Cerebro: A Data System for Optimized Deep Learning Model Selection

Supun Nakandala, Yuhao Zhang, and Arun Kumar
University of California, San Diego
{snakanda, yuz870, arunkk}@eng.ucsd.edu

ABSTRACT

Deep neural networks (deep nets) are revolutionizing many machine learning (ML) applications. But there is a major bottleneck to wider adoption: the pain and resource intensiveness of *model selection*. This empirical process involves exploring deep net architectures and hyper-parameters, often requiring hundreds of trials. Alas, most ML systems focus on training one model at a time, reducing throughput and raising overall resource costs; some also sacrifice reproducibility. We present CEREBRO, a new information system architecture to raise deep net model selection throughput at scale without raising resource costs and without sacrificing reproducibility or accuracy. CEREBRO uses a new parallel SGD execution strategy we call *model hopper parallelism* that hybridizes task- and data-parallelism to mitigate the cons of these prior paradigms and offer the best of both worlds. Experiments on large ML benchmark datasets show CEREBRO offers 3x to 10x runtime savings relative to state-of-the-art data-parallel systems like Parameter Server and Horovod and up to 8x memory/storage savings relative to task-parallel systems. We also enable support for heterogeneous resources and fault tolerance in CEREBRO.

1. INTRODUCTION

Deep learning is revolutionizing many ML applications. Their success at large Web companies has created excitement among practitioners in other settings, including domain sciences, enterprises, and small Web companies, to try deep nets for their applications. But training deep nets is a painful empirical process, since accuracy is tied to the neural architecture and hyper-parameter settings. A common practice to choose these settings is to *empirically compare as many training configurations as possible* for the user. This process is called *model selection*, and it is *unavoidable* because it is how one controls underfitting vs overfitting [56]. Model selection is a major bottleneck for the adoption of deep learning among enterprises and domain scientists due to both the *time spent* and *resource costs*. Not all ML users can afford to throw hundreds of GPUs at their task and burn resources like the Googles and Facebooks of the world.

Case study. We present a real-world model selection scenario. Our collaborators at the UCSD medical school want to develop deep learning-based models for identifying different human activities (e.g., sitting, standing, stepping, etc.) of patients from body-worn accelerometer data. The data is collected from a cohort of 600 people and has a raw data size of 864 GB. During model selection, they want to try different deep net architectures such as convolution neural networks (CNNs), long short-term memory models (LSTMs), and composite models such as CNN-LSTMs, which have recently shown state-of-the-art results for multivariate time-series classification [33, 49]. They also want to explore different prediction window sizes (e.g., predictions generated at every 5 seconds vs. 15 seconds) and different data labeling schemes (e.g., sitting, standing, and stepping vs. sitting and not sitting). Furthermore, the deep learning training process also involves tuning various hyper-parameters like learning rate and regularization coefficient.

In the above scenario it is clear that the model selection process generates dozens, if not hundreds, of different models that need to be evaluated in order to pick the best one for the prediction task. Due to the scale of the data and the complexity of the task, it is too tedious and time-consuming to manually steer this process running models one by one. Parallel execution on a cluster is critical for reasonable runtimes. Moreover, since they often changed the windows and output semantics for their health-related analysis, we had to rerun the whole model selection process over and over several times to get the best accuracy for their evolving task definitions. Finally, reproducible model training is also a key requirement in such scientific settings. All this underscores the importance of automatically scaling deep net model selection on a cluster with high throughput.

System Desiderata. We have the following key desiderata for a deep net model selection system.

1) Scalability. Deep learning often has large training datasets, larger than single-node memory and sometimes even disk. Deep net model selection is also highly compute-intensive. Thus, we desire out-of-the-box scalability to a cluster with large partitioned datasets (*data scalability*) and distributed execution (*compute scalability*).

2) High throughput. Regardless of manual grid/random searches or AutoML searches, a key bottleneck for model selection is *throughput*: how many training configurations are evaluated per unit time. Higher throughput means deep net users can iterate through more configurations in bulk, potentially reaching a better accuracy sooner.

3) Overall resource efficiency. Deep net training uses variants of mini-batch stochastic gradient descent (SGD) [7, 10, 11]. To improve efficiency, the model selection system has to *avoid wasting resources* and *maximize resource utilization* for executing SGD on a cluster. We identify 4 key parts of efficiency: (1) *per-epoch efficiency*:

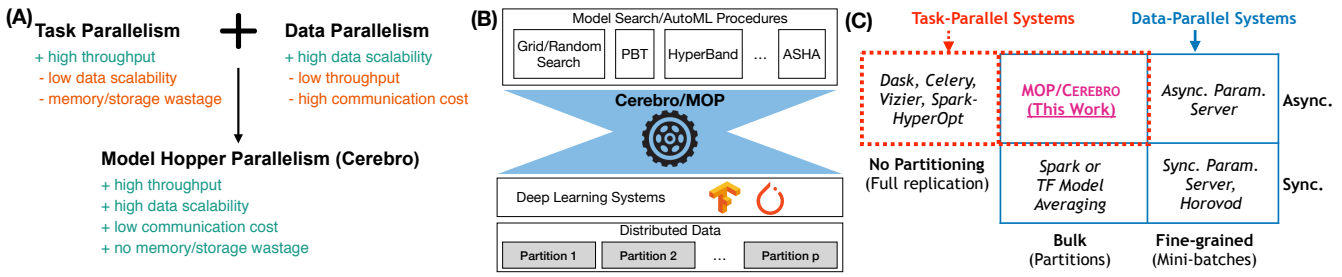


Figure 1: (A) Cerebro combines the advantages of both task- and data-parallelism, (B) System design philosophy and approach of Cerebro/MOP(introduced in [45]): "narrow waist" architecture in which multiple model selection procedures and multiple deep learning tools are supported–unmodified–for specifying/executing deep net computations. MOP is our novel resource-efficient distributed execution approach, and (C) Model hopper parallelism (MOP) as a hybrid approach of task- and data-parallelism. It is the first known form of *bulk asynchronous* parallelism, filling a gap in the parallel data systems literature

time to complete an epoch of training; (2) *convergence efficiency*: time to reach a given accuracy metric; (3) *memory/storage efficiency*: amount of memory/storage used by the system; and (4) *communication efficiency*: amount of network bandwidth used by the system. In cloud settings, compute, memory/storage, and network all matter for overall costs because resources are pay-as-you-go; on shared clusters, which are common in academia, wastefully hogging any resource is unethical.

4) Reproducibility. Ad hoc model selection with distributed training is a key reason for the "reproducibility crisis" in deep learning [60]. While some Web giants may not care about unreproducibility for some use cases, this is a showstopper issue for many enterprises due to auditing, regulations, and/or other legal reasons. Most domain scientists also inherently value reproducibility.

Limitations of Existing Landscape. We compared existing approaches to see how well they cover the above desiderata. Unfortunately, each approach falls short on some major desiderata, as we summarize next. Figure 3 and Section 2.2 present our analysis in depth.

1) False dichotomy of task- and data-parallelism. Prior work on model selection systems, primarily from the ML world, almost exclusively focus on the task-parallel setting [29, 38, 39]. This ignores a pervasive approach to scale to large data on clusters: data partitioning (sharding). A disjoint line of work on data-parallel ML systems do consider partitioned data but focus on training one model at a time, not model selection workloads [40, 55]. Model selection on partitioned data is increasingly important, since HDFS (for Spark/Hadoop) and parallel RDBMSs usually store large datasets in a multi-node partitioned manner.

2) Resource inefficiencies. Due to the false dichotomy, naively combining the above mentioned approaches could cause overheads and resource wastage (Section 2 explains more). For instance, using task-parallelism on HDFS requires extra data movement and potential caching, substantially wasting network and memory/storage resources. An alternative is remote data storage (e.g., S3) and reading repeatedly at every iteration of SGD. But this leads to orders of magnitude higher network costs by flooding the network with lots of redundant data reads. On the other hand, data-parallel systems that train one model at a time (e.g., Horovod [55] and Parameter Servers [40]) incur high communication costs, leading to high runtimes.

Overall, we see a major gap between task- and data-parallel systems today, which leads to substantially lower *overall resource efficiency, i.e., when compute, memory/storage, and network are considered holistically.*

Our Proposed System. We present CEREBRO, a new system for deep learning model selection that mitigates the above issues with both task- and data-parallel execution. As Figure 1 (A) shows, CEREBRO can combine the advantages of both task- and data-parallelism, while avoiding the limitations of each. It raises model selection throughput without raising resource costs. Our target setting is *small clusters* (say, tens of nodes), which covers a vast majority (over 90%) of parallel ML workloads in practice [52]. We focus on the common setting of partitioned data on such clusters. Figure 1 (B) shows the system design philosophy of CEREBRO: a narrow-waist architecture inspired by [37] to support multiple AutoML procedures and deep net frameworks.

Summary of Our Techniques. At the heart of CEREBRO is a simple but novel hybrid of task- and data-parallelism we call *model hopper parallelism* (MOP) that fulfills all of our desiderata. MOP is based on our insight about a formal optimization theoretic property of SGD: *robustness to the random ordering of the data.* Figure 1 (C) positions MOP against prior approaches: it is first known form of "Bulk Asynchronous" parallelism, a hybridization of the Bulk Synchronous parallelism common in the database world and task-parallelism common in the ML world. As Figure 2 shows, MOP has the network and memory/storage efficiency of BSP but offers much better ML convergence behavior. Prior work has shown that the BSP approach for distributed SGD (also called "model averaging") has poor convergence behavior [18]. Overall, *considering all resources holistically–compute, memory/storage, and network–MOP can be the resource-optimal choice* in our target setting.

With MOP as its basis, CEREBRO devises an *optimizing scheduler* to efficiently execute deep net model selection on small clusters. We formalize our scheduling problem as a mixed integer linear program (MILP). We compare alternate candidate algorithms with simulations and find that a simple randomized algorithm has surprisingly good perfor-

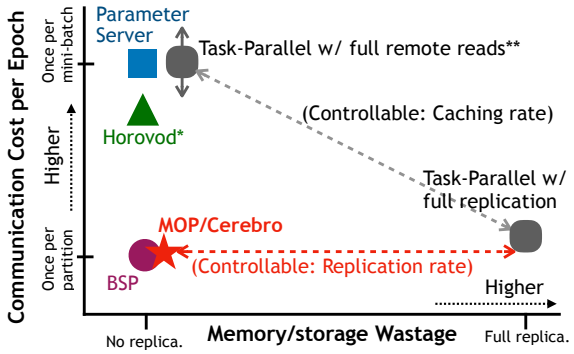


Figure 2: Conceptual comparison of MOP/Cerebro with prior art on two key axes of resource efficiency: communication cost per epoch and memory/storage wastage. Dashed line means that approach has a controllable parameter. *Horovod uses a more efficient communication mechanism than Parameter Server (PS), leading to a relatively lower communication cost. **Task-Parallelism with full remote reads has varying communication costs (higher or lower than PS) based on dataset size.

mance on all aspects (Section 5). We then extend our scheduler to support replication of partitions, fault tolerance, and elasticity out of the box (Section 5.6). Such systems-level features are crucial for deep net model selection workloads, which can often run for days. We also weigh a hybrid of Cerebro with Horovod for model selection workloads with low degrees of parallelism. Overall, this paper makes the following contributions:

- We present a new parallel SGD execution approach we call *model hopper parallelism* (MOP) that satisfies all the desiderata listed earlier by exploiting a formal property of SGD.
- We build Cerebro, a general and extensible deep net model selection system using MOP. Cerebro can support arbitrary deep nets and data types, as well as multiple deep learning tools and AutoML procedures. We integrate it with TensorFlow and PyTorch.
- We formalize the scheduling problem of Cerebro and compare 3 alternatives (MILP solver, approximate, and randomized) using simulations. We find that a randomized scheduler works well in our setting.
- We extend Cerebro to exploit partial data replication and also support fault tolerance and elasticity.
- We perform extensive experiments on real model selection workloads with two large benchmark ML datasets: *ImageNet* and *Criteo*. Cerebro offers 3x to 10x runtime savings against purely data-parallel systems and up to 8x memory/storage savings against purely task-parallel systems. Cerebro also has linear speedup behavior (strong scaling).

