UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2024-22
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 28/03/2024 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN *Informatique*

by

Mauro DALLE LUCCA TOSI
Born on 13 April 1995 in São Paulo, Brazil

ONLINE LEARNING USING DISTRIBUTED NEURAL NETWORKS

Dissertation defence committee

"All we have to decide is what to do with the time that is given us." — Gandalf

The Lord of the Rings: The Fellowship of the Ring
written by J.R.R. Tolkien

Abstract

Online Learning using Distributed Neural Networks

by Mauro DALLE LUCCA TOSI

Online Learning (OL), a sub-field of Machine Learning (ML), focuses on addressing time-sensitive problems through iterative learning from data streams. This field is characterized by the challenge of *concept drift*, where the data distribution evolves over time, necessitating algorithms that can adapt dynamically for accurate predictions. Traditional OL algorithms, while efficient and less resource-intensive than conventional ML methods, often fall short in non-linear high-dimensional problems. This gap has led to the integration of Artificial Neural Networks (ANN) in OL, albeit with limitations —these solutions allow for real-time inference but require offline training, leading to periods of low-quality predictions during concept drifts.

This doctoral thesis explores the potential of online training ANN models within an OL setting, a multidisciplinary endeavor encompassing Big Data, Machine Learning, Optimization, Distributed Computing, and System Development. The thesis introduces TensAIR, the first OL system capable of training ANN models online. TensAIR utilizes the iterative aspect of Stochastic Gradient Descent (SGD) to train models directly from data streams. Its asynchronous and decentralized architecture allows for significantly faster processing — handling 6 to 116 times more data samples per second compared to baselines modified for online training.

Further, the thesis addresses the challenge of unnecessary retraining of ANN models that have already converged and are unaffected by concept drift. Commonly used concept-drift detectors are often tailored for binary data and exhibit a high rate of false positives, rendering them less suitable for ANN models. Addressing this gap, we introduce OPTWIN, which evaluates shifts in both the mean and the standard deviation of prediction errors. This approach not only matches the recall rates of existing detectors but reduces the incidence of false positives. The impact of OPTWIN's advanced detection capabilities lead it to a significant 21% decrease in the time required for retraining models, as compared to traditional methods. This enhancement in retraining efficiency represents a major stride forward in maintaining the accuracy and reliability of ANN models in dynamic OL environments.

In addition to presenting empirical evidence that underscores the effectiveness of TensAIR, this thesis delves into theoretical contributions, notably through the validation of the convergence rate of the newly proposed decentralized and asynchronous Stochastic Gradient Descent (DASGD) algorithm. This critical proof establishes that DASGD can achieve a convergence rate on par with traditional synchronous methods, particularly under conditions characterized by low average communication delays. Importantly, this algorithm not only parallels the efficiency of synchronous approaches but also significantly alleviates the computational and communication bottlenecks often associated with synchronous processing. As a result, DASGD holds the potential to accelerate the training processes of ANN models in an OL setting, thereby enhancing their applicability and effectiveness in online scenarios.

In conclusion, this research successfully demonstrates the viability of training ANN models in an OL context. It not only offers practical solutions like TensAIR and OPTWIN but also contributes theoretically with the DASGD algorithm. This work is expected to inspire further research and lead to the creation of novel applications and use cases, leveraging the advantages of ANN models trained in Online Learning environments.

Acknowledgements

I extend my deepest gratitude to my wife, Renata, who courageously embarked on this adventure with me, leaving behind her familiar world to stand by my side in a foreign land in the middle of a pandemic. I am profoundly grateful for your resilience, love, and unwavering support during this challenging period. Your sacrifices and dedication have been my rock, and I cannot thank you enough.

I am indebted to my parents and siblings for their constant encouragement and belief in my dreams. Despite the distance of an ocean between us, their boundless love and support have been a source of strength, reminding me that distance can never diminish the power of familial bonds.

