TensAIR: Real-Time Training of Neural Networks from Data-streams

Mauro D. L. Tosi
mauro.dalleluccatosi@uni.lu
University of Luxembourg
Luxembourg

Vinu E. Venugopal
vinu.ev@iiitb.ac.in
IIIT Bangalore
India

Martin Theobald
martin.theobald@uni.lu
University of Luxembourg
Luxembourg

ABSTRACT

Online learning (OL) from data streams is an emerging area of research that encompasses numerous challenges from stream processing, machine learning, and networking. Stream-processing platforms, such as Apache Kafka and Flink, have basic extensions for the training of Artificial Neural Networks (ANNs) in a stream-processing pipeline. However, these extensions were not designed to train ANNs in real-time, and they suffer from performance and scalability issues when doing so.

This paper presents TensAIR, the first OL system for training ANNs in real time. TensAIR achieves remarkable performance and scalability by using a decentralized and asynchronous architecture to train ANN models (either freshly initialized or pre-trained) via DASGD (decentralized and asynchronous stochastic gradient descent). We empirically demonstrate that TensAIR achieves a *nearly linear scale-out* performance in terms of (1) the number of worker nodes deployed in the network, and (2) the throughput at which the data batches arrive at the dataflow operators. We depict the versatility of TensAIR by investigating both sparse (word embedding) and dense (image classification) use cases, for which TensAIR achieved from 6 to 116 times higher sustainable throughput rates than state-of-the-art systems for training ANN in a stream-processing pipeline.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Distributed artificial intelligence**; • **Computer systems organization** → **Real-time systems**.

KEYWORDS

Online Learning, Neural Networks, Asynchronous Stream Processing

ACM Reference Format:
Mauro D. L. Tosi, Vinu E. Venugopal, and Martin Theobald. 2024. TensAIR: Real-Time Training of Neural Networks from Data-streams. In *2024 The 8th International Conference on Machine Learning and Soft Computing (ICMLSC 2024), January 26–28, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3647750.3647762

ICMLSC 2024, January 26–28, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-1654-6/24/01
https://doi.org/10.1145/3647750.3647762

1 INTRODUCTION

Online learning (OL) is a branch of Machine Learning (ML) which studies solutions to time-sensitive problems that demand real-time answers based on fractions of data received in the form of *data streams* [14]. A common characteristic of data streams is the presence of *concept drifts* [16], i.e., changes in the statistical properties among the incoming data objects over time [23]. Consequently, pre-trained ML models tend to be inadequate in OL scenarios as their performance usually decreases after concept drifts [23]. Differently, OL models mitigate the negative effects of such concept drifts by being ready to instantly update themselves for any new data from the data stream [14].

Due to the intrinsic time-sensitiveness of OL, it is not feasible to depend on solutions that spend an undue amount of time on retraining. Thus, if concept drifts are frequent, currently, complex OL problems cannot rely on robust solutions common to other ML problems due to their long training time [14], like those involving Artificial Neural Networks (ANNs) [12].

Therefore, how to solve complex OL problems like those involving data streams of audio, video, or even text remains an open research question, especially when they are affected by frequent concept drifts. Currently, most OL researchers are focused on how to improve the quality of the input data and how to adapt to concept drifts [6, 15, 23, 27, 31] and they do not address the issue of solving such complex problems. Our intuition is that current hardware is already capable of training a large class of ANN models in real time if the training is distributed across multiple nodes efficiently. In this case, problems that today are deemed too complex to be effectively solved by standard OL learners could be solved using ANNs.

Nowadays, instead of retraining ANNs in real-time, state-of-the-art extensions [2, 3] for Apache Flink [8] and Kafka [19] were developed to adapt/re-train their models using datasets created from buffered samples from the data-stream. Thus, these approaches enable the usage of ANN in OL by giving up the real-time adaptation of the ANN models, which can only be updated after buffering a dataset substantially large to be used for retraining. If adapted to be trained in real-time, those approaches suffer from performance and scalability issues (cf. Section 4). Thus, they cannot sustain throughput high enough for many real-world problems.

Consequently, when real-time adaptation is not available, one can expect a lower prediction/inference performance of models between the instant a *concept drift* occurs and the moment the model is updated. Thus, considering that non-trivial ANN models demand a high amount of training examples before convergence, one can expect low-quality predictions/inferences for an extended amount of time (until the training dataset is buffered and the model

is retrained). This makes it unfeasible to apply this approach to real-world problems that suffer from frequent concept drifts.

To mitigate the prediction/inference performance decrease, we argue that it is necessary to adapt the ANN models in real-time. However, the real-time adaptation of ANN models on an OL scenario is not straightforward. We therefore highlight the following two challenges:

(1) *real-time data management:* Not all training data is available from the beginning. Thus, it is necessary to incrementally update the model with fractions of data at each step. Different from commonly used pre-defined training datasets.
(2) *backpressure:* The model must process a higher number of data samples per second (for training and for inference/prediction) than the data stream produces. This avoids a sudden surge in latency or even a system crash.

In this paper, we present the architecture of TensAIR, the first OL framework for training ANN models (either freshly initialized or pre-trained) in real time. TensAIR leverages the fact that stochastic gradient descent (SGD) is an iterative method that can update a model based only on a fraction of the training data per iteration. Thus, instead of using pre-defined or buffered datasets for training, TensAIR models are updated after each data sample (or data batch) is made available by the data stream. In addition, TensAIR achieves remarkable scale-out performance by using a fully decentralized and asynchronous architecture throughout its whole dataflow, thus leveraging the usage of DASGD (decentralized and asynchronous SGD) to update the ANN models.

To assess TensAIR, we performed experiments on sparse (word embedding) and dense (image classification) models. In our experiments, TensAIR achieved nearly linear scale-out performance in terms of (1) the number of worker nodes deployed in the network, and (2) the throughput at which the data batches arrive at the dataflow operators. Moreover, we observed the same convergence rate in the distributed models independently of the number of worker nodes, which shows that the usage of DASGD did not negatively impact the models' convergence. When compared to the state of the art, TensAIR's sustainable throughput in the real-time OL setting was from 6 to 175 times higher than Apache Kafka extension [3] and from 6 to 120 times higher than Apache Flink extension [2]. We additionally compared TensAIR to Horovod [30], distributed ANN framework developed by Uber, and achieved from 4 to 335 times higher sustainable throughput than them in the same real-time OL setting.

Below, we summarize the main contributions of this paper.

Contributions

(1) Design and implementation of TensAIR, the first framework for real-time training and prediction in ANN models.
(2) Creation and usage of our Decentralized and Asynchronous SGD (DASGD) algorithm.
(3) Experimental evaluation of TensAIR showing almost linear training time speed-up in terms of nodes deployed.
(4) Sustainable throughput comparison between TensAIR and state-of-the-art systems, with TensAIR achieving from 4 to 120 times higher sustainable throughput than the baselines;
(5) Depiction of real-time Sentiment Analysis use case that would not be feasible with standard OL approaches.

