# Resource-Efficient and Reproducible Model Selection on Deep Learning Systems

**Supun Nakandala** [1] **Yuhao Zhang** [1] **Arun Kumar** [1]

## Abstract

Deep neural networks (deep nets) are revolutionizing many machine learning (ML) applications. But there is a major bottleneck to wider adoption: the pain of *model selection*. This empirical process involves exploring the deep net architecture and hyper-parameters, often requiring hundreds of trials. Alas, most ML systems focus on training one model at a time, reducing throughput and raising costs; some also sacrifice reproducibility. We present CEREBRO, a system to raise deep net model selection throughput at scale without raising resource costs and without sacrificing reproducibility or accuracy. CEREBRO uses a novel parallel SGD execution strategy we call *model hopper parallelism*. Experiments on *Criteo* and *ImageNet* datasets show CEREBRO offers up to 10X speedups and improves resource efficiency significantly compared to existing systems like Parameter Server, Horovod, and task parallel tools.

## 1 Introduction

Deep learning is revolutionizing many machine learning (ML) applications. Their success at major 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. Common practice to choose these settings is to *empirically compare as many training configurations as possible* for the application. This process is called *model selection*, and it is *unavoidable* because it is how one controls underfitting vs overfitting (Shalev-Shwartz & Ben-David, 2014). For instance, Facebook often tries hundreds of configurations for just one model (Facebook, Accessed August 31, 2019).

Model selection is a major bottleneck for 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.

**Example.** Alice is a data scientist working for an e-commerce company. She has been tasked to train a deep convolutional neural network (CNN) to identify brands in product images. During model selection, she tries 5 CNN architectures and 5 values each for the initial *learning rate* and *regularizer*. So, she already has 125 training configurations to try. Later, she also tries 3 new CNNs and uses an "AutoML" procedure such as Hyperband (Li et al., 2016) to automatically decide the hyper-parameter settings. Also, her company mandates that all ML experiments have to be reproducible across multiple runs. It is too tedious and cumbersome for Alice to manually train and evaluate all these configurations. Thus she wishes to use a *model selection system* to manage and execute this workload efficiently.

### 1.1 System Desiderata for Model Selection

We identify four main groups of desiderata:

**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 users like Alice can iterate through more configurations in bulk, potentially reaching a better accuracy sooner.

**Scalability.** Deep learning often involves 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*).

**Efficiency.** Deep net training uses variants of mini-batch stochastic gradient descent (SGD). 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 forms of efficiency: (1) *per-epoch efficiency*: time to complete an epoch of training; (2) *conver-*

[*]Equal contribution [1]University of California, San Diego. Correspondence to: Supun Nakandala <snakanda@eng.ucsd.edu>.

| Desiderata | Embarrassing Task Parallelism (e.g., Dask, Celery) | 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 | ✓ 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 |
| Communication Cost (Total) | $p\langle D\rangle + m\|S\|$ | $2kmp\|S\|$ | $2kmp\|S\| \left\lceil \frac{\|D\|}{bp} \right\rceil$ | $kmp\|S\| \left\lceil \frac{\|D\|}{bp} \right\rceil$ | $kmp\|S\| + m\|S\|$ |

*Figure 1.* Qualitative comparisons of existing systems on key system desiderata for a model selection system.

*Table 1.* Notation used in Section 1

| 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) |

*gence 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.

**Reproducibility.** Ad hoc model selection with distributed training is a major reason for the "reproducibility crisis" in deep learning (Warden, Accessed August 31, 2019). 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.

## 1.2 Limitations of Existing Systems

Popular deep learning training systems like TensorFlow focus on the latency of training *one model at a time*, not on throughput. The simplest way to raise throughput is *parallelism*. Thus, various parallel execution approaches have been studied. But all these approaches have some practical limitations as Figure 1 shows (Table 1 lists the notation). We group these approaches into 4 categories:

**Embarrassingly Task Parallel.** Tools such as Python Dask, Celery, and Ray (Moritz et al., 2018) can run different training configurations on different workers in a task parallel manner. Each worker can use logically sequential SGD which yields 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, raising costs. Alternatively, one could use remote storage (e.g., S3) to read data on the fly. But this will incur massive communication overheads and costs due to remote reads.

**Bulk Synchronous Parallel (BSP).** BSP systems such as Spark and TensorFlow with model averaging (tfm, Accessed August 31, 2019) 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 (Su & Chen, 2015).

**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) (Li et al., 2014). 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-one communications, which is proportional to the number of mini-batches.

**Decentralized Fine-grained.** The best example is Horovod (Sergeev & Balso, 2018). 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 of this method is also asymptotically similar to that of Centralized Fine-grained methods.

### 1.3 Our Work

We present CEREBRO, a new system for deep net model selection that raises 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 (Pafka, Accessed August 31, 2019). This paper makes the following contributions:

- We present a novel parallel SGD execution strategy we call *model hopper parallelism* (MOP) that satisfies all the desiderata in Section 1.1 by exploiting a formal property of SGD: *robustness to data visit ordering*.
- We build CEREBRO, a general and extensible deep net model selection system using MOP. CEREBRO can support multiple deep learning tools and model selection APIs. We integrate it with TensorFlow and PyTorch and currently support grid/random searches and two AutoML procedures.
- We formalize the MOP-based model selection scheduling problem in CEREBRO and compare three alternative schedulers (MILP-based, approximate, and randomized) using simulations. We find that a simple randomized scheduler works well in our setting.
- We extend CEREBRO and its scheduler to exploit partial data replication and also support fault tolerance.
- We perform an extensive empirical comparison on real model selection workloads and datasets. CEREBRO is up to 10x faster and offers significantly higher overall resource efficiency than existing popular ML systems.