I am immensely thankful to my supervisor, Martin, for his invaluable guidance, trust, and encouragement throughout this thesis. Our friendly discussions have always been productive and motivating, alleviating the burden of producing this thesis and inspiring me to keep moving forward. His openness, patience, and constructive feedback have shaped my research journey, and I am truly grateful for his mentorship.

Special thanks also go to Vinu, Alessandro, Jingjing, and Maria for their friendship and camaraderie, which have made my time at the university more enriching and enjoyable. Our conversations, both technical and mundane, have left a lasting impression, and I am grateful for the moments shared.

Lastly, I am grateful to all those whose names may not appear here but whose contributions have nonetheless played a significant role in shaping this thesis. Your support, whether through words of encouragement or practical assistance, has not gone unnoticed and is deeply appreciated.

Contents

Abstract ii

Acknowledgements iii

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Contextual Foundations . . . 3
  - 1.2.1 Artificial Neural Networks (ANNs) . . . 3
  - 1.2.2 Concept Drifts . . . 5
- 1.3 Research Problem and Challenges . . . 6
- 1.4 Research Goal and Approach . . . 9
- 1.5 Contributions . . . 13
- 1.6 Organization . . . 14
- 1.7 Disclaimer . . . 15

2 Convergence Time Analysis of Asynchronous Distributed Artificial Neural Networks 16
- 2.1 Contribution Statement . . . 16
- 2.2 Chapter Contextualization . . . 17
- 2.3 Introduction . . . 17
- 2.4 Asynchronicity and Staleness . . . 18
- 2.5 Convergence Analysis & Directions . . . 19

3 TensAIR: Online Learning from Data Streams via Asynchronous Iterative Routing 21
- 3.1 Contribution Statement . . . 21
- 3.2 Chapter Contextualization . . . 22
- 3.3 Introduction . . . 22
- 3.4 Background & Related Work . . . 24
  - 3.4.1 Online Learning . . . 24
  - 3.4.2 Artificial Neural Networks . . . 25
  - 3.4.3 Distributed Artificial Neural Networks . . . 26
    - 3.4.3.1 Parallelization methods . . . 26
    - 3.4.3.2 System Architectures . . . 26
    - 3.4.3.3 Synchronization Settings . . . 27
  - 3.4.4 AIR – Distributed Dataflow Engine . . . 27
- 3.5 TensAIR Architecture . . . 27
  - 3.5.1 TensAIR . . . 28
  - 3.5.2 Model Consistency . . . 29
  - 3.5.3 Model Convergence . . . 30
  - 3.5.4 Implementation . . . 31
- 3.6 Experiments & Discussion . . . 31
  - 3.6.1 Convergence Analysis . . . 32

3.6.2 Speed-up Analysis . . . 34
3.6.3 Baseline Comparison . . . 35
3.6.4 Sentiment Analysis of COVID19 . . . 38
3.7 Conclusions . . . 39

4 OPTWIN: Drift Identification with Optimal Sub-Windows **41**
4.1 Contribution Statement . . . 41
4.2 Chapter Contextualization . . . 41
4.3 Introduction . . . 42
4.4 Background & Related Works . . . 43
4.4.1 Background . . . 44
4.4.2 Related Works . . . 44
4.5 OPTWIN – Optimal Window Concept Drift Detector . . . 46
4.5.1 Setup & Parameters . . . 46
4.5.1.1 Assumptions . . . 46
4.5.2 Algorithm . . . 47
4.5.3 Implementation & Analysis . . . 49
4.6 Experiments & Results . . . 49
4.6.1 MOA Experiments . . . 50
4.6.2 Classification Experiments . . . 52
4.6.3 Regression Experiments on Neural Networks . . . 55
4.7 Conclusion . . . 56
4.8 Supplementary Material . . . 57
4.8.1 Proof . . . 57
4.8.1.1 Definition of concepts . . . 57
4.8.1.2 Proof of Theorem 4.5.1 . . . 57
4.8.2 Additional Plots . . . 60