2. BACKGROUND AND TRADEOFFS

We briefly explain mini-batch SGD, the method used for training deep nets. We then compare existing approaches for parallel deep net training and their tradeoffs.

2.1 Deep Net Training with Mini-batch SGD

Deep net training is a non-convex optimization problem [23]. It is solved by mini-batch SGD or its variants (e.g., Adam [34] and RMSprop [4]). SGD is an iterative process that performs multiple passes over the data. Each pass is called an *epoch*. In an epoch, it randomly samples a batch of examples without replacement–called a *mini-batch*–and uses that to estimate the gradient and make a model update. Large datasets have 1000s to millions of mini-batches; so, an epoch makes as many model updates. SGD is inherently sequential; deviating from sequential execution may lead to poor convergence behavior, typically raising the number of epochs needed for a given accuracy. We refer the interested reader to [7, 10] for more technical details on SGD.

2.2 Systems for Distributed Deep Net Training

Most deep learning tools (e.g., TensorFlow) focus on the latency of training *one model at a time*, not on throughput. A popular way to raise throughput is *parallelism*. Thus, various multi-node parallel execution approaches have been studied. All of them fall short on some desiderata, as Figure 3 shows. We group these approaches into 4 categories:

Embarrassingly Task Parallel. Tools such as Python Dask, Celery, Vizier [21], and Ray [44] can run different training configurations on different workers in a task-parallel manner. Each worker can use logically sequential SGD, which yields the best convergence efficiency. This is also reproducible. There is no communication across workers during training, but the whole dataset must be copied to each worker, which does not scale to large partitioned datasets. Copying datasets to all workers is also *highly wasteful of resources*, both memory and storage, which raises costs. Alternatively, one can use remote storage (e.g., S3) and read data remotely every epoch. But such repeated reads wastefully flood the network with massively redundant data.

Bulk Synchronous Parallel (BSP). BSP systems such as Spark and TensorFlow with model averaging [1] parallelize one model at a time. They partition the dataset across workers, yielding high memory/storage efficiency. They broadcast a model, train models independently on each worker's partition, collect all models on the master, average the weights (or gradients), and repeat this every epoch. Alas, this approach converges poorly for highly non-convex models; so, it is almost never used for deep net training [58].

Centralized Fine-grained. These systems also parallelize one model at a time on partitioned data but at the finer granularity of each mini-batch. The most prominent example is Parameter Server (PS) [40]. PS is a family of frameworks for data-parallel machine learning. A typical PS consists of servers and workers; servers maintain the globally shared model weights, while workers compute SGD gradients on locally stored and partitioned data. Workers also communicate with servers to update and retrieve the model weights. Based on the nature of these communications, PS has two variants: *synchronous* and *asynchronous*. Asynchronous PS is highly scalable but unreproducible; it often has poorer convergence than synchronous PS due to stale updates but synchronous PS has higher overhead for synchronization. All PS-style approaches have *high communication costs* compared to BSP due to their centralized all-to-

Desiderata	Embarrassing Task Parallelism (e.g., Dask, Celery, Vizier)	Data Parallelism			Model Hopper Parallelism (Our Work)
		Bulk Synchronous (e.g., Spark, Greenplum)	Centralized Fine-grained (e.g., Async Parameter Server)	Decentralized Fine-grained (e.g., Horovod)	
Data Scalability	✗ No (Full Replication) Wasteful (Remote Reads)	✓ Yes	✓ Yes	✓ Yes	✓ Yes
Per-Epoch Efficiency	✓ High	✓ High	✗✗ Lowest	✗ Low	✓ High
SGD Convergence Efficiency	✓✓ Highest	✗✗ Lowest	⟷ Medium	✓ High	✓✓ Highest
Memory/Storage Efficiency	✗✗ Lowest	✓ High	✓ High	✓ High	✓ High
Reproducibility	✓ Yes	✓ Yes	✗ No	✓ Yes	✓ Yes

Figure 3: Qualitative comparisons of existing systems on key desiderata for a model selection system.

Table 1: Notation used in Section 3

Symbol	Description
S	Set of training configurations
p	Number of data partitions/workers
k	Number of epochs for S to be trained
m	Model size (uniform for exposition sake)
b	Mini-batch size
D	Training dataset ($<D>$: dataset size, $\|D\|$: number of records)

one communications, which is proportional to the number of mini-batches.

Decentralized Fine-grained. The best example is Horovod [55]. It adopts HPC-style techniques to enable synchronous all-reduce SGD. While this approach is bandwidth optimal, communication latency is still proportional to the number of workers, and the synchronization barrier can become a bottleneck. The total communication overhead is also proportional to the number of mini-batches.

3. MODEL HOPPER PARALLELISM

We first explain how MOP works and its properties. Table 1 presents some notation. We also theoretically compare the communication costs of MOP and prior approaches.

3.1 Basic Idea of MOP

We are given a set S of training configurations ("configs" for short). For simplicity of exposition, assume for now each runs for k epochs[1]. Shuffle the dataset once and split into p partitions, with each partition located on one of p worker machines. Given these inputs, MOP works as follows. Pick p configs from S and assign one per worker (Section 5 explains how we pick the subset). On each worker, the assigned config is trained on the local partition for a single *sub-epoch*, which we also call a *training unit*. Completing a training unit puts that worker back to the idle state. An idle worker is then assigned a new config that has not already been trained and also not being currently trained on another worker. Overall, a model "hops" from one worker to another after a sub-epoch. Repeat this process until all configs are trained on all partitions, completing one epoch for each model. Repeat this every epoch until all configs in S are trained for k epochs. The invariants of MOP can be summarized as follows:

- *Completeness:* In a single epoch, each training config is trained on all workers exactly once.
- *Model training isolation:* Two training units of the same config are not run simultaneously.
- *Worker/partition exclusive access:* A worker executes only one training unit at a time.
- *Non-preemptive execution:* An individual training unit is run without preemption once started.

Insights Underpinning MOP. MOP exploits a formal property of SGD: *any random ordering* of examples suffices for convergence [7, 10]. Each of the p configs visits the data partitions in a different (pseudorandom) yet in sequential order. Thus, MOP offers high accuracy for all models, comparable to sequential SGD. While SGD's robustness has been exploited before in ML systems, e.g., in Parameter Server [40], MOP exploit it at the *partition level* instead of at the mini-batch level to reduce communication costs. This is possible because we connect this property with model selection workloads instead of training one model at a time.

Positioning MOP. As Figure 1 (C) shows, MOP is a new hybrid of task- and data-parallelism that is a form of "bulk asynchronous" parallelism. Like task-parallelism, MOP trains many configs in parallel but like BSP, it runs on partitions. So, MOP is more fine-grained than task parallelism but more coarse-grained than BSP. MOP has no global synchronization barrier within an epoch. Later in Section 5, we dive into how Cerebro uses MOP to schedule S efficiently and in a general way. Overall, while the core idea of MOP is simple–perhaps even obvious in hindsight–it has hitherto not been exploited in its full generality in ML systems.

Reproducibility. MOP does not restrict the visit ordering. So, reproducibility is trivial in MOP: log the worker visit order for each configuration per epoch and replay with this order. Crucially, this logging incurs very negligible overhead because a model hops only *once per partition*, not for every mini-batch, at each epoch.

3.2 Communication Cost Analysis

[1]Cerebro can train models for different number of epochs. Section 4.2 (Supporting Multiple AutoML Procedures) provides more details about this.

Table 2: Communication cost analysis of MOP and other approaches.

	Communication Cost
Model Hopper Parallelism	$kmps\|S\| + m\|S\|$
Embarrassing Task Parallelism	$p\langle D\rangle + m\|S\|$
Bulk Synchronous Parallelism	$2kmps\|S\|$
Centralized Fine-grained	$2kmps\|S\|\left\lceil \frac{\|D\|}{bp} \right\rceil$
Decentralized Fine-grained	$kmps\|S\|\left\lceil \frac{\|D\|}{bp} \right\rceil$

We summarize the communication cost of MOP and other approaches in Table 2. MOP reaches the theoretical minimum cost of $kmp|S|$ for data-partitioned training, assuming data is fixed and equivalence to sequential SGD is desired. Crucially, note that this cost does not depend on batch size, which underpins MOP's higher efficiency. BSP also has the same asymptotic cost but unlike MOP, BSP typically converges poorly for deep nets and lacks sequential-equivalence. Fine-grained approaches like PS and Horovod have communication costs proportional to the number of mini-batches, which can be orders of magnitude higher. In our setting, p is under low 10s, but the number of mini-batches can even be 1000s to millions based on the batch size.

4. SYSTEM OVERVIEW

We present an overview of Cerebro, an ML system that uses MOP to execute deep net model selection workloads.

4.1 User-facing API

The Cerebro API allows users to do 2 things: (1) register workers and data; and (2) launch a deep net model selection workload and get the results. Workers are registered by their IP addresses. For registering a dataset, Cerebro expects the list of data partitions and their availability on each worker. We assume shuffling and data partitioning across workers is already handled by other means, since distributed data shuffling is well studied. This common data ETL step is also orthogonal to our focus and is not a major part of the total runtime for iterative deep net training.

Cerebro takes the reference to the dataset, set of initial training configs, the AutoML procedure, and 3 user-defined functions: *input_fn*, *model_fn*, and *train_fn*. It first invokes *input_fn* to read and pre-process the data. It then invokes *model_fn* to instantiate the neural architecture and potentially *restore* the model state from a previous *checkpointed* state. The *train_fn* is invoked to perform one sub-epoch of training. We assume validation data is also partitioned and use the same infrastructure for evaluation. During evaluation, Cerebro marks model parameters as non-trainable before invoking *train_fn*. We also support higher-level API methods for AutoML procedures that resemble the popular APIs of Keras [51]. Note that *model_fn* is highly general, i.e., Cerebro supports *all* neural computational graphs on all data types supported by the underlying deep learning tool, including CNNs, RNNs, transformers, etc. on structured data, text, images, video, etc. Due to space constraints, more details of our APIs, including full method signatures and a fleshed out example of how to use Cerebro are provided in the appendix of our technical report [46].

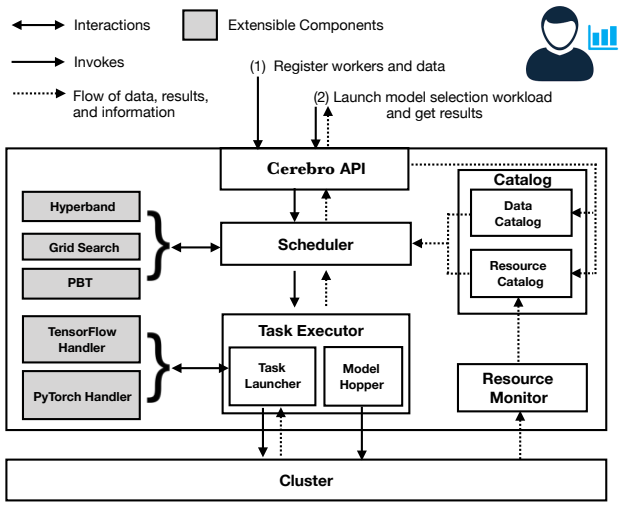


Figure 4: System architecture of Cerebro.