## 2 MODEL HOPPER PARALLELISM

We first explain how MOP works and then describe its properties. We are given a set $S$ of training configurations. For simplicity of exposition, assume each runs for $k$ epochs. 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 $p$ configurations from $S$ and assign one per worker (Section 4 explains how we pick the subset). On each worker, the assigned configuration 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 configuration 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 configurations are trained on all partitions, completing one epoch of SGD for each model. Repeat this every epoch until all configurations in $S$ are trained for $k$ epochs. The invariants of MOP can be summarized as follows:

- *Completeness:* In a single epoch, each training configuration is trained on all workers exactly once.

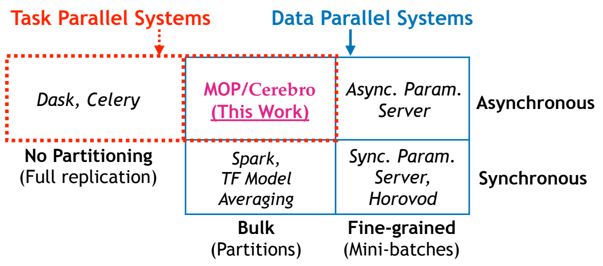


*Figure 2.* Model hopper parallelism (MOP) as a hybrid approach of task and data parallelism.

- *Model training isolation:* Two training units of the same configuration 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 (Bertsekas, 1997; Bottou, 2009). Each of the $p$ configurations visits the data partitions in a different (pseudorandom) order but each visit is sequential. 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 (Li et al., 2014), MOP is the first to do so at the *partition level* instead of mini-batches 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 2 shows, MOP is a novel hybrid of task and data parallelism that can be seen as a form of "bulk asynchronous" parallelism. Like task parallelism, MOP trains many configurations 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 4, 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 ML systems.

**Communication Cost Analysis.** MOP reaches the theoretical minimum cost of $kmp|S|$ for data-partitioned training (Figure 1), 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. As Section 1.2 shows, 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.

```
/**********************************Input Parameters***************************************/
D             : Name of the dataset that has been already registered
S             : Set of initial training configurations
automl_mthd : Name of the AutoML method to be used (e.g., Grid/Hyperband)
input_fn      : Pointer to a function which given an input file path returns
                in-memory array objects of features and labels(for supervised
                learning)
model_fn      : Pointer to a function which given a training configuration creates
                the corresponding model architecture
train_fn      : Pointer to a function which given the model and data performs
                the training for one training unit
/***************************************************************************************/
 workload_summary = launch(D, S, automl_mthd, input_fn, model_fn, train_fn)
```

*Figure 3.* CEREBRO user facing API for launching a model selection workload.

**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 SYSTEM OVERVIEW

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

### 3.1 User Facing API

The CEREBRO API allows users to do 2 things: (1) register workers and data; and (2) launch a model selection workload and get the results. Workers are registered by providing 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. This common data ETL step is orthogonal to our focus and is not a major part of the total time for multi-epoch deep net training.

The API for launching a workload is shown in Figure 3. It takes the reference to the dataset, set of initial training configurations, the AutoML procedure, and 3 user-defined functions: $input\_fn$, $model\_fn$, and $train\_fn$. CEREBRO invokes $input\_fn$ to read and pre-process the data. It then invokes $model\_fn$ to instantiate the model 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 $train\_fn$. More API details, including full signatures of all functions and a fleshed out example of how to use CEREBRO are provided in the Appendix due to space constraints.

### 3.2 System Architecture

We adopt an extensible architecture, as Figure 4 shows. This allows us to easily support multiple deep learning tools and

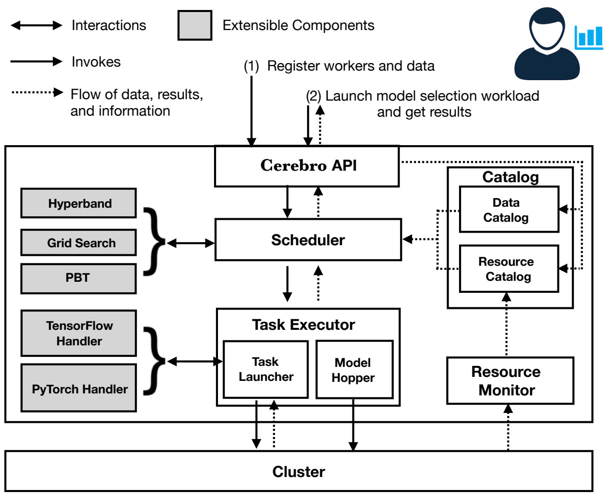


*Figure 4.* High-level system architecture of CEREBRO system.

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. Next, we describe how CEREBRO's architecture enables high system generality. Section 4 explains how the Scheduler works and how we achieve fault tolerance and elasticity.

**Supporting Multiple Deep Learning Tools.** The functions $input\_fn$, $model\_fn$, and $train\_fn$ are written by users in a standard deep learning tool's APIs. 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. We currently support TensorFlow and PyTorch; it is simple to add support for more tools.