5 Convergence Analysis of Decentralized ASGD **65**
5.1 Contribution Statement . . . 65
5.2 Chapter Contextualization . . . 65
5.3 Introduction . . . 66
5.3.1 Contributions . . . 67
5.3.2 Limitations . . . 67
5.4 Related Works . . . 68
5.5 Optimization Objective & DASGD Algorithm . . . 69
5.5.1 Optimization Objective . . . 70
5.5.1.1 Notation . . . 71
5.5.1.2 Assumptions . . . 71
5.5.2 Network Topology & Communication Protocol . . . 72
5.5.3 Decentralized & Asynchronous SGD Algorithm . . . 72
5.6 Convergence Rate Analysis . . . 73
5.6.1 Discussion . . . 76
5.7 Experiments . . . 76
5.8 Conclusion . . . 79
5.9 Supplementary Material . . . 79
5.9.1 Useful Inequalities & Remarks . . . 79
5.9.1.1 Inequalities . . . 79
5.9.1.2 Remarks . . . 80
5.9.2 Proof of Theorem 5.6.1 . . . 80

5.9.2.1 Preliminaries for the Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.12) 82
5.9.2.2 Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.12) 84
5.9.2.3 Preliminaries for the Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.11) 85
5.9.2.4 Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.11) 86
5.9.3 DASGD - reduced communication 87
5.9.4 VGG-16 Experiments 88
5.9.4.1 Experimental setting 88
5.9.4.2 Results and Discussion 89

6 Implementation Aspects & Demonstration Setting **90**
6.1 Contribution Statement 90
6.2 Chapter Contextualization 90
6.3 Introduction 91
6.3.1 TensAIR Extensions for the Demonstration 92
6.4 Online Training ANN Models 92
6.4.1 Implementation 93
6.5 Demonstration Scenario 95
6.6 Conclusion 97

7 Discussion **98**
7.1 Discussion on the Individual Research Objectives 98
7.2 Limitations 101
7.3 Synthesis of Findings and Recommendations 104
7.3.1 Findings 105
7.3.2 Recommendations 105

8 Conclusion **107**
8.1 Summary of Contributions 107
8.2 Disseminating our findings 108
8.3 Future Work 109
8.4 Final Considerations 110

Bibliography **111**

List of Figures

1.1 Illustration of the impact of delayed adaptation on the quality of predictions or inferences in an ANN model under concept drifts. 2
1.2 Sources of concept drift, in which different formats and colors represent different classes. 6
1.3 Methodology to complete key objective of this thesis, leading to the design of a system for effectively employing ANN training into an Online Learning pipeline. 10

2.1 Asynchronous SGD distribution. 17
2.2 Convergence time ratio using 4 Word2Vec models per node varying the mini-batch sizes on 1 and n nodes. 19
2.3 Train and test losses of the distributed Word2Vec usecase using 128 minibatch size on 4 models per node. 20

3.1 TensAIR generic dataflow with n distributed `Model` instances. 28
3.2 ASGD staleness on a distributed framework. 30
3.3 Convergence analysis of TensAIR on the Word2Vec use-case. 33
3.4 Convergence analysis of TensAIR on the CIFAR-10 use-case. 34
3.5 Speedup analysis of TensAIR on the Word2Vec use-case. 35
3.6 Speedup analysis of TensAIR on the CIFAR-10 use-case. 36
3.7 Throughput comparison between TensAIR, TensorFlow (standard and distributed), and Horovod. 37
3.8 TensAIR dataflow with n distributed `Model`^{W2V} instances and a single instance of `Map`, `Split`, `UDF` and `Model`^{SA}. 39

4.1 Concept drift types. 44
4.2 Sudden binary drift detection with average FP rates compared to drift-detection delays. 53
4.3 TP and FP rates for drift detection on the AGRAWAL dataset with sudden concept drifts. 54
4.4 Sudden drift detection over the loss of a CNN. 56
4.5 Gradual binary drift detection with average FP rates compared to drift-detection delays. 61
4.6 Sudden non-binary drift detection with average FP rates compared to drift-detection delays. 62
4.7 Gradual non-binary drift detection with average FP rates compared to drift-detection delays. 62
4.8 TP and FP rates for drift detection on the STAGGER dataset with sudden concept drifts. 63
4.9 TP and FP rates for drift detection on the RandomRBF dataset with sudden concept drifts. 64

