BigDL: A Distributed Deep Learning Framework for Big Data

Jason (Jinquan) Dai
Intel Corporation

Yiheng Wang*
Tencent Inc.

Xin Qiu
Intel Corporation

Ding Ding
Intel Corporation

Yao Zhang*
Sequoia Capital

Yanzhang Wang
Intel Corporation

Xianyan Jia*
Alibaba Group

Cherry (Li) Zhang
Intel Corporation

Yan Wan*
Alibaba Group

Zhichao Li
Intel Corporation

Jiao Wang
Intel Corporation

Shengsheng Huang
Intel Corporation

Zhongyuan Wu
Intel Corporation

Yang Wang
Intel Corporation

Yuhao Yang
Intel Corporation

Bowen She
Intel Corporation

Dongjie Shi
Intel Corporation

Qi Lu
Intel Corporation

Kai Huang
Intel Corporation

Guoqiong Song
Intel Corporation

ABSTRACT

This paper presents BigDL (a distributed deep learning framework for Apache Spark), which has been used by a variety of users in the industry for building deep learning applications on production big data platforms. It allows deep learning applications to run on the Apache Hadoop/Spark cluster so as to directly process the production data, and as a part of the end-to-end data analysis pipeline for deployment and management. Unlike existing deep learning frameworks, BigDL implements distributed, data parallel training directly on top of the functional compute model (with copy-on-write and coarse-grained operations) of Spark. We also share real-world experience and "war stories" of users that have adopted BigDL to address their challenges(i.e., how to easily build end-to-end data analysis and deep learning pipelines for their production data).

CCS CONCEPTS

• **Theory of computation** → **Distributed algorithms**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

distributed deep learning, big data, Apache Spark, end-to-end data pipeline

*Work was done when the author worked at Intel

1 INTRODUCTION

Continued advancements in artificial intelligence applications have brought deep learning to the forefront of a new generation of data analytics development; as the requirements and usage models expand, new systems and architecture beyond existing deep learning frameworks (e.g., Caffe [1], Torch [2], TensorFlow [3], MXNet [4], Chainer [5], PyTorch [6], etc.) have inevitably emerged. In particular, there is increasing demand from organizations to apply deep learning technologies to their big data analysis pipelines.

To support these new requirements, we have developed BigDL, a distributed deep learning framework for big data platforms and workflows. It is implemented as a library on top of Apache Spark [7], and allows users to write their deep learning applications as standard Spark programs, running directly on existing big data (Apache Hadoop [8] or Spark) clusters. It supports an API similar to Torch and Keras [9] for constructing neural network models (as illustrate in Figure 1); it also supports both large-scale distributed training and inference, leveraging the scale-out architecture of the underlying Spark framework (which runs across hundreds or thousands of servers efficiently).

BigDL provides an expressive, "data-analytics integrated" deep learning programming model; within a single, unified data analysis pipeline, users can efficiently process very large dataset using Spark APIs (e.g., RDD [10], Dataframe [11], Spark SQL, ML pipeline, etc.), feed the distributed dataset to the neural network model, and perform distributed training or inference on top of Spark. Contrary to the conventional wisdom of the machine learning community (that fine-grained data access and in-place updates are critical for efficient distributed training [3]), BigDL provides large-scale, data parallel training directly on top of the functional compute model (with copy-on-write and coarse-grained operations)

```
1    #distributed data processing
2    spark = SparkContext(appName="text_classifier", ...)
3    input_rdd = spark.textFile("hdfs://...")
4    train_rdd = input_rdd.map(lambda x: read_text_and_label(x))
5                         .map(lambda data: decode_to_ndarrays(data))
6                         .map(lambda arrays: to_sample(arrays))
7
8    #distributed training
9    model = Sequential().add(Recurrent().add(LSTM(...)))
10                        .add(Linear(...)).add(LogSoftMax())
11   optimizer = Optimizer(model=model, training_rdd=train_rdd,
12                         criterion=ClassNLLCriterion(),
13                         optim_method=Adagrad(), ...)
14   trained_model = optimizer.optimize()
15
16   #distributed inference
17   test_rdd = ...
18   prediction_rdd = trained_model.predict(test_rdd)
```

Figure 1: The end-to-end text classification pipeline (including data loading, processing, training, prediction, etc.) on Spark and BigDL

of Spark. By unifying the execution model of neural network models and big data analytics, BigDL allows new deep learning algorithms to be seamless integrated into production data pipelines, which can then be easily deployed, monitored and managed in a single unified big data platform.

BigDL is developed as an open source project[1]; over the past years, a variety of users in the industry (e.g., Mastercard, World Bank, Cray, Talroo, UCSF, JD, UnionPay, Telefonica, GigaSpaces, etc.) have built their data analytics and deep learning applications on top of BigDL for a wide range of workloads, such as transfer learning based image classification, object detection and feature extraction, sequence-to-sequence prediction for precipitation nowcasting, neural collaborative filtering for recommendations, etc. In this paper, we focus on the execution model of BigDL to support large-scale distributed training (a challenging system problem for deep learning frameworks), as well as empirical results of real-world deep learning applications built on top of BigDL. The main contributions of this paper are:

- It presents BigDL, a working system that have been used by many users in the industry for distributed deep learning on production big data systems.
- It describes the distributed execution model in BigDL (that adopts the state of practice of big data systems), which provides a viable design alternative for distributed model training (compared to existing deep learning frameworks).
- It shares real-world experience and "war stories" of users that have adopted BigDL to address their challenges (i.e.,

how to easily build end-to-end data analysis and deep learning pipelines for their production data).

2   MOTIVATION

A lot of efforts in the deep learning community have been focusing on improving the accuracy and/or speed of standard deep learning benchmarks (such as ImageNet [12] or SQuAD [13]). For these benchmarks, the input dataset have already been curated and explicitly labelled, and it makes sense to run deep learning algorithms on specialized deep learning frameworks for best computing efficiency. On the other hand, if the input dataset are dynamic and messy (e.g., live data streaming into production data pipeline that require complex processing), it makes more sense to adopt BigDL to build the end-to-end, integrated data analytics and deep learning pipelines for production data.