**Supporting Multiple AutoML Procedures.** Metaheuristics called AutoML procedures are common for exploring training configurations. 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 configurations ($S$) and evaluate them after each epoch (or every few epochs). Based on the evaluations, terminate some configurations (e.g., as in Hyperband (Li et al., 2016) and PBT (Jaderberg et al., 2017)) or add new configurations (e.g., as in PBT). Grid/random search is a one-shot instance

*Table 2.* 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) |

of this template. Thus, we use this template in our Scheduler. Given $S$, CEREBRO trains all models in $S$ for one epoch and passes control back to the 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 reshuffling after each epoch. But the default (and common practice) is to shuffle only once up front. Currently, CEREBRO supports grid search, random search, Hyperband, and PBT. It is simple to extend our API to support more AutoML procedures and arbitrary convergence criteria.

### 3.3 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 pushed-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. This doubles the communication cost of MOP to $2kmp|S|$, still a negligible overhead. But using NFS greatly reduces engineering complexity to implement MOP.

## 4 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. Thus, we now formalize the MOP-based scheduling problem and explain how we design our Scheduler. For starters, assume each of the $p$ data partitions is assigned to only one worker.

### 4.1 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). The objective and constraints of the MOP-based scheduling problem is as follows. Table 2 lists the additional notation used here.

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

`Constraints:`

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

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 2. 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 (Gurobi, Accessed August 31, 2019). 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` (Gonzalez & Sahni, 1976). Since $|S|$ can even be 100s, MILP solvers may be too slow (more in Section 4.4); thus, we explore alternative approaches.

### 4.2 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" (Woeginger, 2018; Fiala, 1983). 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 $S$.

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

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

**Algorithm 1** Randomized Scheduling

```
1: **Input:** $S$
2: $Q = \{s_{i,j} : \forall i \in [1, \ldots, |S|], \forall j \in [1, \ldots, p]\}$
3: worker_idle ← [true, ..., true]
4: model_idle ← [true, ..., true]
5: while not empty($Q$) do
6:   for $j \in [1, \ldots, p]$ do
7:     if worker_idle[$j$] then
8:       $Q$ ← shuffle($Q$)
9:       for $s_{i,j'} \in Q$ 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
```

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 (Woeginger, 2018; Fiala, 1983).

### 4.3 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.

### 4.4 Comparing Different Scheduling Methods

We perform simulation experiments to compare the efficiency and makespan yielded by the three alternative schedulers. The MILP and approximate algorithm are implemented using Gurobi. We set a maximum optimization time of 5 minutes for tractability of experimentation. We vary the number of training configurations, size of cluster, and homogeneity/heterogeneity of training configurations and/or workers. We define homogeneous configurations (resp. workers) as those with the same compute cost (resp. capacity).

**Algorithm 2** Upon Completion of Training Unit $s_{i,j}$ on Worker $j$

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

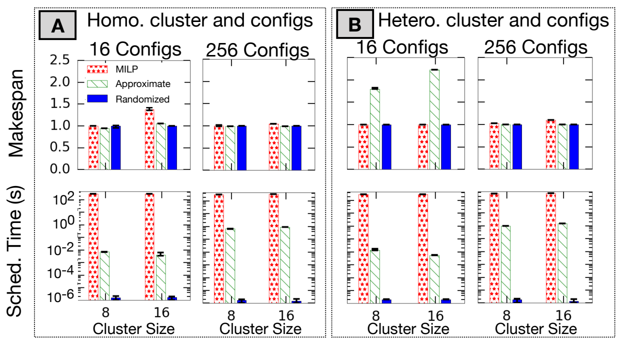


*Figure 5.* Makespan and scheduling time of the generated schedule by different scheduling methods for different settings. (A) Homogeneous cluster and training configurations and (B) heterogeneous cluster training configurations.

Training configuration compute costs are randomly sampled from 36 popular deep CNNs from (Albanie, Accessed August 31, 2019). The costs vary from 360 MFLOPS to 21000 MFLOPS with a mean of 5939 MFLOPS and standard deviation of 5671 MFLOPS. We randomly sample 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). We report the average of 5 runs with different random seeds and also the min and max of all 5 runs. All makespans reported are normalized by the randomized scheduler's makespan. Figure 5 presents the results for homogeneous cluster and configurations, as well as heterogeneous cluster and configurations. More results, including on homogeneous cluster-heterogeneous configurations, and other cluster sizes are presented in the Appendix due to space constraints.

The MILP scheduler sometimes performs poorer than the other two as it has not converged to the optimal solution in the given time budget. Approximate algorithm-based scheduler performs poorly when both the configurations and workers are heterogeneous. It also takes more time 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.

### 4.5 Replica-Aware Scheduling

So far we have assumed that each partition is available only on 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 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.

### 4.6 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 Experimental Evaluation

We empirically validate if CEREBRO can improve throughput and efficiency of deep net model selection workloads. We then perform drill-down experiments to evaluate various aspects of CEREBRO separately.

**Datasets.** We use two large benchmark datasets: *Criteo* (CriteoLabs, Accessed August 31, 2019) and *ImageNet* (Deng et al., 2009). Table 3. summarizes the dataset statistics. *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; we made the decision so that the evaluations can complete in reasonable amount of time (several weeks in this case). *ImageNet* is a popular image classification benchmark dataset. We choose the 2012 version and reshape the images to $112 \times 112$ pixels.[1]

*Table 3.* Dataset details. $^\star$All numbers are after preprocessing and sampling of the datasets.

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

**Workloads.** For end-to-end tests, we use grid search to generate a set of 16 training configurations for each dataset. Table 4 provides the details. We use Adam (Kingma & Ba, 2014) as our SGD-based optimizer. To demonstrate CEREBRO's generality, we also perform an experiment where the training configurations are generated and updated by the PBT method. Due to space constraints, we present the PBT results, including the Gantt chart showing the resource utilization of MOP, in the Appendix (the trends are similar).