4.2 System Architecture

We adopt an extensible architecture, as Figure 4 shows. This allows us to easily support multiple deep learning tools and AutoML procedures. There are 5 main components: (1) API, (2) Scheduler, (3) Task Executor, (4) Catalog, and (5) Resource Monitor. Scheduler is responsible for orchestrating the entire workload. It relies on worker and data availability information from the Catalog. Task Executor launches training units on the cluster and also handles model hops. Resource Monitor is responsible for detecting worker failures and updating the Resource Catalog. Section 5 explains how the Scheduler works and how we achieve fault tolerance and elasticity. Next, we describe how Cerebro's architecture enables high system generality.

Supporting Multiple Deep Learning Tools. The functions *input_fn*, *model_fn*, and *train_fn* are written by users in the deep learning tool's APIs. We currently support TensorFlow and PyTorch (it is simple to add support for more). To support multiple such tools, we adopt a handler-based architecture to delineate tool-specific aspects: model training, checkpointing and restoring. Note that checkpointing and restoring is how Cerebro realizes model hops. Task Executor automatically injects the tool-specific aspects from the corresponding tool's handler and runs these functions on the workers. Overall, Cerebro's architecture is highly general and supports virtually all forms of data types, deep net architectures, loss functions, and SGD-based optimizers.

Supporting Multiple AutoML Procedures Metaheuristics called AutoML procedures are common for exploring training configs. We now make a key observation about such procedures that underpins our Scheduler. Most AutoML procedures fit a *common template*: create an initial set of configs (S) and evaluate them after each epoch (or every few epochs). Based on the evaluations, terminate some configurations (e.g., as in Hyperband [38] and PBT [29]) or add new configurations (e.g., as in PBT). Grid/random search is a one-shot instance of this template. Thus, we adopt this template for our Scheduler. Given S, Cerebro trains all models in S for one epoch and passes control back to the

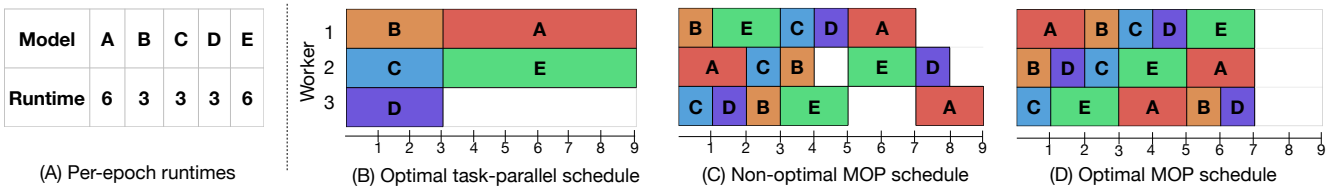


Figure 5: Task parallel and MOP schedules generated for a sample model selection workload.

Table 3: Additional notation used in the MOP MILP formulation

Symbol	Description
$T \in \mathbb{R}^{\lvert S\rvert \times p}$	$T_{i,j}$ is the runtime of unit $s_{i,j}$ (i^{th} configuration on j^{th} worker)
C	Makespan of the workload
$X \in \mathbb{R}^{\lvert S\rvert \times p}$	$X_{i,j}$ is the start time of the execution of i^{th} configuration on j^{th} partition/worker
$Y \in \{0,1\}^{\lvert S\rvert \times p \times p}$	$Y_{i,j,j'} = 1 \iff X_{i,j} < X_{i,j'}$
$Z \in \{0,1\}^{\lvert S\rvert \times \lvert S\rvert \times p}$	$Z_{i,i',j} = 1 \iff X_{i,j} < X_{i',j}$
V	Very large value (Default: sum of training unit runtimes)

corresponding AutoML procedure for convergence/termination/addition evaluations. Cerebro then gets a potentially modified set S' for the next epoch. This approach also lets Cerebro support data re-shuffling after each epoch. But the default (and common practice) is to shuffle only once up front. Grid/random search (perhaps the most popular in practice), Hyperband, and PBT (and more procedures) conform to this common template and are currently supported.

ASHA [39] and Hyperopt [6] are two notable exceptions to the above template, since they do not have a global synchronized evaluation of training configs after an epoch and are somewhat tied to task-parallel execution. Thus, while MOP/Cerebro cannot ensure logically same execution as ASHA or HyperOpt on task-parallelism, it is still possible to emulate them on MOP/Cerebro without any modifications to our system. In fact, our experiments with ASHA show that ASHA on Cerebro has comparable–even slightly better!–convergence behavior than ASHA on pure task-parallelism (Section 6.3).

4.3 System Implementation Details

We prototype Cerebro in Python using the XML-RPC client-server package. Cerebro itself runs on the client. Each worker runs a single service. Scheduling follows a push-based model–Scheduler assigns tasks and periodically checks the responses from the workers. We use a network file server (NFS) as the central repository for checkpointed models and as a common file system visible to all workers. Model hopping is realized implicitly by workers writing models to and reading models from this shared file system. Technically, this doubles the communication cost of MOP to $2kmp|S|$, but this is still a negligible overhead. Using NFS just greatly reduces engineering complexity to implement model hops.

5. CEREBRO SCHEDULER

Scheduling training units on workers properly is critical because pathological orderings can under-utilize resources substantially, especially when the neural architectures and/or workers are heterogeneous. For example, assume we are given the model selection workload shown in Figure 5 (A). Also assume all workers are homogeneous and there is no data replication for MOP. For one epoch of training, Figure 5 (B) shows an optimal task-parallel schedule for this workload that has a makespan of 9 units. Figure 5 (C) shows a non-optimal MOP schedule that also has a makespan of 9 units. But, as shown in Figure 5 (D), an optimal MOP schedule for this workload will have a makespan of only 7 units! In this example, MOP's training unit-based scheduling provides more flexibility to improve resource utilization. Next, we formally define the MOP-based scheduling problem and explain how we design our Scheduler.

5.1 Formal Problem Statement as MILP

Suppose the runtimes of each training unit, aka *unit times*, are given. These can be obtained with, say, a pilot run for a few mini-batches and then extrapolating (this overhead will be marginal). For starters, assume each of the p data partitions is assigned to only one worker. The objective and constraints of the MOP-based scheduling problem is as follows. Table 3 lists the additional notation used here.

$$\texttt{Objective:} \quad \min_{C,X,Y,Z} C \tag{1}$$

$$\begin{aligned}
&\texttt{Constraints:} \\
&\forall i, i' \in [1,\ldots,|S|]\ \forall j, j' \in [1,\ldots,p] \\
&(a)\ X_{i,j} \geq X_{i,j'} + T_{i,j'} - V \cdot Y_{i,j,j'} \\
&(b)\ X_{i,j'} \geq X_{i,j} + T_{i,j} - V \cdot (1 - Y_{i,j,j'}) \\
&(c)\ X_{i,j} \geq X_{i',j} + T_{i',j} - V \cdot Z_{i,i',j} \\
&(d)\ X_{i',j} \geq X_{i,j} + T_{i,j} - V \cdot (1 - Z_{i,i',j}) \\
&(e)\ X_{i,j} \geq 0 \\
&(f)\ C \geq X_{i,j} + T_{i,j}
\end{aligned} \tag{2}$$

We need to minimize makespan C, subject to the constraints on C, unit start times X, model training isolation matrix Y, and worker/partition exclusive access matrix Z. The constraints enforce some of the invariants of MOP listed in Section 3. Equations 2.a and 2.b ensure model training isolation. Equations 2.c and 2.d ensure worker exclusive access. Equation 2.e ensures that training unit start times are non-negative and Equation 2.f ensures that C captures the time taken to complete all training units.

Given the above, a straightforward approach to scheduling is to use an MILP solver like Gurobi [25]. The start times X then yield the actual schedule. But our problem is essentially an instance of the classical open-shop scheduling problem, which is known to be `NP-Hard` [22]. Since $|S|$ can even be 100s, MILP solvers may be too slow (more in Section 5.4); thus, we explore alternative approaches.

5.2 Approximate Algorithm-based Scheduler

For many special cases, there are algorithms with good approximation guarantees that can even be optimal under some conditions. One such algorithm is "vector rearrangement" [19, 62]. It produces an optimal solution when $|S| \gg p$, which is possible in our setting.

The vector rearrangement based method depends on two values: L_{max} (see Equation 3), the maximum load on any worker; and T_{max} (see Equation 4), the maximum unit time of any training configuration in S.

$$L_{max} = \max_{j \in [1,\ldots,M]} \sum_{i=i}^{N} T_{i,j} \quad (3)$$

$$T_{max} = \max_{i \in [1,\ldots,N], j \in [1,\ldots,M]} T_{i,j} \quad (4)$$

If $L_{max} \geq (p^2 + p - 1) \cdot T_{max}$, then this algorithm's output is optimal. When there are lot of training configurations, the chance of the above constraint being satisfied is high, yielding us an optimal schedule. But if the condition is not met, the schedule produced yields a makespan $C \leq C^* + (p-1) \cdot T_{max}$, where C^* is the optimal makespan value. This algorithm scales to large $|S|$ and p because it runs in polynomial time in contrast to the MILP solver. For more details on this algorithm, we refer the interested reader to [19, 62].

5.3 Randomized Algorithm-based Scheduler

The approximate algorithm is complex to implement in some cases, and its optimality condition may be violated often. Thus, we now consider a much simpler scheduler based on *randomization*. This approach is simple to implement and offer much more flexibility (explained more later). Algorithm 1 presents our randomized scheduler.

Given S, create $Q = \{s_{i,j} : \forall i \in [1, ..., |S|], j \in [1, .., p]\}$, the set of all training units. Note that $s_{i,j}$ is the training unit of configuration i on worker j. Initialize the state of all models and workers to idle state. Then find an idle worker and schedule a random training unit from Q on it. This training unit must be such that its configuration is not scheduled on another worker and it corresponds to the data partition placed on that worker (Line 10). Then remove the chosen training unit from Q. Continue this process until no worker is idle and eventually, until Q is empty. After a worker completes training unit $s_{i,j}$ mark its model i and worker j as idle again as per Algorithm 2.

Algorithm 1 Randomized Scheduling

```
1: Input: S
2: Q = {s_{i,j} : ∀i ∈ [1,...,|S|], ∀j ∈ [1,...,p]}
3: worker_idle ← [true,...,true]
4: model_idle ← [true,...,true]
5: while not empty(Q) do
6:   for j ∈ [1,...,p] do
7:     if worker_idle[j] then
8:       Q ← shuffle(Q)
9:       for s_{i,j'} ∈ Q do
10:        if model_idle[i] and j' = j then
11:          Execute s_{i,j'} on worker j
12:          model_idle[i] ← false
13:          worker_idle[j] ← false
14:          remove(Q, s_{i,j'})
15:          break
16:  wait WAIT_TIME
```

Algorithm 2 When $s_{i,j}$ finishes on worker j

```
1: model_idle[i] ← true
2: worker_idle[j] ← true
```

5.4 Comparing Different Scheduling Methods

We use simulations to compare the efficiency and makespans yielded by the three alternative schedulers. The MILP and approximate algorithm are implemented using Gurobi. We set a maximum optimization time of 5min for tractability of experimentation. We compare the scheduling methods on 3 dimensions: 1) number of training configs (two values: 16 and 256), 2) number of workers (two values: 8 and 16), 3) homogeneity/heterogeneity of training configs and workers.

