KungFu: Making Training in Distributed Machine Learning Adaptive

Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis,
Andrei-Octavian Brabete, and Peter Pietzuch, *Imperial College London*

https://www.usenix.org/conference/osdi20/presentation/mai

This paper is included in the Proceedings of the
14th USENIX Symposium on Operating Systems
Design and Implementation

November 4–6, 2020

978-1-939133-19-9

KungFu: Making Training in Distributed Machine Learning Adaptive

Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, Peter Pietzuch
Imperial College London

Abstract

When using distributed machine learning (ML) systems to train models on a cluster of worker machines, users must configure a large number of parameters: hyper-parameters (e.g. the batch size and the learning rate) affect model convergence; system parameters (e.g. the number of workers and their communication topology) impact training performance. In current systems, adapting such parameters during training is ill-supported. Users must set system parameters at deployment time, and provide fixed adaptation schedules for hyper-parameters in the training program.

We describe *KungFu*, a distributed ML library for TensorFlow that is designed to enable adaptive training. KungFu allows users to express high-level *Adaptation Policies* (APs) that describe how to change hyper- and system parameters during training. APs take real-time monitored metrics (e.g. signal-to-noise ratios and noise scale) as input and trigger control actions (e.g. cluster rescaling or synchronisation strategy updates). For execution, APs are translated into monitoring and control operators, which are embedded in the dataflow graph. APs exploit an efficient asynchronous collective communication layer, which ensures concurrency and consistency of monitoring and adaptation operations.

1 Introduction

The popularity of machine learning (ML) in many application domains [3,15,37,75] has led to a wide adoption of distributed ML systems. Systems such as TensorFlow [1], PyTorch [60], MXNet [10] and MindSpore [53] exploit data and model parallelism [1,54,66,73] to train large ML models on clusters of worker machines. Training is typically done using the stochastic gradient descent (SGD) algorithm [43,68], which iteratively computes gradients to refine the model after each mini-batch of training data. ML systems compile training programs into dataflow graphs [1,10,30,60], which can be executed efficiently on GPUs and other accelerators.

When training ML models, users face the challenge of how to set a large number of *configuration parameters*, which split into two classes: *hyper-parameters* configure the training algorithm, such as SGD, and include the batch size [68], learning rate [68], momentum [63] and floating point precision [24]. Since hyper-parameters relate to the training process itself, their value affects the convergence rate and the final accuracy of the trained model. In addition, *system parameters* [42,73,81] control the operation of the distributed ML system, such as the number of workers, the partitioning of the training data, and the communication topology between workers. They impact the training performance, i.e. the time taken for the model to reach a given target accuracy.

Today users spend a substantial fraction of time tuning configuration parameters. Different ML models have different structures, and thus require different hyper-parameter settings [52]: the hyper-parameters for a vision model such as ResNet [26] differ from those for a language model such as BERT [15]. For each model, hyper-parameters such as batch size, learning rate and weight decay must be adjusted separately to reach a high model accuracy [52]. Approaches for automatic hyper-parameter tuning [2,18,35,45,52] search for the best settings offline at a high resource cost. Furthermore, system parameters such as the number of workers affect the resources consumed by training and their efficiency. Especially in a cloud setting, users must control resource usage to bound costs, while achieving good training performance [52].

Recently, we have seen a growing number of proposals [5,12,16,48,71] that argue for parameters to be adapted *dynamically* during training. For example, many models only reach high accuracy if the learning rate is decreased as the model converges [26,76]; the batch size can be set dynamically based on real-time gradient metrics [14,52,83]; and the communication strategy between workers can be adapted to the current training loss [72]. Similarly, system parameters can be updated to react to changes in exploitable levels of parallelism and resource availability. For example, the number of workers can be changed according to the observed resource utilisation, thus improving the utilisation of expensive accelerators such as GPUs and TPUs [46]; and the best communication topology among workers can be decided based on the available network bandwidth [56].

We observe that existing distributed ML systems and associated libraries (e.g. TensorFlow's Distribution Strategies [1], Horovod [73] and BytePS [8]) make it difficult to support the dynamic adaptation of configuration parameters for a number of reasons: (i) systems do not provide built-in mechanisms for adaptation. Users must rely on external frameworks, e.g. AutoScaling [58] adapts the number of machines by deploying extra scaling agents on each worker. Such external mechanisms are specialised to support only one type of adaptation. Since they are not integrated with the training system, they cannot take advantage of existing functionality and optimisations. In addition, (ii) the monitoring of training metrics introduces high overheads. For example, an 8-GPU server training a ResNet model produces 4 GB of gradients per second [55]. Any monitoring system (e.g. MLFlow [84]) that computes statistical metrics (e.g. variance [78]) over this amount of data consumes substantial compute resources and network bandwidth, which impacts the performance of the training process itself. Finally, (iii) the management of worker state with adaptation is challenging. In existing systems, users typically must checkpoint and restore all state when changing configuration parameters, which can take hundreds of seconds [58].

We describe *KungFu*,[1] a distributed ML training library that is designed to adapt configuration parameters at runtime. The key idea behind KungFu is to support *Adaptation Policies* (APs) written by users, which change hyper- and system parameters during training based on real-time monitored metrics. KungFu achieves this by making three contributions:

(1) Expressing Adaptation Policies. APs describe how configuration parameters should evolve based on monitored metrics. They are based on a high-level programming abstraction following the convention of existing ML frameworks, making integration with training environments seamless.

APs are written using *monitoring*, *communication* and *adaptation* functions: (i) monitoring functions compute metrics for gradients, model variables and worker performance; (ii) communication functions combine locally monitored metrics and transfer training state while adapting parameters; and (iii) adaptation functions update configuration parameters, including hyper- and system parameters.

(2) Making training monitoring efficient. KungFu supports the efficient monitoring of the training process, as needed by APs. Monitoring function calls are translated to *monitoring operators*, which are embedded in the execution dataflow graph. This allows monitoring operators to observe local gradients and reuse existing computation for monitoring.

Locally monitored gradients are combined to compute globally-aggregated metrics. This is achieved by an *asynchronous collective communication layer*, which avoids blocking the dataflow during monitoring. This layer uses a decentralised architecture: each worker maintains a local view of the state for collective communication and incrementally updates the state by exchanging messages in a peer-to-peer fashion. To maximise the performance of gradient monitoring, each KungFu worker has its own scheduler for collective communication. The schedulers cooperate in a decentralised fashion to exploit high-speed multi-GPU networks, e.g. as offered by NVLink through the NCCL interface.

(3) Distributed mechanism for adapting parameters. APs can adapt configuration parameters on distributed worker machines. KungFu represents configuration parameters as computational *configuration operators* embedded within the dataflow graph. In each training step, these operators can alter their output by reading configuration parameters provided by KungFu's asynchronous collective communication layer. Reading the parameters from this layer is efficient because it reuses existing data channels between the communication layer and the dataflow.

APs can dynamically change the parameters in the communication layer, and the result is automatically reflected in the dataflow. KungFu's communication layer uses a *distributed parameter adaptation algorithm* to protect the consistency of changes to configuration parameters while exploiting existing collective communication functions. These functions have been optimised for cross-machine communication, and thus allow adaptation to be performed with low latency.

We implement KungFu's communication layer and adaptation mechanisms in Go (~7k LOCs) and C++ (~3k LOCs), independently of the ML framework. KungFu provides Python bindings (~2k LOCs) for the Adaptation Policy interface, which can be used with existing ML frameworks, including TensorFlow [1], PyTorch [60] and Keras [11].

We evaluate experimentally the benefit and overhead of KungFu's Adaptation Policies. We show that KungFu users can implement a policy that dynamically adapts the batch size based on gradient noise scale, therefore significantly reducing the training time of a ResNet model. We also explore a policy that automatically searches for a cost-effective number of GPUs based on monitored worker performance when training a BERT model, reducing the cost by 20% compared to a static deployment. On a large-scale cloud testbed, we show that KungFu achieves negligible monitoring and adaptation overheads. It achieves up to 98% higher training throughput than Horovod, a state-of-the-art distributed ML system.

2 Adaptation in ML Systems

We first give background on distributed ML systems and their configuration parameters. We then describe current approaches for adapting parameters during training, highlighting why existing systems offer limited support for this.

2.1 Parameters in distributed ML systems

For many ML models, increasing the amount of training data and the size of the model improves accuracy [13, 26]. When training, ML systems therefore exploit the parallelism of modern hardware accelerators such as GPUs. Computation is typ-

[1] https://github.com/lsds/KungFu

ically expressed as a *dataflow* graph [1], which consists of individual operators that can be scaled out.

A supervised ML system trains a model using labelled samples, split into training and test data. A model gradually "learns" to predict the labels by adjusting its *model weights* based on the error. It takes several passes (or *epochs*) over the training data to minimise the prediction error. The test data is used to measure the model accuracy on previously unseen data. A key metric is *test accuracy*, which quantifies the model's ability to make predictions "in the wild".

The model weights are refined iteratively until the model achieves a desired test accuracy. Let w be a vector of the weights, and $\ell_{\mathsf{x}}(\mathsf{w})$ be a loss function that, given w, measures the difference between the predicted label of a sample (x,y) and the ground truth y. During training, an ML system tries to find a w^* that minimises the average loss e.g. using *mini-batch stochastic gradient descent* (SGD) [6,7,68]. More formally,

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \frac{\gamma_n}{b} \sum_{\mathsf{x} \in \mathsf{B}_n} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \tag{1}$$

where γ_n is the learning rate in the n-th iteration of the algorithm, B_n is a batch of b training samples, and $\nabla\ell$ is the gradient of the loss function, averaged over the batch samples.

