Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems

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

ABSTRACT

Artificial Neural Networks (ANNs) 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 ANN 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 our vision of Cerebro, a system to raise ANN model selection throughput at scale and ensure reproducibility. Cerebro uses a novel parallel execution strategy we call model hopper parallelism. We discuss the research questions in building Cerebro and present promising initial empirical results.

ACM Reference Format:
Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2019. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *International Workshop on Data Management for End-to-End Machine Learning (DEEM'30), June 30, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3329486.3329496

1 INTRODUCTION

Artificial Neural Networks (ANNs) are revolutionizing many machine learning (ML) applications. Their success at major Web companies has created excitement among many enterprises and domain scientists to try ANNs for their applications. But training ANNs is a painful empirical process, since accuracy is tied to the ANN architecture and hyper-parameter settings. Common practice to choose these settings is to *empirically compare as many training configurations* for the application. This process is called *model selection*, and it is *unavoidable* because it is how one controls underfitting vs overfitting [2]. Model selection is a major bottleneck for adoption of ANNs among enterprises and domain scientists due to both the *time spent* and *resource costs*.

DEEM'30, June 30, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6797-4/19/06…$15.00
https://doi.org/10.1145/3329486.3329496

Example. Alice wants to train a deep convolutional neural network (CNN) to identify brands in product images. She tries 5 different CNN architectures and 5 values each for the initial *learning rate* and *regularizer*. So, she already has 125 training configurations to try. She then tries 3 new CNNs and uses an "AutoML" procedure such as Hyperband [3] to automatically decide the hyper-parameter settings.

Importance of Throughput. Regardless of grid/random searches or AutoML searches, a key bottleneck for Alice's training is *model selection throughput*: how many training configurations are evaluated per unit time. Higher throughput means she can iterate through more configurations in bulk, potentially reaching a better accuracy sooner. High-throughput execution can also reduce total resource costs by improving resource utilization.

Limitations of Existing Systems. Alas, popular ANN tools like TensorFlow focus on the latency of training *one model at a time*, not on throughput. The simplest way to raise throughput is *parallelism*. ANN training uses variants of mini-batch stochastic gradient descent (SGD), but SGD is inherently sequential. Thus, various parallel execution approaches have been studied; we group them into 3 categories: (1) task parallel; (2) bulk synchronous parallel; and (3) fine-grained asynchronous or synchronous data parallel. Each category has some practical limitations: poor scalability, high resource costs, low throughput, and/or *unreproducible* executions. While reproducibility may not be a concern for some Web users, it is a showstopper for many enterprises and domain scientists. Section 3 explain the tradeoffs further.

This Work. We present our vision of a new system for ANN model selection we call Cerebro that *raises throughput without raising resource costs*. Our target setting is *small clusters* (say, 10s of nodes), which covers a vast majority (almost 90%) of parallel ML workloads in practice [6]. We have 4 key system desiderata: *scalability*, *SGD efficiency*, *reproducibility*, and *system generality*; we explain these in Section 2. To satisfy all these desiderata, Cerebro uses a novel parallel

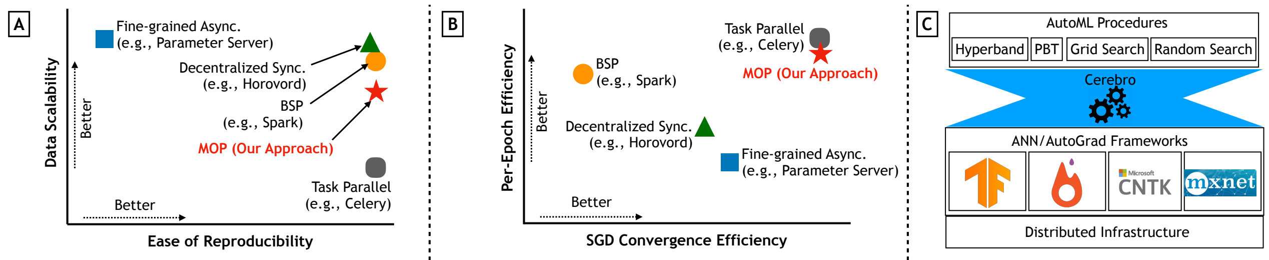


Figure 1: (A,B) Qualitative comparisons of tradeoffs on key system desiderata (data scalability and convergence efficiency affect accuracy). (C) High-level architecture of our envisioned system.

execution strategy we call *model hopper parallelism* (MOP). MOP combines task and data parallelism by exploiting a key formal property of SGD. We explain MOP and our proposed system architecture in Section 4. We discuss the key research challenges in realizing Cerebro and explain our plans. Finally, we present some promising initial empirical results on ImageNet that show the benefits of Cerebro.