2 BACKGROUND

Considering that the real-time training of ANNs in an OL scenario involves multiple areas of research, we give in the following subsections a short summary of the most important concepts and techniques used in this paper.

2.1 Online Learning

Online learning (OL) has gained visibility due to the increase in the velocity and volume of available data sources compared to the past decade [11]. OL algorithms are trained using data streams as input, which differs from traditional ML algorithms that have a pre-defined training dataset.

Streams & Batches. Formally, a data stream $\mathcal{S}$ consists of ordered *events* e with *timestamps* s, i.e., $(e_1, s_1), \ldots, (e_\infty, s_\infty)$, where the s_i denote the processing time at which the corresponding events e_i are ingested into the system. These events are usually analysed in *batches* B_j of fixed size b, as follows:

$$B_1 = (e_1, s_1), \ldots, (e_b, s_b)$$
$$B_2 = (e_{b+1}, s_{b+1}), \ldots, (e_{2b}, s_{2b})$$
$$\ldots$$

Batches B_j are analyzed individually. Thus, if processed in an asynchronous stream-processing scenario, the batches (and in particular the included events e_i) can become out-of-order as they are handled within the system, even if the initial s_i were ordered. In common stream-processing architectures, such as Apache Flink [8], Spark [38] and Samza [25], batches are distinguished into *sliding windows*, *tumbling windows* and (per-user) *sessions* [5].

Latency vs. Throughput. When analyzing systems that process data streams, one typically benchmarks them by their latency and throughput [18]. Formally, *latency* is the time it takes for a system to process an event, from the moment it is ingested to the moment it is used to produce a desired output. *Throughput*, on the other hand, is the number of events that a system can receive and process per time unit. The *sustainable throughput* is the maximum throughput at which a system can handle a stream over a sustained period of time (i.e., without exhibiting a sudden surge in latency, then called "backpressure" [21], or even a crash).

Passive & Active Drift Adaptation. To adapt to concept drifts, one may rely on either passive or active adaptation strategies [13]. The passive strategy updates the trained model indefinitely, with no regard to the actual presence of concept drifts. Active drift adaptation strategies, on the other hand, only adapt the model when a concept drift has been explicitly identified.

2.2 Artificial Neural Networks

ANNs denote a family of supervised ML algorithms which are designed to be trained on a pre-defined dataset [12]. A training dataset is composed of multiple (x, y) pairs, in which x is a training example and y is its corresponding label. ANNs are usually trained using *mini-batches* X, which are sets of (x, y) pairs of fixed size N that are iteratively (randomly) sampled from the training dataset, thus $X = (x_i, y_i), \ldots, (x_{i+N}, y_{i+N})$.

An ANN model is represented by the weights and biases of the network, described together by θ and it is usually trained with

variants of *stochastic gradient descent* (SGD) [29]. SGD updates θ by considering $\nabla L(X, \theta)$, which is the gradient of a pre-defined *loss function* L with respect to θ when taking X as input. Thus, we can represent the update rule of θ as in Equation 1, in which t is the iteration in SGD, and α is a pre-defined learning rate.

$$\theta_{t+1} = \theta_t - \alpha \nabla L(X, \theta_t) \quad (1)$$

Based on Equation 1, θ_{t+1} is defined based on two terms. The second term is the more computationally expensive one to calculate, which we refer to as *gradient calculation* (GC). The remainder of the equation we call *gradient application* (GA), which consists of the subtraction between the two terms and the assignment of the result to θ_{t+1}.

Distributed Artificial Neural Networks. Over the last years, ANN models have substantially grown in size and complexity. Consequently, the usage of traditional centralized architectures has become unfeasible when training complex models due to the high amount of time they spend until convergence [30]. Researchers have been studying how to distribute ANN training to mitigate this. Distributed ANNs reduce the time it takes to train a complex ANN model by distributing its computation across multiple compute nodes. This distribution can follow different parallelization methods, system architectures, and synchronisation settings [24].

The most common form of distributing ANNs, which we also use in this work, is referred to as *data parallelism* [26], in which workers are initialized with replicas of the same initial model and trained with disjoint splits of the training data. Moreover, the synchronisation among the workers' parameters in a data-parallel ANN setting is either *centralised* or *decentralised* [24]. In a centralised architecture [26], workers systematically send their parameter updates to one or multiple parameter servers. Those servers aggregate the updates of all workers and apply them to a centralised model [24]. Thus, by relying on parameter servers to aggregate updates, the parameter servers may become the bottleneck of such an architecture [9]. On the other hand, in a decentralised architecture [26], the workers synchronize themselves using a broadcasting-like form of communication [26]. This broadcast eliminates the bottleneck of the parameter servers but requires a direct communication among worker nodes.

The parameter updates in a data-parallel ANN system can be *synchronous* or *asynchronous*. In a synchronous setting [26], workers have to synchronize themselves after each mini-batch iteration. This synchronization barrier wastes computational resources at idle times (i.e., when workers have to wait for others to resume their computation) [24]. In an asynchronous SGD (ASGD) setting [26], workers are allowed to compute their gradient computations also on stale model parameters. This behaviour obviously minimizes idle times but makes it harder to mathematically prove SGD convergence. Recent developments on ASGD [17, 32, 39], however, have tackled exactly this issue under different assumptions. Zhang et al. [39] recently proved an $\mathbf{o}(1/\sqrt{k})$ convergence rate for unbounded non-convex problems using ASGD under a centralised parameter server setup (where k denotes the iteration among the ASGD updates). Additionally, [7, 22, 34] proved the convergence of ASGD on decentralized networks under distinct assumptions and network topologies.

3 TENSAIR

We now introduce the architecture of *TensAIR*, the first framework for training and predicting in ANNs models in real-time. TensAIR was designed to work in association with stream-processing engines that allow asynchronous and decentralized communication among dataflow operators.

TensAIR introduces the *data-parallel*, *decentralized*, *asynchronous* ANN operator `Model`, with `train` and `predict` as two new OL functions. This means that TensAIR can scale out both the training and prediction tasks of an ANN model to multiple compute nodes, either with or without GPUs associated with them. TensAIR dataflow can be visualized using a graph (see Figure 1). Note that, throughout this paper, we use the terms `prediction` and `inference` interchangeably.

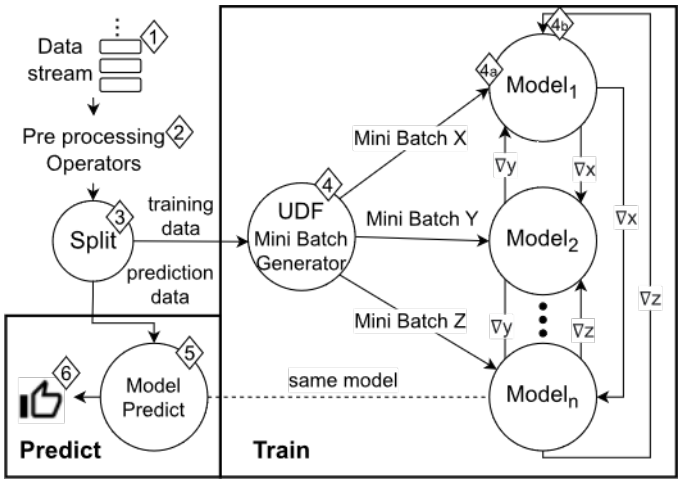


Figure 1: TensAIR generic dataflow with n distributed `Model` instances.