As mentioned in Section 1, BigDL has been used by a variety of users in the industry to build deep learning applications on their production data platform. The key motivation for adopting such a unified data analytics and deep learning system like BigDL is to improve the ease of use (including development, deployment and operations) for applying deep learning in real-world data pipelines.

In real world, it is critical to run deep learning applications directly on where the data are stored, and as a part of the end-to-end data analysis pipelines. Applying deep learning to production big data is very different from the ImageNet [12] or SQuAD [13] problem; real-world big data are both dynamic and messy, and are possibly implicitly labeled (e.g., implicit feedbacks in recommendation applications [14]), which require very complex data processing; furthermore, instead of running ETL (extract, transform and load)

---

[1]https://github.com/intel-analytics/BigDL

and data processing only once, real-world data analytics pipeline is an iterative and recurrent process (e.g., back-and-forth development and debugging, incremental model update with new production data, etc.). Therefore, it is highly inefficient to run these workloads on separate big data and deep learning systems (e.g., processing data on a Spark cluster, and then export the processed data to a separate TensorFlow cluster for training/inference) in terms of not only data transfer, but also development, debugging, deployment and operation productivity.

One way to address the above challenge is to adopt a "connector approach" (e.g., TFX [15], CaffeOnSpark [16], TensorFlowOnSpark [17], SageMaker [18], etc.), which develops proper interfaces to connect different data processing and deep learning components using an integrated workflow (and possibly on a shared cluster). However, the adaptation between different frameworks can impose very large overheads in practice (e.g., inter-process communication, data serialization and persistency, etc.). More importantly, this approach suffers from impedance mismatches [19] that arise from crossing boundaries between heterogeneous components. For instance, many of these systems (such as TensorFlowOnSpark) first use big data (e.g., Spark) tasks to allocate resources (e.g., Spark worker nodes), and then run deep learning (e.g., TensorFlow) tasks on the allocated resources. However, big data and deep learning systems have very different distributed execution model - big data tasks are embarrassingly parallel and independent of each other, while deep learning tasks need to coordinate with and depend on others. For instance, when a Spark worker fails, the Spark system just relaunch the worker (which in turn re-runs the TensorFlow task); this however is incompatible with the TensorFlow execution model and can cause the entire workflow to block indefinitely.

The Big Data community have also started to provide better support for the "connector approach". For instance, the barrier execution mode introduced by Project Hydrogen [20] provides gang scheduling [21] support in Spark, so as to overcome the errors caused by different execution models between Spark and existing deep learning frameworks (as described in the preceding paragraph). On the other hand, this does not eliminate the difference in the two execution models, which can still lead to lower efficiency (e.g., it is unclear how to apply delay scheduling [22] to gang scheduling in Spark, resulting in poorer data locality). In addition, it does not address other impedance mismatches such as different parallelism behaviors between data processing and model computations (e.g., see Section 5.1).

BigDL has taken a different approach that directly implements the distributed deep learning support in the big data system (namely, Apache Spark). Consequently, one can easily build the end-to-end, "data-analytics integrated" deep learning pipelines (under a unified programming paradigm, as illustrated in Figure 1), which can then run as standard Spark jobs to apply large-scale data processing and deep learning training/inference to production dataset within a single framework. This completely eliminates the impedance mismatch problems, and greatly improves the efficiency of development and operations of deep learning applications for big data.

3 BIGDL EXECUTION MODEL

This section describes in detail how BigDL support large-scale, distributed training on top of Apache Spark. While it has adopted the standard practice (such as data parallel training [23], parameter server and AllReduce [3] [24] [25] [26]) [27]for scalable training, the key novelty of BigDL is how to efficiently implement these functionalities on a functional, coarse-grained compute model of Spark.

The conventional wisdom of the machine learning community is that, fine-grained data access and in-place data mutation are critical to support highly efficient parameter server, AllReduce and distributed training [3]. On the other hand, big data systems (such as Spark) usually adopts a very different, functional compute model, where dataset are immutable and can only be transformed into new dataset without side effects (i.e., copy-on-write); in addition, the transformations are coarse-grained operations (i.e., applying the same operation to all data items at once).
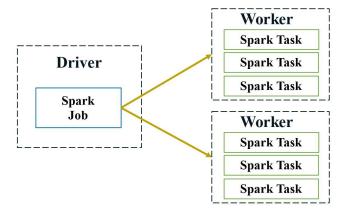

Figure 2: A Spark job consists of many Spark tasks; the driver node is responsible for scheduling and dispatching the tasks to worker nodes, which runs the actual Spark tasks.

Algorithm 1 Data-parallel training in BigDL

1: **for** i = 1 to M **do**
2:     //"model forward-backward" job
3:     **for** each task in the Spark job **do**
4:         read the latest **weights**;
5:         get a random **batch** of data from local *Sample* partition;
6:         compute local **gradients** (forward-backward on local *model* replica);
7:     **end for**
8:     //"parameter synchronization" job
9:     aggregate (sum) all the **gradients**;
10:    update the **weights** per specified optimization method;
11: **end for**

BigDL is implemented as a standard library on Spark and has adopted this functional compute model; nevertheless, it still provides an efficient "parameter server" style architecture for efficient distributed training (by implementing an AllReduce like operation directly using existing primitives in Spark).
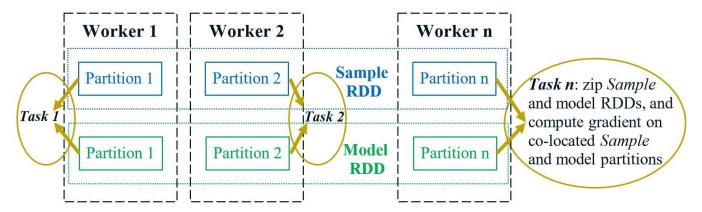
Figure 3: The "model forward-backward" spark job, which computes the local gradients for each model replica in parallel.