5.1 *Staleness* $S^{2,3}_{1,2}$ at the time when Model$_2$ applies gradient ∇_3 calculated by Model$_1$. The grey areas determine the sets of gradients G^2_1 and G^3_2 used to calculate the symmetric difference. 74
5.2 Numbers of iterations and runtimes needed to reach an error of ϵ (each averaged over 4 runs, the shaded areas denote one standard deviation). 78
5.3 VGG-16 model trained with DASGD (Algorithm 4). 89

6.1 TensAIR dataflow with active drift adaptation. 93
6.2 Code exemplifying implementation of generic TensAIR dataflow. . . . 95
6.3 Encoding of message returned by *next message* method. 95
6.4 Prediction-loss curve of a CNN model over the CIFAR-10 dataset with simulated drifts. Each point from the data stream corresponds to a batch of 256 images. 96

List of Tables

3.1 Cosine differences after updating word embeddings. 38

4.1 Statistics of drift identification in classification experiments on synthetic datasets. 51
4.2 Statistics of drift identification in regression experiments on synthetic datasets. 51
4.3 Average statistics of drift identification on synthetic datasets. 52
4.4 Accuracy of NB on synthetic and real-world datasets. "Sud"and "Grad" representing sudden and gradual concept drifts respectively. "Elec" and "Cove" representing the Electricity and the Covertype datasets. . 55

5.1 Comparison between SGD convergence proofs. 70

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AIR	**A**synchronous **I**terative **R**outing
ANN	**A**rtificial **N**eural **N**etwork
API	**A**pplication **P**rogramming **I**nterface
ASGD	**A**synchronous **S**tochastic **G**radient **D**escent
ASP	**A**synchronous **S**tream **P**rocessing
BGD	**B**atch **G**radient **D**escent
BSP	**B**ulk **S**ynchronous **P**arallel
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
CRediT	**C**ontributor **R**oles **T**axonomy
cURL	client for **URL**
DAG	**D**irected **A**cyclic **G**raph
DASGD	**D**ecentralized **A**synchronous **S**tochastic **G**radient **D**escent
DNN	**D**eep **N**eural **N**etwork
ELM	**E**xtreme **L**earning **M**achines
EWMA	**E**xponentially **W**eighted **M**oving **A**verage
FL	**F**ederated **L**earning
FN	**F**alse **N**egative
FP	**F**alse **P**ositive
GA	**G**radient **A**pplication
GAN	**G**enerative **A**dversarial **N**etworks
GC	**G**radient **C**alculation
GPU	**G**raphics **P**rocessing **U**nit
HPC	**H**igh-**P**erformance **C**omputing
IoT	**I**nternet **o**f **T**hings
LLM	**L**arge **L**anguage **M**odel
LSTM	**L**ong **S**hort-**T**erm **M**emory
MBG	**M**ini-**B**atch **G**enerator
ML	**M**achine **L**earning
MOA	**M**assive **O**nline **A**nalysis
MPI	**M**essage **P**assing **I**nterface
NB	**N**aïve **B**ayes
NN	**N**eural **N**etwork
OL	**O**nline **L**earning
POSIX	**P**ortable **O**perating **S**ystem **I**nterface
SGD	**S**tochastic **G**radient **D**escent
TP	**T**rue **P**ositive
UDF	**U**ser-**D**efined **F**unction
W2V	**W**ord**2V**ec
SA	**S**entiment **A**nalysis
URL	**U**niform **R**esource **L**ocator

