Cerebro: A Layered Data Platform for Scalable Deep Learning

Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha
University of California, San Diego
{arunkk,snakanda,yuz870,s7li}@eng.ucsd.edu,{agemawat,knagrech}@ucsd.edu

ABSTRACT

Deep learning (DL) is gaining popularity across many domains thanks to tools such as TensorFlow and easier access to GPUs. But building large-scale DL applications is still too resource-intensive and painful for all but the big tech firms. A key reason for this pain is the expensive *model selection process* needed to get DL to work well. Existing DL systems treat this process as an afterthought, leading to massive resource wastage and a usability mess. To tackle these issues, we present our vision of a first-of-its-kind data platform for scalable DL, Cerebro, inspired by lessons from the database world. We elevate the DL model selection process with *higher-level APIs* already inherent in practice and devise a series of novel *multi-query optimization* techniques to substantially raise resource efficiency. This vision paper presents our system design philosophy and architecture, our recent research and open research questions, initial results, and a discussion of tangible paths to practical impact.

1 DEEP LEARNING: THE PROMISE, THE PAIN

Deep learning (DL) has revolutionized machine learning (ML), powering modern speech recognition, machine translation, e-commerce, radiology, and more. The successes of DL, primarily at Web giants, has led to high interest among domain scientists, enterprises, smaller Web companies, and even healthcare firms in trying DL for their analytics tasks. The excitement is mainly due to DL's power to unlock valuable unstructured data, e.g., sensor time series in public health, satellite images in agriculture, and text corpora in finance. While tools such as Spark and Flink tame the "Volume" and "Velocity" aspects of "Big Data" analytics, the "Variety" issue is still largely open. DL is key to taming this third V for analytics. Naturally, new tools such as TensorFlow aim to make DL easier to use. Cloud computing is also making GPUs more accessible.

In spite of all this progress, DL is still too hard to use at scale, resource-intensive, and costly. While off-the-shelf pre-trained models mitigate this issue for a few well-defined prediction tasks, most ML users still need to build custom models for bespoke datasets and applications, either manually or with AutoML procedures. This process is painful in large part due to DL's *flexibility*, which is unprecedented in ML: the input/output can be almost any data structure and internal layers can be built in many ways [27]. Such end-to-end learning makes DL more accurate but it also makes DL training unusually resistant to software commodification. Even the cloud "whales" do not offer much beyond basic IaaS support for *DL training* (although SaaS exists for some *inference* tasks).

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2021.
11th Annual Conference on Innovative Data Systems Research (CIDR '21), January 10–13, 2021, Chaminade, CA, USA

DL's flexibility also underlies a key reason for its cost: *model selection*, an empirical process comparing alternate training configurations with varying input/output representations, neural architectures, and/or hyperparameters [37]. This process is *unavoidable* due to fundamental theoretical reasons on the bias-variance-noise tradeoff in ML accuracy [57] and the bespoke nature of many tasks. Without proper model selection, users will just squander the power of their labeled data, hurting the application.

Alas, there exist no reliable ways to tell the accuracy of a configuration on a given dataset without actually running it, leading to mass empiricism. If one tuned, say, 3 hyperparameters with, say, 4 possible values each, the number of configurations is 64 already. Multiply that with more trials for varying input/output representations and neural architectures. No wonder practitioners routinely build 10s to even 1000s of models to pick just one [6]. Overall, model selection is critical for effective use of DL but it massively amplifies DL's cost and energy footprints and impedes usability.

1.1 What do Existing Tools Lack? Why?

DL systems such as TensorFlow meet a rather low-level need: specify a neural architecture, train it with stochastic gradient descent (SGD) or its variants, and run inference [12]. They do this rather well, at least on a single node. But the user must also specify the model selection process somehow, which is mainly why APIs such as Keras [21], AutoKeras [31], and Hyperband [42] arose. *Model selection is an afterthought in DL systems*, a burden dumped on the user to be handled with ad hoc outer loops and libraries, not as the mission-critical process it is. This design has led to two issues:

(1) **Usability mess:** Without first-class support for model selection, users handle extra code and metadata in ad hoc ways. This often results in unreproducible executions and stymies cross-organizational collaboration. While some cloud vendors and ML platforms are starting to support this process better, it is still largely piecemeal and often limited to only basic tasks such as hyperparameter tuning.

(2) **High resource wastage:** Different configurations tried in model selection often overlap substantially on data and/or computations. Ignoring this structure leads to high wastage of compute, storage/memory, and/or network resources, especially at scale. This issue is *orthogonal* to both making single-model training faster and to devising better AutoML procedures to navigate the search space.

While the Googles, Facebooks, and Amazons of the world may gluttonously throw 100s of GPUs/TPUs and engineers at their DL tasks with low regard for overall resource efficiency, most other DL users cannot afford that, especially on pay-as-you-go clouds. And while some Web giants may ignore reproducibility for some ML tasks, it is a showstopper issue for most enterprises and domain scientists. There is a pressing need to meet these concerns of the fast-growing user base of DL outside Web giants and cloud whales.

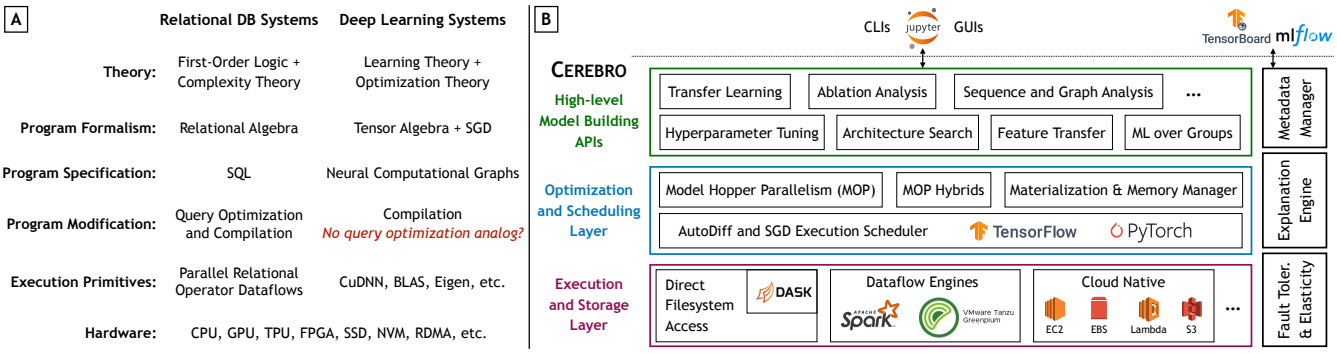


Figure 1: (A) Intellectual stack analogy of RDBMS and DL systems. (B) Our envisioned model selection-first architecture.

1.2 Our Vision

To tackle the resource efficiency and usability issues in one go, we envision a first-of-its-kind DL platform that puts model selection front and center: Cerebro. We draw upon three system design philosophies from the database world: *higher-level specification*, *layered system design*, and *query optimization*.

