Extreme-Scale Online Machine Learning On Stream Processing Platforms

Vissarion Berthold Konidaris

A thesis submitted in partial fulfillment of the requirements for the masters of science of Electrical and Computer Engineering

M.Sc in Electrical and Computer Engineering
focusing on Computer Science and Engineering

submitted to the
School of Electrical and Computer Engineering
of the Technical University of Crete

Date of Defence: 21 July 2022

Supervisor

Prof. Vasilis Samoladas
School of Electrical and Computer Engineering, Technical University of Crete

Committee

Prof. Minos Garofalakis
Technical University of Crete

Prof. Lagoydakis Mihail
Technical University of Crete

Abstract

Online Machine Learning (OML) techniques support training over continuous unbounded training items while simultaneously providing predictions on the same or another unlabeled stream. The explosion in the amount and complexity of digital information generated online is gradually rendering OML techniques essential for modern analytics and forecasting applications due to their ability to handle massive, unbounded, and most importantly, inherently not-static data. Having noted that support for popular Machine Learning (ML) tool-chains is somewhat weak for the OML setting, we have designed the Online Machine Learning and Data Mining (OMLDM) component, a state-of-the-art engine for effortlessly deploying OML pipelines on streaming platforms. Our prototype, built on Apache Flink, validates our architecture, and identifies issues that current streaming platforms should improve on to support OML. To achieve high performance, OMLDM supports distributed online learning by utilizing the Parameter Server paradigm. We have identified the communication cost of synchronizing distributed learners as the major impediment to scalability. To overcome this obstacle, our proposed engine supports several popular model synchronization strategies. In addition, we bring forward and evaluate a novel synchronization strategy, Functional Dynamic Averaging (FDA), that minimizes the prediction loss and network communication all at once. We demonstrate through experiments that FDA is superior to current model synchronization strategies in many settings.

Contents

1 Introduction

Modern analytics and forecasting applications can benefit immensely from Machine Learning techniques. In traditional offline machine learning (ML) settings, training a model is handled apart from and before deploying it for prediction. Conventional ML approaches follow a well-structured and sometimes recursive design cycle. More precisely, the process starts by collecting, cleaning, and transforming the data into a usable dataset. Then, studying and carefully selecting meaningful information is necessary by performing feature selection, feature extraction, and feature creation techniques. Afterwards, a model is selected and trained on the carefully manufactured dataset. Lastly, an evaluation is done on the model to determine its performance. The analyst can always recurse back in each of these steps to perfect its approach. When the training is over, the model can be used to make predictions. This approach works well when models are inherently static or slowly changing, and training data is available a-priori. However, for a diverse number of applications, the offline style is inadequate. Labeled training data can become available late, multiplexed in time with unlabelled data, and the model must adapt continuously to concept drift. These scenarios call for Online Machine Learning (Online ML, or OML, for short) techniques and systems.

OML algorithms are ML algorithms that are trained continuously on unbounded data streams and update themselves incrementally whenever new data arrives. These methods are proven to be very practical in cases when it's computationally infeasible to train some model on the entire dataset. Furthermore, data is generated on the fly by some unknown distribution. It is usually the case that the

continuously created items will stem from a dynamic environment, causing a change to the characteristics of the data stream and hence may lead to the degradation of the predictors' performance. This challenging issue is known as concept drift (CD), in which statistical properties of the input features and target classes or values may shift over time. Many OML algorithms can detect and adapt to those changes. Confusingly, in some of the literature, the term Online ML is often used to refer to incremental ML algorithms, where training happens offline, but the training dataset is processed in a streaming fashion; indeed, most ML algorithms are incremental. In this paper, OML refers to the situation where fitting and predicting co-exist in space and time.

As a motivating example, consider the short-term forecasting problem. Assuming t now stands for the current time, the goal is to estimate a future value $Y(t_{now} + T)$ from the current situation $X(t_{now})$. In a learning setting, a stream processing system may generate an online stream of training data where the ongoing training sample can be the pair $(X(t_{now} - T), Y(t_{now}))$.

Another area of interest, exemplified by spam handling in personal emails, falls under the OML category because of online user interaction: a user may label a previously undetected mail as spam, or conversely.

1.1 OMLDM

Our focus in this study is the Online Machine Learning and Data Mining (OMLDM) architecture, an innovative implementation of the Parameter Server paradigm that is deployable on top of a stream processing platform alongside other data analysis tasks. The proposed architecture has three levels of abstraction. At the lowest level, it is a streaming application able to digest and process massive quantities of

data at a high rate. One level above is an abstract network of independent computing nodes, more accurately a complete bipartite graph with high-level primitives for communication and synchronization. On top of this level, it is a feature-rich, parallel OML pipeline, able to host a variety of OML techniques and coordinate them with varying strategies.

Most cloud-based ML frameworks and libraries aim to meet the needs of a batch setting (e.g., MLLib, FlinkML, scikit.learn, Mahout, e.t.c.), even if they execute on a streaming platform. To our knowledge, the only platform with a clear commitment to the OML setting is Apache SAMOA [KDB19], from Yahoo! Barcelona. Yet, the architectural decisions in SAMOA were different. Although an abstract API for communicating nodes exists, it is not mature or flexible enough for use by developers. By contrast, the OMLDM architecture strives for convenience; OMLDM pipelines orchestrate learners without the need for additional coding. As such, our philosophy is very close to Petuum (http://petuum.com). However, Petuum is appropriate for the batch setting.

The testing and development of the OMLDM engine has been a part of the Interactive Extreme-Scale Analytics and Forecasting (INFORE) research project, a prototype supporting non-expert programmers in performing optimized, cross platform, streaming analytics at scale. INFORE's aim is to address the challenges posed by huge datasets and pave the way for real-time, interactive extreme-scale analytics and forecasting. The project was the first holistic approach in streaming settings and was developed on top of the fields of life science, financial data analysis and maritime awareness problems. Being a part of a bigger project OMLDM needed to be able to communicate effectively with various other components and systems. Hence, our component provides simple interconnecting capabilities by an easy to use JSON API.

1.2 Reducing communication in distributed OML

A principal technical contribution of this study is Functional Dynamic Averaging (FDA), a model synchronization strategy for parameter server-based architectures. In batch learning, ML computations are at the center stage and can consume computational resources voraciously. Distributed learning utilizes more computational power to reduce training time. Most distributed ML protocols assume some form of periodic synchronization of local learners with the parameter server. This synchronization is done typically once for every mini-batch fitted. For large OML models, this approach can generate massive amounts of network traffic and throttle overall performance.

However, an OMLDM deployment must coexist with other data processing tasks under the management of a stream processing engine and have the capacity to tolerate limits to resource use. Motivated by previous work by Kamp et al. [Kam+14; Kam+16], we have designed a novel distributed online learning protocol, which can reduce communication compared to standard minibatch-based training on the Parameter Server.

The salient idea of the FDA is to apply techniques of distributed stream monitoring [CMY08] to determine *model variance*, a quantity related to the differences between models of different learners. Synchronization occurs only when model variance exceeds a certain threshold. We provide some analytical consequences of this scheme, supporting its fundamental premise. On the algorithmic side, accurately monitoring model variance cannot be done cheaply. We resort to approximations instead. We employ ideas from Functional Geometric Monitoring [SG] and propose three variants. An experimental evaluation of this technique shows that it can reduce the communication cost of OML by an order of magnitude.

2 Preliminaries

We use the standard notation to define Online ML concepts. We shall focus on supervised learning for simplicity purposes, but similar concepts apply to learning. Typically, we have a data stream of items produced from one or multiple sources. An *item* is a data record referring to an entity of interest, typically represented as a feature vector. Let the set of all items (feature vectors) be $\mathcal{X}$. To each item corresponds a *label*, coming from the set of all possible labels $\mathcal{Y}$. Labels *label* can be categorical, for classification, or real-valued, for regression. The goal of ML is to construct a *model*, a function $\mathcal{X} \to \mathcal{Y}$, which can *predict* the label corresponding to an item. The model is usually a member in a parametric family of functions, and the parameter vector denoted by $\mathbf{w} \in \mathcal{W}$, where $\mathcal{W}$ is the vector space of all parameter vectors. The *predictor* function is $P : \mathcal{W} \times \mathcal{X} \to \mathcal{Y}$. In the discussion, we shall use the terms parameter vector, *weight vector* and model interchangeably.

In the Online ML setting, at time t, a learner receives either an *unlabelled* sample $\mathbf{x}_t \in \mathcal{X}$, usually some feature vector, or a *labelled* sample $\mathbf{z}_t = \langle \mathbf{x}_t, \mathbf{y}_t \rangle \in \mathcal{X} \times \mathcal{Y}$. In the former case, the learner reports its current *prediction* $\hat{\mathbf{y}}_t = P(\mathbf{w}_t, \mathbf{x}_t)$. In the latter case,

1. The learner suffers loss $\ell(\mathbf{w}_t, \mathbf{z}_t)$, and

2. the model is adjusted to $\mathbf{w}_{t+1}$, by some *update rule*,

$$\mathbf{w}_{t+1} = \phi(\mathbf{w}_t, \mathbf{z}_t, \eta_t)$$

where η_t is a learning rate (which generally varies with time).

To evaluate the learning outcome, losses are accumulated in each round, and the total loss is contrasted to the best possible model after-the-fact (called the *empirical risk minimizer*). The difference, called the *regret* after T steps, is defined as

$$regret(T) = \sum_{t=1}^{T} \ell(\mathbf{w}_t, \mathbf{z}_t) - \min_{\mathbf{w} \in \mathcal{W}} \sum_{t=1}^{T} \ell(\mathbf{w}, \mathbf{z}_t).$$

As witnessed from the above formula, Regret grows with T. Hence, the goal of an OML algorithm is to achieve sublinear regret. Under broad assumptions [BPS09], it is theoretically possible to have

$$regret(T) = O(\sqrt{LD(\mathcal{W}) \cdot T \log T})$$

where $LD(\mathcal{W})$ is the Littlestone dimension of the set of models $\mathcal{W}$.

In general, regret should grow sublinearly with T, so that for large T the *average* regret $regret(T)/T$ vanishes, and therefore the predictive outcome of online-learning, will on average be comparable to the Empirical Risk Minimization outcome.

2.1 Distributed Online ML and the Parameter Server architecture

The most widely accepted settings in distributed training are the data-parallel setting and the model-parallel one. In model parallelism, each processing machine is responsible for the computations of only a fraction of a learning model. For example, in a distributed neural network, each local worker may be assigned with the calculations of a single layer. On the other hand, in data-parallelism, each local worker has a complete copy of the neural network and witnesses a fraction of the data set. However, the two methods are not mutually exclusive. They can be fused into a hybrid approach.

For the distributed case, OMLDM adopts the data-parallel setting, where there is a collection of k learners, each learner executing on a separate computing node. The local model at learner i (for $1 \leq i \leq k$) at time t is denoted by $\mathbf{w}_t^{(i)}$. Samples (labelled or not) are streamed to these nodes learners; the sample received by learner i at time t is $\mathbf{z}_t^{(i)}$. Furthermore, there is a Parameter Server [Jia+17; Li+14; SN10], a distinguished processing node which maintains a global model $\tilde{\mathbf{w}}_t$.

There are different strategies for synchronizing the rounds between learners. For batch processing, two simple approaches are the synchronous (TSP) and the asynchronous (ASP) method [Ver+20]. In both, a round consists of processing a single mini-batch. In the asynchronous case, rounds on different learners happen independently, whereas in the synchronous case, learners synchronize so that all learners are on the same round. The asynchronous method avoids delays at the learners and achieves the best computational performance, but sometimes at the expense of learning quality. Each learner executes in sequence several rounds, where each round has the following stages:

Push: the PS pushes $\tilde{\mathbf{w}}_t$ to some learner i and the learner i updates $\mathbf{w}_t^{(i)}$ based on $\tilde{\mathbf{w}}_t$ (e.g., by assignment $\mathbf{w}_t^{(i)} \leftarrow \tilde{\mathbf{w}}_t$).

Fit: Learner i fits its local model by applying streaming inputs and the update rule, for some number of steps, determined by the synchronization algorithm. The learner maintains the update $\Delta_t^{(i)}$, which is the difference of its current model and the model at the beginning of this fit step.

Pull: At some later time t', the PS *pulls* $\Delta_{t'}^{(i)}$ and updates $\tilde{\mathbf{w}}_{t'}$. Updating is typically done by weighted summation, $\tilde{\mathbf{w}}_{t'} \leftarrow \alpha\tilde{\mathbf{w}}_{t'} + \beta\Delta_{t'}^{(i)}$

2.2 Parameter Server (PS) architecture Synchronization Strategies

We now dive deeper into some of the most used synchronization strategies of the PS architecture. All the algorithms are going to be presented for the online distributed streaming scenario where OML is used. Each worker receives its own data stream where it buffers items until they comprise of a single mini-batch. When the mini-batch is formed then it is used locally by the workers' learner to update the local model. The mini-match is discarded afterwards. The Synchronization Strategies are similar for the offline scenario with the only difference being in the workers having their whole chunk of the entire dataset from the start. Then they use their distinct local dataset to sample a mini-batch to fit. Most of these strategies are implemented in various libraries, like Tensorflow and Spark, that support distributed training for traditional offline (batch) ML algorithms.