List of Symbols

Ω	data stream	$\Omega = (e_0, \omega_0), \ldots, (e_\infty, \omega_\infty)$
e	events from data stream	
ω	time at which an event e is processed	
B	batches from datastream of size b	$B = (e_0, s_0), \ldots, (e_b, \omega_{b-1})$
z	data sample	
y	data label	
ξ	training sample	$\xi = (z, y)$
$\Xi_{0,d}$	dataset composed of d training samples	$\Xi_{0,d} = \xi_0, \ldots, \xi_{d-1}$
θ	mini-batch of training samples of size c	$\theta = (z_i, y_i), \ldots, (z_{i+c}, y_{i+c})$
$\mathcal{D}(\Xi)$	distribution of dataset Ξ	
x	model parameters	
x^0	initial model parameters	
x_i^t	model i at iteration t	
η	stepsize	
ϵ	small-error	
F	loss function	$F(x, \xi)$
f	stochastic version of function F	$f(x) = \mathbb{E}_{\xi \sim \mathcal{D}} F(x, \xi)$
$\nabla f(x_i^t)$	gradient of function f in respect to x_i^t	
g_i^t	gradient calculated using x_i^t and ξ_i^t	$g_i^t = \nabla F(x_i^t, \xi_i^t)$
$\nabla_{x,\xi}$	gradient calculated using model x and data ξ_i^t	
∇_a	gradient a	
Q	bound of gradients	
G_i^t	set of gradients applied from x^0 to x_i^t	$x_i^t = x_i^0 - \eta \sum_{g \in G_i^t} g$
$S_{i,j}^{t,s}$	staleness between sets of gradients G_i^t and G_j^s	
S_{avg}	average staleness	
$\hat{S}_{i,j}^{t,s}$	generalized version of staleness between G_i^t and G_j^s	
$\hat{S}_{max}$	maximum of generalized version of staleness	
$\hat{S}_{avg}$	average of generalized version of staleness	
$\delta_{i,j}^{t,s}$	sum of gradients in $S_{i,j}^{t,s}$ or $\hat{S}_{i,j}^{t,s}$, weighted by η	$\delta_{i,j}^{t,s} = \eta \sum_{g \in S_{i,j}^{t,s}} g$, with $\delta_{i,j}^{t,s} \in \mathbb{R}^d$
$\Delta_{i,j}^{t,s}$	upper bound of $\delta_{i,j}^{t,s}$	$\Delta_{i,j}^{t,s} = \eta \sum_{g \in S_{i,j}^{t,s}} \lvert g \rvert$, with $\Delta_{i,j}^{t,s} \in \mathbb{R}^d$
τ	gradient delay	
τ_{max}	maximum gradient delay	
τ_{avg}	average gradient delay	
N	number of distributed nodes	
n	number of distributed models	
bf	broadcast frequency	
$G_{B \rightsquigarrow \nabla_{x,\xi}}$	gradients applied by Model B before calculating $\nabla_{x,\xi}$	
$G_{\nabla_{x,\xi} \rightsquigarrow A}$	gradients applied by Model A before applying $\nabla_{x,\xi}$	
W	sliding window of error rates	
μ_W	mean of W	
σ_W	standard deviation of W	

δ	confidence level	$\delta \in (0, 1)$
t_ppf	probability point function of the t-test	
f_ppf	probability point function of the f-test	
df	degrees of freedom	
ν	optimal splitting percentage of W	$\nu \in (0, 1)$
ν_{split}	optimal splitting point of W	$\nu_{split} = \lfloor \nu \, \|W\| \rfloor$
ρ	robustness parameter	$\rho \; in(0, \infty)$
ρ_{temp}	temporary ρ, defined when $\|W\| < w_{proof}$	
w_{proof}	minimum size of W to identify drifts using ρ	
w_{min}	minimum W size	
w_{max}	maximum W size	$w_{max} \in [w_{min}, \infty)$
W_{hist}	historical errors from W	$W_{hist} = W_{0\,:\,\nu_{split}}$
W_{new}	recent errors from W	$W_{new} = W_{\nu_{split}\,:\,\|W\|-1}$
X	stream of real numbers r_i	$X = \langle r_1, \dots, r_i, \dots, r_\infty \rangle$
β	small constant to avoid division by 0	
α	level of significance	