High-level APIs for Model Selection. Cerebro is rooted in a key ML observation: *the model selection process is not ad hoc but has rich structures* due to ML-theoretical and/or data type-specific reasons. We identify such structures from our conversations with 30+ ML practitioners in various settings, key ML papers, and our own experiences building DL applications. We distill our lessons into design patterns codified as higher-level *model building APIs* that enable us to decouple the *what* of model selection ("logical" level) from the *how* of its execution ("physical" level).

Layering with the Neural Query Model. We reimagine DL executions (both training and inference) as "queries" in what we call the *neural query model*. It is more coarse-grained than the neural computational graph abstraction of DL tools such as TensorFlow but more fine-grained than APIs such as Keras. This in-betweenness lets us reuse existing DL tools *as is* (without modifying their internal code) for the purposes they serve well, while supplanting other functionalities. Specifically, we believe the weakest link in today's DL tools is *how inefficiently they manage data at scale during query execution*, which needlessly bloats DL's resource footprint.

Multi-Query Optimization. The high-level APIs enable Cerebro to *look across* training configurations during model selection and cross-optimize them, inspired by *multi-query optimization* (MQO) in the database world [55]. We devise a suite of novel MQO techniques based on the access patterns and computational properties of neural queries. All this substantially raises resource efficiency (often by over 10x). Our MQO techniques are also applicable to multiple DL tools, storage backends, and execution environments in a unified manner, thus making Cerebro highly portable.

1.3 Case Study: UCSD Public Health Data

We have first-hand experience of both the pains of scalable DL and Cerebro's potential for impact thanks to an ongoing collaboration with UC San Diego public health researchers. Our collaborators wanted to try DL for identifying activities of human subjects (e.g., sitting, standing, stepping, etc.) from body-worn accelerometers. They have labeled data for 600 people, about 864 GB. No off-the-shelf models suit their task semantics. So, based on the literature on DL-based time series classification, we tried many DL models with TensorFlow, including deep CNNs, LSTMs, and composite CNN-LSTMs. This required repeated model selection on a shared GPU cluster at UCSD. We systematized our neural architecture selection and hyperparameter tuning with Keras-style APIs. TensorFlow worked well on one node but scaled poorly on the cluster because its parallel SGD execution schemes are slow. Thus, we devised a novel execution scheme for SGD on sharded data (Section 4.1).

Our collaborators also kept changing the prediction windows (e.g., 5s vs. 15s) and label semantics (e.g., sitting vs. not sitting), requiring us to rerun model selection over and over. This underscores the importance of resource efficiency and the *throughput* of this process. This was a key motivation for building Cerebro and migrating this workload to it. We were then able to easily explore dozens of models; they had accuracies between 62% and 93% for a binarized task. This underscores the importance of rigorous model selection. The best DL models gave a large lift of 10% over their prior well-tuned RandomForest model built on hand-engineered features. The models built with Cerebro on their data now offer the highest accuracy for this task in their field.

2 OUR LAYERED ARCHITECTURE

By comparing the intellectual stack of RDBMSs and DL systems as shown in Figure 1(A), we see a major gap in the DL systems landscape: no query optimization. We seek a system that makes it easy to infuse novel DL-focused query optimization techniques for model selection. Our design philosophy is three-pronged:

(1) **Integrationist**: Integrate seamlessly with production ML environments without changing the DL tool's internal source code. Enable retention of auxiliary ML tools for visualization and governance such as TensorBoard [10], TFX [14], and MLFlow [19]. This design decision makes it easier for DL practitioners to adopt Cerebro. Such decoupling also lets us piggyback on complementary advances by other communities, e.g., better DL compilers or new hardware accelerators, without changing Cerebro.

(2) **Modular**: Enable users to adopt different functionalities and optimizations on a need-to basis instead of a monolithic RDBMS-style all-or-nothing stack. Not all users may need to use all of the model building APIs in Cerebro. This design decision enables Cerebro to support new capabilities and optimizations over time without losing backward compatibility.

(3) **Ecumenical**: Support multiple DL tools and storage/execution environments in a unified way so that users can pick whatever suits their application setting. This design decision enables Cerebro to benefit a wider range of DL users without needing to take sides in industry wars, e.g., TensorFlow vs PyTorch, in-DBMS vs filesystem-based, and cloud vs on-premise clusters.

Figure 1(B) illustrates Cerebro's architecture that conforms to the above design philosophy. Next we briefly explain each layer.

2.1 High-Level Model Building APIs

We need *not* reinvent higher-level APIs for DL model building: such design patterns are already ubiquitous in ML/DL practice [37]. We just hijack them to enable MQO underneath. Such APIs are popular because they allow *bulk specification* of many training configurations (configs) instead of one by one, improving usability. Some APIs are well-known: hyperparameter tuning and architecture search; we overload Keras-style grid/random search APIs for these. These sufficed for our public health use case. Some APIs are emerging, e.g., transfer learning, especially from CNNs and transformers; we devise intuitive APIs for these. Finally, we also devise new APIs for batching configs across subsets of the data, features, or model parts to enable better debugging and sharing of work across users within an organization. We structure the APIs in a hierarchy with hyperparameter tuning being reused by each. By supporting different workloads this way, we meet our design goal of modularity.

2.2 Optimization and Scheduling Layer

Our vision's key technical novelty lies in this layer. We rewrite the execution of a set of training configs given in bulk at both logical and physical levels. So, we supplant a DL tool's distributed execution layer, a key source of their resource inefficiency at scale, with our novel decoupled and optimized approach. To this end, we define our central query execution abstraction: a *unit query* for training, which is node-local SGD for one epoch on one shard. A full training neural query then is a sequence of unit queries across shards and that sequence is repeated every epoch. By decomposing a training config's execution into unit queries, we obtain the following template for infusing our optimizations in a way that meets our design goals of being integrationist and ecumenical.

(1) We retain the DL tool (e.g., TensorFlow or PyTorch) as is for a unit query to the extent possible. That is, we use it for its most important and complex capability: run mini-batch SGD with backpropagation based on automatic differentiation, specifically, on only one data shard on one local worker.

(2) We automatically inject directives to the DL tool for *checkpointing* and *restoring* model state in between unit queries. All of the popular DL tools offer APIs for checkpointing and restoring DL training state, e.g., [9].

(3) We are now free to devise and infuse our MQO-style optimization techniques *around* and *across* unit queries between workers in myriad ways: rescheduling and reordering, staging materializations of inputs/outputs/intermediates, batching unit queries, and so on.

Using the above template, the heart of Cerebro's orchestration of unit queries is a new form of parallel SGD execution we call *model hopper parallelism* (Section 4.1). It is a hybrid of task parallelism and data parallelism. We also devise other model selection-oriented MQO techniques (Section 4.2) and identify many open opportunities for new MQO techniques (Section 4.3). All of our techniques exploit the data access patterns of unit queries and/or reduce computational redundancy to raise GPU utilization, reduce distributed memory/storage footprints, and/or reduce network costs.