**Experimental Setup.** We use two clusters: one CPU only and one equipped with GPUs, both on CloudLab (Ricci et al., 2014). 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, and we use TensorFlow version 1.12.0 as CEREBRO's underlying deep learning library. For GPU nodes, we use CUDA version 9.0 and cuDNN version 7.4.2. Both datasets are randomly shuffled and partitioned into 8 equal sized partitions.

[1]We downsampled the images slightly to complete all experiments, which are highly compute-intensive, in a reasonable time. This decision does not alter the takeaways from our experiments.

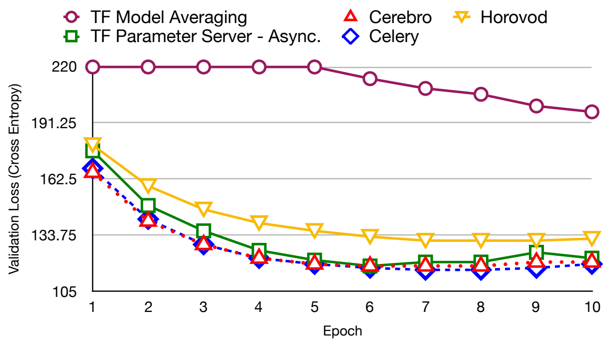
*Table 4.* Workloads.$^\star$architectures similar to VGG16 and ResNet50, respectively.

| Dataset | Model arch. | Batch size | Learning rate | Regularization | Epochs |
|---|---|---|---|---|---|
| Criteo | 3-layer NN, 1000+500 hidden units | $\{32, 64, 256, 512\}$ | $\{10^{-3}, 10^{-4}\}$ | $\{10^{-4}, 10^{-5}\}$ | 5 |
| ImageNet | {VGG16$^\star$, ResNet50$^\star$} | $\{32, 256\}$ | $\{10^{-4}, 10^{-6}\}$ | $\{10^{-4}, 10^{-6}\}$ | 10 |

| System | Criteo | | ImageNet | | Memory/ Storage Footprint |
|---|---|---|---|---|---|
| | Runtime (hrs) | CPU Utili. | Runtime (hrs) | GPU Utili. | |
| TF PS - Async | 144.0 | 6.9% | 190.0 | 8.6% | 1X |
| Horovod | 70.3 | 16.0% | 54.2 | *92.1% | 1X |
| TF Model Averaging | 19.2 | 52.2% | 19.70 | 72.1% | 1X |
| Celery | 27.4 | 41.3% | 19.5 | 73.2% | 8X |
| Cerebro | 17.0 | 51.9% | 17.7 | 79.8% | 1X |

* Horovod uses GPU kernels for communication. Thus, it has high GPU utilization.

(a) Makespans and CPU/GPU utilization.



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

*Figure 6.* End-to-end results on *Criteo* and *ImageNet*.

## 5.1 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 and stream them from disk, as they do not fit in memory. For all other systems, including CEREBRO, each worker node has a single data partition which is in-memory.

Figure 6 presents the results. We see CEREBRO significantly improves the efficiency and throughput of the model selection workload. On *Criteo*, CEREBRO is 14x faster than synchronous PS, and it is over 8x faster than asynchronous PS. Both versions of PS suffer from less than 7% CPU utilization. CEREBRO is also 4x faster than Horovod. CEREBRO's runtime is comparable to model averaging with close to 50% CPU utilization. Celery takes slightly more time than CEREBRO due to a skew in task assignment, where one worker runs two of the most time-consuming tasks. All methods have almost indistinguishable convergence behavior, as the dataset itself is highly skewed, and each technique can obtain over 99% accuracy quickly.

On *ImageNet*, CEREBRO is over 10x faster than asynchronous PS, which has a GPU utilization is as low as 8%! Synchronous PS was even slower. CEREBRO is 3x faster than Horovod. Horovod has high GPU utilization because these values also include the communication time (for Horovod, while the communication is happening, the GPU is marked busy). CEREBRO's runtime is comparable to model averaging and Celery. But model averaging does not converge at all. Celery and CEREBRO have the best learning curves, which are also almost identical, but note

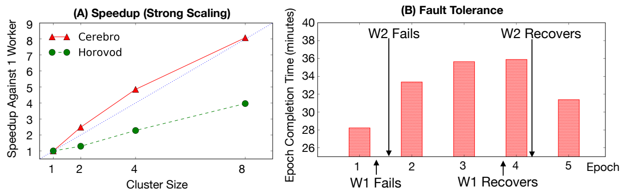


*Figure 7.* (A) Speedup (strong scaling). (B) Fault-tolerance.

that Celery has 8x the memory/storage footprint. Horovod converges slower due to its larger effective mini-batch size. Overall, CEREBRO is among the most resource-efficient and still offers accuracy similar to sequential SGD.

## 5.2 Drill-down Experiments

For these experiments, unless specified otherwise, we use the GPU cluster, *ImageNet*, and a model selection workload of 8 configurations (4 learning rates, 2 regularization values, and ResNet architectures) and train for 5 epochs. Each data partition is only available on a single worker.