TensAIR Dataflows. Figure 1 depicts a generic *TensAIR dataflow*. This dataflow is composed of a single input data stream, n instances of the `Model` operator, and single instances of the `Split` and `UDF` operators. The idea behind a TensAIR dataflow is: (1) to receive training samples from the input data streams; (2) to pre-process the data received using common dataflow operators like `Map`, `Reduce`, `Split`, and `Join` to transform the data as deemed necessary given each use case; (3) to select whether the pre-processed data samples will be used for training, for prediction, or for both; (4) if data is sent for training, to aggregate a pre-defined number of samples in the form of a mini-batch by using a `user-defined function UDF`, and to send this mini-batch to one of the decentralized `Model` instances; (4a) when a `Model` instance receives a mini-batch X from the `UDF`, to calculate an update ∇x based on the current `Model` weights and the mini-batch X, to apply the update to itself, and to broadcast the update to other `Model` instances; (4b) when $Model_i$ receives an update from $Model_j$, to apply the received update locally; (5) if the pre-processed data is sent for prediction, to randomly select one of the distributed models and use it to perform the prediction; (6) to use the prediction previously made as final output of the dataflow or as input of further operators (as deemed necessary by the given use case).

Stream Processing. As shown in Algorithm 1, a TensAIR `Model` operator has two new OL functions `train` and `predict`, which can asynchronously send and receive messages to and from other operators. During `train`, `Model` receives either *encoded mini-batches* X or *gradients* ∇x as messages. Each message encoding a gradient that was computed by another model instance is immediately used to update the local model accordingly. Each mini-batch first invokes a local gradient computation and is then used to update the local model. Each such resulting gradient is also locally summed until a desired number of gradients (*maxGradBuffer*) is reached, upon which the buffer then is broadcast to all other `Model` instances.

Algorithm 1 TensAIR `Model` class

```
1:  Constructor Model (tfModel, maxBuffer):
2:      model = tfModel
3:      maxGradBuffer = maxBuffer
4:      gradients = ∅
5:      gradients_count = 0
6:  procedure PROCESS_MSG(msg)
7:      if msg.mode == TRAIN then
8:          train(msg)
9:      else
10:         predict(msg)
11: procedure TRAIN(msg)
12:     if msg.isGradient then
13:         model = apply_gradient(model, msg)
14:     else
15:         gradient = calculate_gradient(model, msg)
16:         model = apply_gradient(model, gradient)
17:         gradients += gradient
18:         gradients_count += 1
19:         if gradients_count ≥ maxGradBuffer then
20:             send_gradients(gradients)
21:             gradients_count = 0
22: procedure PREDICT(msg)
23:     predictions = model.make_predictions(msg)
24:     send_results(predictions)
```

3.1 Model Consistency

Despite TensAIR's asynchronous nature, it is necessary to maintain the models consistent among themselves during training in order to guarantee that they are aligned and, therefore, they eventually convergence to a same common model. In TensAIR, this is given by the exchange of gradients between the various `Model` instances.

Due to our asynchronous computation and application of the gradients on the distributed model instances, $\texttt{Model}_i$ receives gradients calculated by $\texttt{Model}_j$ (with $j \neq i$) which are similar but not necessarily equal to itself. This occurs whenever $\texttt{Model}_i$, which has already applied to itself a set of $G_i = \{\nabla x, \nabla y, ..., \nabla z\}$ gradients, calculates a new gradient ∇a, and sends it to $\texttt{Model}_j$, such that $G_i \neq G_j$ at the time when $\texttt{Model}_j$ applies ∇a. The difference $|G_i \cup G_j| - |G_i \cap G_j|$ between these two models is defined as *staleness* [33]. This $staleness_{i,j}(\nabla_a)$ metric is the symmetric distance between G_i and G_j with respect to the times at which a new gradient ∇_a was computed by a model i and is applied to model j, respectively. We illustrate this phenomenon and the staleness metric in Figure 2.

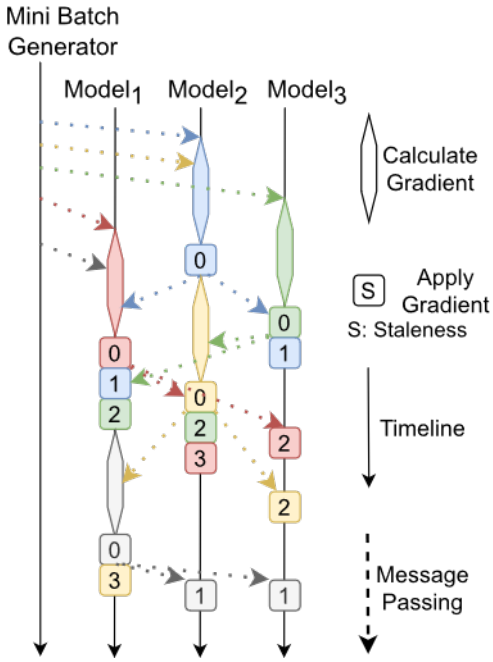


Figure 2: DASGD staleness calculation.

Figure 2 illustrates the timeline of messages (containing both mini-batches and gradients) exchanged among TensAIR models considering *maxGradBuffer* = 1. Assume the `UDF` distributes 5 mini-batches to 3 models. After receiving their first mini-batch, each $\texttt{Model}_i$ calculates a corresponding gradient. Note that, when applied locally, the staleness of any gradient is 0 because it is computed and immediately applied by the same model. While computing or applying a local gradient, each $\texttt{Model}_i$ may receive more gradients to calculate and/or apply from either the `UDF` or other models *asynchronously*. In our protocol, the models first finish their current gradient computation, apply it locally, then buffer and send *maxGradBuffer* many locally computed gradients to the other models, and wait for their next update.

As an illustration, take a look at $\texttt{Model}_2$ in Figure 2. While computing ∇_{blue}, it receives the yellow mini-batch from the `Mini Batch generator`, which it starts computing immediately after it finishes processing the blue one—which it had already started when it received the yellow mini-batch. During the computation of ∇_{yellow}, $\texttt{Model}_2$ receives ∇_{green} to apply, which it does promptly after finishing ∇_{yellow}. Note that when $\texttt{Model}_3$ computed ∇_{green} and $\texttt{Model}_1$ computed ∇_{red}, they have not applied a single gradient to their local models at that time. Thus, $|G_1| = |G_3| = 0$. However, before applying ∇_{green}, $G_2 = \{\nabla_{blue}, \nabla_{yellow}\}$ with $|G_2| = 2$ and $staleness_{3,2}(\nabla_{green}) = 2$. Along the same lines, before applying ∇_{red}, $|G_2| = 3$ and $staleness_{1,2}(\nabla_{red}) = 3$.