2.3 Execution and Storage Layer

This includes datasets and hardware for compute, storage/memory, and networking along with their APIs. In line with our design philosophy, we support multiple types of backends: go directly to the filesystem (e.g., EXT + NFS or HDFS) and OS to schedule unit queries; use the mediated APIs of Dask, parallel RDBMSs (e.g., Greenplum), or parallel dataflow systems (e.g., Spark); or use IaaS APIs for cloud-native execution. The UCSD public health use case described in Section 1.3 used the first type of backend: EXT + NFS on a shared GPU cluster at UCSD operated with Kubernetes.

Our goal is *not* generic resource/cluster management but rather building a resource-efficient data platform for DL model selection. Thus, we do not aim for innovations at this layer but rather focus on the layer above.

3 ONGOING SYSTEM IMPLEMENTATION

3.1 Approach and Status

We are implementing Cerebro as a deliberately lightweight Python module with APIs in Python and connectors for the backends. Our layering lets us outsource most heavy-duty storage and resource handling. We currently support hyperparameter tuning and architecture selection (both manual and automatic) over the following backends: filesystem, Spark, and Greenplum. We recently published a full research paper on the current capabilities of Cerebro [52]. Next we explain the role of two core components that matter for MQO. Section 5 will present the other components in Figure 1(B).

Optimizing Scheduler: This component produces an optimized execution plan for a given model selection workload. It stages out and rearranges unit queries to place them on workers as per the plan along with related computations (e.g., for validation errors).

Materialization and Memory Manager: This component caches preprocessed data shards and other local intermediate materialized data based on the the Scheduler's decisions. It is aware of all levels of the memory hierarchy: DRAM, local disk, and remote storage.

On the filesystem backend, we use XML-RPC client-server package for inter-node communication. The Scheduler runs in the client; each worker runs a lightweight agent which also runs a service for communication. We use push-based scheduling to assign work.

Unit query execution and data loading on a local worker is done by the DL tool itself.

3.2 Usage Walkthrough

Cerebro is easy to install with pip. Its APIs resemble popular DL training APIs like Keras with minimal extensions for our system setting. Listing 1 shows a full usage example in our API. The user performs 4 steps:

(1) Specify the execution backend (e.g., Spark) and the storage area (e.g., HDFS path).

(2) Define a function to generate a *ModelTrainer* object given a specific hyperparameter combination. The *ModelTrainer* object encapsulates the neural architecture and the SGD-based training procedure (e.g., Adam).

(3) Specify the input dataset file or source. It could be pre-processed for DL model training. This code is skipped for brevity.

(4) Launch the model selection workload by specifying the hyperparameter search space and a canned model selection procedure of their choice, e.g., grid search (shown).

After the model selection process ends, the best model based on the user-provided criteria is returned. Our APIs also let the user specify locations to optionally log all model artifacts (not just final results) and execution metadata for detailed post-hoc inspection, debugging, governance, etc. Although the usage example shows a simple grid search, Cerebro already supports several other model selection procedures, including random search and some state-of-the-art AutoML procedures such as Hyperband, ASHA, and TPE [52]. More details and examples of Cerebro's APIs are available on our system documentation webpage [2].

3.3 Extensibility

Cerebro is designed to be easily extensible along two aspects: (1) DL tools and (2) model selection procedures. We briefly explain each next.

We use a *handler-based approach* for our components to talk to DL tools. This makes it relatively simple for Cerebro to support new DL tools. The contract is that the DL tool must support a sequential data access pattern and have APIs for checkpointing and restoring model state. Cerebro uses those APIs to stage out unit queries. This also means Cerebro can technically be extended to support *any* ML model trainable with any SGD-based method.

Our components talk to the model selection procedure via a *per-epoch scheduling template*. The contract for a model selection procedure is as follows: give our Scheduler a *set* of training configs that will all be run for one epoch at a time. Cerebro (re)schedules unit queries at every epoch to enable the model selection procedure to kill and/or add configs on the fly. This enables Cerebro to support both one-shot specification of all configs (e.g., grid/random search) and adaptive creation/killing of configs (e.g., like in Hyperband or HyperOpt) in a unified manner. ML developers can easily plug in any future AutoML procedure into our scheduling template and expand our model building APIs. An obvious limitation here is that if the AutoML procedure is sequential and explores only one config at a time, it will not benefit much from Cerebro.

Listing 1: Example Usage of Cerebro's APIs

```
from cerebro.backend import SparkBackend
from cerebro.storage import HDFSStore
from cerebro.keras import ModelTrainer
from cerebro.tune import GridSearch

## Step 1: Execution and storage backends ##
execution = SparkBackend(num_workers=3)
storage = HDFSStore(path='hdfs://...')

########## Step 2: Input dataset ##########
train_df = ...


# Step 3: ModelTrainer generating function #
# Input: Parameter instance
# Output: ModelTrainer
def trainer_gen_fn(params):
    # Initialize a Keras model
    model = ...
    optimizer = Adam(lr=params['lr'])
    loss = 'binary_crossentropy'
    estimator = ModelTrainer(
        model=model,
        optimizer=optimizer,
        loss=loss,
        metrics=['acc'],
        batch_size=params['batch_size'])
    return estimator


##### Step 4: Launch model selection ######
# Define parameter search space
search_space = {
    'lr': [0.01, 0.001, 0.0001],
    'batch_size': [16, 256, 16]
}

# Instantiate a model selection object
model_selection = GridSearch(
        execution=execution,
        storage=storage,
        trainer_gen_fn=trainer_gen_fn,
        search_space=search_space,
        num_epochs=10,
        validation=0.25,
        evaluation_metric='loss')

# Launch model selection
# Output: Best model
best_model = model_selection.fit(train_df)
```

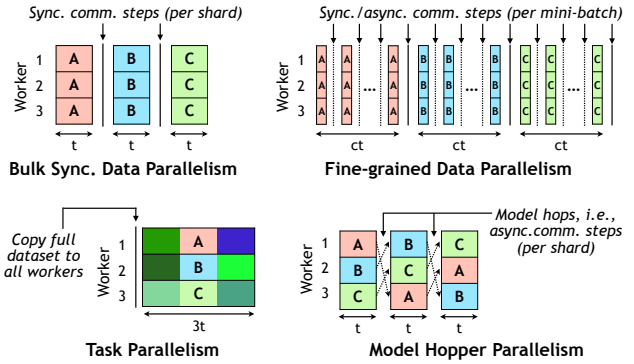


Figure 2: Gantt charts for a simple example with 3 training configs (A/B/C) and 3 workers (manager nodes not shown). Each config's single-node runtime is $3t$. Data-parallelism and MOP run on sharded data, unlike task parallelism. The overhead factor $c > 1$ depends on the network; in [52], we saw it was up to 3.