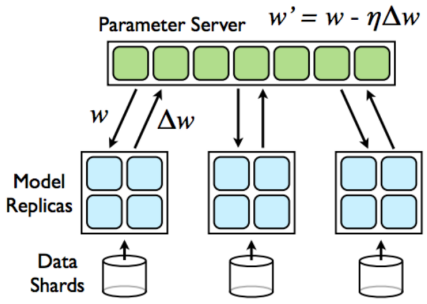

Figure 2.1: Parameter Server architecture.

2.2.1 Bulk Synchronous Parallel (BSP)

The k local models in the BSP setting are combined together via parameter averaging. We present BSP using the Gradient method as the update rule, but the procedure applies to many other update rules. Parameter averaging takes place after each worker i has fitted a mini-batch to its model. Additionally, the setting imposes the constraint that each worker should observe the same number of examples as the rest. Assuming a mini-batch of size m, then the weight update rule

by a single machine is given by

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{km} \sum_{i=1}^{km} \nabla_{\mathbf{w}_t} \ell_i,$$

where ℓ_i is the loss function suffered by the training example i. When we separate this formula to k parallel workers the update rule becomes

$$\mathbf{w}_{t+1} = \frac{1}{k} \sum_{i=1}^{k} \mathbf{w}_t^{(i)} = \frac{1}{k} \sum_{i=1}^{k} \left(\mathbf{w}_t - \frac{\eta}{m} \sum_{j=(i-1)m+1}^{im} \nabla_{\mathbf{w}_t} \ell_j \right) = \mathbf{w}_t - \frac{\eta}{km} \sum_{i=1}^{km} \nabla_{\mathbf{w}_t} \ell_j,$$

a formula that is identical to the single machine setting. Regarding a mini-batch fit for each worker as a round, then in each of these rounds the communication cost incurred by the messages that are send from the workers to the PS is $\Theta(kD)$, and the communication cost incurred by the messages send from the PS to the workers is again $\Theta(kD)$. Hence, the total communication cost of a single round is $\Theta(2kD)$ bytes. Assuming that the entire data set is of size M and that each worker fits m examples to its model per round, then the total number of rounds is $\lceil \frac{M}{km} \rceil$. Consequently, the grand total communication cost of BSP algorithm in terms of bytes is $\Theta(2D\lceil \frac{M}{m} \rceil)$. The algorithm of this distributed training process proceeds as follows:

1. The PS initializes the parameters $\tilde{\mathbf{w}}$ of the learner.
2. The PS broadcasts the parameters to each worker $i \in [1, k]$.
3. Each worker fits its local model on a mini-batch, comprised of the oldest M items that it received from its stream, and discard it.
4. All workers send their parameters $\mathbf{w}^{(i)}$, or their updates $\mathbf{\Delta}^{(\mathbf{i})}$, to the PS were they are averaged by using the weighted summation

$\tilde{\mathbf{w}} \leftarrow \tilde{\mathbf{w}} + \frac{1}{k} \sum_{i=1}^{k} \Delta^{(i)}$. During these phase, all incoming items to the workers are buffered.

5. While there is more data return to step 2.

Parameter averaging is usually performed after each iteration, or else after each worker has observed and fitted a single mini-batch. Although this approach improves dramatically the convergence properties of the training process, it also introduces a considerable amount of overhead to the network. Previous research on the subject [SC15], suggests that averaging once every 10 to 20 mini-batches per worker can still perform well, exchanging the predictive performance for communication gain. In addition, even though BSP outperforms other distributed learning techniques in terms of predicting accuracy, it suffers from the so called 'last-executor' affect, meaning that a synchronous system like this will always have to wait on the slowest executor before completing each iteration. This can be a problem when the total number of workers increases, making BSP a non viable solution in large distributed settings.

2.2.2 Total Asynchronous Parallel (TAP)

A conceptually similar approach to parameter averaging is the update-based data parallelism. In this setting, only the updates of the parameters are send by the workers instead of the parameters themselves. Additionally, each worker sends its update as soon as the fitting process completes, and the PS immediately adds this term to the global weights. The PS then sends back to the worker the new updated parameters. Hence, the updates are done in an asynchronous manner.

This gives an update of the form

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \frac{1}{k}\sum_{i=1}^{k} \Delta\mathbf{w}^{(i)},$$

with $\Delta\mathbf{w}^{(i)}$ encompassing, as before, the update that the optimizer imposes after the observation of a mini-batch by site i (i.e. for the vanilla DG optimizer, the update term for a worker i is $-\frac{\alpha}{m}\sum_{j=1}^{m} \nabla_{\mathbf{w}^{(i)}} \ell(\mathbf{x}_j^{(i)}, y_j^{(i)})$). The communication cost is similar to the S-DSGD method.

A-DSGD, being an asynchronous method, gets rid the 'last-executor' problem, thus gaining a big advantage against S-DSGD in terms of data throughput, and hence execution speed. Workers can also be tuned to apply their gradients after more than one mini-batch has been observed, just as in S-DSGD, providing further throughput gains. On the downside, A-DSGD on its simplest form can result to high staleness values for the gradients. This means that, as the calculation of gradients take time, by the time a worker has finished these calculations and applies its results to the global parameters, the parameters may have been updated a number of times. In realistic scenarios this problem can slow down the predictor's convergence significantly. Many variants of A-DSGD try to alleviate this problem and have been shown to improve convergence over the naive implementation of the A-DSGD algorithm, by utilizing some form of synchronization procedure [Ho+13b; Zha+16].

2.2.3 Stale Synchronous Parallel (SSP)

This synchronization method is a compromise between BSP and TAP, proposed in [Ho+13a]. In the BSP scenario all learners synchronize with each other after a designated number of training data, hence they are forced to be on the same round during the whole training procedure. On the other hand, TAP enforces no such constraint. Each

learner is left free to operate on its own, not having to care about the learning round of other remote learners. Thus, learners in TAP can all be on a different learning round of arbitrary difference between them. The SSP finds a middle ground. It is not as restrictive as BSP and not as as TAP. In SSP all learners can be on a different learning round as long as no learner is $s + 1$ rounds ahead of any other, where s is a positive integer used as a bound. In essence, it is a TAP method with a bound in the difference of learning rounds between the faster and slower learners. When, s get close to infinity SSP transforms to TAP. On the other end of the spectrum, when s is equal to zero then SSP becomes identical to BSP.

2.2.4 Elastic Averaging (EA)

In this strategy, proposed for SGD in [ZCL15], a synchronization between learner and the PS is performed by an "elastic force" $\mathbf{F}_t^{(i)} = \rho\left(\mathbf{w}_t^{(i)} - \tilde{\mathbf{w}}_t\right)$, which is added to $\tilde{\mathbf{w}}_{t+1}$ and subtracted from $\mathbf{w}_{t+1}^{(i)}$. This strategy offers better exploration of the model "landscape" by the learners, and therefore better generalization, while it is still efficient. However, stability has only been proved in the context of SGD. EA can be used within a BSP and in a TAP setting for the simple reason that it only alters the update rule of the learners. In our architecture we will use its asynchronous variant, Asynchronous Elastic Averaging Stochastic Gradient Descent (EASGD) for the simple reason that it is more applicable on the online learning scenario (as all Asynchronous methods are due to their robustness against back-pressure).

2.3 Online Machine Learning Libraries

The rise of computational power combined with industry digitalization and the immense data explosion of the last decades have made

ML methods very popular in numerous fields. Their ever-expanding popularity has led to the development of a myriad of ML libraries and frameworks in various programming languages, each one covering different needs.

Unfortunately, Online ML methods have yet to be established in the development community. Most cloud-based ML frameworks and libraries aim to meet the needs of offline learning scenarios. Namely, they cover the batch setting, even if they execute on a streaming platform. The OML field was conceived to tackle the problem of Big Data, the uncontrollable and rapid generation of unbounded digital information. One of the first Big Data frameworks developed to handle computations on big data sets in a distributed manner was the Apache Hadoop Map-Reduce framework. Apache Mahout, a distributed linear algebra framework implemented on top of the Hadoop Map-Reduce framework, was one of the first attempts to parallelize ML. However, those platforms were designed to perform computations on static datasets. Hence, Online ML could not be supported.

The first widely used and robust platform for online streaming computations was the Apache Spark framework. It is a multi-language engine for executing data engineering, data science, and ML on single-node machines or clusters and is one of the most widely used platforms for distributed ML training. Although it provides streaming processing capabilities, its extensive set of distributed ML algorithms is built only to train on static datasets. The first true OML library that we came up upon was FlinkML, a library of online machine learning algorithms built on top of the streaming engine Apache Flink. However, the library was deprecated and removed from the ecosystem of Flink after version 1.9. Many other well-known libraries support distributed ML, like Tensorflow and Pettum, but they also are only appropriate for the batch setting.

Scikit-Multiflow and MOA are libraries (the former written in Python and the latter in Java) that support the OML setting. Nevertheless, the number of algorithms they provide is limited, and they support only centralized learning.

To our knowledge, the only platform with a clear commitment to the distributed OML setting is Apache SAMOA, from Yahoo! Barcelona. Yet, the architectural decisions in SAMOA were different. Although an abstract API for communicating nodes exists, it is not mature or flexible enough for use by developers. Motivated by the lack of a truly dedicated distributed Online Machine Learning framework, we propose the OMLDM architecture that can be implemented on top of any streaming platform and can potentially support any online ML pipeline.

2.4 Distributed Frameworks

In the last decade, we have been embarking on a new digitized era with the generation of data exploding in terms of volume, velocity, variety, veracity, and value. Traditional Database Management Systems were more than enough to handle the data before the new digitized era came along, but now they fall short. For that reason, new frameworks we developed for filling this vacuum in technological need, Distributed Big Data frameworks. Frameworks like these are built for the sole purpose of handling massive amounts of data. The way they accomplish that is by utilizing more than one computational machine. In essence, they utilize a network of computers, or mostly known as a cluster of computers, and by allocating distributed computational power and storage they solve computational problems of massive and sometimes unbounded datasets. Such frameworks are optimized for memory and disk utilization and provide strong guarantees against machine failures. We chose to build our OML module on the Apache

Flink, for distributed manipulation of data streams, and Apache Kafka as a distributed stream messaging system.

2.4.1 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. It runs in all familiar cluster settings and resource managers (such as Hadoop YARN, Apache Mesos, and Kubernetes) and performs computations at in-memory speed and any scale. For our experiments, we used a Hadoop YARN environment.

A distributed stream processor must possess the ability to recover from failures and run streaming applications 24/7. For that reason, Apache Flink provides strong guarantees against machine and process failures. It does that by using a checkpoints system, thus ensuring that its internal state remains consistent after a failure or during a restart of the distributed application. The checkpointing system utilizes three notions of time for keeping the internal states consistent with each other.

- **Processing Time**: Processing time refers to the system time of the machine that is executing the respective operation.
- **Event Time**: Event time is the time that each event occurred on its producing device. This time is typically embedded in the form of a timestamp within the records before they enter Flink.
- **Ingestion Time**: Ingestion time is the time that events enter Flink.

When coding an application in Apache Flink we essentially create a pipeline of data transformations. The framework forms, upon compiling, a Directed Acyclic Graph (DAG) of this pipelined transformation

that the user can upload into a cluster to run it in parallel or run it in a single local machine.

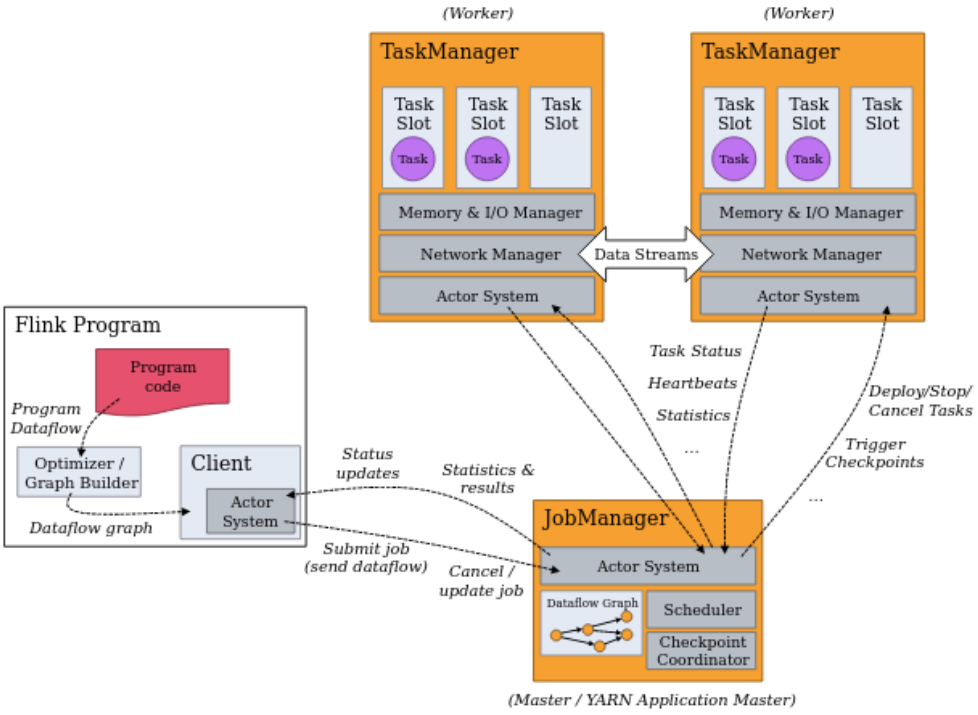

Figure 2.2: Apache Flink cluster architecture.