**Scalability.** We study the speedups (strong scaling) of CEREBRO and Horovod as we vary the cluster sizes. Figure 7(A) shows the speedups, defined as the workload completion time on multiple workers vs a single worker. CEREBRO exhibits linear speedups (even slightly super-linear). This is because of MOP's marginal communication cost and because the data fits entirely in cluster memory compared to the small overhead of reading from disk on the single worker. In contrast, Horovod exhibits sub-linear speedups due to its higher communication costs with more 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 7(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.

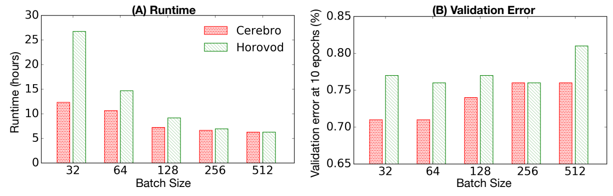


*Figure 8.* Effect of batch size on communication overheads and convergence efficiency. (A) Runtime against batch size. (B) The lowest validation error after 10 epochs against batch size.

**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 8 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 ($32 \times 8$). 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 9 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 the initial read, CEREBRO only communicates trained models during

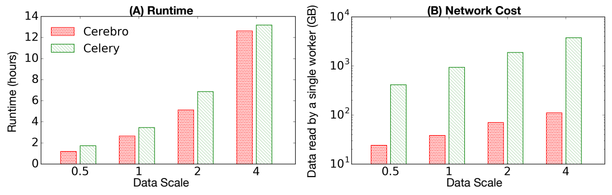


*Figure 9.* Reading data from remote storage.

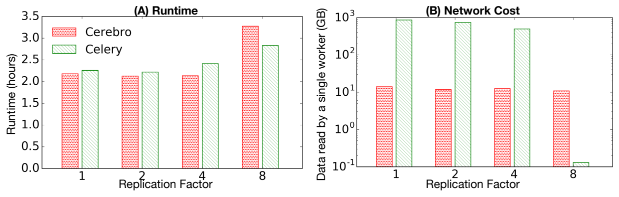


*Figure 10.* Reading data from distributed storage.

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 9 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.

## 6 LIMITATIONS AND DISCUSSION

We recap the key assumptions and limitations of our work.

**Number of Training Configurations.** So far we assumed $|S| \geq p$. While this is reasonable for model selection on small clusters, CEREBRO will under-utilize workers if this condition does not hold. But interestingly, CEREBRO can still often outperform Horovod in some cases when $|S| < p$. Figure 11 shows some results on *Criteo*. Horovod's runtime grows linearly with more configurations, but CEREBRO is

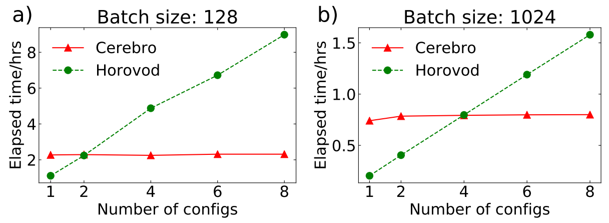


*Figure 11.* Runtime comparison on on *Criteo* on an 8-node cluster when Cerebro is under-utilizing resources. Configs: same model as in Table 4, learning rates drawn from $\{10^{-3}, 10^{-4}, 5 \times 10^{-5}, 10^{-5}\}$, weight decays drawn from $\{10^{-4}, 10^{-5}\}$.

constant. For instance, at batch size 128 with $|S| = 2$ only, CEREBRO matches Horovod's performance. This is because CEREBRO's communication costs are negligible. We also explored a hybrid of MOP and Horovod, discussed further in the Appendix due to space constraints. It was bottlenecked by Horovod's network overheads, and mitigating this issue will require careful data re-partitioning strategies, which we leave to future work. Another hybrid we plan to pursue in future work is MOP with model averaging, say, for the latter stages of Hyperband when $|S|$ can go below $p$.

**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 notion 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.

**Integration into Data-Parallel Systems.** MOP's generality makes it amenable to emulation on top of BSP data-parallel systems such as parallel RDBMSs and dataflow systems. We are collaborating with Pivotal to integrate MOP into Greenplum by extending the MADlib library (MADlib, Accessed August 31, 2019) for running TensorFlow on Greenplum-resident data. Pivotal's customers are interested in this integration for enterprise ML use cases including language processing, image recognition, and fraud detection. We also plan to integrate MOP into Spark in a similar way.

## 7 RELATED WORK

**Cluster Scheduling for Deep Learning.** Gandiva (Xiao et al., 2018) is a cluster scheduling system for deep learning that also targets model selection. But its focus is on lower-level primitives such as GPU time slicing and intra-server locality awareness. CEREBRO is complementary as it operates at a higher abstraction level. Tiresias (Gu et al., 2019) is another GPU cluster manager for deep learning. It targets Parameter Servers and reduces the makespan via a generalized least-attained service scheduling algorithm and better task allocation. It is also orthogonal to CEREBRO, which operates at a higher abstraction level and targets different system environments. How compute hardware is allocated is outside our scope; we believe CEREBRO can work with both of these cluster resource managers. There is a long line of work on general job scheduling algorithms in the operations research and systems literatures (Herroelen et al., 1998; Brucker, 2001; Gautam et al., 2015). Our goal is *not* to create new scheduling algorithms but to apply known algorithms to a new ML systems setting based on MOP.

**AutoML procedures.** Procedures such as Hyperband (Li et al., 2016) and PBT (Jaderberg et al., 2017) automate hyper-parameter tuning efficiently. They are complementary to our work and exist at a higher abstraction level; CEREBRO acts as a distributed execution engine for them.