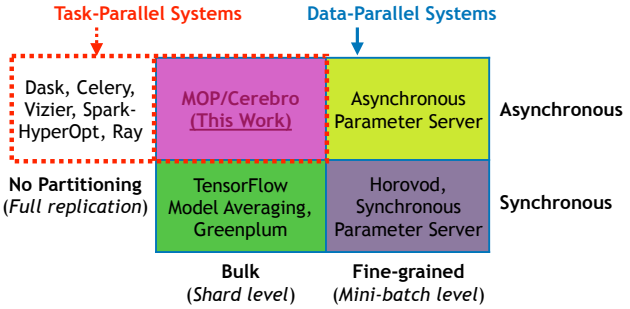


Figure 3: MOP is the first known form of *bulk asynchronous* parallelism in data systems (based on [52]).

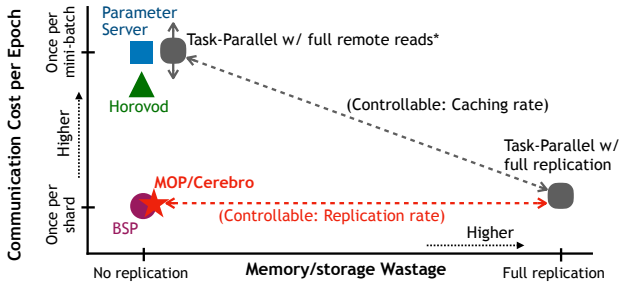


Figure 4: Comparing two key aspects of resource efficiency (from [52]). Dashed line represents a controllable parameter. *Task-parallelism with remote reads has varying communication costs based on dataset size.

4 OUR EXECUTION OPTIMIZATIONS

4.1 Execution Core: Model Hopper Parallelism

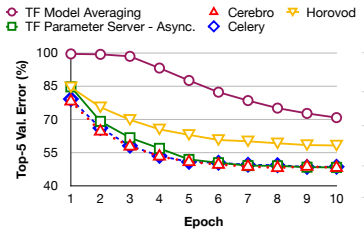
At the heart of Cerebro is a new form of parallel SGD execution we call model hopper parallelism (MOP). It is a hybrid of task parallelism and data parallelism, representing a physical level of multi-query optimization tailored to SGD's data access patterns. We briefly recap the prior approaches before explaining MOP. Later we explain their cons against MOP. Figure 2 illustrates the approaches

Background on Prior Approaches. Given multiple training configs in one go, task-parallel tools (e.g., Dask, Celery, and Ray [47]) train a config entirely on one worker, with different configs placed on different workers in parallel. The entire dataset is copied to every worker (full replication) or read from remote storage (e.g., S3) at each epoch for each config. In contrast, data-parallel tools operate on sharded data and train one model at a time on the whole cluster. Bulk synchronous parallel (BSP) tools perform SGD model averaging, which communicates model updates *once per shard* after each epoch. But because BSP has poor SGD convergence behavior for non-convex losses, it is a poor fit for DL [52]. Horovod [56] and Parameter Server [43] mitigate that issue by communicating model updates more frequently, usually *once per mini-batch*. This improves convergence efficiency but their communication costs also go up dramatically because a single shard may contain 100s to even millions of mini-batches depending on the batch size.

Basic Idea of MOP. MOP is rooted in two ML insights: (1) Model selection typically has a high degree of parallelism (multiple configs given at once); and (2) SGD is robust to the randomness in data ordering [24]. It is also rooted in a system insight: sharding is the most scalable way to parallelize computations on large data.

Based on the above insights, MOP works as follows. Given a dataset, randomly shuffle the dataset, shard it, and give each worker a single shard. Divide the training epoch of each config (training neural query) into its unit queries, i.e., sub-epochs on shards. The Scheduler then places unit queries on workers. After a sub-epoch completes, that model is checkpointed, "hops" to another worker, and resumes training the next sub-epoch. A config visits every worker exactly once to finish one whole epoch. Different models hop across workers asynchronously in parallel as determined by the Scheduler. Figure 2 illustrates the hopping for one epoch. Overall, all configs are trained using pure *sequential SGD*, albeit in parallel. Repeat this whole process for every epoch. Optionally, the dataset can be reshuffled in between epochs just like in other approaches.

Comparative Analysis. Conceptually, MOP is the first known form of *bulk asynchronous* parallelism as Figure 3 shows. It is inspired by "process migration" in multiprocessing OS, albeit adapted to SGD on sharded data. As we formally show in [52], MOP's communication complexity is the same as BSP and much lower than Horovod or Parameter Server, translating to up to 10,000x network cost reductions in practice. Figure 4 conceptually illustrates MOP's benefits. Like task parallelism, MOP offers high throughput for model selection but unlike MOP, task parallelism with replication is highly wasteful of memory/storage. For instance, our 0.9 TB dataset in Section 1.3 will bloat to a massive 7 TB on an 8-node cluster, unethically hogging a shared academic resource. While remote reads can mitigate this issue for task parallelism, it bloats network costs massively, between 100x and 1000x based on the number of configs and epochs. Finally, BSP has poor convergence but MOP and task parallelism have ideal convergence behavior due to their use of sequential SGD. Finally, due to the coarser shard-level scheduling in MOP, its entire training process is trivially reproducible, unlike,



System	Runtime (Hours)	GPU Util. (%)	Space Footprint
TF PS-Async	19.00	8.6	250 GB
Horovod	5.42	92.1	250 GB
TF Model Averaging	1.97	72.1	250 GB
Celery	1.72	82.4	2000 GB
Cerebro	1.77	79.8	250 GB

Figure 5: (Based on [52]): Learning curves and per-epoch time/space efficiency for model selection on ImageNet.

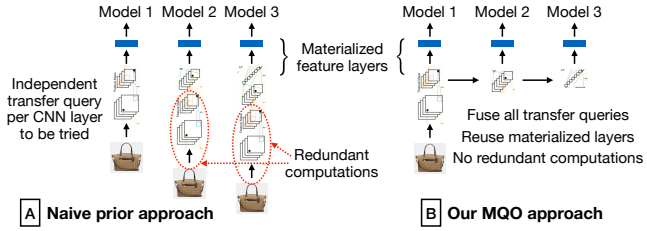


Figure 6: Comparing the naive prior approach with separate CNN feature transfer queries against our MQO approach.

say, asynchronous Parameter Server. *Overall, MOP holistically optimizes for all 3 resources–compute, storage/memory, and network–to offer near-optimal overall resource efficiency for distributed DL model selection.*

Empirical Validation. To validate MOP's benefits, Figure 5 shows a key result on the ImageNet dataset from our recent full research paper [52]. We ran a model selection workload with 16 configs, including 2 different neural architectures and standard hyperparameter tuning on an 8-node GPU cluster. Cerebro and task-parallel Celery are the fastest and have best convergence but note Celery's 8x larger space footprint. TF model averaging (BSP-style) has poor convergence, while Parameter Server and Horovod are respectively 10x and 3x slower than Cerebro due to their mini-batch-level communication costs. Overall, our above points on MOP's benefits on both resource efficiency and accuracy are validated.

