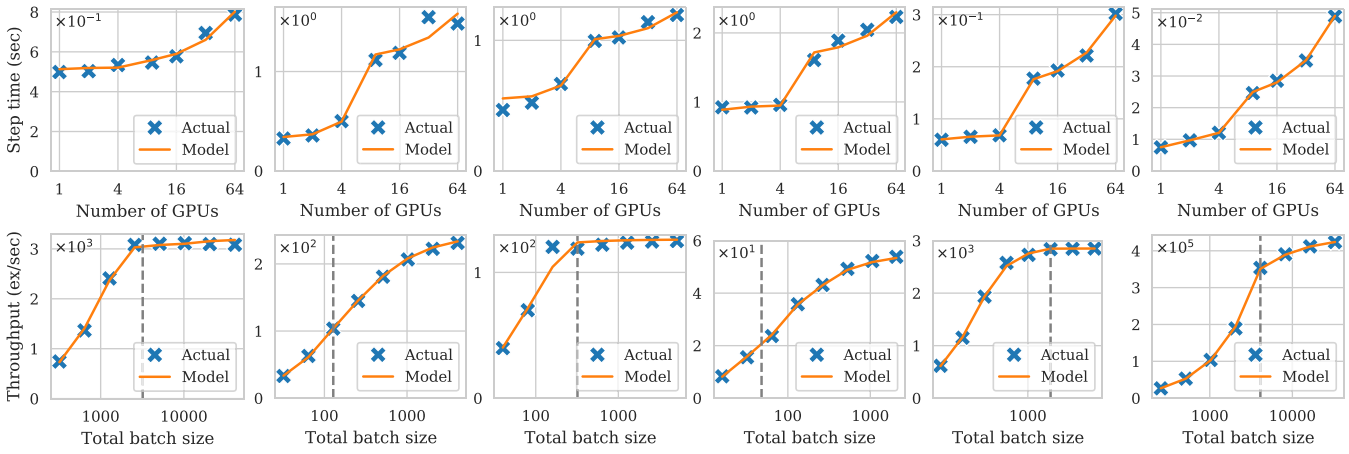
where $\gamma \geq 1$ is a learnable parameter. Eqn. 10 has the property that $T_{iter} = T_{grad} + T_{sync}$ when $\gamma = 1$, and smoothly transitions towards $T_{iter} = \max(T_{grad}, T_{sync})$ as $\gamma \to \infty$.

Gradient Accumulation. In data-parallelism, GPU memory limits the per-GPU batch size, and many DL models hit this limit before the batch size is large enough for T_{grad} to overcome T_{sync} (or experience diminishing statistical efficiency), resulting in suboptimal scalability. Several techniques exist for overcoming the GPU memory limit [9, 10, 27, 30]; we focus on gradient accumulation, which is easily implemented using popular DL frameworks. Per-GPU gradients are aggregated locally over s forward-backward passes before being synchronized across all GPUs during the $(s+1)^{\text{th}}$ pass, achieving a larger total batch size. Thus, one iteration of SGD spans s accumulation steps followed by one synchronization step, modeled as

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

Throughput model validation. Fig. 3 shows an example of our THROUGHPUT function fit to measured throughput values for a range of resource allocations and batch sizes. Each DL task was implemented using PyTorch [51], which overlaps the backward pass' computation and communication. Gradients are synchronized with NCCL 2.7.8, which uses either ring all-reduce or tree all-reduce depending on the detected GPUs and their placements and its own internal performance estimates. Overall, 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 particular, all models we measured except ImageNet exhibited high sensitivity to inter-node synchronization, indicating that they benefit from co-location of GPUs. Furthermore, YOLOv3 and BERT benefit from using gradient accumulation to increase their total batch sizes. These detailed characteristics are well-represented by our THROUGHPUT function, and can be optimized for by Pollux.

In addition to the configurations in Fig. 3, we fitted the THROUGHPUT function on a diverse set of GPU placements and batch sizes in a 64-GPU cluster. Across all DL tasks, the average error of the fitted model was at most 10%, indicating that it represents the observed throughput measurements well.



(a) ImageNet (b) YoloV3 (c) DeepSpeech2 (d) BERT (fine-tune) (e) CIFAR10 (f) Recommendation

Figure 3: System throughput for all models described in Table 1, as measured using g4dn.12xlarge instances in AWS each with 4 NVIDIA T4 GPUs and created within the same placement group. Eqn. 11 was fitted using the observed data that appeared in each plot. TOP: time per training iteration vs. the number of allocated GPUs (log-scaled), with the per-GPU batch size held constant. The GPUs are placed in as few 4-GPU nodes as possible, which causes a sharp increase beyond 4 GPUs (when inter-node network synchronization becomes required). BOTTOM: system throughput (examples per second) vs. total batch size (log-scaled), with the number of GPUs held constant. To the left of the vertical dashed line, the entire mini-batch fits within GPU memory. To the right, the total batch size is achieved using gradient accumulation.