2 SYSTEM DESIDERATA

Scalability. Deep ANNs typically use large training datasets, often larger than single-node memory or even disk. ANN model selection is also highly compute-intensive, which means multiple nodes (with or without GPUs) are typically used to reduce completion time. Thus, we desire out-of-the-box scalability to large partitioned datasets (*data scalability*) and distributed execution on a cluster (*compute scalability*). Our current focus is the small cluster setting, which is common among enterprises [6].

SGD Efficiency. We need to *maximize resource utilization* for executing SGD on a cluster. This has two parts: *per-epoch efficiency* and *convergence efficiency*. For the former, we need to reduce data ETL overheads and network communication costs. It also means we need to reduce idle times. For the latter, the gold standard is *logically sequential SGD*. Anything that deviates from this usually needs more epochs to converge to a similar accuracy. In Section 3, we explain the limitations of prevailing logically parallel SGD approaches: fine-grained asynchronous [4], fine-grained synchronous [8], and bulk synchronous.

Reproducibility. From speaking with ML users in many enterprises and domain sciences, we find that Parameter Server (PS) [4] and related asynchronous/semi-synchronous approaches are rarely used in such settings. This is because they want reproducible executions. But unreproducibility is inherent in such approaches due to the uncontrollable randomness of the physical world. Since such settings are our main focus, we desire exact reproducibility.

System Generality. We seek to support a variety of model selection procedures ranging from manual grid or random searches to AutoML procedures such as Hyperband [3]. These differ in *what* set of training configurations they execute in bulk. This is a higher-level decision that is *orthogonal* to our focus. We also desire to support many popular ANN frameworks such as TensorFlow, PyTorch, MXNet, and CNTK. Such frameworks support arbitrary neural computational graphs, offer many SGD-based optimizers, and have hardware-efficient linear algebra kernels. We do not want to waste their engineering efforts; rather, we desire to *build on top of them* to make practical adoption easier.

3 TRADEOFFS OF EXISTING SYSTEMS

We now explain the tradeoffs of existing approaches for parallel ANN model selection. But first, we recap the data access pattern of SGD. We randomly shuffle the training dataset and then perform a sequential scan per epoch. The most common practice is to shuffle dataset only once. ANN training typically needs dozens of epochs. There are 4 main paradigms of parallelism: embarassingly task parallel, bulk synchronous parallel (BSP), Parameter Servers, and decentralized synchronous parallel. Figure 1(A,B) depicts their tradeoffs.

Embarassingly Task Parallel. Tools such as Python Dask, Celery, and Ray [5] can run different training configurations on different workers in a task parallel manner. Each worker can use logically sequential SGD, which yields reproducibility and best SGD efficiency. There is no communication across workers during training, but the whole dataset is copied to each worker. While this may suffice for small datasets, it is *not scalable* to large partitioned datasets, forcing users to downsample and risk overfitting. Copying the whole datasets to all workers is also *highly wasteful of resources*, both memory and storage, raising costs. Alternatively, one could use distributed storage (e.g., S3) to read data remotely instead of replicating. But this approach will still incur massive overheads and costs due to remote I/O reads.

Bulk Synchronous Parallel (BSP). BSP systems such as Spark and TensorFlow with model averaging parallelize one model at a time. They broadcast the model, train models independently on *each worker's partition*, collect all models on the master, average the weights, and repeat this every epoch. Alas, this approach converges poorly; so, it is almost never used for ANN training.

Parameter Server (PS). PS approaches also parallelize one model at a time but at a finer granularity than BSP. Workers push gradient updates to the master *at each mini-batch* and pull the latest model when ready. If the master waits to collect all updates per cycle, it is *synchronous*; otherwise, if the master continues training whenever it gets an update, it is *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 at each mini-batch.

Decentralized Synchronous Parallel. Decentralized systems such as Horovod [8] adopt HPC-style techniques to enable synchronous all-reduce SGD. It is reproducible and the adopted ring all-reduce algorithm has a time complexity independent of the number of workers for the bandwidth-bound term. However, the synchronization barrier becomes a communication bottleneck.

4 OUR PROPOSED SYSTEM

We present our MOP execution strategy and envisioned system architecture. We also discuss key research questions and present some initial results.