Chapter 1

Introduction

1.1 Context and Motivation

Online Learning (OL) has emerged as a noteworthy sub-field within Machine Learning (ML), capturing increasing attention from both academia and industry. This specialized area of ML focuses on developing solutions that address problems by iteratively learning from individual data samples in a sequential manner (Hoi et al., 2021). The growing volume of generated data (Hariri, Fredericks, and Bowers, 2019) and advancements in technology over recent decades (Thompson, Ge, and Manso, 2022) have propelled OL to the forefront of research, particularly in scenarios where real-time responses are imperative and based on the latest available data samples. Notable real-world applications of OL include Fraud Detection (Paladini et al., 2023), Dynamic Pricing (Chen, Liu, and Hong, 2023), Stock Market Prediction (Tantisripreecha and Soonthomphisaj, 2018), Anomaly Detection (Rettig et al., 2019), and various others (Basterrech et al., 2023; Labrèche et al., 2022; Shahraki et al., 2022).

In contrast to traditional ML approaches, OL solutions dispense with the reliance on pre-defined datasets during training. Instead, they adapt to the dynamic nature of data streams, where individual data samples are continuously produced over time (Hoi et al., 2021). Formally, a data stream Ω is characterized by an ordered sequence of potentially infinite events e with corresponding timestamps ω expressed as $\Omega = (e_1, \omega_1), \ldots, (e_\infty, \omega_\infty)$, where ω_i denotes the processing time at which the event e_i is incorporated into the system.

Due to the distinctive nature of training data sources, we underscore two key differences between standard ML and OL settings. Firstly, in standard ML, it is customary to iterate over the same data sample multiple times during training, known as "epochs" (Goodfellow, Bengio, and Courville, 2016). Conversely, in OL, a data sample is processed a minimal number of times, ideally just once (Hoi et al., 2021). This divergence arises from the impracticality of revisiting each data sample multiple times in the context of potentially infinite data streams.

The second highlighted difference pertains to the presence of *concept drifts* in data streams, defined as unforeseeable changes in the statistical properties of incoming data over time (Lu et al., 2018). In standard ML, a pre-defined dataset with a fixed data distribution is employed. In contrast, OL operates without access to the entire training data in advance, and the data stream distribution may evolve over time. Therefore, OL solutions must adapt to these changes, ideally instantly (in real-time) (Hoi et al., 2021). In this thesis, the term "online training" refers to the incremental training of an ML model from single or small batches of data samples from a data stream, without the reliance on pre-existing datasets. Consequently, deploying a pre-trained ML solution in an OL setting is impractical due to its inability to be adapted online, leading to degraded performance in the face of concept drifts.

In addressing concept drifts, the model must ideally adapt in an online manner to mitigate the production of low-quality predictions or inferences after a concept drift. To facilitate online adaptation, conventional OL solutions often rely on computationally inexpensive ML models such as Naïve Bayes (Webb, 2010), Random Forests (Breiman, 2001), and others (Oza and Russell, 2001; Pelossof et al., 2009; Hulten, Spencer, and Domingos, 2001). However, these models, due to their relatively lower computational complexity, exhibit reduced performance when confronted with complex problems involving streams of audio, video, or other high-dimensional and intricate data (Najafabadi et al., 2015).

To tackle intricate challenges involving audio (Sinha, Awasthi, and Ajmera, 2020), video (Chai et al., 2021), and other multidimensional and highly complex data (Cottrell et al., 2012), researchers and developers commonly turn to Artificial Neural Networks (ANN). Despite being under study for decades, the exploration of ANN training in an Online Learning (OL) setting remains relatively limited. The inherent high computational complexity and substantial data requirements of ANN models usually necessitate offline training, with subsequent online usage for predictions or inferences (*Deep Learning on Flink* 2022; *Robust machine learning on streaming data using Kafka and Tensorflow-IO* 2022). However, this approach results in a delay in adapting the ANN model, leading to an increased number of low-quality predictions or inferences following the occurrence of concept drifts, as depicted in Figure 1.1.