Within a Flink cluster, we have two types of nodes, namely Master Nodes and Worker nodes. Each node type runs a different process that serves a unique purpose within the distributed application. A Master node, which should be at least one within a cluster, runs a JobManager process. The JobManager has a plethora of responsibilities related to coordinating the distributed execution of Flink Applications. It decides when to schedule the next set of tasks, reacts to finished ones or execution failures, coordinates checkpoints, and coordinates recovery on failures, among others. Each worker node incorporates a

TaskManager that undertakes the physical execution of tasks. The TaskManagers, which are essentially JVM processes, execute the tasks of data-flow and buffer and exchange the data streams. There must always be at least one TaskManager. The smallest unit of resource scheduling in a TaskManager is a task slot. Each task slot represents a fixed subset of resources of the TaskManager and their number indicates the number of concurrent processing tasks. Tasks in the same TaskManager share TCP connections (via multiplexing) and heartbeat messages. They may also share data sets and data structures, thus reducing the per-task overhead.

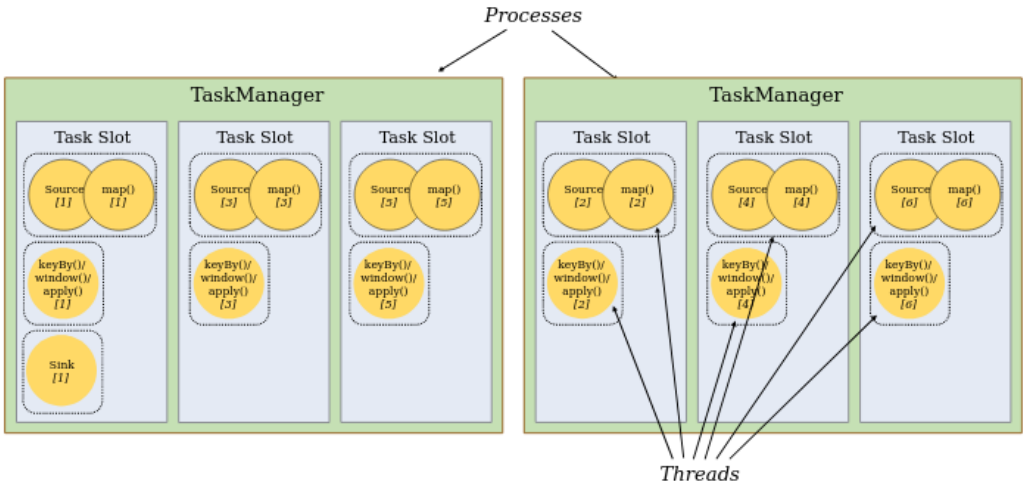

Figure 2.3: Apache Flink cluster architecture.

Apache Flink offers an API for implementing transformations on data streams, the DataStream API. The API supports various data sources (e.g., Kafka, Cassandra, Elasticsearch, socket streams, files) and sinks. It also supports a handful of operators for transforming streams (e.g., map, flatMap, filtering, reduce, e.t.c.). Flink operators may run in several instances executing the same task but on different

data partitions. They divide into sub-tasks and are assigned to a slot. Subtasks of distinct operators can share the same slot, thus leading to better resource utilization. Here are some of the many more useful operators that Apache Flink offers.

- **Map**: Takes one element and produces one element. A map function that doubles the values of the input stream.
- **FlatMap**: Takes one element and produces zero, one, or more elements.
- **Process**: A low-level stream processing operation, giving access to the basic building blocks of all (acyclic) streaming applications.
- **Connect**: "Connects" two data streams retaining their types. Connect allowing for shared state between the two streams.
- **CoMap & CoFlatMap**: Similar to map and flatMap on a connected data stream.
- **Union**: Union of two or more data streams creating a new stream containing all the elements from all the streams.
- **Filter**: Evaluates a boolean function for each element and retains those for which the function returns true.
- **KeyBy**: Logically partitions a stream into disjoint partitions. All records with the same key are assigned to the same partition. Internally, keyBy() is implemented with hash partitioning. We utilize such an operator to send messages from the local workers to the PS in our distributed OML problem.
- **KeydProcess**: A Process with access to keyed state and timers.
- **Reduce**: A "rolling" reduce on a keyed data stream. Combines the current element with the last reduced value and emits the new value.

2.4.2 Apache Kafka

Apache Kafka is a highly scalable messaging system written in Scala and Java back in 2011 by LinkedIn data engineers. In a more precise manner, Apache Kafka is a publish-subscribe-based durable messaging system built for exchanging data between processes, applications, and servers. Its key design principles were formed based on the growing need for high-throughput architectures that are easily scalable and provide the ability to store, process, and reprocess streaming data.

The cornerstone of Kafkas' architecture is the Kafka topics. A topic is a category/feed name to which records are stored and published. Alternatively, a topic can be interpreted as a messaging pipe between applications where they can add, transfer, process, and reprocess records that run through them. Topics are divided into several partitions where each one contains a set of key-value records ordered and uniquely identified by their offsets. Moreover, records are stored and indexed within partitions along with a timestamp and other optional attributes.

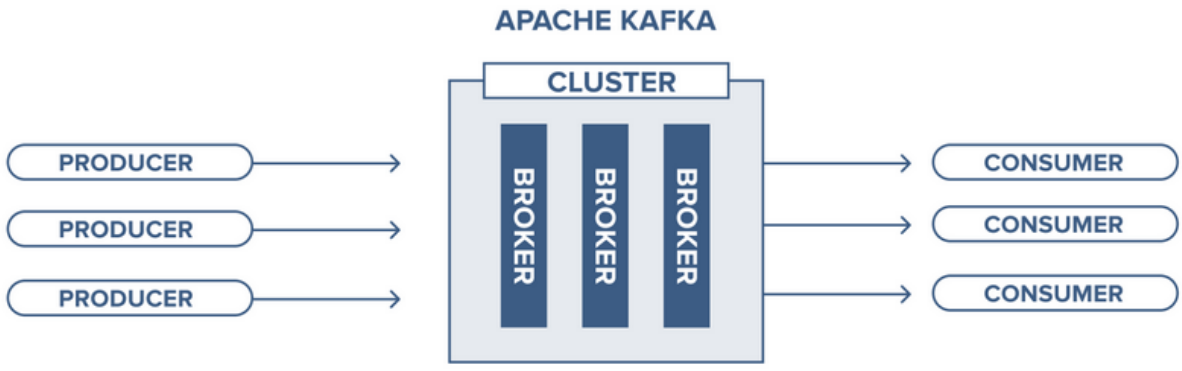

Figure 2.4: Apache Kafka cluster architecture.

Applications that subscribe to a specific topic have read and write permissions to it. The subscribed applications can read records from the partitions and use them for algorithmic purposes. Apps can also read records multiple times. This is possible because once records are published to the cluster they stay in the cluster until a configurable retention period has passed by.

These are the four main parts of a Kafka system.

- **Broker**: Handles all requests from clients (produce, consume, and metadata) and keeps data replicated within the cluster. A Kafka cluster consists of one or more servers (Kafka brokers) running Kafka.
- **Zookeeper**: Keeps the state of the cluster (brokers, topics, users). It is also responsible for managing the brokers within the cluster. There may be multiple Zookeepers in a Kafka cluster.
- **Producer**: Producers are processes that push records into Kafka topics within the broker.
- **Consumer**: Consumes batches of records from the broker. They can also be attached to a consumer group. If two of them are in the same group, they will read different partitions of a topic.

3 OMLDM Architecture

Let us now present the structure of our approach to the problem of distributed OML. The OMLDM architecture is composed of three main layers, the Online Machine Learning library, the Network/Middleware (N/M) layer, and the kernel or back-end layer. The hierarchy of those layers can be depicted in Fig. 3.1.

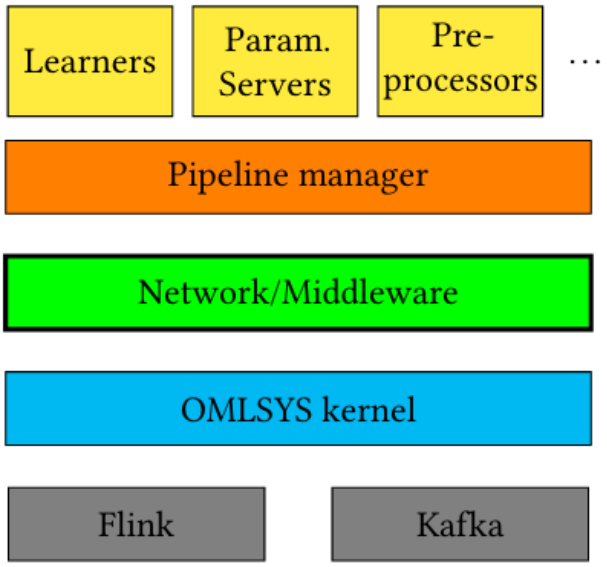

Figure 3.1: The OMLDM software stack.

On top of the OMLDM stack, there is the computational layer. Preprocessing techniques, Machine Learning algorithms, and Data Mining methods live and execute at this level. One level below, and within this library, machine learning pipeline management orches-

trates computation, provides generic services for these components, and controls their life-cycle. Lastly, the distributed learning logic exists in this layer by the algorithmic implementations of workers and Parameter Servers within a distributed learning setting.

Following the computational layer, the Network/Middleware (N/M) layer is at the heart of the architecture. It implements an abstraction of a network of processing nodes and a high-level middleware API for communication between them. The abstract network that the layer implements take the form of a fully connected Bipartite Graph. The purpose of this design choice will become much more translucent in our thorough summary of the N/M layer.

Finally, below the Network layer lies the online streaming kernel. The kernels' current implementation is in Flink and Kafka; in the future, we are contemplating executions on other Big Data infrastructures such as Spark and Akka. Kafka is currently used only as a source and destination for streaming data. Additionally, it serves as a fault-tolerant data buffer between Flink operators. Although Flink and Kafka are highly versatile stream processing platforms, implementing the operations of the network layer presented us with several technical challenges that we will be discussing later in this chapter.

The purpose of deconstructing the OMLDM into three components was to make it more versatile and extendable to the user. This approach provides the user the ability to extend or to completely change the upper layer (the OML library), as well as the lower layer of the architecture (the streaming kernel), without the necessity of having to deal with the rest of the module. We now advance to describe the layers of our architecture in more detail. After that, we will see how they all tie together to provide the user with a plethora of capabilities in the playground of data mining and online machine learning.

3.1 The Network/Middleware layer

The N/M layer, written in Java, mediates between the computational code and the streaming kernel, making each oblivious to the other. This layer serves as a middleware to the whole architecture that glues together our high-level distributed OML library to the low-lever streaming kernel. The primary abstraction to support this, a **Network**, is a collection of processing nodes that manages the transfer of messages between them, incoming and outgoing streaming traffic, and execution of tasks.

3.1.1 Networks

A network comprises a bipartite graph of nodes. Nodes can be of two types, either spokes or hubs. Spokes are the nodes receiving the distributed input stream; apart from that, there are no differences between spokes and hubs. All nodes can generate streaming output, either messages to other nodes or streaming network statistics. See Fig. 3.2. In the distributed OML implementation, we intend for the spokes to host local learners, whereas hubs collectively implement the Parameter Server.

The reason for choosing a bipartite graph distributed topology of nodes and not a classic star network topology was for the sole purpose of dividing the parameter server into multiple nodes. As the models grow in size, as in a deep learning model, it would be naive to hold the global model and all the computations necessary to a single PS. A single point of an immense number of calculations could not hold up to Big Data scenarios. Hence, our architectural network topology enables distributed learning scenarios with multiple PSs sharing the computational load.