Sub-epoch training time (unit time) of a training config is directly proportional to the compute cost of the config and inversely proportional to compute capacity of the worker. For the homogeneous setting, we initialize all training config compute costs to be the same and also all worker compute capacities to be the same. For the heterogeneous setting, training config compute costs are randomly sampled (with replacement) from a set of popular deep CNNs (n=35) obtained from [2]. The costs vary from 360 MFLOPS to 21000 MFLOPS with a mean of 5939 MFLOPS and standard deviation of 5671 MFLOPS. Due to space constraints we provide these computational costs in the Appendix of our technical report [46]. For worker compute capacities, we randomly sample (with replacement) compute capacities from 4 popular Nvidia GPUs: Titan Xp (12.1 TFLOPS/s), K80 (5.6 TFLOPS/s), GTX 1080 (11.3 TFLOPS/s), and P100 (18.7 TFLOPS/s). For each setting, we report the average of 5 runs with different random seeds set to the scheduling algorithms and also the min and max of all 5 runs. All makespans reported are normalized by the randomized scheduler's makespan.

The MILP scheduler sometimes performs poorer than the other two because it has not converged to the optimal in the given time budget. The approximate scheduler performs poorly when both the configs and workers are heterogeneous. It is also slower than the randomized scheduler.

Overall, the randomized approach works surprisingly well on all aspects: near-optimal makespans with minimal variance across runs and very fast scheduling. We believe this interesting superiority of the randomized algorithm against the approximation algorithm is due to some fundamental characteristics of deep net model selection workloads, e.g., large number of configurations and relatively low differences in compute capacities. We leave a thorough theoretical analysis of the randomized algorithm to future work. Based on these results, we use the randomized approach as the default Scheduler in Cerebro.

5.5 Replica-Aware Scheduling

So far we assumed that a partition is available on only one worker. But some file systems (e.g., HDFS) often replicate data files, say, for reliability sake. We now exploit such replicas for more scheduling flexibility and faster plans.

The replica-aware scheduler requires an additional input: availability information of partitions on workers (an availability map). In replica-aware MOP, a training configuration need *not* visit all workers. This extension goes beyond

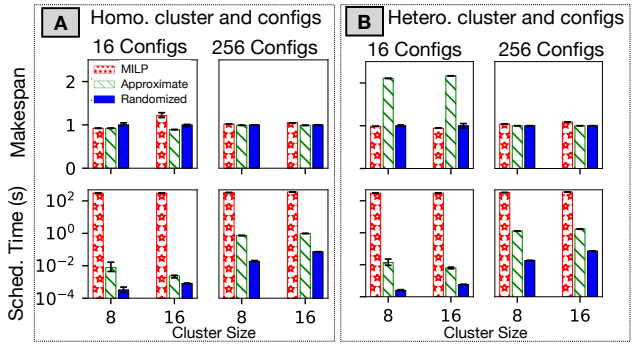


Figure 6: Scheduler runtimes and makespan of produced schedule in different settings. Makespans are normalized with respect to that of the randomized scheduler. (A) Homogeneous cluster and homogeneous training configs and (B) heterogeneous cluster and heterogeneous training configs.

open shop scheduling, but it is still `NP-Hard` because the open shop problem is a special case of this problem with a replication factor of one. We extended the MILP scheduler but it only got slower. So, we do not use it and skip its details. Modifying the approximate algorithm is also non-trivial because it is tightly coupled to the open shop problem; so, we skip that too. In contrast, the randomized scheduler can be easily extended for replica-aware scheduling. The only change needed to Algorithm 1 is in Line 10: instead of checking $j' = j$, consult the availability map to check if the relevant partition is available on that worker.

5.6 Fault Tolerance and Elasticity

We now explain how we make our randomized scheduler fault tolerant. Instead of just Q, we maintain two data structures Q and Q'. Q' is initialized to be empty. The process in Algorithm 1 continues until both Q and Q' are empty. When a training unit is scheduled, it will be removed from Q as before but now also *added* to Q'. It will be removed from Q' when it successfully completes its training on the assigned worker. But if the worker fails before the training unit finishes, it will be moved back from Q' to Q. If the data partitions present on the failed worker are also available elsewhere, the scheduler will successfully execute the corresponding training units on those workers at a future iteration of the loop in Algorithm 1.

Cerebro detects failures via the periodic heart-beat check between the scheduler and workers. Because the trained model states are always checkpointed between training units, they can be recovered and the failed training units can be restarted. Only the very last checkpointed model is needed for the failure recovery and others can be safely deleted for reclaiming storage. The same mechanism can be used to detect availability of new compute resources and support seamless scale-out elasticity in Cerebro.

5.7 Extension: Horovod Hybrid

Some AutoML procedures (e.g., Hyperband) start with large $|S|$ but then kill some non-promising configs after some epochs. So, only a few configs may train till convergence. This means at the later stages, we may encounter a situation where $|S|$ goes below p. In such cases, Cerebro can under-utilize the cluster. To overcome this limitation, we explored the possibility of doubly hybridizing MOP with data-parallelism by implementing a hybrid of Cerebro and Horovod. Just like Cerebro, Horovod is also equivalent to sequential SGD; so, the hybrid is reproducible. The basic idea is simple: divide the cluster into virtual sub-clusters and run Horovod within each sub-cluster and MOP across sub-clusters. Due to space constraints, we explain this hybrid architecture further in Appendix.

Table 4: Dataset details. *All numbers are after preprocessing and sampling of the datasets.

Dataset	On-disk size	Count	Format	Class
ImageNet	250 GB	1.2M	HDF5	1000
Criteo	400 GB	100M	TFRecords	Binary

6. EXPERIMENTAL EVALUATION

We empirically validate if Cerebro can improve throughput and efficiency of deep net model selection workloads and then drill-down into various aspects of Cerebro separately. We also perform experiments demonstrating Cerebro ability to support arbitrary AutoML procedures and experiments with Horovod hybrid.

Datasets. We use two large benchmark datasets: *ImageNet* [17] and *Criteo* [14]. *ImageNet* is a popular image classification benchmark dataset. We choose the 2012 version and reshape the images to 112×112 pixels[2]. *Criteo* is an ad click classification dataset with numeric and categorical features. It is shipped under sparse representation. We one-hot encode the categorical features and densify the data. Only a 2.5% random sample of the dataset is used[1]. Table 4. summarizes the dataset statistics.

Workloads. For our first end-to-end test, we use two different neural architectures and grid search for hyper-parameters, yielding 16 training configs for each dataset. Table 5 offers the details. We use Adam [34] as our SGD method. To demonstrate generality, we also present results for Hyperband and ASHA on Cerebro in Section 7.3.

Experimental Setup. We use two clusters: CPU-only and GPU-enabled, both on CloudLab [54]. Each cluster has 8 worker nodes and 1 master node. Each node in both clusters has two Intel Xeon 10-core 2.20 GHz CPUs, 192GB memory, 1TB HDD and 10 Gbps network. Each GPU cluster worker node has an extra Nvidia P100 GPU. All nodes run Ubuntu 16.04. We use TensorFlow v1.12.0 as Cerebro's underlying deep learning tool. For GPU nodes, we use CUDA version 9.0 and cuDNN version 7.4.2. Both datasets are randomly shuffled and split into 8 equi-sized partitions.

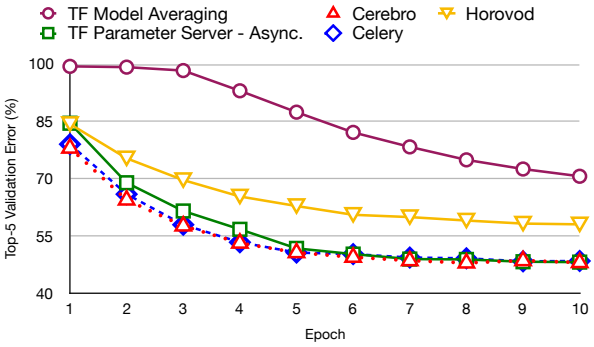
6.1 End-to-End Results

We compare Cerebro with 5 systems: 4 data-parallel–synchronous and asynchronous TensorFlow Parameter Server (PS), Horovod, BSP-style TensorFlow model averaging–and 1 task-parallel (Celery). For Celery, we replicate the entire datasets on each worker beforehand and stream them from disk, since they do not fit in memory.

[2]We made the decision so that the evaluations can complete in reasonable amount of time. This decision does not alter the takeaways from our experiments.

System	ImageNet			Criteo		
	Runtime (hrs)	GPU Utili. (%)	Storage Footprint (GB)	Runtime (hrs)	CPU Utili. (%)	Storage Footprint (GB)
TF PS - Async	19.00	8.6	250	28.80	6.9	400
Horovod	5.42	92.1	250	14.06	16.0	400
TF Model Averaging	1.97	72.1	250	3.84	52.2	400
Celery	1.72	82.4	2000	3.95	53.6	3200
Cerebro	1.77	79.8	250	3.40	51.9	400

(A) Per-epoch makespans and CPU/GPU utilization.



(B) Learning curves of the best model on *ImageNet*.

Figure 7: End-to-end results on *ImageNet* and *Criteo*. For Celery, we report the runtime corresponding to the lowest makespan schedule. Celery *ImageNet* per-epoch runtime can vary between 1.72-2.02 hours; for *Criteo* between 3.95-5.49 hours. Horovod uses GPU kernels for communication and hence high GPU utilization.

Table 5: Workloads.*architectures similar to VGG16 and ResNet50, respectively.†serialized sizes.

Dataset	Model arch.	Model size/MiB†	Batch size	Learning rate	Regularization	Epochs
ImageNet	{VGG16*, ResNet50*}	VGG16: 792, ResNet50: 293	{32, 256}	$\{10^{-4}, 10^{-6}\}$	$\{10^{-4}, 10^{-6}\}$	10
Criteo	3-layer NN, 1000+500 hidden units	179	{32, 64, 256, 512}	$\{10^{-3}, 10^{-4}\}$	$\{10^{-4}, 10^{-5}\}$	5

This is not an issue for runtimes, since for deep nets compute times vastly dominate I/O times and thus can be interleaved. We use built-in features available in TensorFlow to achieve this. For all other systems, including Cerebro, each worker node has a single data partition, which is in-memory. We do not include the time taken to copy data in to the workers in the end-to-end runtimes. For scheduling, Celery uses a FIFO queue and Cerebro uses the randomized scheduler. All other systems train models sequentially.

Figure 7 presents the results. We see that Cerebro significantly improves the efficiency and throughput of model selection. On *ImageNet*, Cerebro is over 10x faster than asynchronous PS, which has a GPU utilization as low as 8%! Synchronous PS was even slower. Cerebro is 3x faster than Horovod. Horovod has high GPU utilization because it also includes communication time (Horovod marks the GPU as busy during communication). Cerebro's runtime is comparable to model averaging, which is as expected. But note that model averaging converges poorly. Celery's runtime is dependent on the scheduling order and thus we report the runtime corresponding to the best schedule. For *ImageNet*, Celery's runtime is comparable to Cerebro but for *Criteo*, it takes significantly more time. This is due to a straggler issue caused by highly heterogeneous model configurations in *Criteo*. Cerebro's model hopping approach provides more flexibility to avoid such straggler issues. More details about this can be found in the appendix of our technical report [46]. Also, Celery has a highly bloated 8x memory/storage footprint.

Overall, Celery and Cerebro have the best learning curves, which are almost identical–this is also as expected because MOP ensures sequential equivalence for SGD, just like task-parallelism. Horovod converges slower due to its larger effective mini-batch size.

On *Criteo*, Cerebro is 14x faster than synchronous PS and 8x faster than asynchronous PS. Both versions of PS suffer have $<$ 7% CPU utilization. Cerebro is also 4x faster than Horovod. Cerebro's runtime is comparable to model averaging, with 50% CPU utilization. Celery is slightly slower than Cerebro due to minor skews in task assignments (one worker ran two of the most time-consuming models). All methods have almost indistinguishable convergence behavior on this dataset: each one reached 99% accuracy quickly, since the class label is skewed in this dataset. Overall, Cerebro is the *most resource-efficient* considering compute, memory/storage, and network holistically, while also offering the *best accuracy behavior* on par with task-parallelism.