To scale out the training computation across multiple CPUs or accelerators, ML systems can exploit data parallelism. In *parallel synchronous SGD* (S-SGD), K parallel workers share model replicas and compute gradients for distinct partitions of training data locally. Local gradients are averaged to correct the shared model:

$$\mathsf{w}_{n+1} = \mathsf{w}_n - \frac{\gamma_n}{Kb} \sum_{\mathsf{j} < \mathsf{K}} \sum_{\mathsf{x} \in \mathsf{B}_{n,j}} \nabla \ell_{\mathsf{x}}(\mathsf{w}_n) \tag{2}$$

The averaging of local gradients is usually implemented using *all-reduce* operations provided by collective communication libraries such as Horovod [73] and BytePS [8].

In a distributed ML system, the above training process is affected by many configuration parameters. These parameters can be placed into two groups: (i) accuracy-oriented *hyper-parameters* such as the learning rate γ_n, the batch size $|\mathsf{B}_n|$, momentum [63] and weight decay [38]; and (ii) performance-oriented *system parameters* such as the set of workers, their communication topology for performing synchronisation [56, 72, 73] and their roles during synchronisation, e.g. acting as primary and back-up workers to mitigate stragglers [9].

Hyper-parameters are properties that govern the training process and thus determine its final accuracy. They include variables that determine the model structure and how the network is trained (e.g. the learning rate). Choosing appropriate hyper-parameters plays a key role in training. For example, if the batch size is too high, the model may quickly settle at a local minimum and thus exhibit poor generalisation ability; conversely, if it is too low, the model may suffer from the noise of small batches and thus fail to converge.

System parameters affect the training throughput and thus the time to reach a given target accuracy. They include the configuration of workers (e.g. the number of parallel workers) and how they synchronise (e.g. the communication topology). Choosing appropriate system parameters is important for performance. For example, if the number of workers is too large, the system may suffer from low GPU utilisation due to communication bottlenecks; if the number of workers is too low, the training time for large models becomes prohibitively long.

2.2 Setting parameters in ML systems

Today users spend a substantial fraction of time setting configuration parameters [40]. They often search a large parameter space and decide on configuration parameters following a trial-and-error approach [26, 76]. Specifically, they empirically decide on a set of candidate values, and launch parallel training jobs to evaluate them [40]. They then measure the accuracy of the trained model and the system throughput, and eliminate under-performing settings using early-stop [40] and searching heuristics [29, 33, 41]. After that, they empirically choose an effective setting that reaches the target accuracy given a deadline or a resource budget.

When choosing candidate values for hyper-parameters, users must consider the characteristics of the datasets and models. For example, if the dataset is large (e.g. ImageNet [69]), the candidate batch size can be set larger (e.g. 2048) to improve the robustness of estimated gradients. If the dataset is small (e.g. CIFAR-10 [36]), the candidate batch size must be small (e.g. 64) so that it results in sufficiently many gradients to correct the model. Many large ML models (e.g. ResNet [26] and BERT [15]) have a complex loss space. When training such models, users often need to use a schedule of hyper-parameters (e.g. changing the learning rate at epochs 30, 60 and 90 for ResNet) to improve the quality of a found minima.

When choosing candidate system parameters, users consider the specification of hardware and the conditions of the network. For example, using a ring-based all-reduce topology among workers exploits the full host network bandwidth but it increases the depth of the topology, which adds to latency [80]. Setting the topology to be a star reduces latency but it requires larger bandwidth at the root node. In addition, good system parameters achieve a balance between compute and network utilisation. For example, in a cloud environment in which bandwidth is limited, training with NVIDIA V100 GPUs should only use few workers to prevent underutilising the expensive GPUs due to network bottlenecks; however, if NVIDIA K80 GPUs are used, a user would typically choose more workers to improve system throughput.

2.3 Dynamic adaptation of parameters

Recently, there has been a growing number of proposals to set configuration parameters *dynamically* based on metrics of the training process [5, 12, 16, 48, 71]. The idea is to incorporate

Class	Practitioners	Monitored metrics	Adaptation action
Accuracy	OpenAI [32, 52], Google [77]	Gradient noise scale	Scale **batch size** when the noise scale increases
	Kuaishou [47]	Gradient signal-to-noise ratio	Create online metrics for **model generalisation ability**
	Apple [31]	Gradient variance	Adapt **learning rate** based on the gradient variance
	DeepMind [4], NVIDIA [59]	Gradient second-order metrics	Adapt **learning rate** based on the second-order metrics
Performance	Microsoft [80], Uber [73]	Worker communication rate	Adapt **worker communication topology** based on rates
	Google [58], Huawei [82]	Worker utilisation	Adapt the **number of workers** based on utilisation
	Google [9], MIT [46]	Worker processing speed	Adapt the **roles of workers** based on straggler detection

Tab. 1: Recent proposals for the dynamic adaptation of parameters

knowledge about the training process and its progress through *gradient properties* and *performance metrics* on-the-fly.

Gradient properties include gradient signals, noise or derived signal-to-noise ratios, and they reflect the status of the trained model and the characteristics of the local loss space. They can be used to improve the setting of hyper-parameters, such as batch size and learning rates. Worker performance metrics, such as worker communication rate and processing rate, reflect the hardware and network conditions. They can be used to decide on the number of workers and their communication topology. Using monitored metrics to choose configuration parameters can significantly reduce the need for trial-and-error approaches when searching for suitable hyper-parameters. Instead of spending resources on a search process offline, fewer resources are used for the continuous calibration of configuration parameters during the learning process.

As summarised in Tab. 1, multiple proposals focus on adapting hyper-parameters to improve model accuracy. They often adapt critical hyper-parameters such as batch size and learning rate based on gradient properties. Researchers from OpenAI and Google Brain propose to monitor *gradient noise scale* to predict the optimal batch size when training deep learning models [32, 52, 77]; researchers at Kuaishou use the *gradient signal-to-noise ratio* to evaluate the generalisation ability of a model [47]; Apple automatically scales the learning rate based on gradient variance [31]; and DeepMind and NVIDIA use approximated metrics for *second-order gradients* to predict the best learning rate [4, 59]. Using such properties to set hyper-parameters has become important when training increasingly complex ML models. Users know little of the pre-conditions of these models, and hyper-parameters must be therefore set based on monitored properties [6].

Other proposals adapt system parameters to achieve higher training performance, e.g. reacting to changes in the exploitable parallelism and resource availability. As shown in Tab. 1, Microsoft and Uber propose to measure workers' *communication rates*, which are useful for optimising the topology of all-reduce operations [73, 79, 80]; Google and Huawei monitor *worker utilisation* to update the number of workers for increased resource utilisation [58, 82]; and Google detects straggling workers by analysing the distribution of *worker processing rates* and adapts the roles of workers, e.g. using backup workers to replace stragglers [9]. Using worker performance metrics to tune system parameters is an increasingly common practice. Many distributed ML systems are being deployed in cloud [25] and heterogeneous environments [23]. In such environments, the hardware specifications and network conditions are hard to predict, and thus system parameters must be adapted in the actual environment at runtime.

2.4 Open challenges

Although promising, proposals to adapt parameters are hard to realise in current systems, such as TensorFlow [1] and PyTorch [60]. Practitioners report three main challenges:

(1) No built-in mechanisms for adaptation. Existing distributed training libraries such as Horovod [73] provide insufficient mechanisms for adaptation. Users must rely on external systems that provide custom monitoring and adaptation components, which must be integrated into training systems: AutoScaling [58] adapts the number of workers at runtime by deploying extra scaling agents on each worker using a custom TensorFlow version, which can be managed by the scaling agents; Horovod Elastic [46] requires users to modify their existing training programs so that they can be executed by a custom elastic training runner.

In general, such external systems are specialised to support only a single type of adaptation, usually elasticity. They are not general adaptive training platforms with support for flexible monitoring and different types of adaptation (e.g. related to the communication topology). The lack of unified adaptation abstractions prevents adaptive training from leveraging existing ML system mechanisms and optimisations.

(2) High monitoring overhead. The dynamic adaptation proposals from Tab. 1 require fine-grained monitored metrics as input, but monitoring is expensive: an 8-GPU server training a ResNet model produces 4 GB of gradients per second [55], and this is even larger for recent language models such as BERT [15]. Shipping such an amount of gradient data from workers to a monitoring system such as TensorBoard [1], MLFlow [84], and Prometheus [64] consumes substantial network bandwidth. In addition, there is the overhead of computing complex statistical metrics (e.g. variance [78] or signal-to-noise ratios [47]) from gradients. All of this may impact the performance of the training process itself.

(3) Expensive state management under change. Workers maintain complex state, including model variables, hyper-parameters and system parameters. This state must be managed carefully under adaptation: changing the number of workers must be reflected correctly in all dependent hyper-parameters, such as the learning rate and the data partitioning; otherwise the training result is affected adversely. In existing systems, users typically must checkpoint and restore all state when changing system parameters. This prevents users from extensively applying adaptation during training, as restoring the state can take hundreds of seconds [58].

3 Adaptation Policies

In this section, we introduce *Adaptation Policies* (APs), as supported by KungFu, which adapt configuration parameters based on monitored metrics. We provide an overview of the features of APs and describe the programming abstraction given to users to develop custom APs.