4.1 Model Hopper Parallelism (MOP)

From Figure 1(A,B), we observe that BSP and task parallelism, between them, cover all of our desiderata. This motivates us to hybridize them in a manner inspired by *multi-query optimization* (MQO) techniques [7]. Our intuition is to *"emulate" BSP data parallelism underneath task parallelism* to increase scalability and avoid replicating the whole dataset.

Given a dataset D, shuffle it once and partition it across p workers. The model selection procedure gives a set S of ANN training configurations. p is up to a few 10s in our setting, but $|S|$ can even be 100s; so, $p << |S|$ typically. MOP is decentralized: the client schedules S. Given a *schedule* (explained shortly), start SGD for p models from S *in parallel* on p workers. When a model finishes a pass over its partition, a *sub-epoch* of SGD is done. That model is *checkpointed* locally, "hops" to the next worker as per schedule, and *resumes the same epoch* on the new partition. After p hops, all of D is seen and a full epoch of SGD is over. Repeat this every epoch. Note that checkpointing time here is tiny relative to ANN training times. MOP is thus scalable to large partitioned datasets *without any replication*, unlike task parallelism. Reproducibility is trivial–just replay the saved schedule. MOP also guarantees a strong property: *logical equivalence to sequential SGD* for each model, unlike logically parallel SGD systems. Optimization theory tells us that *any random data ordering* is acceptable for SGD–MOP leverages this theoretical insight. Finally, MOP's communication costs are much lower than PS, as Figure 1(C) shows, since a model hops *once per partition*, not once per mini-batch.

4.2 Envisioned System Architecture

We propose a narrow-waist architecture inspired by [2] to support many AutoML procedures and many ANN/Autograd frameworks, as Figure 1(C) illustrates. The core of Cerebro is an *optimizing scheduler* to apply MOP to each ANN training configuration over the distributed infrastructure.

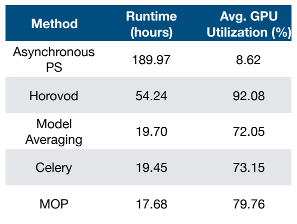
4.3 Research Challenges and Plan

Challenge: Efficient Scheduling. While the core idea of MOP is simple, we need to tackle some algorithmic and systems challenges for the Cerebro Scheduler. Our goal is to *maximize resource utilization*. Randomly allocating ANN training *jobs* from S to workers in a greedy fashion could be sub-optimal, since some ANNs could finish their sub-epochs faster, leading to idle times between model hops. Thus, we need to schedule more carefully for more resource efficiency. This requires estimating the *job unit costs* and *worker speeds*.

Our Plan. We formalize the Cerebro Scheduler as a new variant of the classical *open shop scheduling* [9] problem from the operations research literature. A sub-epoch in MOP is a job's *unit*. Given job unit costs and worker speeds, we want to minimize *makespan* (completion time); this is known to be NP–Hard. But our setting has a special property: $p << |S|$. We plan to exploit this property to adapt PTIME algorithms from [9]. To estimate units costs and worker speeds, one could devise cost models; but these are unwieldy for complex ANNs. Our plan is simpler: use the first epoch as a pilot run to estimate these quantities. Since ANN training often takes dozens of epochs, the pilot run's overhead is marginal.

Challenge: System Generality This has two parts. First, Cerebro should offer extensible APIs to specify the set S of training configurations in any way, including with AutoML procedures. Second, Cerebro must be extensible enough to support any arbitrary ANN/AutoGrad framework (e.g., TensorFlow or PyTorch), all kinds of ANN architectures, and any SGD-based optimizer (e.g., Adam or AdaGrad).

Our Plan. We observe that most AutoML procedures for hyper-parameter tuning and neural architecture search fit a common template: Create an initial set of training configurations and evaluate them after each epoch (or after every few epochs). Based on these evaluations, terminate some configurations (e.g., as in Hyperband [3] and PBT [1]) or add new configurations [1]. Grid/random search is a one-shot instance of this template. Thus, we adopt this template for our Scheduler and run it *one epoch at a time*. Given S, Cerebro trains all models in S for one epoch and passes control back to the AutoML procedure for convergence/termination/addition evaluations and gets a potentially modified set S' for the



Method	Runtime (hours)	Avg. GPU Utilization (%)
Asynchronous PS	189.97	8.62
Horovod	54.24	92.08
Model Averaging	19.70	72.05
Celery	19.45	73.15
MOP	17.68	79.76

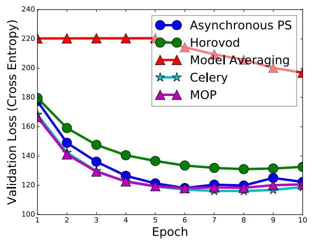


Figure 2: Makespans, GPU utilization, and learning curves for a workload with $|S| = 16$ on an 8-node GPU cluster.

next epoch. This templated approach allows Cerebro to be highly general: we can support all popular convergence definitions for training (fixed or unknown numbers of epochs) and many popular AutoML search procedures.