Limits of the throughput model. Pollux models data-parallel training throughput only in the dimensions it cares about, i.e. number and co-locality of GPUs, batch size, and gradient accumulation steps. The simple linear assumptions made in Eqn. 11, although sufficiently accurate for the settings we tested, may diverge from reality for specialized hardware [33], sophisticated synchronization algorithms [7, 65, 72], different parallelization strategies [28, 47, 58, 59], at larger scales [6, 68], or hidden resource contention not related to network used for gradient synchronization. Rather than attempting to cover all scenarios with a single throughput model, we designed $\texttt{GOODPUT}_t$ (Eqn. 4) to be modular so that different equations for `THROUGHPUT` may be easily plugged in without interfering with the core functionalities provided by Pollux.

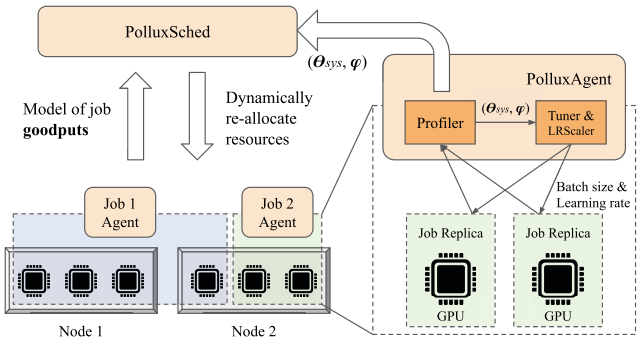


Figure 4: Co-adaptive scheduling architecture of Pollux.

4 Pollux Design and Architecture

Pollux adapts DL job execution 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 combined with cluster-level goals including fairness and job-completion time. To achieve this co-adaptivity in a scalable way, Pollux's design consists of two primary components, as illustrated in Fig. 4.

First, a *PolluxAgent* runs together with each job. It fits the $\texttt{EFFICIENCY}_t$ and `THROUGHPUT` functions 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 goodput function for each job and cluster-wide resource contention. Scheduling decisions made by PolluxSched also account for the overhead associated with resource re-allocations, slowdowns due to network interference between multiple jobs, and resource fairness.

PolluxAgent and PolluxSched *co-adapt* to each other. While PolluxAgent adapts each training job to make efficient use of its allocated resources, PolluxSched dynamically re-allocates each job's resources, taking into account the PolluxAgent's ability to tune its job.

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 it 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 the appropriate plug-in LR scaling rule (e.g. AdaScale for SGD or square-root scaling for Adam).

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

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

which are required to construct the THROUGHPUT function. Together with the PGNS φ_t (for predicting $\texttt{EFFICIENCY}_t$) 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 §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 tuple (a, m, s, T_{iter}) for all combinations of resource allocations a, per-GPU batch size m, and gradient accumulation steps s 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 [73]. 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. 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 (§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 that can be allocated to at most twice the maximum number of GPUs the job has been allocated in its lifetime.

Although other principled approaches to exploration can be applied (e.g., Bayesian optimization), we find that this simple prior-driven strategy is sufficient in our experiments. Sec. 5.3.2 shows that prior-driven exploration performs close (within 2-5%) to an idealized scenario in which the model is fitted offline for each job before being submitted to the cluster.

Training job tuning. With θ_{sys}, φ_t, and M_0, which fully specify the DL job's GOODPUT function at its current training progress, PolluxAgent determines the most efficient per-GPU batch size and gradient accumulation steps,

$$(m^*, s^*) = \underset{m,s}{\text{argmax}}\, \texttt{GOODPUT}(a, m, s), \tag{13}$$

where a is the job's current resource allocation.

Once a new configuration is found, the job will use it for its subsequent training iterations, using the plug-in LR scaling rule to adapt its learning rate appropriately. As the job's $\texttt{EFFICIENCY}_t$ function changes over time, PolluxAgent will periodically re-evaluate the most efficient configuration.

4.2 PolluxSched: Cluster-wide Optimization

The PolluxSched periodically allocates (and re-allocates) resources for every job in the cluster. To determine a set of efficient cluster-wide resource allocations, it maximizes a *fitness function* that is defined as a generalized (power) mean across speedups for each job:

$$\texttt{FITNESS}_p(A) = \left(\frac{1}{J}\sum_{j=1}^{J} \texttt{SPEEDUP}_j(A_j)^p\right)^{1/p}. \tag{14}$$

A is an *allocation matrix* with each row A_j being the allocation vector for a job j, thus A_{jn} is the number of GPUs on node n allocated to job j, and J is the total number of running and pending jobs sharing the cluster. We define the speedup of each job as the factor of goodput improvement using a given resource allocation over using a fair-resource allocation, ie.

$$\texttt{SPEEDUP}_j(A_j) = \frac{\max_{m,s} \texttt{GOODPUT}_j(A_j, m, s)}{\max_{m,s} \texttt{GOODPUT}_j(a_f, m, s)}, \tag{15}$$

where $\texttt{GOODPUT}_j$ is the goodput of job j at its current training iteration, and a_f is a fair resource allocation for the job, defined to be an exclusive $1/J$ share of the cluster.[5]

In §3, we described how the GOODPUT function can be fitted to observed metrics during training and then be evaluated as a predictive model. PolluxSched leverages this ability to predict GOODPUT to maximize FITNESS via a search procedure, and then it applies the outputted allocations to the cluster.