3.2 Model Convergence

Since TensAIR operates on data streams and is both asynchronous and fully decentralized (i.e., it has no centralized parameter server),

it exhibits characteristics that most SGD proofs of convergence [17, 32, 39] do not cover. Therefore, we next discuss under which circumstances TensAIR is guaranteed to converge.

First, we consider that training is performed between significant concept drifts. Therefore, we assume that the data distribution between two subsequent concept drifts does not change. Thus, if a concept drift occurs during the training, the model will not converge until the concept drift ends. By considering this, the data stream between two concept drifts will behave like a fixed data set. In this case, if given enough training examples, as seen in [12], each of the local model instances will eventually converge.

Second, considering TensAIR's decentralized and asynchronous SGD (DASGD), model updates can be staled. Nevertheless, as proven by Tosi and Theobald [34], the model will converge in this setting in up to $O(\frac{\sigma}{\epsilon^2}) + O(\frac{QS_{avg}}{\epsilon^{\frac{3}{2}}}) + O(\frac{S_{avg}}{\epsilon})$ iterations to an ϵ-small error, considering S_{avg} as the average staleness observed during training and Q a constant that bounds the gradients size. If bounded gradients are not assumed, DASGD converges in $O(\frac{\sigma}{\epsilon^2}) + O(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{\epsilon})$ iterations, with $\hat{S}_{max}$ and $\hat{S}_{avg}$ representing the maximum and average staleness, calculated using an additional recursive factor.

3.3 Implementation

TensAIR was implemented on top of the Asynchronous Iterative Routing (AIR) [36, 37] dataflow engine. AIR is a native stream-processing engine that processes complex dataflows in an asynchronous and decentralized manner. *TensAIR dataflow* operators extend a basic `Vertex` superclass in AIR. `Vertex` implements AIR's asynchronous MPI protocol via multi-threaded queues of incoming and outgoing messages, which are exchanged among all nodes (aka. "ranks") in the network asynchronously. This is crucial to guarantee that worker nodes do not stay idle while waiting to send or receive messages during training. The number of instances of each `Vertex` subclass and the number of input data streams can be configured beforehand, as seen in Figure 1.

TensAIR is completely implemented in C++. It includes the TensorFlow 2.8 native C API to load, save, train, and predict ANN models. Therefore, it is possible to develop a TensorFlow/Keras model in Python, save the model to a file, and load it directly into TensAIR. TensAIR is completely open-source and available from our GitHub repository[1].

4 EXPERIMENTS & DISCUSSION

To assess TensAIR, we performed experiments to measure its performance on solving prototypical ML problems such as Word2Vec (*word embeddings*) and CIFAR-10 (image classification). We empirically validate TensAIR's model convergence by comparing its training loss curve at increasing levels of distribution across both CPUs and GPUs. Our results confirm that TensAIR's DASGD updates achieve similar convergence on Word2Vec and CIFAR-10 as a synchronous SGD propagation. At the same time, we achieve a nearly linear reduction in training time on both problems. Due to this reduction, TensAIR significantly outperforms not just the current OL extensions of Apache Kafka and Flink (based on both the standard and distributed TensorFlow APIs), but also Horovod which is a long-standing effort to scale-out ANN training. Finally, by providing an in-depth analysis of a *sentiment analysis* (SA) use-case on Twitter, we demonstrate the importance of OL in the presence of concept drifts (i.e., COVID-19 related tweets with changing sentiments). In particular the SA usecase is an example of task that would be deemed too complex to be adapted in real-time (at a throughput rate of up to 6,000 tweets per second) when using other OL frameworks.

HPC Setup. We carried out the experiments described in this section using the HPC facilities of the University of Luxembourg [35]. We distributed the ANNs training using up to 4 Nvidia Tesla V100 GPUs in a node with 768 GB RAM. We also deployed up to 16 regular nodes, with 28 CPU cores and 128 GB RAM each, for the CPU-based (i.e., without using GPU acceleration) settings.

Event Generation. We trained both sparse (word embeddings[2]) and dense (image classification[3]) models based on English Wikipedia articles and images from CIFAR-10 [20], respectively. Instead of connecting to actual streams, we chose those static datasets to facilitate a consistent analysis of the results and ensure reproducibility. Moreover, to simulate a streaming scenario, we implemented the `MiniBatchGenerator` as an entry-point `Vertex` operator (compare to Figure 1) which generates events e_i with timestamps s_i, groups them into mini-batches X_j by using a tumbling-window semantics, and sends these mini-batches to the subsequent operators in the dataflow. Furthermore, this allows us to simulate streams of unbounded size by iterating over the datasets multiple times (in analogy to training with multiple epochs over a fixed dataset).

Sparse vs. Dense Models. We chose Word2Vec and CIFAR-10 because they represent prototypical ML problems with *sparse* and *dense* model updates, respectively. Sparse updates mean that only a small portion of the neural network variables actually become updated per mini-batch [28]. Hence, sparseness should assist the models' convergence when using DASGD, as observed also in Hogwild! [28]. We trained by sampling 1% from English Wikipedia which corresponds to 11.7M training examples (i.e., word pairs). On the other hand, we chose CIFAR-10 for being dense. Thus, we could analyze how this characteristic possibly hinders convergence when models are distributed and updated asynchronously. We train on all of the 50,000 labeled images of the CIFAR-10 dataset.

4.1 Convergence Analysis

We first explored TensAIR's ability to converge by determining if and how DASGD might degrade the quality of the trained model (Figure 3). We compared the training loss curve of Word2Vec and CIFAR-10 by distributing TensAIR models from 1 to 4 GPUs using 1 TensAIR rank per GPU (Figures 3b & 3d). We additionally explored the models convergence when trained with distributed CPU nodes (Figures 3a & 3c). In this second scenario, we trained up to 64 ranks on 16 nodes simultaneously without GPUs. Note that, when using a single TensAIR rank, TensAIR's gradient updates behave as in a synchronous SGD implementation.