Communication can only occur between nodes of different types. Meaning that spokes cannot communicate with other spokes, and the

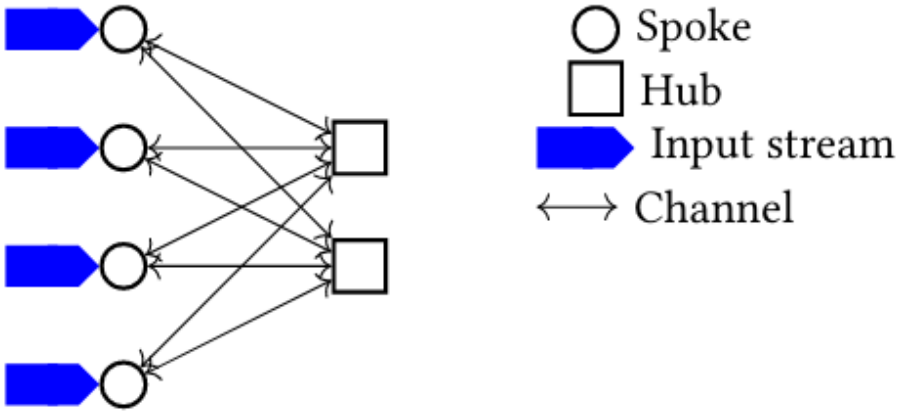

Figure 3.2: A network with 4 spokes and 2 hubs.

same goes for hubs. Each spoke and hub node can send messages to each other via bidirectional, reliable FIFO channels. These channels can carry serializable messages of arbitrary size. Atomic broadcast is also supported, meaning that a node can ship copies of the same message to every node of the opposite type. This functionality can significantly reduce the amount of communication between the nodes. As we will see soon afterward, we use such methods to broadcast learning models to spokes.

The N/M layer is a collection of Java interfaces and abstract classes. The instances of nodes are Java objects whose actual type is opaque to the kernel and that the OML layer implementations need to extend. They provide methods that the back-end kernel can call to deliver messages, streaming data, or control commands. A node can only execute one of these calls and does not have the clearance to create its own threads or block any calling thread. Lastly, nodes cannot share states with other nodes.

3.1.2 Middleware

Remote Method Invocation (RMI) is a Java API that allows an object to invoke a method on a Java object that exists in another address space, which could be on the same machine or a remote one. Motivated by this API, we created a middleware of our own, an asynchronous RMI paradigm that allows programming distributed algorithms by high-level, type-checked, and easily readable code.

Each node implements a remote interface. In addition, the middleware provides each node with a **proxy** object for each node of the opposite type. What is more, all nodes of the same type implement a common remote interface, and a **broadcast proxy** is provided to each node. As a result, each node can communicate with the opposite type nodes one by one directly, or it can broadcast a message to all of them by using their provided proxies. These proxy objects are implemented using Java annotations and run-time reflection.

Remote methods can return values. The caller of a remote method can specify a *callback* to handle the return value. For example, the following line calls the user defined remote method doFoo, via the proxy given to user by the middleware, and provides the calling node's processFooResult method, also user defined , as the callback to handle the response.

```
proxy.doFoo(x, y).then(this::processFooResult);
```

When calling a remote method on the broadcast proxy, the callback is invoked once for each response received. Hence, the middleware provides a node with the following functionalities.

1. To send a **one way** message to a node of the opposite side, without waiting for any response.

2. To send a **two way** message to a node of the opposite side and

expect for a **response** message. The caller can also allocate a callback function to the answer.

3. To broadcast a **one way** message to all nodes of the opposite side.

4. To broadcast a **two way** message to all nodes of the opposite side and expect their responses. The same callback function will be executed for each one of the responses. Hence, any difference in functionality must be implemented by the user in the callback function.

The middleware also provides for deferred responses via **promises**. More often than not, the callee may not possess the reply to the call made to it right away. In those cases where the callee cannot immediately respond to the calling node, it may return a promise object. The promise object is bound to this specific call. When the response value becomes available, the callee provides it to the promise object ("fulfills" the promise) and ships the response message to the calling node. Multiple promises can be given to a calling node and can be "fulfilled" at once by providing the list of all the answers, or one by one by providing only the available ones.

Finally, there is a *broadcast promise* facility. A broadcast promise object can be returned by a remote method instead of a response value. The same broadcast promise object can be bound to many calls. A response message is broadcasted to all callers, when

(a) the broadcast promise object has been bound to every node of the opposite type, and

(b) the promise is fulfilled.

3.1.3 Synchronization

Broadcast promises are essentially a synchronization mechanism. To see this, observe that they can be used to trivially implement *barrier* synchronization between all nodes of the same type (just create a remote function that returns a broadcast promise which is already fulfilled).

Aside from broadcast promises, the middleware supports only limited facilities for synchronization between nodes. In distributed systems parlance, message delivery is FIFO-consistent. Stronger forms of consistency, including causal and sequential consistency, are not provided, in order to keep middleware implementation simple. We are not aware of a need for such consistency guarantees in the domain of machine learning.

However, the middleware does provide for synchronization between messages and the incoming stream for spokes. It is possible to pause the processing of streaming inputs until an RMI call has returned. This pause may cause back-pressure to other parts of the implementation, which the kernel may be able to exploit for more efficient processing. We will be using our previous example to demonstrate how a user may do this. The following line once again provides a synchronous callback method processFooResult to the answer of the remote method doFoo.

```
proxy.doFoo(x, y).thenSync(this::processFooResult);
```

The processing of the stream will resume immediately after the arrival of the response message and the execution of the callback method. The motivation for this functionality came to place after the need of machine learning workers to stop fitting their local model on incoming data when they await to receive the newly requested global model.

3.2 The kernel layer

The kernel layer is responsible for delivering streaming data to nodes, and transferring messages between them, in an efficient and scalable manner.

3.2.1 Kernel implementation

For each node of a network, the N/W layer provides to the kernel a node object N, with methods that the kernel can call to deliver streaming tuples and incoming messages to the node.

For each node object N, the kernel must construct a *network context object*; this object is used by the middleware to implement the RMI proxies given to N. Methods on the network context object are called by the N/M layer to send messages from n to other nodes and support streaming synchronization.

Messages and streaming items are treated as blobs by the kernel. The kernel can encapsulate these messages into its own message objects, adding whatever routing, timing or other metadata is required. Furthermore, as messages can be arbitrarily large, the kernel may decide to fragment them in order to manage them more efficiently.

3.2.2 Kernel on Flink and Kafka: lessons learned

We chose to build our kernel layer using the Apache Flink distributed framework for streaming data manipulation and the Apache Kafka for stream messaging. The implementation is written in Scala due to its inherent nature in distributed programming. Choosing the distributed framework Apache Flink as our main kernel for multi-clustered stream computations in concert with Apache Kafka as its input and output origin wasn't an arbitrary choice. The two distributed frameworks performing in unison can provide all the prin-

cipal requirements for managing extreme-scale interactive analytics in stream processing fashion with fault-tolerance and exactly-once guarantees. The implementation of the bipartite networks is built via Apache Flink. Spokes and hubs, along with their algorithmic processing of the data and messages, live inside a Flink Directed Acyclic Graph (DAG) job. For providing streaming inputs and collecting streaming outputs we use Apache Kafka.

Despite the simplicity of the kernel-N/M contract for scalable and robust kernel implementation, significant attention must be paid to systems issues. The final design of the OMLDM kernel required several iterations and is the result of the evaluation of several alternatives. We expect that some of our design choices may need to be revisited as the Flink platform evolves. However, we believe that future change should be much easier due to the strong separation between the kernel layer and the pipeline layer.

The two areas where significant complexity arose in practice were (a) upstream communication and (b) propagation of back-pressure. In our bipartite graph network, we consider two kinds of data traffic. The downstream data flow and downstream data flow. The first one is the data flow with direction from the spokes to the hubs. The opposite holds for upstream. The data within the Apache Flink implementation flow to the spokes, either streaming data or messages. Spokes handle the necessary computations and forward messages either to the user (outside the Flink job) or to the hubs. Unfortunately, the hubs can only propel their messages outside the job. They cannot transmit data messages backward. For that reason, due to the acyclic nature of the framework, upstream data flow (from the hubs to the spokes) is not supported.

For most streaming frameworks, "upstream" data traffic is not handled as conveniently as downstream; acyclicity affords significant opportunities for optimization and fault resilience. Apache Flink sup-

ports upstream traffic by a mechanism called *iteration*. Unfortunately, the versions of Flink available when OMLDM development started, imposed several restrictions on iterations. To work around them, we used Kafka to provide upstream communication. The hubs provide their messages to a Kafka topic, a classic sink for a Flink job. We then connect the other end of this Kafka topic to the input of the Flink job, creating in this way a feedback loop, a direct way for the data to flow from the hubs to the spokes. Surprisingly, we discovered that using Kafka slightly improved the performance over the Flink-native solution of iterations.

3.3 Pipelines and ML library

The last and top most layer of our architecture is our own distributed OML library written exclusively in Scala. All the algorithms that you are going to see from this point on, either preprocessors, ML algorimths or distributed learning synchronization methods, are implemented from scratch with the exception of Deep Neural Networks. The computational component of our architecture is a pipeline. Each pipeline executes in a *network*. If k is the number of spokes and m the number of hubs, a pipeline comprises of k independent learners, and an m-node Parameter Server. Before each local learner, a number of preprocessors is prepended in pipeline fashion. Fig. 3.3 depicts a pipeline with 3 local nodes.

A pipeline is instantiated on a network, and is implemented as a pair of Java classes, the spoke class and the hub class. The main pipeline responsibilities are:

- To manage **synchronization** of local learners with the Parameter Server. A number of synchronization strategies are implemented. Each of these strategies is implemented by a different pair of spoke and hub classes.

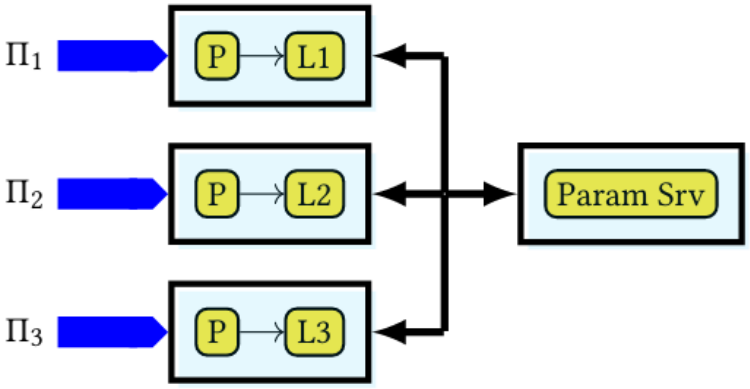

Figure 3.3: A pipeline with 3 learners and a 1-node parameter server. Each learner is preceded by a preprocessor. Black arrows depict communication channels.

- To drive streaming input through the preprocessors to the local learners. The pipeline determines whether to fit or predict on a sample, depending on the presence of a label. For training, input can be aggregated in mini-batches, configurable by the user.
- To maintain statistics per-learner and globally, including learning quality (accuracy, regret, etc).
- To interact with the environment through a special control API, in order to monitor execution, reconfigure learning hyperparameters, etc.

3.3.1 Preprocessors

The incoming stream of samples is piped through a number of preprocessors, before it is seen by the learner. The number and the type of preprocessors that are going to be placed in front of the learner are left for the user to decide. Preprocessors can be concatenated, are aware of the pipeline, and can be created and destroyed dynamically. Also, simple preprocessors can be easily be implemented in a few lines of code. As such, they are a valuable convenience tool for OML

coding and execution. Our current implementation of the distributed OML library contains the following preprocessors implemented from scratch.

- Feature filtering/selection.
- MinMax scaler.
- Standard scaler.
- Running mean and variance.
- Polynomial features.

3.3.2 Synchronization strategies

The synchronization strategy is the heart of the *distributed* pipeline logic, and a crucial factor to learning quality and performance as it can "make or brake" a distributed learner. The current implementation supports the following synchronization strategies:

- The **Total Asynchronous Parallel** (TAP) synchronization protocol, a method that maximizes throughput but can potentially tamper the convergence of the learner.
- The **Bulk Synchronous Parallel** (BSP) synchronization protocol, a method that optimizes convergence but can potentially limit the throughput of the streaming topology due to excessive use of communication.
- The **Stale Synchronous Parallel** (SSP) synchronization protocol that is a compromise between BSP and TAP and tries to bring the best of both worlds, smooth convergence and throughput.
- The **Elastic Averaging** (EA) synchronization protocol, a strategy that offers better exploration of the model "landscape".

- Last but not least, our own synchronization protocol, the **Functional Dynamic Averaging** (FDA) protocol. The main idea of this protocol is to minimize communication between the nodes, hence maximize throughput, and maximize model performance at the same time by performing model synchronization only when necessary instead of periodically sending the model over the network. Our method is discussed in detail in the next section of our study.

3.3.3 Learning algorithms

For the implementations of the OML algorithms we used the Breeze library. Breeze is a library for numerical processing written in Scala that aims to be generic, clean, and powerful without sacrificing efficiency. All the necessary linear algebra computations are done using Breeze. A number of learner algorithms have been implemented on the OML pipeline.

- Passive-Aggressive (PA) learners [Cra+06] for binary and multiclass classification.
- Passive-Aggressive (PA) [Cra+06] regressor.
- Online Support Vector Machine (SVM) classifier.
- Online Ridge Regression.
- KMeans++ clustering.
- The CVFDT [HSD01] algorithm for constructing Hoeffding trees on both discrete and real valued attributes.
- Neural networks algorithms contained in the `DeepLearning4J` library [Tea16].