Fairness and the effect of p. When $p = 1$, $\texttt{FITNESS}_p$ is the average of SPEEDUP values across all jobs. This causes PolluxSched to allocate more GPUs to jobs that achieve a high SPEEDUP when provided with many GPUs (i.e., jobs that scale

[5] We note that SPEEDUP has similarities with *finish-time fairness* [43]. But, SPEEDUP is related to training performance at a moment in time, whereas finish-time fairness is related to end-to-end job completion time.

well). However, as $p \to -\infty$, $\texttt{FITNESS}_p$ smoothly approaches the minimum of SPEEDUP values, in which case maximizing $\texttt{FITNESS}_p$ promotes equal SPEEDUP between training jobs, but ignores the overall cluster goodput and resource efficiency.

Thus, p can be considered a "fairness knob", with larger negative values being more fair. A cluster operator may select a suitable value, based on organizational priorities. In our experience and results in §5, we find that $p = -1$ achieves most goodput improvements and reasonable fairness.

Re-allocation penalty. Each time a job is re-allocated to a different set of GPUs, it incurs some delay to re-configure the training process. Using the the popular checkpoint-restart method, we measured between 15 and 120 seconds of delay depending on the size of the model being trained and other initialization tasks in the training code. 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 be re-allocated,

$$\texttt{SPEEDUP}_j(A_j) \longleftarrow \texttt{SPEEDUP}_j(A_j) \times \texttt{REALLOC\_FACTOR}_j(\delta).$$

We define $\texttt{REALLOC\_FACTOR}_j(\delta) = (T_j - R_j\delta)/(T_j + \delta)$, where T_j is the age of the training job, R_j is the number of re-allocations incurred by the job so far, and δ is an estimate of the re-allocation delay. Intuitively, $\texttt{REALLOC\_FACTOR}_j(\delta)$ scales $\texttt{SPEEDUP}_j(A_j)$ according to the assumption that the historical average rate of re-allocations for job j will continue indefinitely into the future. Thus, a job that has historically experienced a higher rate of re-allocations will be penalized more for future re-allocations.

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 [31]; Xiao et al. [66] 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.

Interference avoidance is implemented as a constraint in Pollux's search algorithm, by ensuring at most one distributed job is allocated to each node. We study the effects of interference avoidance in §5.3.2.

Supporting non-adaptive jobs. In certain cases, a user may want to run a job with a fixed batch size, i.e. $M = M_0$. These jobs are well-supported by PolluxSched, which simply fixes $\texttt{EFFICIENCY}_t$ for that job to 1 and can continue to adapt its resource allocations based solely on its system throughput.

4.3 Implementation

PolluxAgent is implemented as a Python library that is imported into DL training code. We integrated PolluxAgent with PyTorch [51], which uses all-reduce as its gradient synchronization algorithm. PolluxAgent inserts performance profiling code that 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. 10) 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 per-GPU batch size and gradient accumulation steps, by optimizing its now up-to-date goodput function (Eqn. 4) with its currently allocated resources.

PolluxSched is implemented as a service in Kubernetes [2]. At a fixed time interval, PolluxSched runs its search algorithm, and then applies the resultant allocation matrix by creating and terminating Kubernetes Pods that run the job workers. To find a good allocation matrix, PolluxSched uses a population-based search algorithm that perturbs and combines candidate allocation matrices to produce higher-value allocation matrices, and finally modifies them to satisfy node resource constraints and interference avoidance. The allocation matrix with the highest fitness score is applied to the jobs running in the cluster.

Both PolluxAgent and PolluxSched require a sub-procedure that optimizes $\texttt{GOODPUT}_t(a, m, s)$ given a fixed a (Eqn. 13). We implemented this procedure by first sampling a range of candidate values for the total batch size M, then finding the smallest s such that $m = \lceil M/s \rceil$ fits into GPU memory according to a user-defined upper-bound, and finally taking the configuration which results in the highest GOODPUT value.

5 Evaluation

We compare Pollux with two state-of-the-art DL schedulers using a testbed cluster with 64 GPUs. Although one primary advantage of Pollux is automatically selecting the configurations for each job, we find that Pollux still reduces average job completion times by 37–50% even when the baseline schedulers are supplied with well-tuned job configurations (a scenario that strongly favors the baseline schedulers). Pollux is able to dynamically adapt each job by trading-off between high-throughput/low-efficiency and low-throughput/high-efficiency modes of training, depending on the current cluster state and training progress.

Using a cluster simulator, we evaluate the impact of specific settings on Pollux, including the total workload intensity, prior-driven exploration, scheduling interval, and interference avoidance. With its fairness knob, Pollux can improve finish-time fairness [43] by 1.5–5.4× compared to baseline DL schedulers. We also reveal a new opportunity for auto-scaling in the cloud by showing that a Pollux-based auto-scaler can potentially reduce the cost of training large models (e.g. ImageNet) by 25%.

5.1 Experimental Setup

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 NVIDIA 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.

Synthetic Workload Construction. We randomly sampled 160 jobs from the busiest 8-hour range (hours 3–10) in the deep learning cluster traces published by Microsoft [31]. Each job in the orginal trace 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: Small (0–1 GPU-hours), Medium (1–10 GPU-hours), Large (10–100 GPU-hours), and XLarge (100–1000 GPU-hours). For each job in the trace, we picked a training job from Table 1 that is in the same category.

Manually-tuned jobs for baseline DL schedulers. 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 §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 (i.e., perfectly linear) scalability versus using the optimal batch size on a single GPU. For each job submitted from our synthetic workload, we selected its number of GPUs and batch size randomly from its set of valid configurations.