3.1 Spark execution model

Similar to other Big Data systems (such as MapReduce [28] and Dryad [29]), a Spark cluster consists of a single driver node and multiple worker nodes, as shown in Figure 2. The driver is responsible for coordinating tasks in a Spark job (e.g., task scheduling and dispatching), while the workers are responsible for the actual computation. To automatically parallelize the data processing across the cluster in a fault-tolerant fashion, Spark provides a data-parallel, functional compute model. In a Spark job, data are represented as Resilient Distributed Dataset (RDD) [10], which is an immutable collection of records partitioned across a cluster, and can only be transformed to derive new RDDs (i.e., copy-on-write) through functional operators like *map*, *filter* and *reduce* (e.g., see line 4 - 6 in Figure 1); in addition, these operations are both data-parallel (i.e., applied to individual data partitions in parallel by different Spark tasks) and coarse-grained (i.e., applying the same operation to all data items at once).

3.2 Data-parallel training in BigDL

Built on top of the data-parallel, functional compute model of Spark, BigDL provides synchronous data-parallel training to train a deep neural network model across the cluster, which is shown to achieve better scalability and efficiency (in terms of time-to-quality) compared to asynchronous training [30]. Specifically, the distributed training in BigDL is implemented as an iterative process, as illustrated in Algorithm 1; each iteration runs a couple of Spark jobs to first compute the gradients using the current mini-batch (by a "model forward-backward" job), and then make a single update to the parameters of the neural network model (by a "parameter synchronization" job).

The training data in BigDL are represented as an RDD of Samples (see line 6 in Figure 1), which are automatically partitioned across the Spark cluster. In addition, to implement the data-parallel training, BigDL also constructs an RDD of models, each of which is a replica of the original neural network model. Before the training, both the model and Sample RDDs are cached in memory, and co-partitioned and co-located across the cluster, as shown in Figure 3; consequently, in each iteration of the model training, a single

"model forward-backward" Spark job can simply apply the functional *zip* operator to the co-located partitions of the two RDDs (with no extra cost), and compute the local gradients in parallel for each model replica (using a small batch of data in the co-located Sample partition), as illustrated in Figure 3.

BigDL does not support model parallelism (i.e., no distribution of the model across different workers). This is not a limitation in practice, as BigDL runs on Intel Xeon CPU servers, which usually have large (100s of GB) memory size and can easily hold very large models.

3.3 Parameter synchronization in BigDL

Parameter synchronization is a performance critical operation for data parallel distributed model training (in terms of speed and scalability). To support efficient parameter synchronization, existing deep learning frameworks usually implement parameter server or AllReduce using operations like fine-grained data access and in-place data mutation. Unfortunately, these operations are not supported by the functional compute model of big data systems (such as Spark).

Algorithm 2 "Parameter synchronization" job

1: **for** each task n in the "parameter synchronization" job **do**
2: **shuffle** the nth partition of all gradients to this task;
3: aggregate (sum) these gradients;
4: updates the nth partition of the weights;
5: **broadcast** the nth partition of the updated weights;
6: **end for**

BigDL has taken a completely different approach that directly implements an efficient AllReduce like operation using existing primitives in Spark (e.g., shuffle, broadcast, in-memory cache, etc.), so as to mimic the functionality of a parameter server architecture (as illustrated in Figure 4).

- A Spark job has N tasks, each of which is assigned a unique Id ranging from 1 to N in BigDL. After each task in the "model forward-backward" job computes the local gradients (as described in Section 3.2 and illustrated in Figure 3), it
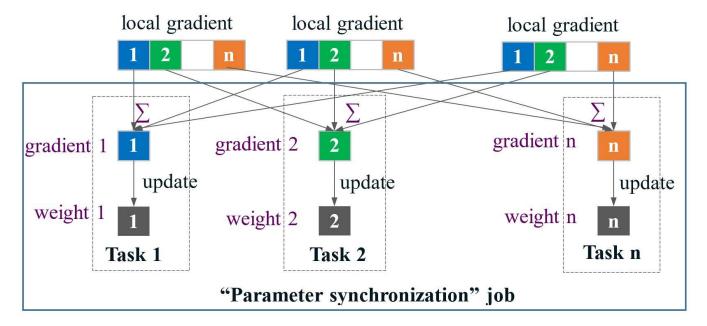
Figure 4: Parameter synchronization in BigDL. Each local gradient (computed by a task in the "model forward-backward" job) is evenly divided into N partitions; then each task n in the "parameter synchronization" job aggregates these local gradients and updates the weights for the nth partition.

evenly divides the local gradients into N partitions, as shown in Figure 4.

- Next, another "parameter synchronization" job is launched; each task n of this job is responsible for managing the nth partition of the parameters (as shown in Algorithm 2), just like a parameter server does. Specifically, the nth partition of the local gradients (computed by the previous "model forward-backward" job) are first shuffled to task n, which aggregates these gradients and applies the updates to the nth partition of the weights, as illustrated in Figure 4.

- After that, each task n in the "parameter synchronization" job broadcasts the nth partition of the updated weights; consequently, tasks in the "model forward-backward" job of the next iteration can read the latest value of all the weights before the next training step begins.

- The shuffle and task-side broadcast operations described above are implemented on top of the distributed in-memory storage in Spark: the relevant tasks simply store the local gradients and updated weights in the in-memory storage, which can then be read remotely by the Spark tasks with extremely low latency.

The implementation of AllReduce in BigDL has similar performance characteristics compared to *Ring AllReduce* from Baidu Research [31]. As described in [31], the total amount of data transferred to and from every node is *2K(N-1)/N* in Ring AllReduce (where N is the number of nodes and K is the total size of the parameters); similarly, in BigDL, the total amount of data transferred to and from every node is *2K*. In addition, all the bandwidth of every node in the cluster are fully utilized in both BigDL and Ring AllReduce. As a result, BigDL can efficiently train large deep

neural network across large (e.g., hundreds of servers) clusters, as shown in Section 4.3.