The *extremely low variance* among all loss curves shown in Figures 3a and 3b demonstrates that our asynchronous and distributed

[1] https://github.com/maurodlt/TensAIR

[2] https://www.tensorflow.org/tutorials/text/word2vec

[3] https://www.tensorflow.org/tutorials/images/cnn

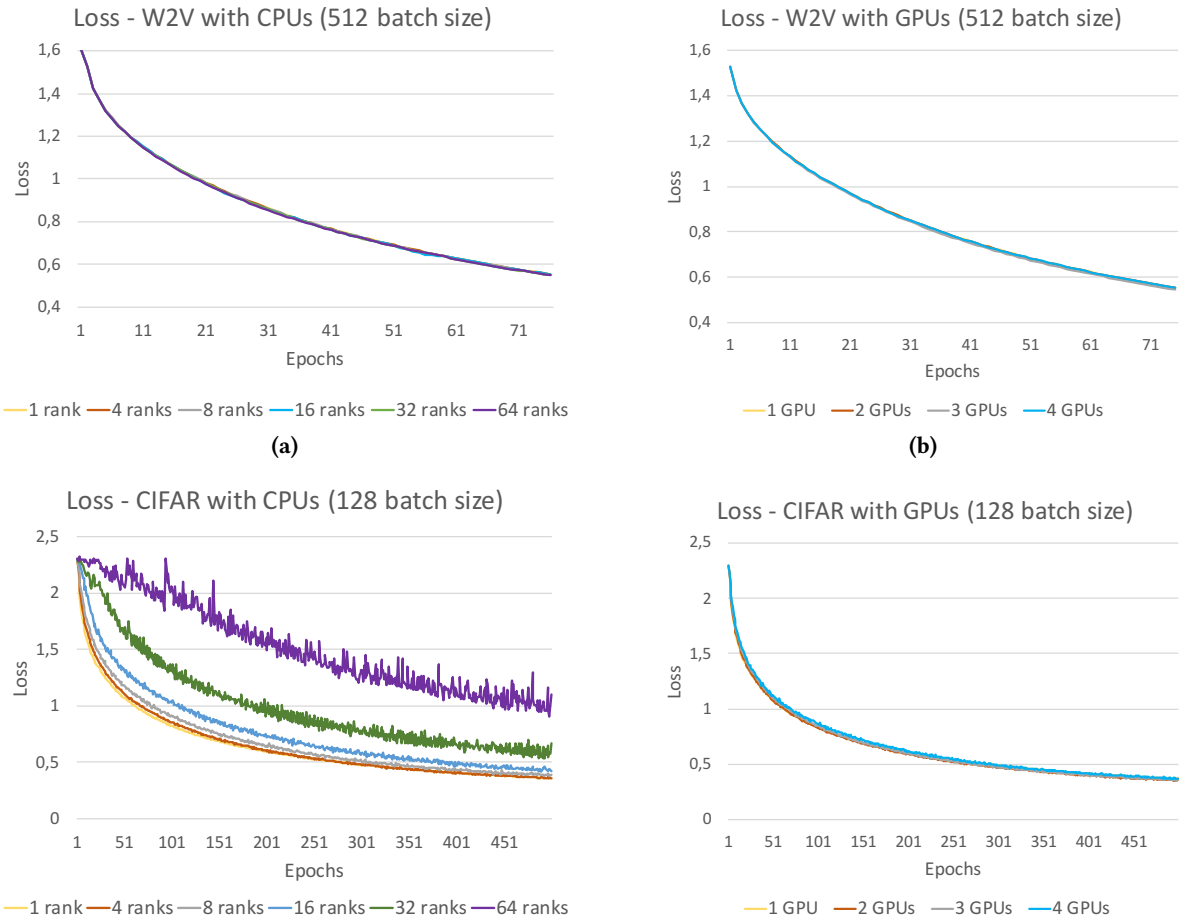


Figure 3: Convergence analysis of TensAIR on the Word2Vec and CIFAR-10 use-cases.

SGD updates do not at all negatively affect the convergence of the Word2Vec models. We assume that this is due to (1) the sparseness of Word2Vec, and (2) a low staleness of the gradients (which are relatively inexpensive to compute and apply for Word2Vec). The low staleness indicates a fast exchange of gradients among models.

In Figure 3c, we however observe a remarkable degradation of the loss when distributing CIFAR-10 across multiple nodes. This is due to the fixed learning rate used on all settings being the same. When distributing dense models on multiple ranks without adapting the mini-batch size, it is well known to result in a degradation of the loss curve (even on synchronous settings). This degradation occurs because the behaviour of training N models with mini-batches of size b is similar to training 1 model with mini-batches of size $N \cdot b$. To mitigate this issue, Horovod increases the learning rate α by the number of ranks used to distribute the model [1], i.e., $\alpha_{new} = \alpha \cdot N$. Accordingly, in Figure 3d, we again do not see any degradation of the loss when distributing CIFAR-10 because we use a maximum of 4 GPUs.

4.2 Speed-up Analysis

Next, we explore the performance of TensAIR under increasing levels of distribution and with respect to varying mini-batch sizes over both Word2Vec and CIFAR-10. This experiment is also deployed on up to 64 ranks (16 nodes) and up to 4 GPUs (1 node). We observe in Figure 4 that TensAIR achieves a *nearly-linear scale-out* under most of our settings.

In most cases, TensAIR achieves a better speedup when training with smaller mini-batches. This difference is because, differently than the gradient calculation, the gradient application is not distributed and, with smaller mini-batches, more gradients are applied per epoch. Thus, models with expensive gradient computations will have a better scale-out performance. Nevertheless, when gradient calculation is not the bottleneck of the dataflow, one can reduce the computational impact of the gradients application and the network impact of their broadcasts by simply increasing *maxBuffer*. For instance, by increasing *maxBuffer* in n times, the network complexity and the computational impact of the gradients applications are also expected to be reduce in n times.