Our job configurations assume that the users are highly rational and knowledgeable about the scalability of the models they are training. Less than 50% of the ideal scalability would lead to under-utilization of resources, and more than 80% of the ideal scalability means the job can still utilize more GPUs efficiently. We emphasize that this assumption of uniformly sophisticated users is unrealistically biased in favor of the baseline schedulers and only serves for comparing Pollux with the ideal performance of baseline systems.

Comparison of DL schedulers. We compare Pollux to two recent deep learning schedulers, Tiresias [22] and Optimus [52], as described in §2.3. Whereas Pollux dynamically co-adapts the number of GPUs and batch sizes of DL training jobs, Optimus only adapts the number of GPUs, and Tiresias adapts neither. To establish a fair baseline for comparison, for all three schedulers, we scale the learning rate using AdaScale for SGD, and the square-root scaling rule for Adam and AdamW.

Pollux. We configured PolluxSched to use a 60s scheduling interval, and compute $\texttt{REALLOC\_FACTOR}(\delta)$ using $\delta = 30$s. PolluxAgent reports its most up-to-date system throughput parameters and gradient statistics every 30s. Unless otherwise specified, the default fairness knob value of $p = -1$ is used.

Tiresias. We configured Tiresias as described in the testbed experiments of Gu et al. [22], with two priority queues and the `PromoteKnob` disabled. We manually tuned the queue threshold to perform well for our synthetic workload. Whenever possible, we placed jobs onto as few different nodes as possible to promote worker locality.

Optimus+Oracle. Optimus leverages a throughput prediction model that is specific to jobs using the parameter server architecture. To account for differences due to the performance model, our implementation of Optimus uses our own throughput model as described in §3.2. Furthermore, Optimus predicts the number of training iterations until convergence by fitting a simple function to the model's convergence curve. Since this method does not work consistently for all models in our synthetic workload, we run each job ahead of time and provide Optimus with the exact number of iterations until completion. We call this version of Optimus *Optimus+Oracle*.

For each job, Tiresias uses the number of GPUs and batch size specified in our synthetic workload. Optimus+Oracle uses the batch size specified, but determines the number of GPUs dynamically. Each job uses gradient accumulation if they are allocated too few GPUs to support the specified batch size.

5.2 Testbed Macrobenchmark Experiments

Table 2 summarizes the results of our testbed experiments for seven configurations: Pollux compared with, first, baseline schedulers using well-tuned job configurations; second, baseline schedulers using more realistic job configurations; third, Pollux using two alternate values for its fairness knob.

Comparisons using well-tuned job configurations. Even when Optimus+Oracle and Tiresias are given well-tuned job configurations as described in §5.1, they are still significantly behind Pollux. In this setting, Pollux (with $p = -1$) achieved 50% and 37% shorter average JCT, 27% and 27% shorter tail (99th percentile) JCT, and 20% and 33% shorter makespan, in comparison to Optimus+Oracle+TunedJobs and Tiresias+TunedJobs, respectively. As we previously noted, this setting highly favors the baseline schedulers, essentially mimicking users who possess expert knowledge about system throughput, statistical efficiency, and how their values change with respect to resource allocations and batch sizes.

One key source of improvement for Pollux is its ability to trade-off between high-throughput/low-efficiency and low-throughput/high-efficiency modes during training. Fig. 5 shows the total number of allocated GPUs and average $\texttt{EFFICIENCY}_t$ during the execution of our synthetic workload. During periods of low cluster contention, Pollux can allocate more GPUs (indicated by **(A)**) and use larger batch sizes to boost training throughput, even at the cost of lower statistical efficiency, because doing so results in an overall higher goodput. On the other hand, during periods of high cluster contention, Pollux may instead use smaller batch sizes to increase statistical efficiency (indicated by **(B)**).

Comparisons using realistic job configurations. Without assistance from a system like Pollux, users are likely to try various numbers of GPUs and batch sizes, before finding a configuration that is efficient. Other users may not invest time

Task	Dataset	Model	Optimizer	LR Scaler	M_0	Validation	Size	Frac. Jobs
Image Classification	ImageNet [12]	ResNet-50 [24]	SGD	AdaScale	200 imgs	75% top1 acc.	XL	2%
Object Detection	PASCAL-VOC [16]	YOLOv3 [55]	SGD	AdaScale	8 imgs	84% mAP	L	6%
Speech Recognition	CMU-ARCTIC [38]	DeepSpeech2 [3]	SGD	AdaScale	20 seqs	25% word err.	M	10%
Question Answering	SQuAD [54]	BERT (finetune) [14]	AdamW	Square-Root	12 seqs	88% F1 score	M	10%
Image Classification	Cifar10 [39]	ResNet18 [24]	SGD	AdaScale	128 imgs	94% top1 acc.	S	36%
Recommendation	MovieLens [23]	NeuMF [25]	Adam	Square-Root	256 pairs	69% hit rate	S	36%

Table 1: Models and datasets used in our evaluation workload. Each training task achieves the provided validation metrics. The fraction of jobs from each category are chosen according to the public Microsoft cluster traces.