These learners—except CVFDT—can be parallelized under any of the supported synchronization strategies. Naturally, some combinations may not work as well as others, but OMLDM does not impose the decision on the user. Finally, the library provides a set of interfaces so that users can create their own OML algorithm.

Scala traits are used to share interfaces and fields between classes. They are similar to Java 8's interfaces. Classes and objects can extend traits, but traits cannot be instantiated and therefore have no parameters. In our implementation, the trait that the user needs to implement in order to develop its own OML algorithm is the Learn.scala trait. This trait contains several mandatory methods that need to be overwritten by the learner implementation of the user. Methods like these include are functions for fitting and calculating the loss the model on the provided data, making predictions, scoring the model, setting and updating hyperparameters and last by not least serializing and deserializing the model. The serialization and deserialization of the model and its parameters is an essential part for the performance of the module. It is necessary for the user to be able to convert its model to a POJO object (as they can be easily serialized by many tools) and convert it back to its own class (effectively deserializing it).

Sending the parameters over the network can be a very cost procedure in cases where the actual model is too big in size. Take for example a deep neural network with one million features. The OML library supports techniques for dividing the model into smaller pieces so it can be sent in reasonably sized serialized messages. The model then reconstructs itself upon arrival. The option to send the model in sparse vectors is also supported. This "marshaling" (deconstruction, sparification and serialization) before sending the model over the network and "unmarshaling" upon receiving it is handled behind the scenes by our library in an efficient and optimized way.

3.4 A holistic view of the OMLDM component

The composition of the three main layers gives rise to OMLDM, a Big Data component intended for online learning and prediction. OMLDM provides the capability to train more than one OML in a distributed fashion, each one using its own synchronization method. It is up to the user to decide which synchronization protocol suits best for each distributed model. In its current form, the component can run locally or in a cluster of computers. As soon as it is up and running, the user can interact with it through a provided API written in JSON format.

The API gives the user the flexibility to create, delete, modify and query OML algorithms on OMLDM. To be precise, the programmer is given the following option via the API.

- Query a general summary of all the OML algorithms that run on the module.
- Obtain scores, loss rates, the model itself, and other valuable information about each OML algorithm.
- Create new OML algorithms to train and use for prediction.
- Delete existing OML algorithms that already run on the component.
- Change/modify the hyperparameters of an algorithm that already runs on the module and fine-tune it in real-time.

The requests made to the component take the form of a dictionary, with key-value pairs. There are 5 different main keys.

- **id**: An non-negative integer used as unique identifier for an OML pipeline. All requests must contain an id so that the component knows in which OML pipeline the request is referring to.

- **request**: A string name referring to the request type. There are four request types. "Create", "Update", "Delete" and "Query". The "Query" request type is used when the user needs additional information about the running OML pipeline. As it stands now, all the information about the model is returned to the user (number of fitted data, score, accumulated and mean loss) during a "Query", along with the model itself in a serialized format.
- **preProcessors**: A list of dictionaries describing all the pre-processors the user wants to concatenate on its OML pipeline in case of a "Create" or "Update" request.
- **Learner**: A dictionary describing the OML algorithm the user wants to instantiate in case of a "Create" or "Update" query.
- **trainingConfiguration**: A dictionary containing all the information needed about the distributed synchronization method for training the OML pipeline. It is used in cases of a "Create" and "Update" request.

Each pre-processor and learner dictionary is comprised of the following keys.

- **name**: The string name of the pre-processing or the OML method. This field cannot be null as it is used to recognize and instantiate an transforming or a learning object.
- **hyperParameters**: A dictionary containing the hyperparameters of the transforming or the learning object. Any field, or even the whole key, could be left null. In this case the default hyperparameters of the object are used.
- **parameters**: A dictionary containing the parameters of the transforming or the learning object. Any field, or even the whole

key, could be left null. In this case the default parameters of the object are used.

Lastly, the "trainingConfiguration" key contains the following keys.

- **protocol**: The string name of the synchronization method to be used for distributed training.
- **hubParallelism**: A positive integer indicating the number of distributed parameter servers to be used. The number of workers is determined by the parallelization of the job.
- **miniBatchSize**: The size of the mini batch. This is the smallest number of data points that the learner can use to train itself.
- **miniBatches**: The number of mini batches the learner fits before each synchronization mechanism is triggered.
- **safeZone**: The string name of the safezone function used in the FDA method. Only used when protocol takes the value "FDA". More on FDA synchronization in the next section.
- **threshold**: The threshold parameter of the FDA method. Only used when protocol takes the value "FDA". More on FDA synchronization in the next section.
- **precision**: The precision parameter of the FDA method. Only used when protocol takes the value "FDA". More on FDA synchronization in the next section.

By combining the above key-value pairs the user has a variety of actions regarding the creation, modification, retrieval and deletion of any OML algorithm running on the component.

We provide OMLDM with the means of communication by using Apache Kafka. The input Kafka topics of OMLDM are:

- **Training Stream**: The Kafka stream contains the data used for training. The data points of this channel can be labeled or not, depending on the nature of the ML problem, supervised or unsupervised.
- **Prediction Stream**: The Kafka unlabeled data stream used by the already running OML algorithms within the component to make predictions on them.
- **Requests Stream**: A Kafka topic containing user requests for creating, updating, querying, and deleting learning pipelines. Messages that go through this channel must be provided in JSON format.
- **PS messages**: The messages of the hubs towards the spokes. This topic constitutes the feedback loop topic for the upstream communication.

The output Kafka topics of OMLDM are:

- **Predictions**: This Kafka topic contains data points from the prediction input stream along with their predicted label.
- **Responses**: The responses to users' queries.
- **PS messages**: The messages of the hubs to the spokes. This topic is the feedback loop topic for upstream communication.

Fig 3.4 depicts our Flink/Kafka implementation of OMLDM. The data points from the *Training Stream* and the *Prediction Stream* are parsed, transformed into training and prediction vectors, and united into a single stream. Similarly, the *Requests Stream* and the *PS Messages* are parsed into message objects and united into a single stream. This ends the parsing phase of the implementation. Then, vectors and messages end up together to the spokes, a *CoProcessFunction* in

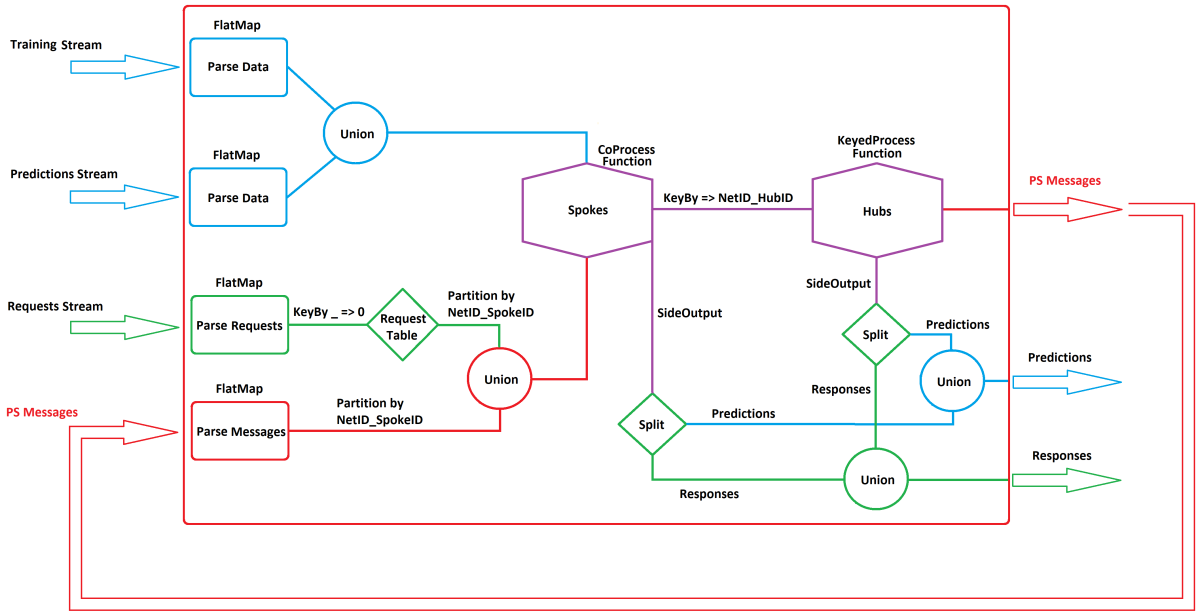

Figure 3.4: OMLDM Flink Implementation.

Apache Flink. Inside the CoProcessFunction the spokes ingest the two streams and execute their functionality. If necessary, spokes send messages to the hubs written inside a *KeydProcessFunction* in Flink. Messages to the hubs are hashed by hub key. This is the most costly redistribution of data that happens within the module as more often than not those messages are model parameters. The hubs send their messages to a Kafka sink that is also our feedback loop. Both Spokes and Hubs emits side output streams containing predictions to the Prediction Stream, and responses to the Requests Stream.

4 Functional Dynamic Averaging

Most of the literature on distributed ML studies the batch setting, where all training data is available to the system at the start of the computation. A common, mostly unstated assumption, is that computational resources are exclusively used by the ML training system; therefore, the goal of synchronization strategies has been to improve the trade-off of computation time vs. learning quality.

Furthermore, much of the work has been done in the context of Deep Learning, which in turn implies stochastic gradient-based optimization techniques, notably SGD. Combined with the emergence of GPUs, the mini-batch approach has dominated extreme-scale Machine Learning in the literature.

Functional Dynamic Averaging (FDA) is motivated by cases where the distributed OML computation is long-running, with the arrival of training data fluctuating over time, computational resources (esp. network bandwidth) are shared with other jobs, and the ML techniques may not be gradient-based (for example, Passive-Aggressive learning).

4.0.1 Intuition

The motivation of FDA came from previous work on Geometric Monitoring of distributed data streams [GKS13; SG; SSK07], and the promising results of Kamp et al. [Kam+14; Kam+16].

In FDA, the assumption made is that the (generally unknown) mean model $\overline{\mathbf{w}}_t = \frac{1}{k}\sum_{i=1}^{k} \mathbf{w}_t^{(i)}$ is a "good" model, provided that $\mathbf{w}_t^{(i)}$ are not too far apart. This is a standard assumption in Machine Learning,

based on long empirical evidence in a plethora of cases. Synchronization methods, including BSP, TAP, and SSP, track this average explicitly, by periodic synchronization to the PS, updating $\tilde{\mathbf{w}}_t$ to track $\overline{\mathbf{w}}_t$ at each round. The frequency of synchronization is determined rather arbitrarily by the mini-batch size of the training. Larger mini-batches reduce communication, but possibly at the expense of the local learners drifting too far apart, and their average not being "good" anymore.

Succinctly, FDA works in rounds, similar to BSP. At the beginning of a round, say at time t_0, the model $\tilde{\mathbf{w}}_{t_0}$ is pushed to local learners. Then, local models are trained independently for some time, processing local streams of training samples, while the nodes cooperatively monitor the following Round Termination Condition (**RTC**):

$$\frac{1}{k}\sum_{i=1}^{k} \|\mathbf{w}_t^{(i)} - \overline{\mathbf{w}}_t\|^2 \leq \Theta, \tag{4.1}$$

where the left-hand side is the **model variance**, and threshold Θ is a hyperparameter of the FDA, defined at the beginning of the round; it may change at each round. When the monitoring logic cannot guarantee the validity of RTC, the round terminates. All local models are pulled into the Parameter Server, and $\tilde{\mathbf{w}}$ is set to their average. Then, another round begins.

4.0.2 Theoretical properties of FDA

We present some formal results on the effect of the RTC on prediction and training performance. For this, we consider the *risk* function, i.e. the expected loss, defined as $R(\mathbf{w}) = E_{\mathbf{Z}}[\ell(\mathbf{w}, \mathbf{Z})]$ where $\mathbf{Z}$ is a random training sample, drawn from the same distribution as the training data. We derive analytical results for the case where R is convex, and L-smooth in some large enough neighborhood of $\overline{\mathbf{w}}_t$, i.e.,

for any two models $\mathbf{w}, \mathbf{w}'$, it is

$$R(\mathbf{w}') \leq R(\mathbf{w}) + \nabla R(\mathbf{w}) \cdot (\mathbf{w}' - \mathbf{w}) + \frac{L}{2}\|\mathbf{w}' - \mathbf{w}\|^2. \tag{4.2}$$

Bounding the expected prediction risk

In our distributed OML setting, an unlabelled item is sent to one of the local learners to have a label predicted. At node i and time t, $R(\mathbf{w}_t^{(i)})$ is the expected loss incurred by labeling a random unlabeled item using model $\mathbf{w}_t^{(i)}$. Since an unlabelled sample is equally likely to be sent to any learner i for labeling, on expectation the loss of the ensemble of local models in labeling a random item is $\overline{R}_t = \frac{1}{k}\sum_{i=1}^{k} R(\mathbf{w}_t^{(i)})$.