Cerebro's Scheduling Problem. MOP is highly general because it has no synchronization barrier between configs or shards within an epoch. Cerebro's Scheduler thus enforces only two invariants: (1) A config is trained at most one sub-epoch at any given time; and (2) A config visits each worker exactly once within an epoch. In full generality, Cerebro's scheduling formulation minimizes makespan given the above constraints. It is NP-Hard in the number of configs (which can even be 100s) via a reduction from open shop scheduling [52]. But we find that a simple *randomized scheduler* is effective, efficient, and easy to implement. At an epoch boundary for the model selection workload, our randomized scheduler randomly places a unit query (that is yet to be run) on an idle worker, while respecting the constraints. It is also highly flexible and enables Cerebro to easily support heterogeneous resources, heterogeneous configs (say, disparate neural architectures), partial replication of data shards (say, for more parallelism), fault tolerance, and elasticity–all in a unified manner. Basically, we can map them all to constraints in our Scheduler for handling unit queries.

4.2 Recent and Ongoing Research

We briefly discuss some recent work by us on more MQO techniques for DL model selection that are being integrated into Cerebro.

4.2.1 ***Feature Transfer and Transfer Learning.*** A major reason for the success of DL is *transfer learning*, a model selection methodology that is now ubiquitous for image/video analytics [59]. The basic ideas is to transfer "knowledge" from a prior source task to a new downstream target task. A specific form of transfer learning called *feature transfer* has made CNNs especially popular. It works as follows: use a pre-trained CNN (say, trained on ImageNet), remove its "head" up to a specific layer, and use the rest of it to featurize images. This reduces both the amount of labeled data and resources needed for the downstream task, since it avoids training a large model from scratch.

Alas, feature transfer faces a bottleneck at scale that hurts efficiency, reliability, and usability: no single layer of a pre-trained CNN is always best for downstream accuracy. DL users usually compare alternate layers, which is yet another form of model selection, specifically for feature engineering [37]. For each layer of interest, CNN inference is run till that layer for all images and a downstream model is trained on those materialized output features. This is repeated from scratch for every other layer of interest, yielding many downstream models. But this process has high computational redundancy because CNN computations to obtain a higher layer are a *superset* of those to obtain a lower layer. Figure 6(A) illustrates the process and the redundancy. Naturally, one may ask why not materialize and cache all layers of interest in one go. CNN features can be even 100x larger than base images. So, caching all layers of interest in one go could raise memory pressure and cause system crashes, unless the user manages memory carefully. Overall, scalable feature transfer is a tedious and slow process.

To tackle the above issues, we recently built the Vista system [48]. It raises the specification of feature transfer to a higher level and automatically optimizes execution, just like Cerebro. We avoid computational redundancy by devising a novel MQO-oriented plan that stages out the materialization of feature layers, as Figure 6(B) illustrates. Essentially, Vista chops up the CNN inference neural query into sub-queries based on the layers to transfer, automatically creates a materialized view of a lower layer, and automatically reuses that view for the next higher layer. Prototyped on top of Spark and TensorFlow, Vista carefully apportions distributed memory to maximize efficiency and avoid memory-related system crashes. Empirically, we found that Vista's MQO-based approach to feature transfer is more seamless and faster than baselines by between 2x to 10x on real workloads.

In ongoing work, we are folding Vista into Cerebro, fusing it with MOP and adding support for more general transfer learning in DL. Transfer learning has recently exploded in popularity for text data in natural language processing (NLP) too thanks to giant pre-trained transformer models such as BERT and GPT [16, 23]. Such models present new challenges because NLP users often stitch multiple feature layers from transformers in ad hoc ways, unlike with CNNs.

The memory blowups are also larger, e.g., BERT features can be 4000x the size of base text! How to abstract this general problem in our neural query model? How to generalize memory management and materialized views in Cerebro without losing MOP's benefits? Building on our experience with Vista, we are actively working on evolving these components of Cerebro to enable comprehensive support for resource-efficient deep transfer learning on arbitrary data types with arbitrary DL models.

4.2.2 ***Integration with Data Systems.*** The DB community has worked on adding support for ML in DBMSs and dataflow systems for over 20 years [18, 34]. MADlib [29] and MLlib [46] are perhaps the most well-known and widely used among such tools. While research on ML over data systems has waxed and waned over the years, a sizable chunk of enterprise ML users, especially business analysts, still prefer to access ML from the data system environment. For instance, our conversations with VMware revealed that at least 20% of Greenplum's customers still use MADlib. Cloud DBMSs such as BigQuery and Redshift have also been adding more ML integration [3, 11].

Motivated by the above, we have also been integrating MOP/Cerebro with data systems, in particular, Greenplum and Spark. Such tools are natively data-parallel, which suits Cerebro. Such tools also natively support distributed data shuffling, e.g., via ORDER BY RANDOM in SQL [24]. But the challenge is their BSP paradigm may not gel with MOP's asynchrony (see Figure 3). Coincidentally, Spark recently added native support for asynchronous tasks on Spark workers [22]. We quickly leveraged that capability to integrate Cerebro with Spark without needing to change Cerebro's Scheduler. This integration is open sourced [2]; we recommended this version over the filesystem-based version for general use of Cerebro. This is due to Spark's complementary strengths–efficient data shuffling, low-level resource provisioning, and ML governance with MLFlow–all of which can help DL users adopt Cerebro-Spark.

As for Greenplum, MADlib already had in-RDBMS DL support for invoking Keras and TensorFlow via special user-defined functions that marshall mini-batches from the database to TensorFlow [7]. But that approach was tied to model averaging, which as Figure 5 shows, converges poorly for DL. Thus, the MADlib team collaborated with us to explore new variants of MOP's basic randomized scheduling to suit the BSP restriction. We have developed 3 canonical approaches that make different tradeoffs on runtime efficiency, storage wastage, and ease of governance: a fully in-DBMS approach (data is in DB and Keras is invoked from SQL), a partially in-DBMS approach (data is in DB and Keras is invoked from outside), and a novel in-DB but not in-DBMS approach, apart from the standard out-of-DBMS approach by exporting data to Cerebro-Spark. Our comparative empirical analyses show that it is non-trivial to meet all practical desiderata well and a Pareto frontier exists [66]. One of our integration approaches (fully in-DBMS) has been adopted by VMware and released as part of Apache MADlib [1].

4.3 Open Research Questions

We now briefly discuss several new open research questions that we plan to tackle in the near future in the context of Cerebro.

4.3.1 ***Model Parallelism and Batching.*** A cliche in DL is that GPU memory keeps trailing DL model sizes [25]. The common way to resolve this issue is "model parallelism": shard the large DL model across devices and communicate updates frequently. Alas, this approach has poor scalability behavior: the runtime speedup trends are often quite sub-linear or worse [30]. Of course, MOP per se does not resolve this issue. However, we observe that Cerebro's model selection context offers new avenues to stage out computations due to the higher degree of parallelism. At the other extreme, small DL models are also common (e.g., for IoT), and they substantially under-utilize GPU capacity. Batching models on the GPU can raise resource efficiency. While there is some recent work in this space [53], they modify the internals of the DL tool. We believe new approaches that avoid such modifications can help. Overall, how to generalize Cerebro to support flexible hybridization of model-, data-, and task-parallelism? How to unify it with model batching and integrate MOP with staged backpropagation for unit queries without modifying the DL tool?