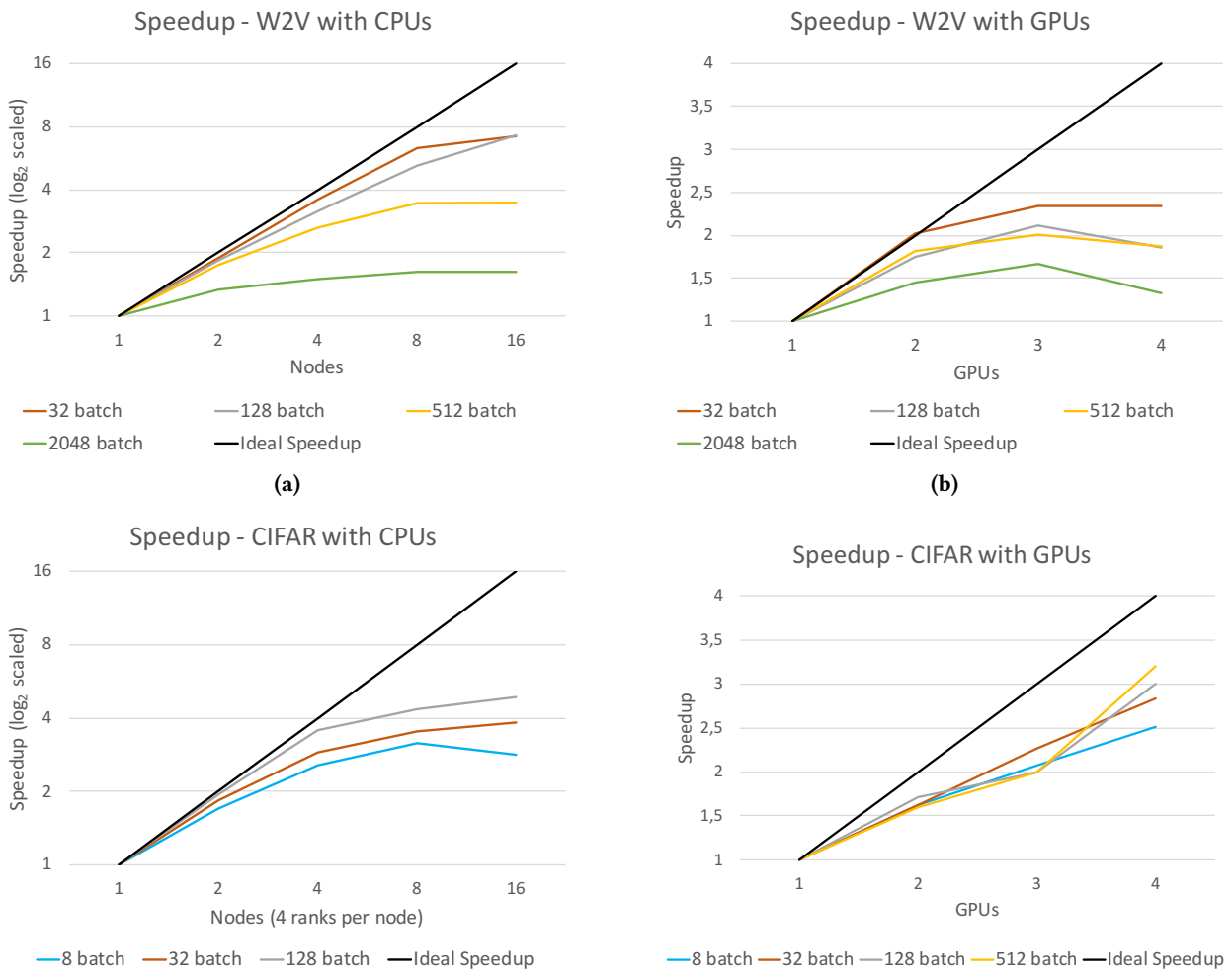


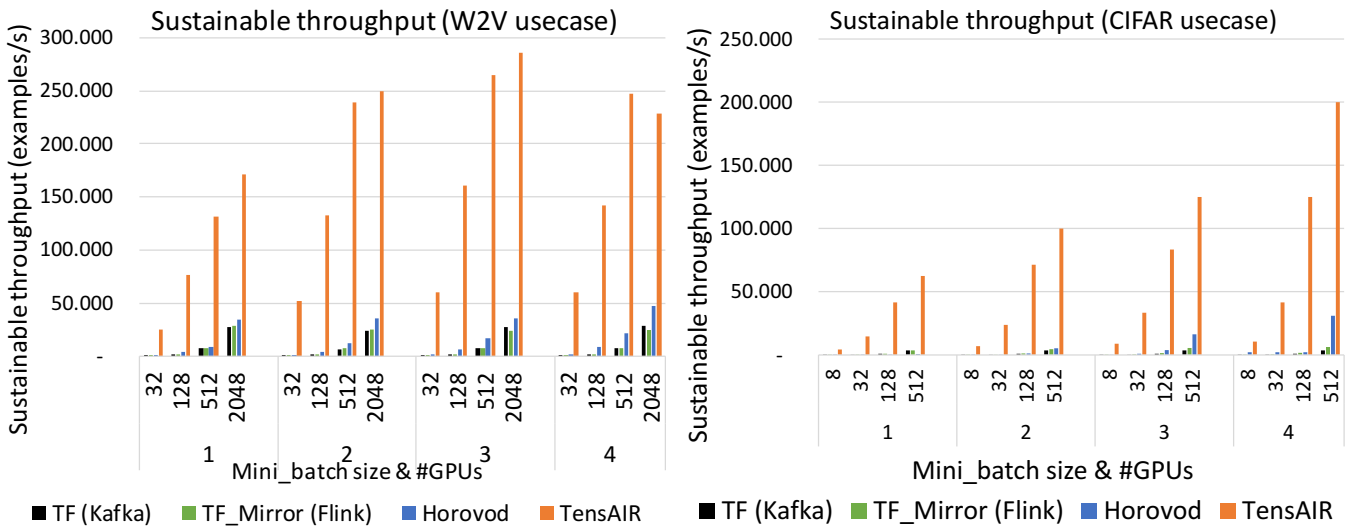
Figure 4: Speedup analysis of TensAIR on the Word2Vec and CIFAR-10 use-cases.

4.3 Baseline Comparison

Apart from TensAIR, it is also possible to train ANNs by using Apache Kafka and Flink as message brokers to generate data streams of varying throughputs. Kafka is already included in the standard TensorFlow I/O library (`tensorflow_io`), which however allows no actual distribution in the training phase [3]. Flink, on the other hand, employs the distributed TensorFlow API (`tensorflow.distribute`). However, we were not able to run the provided *dl-on-flink* use-case [2] even after various attempts on our HPC setup. We therefore report the direct deployment of our Word2Vec and CIFAR-10 use-cases (Figures 5a & 5b) on both the standard and distributed TensorFlow APIs (the latter using the `MirroredStrategy` option of `tensorflow.distribute`). We thereby, simulate a streaming scenario by feeding one mini-batch per training iteration into TensorFlow, which yields a very optimistic upper-bound for the maximum throughput that Kafka and Flink could achieve. In a similar manner, we also determined the maximum throughput of Horovod [30], which is however not a streaming engine by default.

In Figures 5a and 5b, we see that TensAIR clearly surpasses both the standard and distributed TensorFlow setups as well as Horovod. This occurs because, as opposed to TensAIR, their architectures were not developed to train on batches arriving from data streams. Thus, in a streaming scenario, the overhead of transferring the training data to the worker nodes increases by the number of training steps. On the other hand, TensAIR was designed to train ANN models from high throughput data streams in real-time. Thus, the transfer of training data overhead is mitigated by the asynchronous protocol adopted and the training is speed-up by DASGD. This allows TensAIR to (1) reduce both computational resources and idle times while the data is being transferred, and (2) have an optimized buffer management for incoming mini-batches and outgoing gradients, respectively.

In our experiments, we could sustain a maximum training rate of 285,560 training examples per second on Word2Vec and 200,000 images per second on CIFAR-10, which corresponds to sustainable throughputs of 14.16 MB/s and 585 MB/s respectively. We reached these values by training with 3 GPUs on Word2Vec and 4 GPUs on



(a) W2V use-case. **(b) CIFAR-10 use-case.**

Figure 5: Throughput comparison between TensAIR, TensorFlow (standard and distributed), and Horovod.

CIFAR-10. Note that, while using more than 3 GPUs simultaneously, TensAIR did not achieve better sustainable throughput in the W2V usecase due to the relatively low complexity of the gradient calculations. In this scenario, the training bottleneck, typically associated with gradient calculation, shifted to the gradient application when using more than 3 GPUs, as the former is not distributed. Nevertheless, this issue can be mitigated by simply increasing the variable $maxBuffer$ (as explained in Section 4.2). This adjustment, delays the communication among distributed models while reducing the locally applied gradients by a factor of $maxBuffer$.