The RTC can be used to bound $\overline{R}_t$. By convexity, it is $\overline{R}_t \geq R(\overline{\mathbf{w}}_t)$. In this sense, the mean model is "better" than the ensemble of local models. By plugging $\mathbf{w}_t^{(i)}$ into $\mathbf{w}'$ and $\overline{\mathbf{w}}_t$ into $\mathbf{w}$ in the above formula, summing all inequalities and taking the average, we conclude that

$$R(\overline{\mathbf{w}}_t) \leq \overline{R}_t \leq R(\overline{\mathbf{w}}_t) + \frac{L\Theta}{2}.$$

Convergence effect of FDA

We now examine the effect of the RTC on the training. For simplicity, we shall assume a scenario where learning is performed by SGD.

Consider some time t during a round, where all learners perform an SGD iteration. At learner i, the step taken is

$$\mathbf{w}_{t+1}^{(i)} = \mathbf{w}_t^{(i)} - \eta \nabla \ell(\mathbf{w}_t^{(i)}, \mathbf{z}_t^{(i)}).$$

Therefore, the update to the mean model has the form

$$\overline{\mathbf{w}}_{t+1} = \overline{\mathbf{w}}_t - \eta \frac{1}{k} \sum_{i=1}^{k} \nabla \ell(\mathbf{w}_t^{(i)}, \mathbf{z}_t^{(i)}). \tag{4.3}$$

Now, comparing Eq. 4.3 to a hypothetical (non-stochastic) gradient descent step on the risk at $\overline{\mathbf{w}}_t$:

$$\overline{\mathbf{w}}_{t+1} = \overline{\mathbf{w}}_t - \eta \nabla R(\overline{\mathbf{w}}_t). \tag{4.4}$$

we get to contrast the actual update $\mathbf{G}_t = \frac{1}{k}\sum_{i=1}^{k} \nabla \ell(\mathbf{w}_t^{(i)}, \mathbf{z}_t^{(i)})$ with the ideal update $\nabla R(\overline{\mathbf{w}}_t)$.

Local updates $\mathbf{z}_t^{(i)}$ are i.i.d. random variables. By linearity and the definition of risk, we have

$$E[\nabla \ell(\mathbf{w}_t^{(i)}, \mathbf{z}_t^{(i)})] = \nabla E[\ell(\mathbf{w}_t^{(i)}, \mathbf{z}_t^{(i)})] = \nabla R(\mathbf{w}_t^{(i)}),$$

and we can write

$$\nabla \ell(\mathbf{w}_t^{(i)}, \mathbf{z}_t^{(i)}) = \nabla R(\mathbf{w}_t^{(i)}) + \mathbf{E}_t^{(i)},$$

where $\mathbf{E}_t^{(i)}$ are "noise vectors" of 0-mean, and independent to each other (though not identically distributed). It is now seen that the actual update of $\overline{\mathbf{w}}_t$ is

$$\mathbf{G}_t = \nabla R(\overline{\mathbf{w}}_t) + \mathbf{B}_t + \overline{\mathbf{E}}_t, \tag{4.5}$$

where $\overline{\mathbf{E}}_t = (1/k)\sum_i \mathbf{E}_t^{(i)}$ is again a 0-mean "noise" vector, and

$$\mathbf{B}_t = \frac{1}{k}\sum_{i=1}^{t} \nabla R(\mathbf{w}_t^{(i)}) - \nabla R(\overline{\mathbf{w}}_t)$$

is a *bias vector*.

Bounding the bias

The bias vector depends only on the shape of the risk function R in the neighborhood of $\overline{\mathbf{w}}_t$. Note that the smoothness of R implies a Lipschitz condition on the gradients, i.e., Eq. 4.2 is equivalent to $\|\nabla R(\mathbf{w}') - \nabla R(\mathbf{w})\| \leq L\|\mathbf{w}' - \mathbf{w}\|$. This inequality can be used to

bound the bias vector; applying we get (after manipulations)

$$\|\mathbf{B}_t\| \leq L\sqrt{\Theta}$$

Tighter bound on bias

A probably tighter bound can be obtained by considering the Lipschitz condition on the *Hessian matrix* $\nabla^2 R$ (this is simply the Jacobian of the gradient, i.e., the multidimensional second derivative of R). Indeed, if the Hessian is Λ-Lipschitz, i.e., $\|\nabla^2 R(\mathbf{w}') - \nabla^2 R(\mathbf{w})\| \leq \Lambda\|\mathbf{w}' - \mathbf{w}\|$, then (Lemma 1 in [NP06]),

$$\|\nabla R(\mathbf{w}') - \nabla R(\mathbf{w}) - \nabla^2 R(\mathbf{w}) \cdot (\mathbf{w}' - \mathbf{w})\| \leq \frac{\Lambda}{2}\|\mathbf{w}' - \mathbf{w}\|^2. \tag{4.6}$$

Observe that

$$\mathbf{B}_t = \frac{1}{k}\sum_{i=1}^{k}\left(\nabla R(\mathbf{w}_t^{(i)}) - \nabla R(\overline{\mathbf{w}}_t) - \nabla^2 R(\overline{\mathbf{w}}_t) \cdot (\mathbf{w}_t^{(i)} - \overline{\mathbf{w}}_t)\right).$$

By applying Eq. 4.6 and the triangle inequality of the norm, it follows that the RTC implies the the following bound:

$$\|\mathbf{B}_t\| \leq \frac{\Lambda\Theta}{2}. \tag{4.7}$$

Note that, Λ can be much smaller than L for many risk functions. In particular, if R is a quadratic function, then the Hessian is constant, and therefore $\Lambda = 0$, whereas of course, L is the maximum eigenvalue of the Hessian. Intuitively, Λ measures how closely R can be approximated by a quadratic function, whereas L measures how well R can be approximated by a linear one.

4.0.3 Monitoring the RTC

FDA monitors the RTC by applying techniques from Functional Geometric Monitoring, as described in [SG] and the references therein.

Our first step is to restate the problem of monitoring the RTC, into the standard distributed stream monitoring formulation. This formulation, we define the "local state" $\mathbf{S}_i(t) \in \mathbb{R}^n$ for $i = 1, \ldots, k$. Local state is updated arbitrarily. The "global state" of the system $\mathbf{S}(t) \in \mathbb{R}^n$ is the average of the "local states". The goal is to monitor a threshold condition on the global vector, of the form $F(\mathbf{S}(t)) \leq \Theta$, where $F : \mathbb{R}^n \to \mathbb{R}$ is a non-linear function.

Let $\Delta_t^{(i)} = \mathbf{w}_t^{(i)} - \mathbf{w}_{t_0}^{(i)}$ be the *update* at learner i, that is, the change to the local model at time t since the beginning of the current round at time t_0. Defining the average update, $\overline{\Delta}_t = \frac{1}{k}\sum_{i=1}^{k} \Delta_t^{(i)}$, and noting that at t_0 all local models were equal, it follows that the model variance can be written as

$$\frac{1}{k}\sum_{i=1}^{k} \|\mathbf{w}_t^{(i)} - \overline{\mathbf{w}}_t\|^2 = \frac{1}{k}\sum_{i=1}^{k} \|\Delta_t^{(i)} - \overline{\Delta}_t\|^2 = \left(\frac{1}{k}\sum_{i=1}^{k} \|\Delta_t^{(i)}\|^2\right) - \|\overline{\Delta}_t\|^2 \tag{4.8}$$

So, conceptually, if we define

$$S_i(t) = \begin{bmatrix} \|\Delta_t^{(i)}\|^2 \\ \overline{\Delta}_t^{(i)} \end{bmatrix} \quad \text{and} \quad F\left(\begin{bmatrix} v \\ \mathbf{x} \end{bmatrix}\right) = v - \|\mathbf{x}\|^2, \tag{4.9}$$

the RTC is equivalent to condition $F(S(t)) \leq \Theta$.

4.0.4 Approximately monitoring the RTC

Unfortunately, with the definitions of Eq. 4.9, the FGM will not be able to monitor the RTC with low communication cost; the dimension of state S_i is $\dim \mathcal{W} + 1$. To reduce communication, we must apply

dimensionality reduction on the local updates $\Delta^{(i)}$, and determine a suitable function to monitor, based on the reduction chosen.

However, by reducing the information in the state vectors, monitoring the RTC becomes approximate. There is a trade-off between performance and tightness. We now present three reductions for monitoring the RTC with little communication.

Naive FDA

In the naive approach, we eliminate the update vector from the local state (i.e. reduce the dimension to 0). Define local state as $S_i(t) = \|\Delta_t^{(i)}\|^2 \in \mathbb{R}$ and $F(v) = v$ the identity function. It is trivial to check that condition $F(S(t)) \leq \Theta$ implies the RTC.

Linear FDA

In the linear case, we reduce the update vector to a scalar, $\xi \cdot \Delta_t^{(i)} \in \mathbb{R}$, where ξ is any unit vector.

Define local state to be $\mathrm{S}_i(t) = \left(\|\Delta_t^{(i)}\|^2, \xi \cdot \Delta_t^{(i)}\right) \in \mathbb{R}^2$. Also, define $F(v, x) = v - x^2$. To show that $F(\mathrm{S}(t)) \leq \Theta$ implies the RTC, observe that

$$F(\mathrm{S}(t)) = \left(\frac{1}{k}\sum_{i=1}^{k}\|\Delta_t^{(i)}\|^2\right) - \left(\xi \cdot \overline{\Delta}_t\right)^2 \geq \left(\frac{1}{k}\sum_{i=1}^{k}\|\Delta_t^{(i)}\|^2\right) - \|\overline{\Delta}_t\|^2$$

A random choice of ξ is likely to perform poorly (terminate a round prematurely), as it will likely be close to orthogonal to $\overline{\Delta}_t$. A good choice would be a vector ξ correlated to $\overline{\Delta}_t$. A heuristic choice is to take $\overline{\Delta}_{t_0}$ (after scaling it to norm 1), i.e., the update vector right before the current round started. All nodes can estimate this without communication, as $\tilde{\mathbf{w}}_{t_0} - \tilde{\mathbf{w}}_{t_{-1}}$, the difference of the last two models pushed by the Parameter Server.

Sketch FDA

An optimal estimator for $\|\overline{\Delta}_t\|^2$ can be obtained by employing AMS sketches [CG05]. An AMS sketch of a vector $\mathbf{v} \in \mathbb{R}^M$ is a $m \times d$ real matrix

$$\Xi = sk(\mathbf{v}) = \begin{bmatrix} \Xi_1 & \Xi_2 & \dots & \Xi_d \end{bmatrix},$$

where $d \cdot m \ll M$. Operator $sk(\cdot)$ is linear and can be computed in $O(dM)$ steps.

Let $m = O(1/\varepsilon^2)$ and $d = O(\log \frac{1}{\delta})$. Function

$$\mathcal{M}_2(sk(\mathbf{v})) = median_{i=1,\dots,d} \|\xi_i\|^2$$

is an excellent estimator of the squared norm of $\mathbf{v}$: with probability at least $1 - \delta$, $\mathcal{M}_2(sk(\mathbf{v})) \in (1 \pm \varepsilon)\|\mathbf{v}\|^2$.

Define

$$\mathrm{S}_i(t) = \begin{bmatrix} \|\Delta_t^{(i)}\|^2 \\ sk(\Delta_t^{(i)}) \end{bmatrix} \in \mathbb{R}^{1+d\times m},$$

Also, define the function $F(\begin{bmatrix} v \\ \Xi \end{bmatrix}) = v - \frac{1}{1+\varepsilon}\mathcal{M}_2(\Xi)$. Then, by linearity of the sketch operator, if follows that $F(S(t)) \leq \Theta$ implies the RTC with probability at least $1 - \delta$.

4.0.5 Functional Geometric Monitoring for RTC

All three variants of FDA reduce the RTC to a condition of the form $F(S(t)) \leq \Theta$. To monitor this condition, the local learners coordinate via a coordinating Parameter Server node. A naive strategy could be, for every learner, to ship updates of its local $\mathrm{S}_i(t)$ to the *Coordinator*, which will compute the global state $\mathrm{S}(t)$ and monitor RTC approximately.

While such simple schemes can arguably work well, especially for Naive and linear FDA, there is also a large body of work on techniques

for decreasing this communication cost by 1 or 2 orders of magnitude. The sequence of local state updates can be thought of as a distributed stream. Instead of collecting this stream to the coordinator, local nodes may process their local streams *in situ*, and only ship updates to a coordinator when needed, reducing the overall communication cost.

A state-of-the-art technique for this type of communication reduction is Functional Geometric Monitoring (FGM). The essence of FGM is in reducing a distributed monitoring problem to a simpler one. To monitor a condition $F(S(t)) \leq \Theta$, starting at time t_0, construct another condition, $\phi(S(t)) \leq 0$, such that the second implies the first, and ϕ is a convex function, called a *safe function*. The safe function can depend on the global state $\vec{S}(t_0)$. Then, the distributed scalar sum condition

$$\frac{1}{k}\sum_{i=1}^{k}\phi(S_i(t)) \leq 0 \tag{4.10}$$

ensures $\phi(S(t)) \leq 0$ by convexity, and therefore $F(S(t)) \leq \Theta$. At the heart of FGM is an efficient distributed protocol for monitoring a distributed scalar sum threshold condition like (4.10)—we refer to [SG] for details.