Policy	Job Completion Time		Makespan
	Average	99%tile	
Pollux ($p=-1$)	**0.76h**	11h	**16h**
Optimus+Oracle+TunedJobs	1.5h	15h	20h
Tiresias+TunedJobs	1.2h	15h	24h
Optimus+Oracle	2.7h	22h	28h
Tiresias	2.8h	25h	31h
Pollux ($p=+1$)	0.83h	**10h**	**16h**
Pollux ($p=-10$)	0.84h	12h	18h

Table 2: Summary of testbed experiments.

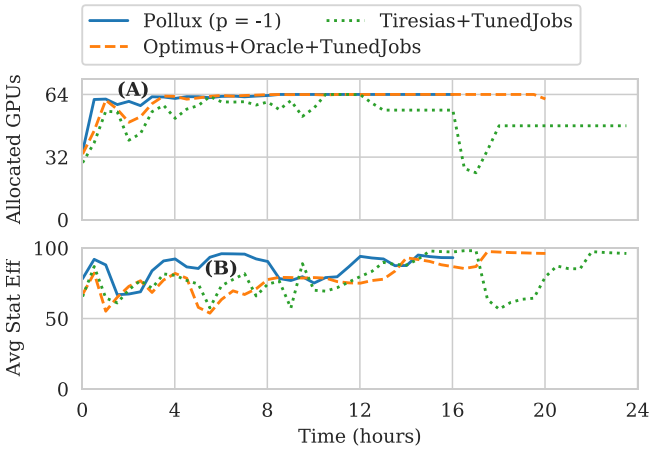


Figure 5: Comparison between Pollux ($p=-1$), Optimus, and Tiresias while executing our synthetic workload (with tuned jobs). TOP: average cluster-wide allocated GPUs over time. BOTTOM: average cluster-wide statistical efficiency over time. Tiresias+TunedJobs dips between hours 16 and 20 due to a 24-GPU job blocking a 48-GPU job from running.

into configuring their jobs well in the first place.

To set a more realistically configured baseline, we ran Optimus+Oracle and Tiresias on a version of our synthetic workload with the number of GPUs exactly as specified in the Microsoft cluster trace. The batch size was chosen to be the baseline batch size M_0 times the number of GPUs, which is how we expect most users to initially configure their distributed training jobs. We find that these jobs typically use fewer GPUs and smaller batch sizes than their well-configured counterparts.

Using this workload, we find that Pollux has 72% and 73% shorter average JCT, 50% and 56% shorter tail JCT, and 43% and 48% shorter makespan, in comparison to Optimus+Oracle and Tiresias, respectively. Even though Optimus+Oracle can dynamically increase the GPU allocation of each job, it still only slightly outperforms Tiresias because it does not also increase the batch size to better utilize those additional GPUs.

A closer look at co-adapted job configurations. Fig. 6 (LEFT) shows the configurations chosen by Pollux for one ImageNet training job as the synthetic workload progresses. **(A)** during the initial period of low cluster contention, more GPUs are allocated to ImageNet, causing a larger batch size to be used and lowering statistical efficiency. **(B)** during the subsequent period of high cluster contention, fewer GPUs are allocated to ImageNet, causing a smaller batch size to be used and raising statistical efficiency. **(C)** when the cluster contention comes back down, ImageNet continues to be allocated more GPUs and uses a larger batch size. However, we note that the batch size per GPU is much higher than in the first low-contention period, since the job is now in its final, high-statistical-efficiency phase of training. We see similar trade-offs being made over time for two YOLOv3 jobs (RIGHT).

Effect of the fairness knob. We ran Pollux using three values of the fairness knob, $p = 1, -1, -10$. Compared with no fairness ($p = 1$), introducing a moderate degree of fairness ($p = -1$) improved the average job completion time (JCT) but degraded the tail JCT. This is because[6], in our synthetic workload, the tail JCT comprises of long but scalable jobs (i.e. ImageNet), which take a large number of GPUs away from other jobs in the absence of fairness ($p=1$). However, further increasing fairness ($p=-10$) degraded performance in average JCT, tail JCT, and makespan. In §5.3.1, we present a more detailed analysis of the impact of p on scheduling fairness.

System overheads. During each 60s scheduling interval, PolluxSched spent an average of 1 second on 1 vCPU computing the cluster allocations by optimizing the $\texttt{FITNESS}_p$ function. On average, each job was re-allocated resources once every 7 minutes, resulting in an average 8% run-time overhead due to checkpoint-restarts. Each PolluxAgent fits its throughput model parameters on its latest observed metrics every 30 seconds, taking an average of 0.2 seconds each time. Finding the

[6]We note that $p = -1$ (harmonic mean over speedups) may be more suitable than $p=1$ (arithmetic mean) when optimizing for the average JCT.