3.4 Discussions

While BigDL has followed the standard practice (such as data parallel training and AllReduce operations) for scalable training, its implementation is very different from existing deep learning frameworks. By adopting the state of practice of big data systems (i.e., coarse-grained functional compute model), BigDL provides a viable design alternative for distributed model training. This allows deep learning algorithms and big data analytics to be seamless integrated into a single unified data pipeline, and completely eliminates the impedance mismatch problem described in Section 2. Furthermore, this also makes it easy to handle failures, resource changes, task preemptions, etc., which are expected to be norm rather than exception in large-scale systems.

Existing distributed deep learning frameworks (e.g., TensorFlow, MXNet, Petuum [26], ChainerMN [32], etc.) have adopted an architecture where multiple long-running, stateful tasks interact with others for model computation and parameter synchronization, usually in a blocking fashion to support synchronous distributed training. While this is optimized for constant communications among the tasks, it can only support coarse-grained failure recovery by completely starting over from previous (e.g., a couple of epochs before) snapshots.

In contrast, BigDL runs a series of short-lived Spark jobs (e.g., two jobs per mini-batch as described earlier), and each task in the job is stateless, non-blocking, and completely independent of each

other; as a result, BigDL tasks can simply run without gang scheduling. In addition, it can also efficiently support fine-grained failure recovery by just re-running the failed task (which can then regenerate the associated partition of the local gradient or updated weight in the in-memory storage of Spark); this allows the framework to automatically and efficiently address failures (e.g., cluster scale-down, task preemption, random bugs in the code, etc.) in a timely fashion.

While AllReduce has been implemented in almost all existing deep learning frameworks, the implementation in BigDL is very different. In particular, existing deep learning frameworks usually implement the AllReduce operation using MPI-like primitives; as a result, they often create long-running task replicas that coordinate among themselves with no central control. On the other hand, BigDL has adopted a logically centralized control for distributed training [33]; that is, a single driver program coordinates the distributed training (as illustrated in Algorithm 1). The driver program first launches the "model forward-backward" job to compute the local gradients, and then launches the "parameter synchronization" job to update the weights. The dependence between the two jobs are explicitly managed by the driver program, and each individual task in the two jobs are completely stateless and non-blocking once they are launched by the driver.

4   EVALUATION

This section evaluates the computing performance and scalability of neural network training in BigDL. In addition, while we do not report inference performance results in this section, Section 5.1 shows the comparison of a real-world object detection inference pipeline running on BigDL vs. Caffe (and as reported by JD.com, the BigDL inference pipeline running on 24 Intel Xeon servers is 3.83x faster than Caffe running on 5 servers and 20 GPU cards).

4.1   Experiments

Two categories of neural network models are used in this section to evaluate the performance and scalability of BigDL, namely, neural collaborative filtering (NCF) and convolutional neural network (CNN), which are representatives of the workloads that BigDL users run in their production Big Data platform.

Neural Collaborative Filtering (NCF) [34] is one of most commonly used neural network models for recommendation, and has also been included in MLPerf [35], a widely used benchmark suite for measuring training and inference performance of machine learning hardware, software, and services. In our experiments, we compare the training performance of BigDL (running on Intel Xeon server) vs. PyTorch (running on GPU).

In addition, deep convolutional neural networks (CNNs) have achieved human-level accuracy and are widely used for many computer vision tasks (such as image classifications and object detection). In our experiments, we study the scalability and efficiency of training Inception-v1 [36] on ImageNet dataset [37] in BigDL with various number of Intel Xeon servers and Spark task; the results for other deep convolutional models, such as Inception-v3 [38] and

ResNet50 [39], are similar. We do not include results for RNN (recurrent neural networks) training in this section, because it actually has better scalability compared to CNN training. This is because RNN computation is much slower than CNN, and therefore the parameter synchronization overhead (as a fraction of model compute time) is also much lower.

4.2   Computing Performance

To study the computing performance of BigDL, we compare the training speed of the NCF model using BigDL and PyTorch. MLPerf has provided a reference implementation of the NCF program [40] based on PyTorch 0.4, which trains a movie recommender using the MovieLens 20Million dataset (ml-20m) [41], a widely used benchmark dataset with 20 million ratings and 465,000 tags applied to 27,000 movies by 138,000 users. It also provides the reference training speed of the PyTorch implementation (to achieve the target accuracy goal) on a single Nvidia P100 GPU.

We have implemented the same NCF program using BigDL 0.7.0 and Spark 2.1.0 [42]. We then trained the program on a dual-socket Intel Skylake 8180 2.5GHz server (with 56 cores in total and 384GB memory), and it took 29.8 minutes to converge and achieve the same accuracy goal.
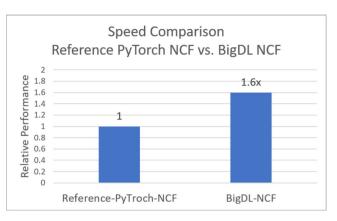

Figure 5: The training performance of NCF using the BigDL implementation is 1.6x faster than the reference PyTorch implementation, as reported by MLPerf [43].

As reported by MLPerf, the training performance of NCF using the BigDL implementation is 1.6x faster than the reference PyTorch implementation [43] (as shown in Figure 5). While this only compares the training performance of BigDL on a single CPU server to PyTorch on a single GPU, it shows BigDL provides efficient implementations for neural network model computation (forward and backward). We will study the scalability and efficiency of the distributed training in BigDL in Section 4.3 and 4.4.

4.3   Scalability of distributed training

In the machine learning community, it is commonly believed that fine-grained data access and in-place data mutation are critical for efficient distributed training, and mechanisms like Spark's RDDs would impose significant overheads [3]. In this section, we show that BigDL provides highly efficient and scalable training, despite

it is built on top of the coarse-grained functional compute model and immutable RDDs of Spark.