3.1 Overview

Our goals for APs are as follows: (i) we want to provide an expressive policy programming abstraction. The abstraction should follow conventions of existing ML frameworks. Users can thus develop their own policies with low effort. Moreover, (ii) we want to make policies easy to integrate with existing ML frameworks. This will allow users to choose policies based on their training scenarios and combine multiple polices for more advanced adaptation.

APs provide functions to help users implement custom monitoring and adaptation logic. Policies use *monitoring functions* to compute real-time metrics for worker performance and gradients. Locally monitored metrics can be combined using *communication functions*, which cover collective (broadcast and all-reduce) and point-to-point (serve and request) operations. Based on the monitored metrics, policies invoke *adaptation functions* to update the hyper-parameters and system parameters of the systems.

ML frameworks such as TensorFlow [1], Keras [11] and MXNet [10] provide a high-level training abstraction. Users call a generalised training method, which automatically trains a model until certain conditions (e.g. epoch counts) have been met. We want APs to be ported easily between ML frameworks, and we base APs on a framework-independent adaptation API (see Tab. 2).

To integrate this API with a framework, we observe that frameworks often support *user-defined callbacks* (e.g. Hooks in TensorFlow), which are repeatedly called during training. Today these callbacks have limited use—they usually implement checks for finishing conditions and logging functionality. KungFu's adaptation API can be implemented with callbacks, thus facilitating the integration with existing ML frameworks.

3.2 Sample AP for batch size adaptation

Next we describe a sample AP for dynamically increasing the batch size of S-SGD training based on online *gradient noise scale* (GNS) [52, 77]. The increase in batch size is implemented by adding extra workers. This allows the policy to increase training throughput, thus reducing completion time.

```
1  ... # Import ML framework libraries
2  import kungfu as kf
3
4  class GNSPolicy(kf.BasePolicy):
5    # Create policy state
6    def __init__(self, gns_opt):
7      self.opt = gns_opt
8      self.prev_gns = None
9      self.sync = True
10
11   # Synchronise model variables under adaptation
12   def before_epoch(self):
13     if self.sync:
14       for v in self.opt.variables():
15         v = kf.broadcast(v, 0) # Synchronise state
16       self.sync = False
17
18   # Adapt the number of workers if the GNS is growing
19   def after_epoch(self):
20     avg_gns = kf.allreduce(self.opt.gns()) / kf.size()
21     if self.prev_gns is None:
22       self.prev_gns = avg_gns
23     elif avg_gns > self.prev_gns:
24       new_size = int(kf.size() * avg_gns / self.prev_gns)
25       if new_size != kf.size():
26         kf.resize(new_size) # Scale the system
27         self.sync = True
28         self.prev_gns = avg_gns
29
30 model, data = ... # Import a model and a dataset
31 opt = SGDOptimizer(...)
32 opt = kf.OptimizerWithGNS(opt) # Embed monitoring
33 estimator = Estimator(model, opt, ...) # Create a trainer
34 policy = GNSPolicy(opt) # Instantiate the policy
35 estimator.train(data, hooks=[kf.PolicyHook([policy], ...)])
```

Listing 1: Sample Adaptation Policy for GNS

As shown Listing 1, the `GNSPolicy` is defined by extending a `BasePolicy` class (line 4). The policy includes the `__init__` function (line 6), which defines variables that maintain the policy state, such as the previously observed GNS metrics and a flag indicating if workers must synchronise their state (lines 7–9). The policy further has user-defined functions that trigger the adaptation logic at different times in a training process. The `before_epoch` function (line 12) is called at the start of each training epoch. Newly joined workers do not have state that is consistent with existing workers. It is thus necessary to broadcast (line 15) the model state. The `after_epoch` function (line 19) computes the averaged GNS metric at the end of each epoch using an all-reduce operation (line 20). Based on its value, the number of workers is increased by the `resize` function (line 26). To enable GNS monitoring, a user wraps the original `SGDOptimizer` with `kf.OptimizerWithGNS` (lines 31–33), which embeds the GNS monitoring operators into the training dataflow. The `GNSPolicy` is then passed to `PolicyHook` (line 35) to schedule its execution during training.

3.3 Adaptation Policy interface

To define APs, users implement custom *policy functions*. These functions can make API calls for *communication*, *monitoring* and *adaptation*, which are called at different times during the training process. There are three groups of pol-

Class	Functions	Description
Communication	**broadcast**(tensor, rank) → Tensor **allgather**(tensor) → [Tensor] **allreduce**(tensor) → Tensor **keep**(tensor, tag) **request**(rank, name, tag) → Tensor	Broadcast a tensor from a worker to all other workers Gather tensors from all workers and distribute the combined tensor to them Aggregate tensors from all workers and distributes result back to them Keep a tagged tensor which can be requested by other workers Request a tagged tensor from a specified worker
Monitoring	**comm_rates**() → Tensor **gns**(grads, avg_grads) → float . . .	Measure tensor communication rates with other workers Compute the gradient noise scale Custom gradient monitoring operators
Adaptation	**rank**() → int **size**() → int **set_tree**(tree) → bool **resize**(size, workers=None) → bool **detached**() → bool	Get the worker rank Get the number of workers Set the tree of collective communication. Return True if succeed Resize the cluster based on a worker list. Return True if succeed Check if the worker is detached due to resizing

Tab. 2: KungFu APIs for Adaptation Policies

icy functions: (i) the `before/after_train` functions are called at the start and end of a training job, respectively; (ii) the `before/after_epoch` functions are called at the start and end of each training epoch, respectively; and (iii) the `before/after_step` functions are called at the start and end of each training step (i.e. iteration), respectively.

In these policy functions, users can call APIs for training communication, monitoring and adaptation:

Communication. Tab. 2 lists the communication functions in APs. ML frameworks typically use tensors as their basic data types for gradients and model variables. To work with such data, the communication functions take tensors as inputs. APs need to collect monitored metrics from all workers, which can be achieved by calling *collective communication* functions: (i) the `broadcast` function distributes a tensor from a worker to all other workers; (ii) the `allgather` function gathers tensors from all workers and sends the combined tensor to all workers; and (iii) the `allreduce` function aggregates tensors from all workers and returns the results back to them.

In addition, APs must manage and communicate the state of trained models among workers. For example, APs for communication-efficient asynchronous training [44] or robust model averaging [85] must explicitly manage the lifecycle of model states and communicate states to synchronise diverged workers. To support state management and communication, the KungFu API provides a `keep` function that tags a model that is being trained (i.e. the state) and caches it in memory. APs can then read tagged models on other workers asynchronously using a `request` method.

Monitoring. Tab. 2 lists the monitoring functions, which APs use to monitor worker performance and gradients. APs use a `comm_rate` function to measure the tensor communication rates between a local worker and its peers. These rates are useful for deciding the optimal communication topology among workers. To monitor gradients, APs can use `gns` to compute the gradient noise scale. For other statistical metrics, such as variance, policies can use the above collective communication operators. For example, the computation of gradient variance requires both the sum of gradients and the sum of the square of gradients element-wise [78]; both summations can be computed using the `allreduce` function.

Adaptation. Based on monitored metrics, APs call adaptation functions to update configuration parameters. To update hyper-parameters, APs use the `allreduce` function to compute new values and assign them to hyper-parameters, represented as `params`. To update system parameters, APs call: (i) `set_tree` to set the collective communication topology among workers; and (ii) `resize` to update the number of workers. Some workers may need to leave the training after adaptation. APs can use the `detached` function to check if a local worker is still part of the training. If not, the AP can direct workers to exit gracefully.

3.4 Practical considerations

To support APs in real-world distributed ML systems, we must address several practical considerations:

Imperative and symbolic execution. To balance ease-of-use and performance, TensorFlow and PyTorch support imperative (TensorFlow Eager) and symbolic (TensorFlow AutoGraph and PyTorch TorchScript) execution, and APs must therefore also support both.

In TensorFlow Eager and PyTorch programs, users often want to customise the training process. Therefore they explicitly implement the training loop and call custom training functions (e.g. to compute gradients) imperatively in each iteration (i.e. step). This offers great flexibility but prevents callbacks from being used. In this case, KungFu allows the communication, monitoring and adaptation APIs from Tab. 2 to be called directly from inside the training loop.

To support symbolic execution, each function in Tab. 2 has a symbolic version. For example, the `resize` function has an equivalent symbolic version: `resize_op`. This allows KungFu APIs to be embedded into symbolic training programs (e.g. `tf.function`).

Listing 1 shows the hybrid usage of the KungFu imperative and symbolic APIs. The `OptimizerWithGNS` optimiser (line 32) appends `gns_op` operators to each gradient computation operator at compilation time of the dataflow graph, ensuring that the monitoring operator can execute immediately as long as its upstream gradient is available. The policy functions (lines 12–19) are called imperatively. This hybrid usage has an important advantage: the compute-intensive monitoring operators are embedded into the training dataflow graph, while the inexpensive user-defined adaptation logic can be triggered in different policy functions without re-compiling the dataflow graph.

Policy composition. Users can compose multiple APs to create advanced adaptive training applications (i.e. the `PolicyHook` (line 35 in Listing 1) can take multiple APs as input). For example, they can use two APs, one implementing elastic training (denoted as AP1) and the other an adaptive learning rate (denoted as AP2). These APs are chained as a list, which is passed to the training program. A current limitation is that users must decide manually on the correct invocation order: assuming AP1 modifies the worker count and AP2 uses the count to scale the learning rate, the execution order must be AP1 followed by AP2. We leave a mechanism for automatically determining the AP order to future work.