4.4 Sentiment Analysis of COVID19

Here, we exemplify the benefits of training an ANN in real-time from streaming data. To this end, we analyze the impact of concept drifts on a sentiment analysis setting, specifically drifts that occurred during and due to the COVID19 pandemic. First, we trained a large Word2Vec model using 20% of English Wikipedia plus the Sentiment140 dataset [10]. Then, we trained an LSTM model [4] using the Sentiment140 dataset together with the word embeddings we trained previously. After three epochs, we reached 78% accuracy on the training and the test set. However, language is always evolving. Thus, this model may not sustain its accuracy for long if deployed to analyze streaming data in real-time. We exemplify this by fine-tuning the word embeddings with 2M additional tweets published from November 1st, 2019 to October 10th, 2021 containing the following keywords: *covid19*, *corona*, *coronavirus*, *pandemic*, *quarantine*, *lockdown*, *sarscov2*. Then, we compared the previously trained word embeddings and the fine-tuned ones and found an average cosine difference of only 2%. However, despite being small, this difference is concentrated onto specific keywords.

As shown in Table 1, keywords related to the COVID-19 pandemic are the ones that most suffered from a concept drift. Take as example *pandemic*, *booster* and *corona*, which had over 62% of cosine difference before and after the Word2Vec models have been

Term	rt	corona	pandemic	booster	2021
Difference	0.728	0.658	0.646	0.625	0.620

Table 1: Cosine differences after updating word embeddings.

updated. Due to the concept drift, the sentiment over specific terms and, consequently, entire tweets also changed. One observes this change by comparing the output of our LSTM model when: (1) inputting tweets embedded with the pre-trained word embeddings; (2) inputting tweets embedded with the fine-tuned word embeddings. Take as an example the sentence "*I got corona.*", which had a sentiment of +2.0463 when predicted with the pre-trained embeddings; and −2.4873 when predicted with the fine-tuned embeddings. Considering that the higher the sentiment value the more positive the tweet is, we can observe that *corona* (also representing a brand of a beer) was seen as positive and now is related to a very negative sentiment.

To tackle concept drifts in this use-case, we argue that TensAIR with its OL components (as depicted in Figure 6) could be readily deployed. A real-time pipeline with Twitter would allow us to constantly update the word embeddings (our sustainable throughput would be more than sufficient compared to the estimated throughput of Twitter). Consequently, the sentiment analysis algorithm would always be up-to-date with respect to such concept drifts.

Figure 6 depicts the dataflow for a *Sentiment Analysis* (SA) usecase on a Twitter data stream. This dataflow predicts the sentiments of live tweets using a pre-trained ANN model ($\texttt{Model1}^{SA}$). However, it does not rely on pre-defined word embeddings. The dataflow constantly improves its embeddings on a second *Word2Vec* (W2V) ANN model ($\texttt{Model1}^{W2V}$), which it trains using the same input stream as used for the predictions. By following a passive concept-drift adaptation strategy, it can adapt its sentiment predictions in real-time based on changing word distributions among the input tweets. Moreover, it does not require any sentiment labels

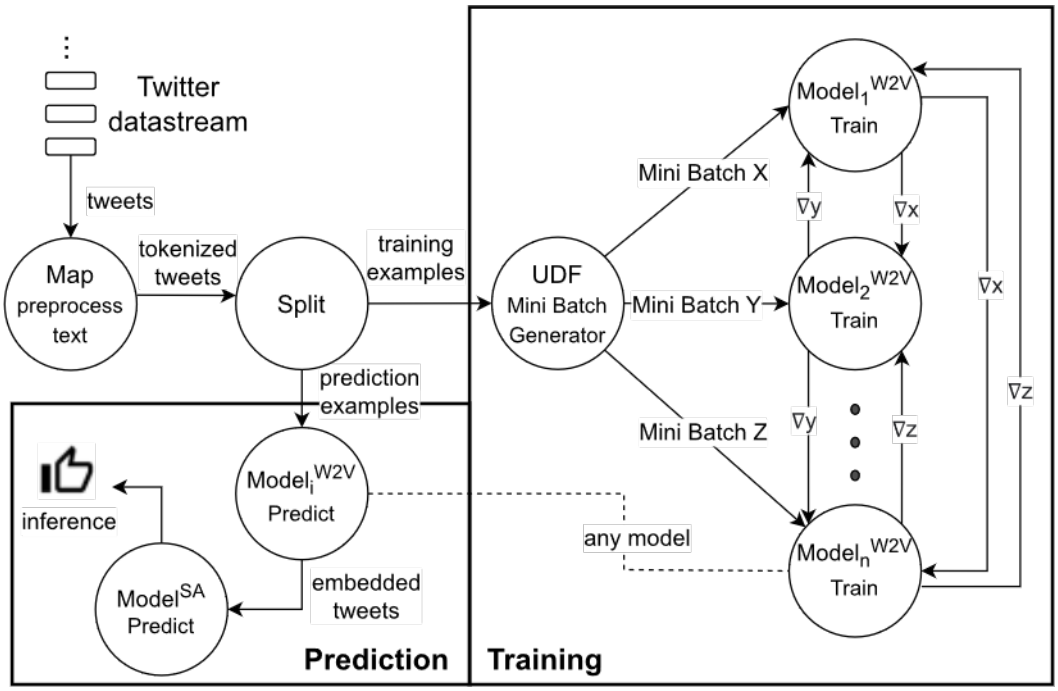


Figure 6: TensAIR dataflow with n distributed $\texttt{Model}^{W2V}$ instances and a single instance of `Map`, `Split`, `UDF` and $\texttt{Model}^{SA}$.

for newly streamed tweets at $\texttt{Model}^{SA}$, since only $\texttt{Model}^{W2V}$ is re-trained in a self-supervised manner by generating mini-batches of word pairs (x, y) directly from the input tweets.

Our SA dataflow starts with `Map` which receives tweets from a Twitter input stream (implemented via cURL or a file interface) and tokenizes the tweets based on the same word dictionary also used by $\texttt{Model}^{W2V}$ and $\texttt{Model}^{SA}$. `Split` then identifies whether the tokenized tweets shall be used for re-training the word embeddings, for sentiment prediction, or for both. If the tokenized tweets are selected for training, they are turned into mini-batches via the `UDF` operator. The (x, y) word pairs in each mini-batch X are sharded across $\texttt{Model}_1^{W2V}, \ldots, \texttt{Model}_n^{W2V}$ with a standard hash-partitioner using words x as keys. $\texttt{Model}^{W2V}$ implements a default skip-gram model. If the tokenized tweets are selected for prediction, a tweet is vectorized by using the word embeddings obtained from any of the $\texttt{Model}^{W2V}$ instances and sent to the pre-trained $\texttt{Model}^{SA}$ which then predicts the tweets' sentiments.