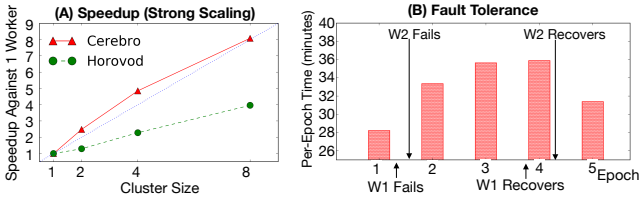


Figure 8: (A) Speedup (strong scaling). (B) Fault-tolerance.

6.2 Drill-down Experiments

Unless specified otherwise, we now show experiments on the GPU cluster, *ImageNet*, and a model selection workload of 8 configs (4 learning rates, 2 regularization values, and ResNet architectures) trained for 5 epochs. Each data partition is placed on only one worker.

Scalability. We study the speedups (strong scaling) of Cerebro and Horovod as we vary the cluster sizes. Figure 8(A) shows the speedups, defined as the workload completion time on multiple workers vs a single worker. Cerebro exhibits linear speedups due to MOP's marginal communication costs; in fact, it seems slightly super-linear here because the dataset fits entirely in cluster memory compared to the minor overhead of reading from disk on the

single worker. In contrast, Horovod exhibits substantially sub-linear speedups due to its much higher communication costs with multiple workers.

Fault Tolerance. We repeat our drill-down workload with a replication factor of 3, i.e., each data partition is available on 3 workers. We first inject two node failures and bring the workers back online later. Figure 8(B) shows the time taken for each epoch and the points where the workers failed and returned online. Overall, we see CEREBRO's replica-aware randomized scheduler can seamlessly execute the workload despite worker failures.

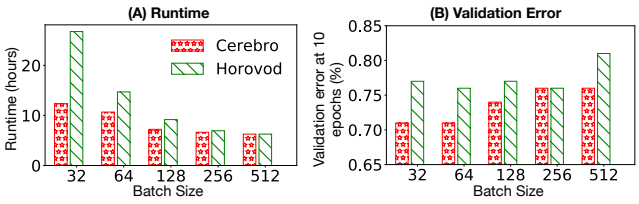


Figure 9: Effect of batch size on communication overheads and convergence efficiency. (A) Runtime against batch size. (B) The lowest validation error after 10 epochs against batch size.

Effect of Batch Size. We now evaluate the effect of training mini-batch size for CEREBRO and Horovod. We try 5 different batch sizes and report makespans and the validation error of the best model for each batch size after 10 epochs. Figure 9 presents the results. With batch size 32 Horovod is 2x slower than CEREBRO. However, as the batch size increases, the difference narrows since the relative communication overhead per epoch decreases. CEREBRO also runs faster with larger batch size due to better hardware utilization. The models converge slower as batch size increases. The best validation error is achieved by CEREBRO with a batch size of 32. With the same setting, Horovod's best validation error is higher than CEREBRO; this is because its effective batch size is 256 (32×8). Horovod's best validation error is closer to CEREBRO's at a batch size of 256. Overall, CEREBRO's efficiency is more stable to the batch size, since models hop per sub-epoch, not per mini-batch.

Network and Storage Efficiency. We study the tradeoff between redundant remote reads (wastes network) vs redundant data copies across workers (wastes memory/storage). Task parallelism forces users to either duplicate the dataset to all workers or store it in a common repository/distributed filesystem and read remotely at each epoch. CEREBRO can avoid both forms of resource wastage. We assume the whole dataset cannot fit on single-node memory. We compare CEREBRO and Celery in the following 2 settings:

Reading from remote storage (e.g., S3). In this setting, Celery reads data from a remote storage repeatedly each epoch. For CEREBRO we assign each worker with a data partition, which is read remotely once and cached into the memory. We change the data scale to evaluate effects on the makespan and the amount of remote reads per worker. Figure 10 shows the results. Celery takes slightly more time than CEREBRO due to the overhead of remote reads. The most significant advantage of CEREBRO is its network bandwidth cost, which is over 10x lower than Celery's. After

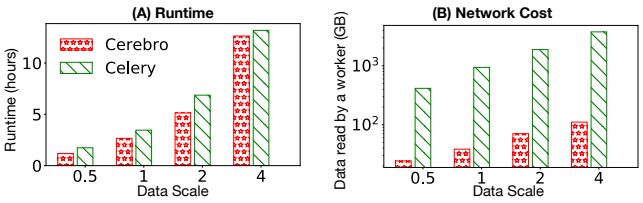


Figure 10: Reading data from remote storage.

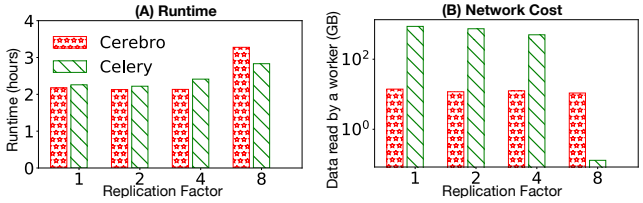


Figure 11: Reading data from distributed storage.

the initial read, CEREBRO only communicates trained models during the model hopping process. In situations where remote data reads and network communications are not free (e.g., cloud providers) Celery will incur higher monetary costs compared to CEREBRO. These results show it is perhaps better to partition the dataset on S3, cache partitions on workers on first read, and then run CEREBRO instead of using Celery and reading the full dataset from S3 at each epoch to avoid copying the whole dataset across workers.

Reading from distributed storage (e.g., HDFS). In this setting, the dataset is partitioned, replicated, and stored on 8 workers. We then load all local data partitions into each worker's memory. Celery performs remote reads for non-local partitions. We vary the replication factor and study its effect on the makespan and the amount of remote reads per worker. Figure 10 presents the results. For replication factors 1 (no replication), 2, and 4, CEREBRO incurs 100x less network usage and is slightly faster than Celery. But at a replication factor of 8 (i.e., entire dataset copied locally), CEREBRO is slightly slower due to the overhead of model hops. For the same reason, CEREBRO incurs marginal network usage, while Celery has almost no network usage other than control actions. Note that the higher the replication factor needed for Celery, the more local memory/storage is wasted due to redundant data. Overall, CEREBRO offers the best overall resource efficiency–compute, memory/storage, and network put together–for deep net model selection.

Experiments with Horovod Hybrid. Our experiment with the Horovod Hybrid gave an anti-climactic result: the intrinsic network overheads of Horovod meant the hybrid is often slower than regular CEREBRO with some workers being idle! We realized that mitigating this issue requires more careful data repartitioning. We deemed this complexity as perhaps not worth it. Instead, we propose a simpler resolution: if $|S|$ falls below p but above $p/2$, use CEREBRO; if $|S|$ falls below $p/2$, just switch to Horovod. This switch incurs no extra overhead. Due to space constraints, we skip the details here and explain this experiment further in Appendix.

6.3 Experiments with AutoML Procedures

We experiment with two popular AutoML procedures: HyperOpt [6] and ASHA [39]. For HyperOpt, we compare

Table 6: Parameter grid used to randomly sample configuration for Section 6.3.

	Values sampled from
Model	[Resnet18, Resnet34]
Learning rate	$[10^{-5}, \ldots, 10^{-1}]$
Weight decay coefficient	$[10^{-5}, \ldots, 10^{-1}]$
Batch size	$[16, \ldots, 256]$

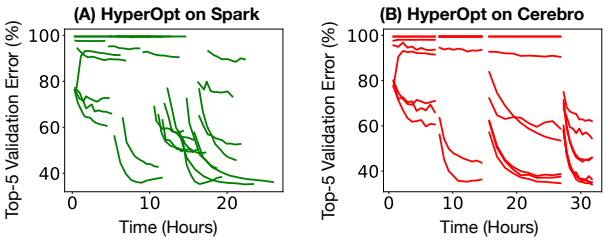


Figure 12: HyperOpt learning curves by time.

Cerebro and Spark as the execution backends. Spark is one of the backends supported natively by HyperOpt. However, it uses Spark to distribute the configurations and not the data. For ASHA, we compare Cerebro and Celery as the execution backends. We use *ImageNet*, GPU cluster, and PyTorch. Training configs are sampled from the grid shown in Table 6. For Cerebro each data partition is only available on one worker; for Spark and Celery the dataset is fully replicated.

Both HyperOpt and ASHA continue to explore different configs until a maximum resource limit is reached. For HyperOpt this limit is specified as a maximum number of configs; for ASHA as a maximum wall-clock time. During the exploration HyperOpt uses Bayesian sampling to generate new configs; ASHA uses random sampling. For both methods, the set of generated configs are dependent on the wall-clock completion order of configs across task-parallel workers. Thus, it is impossible to exactly replicate a run of HyperOpt or ASHA on task-parallelism in Cerebro. However, we can indeed emulate HyperOpt and ASHA on Cerebro by making the number of simultaneously trained configs ($|S|$) equal to the number of workers (p) and without making any changes to Cerebro.

HyperOpt. We run an experiment using HyperOpt with a max config budget of 32. We train each config for 10 epochs. With this configuration, HyperOpt on Cerebro (resp. Spark) took 31.8 (resp. 25.9) hours to complete. Figure 12 shows all learning curves. We found that the slightly higher (23% higher) runtime of Cerebro is mainly due to the low number of simultaneously trained configs ($|S| = 8$) which causes some workers to become idle due to not having enough configs to train. However, such issues can be mitigated by increasing the number of simultaneously trained configs. Though the individual configs are not comparable across the two systems, the best errors achieved are close (34.1% on Cerebro; 33.2% on Celery).

ASHA. We run an experiment using ASHA with a max epoch budget (R) of 9 epochs, a selection fraction (η) of 3, and a time limit of 24hr. With this configurations ASHA can train the models up to a maximum of 13 epochs at

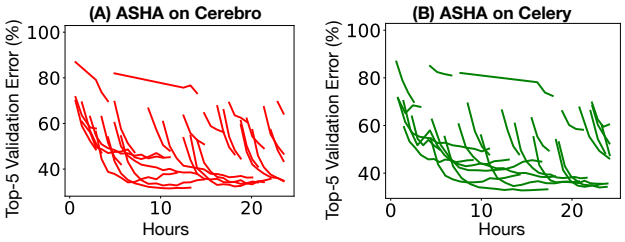


Figure 13: ASHA learning curves by time.

three stages: 1 epoch, 3 epochs, and 9 epochs. Only the most promising configurations are trained for a higher number of epochs. In the given time limit, ASHA on Cerebro (resp. Celery) explored 83 (resp. 67) configs. Figure 13 shows all learning curves. Similar to HyperOpt, even though the individual configs are not comparable across the two systems, the best errors achieved are close (31.9% on Cerebro; 33.2% on Celery). More details about this experiment and experiments with another related AutoML procedure called HyperBand can be found in the appendix of our technical report [46].

7. DISCUSSION AND LIMITATIONS

We discuss extensions and limitations of our work.

Integration into Other Systems. MOP's generality makes it amenable to emulation on top of BSP data-parallel systems such as parallel RDBMSs and dataflow systems. Pivotal/VMWare has recently collaborated with us to integrate MOP into Greenplum by extending the MADlib library [43,59] for running TensorFlow on Greenplum-resident data. Greenplums's customers are interested in this integration for enterprise ML use cases including language processing, image recognition, and fraud detection. We have also integrated Cerebro into Apache Spark [16]. Cerebro's Spark implementation can be used to deploy MOP on existing cluster resource managers such as Yarn and Mesos. Alternatively, one can also develop Cerebro as a standalone application which can be executed directly on cluster resource managers by wrapping the MOP scheduler inside the application master and workers as containers. We leave such extensions to future work.