The scalability of distributed training in BigDL is determined by the efficiency (or overheads) of its parameter synchronizations. We first study the parameter synchronization overheads in BigDL by running ImageNet Inception-v1 model training using BigDL on various number of Xeon servers (dual-socket Intel Broadwell 2.20GHz, 256GB RAM and 10GbE network) [44]. As shown in Figure 6, the parameter synchronization overheads, measured as a fraction of the average model computation (forward and backward) time, turn out to be small (e.g., less than 7% for Inception-v1 training on 32 nodes) in BigDL.

To study the scalability of the distributed training of BigDL on very large-scale Intel Xeon clusters, Cray have run ImageNet Inception-v1 model training using BigDL 0.3.0 with various node counts (starting at 16 nodes and scaling up to 256 nodes) [45]. Each node is a dual-socket Intel Broadwell 2.1 GHz (CCU 36 and DDR4 2400) server; the learning rate and Spark's executor memory are set to 0.10 and 120 GB respectively in the experiments.
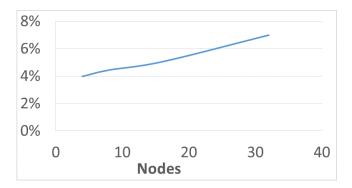

Figure 6: Overheads of parameter synchronization (as a fraction of average model computation time) of ImageNet Inception-v1 training in BigDL [44].
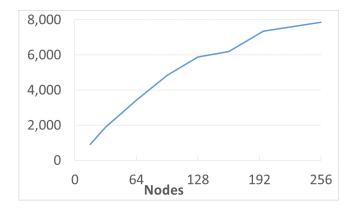

Figure 7: Throughput of ImageNet Inception-v1 training in BigDL 0.3.0 reported by Cray, which scales almost linearly up to 96 nodes (and continue to scale reasonably up to 256 nodes) [45].
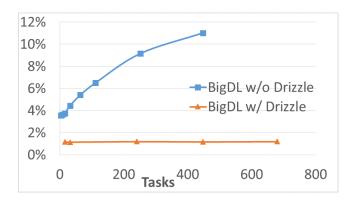

Figure 8: Overheads of task scheduling and dispatch (as a fraction of average computation time) for ImageNet Inception-v1 training in BigDL [46].

Figure 7 shows the throughput of ImageNet Inception-v1 training; the training throughput scales almost linearly up to 96 nodes (e.g., about 5.3x speedup on 96 nodes compared to 16 nodes), and continue to scale reasonably well up to 256 nodes [45]. The results show that, even though BigDL implements its parameter server architecture directly on top of Spark (with immutable RDDs and coarse-grained functional operations), it can still provide efficient distributed training on large clusters.

4.4 Efficiency of task scheduling

As described in Section 3.4, BigDL needs to run a very large number of shot-lived tasks on Spark (e.g., the ImageNet Inception-v1 training may run 100s of thousands of iterations and 100s of tasks in parallel per iteration, while each task runs for just a couple of seconds); as a result, the underlying Spark framework needs to schedule a very large number of tasks across the cluster in a short period of time, which can potentially become a bottleneck on large clusters. For instance, Figure 8 shows that the overhead of launching tasks (as a fraction of average model computation time) in ImageNet Inception-v1 training on BigDL, while low for 100-200 tasks per iteration, can grows to over 10% when there are close to 500 tasks per iteration [46].

To address this issue, in each training iteration BigDL will launch only a single (multi-threaded) task on each server, so as to achieve high scalability on large clusters (e.g., up to 256 machines, as described in Section 4.3). To scale to an even larger number (e.g., over 500) of servers, one can potentially leverage the iterative nature of model training (in which the same operations are executed repeatedly). For instance, group scheduling introduced by Drizzle [47], a low latency execution engine for Spark, can help schedule multiple iterations (or a group) of computations at once, so as to greatly reduce scheduling overheads even if there are a large number of tasks in each iteration, as shown in Figure 8 (which ran on AWS EC2 using r4.x2large instances) [46].

5 APPLICATIONS

Since its initial open source release (on Dec 30, 2016), BigDL users have built many deep learning applications on Spark and big data
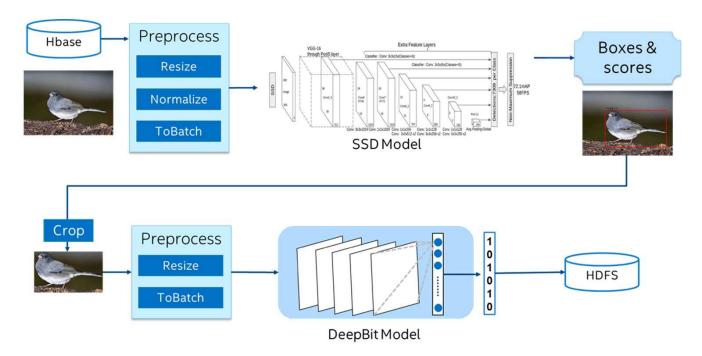
Figure 9: End-to-end object detection and image feature extraction pipeline (using SSD and DeepBit models) on top of Spark and BigDL [48].

platforms. In this section, we share the real-world experience and "war stories" of our users that adopts BigDL to build the end-to-end data analysis and deep learning pipelines for their production data.
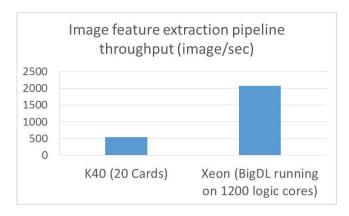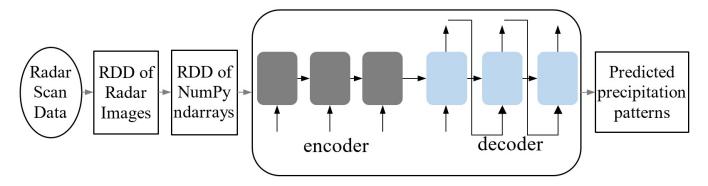

Figure 10: Throughput of GPU clusters and Xeon clusters for the image feature extraction pipeline benchmarked by JD; the GPU cluster consists of 20 NVIDIA Tesla K40 cards, and the Xeon cluster consists of 1200 logical cores (with each Intel Xeon E5-2650 v4 2.2GHz server running 50 logical cores) [48].