API restrictions. KungFu only imposes minimal API restrictions on AP developers, and APs can call any of the communication/monitoring/adaptation APIs in their callback functions. The calls have global atomic semantics, and there are no constraints on the call order. The only exception is that if a worker has left the cluster (checked by `detached`), it cannot further invoke collective communication APIs.

Error handling. AP developers must handle errors such as worker failures in a traditional fashion. If a KungFu API triggers an internal error, the exception is exposed as a dataflow error, as defined in existing ML frameworks, and checkpoints can be used for recovery.

4 Supporting Monitoring in KungFu

We describe *KungFu*, a distributed training library that can efficiently execute the proposed APs. APs must continuously monitor gradients to determine online adaption decisions, which must be done with low overhead. We begin with an overview of KungFu's design and then describe its support for efficient gradient monitoring in detail.

4.1 Design overview

To support monitoring, KungFu's design has the following goals: (i) KungFu should minimise extra computation when monitoring gradients, and worker resources should focus on training the model; (ii) KungFu should not block the training when monitoring gradients using collective communication operations due to the length of such operations; and (iii) KungFu should efficiently monitor gradients, given the large volume of gradients in today's models.

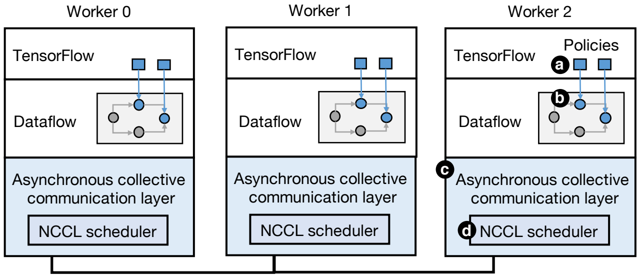


Fig. 1: KungFu architecture

Fig. 1 gives an overview of the KungFu architecture. Users can declare an AP (see ⓐ) as part of a ML training program written in TensorFlow. TensorFlow then creates a *dataflow* program to train the model. To monitor gradients in this dataflow, KungFu transforms the monitoring calls from the AP into *monitoring operators* (see ⓑ), which are embedded in the dataflow. This allows monitoring operators to (i) directly monitor gradients produced by the dataflow and (ii) reuse intermediate computation results in the dataflow for monitoring. For example, it becomes possible to exploit the existing averaged gradients computed when synchronising model replicas.

The monitoring process must compute globally-aggregated metrics from local gradients on workers. In KungFu, this exploits regular collective communication primitives (e.g. all-reduce and all-gather). To overlap monitoring and synchronisation as much as possible, KungFu has a new *asynchronous collective communication layer* (see ⓒ). Using this layer, the dataflow executed by workers can launch asynchronous collective communication operations without blocking.

The asynchronous collective communication layer also avoids having an expensive central coordinator, as used for invoking synchronous collective communication operations in existing systems, such as Horovod [73]. Instead, the KungFu communication layer follows a decentralised architecture: each worker maintains a local view of the complete cluster state used for collective communication and incrementally updates the state by exchanging messages with workers in a peer-to-peer fashion. This decentralised design avoids the need for APs to coordinate the order of collective communication across the system. It also prevents a central coordinator from becoming a potential bottleneck.

To improve the performance of collective communication, each KungFu worker has an NCCL scheduler (see ⓓ). This allows the worker to exploit high-speed multi-GPU networks, such as NVLink [57] and GPU RDMA, through the NCCL interface [56]. The scheduler tracks the availability of gradients on each GPU on the machine, and invokes a local NCCL library to execute a collective communication operation for fetching gradients. To combine the results on multiple workers across different machines, workers use KungFu's asynchronous collective communication layer, thus following a hybrid architecture for collective communication.

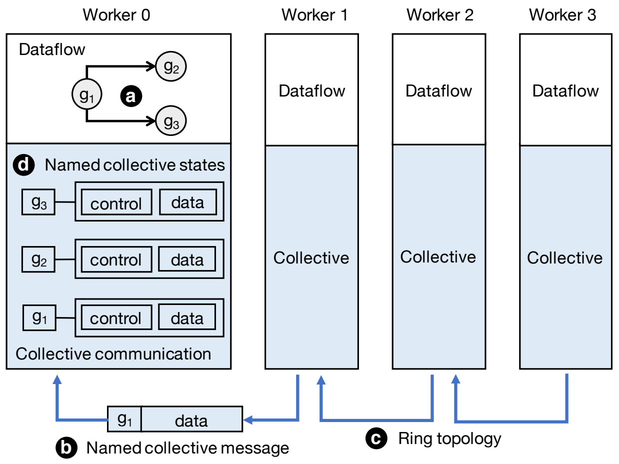


Fig. 2: Dataflow collective communication

4.2 Embedding monitoring within dataflows

To reduce the compute cost of calculating monitored metrics, KungFu exploits the fact that modern ML frameworks (e.g. TensorFlow, MXNet and PyTorch) have built-in *dataflow engines*. These engines offer efficient operators for tensor computation. They also handle the device placement of operators, leveraging parallel computation on accelerators such as GPUs and TPUs. Our observation is that a dataflow engine can also execute monitoring operators by embedding them within the dataflow graph for efficient execution.

To realise this design, KungFu implements the gradient monitoring functions (e.g. `gns` in Tab. 2) and the collective communication functions (e.g. `allreduce`, `broadcast` and `allgather`) as dataflow operators. Since gradients are represented as tensors in the dataflow graph, KungFu's dataflow operators must accept tensors as input. The embedding occurs at compilation time of the dataflow. The monitoring operators are thus part of the dataflow, and they can be scheduled immediately by the dataflow engine when their inputs, i.e. gradients, become available.

To embed its functions as operators, KungFu provides distributed optimisers (e.g. `OptimizerWithGNS` in line 32 in Listing 1) to wrap the original gradient descent optimisers. The KungFu optimisers automatically embed monitoring operators into the training dataflows. These operators intercept gradient tensors produced in each training iteration and forward them to gradient computation operators. The results are maintained in the dataflow and can be read subsequently by the policy functions in APs (line 20).

4.3 Collective communication for dataflows

Dataflows that implement APs use collective communication when computing global gradient metrics. While some gradient metrics (e.g. GNS) can be fused with synchronisation operations, others (e.g. gradient variance) cannot and require extra rounds of collective communication. Asynchronous collective communication thus allows these to be overlapped with gradient synchronisation, reducing the overhead of gradient monitoring. In addition, since dataflows are often executed asynchronously, the coordination with synchronous collective communication, as in Horovod, increases latency, which asynchronous communication avoids.

Allowing dataflows to launch collective communication asynchronously, however, can result in inconsistent computation. For example, the dataflows executed on different workers can produce gradients in different orders. If a worker receives the collective communication messages belonging to different gradients, they may compute inconsistent results.

Fig. 2 illustrates this problem. The example considers 4 workers that perform collective communication. They execute the same dataflow graph (shown as ⓐ), which contains 3 operators for computing gradients, g_1, g_2 and g_3. On different workers, the operators g_2 and g_3 can complete in a different order. To avoid mixing the collective communication data for g_2 and g_3, Horovod [73] employs a centralised coordinator. The coordinator tracks which gradients are ready on workers and launches collective communication operators for these in the correct order. This, however, not only reduces concurrency in the collective communication layer but it also makes the central coordinator a scalability bottleneck.

KungFu adopts a decentralised architecture that efficiently and safely implements asynchronous collective communication. It comprises several components:

Named collective messages. The collective communication layer in KungFu uses *named collective messages* (see ⓑ in Fig. 2) to communicate data. The delivery of these messages follows the collective communication topology (e.g. the ring topology shown in ⓒ). Each named collective message carries (i) the data and (ii) a key, which is used to identify which gradient the data belongs to. The key is derived from the unique key assigned by the ML framework to each dataflow operator. If such as key is unavailable, users can explicitly set it through KungFu's collective communication API.

Named collective states. When receiving a named collective message, a KungFu worker uses it to update its local *named collective state* (see ⓓ). The worker extracts the key from the message and identifies the state entry with the intermediate collective communication. Each entry contains a data and a control part: the data part is the buffer with the intermediate collective communication result, e.g. *max*, *min* or *sum*, which has been accumulated so far; the control part records how many named collective messages have been processed and which worker is the next hop to deliver the local intermediate collective communication results. If the worker finds itself as the last hop in the collective communication topology, it returns the result to the dataflow.

KungFu minimises the memory footprint of the collective messages and states. Since KungFu targets synchronous data parallel training, all asynchronous all-reduce operations must have completed in one training iteration before starting the next. This limits the number of concurrent collective messages and states in memory. To further reduce memory consumption, KungFu frees the states and messages when an asynchronous all-reduce operation has completed. If possible, KungFu reuses buffers from the ML framework (TensorFlow/PyTorch), and it uses a pool to recycle buffers.

4.4 Accelerating collective communication with NCCL

High-end deep learning servers have fast communication links between GPUs (e.g. NVLink, which is 10× faster than PCIe [57]) and fast network connectivity between servers (e.g. GPU RDMA using InfiniBand). To speed up gradient communication and monitoring, KungFu workers exploit these fast links for collective communication.

In practice, users often employ NVLink and GPU RDMA through the NCCL collective communication library [56]. NCCL provides a synchronous collective communication API, following an MPI model [21]. At any time, an NCCL client can only launch a single collective communication operation; otherwise multiple NCCL operations interfere on the NVLink. Existing NCCL-enabled systems (e.g. Horovod-NCCL [73]) therefore adopt a centralised master architecture to coordinate distributed workers when using NCCL operations with gradients. This design, however, is not compatible with KungFu because its collective communication layer has a decentralised architecture.