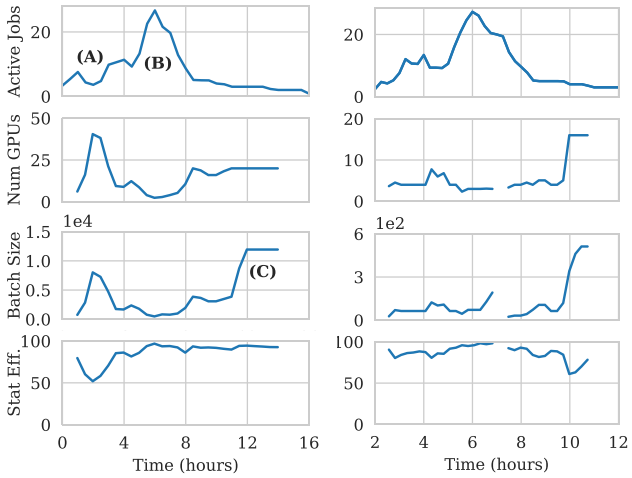


Figure 6: Co-adaptation over time of one ImageNet job (LEFT) and two YOLOv3 jobs (RIGHT) using Pollux ($p=-1$). ROW 1: number of jobs actively sharing the cluster. ROW 2: number of GPUs allocated to the job. ROW 3: batch size (images) used. ROW 4: statistical efficiency (%).

optimal per-GPU batch size and gradient accumulation steps by optimizing $\texttt{GOODPUT}_t$ takes an average of 0.4 milliseconds.

5.3 Simulator Experiments

We built a discrete-time 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 on 8 different workload traces generated using the same duration, number of jobs, and job size distributions as in §5.2, and we report the average results across all 8 traces.

Simulator construction. For each job in Table 1, we measured the time per training iteration for 146 different GPU allocations+placements in our testbed cluster of 16 nodes and 64 total GPUs. For each allocation, we measured a range of batch sizes up to the GPU memory limit. To simulate the throughput for a job, we queried a multi-dimensional linear interpolation on the configurations we measured. For each model, we also measured the (pre-conditioned) gradient noise scale during training using a range of batch sizes, and across every epoch. To simulate the statistical efficiency for a job using a certain batch size, we linearly interpolated its value of the PGNS 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 that has its resources re-allocated. Unless stated otherwise, we do not simulate any network interference between different jobs. We study the effects of interference in more detail in §5.3.2.

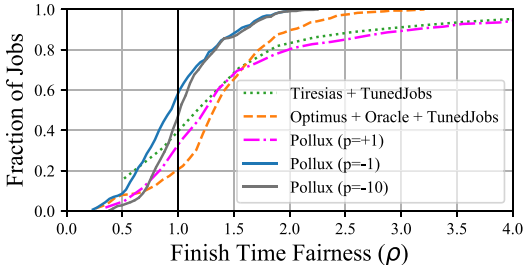


Figure 7: CDF of Finish Time Fairness (ρ).

Compared with our testbed experiments in §5.2, we find that our simulator obtains similar factors of improvement, showing that Pollux reduces the average JCT by 48% and 32% over Optimus+Oracle+TunedJobs and Tiresias+TunedJobs.

5.3.1 Scheduling Fairness

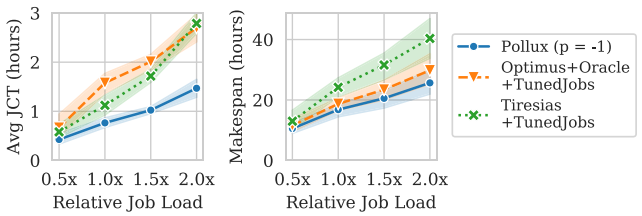
We evaluate the scheduling fairness of Pollux using *finish-time fairness* [43] (denoted by ρ), which is defined to be the ratio of a job's JCT running on shared resources to that of the job running in an isolated and equally-partitioned cluster. Under this metric, jobs with $\rho < 1$ have been treated better-than-fair by the cluster scheduler, while jobs with $\rho > 1$ have been treated worse-than-fair.

In Fig. 7, we compare the finish-time fairness of Pollux with Optimus+Oracle+TunedJobs and Tiresias+TunedJobs. Pollux with $p = 1$ results in poor fairness, similar to Tiresias+TunedJobs, which is apparent as a long tail of jobs with $\rho > 4$. Optimus+Oracle+TunedJobs obtains better fairness due to its allocation algorithm which attempts to equalize the JCT improvement for each job. Pollux with $p = -1$ provides the best fairness, with 99% of jobs achieving $\rho < 2$, and does so while still providing significant performance increases (Table 2). For $p = -10$, we observe slightly worse fairness overall, caused by PolluxSched incurring a larger number of re-allocations due to ignoring the cost in favor of equalizing speedups at all times.

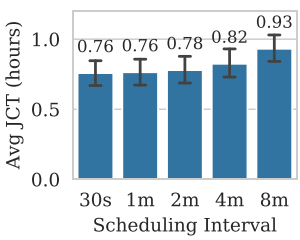
To provide context, we note that the curves for Tiresias and Optimus are consistent with those reported (for different workloads) by Mahajan et al. [43]. Although their Themis system is not available for direct comparison, the ρ range for Pollux with $p = -1$ is similar to the range reported for Themis. The max-ρ improvements (1.5× and 5.4×) over Tiresias and Optimus are also similar.

5.3.2 Other Effects on Scheduling

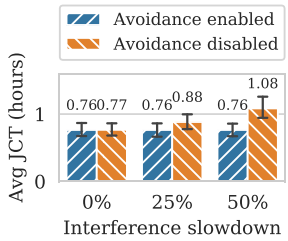
Sensitivity to job load. We compare the performance of Pollux, Optimus+Oracle+TunedJobs, and Tiresias+TunedJobs



(a) Varying the workload intensity.