When, during monitoring, (4.10) fails at time t_f, all local states are sent to the coordinating node. If it is still $F(S(t_f)) \leq \Theta$, the process repeats (with a safe function for $S(t_f)$). Else, the monitoring routine declares a violation and terminates the current round.

Naive FDA

The Naive FDA variant is already expressed as a scalar sum like that of (4.10), by letting $\phi(v) = v - \Theta$. Therefore, Naive FDA is straightforward to handle.

Linear FDA

It is $F(v, x) = v - x^2$. Since F is not convex, construction of a safe function is required. Let (v_0, x_0) be the current state; it must be $F(v_0, x_0) < \Theta$. Referring to Fig. 4.1, let (p, q) be the projection of the

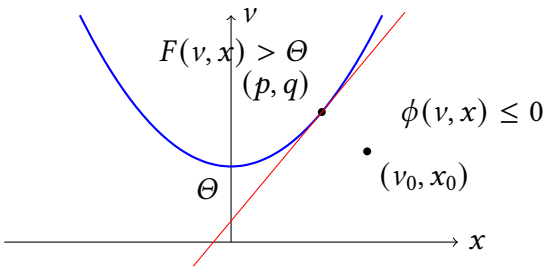

Figure 4.1: Construction of a safe function for global state (v_0, x_0).

current state on the convex set $\overline{A} = \{(v, x) \in \mathbb{R}^2 \mid v - x^2 > \Theta\}$. The tangent halfspace to $\overline{A}$ at (p, q) is defined by linear function

$$\phi(v, x) = v - p + 2q(q - x).$$

Noticing that $\phi(v, x) \leq 0 \implies (v, x) \notin \overline{A} \iff F(v, x) \leq \Theta$, completes the construction.

Sketch FDA

The construction of a safe function is significantly more involved. We apply the compositional technique of [GS17]. A detailed justification of our construction appears therein. Here we give a brief overview of the construction, for completeness.

By properties of the median, condition $F(v, \Xi) \leq \Theta$ is equivalent to

$$median_{i=1,\ldots,d}\left(v - \frac{1}{1+\varepsilon}\|\xi_i\|^2\right) \leq \Theta.$$

To compose a safe function at some global state (v_0, Ξ_0), we first

look at a safe function for conditions of the form

$$v - \frac{1}{1+\varepsilon}\|\xi_i\|^2 \leq \Theta. \tag{4.11}$$

Let $I \subseteq \{1, 2, \ldots, d\}$ be the set of indices where (4.11) holds. Since $F(v_0, \Xi_0) \leq \Theta$, it is $|I| \geq d/2$.

Fix some $i \in I$. The safe function of Eq. 4.11 can be treated similar to the case for linear FDA (refer again to Fig. 4.1). Again, the current global state $(v_0, \xi_{0,i})$ must be projected on a paraboloid and a tangent hyperplane at the projection define function $\phi_i(v, \xi_i)$. Finally, the overall safe function ϕ is a weighted max-k-sum of ϕ_i.

5 Experimental Evaluation

In this section, we present experiments to validate the main contributions presented. We have focused on scalability of the architecture and the effect of synchronization.

5.1 Experimental setup

Our first learning task involves training a deep Convolutional Neural Network on AMNIST, a dataset derived from the well-known MNIST dataset. This is a dataset of 2,000,000 items, constructed from the standard MNIST samples by adding random distortion, shear, rotation and skew (we used https://augmentor.readthedocs.io/en/master/index.html for this augmentation). The AMNIST dataset was stored as text, occupying 6.7 GBytes on disk. This stream was used to train online a deep CNN, implemented by `DeepLearning4J`. The network contained 14 layers, with a total of about 100k parameters. The network was trained locally by SGD using ADAM updates, with L2 regularization.

For our second experiment, we created a synthetic classification dataset using the facilities of the `sklearn` Python library. The dataset contained 10,000,000 samples, of 43 features each, of which 4 features were pure noise. Each class was represented by 2 clusters. The dataset occupied 8 GBytes on disk. We trained a Passive-Aggressive kernel classifier, with a polynomial kernel of degree 2. The PA-I variant, while aggressiveness was set to $C = 0.01$ (refer to [Cra+06] for the relevant definitions).

5.1.1 Experiment settings

For our experiment we used an existing Flink/Kafka installation, on a cluster of 14 machines, each equipped with a 12-core Intel Xeon CPU and 32 GBytes of RAM. Resources on this cluster were managed by YARN. In the Flink setup, we limited the amount of work per node, by allocating a maximum of 3 Flink tasks (TaskManagers) per cluster machine, while the Kafka setup ran on 4 nodes.

Each training task was executed in a distributed pipeline, under all synchronization strategies, with the exception of EA for the PA learner, resulting in 9 distributed scenaria. In addition, the threshold Θ for the FDA strategy was set to 8 for the DL task and 0.008 for the PA task.

Each of these scenarios was executed on an OMLDM network with parallelism 2, 4, 8, 16 and 32 (as determined by the number of spokes in the OMLDM network). In all cases, there was a single hub node. The pipeline was configured with a mini-batch size of 256 training samples in every case.

5.2 Results

5.2.1 Quality of learning

We monitored various metrics of the learning quality in our executed scenarios. In general, the quality of learning was comparable under all synchronization strategies and parallelism. Fig. 5.1 depicts the average learning accuracy of classification. In the DL tasks, a trend of slow decrease in the accuracy is exhibited by every method, with SSP and BSP being the marginal winners. In the PA tasks, the differences were even smaller.

Fig. 5.2 depicts the average loss during training the deep CNN on AMNIST dataset for all the synchronization methods and for 16

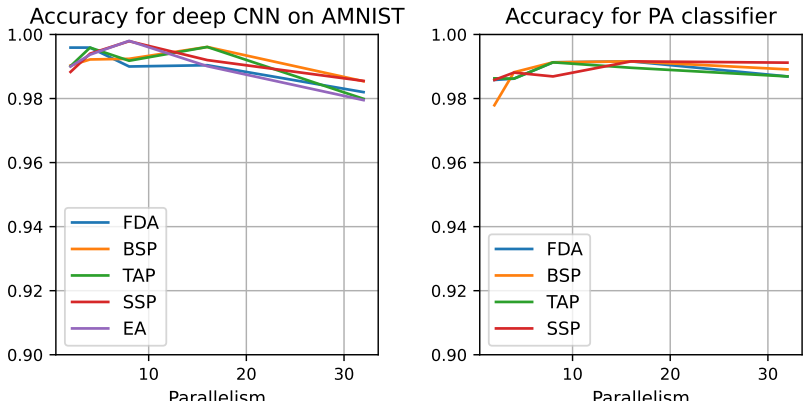

Figure 5.1: Mean classification accuracy.

workers. It is evident form the graphs that FDA catches up with the other methods after the first 100000 examples. Nevertheless, this

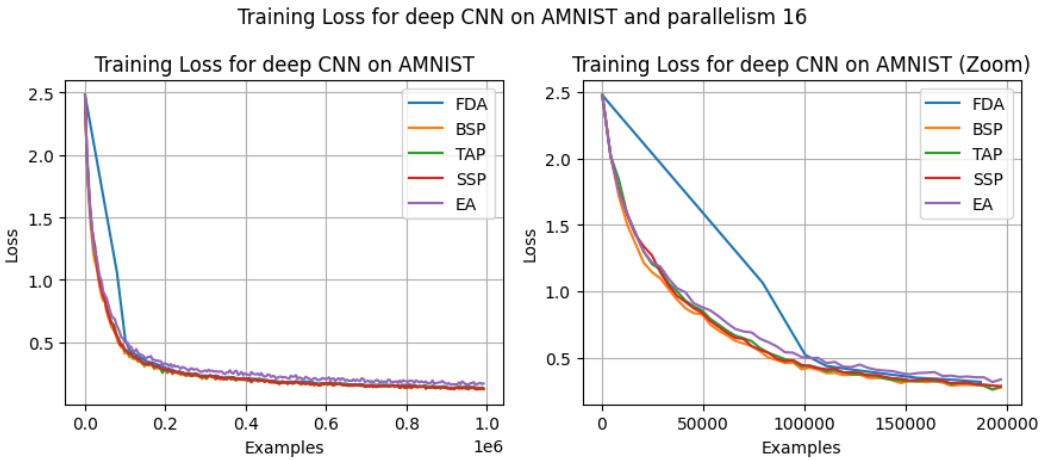

Figure 5.2: Training loss for deep CNN on AMNIST.

makes sense as FDA makes much fewer synchronizations compared to the periodic averaging of the other distributed training methods. More specifically, in this specific example FDA makes 2 synchronizations up until the 100000 training examples, were the other methods made

approximately 25. Similar results can be observed in Fig. 5.3 for the PA classifier.

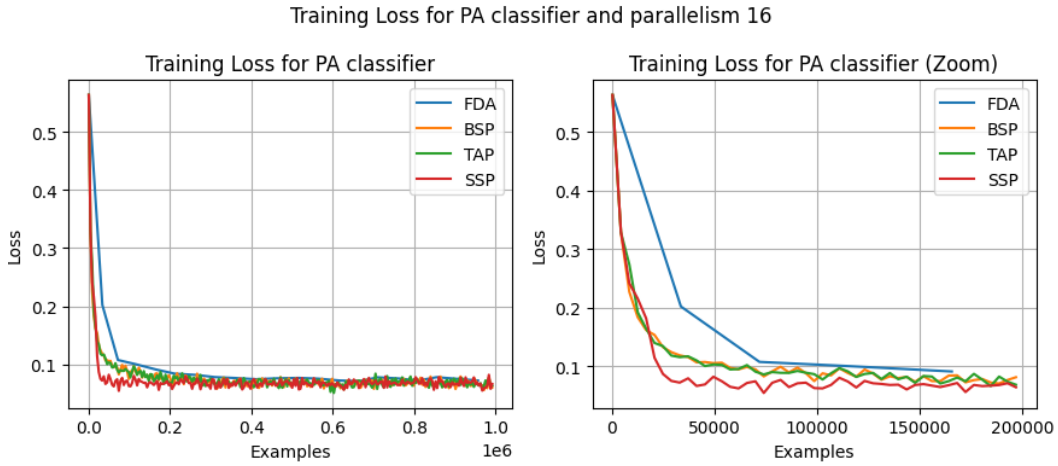

Figure 5.3: Training loss for deep CNN on AMNIST.

Lastly, in Fig. 5.4 we can observe the training loss for the CNN and PA classifiers. All the experiments where conducted for classifiers of parallelism 16 and for various thresholds Θ of the FDA method. Each type of classifier demanded a very different value range for the threshold Θ due to their different structures, training losses and number of trainable parameters. We observed that in both cases, the smaller the threshold Θ the steeper the decline of the training loss. This is understandable and can be explained from the fact that as Θ declines, the safe zone becomes smaller and hence the easier it is for the global variance to exceed the threshold and trigger a synchronization. As such, synchronizations become more frequent driving the training loss down more quickly.

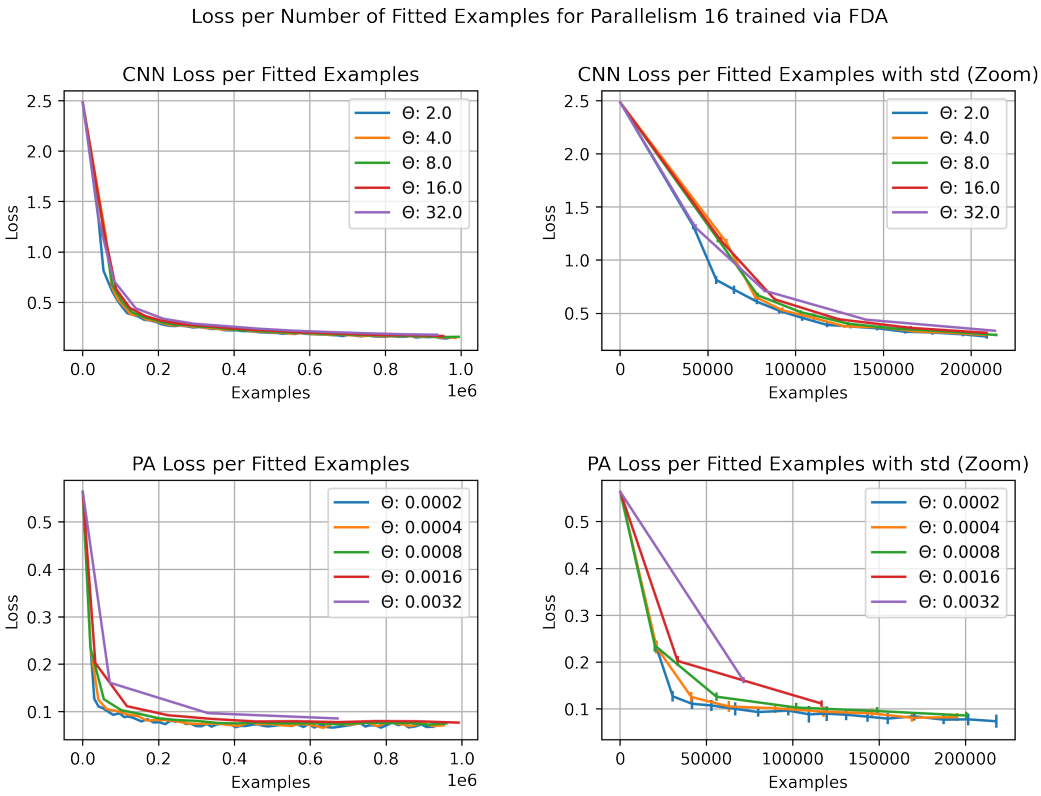

Figure 5.4: Training Loss for various FDA Thresholds.