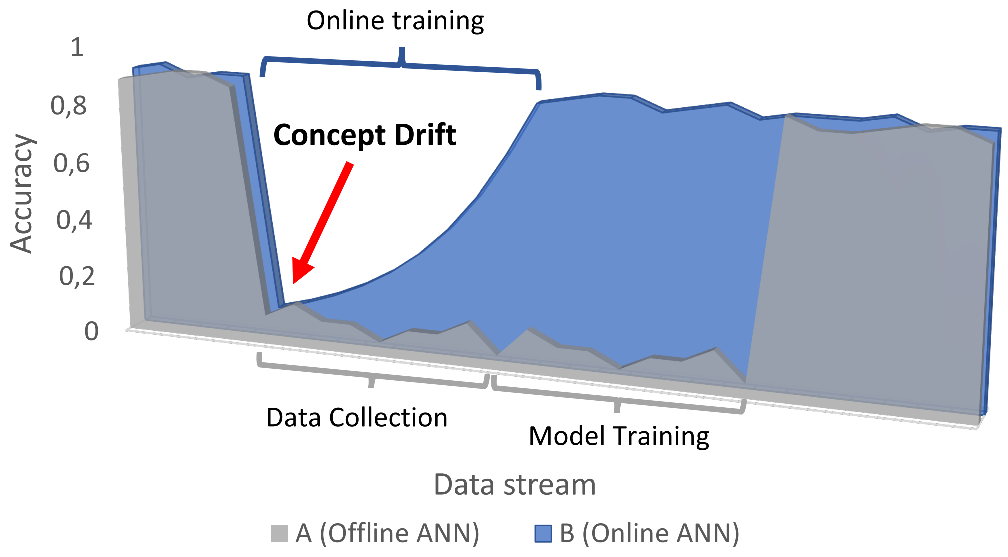


FIGURE 1.1: Illustration of the impact of delayed adaptation on the quality of predictions or inferences in an ANN model under concept drifts.

In Figure 1.1, the accuracy of Models A and B in an OL scenario is depicted. Both models maintain an average accuracy of 0.9 until a concept drift occurs. In this simplified example, both models promptly initiate their adaptation process at the moment of the concept drift. Model A starts by collecting data samples from the data stream until it forms a small dataset. Subsequently, Model A fine-tunes a copy of itself with the gathered small dataset and updates itself with the fine-tuned model. In contrast, Model B fine-tunes itself in an online manner, continually increasing its accuracy during the fine-tuning process. Consequently, Model B adapts to the concept drift faster by simultaneously performing data collection and fine-tuning. Moreover, during fine-tuning, Model B can provide better result estimates by training with the most recent data samples from the data stream.

Presently, the majority of OL research focuses on enhancing the quality of input data and adapting to concept drifts (Lu et al., 2018; Hu, Kantardzic, and Sethi, 2020; Priya and Uthra, 2021; Barros and Santos, 2018; Sethi and Kantardzic, 2017), with limited attention given to solving complex problems like those mentioned earlier. Addressing the challenge of training Deep Neural Network (DNN) models in an online setting, Online Sequential Extreme Learning Machine (OS-ELM) (Zhang, Tan, and Li, 2018) applies Extreme Learning Machines (ELM) models in the OL settings. ELMs are Feedforward Neural Networks with randomly assigned weights that only train their last layer via a linear solution (Wang et al., 2022). This characteristic allows ELMs to be fast to train compared to other ANN architectures while still producing satisfactory results. However, ELMs are known to face performance issues when dealing with complex problems as exemplified in (Nguyen et al., 2021b).