5.1 Image feature extraction using object detection models

JD.com has built an end-to-end object detection and image feature extraction pipeline on top of Spark and BigDL [48], as illustrated in Figure 9.

- The pipeline first reads hundreds of millions of pictures from a distributed database into Spark (as an RDD of pictures), and then pre-processes the RDD of pictures in a distributed fashion using Spark.
- It then uses BigDL to load a SSD [49] model (pre-trained in Caffe) for large scale, distributed object detection on Spark, which generates the coordinates and scores for the detected objects in each of the pictures.
- It then generates the RDD of target images (by keeping the object with highest score as the target, and cropping the original picture based on the coordinates of the target), and further pre-processes the RDD of target images.
- Finally it uses BigDL to load a DeepBit [50] model (again pre-trained in Caffe) for distributed feature extraction of the target images, and stores the results (RDD of extracted object features) in HDFS.

Previously JD engineers have deployed the same solution on a 5-node GPU cluster with 20 NVIDIA Tesla K40 following a "connector approach" (similar to CaffeOnSpark): reading data from HBase, partitioning and processing the data across the cluster, and then running the deep learning models on Caffe. This turns out to be very complex and error-prone (because all of the data partitioning, load balancing, fault tolerance, etc., need to be manually managed). In addition, it also reveals an impedance mismatch of the

Sequence to sequence model

Figure 11: End-to-end precipitation nowcasting workflow (using sequence-to-sequence models) on Spark and BigDL [45].
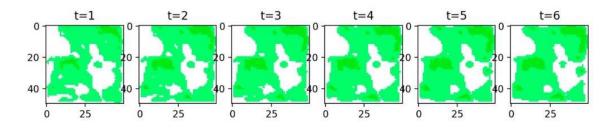

Figure 12: Predicting precipitation patterns for the next hour (i.e., a sequence of images for the future time steps of the next hour) on Spark and BigDL [45].

"connector approach" (HBase + Caffe in this case) - reading data from HBase takes about half of the time in this solution (because the task parallelism is tied to the number of GPU cards in the system, which is too low for interacting with HBase to read the data).

After migrating the solution to BigDL, JD engineers can easily implement the entire data analysis and deep learning pipeline (including data loading, partitioning, pre-processing, model inference, etc.,) under a unified programming paradigm on Spark. This not only greatly improves the efficiency of development and deployment, but also delivers about 3.83x speedup (running on about 24 Intel Broadwell 2.2GHz servers) compared to running the Caffe-based solution on the GPU cluster (with 20 NVIDIA Tesla K40 cards), as reported by JD [48] and shown in Figure 10.

5.2 Precipitation nowcasting using Seq2Seq models

Cray has built a precipitation nowcasting (predicting short-term precipitation) application using a Seq2Seq [51] model (with a stacked convolutional LSTM network [52] as the encoder, and another stacked convolutional LSTM network as the decoder); the end-to-end pipeline runs on Spark and BigDL [45], including data preparation, model training and inference (as illustrated in Figure 11).

- The application first reads over a terabyte of raw radar scan data into Spark (as an RDD of radar images), and then converts it into an RDD of NumPy ndarrays.
- It then trains a sequence-to-sequence model, using a sequence of images leading up to the current time as the input, and a sequence of predicted images in the future as the output.
- After the model is trained, it can be used to predict, say, the precipitation patterns (i.e., a sequence of images for the future time steps) of the next hour, as illustrated in Figure 12.

Cray engineers have previously implemented the application using two separate workflows: running data processing on a highly distributed Spark cluster, and deep learning training on another GPU cluster running TensorFlow. It turns out that this approach not only brings high data movement overheads, but also greatly hurts the development productivity due to the fragmented workflow. As a result, Cray engineers chose to implement the solution using a single unified data analysis and deep learning pipeline on Spark and BigDL, which greatly improves the efficiency of development and deployment.

5.3 Real-time streaming speech classification

GigaSpaces has built a speech classification application for efficient call center management [53], which automatically routes client calls to corresponding support specialists in real-time. The end-to-end workflow is implemented using BigDL with Apache Kafka
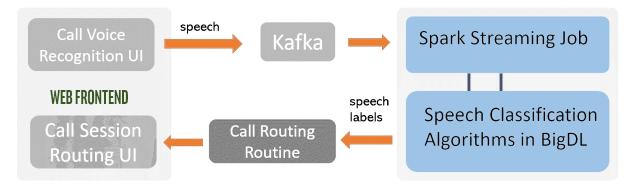
Figure 13: The end-to-end workflow of real-time streaming speech classification on Kafka, Spark Streaming and BigDL [53].

[54] and Spark Streaming [55] (as illustrated Figure 13), so as to provide distributed realtime streaming model inference.

- When a customer calls the call center, his or her speech is first processed on the fly by a speech recognition unit and result is stored in Kafka.
- A Spark Streaming job then reads speech recognition results from Kafka and classifies each call using the BigDL model in real-time.
- The classification result is in turn used by a routing system to redirect the call to the proper support specialist.

One of the key challenges for GigaSpaces engineers to implement the end-to-end workflow is how to efficiently integrate the new neural network models in the realtime stream processing pipeline, and how to seamlessly scale the streaming applications from a handful machines to thousands of nodes. BigDL allows neural network models to be directly applied in standard distributed streaming architecture for Big Data (using Apache Kafka and Spark Streaming), which can then efficiently scales out to a large number of nodes in a transparent fashion. As a result, this greatly improves the developer productivity and deployment efficiency of the end-to-end streaming workflow.