Applicability to Other ML Model Families. We primarily focused on deep learning models due to their high sensitivity to the chosen hyperparameters and model architecture. However, our method can be directly applied to model selection of other ML model families, that are trained using SGD. These include linear/logistic regression, support vector machines, low-rank matrix factorization, and conditional random fields. Cerebro's high memory efficiency enables users to store the entire datasets in memory, which can significantly reduce runtimes in many of these I/O-bound workloads. We leave the experimental analysis of these ML model families to future work.

Model Parallelism and Batching. Cerebro currently does not support model parallelism (for models larger than single-node memory) and batching (i.e., running multiple models on a worker at a time). But nothing in Cerebro makes it impossible to remove these limitations. For instance, model parallelism can be supported with the no-

tion of virtual nodes composed of multiple physical nodes that together hold an ultra-large model. Model batching can be supported by having multiple virtual nodes mapped to a physical node. We leave such extensions to future work.

8. RELATED WORK

Systems for Model Selection. Google Vizier [21], Ray Tune [42], Dask-Hyperband [57], SparkDL [15], and Spark-Hyperopt [28] are also systems for model selection. Vizier, Ray, and Dask-Hyperband are pure task-parallel systems that provide implementations of some AutoML procedures. SparkDL and Spark-Hyperopt use Spark underneath but only for distributing the configs, not the data–they replicate the full dataset to each worker like task-parallelism. CEREBRO offers higher overall resource efficiency compared to this prior landscape of pure task- or pure data-parallelism.

Hybrid Parallelism in ML Systems. MOP is inspired by the classical idea of process migration in OS multiprocessing [3]. We bring this idea to the multi-node data-partitioned regime. This general idea has been applied before in limited contexts in ML systems. The closest to our work is [13], which proposes a hybrid scheme for training many homogeneous CNNs for images on a homogeneous GPU cluster. They propose a ring topology to migrate models, resembling a restricted form of MOP. But their strong homogeneity assumptions can cause stalls in general model selection workloads, e.g., due to heterogeneous neural architectures or clusters. In contrast, our work approaches this problem from first principles and formalizes it as an instance of open shop scheduling. This powerful abstraction lets CEREBRO support arbitrary deep nets and data types, as well as heterogeneous neural architectures and clusters. It also lets CEREBRO offer other crucial systems capabilities: replication, fault tolerance, elasticity, and arbitrary AutoML procedures unlike prior work. SystemML also supports a hybrid of task- and data-parallelism for linear algebra-based classical ML [9]. They enable task-parallelism on top of data-parallel MapReduce and focus on better plan generation to exploit multi-cores and clusters. CEREBRO is complementary because we focus on deep nets and SGD's data access pattern, not linear algebra-based classical ML.

AutoML Procedures. AutoML procedures such as Hyperband [38] and PBT [29] are orthogonal to our work and exist at a higher abstraction level. They fit a common template of per-epoch scheduling in CEREBRO. While ASHA [39] does not fit this template, CEREBRO can still emulate them well and offer similar accuracy as pure task-parallelism without any modifications to CEREBRO. Bayesian optimization is another popular line of AutoML metaheuristics in the ML world. Some such techniques have a high degree of parallelism in searching configs (e.g., Hyperopt [6]); CEREBRO supports these variants. Some others perform sequential search, leading to a low degree to parallelism (e.g., [5, 35]); these may not be a fit for CEREBRO.

Cluster Scheduling for Deep Learning. Gandiva [63], Tiresias [24], and SLAQ [66] are cluster scheduling system for deep learning. Their focus is on lower-level primitives such as resource allocation and intra-server locality awareness for optimizing the average job completion time. CEREBRO is complementary to them as it operates at a higher abstraction level and targets the throughput of model selection. How compute hardware is allocated is outside our scope. There is a long line of work on general job scheduling algorithms in the operations research and systems literatures [12, 20, 26]. Our goal is *not* to create new scheduling algorithms but to apply known algorithms to a new ML systems setting based on MOP.

Systems for Distributed SGD. There is much prior work on systems to reduce the latency of distributed SGD. Most of them focus on optimizing centralized fine-grained SGD [27, 31, 32, 50, 53, 65, 67, 68] and/or decentralized fine-grained SGD [41, 50, 61, 64]. Such systems are complementary to our work, as they improve parallelism for training one model at a time, while our focus is on model selection. As we showed, such systems have higher communication complexity (and thus, higher runtimes) than MOP in our setting. Also, since CEREBRO performs logically sequential SGD, it ensures theoretically best convergence efficiency. CROSSBOW [36] proposes a novel variant of model averaging for single-server multi-GPU environment to overcome the convergence issues of naive model averaging. But it too is complementary to our work, since it too focuses on training one model at a time. Overall, our work breaks the dichotomy between such data-parallel approaches and task-parallel approaches, thus offering better overall resource efficiency.

System Optimizations. Another recent line of work optimizes the deep learning tools' internals, including better pipelining of computation and communication [47], better compilation techniques [8, 30] and model batching [48]. All these are complementary to CEREBRO, since they optimize low-level operator execution of a neural computational graph. MOP is general enough to allow CEREBRO to be hybridized with such ideas.

9. CONCLUSIONS AND FUTURE WORK

Simplicity that still achieves maximal functionality and efficiency is a paragon of virtue in real-world systems. We present a simple but novel and highly general form of parallel SGD execution, MOP, that raises the resource efficiency of deep net model selection without sacrificing accuracy or reproducibility. MOP is also simple to implement, which we demonstrate by building CEREBRO, a fault-tolerant deep net model selection system that supports multiple popular deep learning tools and model selection procedures. Experiments with large ML benchmark datasets confirm the benefits of CEREBRO. As for future work, we plan to hybridize MOP with model parallelism and batching and also support more complex model selection scenarios such as transfer learning.

10. REFERENCES

[1] Script for Tensorflow Model Averaging, Accessed January 31, 2020. `https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/utils/avg_checkpoints.py`.

[2] S. Albanie. Memory Consumption and FLOP Count Estimates for Convnets, Accessed January 31, 2020. `https://github.com/albanie/convnet-burden`.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[4] Y. Bengio. Rmsprop and equilibrated adaptive learning rates for nonconvex optimization. *corr abs/1502.04390*, 2015.

[5] J. Bergstra, R. Bardenet, B. Kgl, and Y. Bengio. Algorithms for hyper-parameter optimization. 12 2011.

[6] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[7] D. P. Bertsekas. A new class of incremental gradient methods for least squares problems. *SIAM J. on Optimization*, 7(4):913–926, Apr. 1997.

[8] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On optimizing operator fusion plans for large-scale machine learning in systemml. *Proc. VLDB Endow.*, 11(12):17551768, Aug. 2018.

[9] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *Proc. VLDB Endow.*, 7(7):553564, 2014.

[10] L. Bottou. Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. In *Proceedings of the Symposium on Learning and Data Science*, 2009.

[11] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[12] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 3rd edition, 2001.

[13] C.-F. R. Chen, G. G. C. Lee, Y. Xia, W. S. Lin, T. Suzumura, and C.-Y. Lin. Efficient multi-training framework of image deep learning on gpu cluster. In *2015 IEEE International Symposium on Multimedia (ISM)*, pages 489–494. IEEE, 2015.

[14] CriteoLabs. Kaggle Contest Dataset Is Now Available for Academic Use!, Accessed January 31, 2020. `https://ailab.criteo.com/category/dataset`.

[15] databricks. Deep Learning Pipelines for Apache Spark, Accessed January 31, 2020. `https://github.com/databricks/spark-deep-learning`.

[16] Databricks. Resource-efficient Deep Learning Model Selection on Apache Spark, Accessed May 30, 2020. `https://bit.ly/3esN3JT`.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A Large-scale Hierarchical Image Database. In *CVPR*, pages 248–255. IEEE, 2009.

[18] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for In-RDBMS Analytics. In *SIGMOD*, 2012.

[19] T. Fiala. An Algorithm for the Open-shop Problem. *Mathematics of Operations Research*, 8(1):100–109, 1983.

[20] J. V. Gautam, H. B. Prajapati, V. K. Dabhi, and S. Chaudhary. A survey on job scheduling algorithms in big data processing. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–11, March 2015.

[21] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[22] T. Gonzalez and S. Sahni. Open Shop Scheduling to Minimize Finish Time. *JACM*, 1976.

[23] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT press, 2016.

[24] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[25] Gurobi. Gurobi Optimization, Accessed January 31, 2020. `https://www.gurobi.com`.

[26] W. Herroelen, B. D. Reyck, and E. Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4), 1998.

[27] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng. Flexps: Flexible parallelism control in parameter server architecture. *Proc. VLDB Endow.*, 11(5):566579, 2018.

[28] hyperopt. Scaling out search with Apache Spark, Accessed January 31, 2020. `http://hyperopt.github.io/hyperopt/scaleout/spark/`.

[29] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu. Population Based Training of Neural Networks. *arXiv preprint arXiv:1711.09846*, 2017.

[30] Z. Jia, M. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *SYSML 2019*, 2019.

[31] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD 17, page 463478, 2017.

[32] J. Jiang, F. Fu, T. Yang, and B. Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD 18, page 12691284, 2018.

[33] T.-Y. Kim and S.-B. Cho. Predicting residential energy consumption using cnn-lstm neural networks. *Energy*, 182:72 – 81, 2019.

[34] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[35] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS 2017)*, volume 54 of *Proceedings of Machine Learning Research*, pages 528–536. PMLR, Apr. 2017.

[36] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *Proc. VLDB Endow.*, 12(11):13991412, 2019.

[37] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: the Next Frontier of Advanced Analytics. *SIGMOD Record*, 2016.

[38] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[39] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

[40] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014.

[41] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous decentralized parallel stochastic gradient descent. *arXiv preprint arXiv:1710.06952*, 2017.

[42] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[43] MADLib. User Documentation for Apache MADlib, Accessed May 30, 2020. `https://bit.ly/3epbEyS`.

[44] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[45] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: Efficient and reproducible model selection on deep learning systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, 2019.

[46] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. `https://adalabucsd.github.io/papers/TR_2020_Cerebro.pdf`, 2020. [Tech report].

[47] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. Devanur, G. Granger, P. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP 2019*, 2019.

[48] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NIPS Workshop on Systems for Machine Learning*, 2018.

[49] S. L. Oh, E. Y. Ng, R. S. Tan, and U. R. Acharya. Automated diagnosis of arrhythmia using combination of cnn and lstm techniques with variable length heart beats. *Computers in Biology and Medicine*, 102:278 – 287, 2018.

[50] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. Tung, Y. Wang, and et al. Singa: A distributed deep learning platform. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM 15, page 685688, 2015.

[51] T. OMalley. Hyperparameter tuning with Keras Tuner, Accessed January 31, 2020. `https://blog.tensorflow.org/2020/01/hyperparameter-tuning-with-keras-tuner.html?linkId=81371017`.

[52] S. Pafka. Big RAM is Eating Big Data - Size of Datasets Used for Analytics, Accessed January 31, 2020. `https://www.kdnuggets.com/2015/11/big-ram-big-data-size-datasets.html`.

[53] C. Qin, M. Torres, and F. Rusu. Scalable asynchronous gradient descent optimization for out-of-core models. *Proc. VLDB Endow.*, 10(10):986997, 2017.

[54] R. Ricci, E. Eide, and CloudLabTeam. Introducing Cloudlab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[55] A. Sergeev and M. D. Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *arXiv preprint arXiv:1802.05799*, 2018.

[56] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: from Theory to Algorithms*. Cambridge university press, 2014.

[57] S. Sievert, T. Augspurger, and M. Rocklin. Better and faster hyperparameter optimization with dask. 2019.