4.3.2 ***More MOP Hybrids.*** MOP inherits a con of task parallelism: if the degree of parallelism of the model building task is less than the number of workers, some workers will go idle. To mitigate this, we hybridized MOP with data-parallel Horovod. Interestingly, our early results showed that Cerebro is actually faster with only half as many workers as Horovod–this is due to the latter's inherent communication overheads [52]. More careful data repartitioning can help mitigate this issue in the MOP-Horovod hybrid. Another hybrid we are pursuing is MOP with BSP model averaging but only for the latter epochs. The intuition is that model averaging works fine for SGD on convex loss surfaces [67]. Although DL is highly non-convex at the outset, some models could behave more convex-like near a local minimum in the latter epochs.

4.3.3 ***More High-Level APIs.*** The model building APIs we have discussed so far are not exhaustive–we think it is impossible to make a complete list. But as adoption of DL grows, newer applications will emerge and new design patterns for model building will appear. These patterns may be twists on prior model selection patterns and/or contain new forms of sub-tasks that overlap in data and/or computation. Regardless, we believe the general system template we laid out for Cerebro will be a fruitful and impactful direction for devising new DL-specific MQO techniques to raise resource efficiency in DL applications. We offer a few more key examples of emerging model building APIs that fit our template. We believe MQO can benefit all of these settings.

Ablation Analyses. The goal here is model diagnostics–understand what parts of the model really mattered for accuracy. It can be seen as a post-hoc twist on architecture selection. Nevertheless, it too can be cast as a form of model selection, combining architecture and hyperparameter search in a new way.

Learning over Groups. The goal here is to build separate models for separate groups within the dataset, e.g., different models for different states in a product recommendation workflow. This could help boost accuracy or could even be mandatory due to business rules. Such group-oriented training tasks also overlap on the data, representing yet another avenue to better optimize model selection over groups.

Sequence and Graph Data. Time series and video data are now killer use cases for DL. Likewise, graph neural networks (GNNs), and their fusion with CNNs, are exploding in popularity for various geospatial and domain science applications [60]. Such applications have non-traditional data access patterns and manipulations to run SGD, e.g,. configuring and chunking time windows in time series or retrieving features of a node's neighbors in a graph. We believe these access patterns can be exploited better in the model selection context to improve overall resource efficiency and usability.

4.3.4 ***Cloud-Native Execution.*** Finally, as analytics workloads increasingly move to the cloud, we plan to enable cloud-native execution for Cerebro in all major renting paradigms: On-Demand, Spot, and Serverless. Each paradigm poses new challenges due to its different kind of elasticity and resource constraints, e.g., interruptions in Spot instances and memory constraints in Serverless. But we observe that MOP/Cerebro is perfectly suited for all kinds of interruption-heavy and delay-tolerant settings because fault tolerance is in its very DNA: checkpoint-hop-resume. In contrast, approaches like Horovod are notoriously difficult to port to such settings due to their tight coupling of workers [5]. Overall, we believe Cerebro can offer near-optimal resource efficiency for DL in the cloud too and help cut its cost and energy footprints. Looking further out, we will also revisit Cerebro's design to factor in microeconomics of cloud resource pricing. Ultimately, we seek to enable DL users to easily traverse the entire Pareto surface of runtime, total cost, and accuracy based on their application-specific constraints.

5 CROSS-LAYER AND OTHER COMPONENTS

Apart from the components handling model building, we also envision three key cross-layer and other components in Cerebro.

(1) **Fault Tolerance and Elasticity Manager.** This will monitor workers during execution and restart a unit query if its worker fails. It will also enable elastic adding/removing of workers. These are all possible thanks to the coarser granularity of MOP, which incurs negligible overhead for checkpointing between unit queries. This component will be especially critical for cloud-native execution.

(2) **Metadata Manager.** This will transparently track and log all execution-related metadata: specification details, per-epoch metrics, per-model metrics, and overall results and artifacts for all training configurations executed via our model building APIs. These will be stored as standard tables. This component will connect with auxiliary ML usability tools such as MLFlow, TFX, and TensorBoard as mentioned before.

(3) **Explanation Engine.** This will offer "explanation" methods to help debug the model building process and validate selected models. We will include both popular per-example perturbation-based approaches and training set-based debugging. Such inference-focused computations will also be sped up with MQO techniques. For instance, we showed recently that a perturbation-based approach called occlusion-based explanation (OBE) for CNN predictions could be cast as multi-query execution [49]. We devised new incremental view maintenance and MQO techniques to reduce computations and runtimes of OBE. In future work, we will support such functionality for more data types and DL architectures.

6 COLLABORATIONS AND IMPACT PATHS

Back to UCSD Public Health. The DL models built with Cerebro are being applied to more cohorts, including people in assisted living facilities and people with obesity, to help them live healthier lives. More workloads on this front are moving to the cloud, making Cerebro more crucial. We are delighted to be a part of this impactful collaboration between computer scientists and domain scientists that is advancing both fields and also helps broader society.

More Domain Sciences. We are also speaking with more domain scientists at UCSD and a few other universities who are interested in exploring DL for their large-scale analytics tasks. So far, we have identified the following applications for scalable DL: high-resolution satellite imagery to study gentrification in economics, as well as wildfire evolution; time series representation learning in IoT-based mHealth; multimodal social media analytics fusing graph, text, and tabular data for political science; and high-resolution mouse brain videos to study brain evolution in neuroscience. We believe Cerebro can help all these applications and hopefully over time also help standardize the very practice of DL model selection in these fields.

Open Source and Industry Collaboration. As mentioned earlier, Cerebro is fully open-sourced to enable community adoption, extensions, and feedback [2]. Pivotal/VMware collaborated with us to adopt MOP for DL model selection in Apache MADlib with TensorFlow being run on Greenplum [1]. Their enterprise customers are interested in this integration for image analytics, fraud detection, and NLP use cases. We have also released Cerebro's integration with Apache Spark. Both of these integrations have already been presented at major industrial conferences: MOP-in-MADlib was presented by our MADlib collaborators at FOSDEM [4]; we presented Cerebro-Spark at the Spark+AI Summit [8]. We are also collaborating with VMware and also speaking with other cloud and DBMS vendors to help broaden our impact on industry.

More Vertical APIs. Our core design philosophy is to bring DL APIs closer to the levels at which DL users think. We plan to take this all the way with more vertical-specific APIs on top of Cerebro's APIs and eventually add more GUI support and application lifecycle oversight. Although our work does not address all aspects of commodifying DL, we believe our cross-stack usability-efficiency integrated approach is a crucial part of enhancing accessibility to DL in a wide swathe of scalable data analytics applications.