Instead, KungFu workers use decentralised NCCL schedulers. Each scheduler tracks which gradients are ready on which GPU. The schedulers guarantee that gradients are processed by each NCCL instance in the same order. In the first training step, all NCCL schedulers monitor the order of gradients produced by local dataflow computations. They gather all orders and determine which order is most frequent. The most-common order (named *gradient order*) is broadcast to all schedulers. The schedulers must strictly follow the gradient order when calling NCCL. This ensures that NCCL schedulers launch collective communication for gradients consistently, without a need for central coordination.

KungFu currently offloads all collective communication requests, including those for gradient synchronisation and monitoring, to its NCCL schedulers if NVLink and InfiniBand are available locally. A future extension is to decide *which* requests to offload based on latency requirements: gradient monitoring could use asynchronous collective communication to overlap with training as much as possible; and throughput-intensive gradient synchronisation could use the NCCL-based collective communication.

5 Adapting Parameters of Workers

In this section, we describe how KungFu uses APs to adapt the parameters of its distributed workers.

5.1 Adapting dataflow parameters

Changing configuration parameters of a distributed ML system introduces challenges. Most systems require static parameters, which can be treated as constants when compiling the dataflow graph. After compilation, the dataflow graph is finalised and offloaded to GPUs for execution. Further changes to parameters are thus no longer reflected in the dataflow.

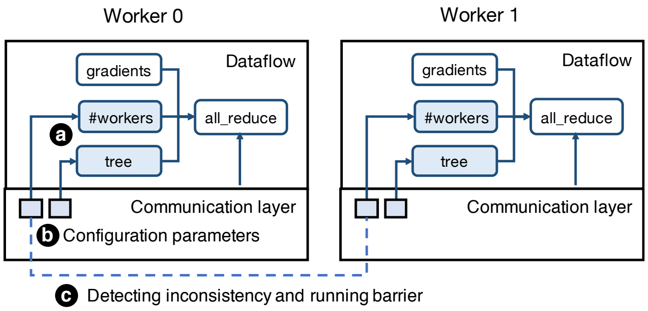


Fig. 3: Parameters as configuration operators

Therefore, elastic ML systems, such as Horovod Elastic [73] or Auto-Scaling [58], require users to use a *dynamic execution mode* of the ML framework, e.g. the “eager” mode in TensorFlow. The dynamic mode allows parameters to be updated in each training step, but it prevents the dataflow from being compiled, which results in large performance overheads. In addition, elastic ML systems only support changes to certain parameters, such as the number of workers. Users must still develop ad-hoc approaches when changing other parameters, such as the communication topology.

KungFu’s design supports the online adaptation of dataflow parameters, while allowing the dataflow graph to be compiled. The core idea is that, instead of providing configuration parameters as static parameters when compiling the dataflow, KungFu adds parameters as computational *configuration operators* as part of the dataflow graph. In each training step, these configuration operators can dynamically alter their output by reading configuration parameters provided by KungFu’s communication layer. This is efficient because it reuses existing data channels between the communication layer and the GPU. APs can dynamically change the parameters in the communication layer, and the result is reflected within the dataflow graph during execution.

Fig. 3 illustrates this idea. We assume that the dataflow graph is used to average local gradients, and it computes the sum of local gradients using an all-reduce operator. The AP changes (i) the number of workers and (ii) their collective communication topology. These two parameters are therefore provided as dataflow configuration operators (see ⓐ) and are used as the input to the all-reduce operator. During execution, the operators read the corresponding configuration parameters (see ⓑ) from the communication layer, and forward them to the all-reduce operator.

5.2 Protecting consistency under adaptation

APs must be able to change the configuration parameters in KungFu’s distributed communication layer. At runtime, KungFu, however, must ensure that these parameters remain

Algorithm 1 Distributed adaptation algorithm for parameters

```
 1: procedure DistributedAdaptation($p, v$)
 2:     $b \leftarrow bytes(v)$                          ▷ Convert $v$ into byte array
 3:     $l \leftarrow length(b)$                         ▷ Get length of b
 4:     $l_0 \leftarrow allreduce(bytes(l), min)$        ▷ byte-wise min
 5:     $l_1 \leftarrow allreduce(bytes(l), max)$        ▷ byte-wise max
 6:     if $l_0 \neq l_1$ then                           ▷ byte-wise comparison
 7:         return false
 8:     end if
 9:     $b_0 \leftarrow allreduce(b, min)$
10:     $b_1 \leftarrow allreduce(b, max)$
11:     if $b_0 \neq b_1$ then
12:         return false
13:     end if
14:     $p.update(v)$                                    ▷ Call the update function of $p$
15:     $\_ \leftarrow allreduce(['0'], min)$            ▷ Run global barrier
16:     return true
17: end procedure
```

consistent when read by the distributed dataflows on workers.

Making global parameter changes consistent with APs introduces two requirements: (i) APs are replicated by workers and executed in parallel. They hold local monitoring state and can receive adaptation commands asynchronously. APs can thus obtain inconsistent values for a given parameter, especially in a large cluster in which many GPU workers asynchronously read new parameter values with high frequency. KungFu must have a mechanism to reject such inconsistent reads. In addition, (ii) when a consistent value is given, KungFu workers assign this value to their local parameters in parallel. The workers must then share a *global barrier* when completing the assignment, which prevents the execution of different dataflows with inconsistent values.

Distributed adaptation algorithm. We describe a distributed parameter adaptation algorithm that fulfils these requirements. To execute with low latency, thus reducing the time during which dataflow execution is blocked under adaptation, it exploits the collective communication layer: since configuration parameters are already hosted by that layer, KungFu re-uses the highly optimised collective communication functions to (i) detect inconsistent updates and (ii) implement a global barrier (shown as Ⓒ in Fig. 3).

Alg. 1 is executed by each KungFu worker when adapting a configuration parameter p with a new value v. It first transforms v into a byte array so that it can be consumed by an all-reduce function, together with a reduce function such as *min* or *max*. After that, the algorithm launches two all-reduce functions to check if the length of b is identical on all workers (lines 4–7). If so, it calls another two all-reduce functions to check if the content of b is consistent (lines 9–13). If this check also passes, v can be safely used for updating p (line 14). All workers must wait on a global barrier until the updates have completed. The barrier is implemented by calling an all-reduce function with a one-byte array (line 15).

Some parameters require custom adaptation logic other than a simple value assignment. For example, changing the number of workers requires workers to exit or join during adaptation. To support this, Alg. 1 can invoke a custom function when updating a parameter (line 14). In the case of the worker set, the function chooses one worker to signal other workers to exit or launch.

Managing data under adaptation. APs can modify the worker count and the batch size. These parameters affect how the training dataset is read and thus the training result. To ensure consistent results under adaptation, all KungFu workers have access to the full dataset.

KungFu supports two approaches to read data batches, depending on if users require *data epochs* to control the training process: (i) if data epochs are not needed, users can use random sampling to read data batches, and the adaptation logic can be triggered at any training step; (ii) with data epochs, KungFu provides a *dynamic data partitioning operator* that replaces the static partitioning operator (e.g. `tf.data.shard`) in the data input pipeline (e.g. DataSet). The dynamic partitioning operator is replicated on all KungFu workers and the operators are synchronised to enact a new parallelism level after a scaling operation. To preserve data epochs, users must invoke the adaptation logic on epoch boundaries only.

Handling failures during adaptation. To tolerate failures, KungFu relies on a highly-available configuration provider (e.g. `ConfigMap` in Kubernetes) to maintain its cluster configuration. The configuration must be updated when a scaling action is committed. In the case of worker failures, the cluster scheduler uses the configuration to restart workers.

6 Evaluation

We experimentally explore the following aspects of the KungFu design and implementation: (i) What are the benefits of enabling adaptation in distributed ML training? (ii) What is the monitoring and adaptation overhead in the training process? (iii) How does KungFu perform in large clusters compared to existing distributed ML systems?

6.1 Experimental set-up

We use both dedicated machines and cloud VMs in our experiments: the dedicated machines are (i) an NVIDIA DGX-1 machine with 8 NVIDIA V100 GPUs interconnected using NVLink, and 72 CPU cores; and (ii) a 20-CPU-core server with 4 NVIDIA Titan X GPUs interconnected using the PCIe bus. The cloud test-bed has 32 VMs, each with 8 vCPUs, 64 GB of memory and 1 NVIDIA K80 GPU.

We use various training workloads as part of the official models provided by TensorFlow [1]: the MobileNetV2 [70] and ResNet-50 [26] models for the ImageNet image classification task [37]; and the BERT [15] model for a natural language processing task, SQuAD [67]. The MobileNetV2 model is 23 MB, ResNet-50 is 98 MB, and BERT is 1 GB in size. These model sizes cover a large spectrum that users observe in practice. We use TensorFlow v1.13.2 to train the models. When possible, we compare the performance to Horovod v0.16.1.