5.2.2 Scalability

Our next set of results concerns the scalability of the distributed computation. We executed each learning task in a centralized pipeline (implemented in OMLDM, but without a Parameter Server). The results shown in Fig. 5.5 validate our expectations based on previous work, and confirm our main claims.

A general observation, we should note that the two methods have different processing and communication costs, with DL being heavier computationally, whereas PA is bounded by the total communication.

As shown, in all cases FDA dominated in performance over all other methods. In the DL task, the asynchronous method (TAP) performed

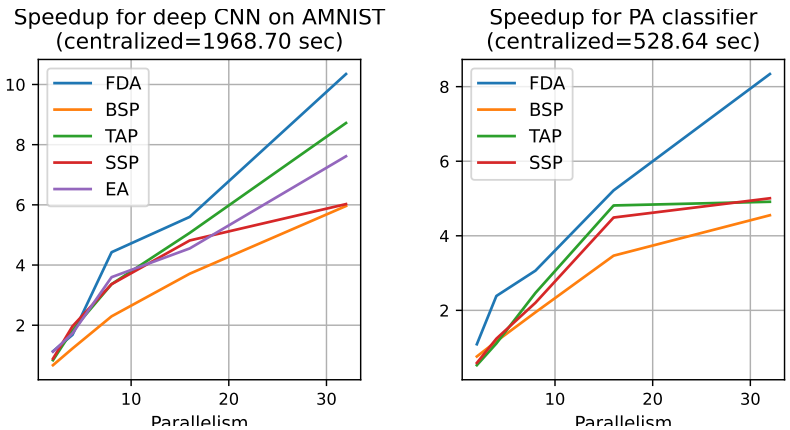

Figure 5.5: Speedup for the DL task (left) and the PA task (right). On top of each plot are the times of centralized execution, over which speedup is computed.

comparably, implying that the communication cost is only a small part of total time. In the PA task however, all strategies except FDA taper off after a parallelism of 16. In all cases, the synchronous BSP method has the worst speedup, reflecting the decreased CPU utilization caused by aggressive synchronization.

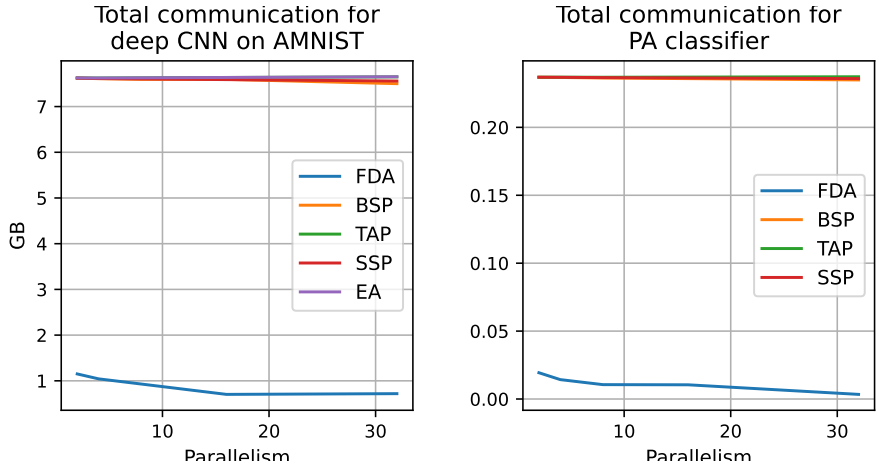

Figure 5.6: Total communication in GBytes for each execution.

The explanation of FDA's superiority is in Fig. 5.6, showing the

total amount of communication between OMLDM nodes. Observe that BSP, TAP and EA carry out the same amount of communication (proportional to the total number of mini-batches and the size of the model). There is slightly smaller communication in the case of SSP, reflecting a small number of stale messages.

On the other hand, FDA performs almost an order of magnitude less communication, owning to the method avoiding synchronization when local models are sufficiently similar. A counter-intuitive observation is the decrease of total communication as parallelism grows. Our theoretical analysis does not explain this, however it was consistently observed in a number of experiments.

6 Conclusions & Outlook

In this study we presented OMLDM. A novel state-of-the-art distributed architecture for effortlessly deploying OML pipelines on streaming platforms. Our module was designed as a part of the INFORE project, a holistic approach in streaming settings aiming to provide cross-platform streaming analytics at scale to non-expert programmers. The module maintains and updates data models while providing interactive data analytics capabilities along with online predictions on unlabeled data. Users can interact with the module via a JSON API and pose queries, create, delete and actively update data models.

OMLDM supports distributed online learning by utilizing the Parameter Server paradigm. To support maximum scalability, we have implemented a variety of distributed training techniques within OMLDM to train the models. Because traditional synchronization strategies tend to not scale well due to the excessive communication between the learners and the parameter server as parallelism increases, we brought forward and evaluated a novel synchronization strategy, Functional Dynamic Averaging (FDA), that minimizes the prediction loss and network communication all at once. Our technique utilizes distributed stream monitoring methods. We proved experimentally that FDA achieves high predictive performance, yet requires almost an order of magnitude less communication than any other contemporary static synchronization protocol, a fact that gives it the best speedup results.

Last but not least, the architecture of OMLDM is composed of three layers, the Online Machine Learning library, the Network/Middle-

ware(N/M) layer, and the kernel or back-end layer. This design choice makes it easy for users to extend the module with their Online Machine Learning models and synchronization techniques with a few lines of code. Many state-of-the-art online algorithms are yet to be implemented in the library such as online random forests and other bagging methods. What is more, the OML library provides a battleground for designing and testing new synchronization protocols, line a more sophisticated FDA with rebalancing abilities. The Network/Middleware layer can also be seen as a project of its own in the distributed systems domain. Implementation of stronger forms of consistency, including causal and sequential, could be a valuable addition to the architecture. Another pleasant aftereffect of this architecture is the ability to change the lower-level streaming kernel with your streaming platform of choice. In future studies, we aim to implement the low-level streaming kernel by using a variety of streaming frameworks like Apache Spark and Akka. This will prove that our architecture is modular and will provide performance comparisons against the distributed frameworks

Bibliography

[BPS09] Shai Ben-David, D. Pál, and S. Shalev-Shwartz. **Agnostic Online Learning**. In: *COLT*. 2009 (see page 6).

[CG05] Graham Cormode and Minos Garofalakis. **"Sketching Streams Through the Net: Distributed Approximate Query Tracking"**. In: *Proc. of the 31st Intl. Conference on Very Large Data Bases*. Trondheim, Norway, Sept. 2005 (see page 48).

[CMY08] Graham Cormode, S. Muthukrishnan, and Ke Yi. **Algorithms for distributed functional monitoring**. In: *SODA*. 2008 (see page 4).

[Cra+06] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. **Online Passive-Aggressive Algorithms**. en. 7 (2006), 551–585. ISSN: 1532-4435 (see pages 33, 53).

[GKS13] Minos N. Garofalakis, Daniel Keren, and Vasilis Samoladas. **Sketch-based Geometric Monitoring of Distributed Stream Queries**. *PVLDB* (2013) (see page 41).

[GS17] Minos N. Garofalakis and Vasilis Samoladas. **Distributed Query Monitoring through Convex Analysis: Towards Composable Safe Zones**. In: *ICDT*. 2017. DOI: 10.4230/LIPIcs.ICDT.2017.14 (see page 50).

[Ho+13a] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. **More effective distributed ML via a Stale Synchronous Parallel parameter server**. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., Dec. 2013, 1223–1231. (Visited on 07/02/2021) (see page 11).

[Ho+13b] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. **More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server**. In: *Advances in Neural Information Processing Systems*. Ed. by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger. Vol. 26. Curran Associates, Inc., 2013. URL: https://

proceedings.neurips.cc/paper/2013/file/b7bb35b9c6ca2aee2df08cf09d7016c2-Paper.pdf (see page 11).

[HSD01] Geoff Hulten, Laurie Spencer, and Pedro M. Domingos. **Mining time-changing data streams**. In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26-29, 2001*. Ed. by Doheon Lee, Mario Schkolnick, Foster J. Provost, and Ramakrishnan Srikant. ACM, 2001, 97–106. DOI: 10.1145/502512.502529. URL: https://doi.org/10.1145/502512.502529 (see page 33).

[Jia+17] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. **Heterogeneity-aware Distributed Parameter Servers**. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. New York, NY, USA: ACM, 2017, 463–478. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3035933. URL: http://doi.acm.org/10.1145/3035918.3035933 (visited on 10/26/2018) (see page 7).

[Kam+14] Michael Kamp, Mario Boley, Daniel Keren, Assaf Schuster, and Izchak Sharfman. **Communication-efficient Distributed Online Prediction by Dynamic Model Synchronization**. In: *Proceedings of the 2014th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I*. ECMLPKDD'14. Berlin, Heidelberg: Springer-Verlag, 2014, 623–639. ISBN: 978-3-662-44847-2. DOI: 10.1007/978-3-662-44848-9_40. URL: https://doi.org/10.1007/978-3-662-44848-9_40 (visited on 10/26/2018) (see pages 4, 41).

[Kam+16] Michael Kamp, Sebastian Bothe, Mario Boley, and Michael Mock. **Communication-Efficient Distributed Online Learning with Kernels**. en. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken. Lecture Notes in Computer Science. Springer International Publishing, 2016, 805–819. ISBN: 978-3-319-46227-1 (see pages 4, 41).

[KDB19] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Albert Bifet. "Large-Scale Learning from Data Streams with Apache SAMOA." en. In: *Learning from Data Streams in Evolving Environments: Methods and Applications*. Ed. by Moamar Sayed-Mouchaweh. Studies in Big Data. Cham: Springer International Publishing, 2019, 177–207. ISBN: 978-3-319-89803-2. DOI: 10.1007/978-3-319-89803-2_8. URL: https://doi.org/10.1007/978-3-319-89803-2_8 (visited on 04/06/2020) (see page 3).

[Li+14] Mu Li, David G. Andersen, Alexander Smola, and Kai Yu. **Communication Efficient Distributed Machine Learning with the Parameter Server**. In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, 19–27. URL: http://dl.acm.org/citation.cfm?id=2968826.2968829 (visited on 10/26/2018) (see page 7).

[NP06] Yurii E. Nesterov and Boris T. Polyak. **Cubic regularization of Newton method and its global performance**. *Math. Program.* 108:1 (2006), 177–205. DOI: 10.1007/s10107-006-0706-8. URL: https://doi.org/10.1007/s10107-006-0706-8 (see page 45).

[SC15] Hang Su and Haoyu Chen. **Experiments on Parallel Training of Deep Neural Network using Model Averaging**. *ArXiv* abs/1507.01239 (2015) (see page 10).

[SG] Vasilis Samoladas and Minos Garofalakis. **Functional Geometric Monitoring for Distributed Streams**. In: *EDBT2019*. Lisbon, Portugal (see pages 4, 41, 46, 49).

[SN10] Alexander Smola and Shravan Narayanamurthy. **An architecture for parallel topic models**. *Proceedings of the VLDB Endowment* 3:1-2 (Sept. 2010), 703–710. ISSN: 2150-8097. DOI: 10.14778/1920841.1920931. URL: https://doi.org/10.14778/1920841.1920931 (visited on 07/01/2021) (see page 7).

[SSK07] Izchak Sharfman, Assaf Schuster, and Daniel Keren. **"A geometric approach to monitoring threshold functions over distributed data streams"**. *ACM Trans. Database Syst.* 32:4 (2007) (see page 41).

[Tea16] Eclipse Deeplearning4j Development Team. **DeepLearning4J: Open-source distributed deep learning for the JVM** (2016). URL: http://deeplearning4j.org (see page 33).

[Ver+20] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. **A Survey on Distributed Machine Learning**. *ACM Comput. Surv.* 53:2 (Mar. 2020). ISSN: 0360-0300. DOI: 10.1145/3377454. URL: https://doi.org/10.1145/3377454 (see page 7).

[ZCL15] Sixin Zhang, Anna Choromanska, and Yann LeCun. **Deep learning with elastic averaging SGD**. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15.

Cambridge, MA, USA: MIT Press, Dec. 2015, 685–693. (Visited on 07/02/2021) (see page 12).

[Zha+16] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. *Staleness-aware Async-SGD for Distributed Deep Learning*. 2016. arXiv: 1511.05950 [cs.LG] (see page 11).