7 RELATED WORK

There is a spurt of recent work on cluster scheduling and resource management for DL. Examples include Gandiva [61], Tiresias [28], SLAQ [64], and [33]. However, they all focus on lower-level primitives such as hardware allocation and intra-server locality to reduce completion times. Cerebro's model selection-first approach is novel, complementary, and exists at a higher abstraction level, enabling us to devise novel cross-config MQO techniques such as MOP and materialized views for transfer learning. Cerebro can be combined with such lower-level cluster handling frameworks. There is also a long line of work on job scheduling in the operations research

and systems literatures [17]. Our goal is *not* to innovate on scheduling algorithms but to adapt known algorithms to realize our MQO techniques in our new DL systems setting.

There is also much work in the ML world on AutoML procedures, including new heuristics for hyperparameter tuning and neural architecture search. Cerebro is *complementary* to all of them because our goal is *not* to create new AutoML procedures but rather offer a unified and scalable data platform to execute them and to improve resource efficiency using MQO.

Many vendors offer hosted DL training or self-service tools with support for model selection workloads. Prominent examples include AWS SageMaker, GCP AutoML, Microsoft AzureML, DataRobot, H2O, and Determined AI. Tools such as Ray [47] and Vizier [26] also help scale hyperparameter tuning. But to the best of our knowledge, all of these tool follow the paradigm of *pure* task parallelism: naively copying and caching the entire dataset on all workers or naive remote reads for every epoch for every model. As Section 4.1 shows, such task parallelism is highly wasteful of resources: memory/storage or network or both. Cerebro departs from that paradigm by breaking the false dichotomy between task parallelism and data parallelism (Figure 3). Cerebro's novel layered architecture also enables us to support many more model building APIs than these other platforms in a unified stack, inspired by [37]. Finally, our architecture enables us to devise and infuse new MQO techniques to raise overall resource efficiency. That said, we believe it is possible for these other platforms to adopt MOP without deep code changes because MOP too is a form of task parallelism.

There is much work *pure* data parallelism to scale the training of a single model, including Parameter Server [43], Horovod [56], PyTorch DDP [45] and their many derivatives. MOP/Cerebro is *complementary* to all of them because our focus is on model selection workloads, not single-model training. ML theory teaches us that rigorous model selection is crucial for accuracy [57]. Anecdotally, however, some DL practitioners sometimes ignore that lesson and make do with training just one model due to time or cost pressures. Pure data parallelism can help in such cases. That said, we showed how MOP can easily be hybridized with such data-parallel ML systems to bridge the gap on the degree of parallelism because MOP too is a form of data parallelism.

We recently published full papers on MOP and our first version of Cerebro [52], as well as Vista [48]. The key novelty in this paper is: (1) Our holistic long-term vision for Cerebro's generalized model selection-first architecture; and (2) Showing how (multi-)query optimization is a promising new research direction in the context of DL systems. Our vision is based on synthesizing and building upon the lessons from our conversations with 30+ ML/DL users and developers across different settings: domain sciences, healthcare, enterprises, Web companies, cloud and DBMS vendors, and ML platform vendors. Our technical agenda is based on extrapolating from our decade-long experience with building, optimizing, and deploying multiples kinds of ML/DL systems [15, 35, 37], spanning multiple relevant topics: MQO for other DL workloads [49–51, 54], MQO for classical ML [13, 32, 62, 63], other query optimizations for classical ML [20, 36, 38, 44], new query execution techniques for ML/DL [41, 65], benchmarking the scalability of ML systems [58], and integrating ML into data systems [24, 29, 39, 40]

8 CONCLUDING REMARKS

The systems canon of the DB world is vast and deep. But the DB community must stop deluding itself that RDBMSs, dataflow systems, or mere cloud IaaS suffice for scalable ML and instead be more imaginative by innovating in settings that *actually matter* to ML users. We need to get out of our comfort zones to understand DL/ML in depth, work with real users, and navigate the whole system stack appropriately. Likewise, the emerging ML systems community must learn from the DB systems community's decades-long experience with commodifying scalable data software. Failing to achieve such cross-pollination will be a huge missed opportunity for both worlds, resulting in wasteful reinventions of the wheel by the research community, industry, and/or open source community, as well as massive wastage of resources, time, money, and energy by DL users. We presented Cerebro, our vision for a novel holistic DL platform based on these insights. It is a vehicle for marrying DB ideas and DL systems. We also discussed tangible routes to impact on DL practice. We hope our work inspires the DB community to partner with ML users and developers to help democratize DL-based data analytics.

Acknowledgments. This work was supported in part by a Hellman Fellowship, the NIDDK of the NIH under award number R01DK114945, an NSF CAREER Award under award number 1942724, and a gift from VMware. The content is solely the responsibility of the authors and does not necessarily represent the views of any of these organizations. We thank the members of UC San Diego's Database Lab and Center for Networked Systems, Loki Natarajan and our public health collaborators at UC San Diego, Frank McQuillan and the Apache MADlib/Greenplum team at VMware, Joe Hellerstein, Chris Jermaine, Sam Madden, Sebastian Schelter, and Dan Suciu for their feedback on this work and/or this paper.

REFERENCES

[1] Apache MADlib DL Model Selection with MOP, Accessed December 16, 2020. http://madlib.apache.org/docs/latest/group__grp__keras__run__model__selection.html.
[2] Cerebro Documentation, Accessed December 16, 2020. https://adalabucsd.github.io/cerebro-system/.
[3] Create, Train, and Deploy Machine Learning Models in Amazon Redshift Using SQL with Amazon Redshift ML, Accessed December 16, 2020. https://aws.amazon.com/blogs/big-data/create-train-and-deploy-machine-learning-models-in-amazon-redshift-using-sql-with-amazon-redshift-ml/.
[4] Efficient Model Selection for Deep Neural Networks on Massively Parallel Processing Databases, Accessed December 16, 2020. https://archive.fosdem.org/2020/schedule/event/mppdb/.
[5] Elastic Horovod, Accessed December 16, 2020. https://horovod.readthedocs.io/en/latest/elastic_include.html.
[6] Facebook FBLearnerFlow blog post, Accessed December 16, 2020. https://engineering.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/.
[7] MADlib Deep Learning, Accessed December 16, 2020. https://madlib.apache.org/docs/latest/group__grp__dl.html.
[8] Resource-Efficient Deep Learning Model Selection on Apache Spark, Accessed December 16, 2020. https://databricks.com/session_na20/resource-efficient-deep-learning-model-selection-on-apache-spark.
[9] TensorFlow Checkpointing, Accessed December 16, 2020. https://www.tensorflow.org/guide/checkpoint.
[10] TensorFlow TensorBoard, Accessed December 16, 2020. https://www.tensorflow.org/tensorboard.
[11] The CREATE MODEL Statement for Deep Neural Network (DNN) Models, Accessed December 16, 2020. https://cloud.google.com/bigquery-ml/docs/reference/standard-sql/bigqueryml-syntax-create-dnn-models.
[12] M. Abadi et al. TensorFlow: A System for Large-scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*,