[58] H. Su and H. Chen. Experiments on Parallel Training of Deep Neural Network using Model Averaging. *CoRR*, abs/1507.01239, 2015.

[59] VMWare. Model Selection for Deep Neural Networks on Greenplum Database, Accessed May 30, 2020. `https://bit.ly/2AaQLc2`.

[60] P. Warden. The Machine Learning Reproducibility Crisis, Accessed January 31, 2020. `https://petewarden.com/2018/03/19/the-machine-learning-reproducibility-crisis`.

[61] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC 16, page 8497, 2016.

[62] G. J. Woeginger. The Open Shop Scheduling Problem. In *STACS*, 2018.

[63] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[64] P. Xie, J. K. Kim, Q. Ho, Y. Yu, and E. Xing. Orpheus: Efficient distributed machine learning via system and algorithm co-design. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC 18, page 113, 2018.

[65] B. Yuan, A. Kyrillidis, and C. Jermaine. Distributed learning of deep neural networks using independent subnet training. *ArXiv*, abs/1910.02120, 2019.

[66] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC 17, page 390404, 2017.

[67] Z. Zhang, B. Cui, Y. Shao, L. Yu, J. Jiang, and X. Miao. Ps2: Parameter server on spark. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD 19, page 376388, 2019.

[68] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao. Mariana: Tencent deep learning platform and its applications. 7:1772–1777, 01 2014.

APPENDIX

A. CEREBRO API USAGE EXAMPLE

In this Section, we present a detailed example on how the CEREBRO API can be used to perform the *ImageNet* model selection workload explained in Section 4.1.

Before invoking the model selection workload users have to first register workers and data. This can be done as per the API methods shown in Listing 1 and Listing 2.

Listing 1: Registering Workers

```
#### API method to register workers ###
# worker_id:     Id of the worker
# ip       :     worker IP
#
# Example usage:
#   register_worker(0, 10.0.0.1)
#   register_worker(1, 10.0.0.2)
#   ....
#   register_worker(7, 10.0.0.8)
########################################
register_worker(worker_id, ip)
```

Listing 2: Registering Data

```
## API method to register a dataset ###
# name           : Name of the dataset
# num_partitions : # of partitions
#
# Example usage:
#       register_dataset(ImageNet, 8)
########################################
register_dataset(ImageNet, 8)

## API method to register partition ###
##          availability              ###
# dataset_name  : Name of the dataset
# data_type     : train or eval
# partition_id  : Id of the partition
# worker        : Id of the worker
# file_path     : file_path on the
#                 worker
#
# register_partition(ImageNet, train,
#         0,
#         0, /data/imagenet/train_0)
########################################
register_partition(dataset_name,
        data_type,
        partition_id, worker,
        file_path)
```

Next, users need to define the initial set of training configurations as shown in Listing 3.

Listing 3: Initial Training Configurations

```
S = []
for batch_size in [64, 128]:
  for lr in [1e-4, 1e-5]:
    for reg in [1e-4, 1e-5]:
      for model in [ResNet, VGG]:
        config = {
          batch_size: batch_size,
          learn_rate: lr,
          reg: reg,
          model: model
        }
        S.append(config)
```

Users also need to define three functions: *input_fn*, *model_fn*, and *train_fn*. *input_fn* as shown in Listing 4, takes in the file path of a partition, performs preprocessing, and returns in-memory data objects. Inside the *input_fn* users are free to use their preferred libraries and tools provided they are already installed on the worker machines. These in-memory data objects are then cached in the worker's memory for later usage.

Listing 4: *input_fn*

```
#### User defined input function ######
# file_path :   File path of a local
#               data partition
#
# Example usage:
#    processed_data = input_fn(file_path)
########################################
def input_fn(file_path):
        data = read_file(file_path)
        processed_data = preprocess(data)
        return processed_data
```

After the data is read into the worker's memory, CEREBRO then launches the model selection workload. This is done by launching training units on worker machines. For this CEREBRO first invokes the user defined *model_fn*. As shown in Listing 5, it takes in the training configuration as input and initializes the model architecture and training optimizer based on the configuration parameters. Users are free to use their preferred tool for defining the model architecture and the optimizer. After invoking the *model_fn*, CEREBRO injects a checkpoint restore operation to restore the model and optimizer state from the previous checkpointed state.

Listing 5: *input_fn*

```
#### User defined model function ######
# config : Training config.
#
# Example usage:
#       model, opt = model_fn(config)
########################################
def model_fn(config):
    if config[model] == VGG:
        model = VGG()
    else:
        model = ResNet()

    opt = Adam(lr=config[learn_rate])
    return model, opt
```

After restoring the state of the model and the optimizer, CEREBRO then invokes the user provided *train_fn* to perform one sub-epoch of training. As shown in Listing 5, it takes in the data, model, optimizer, and training configuration as input and returns convergence metrics. Training abstractions used by different deep learning tools are different and this function abstracts it out from the CEREBRO system. After the *train_fn* is complete the state of the model and the optimizer is checkpointed again.

Listing 6: *input_fn*

```
#### User defined train function ######
# data      : Preprocessed data
# model     : Deep learning model
# optimizer : Training optimizer
# config    : Train config.
#
# Example usage:
#       loss = train_fn(data, model,
#                       optimizer, config)
########################################
def train_fn(data, model, optimizer, config):

    X, Y = create_batches(data,
                config[batch_size])

    losses = []
    for batch_x, batch_y in (X,Y):
        loss = train(model, opt,
            [batch_x, batch_y])
        losses.append(loss)

    return reduce_sum(losses)
```

For evaluating the models, we assume the evaluation dataset is also partitioned and perform the same process. We mark the model parameters as non-trainable before passing it to the *train_fn*. After a single epoch of training and evaluation is done, CEREBRO aggregates the convergence metrics from all training units from the same configuration to derive the epoch-level convergence metrics. Convergence metrics are stored in a configuration state object which keeps track of the training process of each training configuration. At the end of an epoch, configuration state objects are passed to the *automl_mthd* implementation for evaluation. It returns a set of configurations that needs to be stopped and/or the set of new configurations to start. For example in the case of performing Grid Search for 10 epochs, the *automl_mthd* will simply check whether an initial configuration has been trained for 10 epochs, and if so it will mark it for stopping.

B. SECTION 5.4 CNN COMPUTE COSTS

Table 7 lists the computational costs of the CNNs used for the simulation experiment which compares different scheduling methods. These costs were obtained from a publicly available benchmark[3].

Model	FLOPs
AlexNet	727 MFLOPs
CaffeNet	724 MFLOPs
SqueezeNet1-0	837 MFLOPs
SqueezeNet1-1	360 MFLOPs
VGG-f	727 MFLOPs
VGG-m	2 GFLOPs
VGG-s	3 GFLOPs
VGG-m-2048	2 GFLOPs
VGG-m-1024	2 GFLOPs
VGG-m-128	2 GFLOPs
VGG-vd-16-atrous	16 GFLOPs
VGG-vd-16	16 GFLOPs
VGG-vd-19	20 GFLOPs
GoogleNet	2 GFLOPs
ResNet18	2 GFLOPs
ResNet34	4 GFLOPs
ResNet50	4 GFLOPs
ResNet101	8 GFLOPs
ResNet152	11 GFLOPs
ResNext-50-32x4d	4 GFLOPs
ResNext-101-32x4d	8 GFLOPs
ResNext-101-64x4d	16 GFLOPs
Inception-V3	6 GFLOPs
SE-ResNet-50	4 GFLOPs
SE-ResNet-101	8 GFLOPs
SE-ResNet-152	11 GFLOPs
SE-ResNeXt-50-32x4d	4 GFLOPs
SE-ResNeXt-101-32x4d	8 GFLOPs
SENet	21 GFLOPs
SE-BN-Inception	2 GFLOPs
DenseNet121	3 GFLOPs
DenseNet161	8 GFLOPs
DenseNet169	3 GFLOPs
DenseNet201	4 GFLOPs
MobileNet	579 MFLOPs

Table 7: Computation costs of the CNNs used for the simulation experiment comparing different scheduling methods.

C. STRAGGLER ISSUE IN CELERY

One potential issue that could impact task-parallel systems' performance is load balancing. Given the large variance of runtimes for deep-nets training, the scheduling generated by Celery could lead to severe straggler issues that impairs the end-to-end runtime of the whole workload. On the other hand, CEREBRO suffers far less from this problem because it operates on a finer granularity; our tasks are chunked into sup-epochs and hence it is less likely for long-running stragglers to appear.

We take the Criteo tests showed in Section 6.1 as example. Without any prior or domain knowledge, it is impossible to know the runtime of each task before-hand and therefore Celery could schedule a plan like Figure 14. The execution suffers from the straggler config#0 and needs 27.4 hrs to run.

However, if with a proper estimation/profiling of the runtimes/workloads, it is possible to fix this straggler issue with a carefully curated schedule as showed in Figure 15. This schedule drastically reduces the runtime to 19.7 hrs.

[3] https://github.com/albanie/convnet-burden

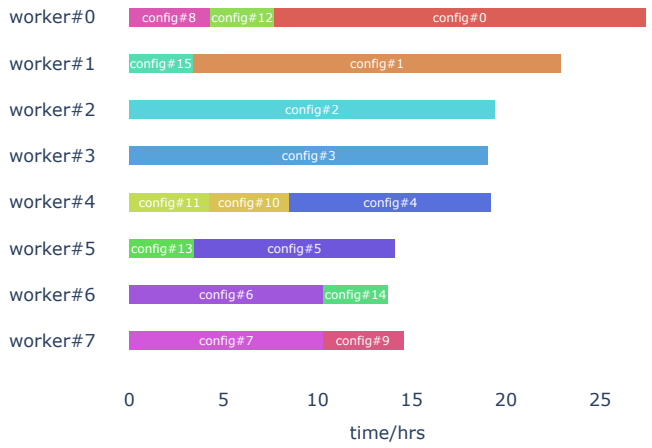


Figure 14: An unbalanced work schedule generated by Celery for Criteo tests.

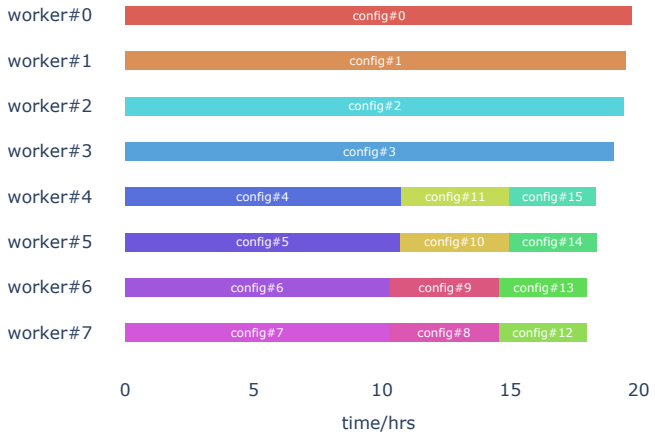


Figure 15: Best possible work schedule with Celery for Criteo tests.

In Section 6.1, we decided to show the runtime with the best-possible scheduling for Celery, as we do not wish to unfairly punish the adversarial systems, and load balancing/runtime estimation of deep learning workloads are out of the scope of this paper. We believe these decisions can ultimately help the reader focus on the benefits and advantages of our system.

D. EXTENSION: HOROVOD HYBRID

A typical model selection workflow begins with a large number of model configs, and narrows down the scope gradually over epochs, ending up with a handful of model configs to train till convergence. It means that at the later stages, we may encounter scenarios where the number of model configs, $|S|$, can be smaller than the number of workers, p. In these scenarios CEREBRO can lead to under-utilization of the cluster.