**System optimizations.** Another recent stream of research focuses on the optimization of deep learning tools. Such optimization is achieved by increasing the overlap of communication (PipeDream (Narayanan et al., 2019), P3 (Jayarajan et al., 2019)), smart graph rewriting (SOAP (Jia et al., 2019)) and model batching (HiveMind (Narayanan et al., 2018)). These are also orthogonal to CEREBRO as they target different settings. We believe MOP is general enough to hybridize with these new systems.

## 8 CONCLUSIONS AND FUTURE WORK

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

## REFERENCES

Script for Tensorflow Model Averaging, Accessed August 31, 2019. `https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/utils/avg_checkpoints.py`.

Albanie, S. Memory Consumption and FLOP Count Estimates for Convnets, Accessed August 31, 2019. `https://github.com/albanie/convnet-burden`.

Bertsekas, D. P. A new class of incremental gradient methods for least squares problems. *SIAM J. on Optimization*, 7(4):913–926, April 1997. ISSN 1052-6234.

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

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

CriteoLabs. Kaggle Contest Dataset Is Now Available for Academic Use!, Accessed August 31, 2019. `https://ailab.criteo.com/category/dataset`.

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

Facebook. Introducing FBLearner Flow: Facebook's AI backbone, Accessed August 31, 2019. `https://engineering.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/`.

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

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

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

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

Gurobi. Gurobi Optimization, Accessed August 31, 2019. `https://www.gurobi.com`.

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

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

Jayarajan, A., Wei, J., Gibson, G., Fedorova, A., and Pekhimenko, G. Priority-based Parameter Propagation for Distributed DNN Training. In *SYSML 2019*, 2019.

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

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

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

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

MADlib. Apache MADlib: Big Data Machine Learning in SQL, Accessed August 31, 2019. `https://madlib.apache.org/`.

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

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

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

Pafka, S. Big RAM is Eating Big Data - Size of Datasets Used for Analytics, Accessed August 31, 2019. `https://www.kdnuggets.com/2015/11/big-ram-big-data-size-datasets.html`.

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

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

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

Su, H. and Chen, H. Experiments on Parallel Training of Deep Neural Network using Model Averaging. *CoRR*, abs/1507.01239, 2015. URL `http://arxiv.org/abs/1507.01239`.

Warden, P. The Machine Learning Reproducibility Crisis, Accessed August 31, 2019. `https://petewarden.com/2018/03/19/the-machine-learning-reproducibility-crisis`.

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

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

## 9 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 pre-processing, 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.

## 10 REPLICA-AWARE MIP SCHEDULER

We extend the mixed integer linear program formulation described in Section 3.1 to make it replica-aware. It requires the availability information of partitions on workers ($A$) as an additional input. The resulting formulation is a mixed integer quadratic program. The objective and the constraints can be formalized as follows. Additional notation used is explained in Table 5.

*Table 5.* Notation used for the replica-aware MIP formulation

| Symbol | Description |
|---|---|
| $\|S\|$ | Number of training configurations |
| p | Number of data partitions |
| w | Number of Workers |
| $T \in \mathbb{R}^{\|S\| \times p \times w}$ | $T_{i,j,k}$ is the runtime of the $i^{th}$ configuration on the $j^{th}$ partition at the $k^{th}$ worker |
| $A \in \{0,1\}^{p \times w}$ | $A_{j,k} = 1 \iff j^{th}$ partition is available on the $k^{th}$ worker |
| $C$ | Makespan of the model selection workload |
| $Q \in \{0,1\}^{\|S\| \times p \times w}$ | $Q_{i,j,k} = 1 \iff$ execution of the $i^{th}$ configuration for $j^{th}$ partition is executed on $k^{th}$ worker |
| $X \in \mathbb{R}^{\|S\| \times p}$ | $X_{i,j}$ is the start time of the execution of $i^{th}$ configuration on $j^{th}$ partition |
| $Y \in \{0,1\}^{\|S\| \times p \times p}$ | $Y_{i,j,j'} = 1 \iff X_{i,j} < X_{i,j'}$ |
| $Z \in \{0,1\}^{\|S\| \times p \times \|S\| \times p \times w}$ | $Z_{i,j,i',j',k} = 1 \iff Q_{i,j,k} = Q_{i',j',k} = 1$ and $X_{i,j} < X_{i',j'}$ |
| $V$ | Very large value |

`Objective:` $\min_{C,X,Y,Z,Q} C$

`Constraints:`

(1) $A_{j,k} = 0 \rightarrow Q_{i,j,k} = 0$

(2) $\sum_{k=1}^{w} Q_{i,j,k} = 1$

(3) $Q_{i,j,k} \cdot Q_{i,j',k'} = 1 \; and \; j \neq j' \rightarrow$
$(i)\; X_{i,j} \geq X_{i,j'} + T_{i,j',k'} - V \cdot Y_{i,j,j'}$
$(ii)\; X_{i,j'} \geq X_{i,j} + T_{i,j,k} - V \cdot (1 - Y_{i,j,j'})$

(4) $Q_{i,j,k} \cdot Q_{i',j',k} = 1 \; and \; i \neq i' \rightarrow$
$(i)\; X_{i,j} \geq X_{i',j'} + T_{i',j',k} - V \cdot Z_{i,j,i',j',k}$
$(ii)\; X_{i',j'} \geq X_{i,j} + T_{i,j,k} - V \cdot (1 - Z_{i,j,i',j',k})$

(5) $X_{i,j} \geq 0$

(6) $C \geq X_{i,j} + T_{i,j,k}$

$$\forall i, i' \in [1, \ldots, |S|], \forall j, j' \in [1, \ldots, p],\ \forall k \in [1, \ldots, w]$$

The objective is to minimize the makespan C, subject to the constraints on the makespan C, training unit assignment matrix Q, configuration unit start times X, model training isolation matrix Y, and worker/partition exclusive access matrix Z. As per constraint 1, a training unit can be run on a worker only if the corresponding partition is available. Constraint 2 ensures that a training unit is assigned to only one worker. Constraints 3 (i) and (ii) ensure model training isolation and constraints 4 (i) and (ii) ensure worker exclusive access. Constraint 5 ensures that training unit start times are nonnegative and constraint 6 ensures that $C$ captures the time taken to complete all training units. The training unit assignment Q and start times X yield the actual schedule.