To support arbitrary ANN/AutoGrad frameworks, we will have an extensible architecture with custom handlers to delineate framework-specific aspects as model checkpointing and restoring. Users will have to provide the data ETL pipeline, ANN architecture definition, training and testing criteria, and the SGD-based optimizer as functions in their chosen ANN framework. Cerebro will automatically schedule these functions, checkpointing, and restoring on the workers. Overall, this planned implementation will make Cerebro highly general and support all forms of data types, ANN architectures, and SGD-based optimizers.

Challenge: Heterogeneity, Elasticity, and Fault Tolerance We desire Cerebro to be robust to these issues. Cluster heterogeneity can arise when workers have different (numbers of) CPUs, GPUs, etc. Cerebro Scheduler must take these differences into account to avoid unexpected idle times due to stragglers. We also desire elasticity to let ML users add/remove workers on the fly (say, to reduce completion times) and support tolerance to worker faults.

Our Plan To handle heterogeneity (and support fault tolerance), we plan to support partial data replication and create a two-stage Scheduler. First, we minimize the maximum load on any worker by assigning job units to workers using a load balancing algorithm that uses our job unit costs and machine speed estimates. This assignment will then be used by our open shop Scheduler to produce the optimized schedule. Since our Scheduler operates at every epoch, fault tolerance and elasticity essentially come for free. When extra workers are added, include them for scheduling at the very next epoch. Workers can be removed in between epochs, as long as a replica of that partition is still available. Fault tolerance is easy to offer, since MOP anyway checkpoints models after sub-epochs. To detect worker failures, we will use periodic "heartbeat" messages. When a failure is detected, Cerebro will use a *recovery mode* wherein job units assigned on the failed worker will be reassigned to other workers based on replica availability after the current epoch is finished. Overall, Cerebro's unified templated Scheduler will robustly handle all these distributed systems-oriented issues.

4.4 Preliminary Results

We compare a prototype implementation of MOP on TensorFlow to many state-of-the-art systems on an 8-node GPU cluster with Nvidia P100 GPUs. We train 16 models on ImageNet for 10 epochs: 2 architectures (VGG16 and ResNet50), 2 batch sizes (32 and 256); 2 learning rates (10^{-4} and 10^{-6}), and 2 regularizers (10^{-4} and 10^{-6}). Figure 2 shows the makespans, average GPU utilization, and learning curves of the best model from each system.

MOP is over 10x faster than TF's in-built asynchronous PS. PS's GPU utilization is as low as 8%. MOP is also 3x faster than Horovod. However, Horovod has high GPU utilization close to 92%. This is because GPU utilization values for Horovod also captures the communication time as its communication utilizes GPU kernels. MOP's runtime is comparable to TF's model averaging (BSP-style) and Celery's (disk-aware) task parallelism. But model averaging does not converge at all. Celery and MOP have the best learning curves, which are also almost identical, but note that Celery has 8x the memory/storage footprint as MOP due to dataset copies. Horovod's convergence is slower due to the larger effective mini-batch in its synchronous gradient aggregation. MOP can also cache and reuse pre-processed partitions across models, while other systems redo pre-processing for every model. Overall, MOP is the most resource-efficient and still offers accuracy similar to sequential SGD.

Acknowledgments. This work was supported in part by a Hellman Fellowship and by the NIDDK of the NIH under award number R01DK114945. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

REFERENCES

[1] Max Jaderberg et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).
[2] Arun Kumar et al. 2016. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record* (2016).
[3] Lisha Li et al. 2016. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560* (2016).
[4] Mu Li et al. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*.
[5] Philipp Moritz et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*.
[6] Szilard Pafka. Accessed February 28, 2019. Big RAM is eating big data - Size of datasets used for analytics. https://www.kdnuggets.com/2015/11/big-ram-big-data-size-datasets.html.
[7] Timos K Sellis. 1988. Multiple-query optimization. *TODS* (1988).
[8] Alexander Sergeev et al. 2018. Horovod: fast and easy distributed deep learning in TF. *arXiv preprint arXiv:1802.05799* (2018).
[9] Gerhard J Woeginger. 2018. The Open Shop Scheduling Problem. In *STACS*.