We mitigate this issue by doubly hybridizing MOP with data parallelism. To this end, we implement a hybrid version of CEREBRO with Horovod we call Horovod hybrid. Just like CEREBRO, Horovod is also equivalent to sequential SGD concerning convergence behavior. Therefore the hybrid of them will remain reproducible.

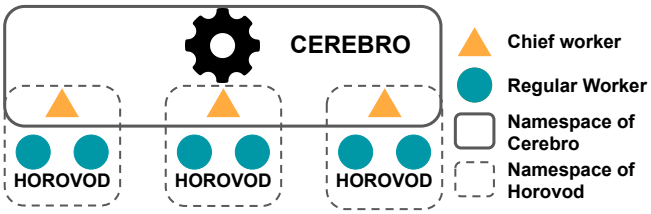


Figure 16: The architecture of Horovod Hybrid. Within different namespaces, we run Cerebro and Horovod, respectively. The chief workers, acting as Cerebro workers, are responsible for driving Horovod tasks and handling the communication between the two systems. In the figure, we show a 9-node cluster with 3 model configs to train.

Figure 16 summarizes the architecture of Horovod Hybrid, where instead of workers, we have worker groups. Inside each worker group, we run a data-parallel Horovod task. Then after each worker group finishes their assigned task, we hop the trained models just as the regular CEREBRO. We assume there are more workers than model configs. We create an equal number of groups for the number of configs. Workers are placed into these groups evenly.

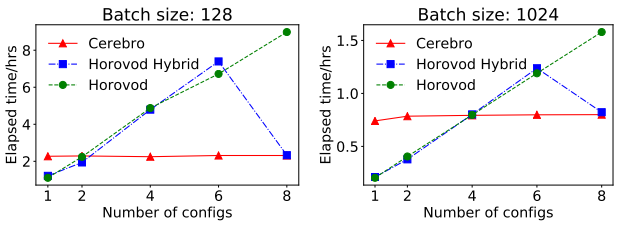


Figure 17: Performance tests of Horovod Hybrid with varying batch size and $|S|$ on 8-node cluster. Configs: same model as in Section 4 Table 5, learning rates drawn from $\{10^{-3}, 10^{-4}, 5 \times 10^{-5}, 10^{-5}\}$, weight decays drawn from $\{10^{-4}, 10^{-5}\}$. We test on 2 different batch sizes, respectively.

We now explore the possibility of hybridizing MOP with Horovod to better utilizer resources in $|S| < p$ regime. For this we run an experiment using *Criteo* on the CPU cluster with varying number of configs ($|S|$) and batch sizes. We compare three different methods: (1) Horovod, (2) MOP, and (3) Horovod Hybrid. Figure 17 shows the results.

Horovod's runtime grows linearly with more configs, but CEREBRO is constant. For instance at batch size 128 and $|S| = 2$, CEREBRO matches Horovod's performance. This is because CEREBRO's communication costs are negligible. For the Horovod Hybrid the runtimes are comparable to Horovod, except when $|S| = p$ it reduces to CEREBRO. This is because Horovod Hybrid is bottlenecked by Horovod's network overheads; mitigating this issue will require careful data re-partitioning, which we leave to future work. It is interesting that even with underutilization CEREBRO can still outperform Horovod in most scenarios. Depending on $|S|$ and batch size, there is a cross-over point when the

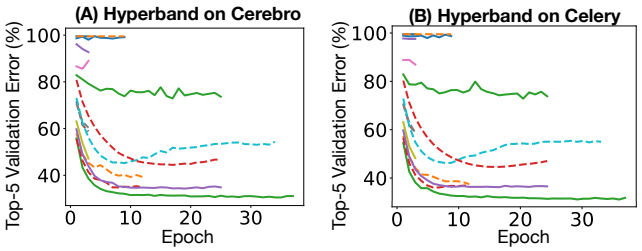


Figure 18: Hyperband learning curves by epochs.

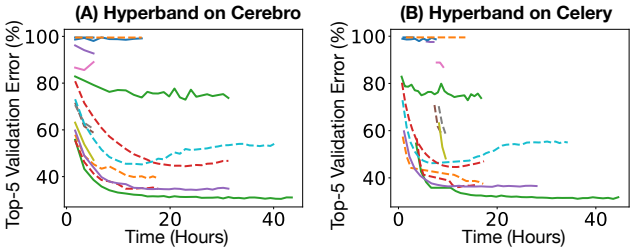


Figure 19: Hyperband learning curves by time.

three methods meet. Typically when $|S| \ll p$, Horovod and Horovod Hybrid are faster as CEREBRO is heavily underutilizing the workers. We heuristically choose $p/2$ as the dividing point: still run CEREBRO if $p/2 < |S|$, otherwise just run Horovod. Overall, the current Horovod Hybrid does not provide much benefit over Horovod as it mainly optimizes Horovod for its latency part of the communication cost, which turns out to be marginal.

E. AUTOML PROCEDURES

E.1 Experiments with HyperBand

We compare CEREBRO and Celery for executing Hyperband [38]; a popular AutoML search procedure. We use *ImageNet*, GPU cluster, and PyTorch. Training configs are randomly sampled from the grid shown in Table 6. For CEREBRO each data partition is only available on one worker; for Celery the dataset is fully replicated.

Hyperband combines random search with early stopping. It starts with a fixed set of model configs and trains them for a given number of epochs in the first "rung." After completion, it picks a subset of the best models and promotes them to the next rung; this is repeated several times until a max epoch budget is hit. We run an experiment with a max resource budget (R) of 25 epochs and a downsampling rate (η) of 3, two parameters from the Hyperband paper. Figure 18 compares the learning curves of the configs run by Hyperband on CEREBRO and Celery.

We see that CEREBRO and Celery have almost indistinguishable convergence behaviors, validating our claim that MOP benefits from SGD's robustness to random data ordering. As Figure 19 shows, both systems have similar completion times (42.1hr for Celery; 43.5hr for CEREBRO). Some configs finish sooner on Celery than their counterparts on CEREBRO. This is because CEREBRO's per-epoch scheduling template enforces all configs to be scheduled for the same epoch. But in Celery, configs in their last rung can finish earlier without waiting for other configs.

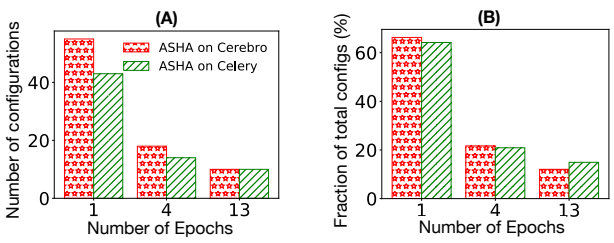


Figure 20: Number of configs vs. the amount of epochs they were run for by. (A) Count of configs and (B) Fraction of total config count.

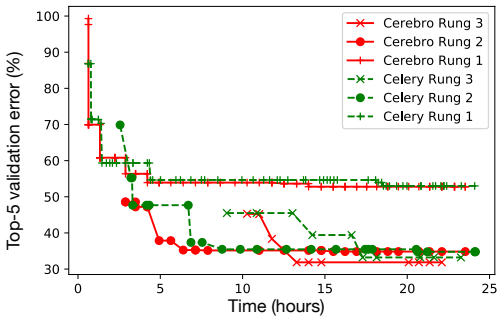


Figure 21: Best validation error for each rung of ASHA.

E.2 Experiments with ASHA

ASHA combines random search with early stopping. It starts with a set of model configs and trains them for a fixed number of epochs in the first "rung." After a training config finishes its current rung, it is assigned to a pool of completed configs for that rung. ASHA will then pick a promising config from this pool based on a selection fraction (η) and promote it to the next rung for training. If the selection fraction is already exhausted, a new config will be created and trained for the first rung. A model can be promoted between rungs until it is trained for a maximum number of rungs/epochs. This process is continued until the allocated time budget for model selection workload is reached.

ASHA's decisions on configs are dependent on the wall-clock completion order of configs across task-parallel workers. Thus, it is impossible to exactly replicate a run of ASHA on task-parallelism in CEREBRO. However, we can indeed emulate ASHA on CEREBRO without making any changes to CEREBRO. We run an experiment using ASHA with a max resource budget (R) of 9 epochs, a selection fraction (η) of 3, and a time limit of 24hr.

In the given time limit, ASHA on CEREBRO (resp. Celery) explored 83 (resp. 67) configs. Figure 13 shows all learning curves. Though the individual configs are not comparable across the two systems, the best errors achieved are close (31.9% on CEREBRO ; 33.2% on Celery). A serendipity is that ASHA-on-CEREBRO seems to perform slightly *better* than the regular task-parallel version in the ASHA! We believe this is because the epoch-level synchronization in CEREBRO actually helps ASHA pick and promote better configs due to its knowledge of a larger set of configs. Regular ASHA gains this knowledge spread over time, which makes it prone to more wrong promotions. Figure 20 con-

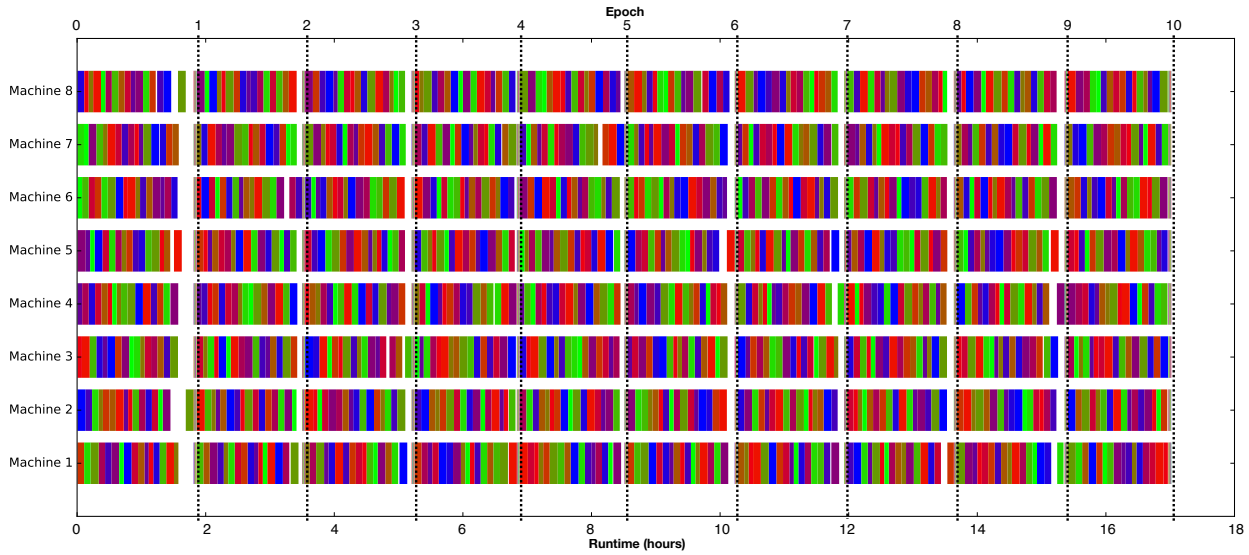


Figure 22: Gantt chart corresponding to the scheduler produced by Cerebro for the *ImageNet* workload. Each color corresponds to a different training configuration. Best viewed in color.

firms our intuition: ASHA-on-Cerebro explores more configs in the lower rungs than regular ASHA. Also, as Figure 21 shows, ASHA-on-Cerebro reaches lower errors for all rungs sooner than regular ASHA. We leave a more rigorous statistical analysis of this apparent superiority of ASHA-on-Cerebro over regular ASHA to future work.

F. GANTT CHART

Figure 22 presents the Gantt chart corresponding to the scheduler produced by Cerebro for the *ImageNet* workload. Each color bar corresponds to a different training configuration (16 in total).