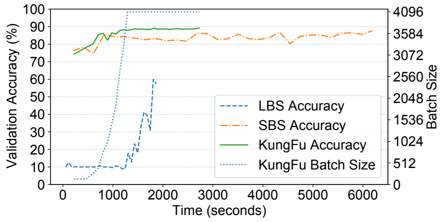


Fig. 4: Adaptive batch size (ResNet-56)

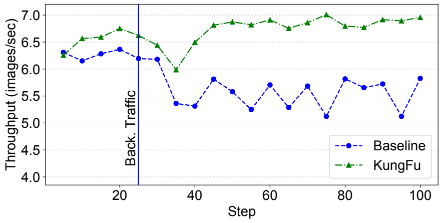


Fig. 5: Adaptive communication strategy (ResNet-50)

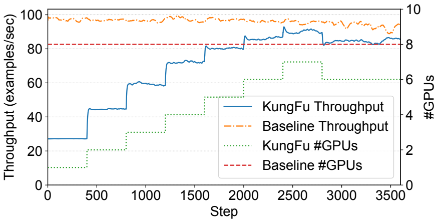


Fig. 6: Adaptive resource provisioning (BERT)

6.2 Adaptation policies

We evaluate three representative APs with KungFu that change various aspects of distributed training:

(1) Adaptive batch size. We implement an AP that adapts the batch size based on GNS when training the ResNet-56 model with the CIFAR-10 dataset. To the best of our knowledge, this AP is the first implementation that evaluates GNS-based batch size tuning in an online training scenario. Past work [52] only empirically evaluates it using offline training traces.

The AP computes GNS using an exponential moving average (α=0.1) and adapts the batch size every 10 epochs as follows: if GNS has increased by a factor of r, it also scales the batch size by r, up to 4096. We compare this AP with two static baselines, which adopt fixed batch sizes of 128 and 4096, respectively. These baselines represent typical choices for small batch size (SBS) and large batch size (LBS). In this experiment, the model is trained for 300 epochs with a learning rate of 0.1, based on TensorFlow's official model. The training is done on the 4 GPU Titan X testbed, with batches shared evenly across GPUs.

Fig. 4 shows the validation accuracy of the model over time. LBS reaches a low validation accuracy (60%) but finishes quickly. SBS reaches a higher validation accuracy (88%) but the constant noise in gradients due to the small batches makes it hard to converge, and the accuracy oscillates between 80% and 90%. A typical issue of SBS is the underutilisation of GPUs: SBS takes around 6000 s to complete training, 2.4× longer than LBS. In practice, fixing the choice of batch size is challenging for users—they have to trade off between model accuracy and hardware utilisation.

The above AP addresses this challenge. As shown by the right y-axis in Fig. 4, the policy dynamically increases the batch size from 128 to 4096 based on GNS. This type of adaptation improves model accuracy: it reaches 88% after around 1000 s, 5× faster than SBS, and eventually converges to 90% after 1300 s. Dynamically increasing the batch size reduces the noise in gradients, which enables the model to converge. Furthermore, it allows the model to better utilise the hardware: the model spends only 400 s more than LBS but achieves 30% higher accuracy.

(2) Adaptive communication strategy. Network infrastructure in cloud environments and multi-tenant clusters may suffer from contention when using all-reduce operations to synchronise gradients, and straggling workers may then slow down the entire system [49]. To address this, we provide an AP that monitors training throughput. If the throughput drops due to network contention, the policy adjusts the topology used by all-reduce, limiting the use of contended network links. In this experiment, we train the ResNet-50 model for 100 steps on 32 VMs. After 25 steps, we introduce background traffic to create network contention. This mimics a cloud environment in which there is dynamic interference in an over-subscribed network.

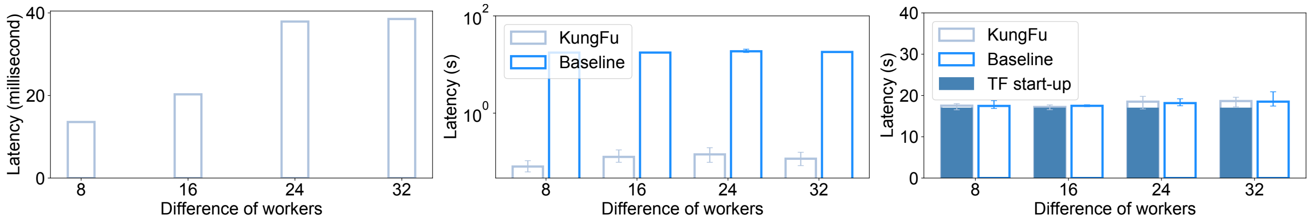
We compare this AP with a static baseline that uses a fixed all-reduce topology. We also attempted to implement a dynamic baseline using OpenMPI and NCCL, but these libraries do not allow runtime control of the all-reduce topology.

Fig. 5 presents the average worker training throughput over training steps. The baseline shows that the workers reach 6.5 images/s at the beginning but this number drops to 5.5 after the network becomes contended. The AP monitors the throughput and detects network contention at step 35. It adapts the communication topology, and the topology recovers throughput: it increases to 7 images/s, even though the background traffic is still on-going.

(3) Adaptive resource provisioning. Users want to decide on a cost-effective number of GPUs when training models. Using many GPUs leads to high training throughput but it also increases cost. Large ML models are synchronising large volumes of gradients. Above a certain amount of resources, communication becomes a bottleneck. In such a case, using more GPUs only gives a marginal performance improvement, despite the higher cost.

We explore an AP that finds the most cost-effective number of GPUs. This policy adds one worker every K steps. It then evaluates the average total accumulated throughput and, if the new throughput is not $1+\alpha(1/\text{size})$ times higher, it removes the worker and stops scaling. We choose α=0.33 and K=400. We compare to a static baseline that always uses the most GPUs. We train the BERT model with a per-GPU batch size of 8 on the 8 GPU V100 testbed.

Fig. 6 shows the results. When all 8 GPUs (right-hand y-axis) are used from the beginning, the total throughput is above 90 examples/second (left-hand y-axis). For KungFu, we see that the throughput rises with the number of GPUs until only a slight increase from 6 to 7 GPUs (step 2400). Due to the small increase with 7 workers, KungFu removes worker 7 at step 2800, stops scaling and resumes training



(a) Set tree latency

(b) Scale in latency

(c) Scale out latency

Fig. 7: Adaptation overhead

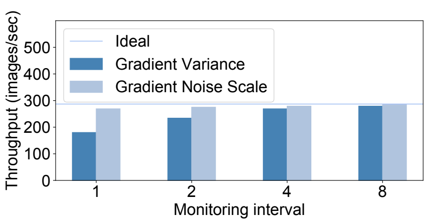
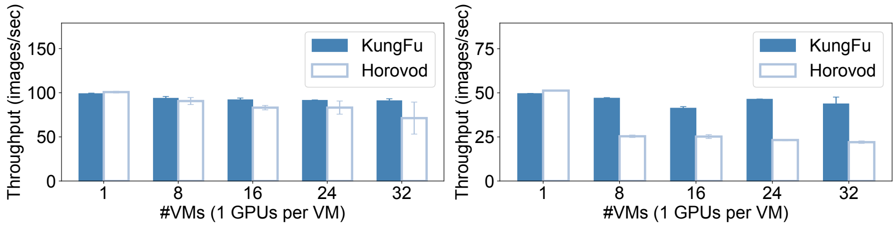


Fig. 8: Monitoring overhead



(a) MobileNetV2

(b) ResNet-50

Fig. 9: Worker throughput with different cluster sizes

with 6 workers. We use the price of a V100 GPU on Azure to estimate the cost efficiency of KungFu and the baseline. The baseline has a cost efficiency of 10,902 examples/USD, and KungFu has 13,097 examples/USD. This indicates that KungFu improves the cost efficiency of the job by 20%.

6.3 Adaptation overhead

Next we evaluate the overhead of adaptation and monitoring.

Adaptation. We evaluate the adaptation latency when changing the communication topology and worker set. We conduct the experiments on the 32 VMs testbed and train ResNet-50. During training, we repetitively change the parameters.

Fig. 7a shows the latency when changing worker communication topology. With 8 VMs, the adaptation completes in 15 ms. With 32 VMs, the delay only increases to 37 ms. This shows the benefits of using the all-reduce function to implement the required consistency checking and global barrier during adaptation.

Fig. 7b shows the latency when scaling down. We decrease the number of workers from 32 to 1 by calling the `resize` function. The function takes 0.2 s to complete. Scaling the system using the checkpoint/recovery mechanism of TensorFlow takes around 20 s to complete, 100× slower than KungFu. This high latency is mainly due to the stop-and-resume time of TensorFlow, and it is consistent with the observations made by others [58, 82]. This result shows the need for supporting efficient adaptation to enable scaling in practice.

Fig. 7c shows the latency when scaling out. Increasing the number of workers from 1 to 32 takes 20 s, the same as the baseline. Since KungFu must preserve the consistency of the training state on workers, it must wait for new workers to be started by TensorFlow. Breaking down this delay, we can see that KungFu spends 0.5 s to complete the scale-out operation, and waits the remaining 19.5 s for the TensorFlow set-up. The long start-up time of TensorFlow can be masked by implementing worker pre-loading [58].

Monitoring. We also consider the overhead when monitoring two metrics, *gradient noise scale* (GNS) and *gradient variance* (GV). The computation of GNS can reuse the averaged gradients produced by the S-SGD computation and thus can be computed locally without extra collective communication; GV, however, compares the square of the sum of gradients and the sum of gradient squares [78]. To compute it, KungFu must launch an additional all-reduce operation for each gradient. We monitor these two metrics when training ResNet-50 for ImageNet on the 8 GPUs V100 testbed. We vary the monitoring interval from 1–8 steps to change load.

Fig. 8 shows the average per-worker training throughput with gradient monitoring. We compare it to the per-worker throughput without monitoring (i.e. the ideal case). The monitoring of GNS has a negligible impact on training, dropping the training throughput from 6.3% to 1.0% based on the monitoring interval. This shows that embedding the monitoring operators as part of the dataflow graph results in low overhead.