(b) Varying scheduling interval.



(c) Varying job interference.

Figure 8: Effects of various parameters on Pollux, error bars and bands represent 95% confidence intervals.

for increasing workload intensity in terms of rate of job submissions. Fig. 8a shows the results. As expected, all three scheduling policies suffer longer average JCT and makespan as the load is increased. Across all job loads, Pollux maintains similar relative improvements over the baseline schedulers.

Impact of prior-driven exploration. Pollux explores GPU allocations for each DL job from scratch during training (Sec. 4.1). We evaluated the potential improvement from more efficient exploration by seeding each job's throughput models using historical data collected offline. We observed minor (2–5%) reduction in JCT for short jobs like CIFAR10, but no significant change for longer running jobs, indicating low overhead from Pollux's prior-driven exploration.

Impact of scheduling interval. We ran Pollux using a range of values for its scheduling interval, as shown in Fig. 8b. We find that Pollux performs similarly well in terms of average JCT for intervals up to 2 minutes, while longer intervals result in performance degradation. Since newly-submitted jobs can only start during the next scheduling interval, we would expect an increase in the average queuing time due to longer scheduling intervals. However, we find that queuing contributed to roughly half of the performance degradation observed, indicating that Pollux still benefits from a relatively frequent adjustment of resource allocations.

Impact of interference avoidance. 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. 8c shows the results. With interference avoidance enabled, the average JCT is unaffected by even severe slowdowns, because network contention is completely mitigated. However, without interference avoidance, the average JCT is 1.4× longer when the interference slowdown is 50%. On the other hand, in the ideal scenario when there is zero slowdown due to interference, PolluxSched performs similarly whether or not interference avoidance is enabled. This indicates that PolluxSched is still able to find efficient cluster allocations while obeying the interference avoidance constraint.

5.4 More Applications of Pollux

5.4.1 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. Goodput-driven scheduling presents a unique opportunity: when a DL model's statistical efficiency increases during training, it may be more cost-effective to provision more cloud resources and use larger batch sizes during the later epochs of a large training job, rather than earlier on. We present some preliminary evidence using our cluster simulator, and note that a full design of an auto-scaling system based on goodput may be the subject of future work.

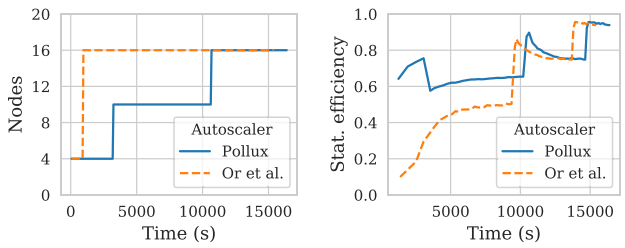
Auto-scaling ImageNet training. We implemented a simple auto-scaling policy using Pollux's goodput function. During training, we scaled up the number of nodes whenever $\max_{m,s} \texttt{GOODPUT}_t(a,m,s)/\texttt{SUM}(a) > U \cdot \max_{m,s} \texttt{GOODPUT}_t(1,m,s)$, i.e. the goodput exceeds some fraction U of the predicted ideal goodput assuming perfect scalability. We set $U = 2/3$, and increased to a number of nodes such that the predicted goodput is approximately $L = 1/2$ of the predicted ideal goodput.

Fig. 9 compares our Pollux-based auto-scaler with the auto-scaler proposed by Or *et al.* [50], which allows the batch size to be increased during training, but models job performance using the system throughput rather than the goodput. Since the system throughput does not change with training progress, throughput-based autoscaling (Or *et al.*) quickly scales out to more nodes and a larger batch size (Fig. 9a), which remains constant thereafter. On the other hand, Pollux starts with a small number of nodes, and gradually increases the number of nodes as the effectiveness of larger batch sizes improves over time. Fig. 9b shows that Pollux maintains a high statistical efficiency throughout training. Overall, compared to Or *et al.*'s throughput-based auto-scaling, Pollux trains ImageNet with 25% cheaper cost, with only a 6% longer completion time.

5.4.2 Hyper-parameter Optimization (HPO)

Hyper-parameter optimization (HPO) is an important DL workload. In HPO, the user defines a *search space* over relevant model hyper-parameters. A HPO algorithm (aka a trial scheduler) submits many training jobs (trials) to evaluate the effectiveness of particular hyper-parameters, in terms of objectives such as model accuracy or energy efficiency.

Different HPO algorithm types manage trials differently. For example, Bayesian optimization algorithms [37, 62] may submit a few training jobs at a time, and determine future trials based on the fully-trained results of previous trials.



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

Figure 9: Goodput-based auto-scaling (Pollux) vs throughput-based auto-scaling (Or et al.) for ImageNet training.

Policy	Accuracy (Top 5 trials)	Avg JCT	Makespan
Pollux	95.4±0.2	25min	10h
Baseline	95.5±0.3	34min	14h

Table 3: Summary of HPO experiments.

Bandit-based algorithms [41] may launch a large number of trials at once and early-stop ones that appear unpromising.