pages 265–283, 2016.
[13] M. R. Anderson et al. Brainwash: A Data System for Feature Engineering. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 2013.
[14] D. Baylor et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1387–1395. Association for Computing Machinery, 2017.
[15] M. Boehm, A. Kumar, and J. Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.
[16] T. B. Brown et al. Language Models are Few-Shot Learners. In *NeurIPS*, 2020.
[17] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, 3rd edition, 2001.
[18] S. Chaudhuri. Data Mining and Database Systems: Where is the Intersection? *IEEE Data Engineering Bulletin*, 21, 1998.
[19] A. Chen et al. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, DEEM'20. Association for Computing Machinery, 2020.
[20] L. Chen, A. Kumar, J. F. Naughton, and J. M. Patel. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.*, 10(11):1214–1225, 2017.
[21] F. Chollet et al. Keras, Accessed December 16, 2020. https://github.com/fchollet/keras.
[22] Databricks. Introducing Apache Spark 2.4, Accessed December 16, 2020. https://databricks.com/blog/2018/11/08/introducing-apache-spark-2-4.html.
[23] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*, pages 4171–4186. Association for Computational Linguistics, 2019.
[24] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for In-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336. Association for Computing Machinery, 2012.
[25] Y. Gao et al. Estimating GPU Memory Consumption of Deep Learning Models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1342–1352. Association for Computing Machinery, 2020.
[26] 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.
[27] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[28] J. Gu et al. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.
[29] J. M. Hellerstein et al. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.
[30] Z. Jia, M. Zaharia, and A. Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *MLSys*. mlsys.org, 2019.
[31] H. Jin, Q. Song, and X. Hu. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, pages 1946–1956. Association for Computing Machinery, 2019.
[32] P. Konda, A. Kumar, C. Ré, and V. Sashikanth. Feature Selection in Enterprise Analytics: A Demonstration using an R-based Data Analytics System. *Proc. VLDB Endow.*, 6(12):1306–1309, 2013.
[33] S. Krishnan et al. Artificial Intelligence in Resource-Constrained and Shared Environments. *SIGOPS Oper. Syst. Rev.*, 53(1), July 2019.
[34] A. Kumar. ML/AI Systems and Applications: Is the SIGMOD/VLDB Community Losing Relevance?, Accessed December 16, 2020. https://wp.sigmod.org/?p=2454.
[35] A. Kumar, M. Boehm, and J. Yang. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017*, pages 1717–1722. ACM, 2017.
[36] A. Kumar, M. Jalal, B. Yan, J. F. Naughton, and J. M. Patel. Demonstration of Santoku: Optimizing Machine Learning over Normalized Data. *Proc. VLDB Endow.*, 8(12):1864–1867, 2015.
[37] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.*, 44(4):17–22, May 2016.
[38] A. Kumar, J. F. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD Conference*, pages 1969–1984. ACM, 2015.
[39] A. Kumar, F. Niu, and C. Ré. Hazy: Making it Easier to Build and Maintain Big-data Analytics. *ACM Queue*, 11(1):30, 2013.
[40] A. Kumar and C. Ré. Probabilistic Management of OCR Data using an RDBMS. *Proc. VLDB Endow.*, 5(4):322–333, 2011.
[41] F. Li, L. Chen, Y. Zeng, A. Kumar, X. Wu, J. F. Naughton, and J. M. Patel. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. In *SIGMOD Conference*, pages 1517–1534. ACM, 2019.
[42] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, Jan. 2017.
[43] M. Li et al. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 583–598. USENIX Association, 2014.
[44] S. Li, L. Chen, and A. Kumar. Enabling and Optimizing Non-Linear Feature Interactions in Factorized Linear Algebra. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1571–1588. Association for Computing Machinery, 2019.
[45] S. Li et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, Aug. 2020.
[46] X. Meng et al. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.
[47] P. Moritz et al. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577. USENIX Association, 2018.
[48] S. Nakandala and A. Kumar. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1685–1700. Association for Computing Machinery, 2020.
[49] S. Nakandala, A. Kumar, and Y. Papakonstantinou. Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD âĂŹ19, pages 1589–1606. Association for Computing Machinery, 2019.
[50] S. Nakandala, A. Kumar, and Y. Papakonstantinou. Query Optimization for Faster Deep CNN Explanations. *SIGMOD Rec.*, 49(1):61–68, 2020.
[51] S. Nakandala, K. Nagrecha, A. Kumar, and Y. Papakonstantinou. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM Trans. Database Syst.*, 45(4), Dec. 2020.
[52] S. Nakandala, Y. Zhang, and A. Kumar. Cerebro: A Data System for Optimized Deep Learning Model Selection. volume 13, pages 2159–2173. VLDB Endowment, July 2020.
[53] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating Deep Learning Workloads Through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning*, December 2018.
[54] A. Ordookhanians, X. Li, S. Nakandala, and A. Kumar. Demonstration of Krypton: Optimized CNN Inference for Occlusion-based Deep CNN Explanations. *Proc. VLDB Endow.*, 12(12):1894–1897, 2019.
[55] T. K. Sellis. Multiple-query Optimization. *ACM TODS*, 13(1), Mar. 1988.
[56] A. Sergeev and M. D. Balso. Horovod: Fast and Easy Distributed Deep Learning in TF. *CoRR*, abs/1802.05799, 2018.
[57] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: from Theory to Algorithms*. Cambridge University Press, 2014.
[58] A. Thomas and A. Kumar. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *Proc. VLDB Endow.*, 11(13):2168–2182, 2018.
[59] K. R. Weiss, T. M. Khoshgoftaar, and D. Wang. A Survey of Transfer Learning. *J. Big Data*, 3:9, 2016.
[60] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A Comprehensive Survey on Graph Neural Networks. *CoRR*, abs/1901.00596, 2019.
[61] W. Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 595–610. USENIX Association, 2018.
[62] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 265–276. Association for Computing Machinery, 2014.
[63] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. *ACM Trans. Database Syst.*, 41(1):2:1–2:32, 2016.
[64] 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, pages 390–404. Association for Computing Machinery, 2017.
[65] Y. Zhang and A. Kumar. Panorama: A Data System for Unbounded Vocabulary Querying over Video. *Proc. VLDB Endow.*, 13(4):477–491, 2019.
[66] Y. Zhang, A. Kumar, F. McQuillan, N. Jayaram, N. Kak, E. Khanna, O. Kislal, and D. Valdano. Distributed Deep Learning on Data Systems: A Comparative Analysis of Approaches. https://adalabucsd.github.io/papers/TR_2021_Cerebro-DS.pdf, 2020. [Tech report].
[67] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS*, pages 2595–2603. Curran Associates, Inc., 2010.