## 11 Population-based Tuning with Cerebro

To demonstrate Cerebro's ability to support arbitrary AutoML procedures, we run an experiment with population-based tuning method (PBT). For this we choose PyTorch as the back-end deep learning tool. We use the *ImageNet* dataset and 12 initial training configurations: 3 model architectures, 1 batch size, 2 learning rates, and 2 weight decay values (see Table 6). The experimental setup is the same GPU cluster used in Section 4.

We set the iteration size of PBT to 5 epochs. After every iteration, based on the validation loss PBT method halts the lower performing half of configurations. It replaces these configurations by new configurations which are derived by sampling from the top half and incorporating a mutation to the batch size, learning rate, and weight decay. Mutations are sampled as follows: for batch size from $\{16, -16\}$ and for learning rate and weight decay from

*Table 6.* Initial configurations used in the PBT experiment.

| Dataset | Model arch. | Batch size | Learning rate | Weight decay | Epochs |
|---|---|---|---|---|---|
| ImageNet | {ResNet18, ResNet34, ResNet50} | {64} | $\{10^{-2}, 10^{-4}\}$ | $\{10^{-2}, 10^{-4}\}$ | 40 |

$\{5^{-4}, -5^{-4}, 5^{-5}, -5^{-5}, 5^{-6}, -5^{-6}\}$. We repeat this process for 8 iterations. The Gantt chart for the schedule produced by CEREBRO for this workload is shown in Figure 12. We see CEREBRO yields very high system utilization and seamlessly supports the PBT model selection workload.

## 12 HOROVOD HOPPER(HOHO)

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 mixing MOP with data parallelism. Towards this end, we implement a hybrid version of CEREBRO with Horovod we call Horovod Hopper (HOHO). Just like CEREBRO, Horovod is also equivalent to sequential SGD concerning convergence behavior. Therefore the hybrid of them will remain reproducible.

Figure 13 summarizes the architecture of HOHO, 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.

We test HOHO on *criteo* with varying number of model configs with the same CPU cluster used before. The batch sizes and makespans of these model configs are identical. We then conduct the same test multiple times with different batch sizes.

Figure 14 shows the results. The runtime of CEREBRO is constant, while the runtimes of Horovod and HOHO are linear, except when $|S|$ equals $p$, HOHO reduces to CEREBRO . It is interesting even with underutilization, CEREBRO can still outperform Horovod in some scenarios. There is a cross-over point when the three methods meet, depending on $|S|$. Typically, when $|S|$ is much smaller than $p$, HOHO and Horovod are faster, as CEREBRO is heavily under-utilized. HOHO also does not provide much benefit over Horovod, as HOHO mainly optimizes Horovod for its latency part of the communication cost. However, it turns out this part of the cost is marginal.

The cross-over point depends on the batch size. We then heuristically choose $p/2$ as the dividing point: still run CEREBRO if $p/2 < |S|$, otherwise just run Horovod.