5 CONCLUSIONS

OL is an emerging area of research which still has not extensively explored the real-time training of ANNs. In this paper, we introduced TensAIR, a novel system for real-time training of ANNs from data streams. It uses the asynchronous iterative routing (AIR) protocol to train and predict ANNs in a decentralized manner. The two main features of TensAIR are: (1) leveraging the iterative nature of SGD by updating the ANN model with fresh samples from the data stream instead of relying on buffered or pre-defined datasets; (2) its fully asynchronous and decentralized architecture used to update the ANN models using decentralized and asynchronous SGD (DASGD). Due to those two features, TensAIR achieves a nearly linear scale-out performance in terms of sustainable throughput and with respect to its number of worker nodes. Moreover, it was implemented using TensorFlow, which facilitates the deployment of diverse use-cases. Therefore, we highlight the following capabilities of TensAIR: (1) processing multiple data streams simultaneously; (2) training models using either CPUs, GPUs, or both; (3) training ANNs in an asynchronous and distributed manner; and (4) incorporating user-defined dataflow pipelines. We empirically demonstrate that—in a real-time streaming scenario—TensAIR supports from 4 to 120 more sustainable throughput than Horovod and both the standard and distributed TensorFlow APIs (representing upper bounds for Apache Kafka and Flink extensions).

As future work, we believe that TensAIR may also lead to novel online learning use cases which were previously considered too complex but now become feasible due to the very good sustainable throughput of TensAIR. Specifically, we intend to study similar learning tasks over audio/video streams, which we see as the main target domain for stream processing and OL. To reduce the computational cost of training an ANN indefinitely, we shall also investigate how different active concept-drift detection algorithms behave under an OL setting with ANNs.

ACKNOWLEDGMENTS

This work is funded by the Luxembourg National Research Fund under the PRIDE program (PRIDE17/12252781). The paper benefited from helpful comments and suggestions by Ovidiu Cristian Marcu. The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [35] (see `hpc.uni.lu`).

REFERENCES

[1] 2019. *Horovod with Keras.* https://horovod.readthedocs.io/en/stable/keras.html Accessed: 2022-05-18.

[2] 2022. *Deep Learning on Flink.* https://github.com/flink-extended/dl-on-flink Accessed: 2022-08-05.

[3] 2022. *Robust machine learning on streaming data using Kafka and Tensorflow-IO.* https://www.tensorflow.org/io/tutorials/kafka Accessed: 2022-08-05.

[4] 2022. *TensorFlow.* https://www.tensorflow.org/text/tutorials/text_classification_rnn Accessed: 2022-05-27.

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[6] Roberto Souto Maior Barros and Silas Garrido T Carvalho Santos. 2018. A large-scale comparison of concept drift detectors. *Information Sciences* 451 (2018), 348–370.

[7] Marco Bornstein, Tahseen Rabbani, Evan Wang, Amrit S. Bedi, and Furong Huang. 2023. SWIFT: Rapid Decentralized Federated Learning via Wait-Free Model Communication. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=jh1nCir1R3d

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[9] Chen Chen, Wei Wang, and Bo Li. 2019. Round-robin synchronization: Mitigating communication bottlenecks in parameter servers. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 532–540.

[10] Alec Go, Richa Bhayani, and Lei Huang. 2009. Twitter sentiment classification using distant supervision. *CS224N project report, Stanford* 1, 12 (2009).

[11] Heitor Murilo Gomes, Jesse Read, Albert Bifet, Jean Paul Barddal, and João Gama. 2019. Machine learning for streaming data: state of the art, challenges, and opportunities. *ACM SIGKDD Explorations Newsletter* 21, 2 (2019), 6–22.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press. http://www.deeplearningbook.org.

[13] Moritz Heusinger, Christoph Raab, and Frank-Michael Schleif. 2020. Passive concept drift handling via variations of learning vector quantization. *Neural Computing and Applications* (2020), 1–12.

[14] Steven CH Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. 2021. Online learning: A comprehensive survey. *Neurocomputing* 459 (2021), 249–289.

[15] Hanqing Hu, Mehmed Kantardzic, and Tegjyot S Sethi. 2020. No Free Lunch Theorem for concept drift detection in streaming data classification: A review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10, 2 (2020), e1327.

[16] Adriana Sayuri Iwashita and Joao Paulo Papa. 2018. An overview on concept drift learning. *IEEE access* 7 (2018), 1532–1547.

[17] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 463–478.

[18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1507–1518.

[19] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[20] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images.* Technical Report. University of Toronto, Department of Computer Science.

[21] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 239–250.

[22] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. 2018. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning*. PMLR, 3043–3052.

[23] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2018), 2346–2363.

[24] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–37.

[25] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[26] Shuo Ouyang, Dezun Dong, Yemao Xu, and Liquan Xiao. 2021. Communication optimization strategies for distributed deep neural network training: A survey. *J. Parallel and Distrib. Comput.* 149 (2021), 52–65.

[27] S Priya and R Annie Uthra. 2021. Deep learning framework for handling concept drift and class imbalanced complex decision-making on streaming data. *Complex & Intelligent Systems* (2021), 1–17.

[28] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011).

[29] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951), 400–407.

[30] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[31] Tegjyot Singh Sethi and Mehmed Kantardzic. 2017. On the reliable detection of concept drift from streaming unlabeled data. *Expert Systems with Applications* 82 (2017), 77–99.

[32] Tao Sun, Robert Hannah, and Wotao Yin. 2017. Asynchronous coordinate descent under more realistic assumptions. *Advances in Neural Information Processing Systems* 30 (2017).

[33] Mauro DL Tosi, Vinu Ellampallil Venugopal, and Martin Theobald. 2022. Convergence-Time Analysis of Asynchronous Distributed Artificial Neural Networks. In *5th Joint International Conference on Data Science & Management of Data (CODS/COMAD)*. 314–315.

[34] Mauro DL Tosi and Martin Theobald. 2023. Convergence Analysis of Decentralized ASGD. *arXiv e-prints* (2023), arXiv–2309.

[35] S. Varrette, H. Cartiaux, S. Peter, E. Kieffer, T. Valette, and A. Olloh. 2022. Management of an Academic HPC & Research Computing Facility: The ULHPC Experience 2.0. In *Proc. of the 6th ACM High Performance Computing and Cluster Technologies Conf. (HPCCT 2022)*. Association for Computing Machinery (ACM), Fuzhou, China.

[36] Vinu E Venugopal, Martin Theobald, Samira Chaychi, and Amal Tawakuli. 2020. AIR: A light-weight yet high-performance dataflow engine based on asynchronous iterative routing. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 51–58.

[37] Vinu Ellampallil Venugopal, Martin Theobald, Damien Tassetti, Samira Chaychi, and Amal Tawakuli. 2022. Targeting a Light-Weight and Multi-Channel Approach for Distributed Stream Processing. *J. Parallel and Distrib. Comput.* (2022).

[38] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[39] Xin Zhang, Jia Liu, and Zhengyuan Zhu. 2020. Taming convergence for asynchronous stochastic gradient descent with unbounded delay in non-convex learning. In *2020 59th IEEE Conference on Decision and Control (CDC)*. 3580–3585.