6 RELATED WORK

Existing deep learning frameworks (such as TensorFlow, MXNet, Petuum, ChainerMN, etc.) typically provide efficient parameter server and/or AllReduce implementation (using fine-grained data access and in-place data mutation) for distributed training. In contrast, BigDL provides distributed training support directly on top of a functional compute model of big data systems (with copy-on-write and coarse-grained operations), which is completely different from the implementation in existing deep learning frameworks. This provides a viable design alternative for distributed model training by adopting the state of practice of big data systems, and makes it easy to handle failures, resource changes, task preemptions, etc., in a more timely and fine-grained fashion.

As discussed in Section 2, to address the challenge of integrating deep learning into real-world data pipelines, there have been many efforts in the industry that adopt a "connector approach" (e.g., TFX, CaffeOnSpark, TensorFlowOnSpark, SageMaker, etc.). Unfortunately, these frameworks can incur very large overheads

in practice due to the adaptation layer between different frameworks; more importantly, they often suffer from impedance mismatches that arise from crossing boundaries between heterogeneous components. While efforts in the Big Data community (such as Project Hydrogen in Spark) attempt to overcome some of these issues brought by the "connector approach", they still do not address the fundamental "impedance mismatch" problem (as discussed in Section 2). By unifying the distributed execution model of deep neural network models and big data analysis, BigDL provides a single unified data pipeline for both deep learning and big data analysis, which eliminates the adaptation overheads or impedance mismatch.

7 SUMMARY

We have described BigDL, including its distributed execution model, computation performance, training scalability, and real-world use cases. It allows users to build deep learning applications for big data using a single unified data pipeline; the entire pipeline can directly run on top of existing big data systems in a distributed fashion. Unlike existing deep learning frameworks, it provides efficient and scalable distributed training directly on top of the functional compute model (with copy-on-write and coarse-grained operations) of Spark. BigDL is a work in progress, but our initial experience is encouraging. Since its initial open source release on Dec 30, 2016, it has received over 3100 stars on Github; and it has enabled many users (e.g., Mastercard, World Bank, Cray, Talroo, UCSF, JD, UnionPay, Telefonica, GigaSpaces, etc.) to build new analytics and deep learning applications for their production data pipelines.

REFERENCES

[1] Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevor. Caffe: Convolutional architecture for fast feature embedding. in *Proceedings of the 22nd ACM international conference on Multimedia.* MM'14.

[2] Collobert, Ronan and Kavukcuoglu, Koray and Farabet, Clément. Torch7: A matlab-like environment for machine learning. in *BigLearn, NIPS workshop.* (2011).

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* OSDI'16.

[4] Chen, Tianqi and Li, Mu and Li, Yutian and Lin, Min and Wang, Naiyan and Wang, Minjie and Xiao, Tianjun and Xu, Bing and Zhang, Chiyuan and Zhang, Zheng. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *In Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS).* (2015).

[5] Tokui, Seiya and Oono, Kenta and Hido, Shohei and Clayton, Justin Chainer: a next-generation open source framework for deep learning in *In Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS).* (2015).

[6] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam Automatic differentiation in pytorch. *NIPS 2017 Autodiff Workshop.* (2017).

[7] Apache spark Apache software foundation. (2014) (https://spark.apache.org).

[8] Apache hadoop Apache software foundation. (2006) (https://hadoop.apache.org).

[9] Chollet,F.et al. Keras. (https://keras.io).

[10] Zaharia, Matei and Chowdhury, Mosharaf and Das, Tathagata and Dave, Ankur and Ma, Justin and McCauley, Murphy and Franklin, Michael J and Shenker, Scott and Stoica, Ion. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* NSDI'12.

[11] Armbrust, Michael and Xin, Reynold S and Lian, Cheng and Huai, Yin and Liu, Davies and Bradley, Joseph K and Meng, Xiangrui and Kaftan, Tomer and Franklin, Michael J and Ghodsi, Ali and others. Spark sql: Relational data processing in spark. in *2015 ACM SIGMOD international conference on management of data.* SIGMOD'15.

[12] Russakovsky,Olga and Deng, Jia and Su, Hao and Krause, Jonathan and Satheesh, Sanjeev and Ma, Sean and Huang, Zhiheng and Karpathy, Andrej and Khosla, Aditya and Bernstein, Michael and others. Imagenet large scale visual recognition challenge. *International journal of computer vision(IJCV).* (2015).

[13] Rajpurkar,P and Zhang,J and Lopyrev,K and Liang,P. Squad: 100,000+ questions for machine comprehension of text. *In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP.* (2016).

[14] Jawaheer,G and Szomszor,M and Kostkova,P. Comparison of implicit and explicit feedback from an online music recommendation service. in *proceedings of the 1st international workshop on information heterogeneity and fusion in recommender systems.* (2010) HetRec'10.

[15] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., Koo, C. Y., Lew, L., Mewald, C., Modi, A. N., Polyzotis, N., Ramesh, S., Roy, S., Whang, S. E., Wicke, M., Wilkiewicz, J., Zhang, X., and Zinkevich, M. 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.

[16] CaffeOnSpark. Yahoo. (2016) (https://github.com/yahoo/CaffeOnSpark).

[17] TensorflowOnSpark. Yahoo. (2017) (https://github.com/yahoo/TensorFlowOnSpark).

[18] Sagemaker. Amazon. (2017) (https://aws.amazon.com/sagemaker/).

[19] Lin, Jimmy and Ryaboy, Dmitriy Scaling big data mining infrastructure: the twitter experience. *ACM SIGKDD Explorations Newsletter* 14(2). (December 2012).

[20] Reynold Xin. "project hydrogen: Unifying state-of-the-art ai and big data in apache spark". spark + ai summit 2018.

[21] Gang scheduling. (https://en.wikipedia.org/wiki/Gang_scheduling/).

[22] Zaharia, Matei and Borthakur, Dhruba and Sen Sarma, Joydeep and Elmeleegy, Khaled and Shenker, Scott and Stoica, Ion. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. in *Proceedings of the 5th European conference on Computer systems,.* EuroSys'10.