The calculation of GV has a tangible throughput impact. The overhead, however, can be amortised by increasing the monitoring interval. The throughput drops by 2.8% when the interval is 8 steps, while still providing acceptable monitoring for APs. APs keep monitored metrics in data sketches and use the accumulated result, usually every several epochs. Iterating through an ImageNet dataset takes more than 40,000 steps, which means that 5000 GV values in an epoch still make estimation reliable.

6.4 Performance

Finally, we evaluate two aspects of KungFu that contribute to overall performance: (i) the asynchronous collective communication layer and (ii) the NCCL scheduler.

Asynchronous Collective Communication Layer. We explore how the performance of KungFu's communication layer compares to Horovod [73], which is a popular high-performance collective communication library used for distributed ML training. We compare the performance with 8, 16, 24 and 32 VMs. By varying the cluster size, we place different loads on collective communication.

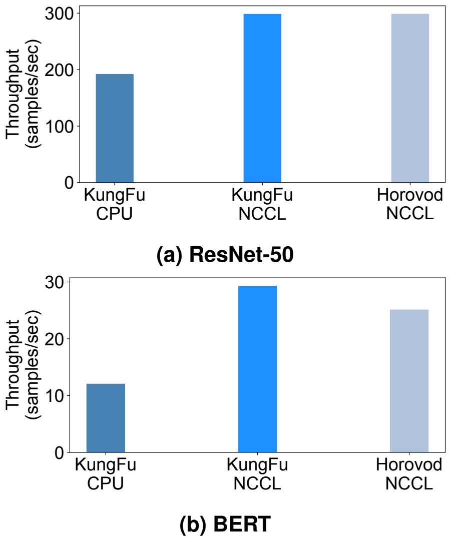
Fig. 9a shows the per-VM training throughput for MobileNetV2/ImageNet under KungFu and Horovod. With 8 VMs, Horovod and KungFu achieve the same throughput. With 32 VMs, however, KungFu outperforms Horovod by 28% due to the benefits of its decentralised design for the communication layer, which avoids the bottleneck of Horovod's master. We also note that Horovod shows a high variance in the training throughput for 32 VMs (up to 24% between min/max). This is caused by network jitter in the cloud environment affecting Horovod's coordinator. Since KungFu workers asynchronously exchange messages for collective communication, they compensate for the network latency and thus achieve stable training throughput even with 32 VMs.

Fig. 9b shows the per-VM training throughput for ResNet50-ImageNet, which is 4× larger than MobileNetV2. With this model, there is more network traffic, and KungFu achieves 98% higher throughput than Horovod with 32 VMs. This improvement is larger than in the case of MobileNetV2 because Horovod must execute collective communication in order, following the MPI convention. KungFu, however, supports concurrent collective communication operations through its named collective message and state mechanisms. This increases concurrency in the communication layer, making it achieve a higher throughput than Horovod, especially with large models such as ResNet.

NCCL Schedulers. We also explore the benefit of the NCCL schedulers in comparison to CPU-based collective communication (i.e. CPU all-reduce) and centralised NCCL scheduling, as used by Horovod-NCCL. The experiment is executed on the DGX-1 machine with all 8 NVIDIA V100 GPUs.

Fig. 10 shows the throughput with CPU-based collective communication and NCCL as used by KungFu and Horovod, respectively. For communication between GPUs on the same machine, NCCL offers a significant performance benefit for both KungFu and Horovod. Comparing KungFu and Horovod with NCCL, we see that, for ResNet (~200 gradients; 97 MB size), KungFu and Horovod experience almost identical performance; for BERT-base (~600 gradients; 1 GB size), KungFu achieves 17% higher throughput than Horovod.

This difference can be attributed to the centralised nature of Horovod's NCCL scheduling. The central scheduler contacts each worker to track which gradients have become available. When gradients are available on all workers, the scheduler calls an all-reduce operation to average gradients. The scheduling overhead grows with the number of gradients, and it becomes a bottleneck with many gradients (e.g. 600 gradients in BERT). A large number of gradients is increasingly common because large models out-perform smaller ones.



(a) ResNet-50

(b) BERT

Fig. 10: Training throughput

7 Related Work

Distributed ML systems. A dataflow abstraction is used in many ML systems, including TensorFlow, PyTorch [60], MXNet [10], Caffe [30] and MindSpore [53]. These systems share similar dataflow designs in which computational operators are used for tensor computation. Compiled dataflows are offloaded to GPUs for the training computation. KungFu reuses the dataflow abstraction to embed operators for the purpose of adaptation.

KungFu uses collective communication functions to implement monitoring and adaptation operations. Such functions are available in most distributed ML systems, including those built on top of MPI [1, 56, 73] as well as parameter-server-based systems [8, 42]. Compared to existing collective communication approaches, KungFu explores a decentralised architecture that is tailored to supporting dataflows used in ML frameworks. It allows multiple collective communication operations to execute concurrently, making it different from current MPI-compatible systems.

Hyper-parameter optimisation and tuning. To find the best settings for hyper-parameters, practitioners and researchers have proposed tuning systems [2, 17, 18, 35, 40, 45] with associated search algorithms [28, 29, 33, 41]. These systems launch parallel training jobs to evaluate different candidate settings of target hyper-parameters. They often aim to minimise resource consumption for finding the best setting. In contrast, KungFu explores how to optimise hyper-parameters continuously in a single training job. It thus proposes mechanisms for efficient monitoring and online adaptation of hyper-parameters during training. It can be used by existing tuning systems to speed up the time of individual training jobs.

Elastic training systems have been proposed to improve the resource utilisation of ML clusters. EDL [82] studies stopfree scaling for TensorFlow workers, and Litz [65] proposes an elastic training framework for ML clusters that consist of parameter servers and training workers. Horovod Elastic [73] and PyTorch Elastic [60] are two open-source elastic training libraries. Compared to these dedicated elastic training systems, KungFu provides a unified framework that can execute different adaptive training jobs efficiently.

Adaptation policies have been explored in streaming systems [20, 22, 27, 61, 62]. Dhalion [19] provides policy support for Apache Storm, and its policies measure data analytics metrics, such as latency and throughput; in contrast, APs in KungFu enable the monitoring of gradients in ML systems. Chi [51] is a control plane for stream processing systems, and it supports online monitoring and adaptation. Compared to Chi, KungFu provides a solution to build adaptive distributed ML systems with high-performance gradient monitoring using dataflows and asynchronous collective communication.

Recently, practitioners have proposed adaptation policies [39, 58] tailored to ML systems. These policies use cost models to infer the performance of a training system and make scaling decisions in response. They could be implemented as APs on top of KungFu to exploit its optimised adaptation and communication infrastructure.

Monitoring training. The ML communities have recognised the importance of monitoring training [71]. CrossBow [34] monitors accelerator utilisation to infer the optimal level of data parallelism when training models. Moreover, gradients metrics are useful to optimise hyper-parameters [50]. There have been efforts on setting the batch size according to signal-to-noise ratios [12] and loss [5], or the learning rate based on other gradient metrics, e.g. square norm of expectation of gradients [16, 71, 74]. Due to the lack of support in current distributed ML systems, such efforts typically only evaluate efficacy using offline collected gradients. KungFu is inspired by these efforts and addresses the missing systems support to implement such proposals.

8 Conclusions

When training modern complex ML models, users want to adapt a wide range of hyper- and system parameters. Existing distributed ML systems were designed at a time when static training regimes were the norm. They thus lack mechanisms for monitoring training metrics and adapting configuration parameters at runtime.

We have presented *KungFu*, a distributed training library that allows users to specify and execute Adaptation Policies. KungFu executes policies efficiently by embedding monitoring and configuration operators as part of the compiled dataflow graph. All communication leverages efficient asynchronous collective communication functions, without interfering with the training process or compromising consistency.

Acknowledgements. We thank our shepherd, Derek Murray, for his thoughtful and detailed comments on the paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, Savannah, GA, USA, 2-4 November 2016.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework. In *25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 2623–2631, Anchorage, AK, USA, 4-8 August 2019.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, and Guoliang Chen. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *33rd International Conference on Machine Learning (ICML)*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20-22 June 2016.

[4] Jimmy Ba, Roger B. Grosse, and James Martens. Distributed Second-Order Optimization using Kronecker-Factored Approximations. In *5th International Conference on Learning Representations (ICLR)*, Toulon, France, 24-26 April 2017.

[5] Lukas Balles, Javier Romero, and Philipp Hennig. Coupling Adaptive Batch Sizes with Learning Rates. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 410–419, 11-15 August 2017.

[6] L. Bottou, F. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 60(2):223–311, 2018.

[7] Léon Bottou. On-line Learning and Stochastic Approximations. In *On-line Learning in Neural Networks*, pages 9–42. New York, NY, USA, 1998.

[8] ByteDance. BytePS - A High Performance and Generic Framework for Distributed DNN Training. https://github.com/bytedance/byteps, 2020.

[9] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting Distributed Synchronous SGD. *CoRR*, abs/1604.00981, 2016.

[10] 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.

[11] François Chollet. Keras. https://keras.io, 2015.

[12] Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated Inference with Adaptive Batches. In *20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1504–1513, Fort Lauderdale, FL, USA, 20–22 Apr 2017.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, and Ke Yang. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, pages 1223–1231. 2012.

[14] Aditya Devarakonda, Maxim Naumov, and Michael Garland. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. *CoRR*, abs/1712.02029, 2017.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pages 4171–4186, Minneapolis, MN, USA, 2-7 June 2019.