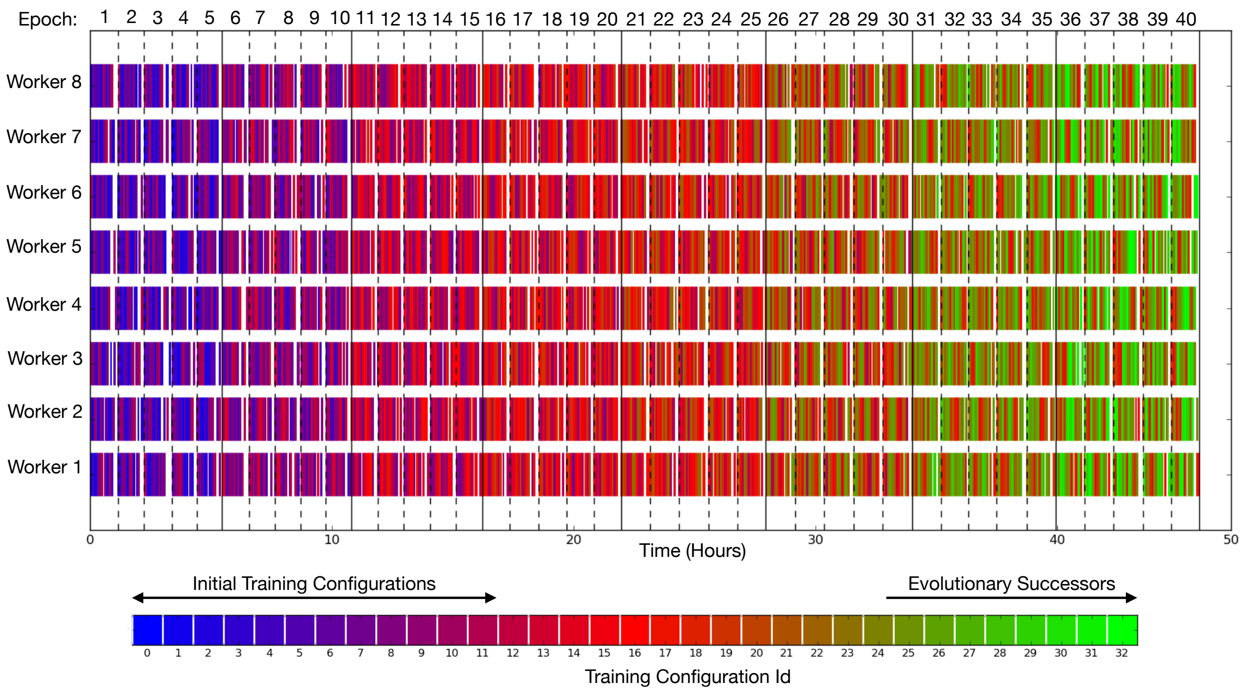


*Figure 12.* Gantt chart produced for the execution of PBT AutoML procedure using CEREBRO. We start with 12 initial training configurations. After every 5 epochs the worst performing 6 training configurations are killed and are replaced by mutants of the top performing 6. We mutate the learning rate, weight decay, and the training mini-batch size. Best viewed in color.

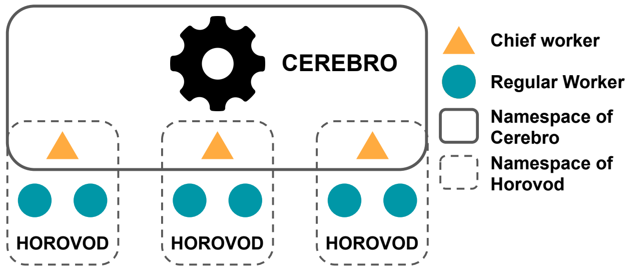


*Figure 13.* The architecture of HOHO. Within different namespaces, we run CEREBRO and Horovod, respectively. The chief workers, acting as CEREBRO workers, are responsible for driving Horovod tasks and handling the communication between the two systems. In the figure, we show a 9-node cluster with 3 model configs to train.

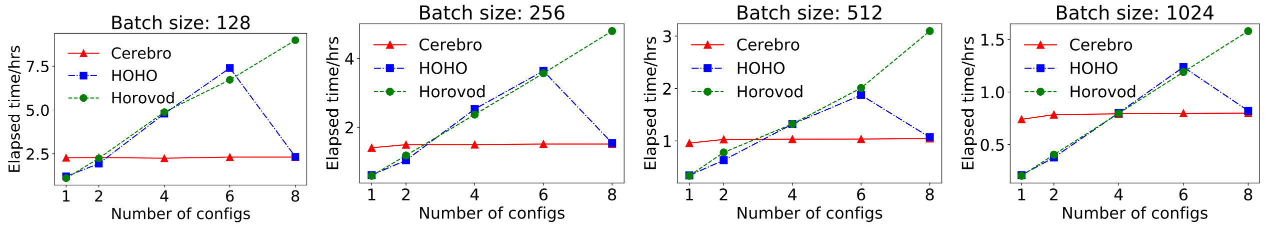


*Figure 14.* Performance tests of HOHO with varying batch size and $|S|$ on 8-node cluster. Configs: same model as in Section 4 Table 4, learning rates drawn from $\{10^{-3}, 10^{-4}, 5 \times 10^{-5}, 10^{-5}\}$, weight decays drawn from $\{10^{-4}, 10^{-5}\}$. We test on 4 different batch sizes, respectively.

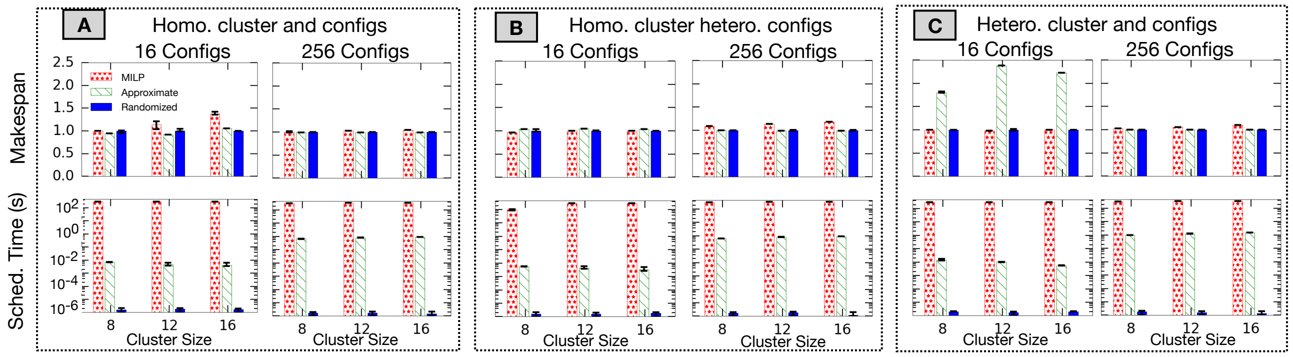


*Figure 15.* Makespan and scheduling time of the generated schedule by different scheduling methods for different settings. Makespan values are normalized with respect to the makespan of the randomized scheduling approach. (A) Homogeneous cluster and homogeneous training configurations, (B) homogeneous cluster and heterogeneous training configurations, and (C) heterogeneous cluster and heterogeneous training configurations.