[23] Dean,J., Corrado,G., Monga,R., Chen,K., Devin,M., Mao,M., Ranzato,Marc'aurelio, Senior,A., Tucker,P., Yang,K., Le,Q.V., Ng,A.Y. Large scale distributed deep networks. in *Proceedings of the 25th International Conference on Neural Information Processing Systems.* NIPS'12.

[24] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.* OSDI'14.

[25] Chilimbi,T., Suzue,Y., Apacible,J., and Kalyanaraman,K. Project adam: Building an efficient and scalable deep learning training system. in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.* OSDI'14.

[26] Xing,E.P., Ho,Q., Dai,W., Kim,J.-K., Wei,J., Lee,S., Zheng,X., Xie,P., Kumar,A., and Yu,Y. Petuum: A new platform for distributed machine learning on big data. *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD'15.

[27] Zhang,H., Zheng,Z., Xu,S., Dai,W., Ho,Q., Liang,X., Hu,Z., Wei,J., Xie,P., and Xing,E.P. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. in *2017 USENIX Annual Technical Conference (USENIX ATC 17).* (2017).

[28] Jeffrey Dean, Sanjay Ghemawat Mapreduce: simplified data processing on large clusters. *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation,{OSDI}.* (2004).

[29] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007.* EuroSys'07.

[30] Chen,J., Monga,R., Bengio,S., and Jozefowicz,R. Revisiting distributed synchronous sgd. *In International Conference on Learning Representations Workshop Track.* (2016).

[31] Gibiansky,Andrew. "bringing hpc techniques to deep learning". (http://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/).

[32] Akiba,T., Fukuda,K., and Suzuki,S. Chainermn: Scalable distributed deep learning framework. *Proceedings of Workshop on ML Systems in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS).* (2017).

[33] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. *International Conference on Machine Learning (ICML).* (2018).

[34] He, Xiangnan and Liao, Lizi and Zhang, Hanwang and Nie, Liqiang and Hu, Xia and Chua, Tat-Seng Neural collaborative filtering. in *Proceedings of the 26th international conference on world wide web. International World Wide Web Conferences Steering Committee.* (2017).

[35] Mlperf. (https://mlperf.org/).

[36] Szegedy,C., Liu,W., Jia,Y., Sermanet,P., Reed,S., Anguelov,D., Erhan,D., Vanhoucke,V., and Rabinovich,A. Going deeper with convolutions in *Computer Vision and Pattern Recognition (CVPR).* (2015).

[37] Deng,J., Socher,R., Fei-Fei,L., Dong,W., Li,K., and Li,L.-J. Imagenet: A large-scale hierarchical image database. in *2009 IEEE conference on computer vision and pattern recognition(CVPR).* (2009).

[38] Szegedy,C., Vanhoucke,V., Ioffe,S., Shlens,J., and Wojna,Z. Rethinking the inception architecture for computer vision in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* (2016).

[39] He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian Deep residual learning for image recognition. in *Proceedings of the IEEE conference on computer vision and pattern recognition.* (2016).

[40] Reference ncf implementation using pytorch in mlperf. (https://github.com/mlperf/training/blob/master/recommendation/pytorch/README.md).

[41] Harper, F Maxwell and Konstan, Joseph A. "the movielens datasets: History and context". *ACM Trans. Interact. Intell. Syst.* 5(4):19. (2015).

[42] Ncf implementation in bigdl. (https://github.com/mlperf/training_results_v0.5/tree/master/v0.5.0/intel/intel_ncf_submission).

[43] Mlperf 0.5 training results. (https://mlperf.org/training-results-0-5).

[44] Jason (Jinquan) Dai, and Ding Ding. Very large-scale distributed deep learning with bigdl. o'reilly ai conference, san francisco. (2017).

[45] Alex Heye, et al. "scalable deep learning with bigdl on the urika-xc software suite". (https://www.cray.com/blog/scalable-deep-learning-bigdl-urika-xc-software-suite/).

[46] Shivaram Venkataraman, et al. "accelerating deep learning training with bigdl and drizzle on apache spark". (https://rise.cs.berkeley.edu/blog/accelerating-deep-learning-training-with-bigdl-and-drizzle-on-apache-spark/).

[47] Venkataraman,S., Panda,A., Ousterhout,K., Armbrust,M., Ghodsi,A., Franklin,M.J., Recht,B., and Stoica,I. Drizzle: Fast and adaptable stream processing at scale in *Proceedings of the 26th Symposium on Operating Systems Principles.* SOSP'17.

[48] Jason (Jinquan) Dai, et al. Building large-scale image feature extraction with bigdl at jd.com. (https://software.intel.com/en-us/articles/building-large-scale-image-feature-extraction-with-bigdl-at-jdcom).

[49] Liu,W., Anguelov,D., Erhan,D., Szegedy,C., Reed,S.E., Fu,C.-Y., and Berg,A.C. Ssd: Single shot multibox detector in *ECCV.* (2016).

[50] Lin, K., Lu, J., Chen, C.-S., and Zhou, J. Learning compact binary descriptors with unsupervised deep neural networks. in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* (2016).

[51] Sutskever,I., Vinyals,O., and Le,Q.V. Sequence to sequence learning with neural networks. in *Proceedings of the 27th International Conference on Neural Information Processing Systems.* Vol. 2. NIPS'14.

[52] Shi,X., Chen,Z., Wang,H., Yeung,D.-Y., Wong,W.-k., and Woo,W.-c. Convolutional lstm network: A machine learning approach for precipitation nowcasting. in *Proceedings of the 28th International Conference on Neural Information Processing Systems.* Vol. 1. NIPS'15.

[53] Rajiv Shah. Gigaspaces integrates insightedge platform with intel's bigdl for scalable deep learning innovation. (https://www.gigaspaces.com/blog/gigaspaces-to-demo-with-intel-at-strata-data-conference-and-microsoft-ignite/).

[54] Apache Kafka. (https://kafka.apache.org/).

[55] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale in *The Twenty-Fourth ACM Symposium on Operating Systems Principles.* (2013) SOSP'13.