[16] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. In *23rd Conference on Learning Theory (COLT)*, pages 257–269, Haifa, Israel, 27-29 June 2010.

[17] Raul Castro Fernandez, William Culhane, Pijika Watcharapichat, Matthias Weidlich, Victoria Lopez Morales, and Peter R. Pietzuch. Meta-Dataflows: Efficient Exploratory Dataflow Jobs. In *International Conference on Management of Data (SIGMOD)*, pages 1157–1172, Houston, TX, USA, 10-15 June 2018.

[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 28*, pages 2962–2970. 2015.

[19] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-Regulating Stream Processing in Heron. *Proc. VLDB Endow.*, 10(12):1825–1836, August 2017.

[20] Tom Z J Fu, Jianbing Ding, Richard T B Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams. In *35th International Conference on Distributed Computing Systems (ICDCS)*, volume 2015-July, pages 411–420, Columbus, Ohio, USA, 29 June - 2 July 2015.

[21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Lecture Notes in Computer Science*, volume 3241, pages 97–104, USA, 2004.

[22] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2014.

[23] 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.

[24] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *32nd International Conference on Machine Learning (ICML)*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1737–1746, Lille, France, 6-11 July 2015.

[25] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Addressing the Straggler Problem for Iterative Convergent Parallel ML. In *7th ACM Symposium on Cloud Computing (SoCC)*, pages 98–111, 5-7 October 2016.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, 27-30 June 2016.

[27] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 261–272, Asilomar, CA, USA, 9-12 January 2011.

[28] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol

Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *CoRR*, abs/1711.09846, 2017.

[29] Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 51 of *Proceedings of Machine Learning Research*, pages 240–248, Cadiz, Spain, 9-11 May 2016.

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM International Conference on Multimedia (MM)*, pages 675–678, Orlando, FL, USA, 03-07 November 2014.

[31] Tyler B. Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. AdaScale SGD: A User-Friendly Algorithm for Distributed Training. *CoRR*, abs/2007.05105, 2020.

[32] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *CoRR*, abs/2001.08361, 2020.

[33] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *30th International Conference on Machine Learning (ICML)*, volume 28 of *Proceedings of Machine Learning Research*, pages 1238–1246, Atlanta, Georgia, USA, 17-19 June 2013.

[34] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling Deep Learning With Small Batch Sizes on Multi-Gpu Servers. *Proceedings of the VLDB Endowment*, 12(11):1399–1412, 2019.

[35] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-WEKA 2.0: Automatic Model Selection and Hyperparameter Optimization In WEKA. *Journal of Machine Learning Research*, 18(25):1–5, 2017.

[36] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances Neural Information Processing Systems 24*, pages 1097–1105, USA, 2012.

[38] Anders Krogh and John A. Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in Neural Information Processing Systems 4*, pages 950–957. 1992.

[39] Woo-Yeon Lee, Yunseong Lee, Joo Seong Jeong, Gyeong-In Yu, Joo Yeon Kim, Ho Jin Park, Beomyeol Jeon, Wonwook Song, Gunhee Kim, and Markus Weimer. Automating System Configuration of Distributed Machine Learning. In *39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2057–2067, Dallas, Texas, USA, 7-9 July 2019. IEEE.

[40] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively Parallel Hyperparameter Tuning. In *Machine Learning and Systems (MLSys)*, pages 230–246. 2-4 March 2020.

[41] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, January 2017.

[42] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning With the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 583–598, Broomfield, CO, USA, 6-8 October 2014.

[43] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient Mini-batch Training for Stochastic Optimization. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (DMKD)*, pages 661–670, New York, NY, USA, 2014.

[44] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous Decentralized Parallel Stochastic Gradient Descent. In *35th International Conference on Machine Learning (ICML)*, volume 80, pages 3043–3052, Stockholm, Sweden, 10-15 July 2018.

[45] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A Research Platform for Distributed Model Selection and Training. *CoRR*, abs/1807.05118, 2018.

[46] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *CoRR*, abs/1904.12043, 2019.

[47] Jinlong Liu, Guoqing Jiang, Yunzhi Bai, Ting Chen, and Huayan Wang. Understanding Why Neural Networks Generalize Well Through GSNR of Parameters. *CoRR*, abs/2001.07384, 2020.

[48] Maren Mahsereci and Philipp Hennig. Probabilistic Line Searches for Stochastic Optimization. In *Advances in Neural Information Processing Systems 28*, pages 181–189. 2015.

[49] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing Network Performance in Distributed Machine Learning. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Santa Clara, CA, USA, 6-7 July 2015.

[50] Luo Mai, Alexandros Koliousis, Guo Li, Andrei-Octavian Brabete, and Peter R. Pietzuch. Taming Hyper-parameters in Deep Learning Systems. *ACM SIGOPS Oper. Syst. Rev.*, 53(1):52–58, 2019.

[51] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, and Vamsi Kuppa. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.

[52] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An Empirical Model of Large-Batch Training. *CoRR*, abs/1812.06162, 2018.

[53] MindSpore. Mindspore Deep Learning Training/Inference Framework. https://github.com/mindspore-ai/mindspore, 2020.

[54] 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 *27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Huntsville, ON, Canada, 27-30 October 2019.

[55] NVIDIA. Data Center Deep Learning Product Performance. https://developer.nvidia.com/deep-learning-performance-training-inference, 2020.

[56] NVIDIA. Optimized Primitives for Collective Multi-GPU Communication. https://github.com/NVIDIA/nccl, 2020.

[57] NVIDIA. The Building Blocks of Advanced Multi-GPU Communication. https://www.nvidia.com/en-us/data-center/nvlink/, 2020.

[58] Andrew Or, Haoyu Zhang, and Michael J. Freedman. Resource Elasticity in Distributed Deep Learning. In *Machine Learning and Systems (MLSys)*, Austin, TX, USA, 2-4 March 2020.

[59] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12359–12367, Long Beach, CA, USA, 16-20 June 2019.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. 2019.

[61] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-Driving Database Management Systems. In *8th Biennial Conference on Innovative Data Systems Research (CIDR)*, Chaminade, CA, USA, 8-11 January 2017.

[62] Thao N. Pham, Panos K. Chrysanthis, and Alexandros Labrinidis. Avoiding Class Warfare: Managing Continuous Queries With Differentiated Classes of Service. *VLDB J.*, 25(2):197–221, 2016.

[63] Boris Polyak. Some Methods of Speeding up the Convergence of Iteration Methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[64] Prometheus. The Prometheus Monitoring System and Time Series Database. https://github.com/prometheus/prometheus, 2019.

[65] 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 *USENIX Annual Technical Conference (ATC)*, pages 631–644, Boston, MA, 11-13 July 2018.

[66] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *CoRR*, abs/1910.02054, 2019.

[67] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know What You Don't Know: Unanswerable Questions for

SQuAD. In *56th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 784–789, Melbourne, Australia, 15-20 July 2018.

[68] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.

[69] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[70] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4510–4520, Salt Lake City, UT, USA, 18-22 June 2018.

[71] Tom Schaul, Sixin Zhang, and Yann LeCun. No More Pesky Learning Rates. In *30th International Conference on Machine Learning (ICML)*, volume 28 of *Proceedings of Machine Learning Research*, pages 343–351, Atlanta, Georgia, USA, 17-19 June 2013.

[72] Vetter Scott, Elpelt Tobias, Franke Rico, and Miranda Yanil Z. Networking Design for HPC and AI on IBM Power Systems (Red Paper), IBM PowerAI Distributed Deep Learning. http://www.redbooks.ibm.com/redpapers/pdfs/redp5478.pdf, April 2018.

[73] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *CoRR*, abs/1802.05799, 2018.

[74] Ravid Shwartz-Ziv and Naftali Tishby. Opening the Black Box of Deep Neural Networks via Information. *CoRR*, abs/1703.00810, 2017.

[75] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. Mastering the Game of Go With Deep Neural Networks and Tree Search. *Nature*, 529:484–503, 2016.

[76] 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.

[77] Samuel L. Smith and Quoc V. Le. A Bayesian Perspective on Generalization and Stochastic Gradient Descent. In *6th International Conference on Learning Representations (ICLR)*, Vancouver, BC, Canada, 30 April - 3 May 2018.

[78] Y. Tsuzuku, Hi. Imachi, and T. Akiba. Variance-based Gradient Compression for Efficient Distributed Deep Learning. In *6th International Conference on Learning Representations (ICLR)*, Vancouver, BC, Canada, 30 April - 3 May 2018.

[79] Marcel Wagenländer, Luo Mai, Guo Li, and Peter R. Pietzuch. Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 13-14 July 2020.

[80] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil R. Devanur, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. *CoRR*, abs/1910.04940, 2019.

[81] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised Deep Learning with Partial Gradient Exchange. In *7th ACM Symposium on Cloud Computing (SoCC)*, SoCC '16, pages 84–97, New York, NY, USA, 5-7 October 2016.

[82] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, and James Cheng. Elastic Deep Learning in Multi-Tenant GPU Cluster. *CoRR*, abs/1909.11985, 2019.

[83] Yang You, Jonathan Hseu, Chris Ying, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large-Batch Training for LSTM and Beyond. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 9:1–9:16, Denver, Colorado, USA, 17-19 November 2019.

[84] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4):39–45, 2018.

[85] Michael R. Zhang, James Lucas, Jimmy Ba, and Geoffrey E. Hinton. Lookahead Optimizer: k Steps Forward, 1 Step Back. In *Advances in Neural Information Processing Systems 32*, pages 9593–9604, Vancouver, BC, Canada, 8-14 December 2019.