A full evaluation on how Pollux affects different HPO algorithm types is future work. Table 3 shows results from tuning a ResNet18 model trained on the CIFAR10 dataset, using a popular Bayesian optimization-based HPO algorithm known as the Tree-structured Parzen Estimator (TPE) [5]. The search space covers the learning rate and annealing, momentum, weight decay, and network width hyper-parameters. We configured TPE so that 4 trials run concurrently with each other, and 100 trials are run in total. The testbed consists of two NVIDIA DGX A100 nodes, each with 8 A100 GPUs. The baseline scheduler assigns a static allocation of 4 GPUs (all on the same node) to each trial and uses a fixed per-GPU batch size for every trial. As expected, similar accuracy values are achieved, but Pollux completes HPO 30% faster due to adaptive (re-)allocation of resources as trials progress and adaptive batch sizes.

5.5 Artifact

We provide an artifact containing the full implementation of Pollux, benchmark model implementations (Table 1), testbed experiment scripts (Sec. 5.2), cluster simulator implementation and results (Sec. 5.3), available at `https://github.com/petuum/adaptdl/tree/osdi21-artifact`. The raw testbed experiment (Sec. 5.2) logs and analysis scripts are provided at `https://github.com/petuum/pollux-results`.

6 Additional Related Work

Prior DL schedulers are discussed in §2.3.

Adaptive batch size training. Recent work on DL training algorithms have explored dynamically adapting batch sizes for better efficiency and parallelization. AdaBatch [13] increases the batch size at pre-determined iterations during training, while linearly scaling the learning rate. Smith et al. [61] 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. Alternatively, anytime minibatch [17] adapts the batch size to mitigate stragglers in distributed training.

KungFu [44] supports adaptive training algorithms, including adaptive batch sizes, by allowing applications to define custom adaptation policies and enabling efficient adaptation and monitoring during training. Although KungFu is directed at single-job training and Pollux at cluster scheduling, we believe KungFu offers useful tools which can be used to implement the adaptive policies used by the PolluxAgent.

Hyper-parameter tuning. A large body of work focuses on tuning the hyper-parameters for ML and DL models [5, 18, 29, 34, 49], which typically involves many training jobs [1, 20] as discussed earlier. Although batch size and learning rate are within the space of hyper-parameters often optimized by these systems, Pollux's goal is fundamentally different. Whereas HPO algorithms 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.

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 and is more fair than recent DL schedulers, even if users can configure their jobs well, and provides even bigger benefits with more realistic user knowledge.

8 Acknowledgements

We thank our shepherd, Michael Isard, and the anonymous OSDI reviewers for their insightful comments and suggestions that improved our work. We also thank our colleagues from Petuum — Omkar Pangarkar, Richard Fan, Peng Wu, Jayesh Gada, and Vishnu Vardhan — for their invaluable contributions toward the open source implementation of Pollux.

References

[1] Introduction to katib. 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] 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.

[7] J. Canny and Huasha Zhao. Butterfly mixing: Accelerating incremental-update algorithms on clusters. In *SDM*, 2013.

[8] 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.

[9] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[10] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[11] 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.

[12] 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.

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

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011.

[16] 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.

[17] Nuwan Ferdinand, Haider Al-Lawati, Stark Draper, and Matthew Nokleby. ANYTIME MINIBATCH: EXPLOITING STRAGGLERS IN ONLINE DISTRIBUTED OPTIMIZATION. In *International Conference on Learning Representations*, 2019.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1341–1355, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[29] 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.

[30] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020.

[31] 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.

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

[33] N. Jouppi, C. Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, R. Bajwa, Sarah Bates, Suresh Bhatia, N. Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, M. Dau, J. Dean, Ben Gelb, T. Ghaemmaghami, R. Gottipati, William Gulland, R. Hagmann, C. Ho, Doug Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, N. Kumar, Steve Lacy, J. Laudon, James Law, Diemthu Le, Chris Leary, Z. Liu, Kyle A. Lucke, Alan Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, Ravi Narayanaswami, Ray Ni, K. Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, A. Phelps, J. Ross, Matt Ross, Amir Salek, E. Samadiani, C. Severn,

G. Sizikov, Matthew Snelham, J. Souter, D. Steinberg, Andy Swing, Mercedes Tan, G. Thorson, Bo Tian, H. Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, W. Wang, Eric Wilcox, and D. Yoon. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[34] 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.

[35] 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. *CoRR*, abs/1609.04836, 2016.

[36] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[37] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR, 2017.

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

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

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

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[42] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[43] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 289–304, 2020.

[44] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 937–954. USENIX Association, November 2020.

[45] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018.

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

[47] 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.

[48] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

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

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

[57] 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.

[58] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-tensorflow: Deep learning for supercomputers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[59] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019.

[60] S. L. Smith and Quoc V. Le. A bayesian perspective on generalization and stochastic gradient descent. *ArXiv*, abs/1710.06451, 2018.

[61] 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.

[62] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25:2951–2959, 2012.

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

[64] Rachel Ward, Xiaoxia Wu, and Leon Bottou. Adagrad stepsizes: Sharp convergence over nonconvex landscapes. In *International Conference on Machine Learning*, pages 6677–6686. PMLR, 2019.

[65] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394, 2015.

[66] 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.

[67] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[68] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds, 2019.

[69] 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.

[70] 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.

[71] 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.

[72] Huasha Zhao and J. Canny. Kylix: A sparse allreduce for commodity clusters. *2014 43rd International Conference on Parallel Processing*, pages 273–282, 2014.

[73] 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.