In contrast, (Basterrech et al., 2023) presented a use-case involving fake-news detection trained under an OL setting. They explored the online adaptation of an ANN model to enhance fake-news detection over a data stream continually experiencing concept drifts. Their study exemplifies a use-case that may not be solvable with classical OL solutions and could yield unsatisfactory results if not trained online, depending on the frequency and magnitude of the concept drifts. The primarily focus of their research is on addressing the continual learning problem, thus adapting to concept drifts without forgetting previous knowledge (Hadsell et al., 2020). However, they did not delve into the practical challenges of implementing their use-case in a real-world setting, which we believe might be unfeasible with current solutions.

This Ph.D. thesis contends that, given the current computational power and networking technologies, it is already feasible to train a broad range of ANN models online. By doing so, problems previously deemed unsolvable due to high complexity, time sensitivity, and concept drifts, become attainable.

1.2 Contextual Foundations

In this section, we provide an introductory overview of the two primary topics addressed in this thesis, setting the stage for a more detailed exploration in the subsequent chapters. Initially, we delve into the study of Artificial Neural Networks (ANNs), briefly touching upon several key concepts integral to this thesis, including the distribution of ANNs and their optimization strategies. These areas will be examined in greater detail later in the thesis (cf. Chapters 3 and 5). Following this introduction to ANNs, we offer a concise explanation of Concept Drifts, focusing on their identification and adaptation within the context of online learning scenarios. Chapter 4 will expand upon these introductory remarks, providing a comprehensive analysis of Concept Drifts and their implications in the field of Machine Learning.

1.2.1 Artificial Neural Networks (ANNs)

ANNs, prominent in the field of Machine Learning, have been studied since the 1940s (McCulloch and Pitts, 1943), drawing inspiration from the processing mechanisms of biological neural networks. These networks process information through a complex array of interconnected neurons using electrical signals (Hassabis et al., 2017). ANNs, particularly in the last decades, have seen a surge in interest from both academia and industry, attributed to their excellence in handling non-linear, multidimensional problems (LeCun, Bengio, and Hinton, 2015).

At their core, ANNs comprise nodes or "neurons" interconnected by "edges". These neurons are organized into layers, with each layer serving a specific function in the network. The architecture begins with an input layer, progresses through one or more hidden layers, and concludes with an output layer. As the network operates, signals propagate from the input layer through the hidden layers to the output layer (LeCun, Bengio, and Hinton, 2015).

The training of ANNs involves optimizing model parameters (denoted as x) to closely mirror the problem's actual data distribution. This optimization, aimed at minimizing the model's error or loss function $F(x)$ against the training data, is typically achieved through an iterative process. The method of back-propagation using Stochastic Gradient Descent (SGD) is commonly employed, represented as $x^{t+1} = x^t - \eta \nabla F(x^t, \xi_i)$. Here, x^t signifies the model parameters at iteration t, η is a hyperparameter dictating the stepsize, and $\nabla F(x^t, \xi_i)$ is the gradient of the loss function concerning the weights and the ith training sample. Importantly, each training sample, ξ, comprises a pair (z_i, y_i), where z_i is the training example and y_i is its corresponding label (Goodfellow, Bengio, and Courville, 2016).

The effectiveness of an ANN in representing the data distribution of a problem is directly influenced by its architecture and the problem's complexity. Complex problems, characterized by intricate data patterns, non-linearity, and high-dimensional data spaces, often necessitate a Deep Neural Network (DNN) architecture (Goodfellow, Bengio, and Courville, 2016). DNNs, with their multiple hidden layers, have demonstrated superior performance in tackling intricate problems, as those involving audio, video, and natural language processing. Advancements in ANN architectures, including LSTM (Hochreiter and Schmidhuber, 1997), CNN (LeCun et al., 1989), GANs (Goodfellow et al., 2014), and Transformers (Vaswani et al., 2017), have led to significant breakthroughs across various areas, such as natural language processing (Brown et al., 2020), image classification and generation (Krizhevsky, Sutskever, and Hinton, 2012; Dhariwal and Nichol, 2021), and audio processing (Oord et al., 2016). However, training large-scale models requires substantial time and computational steps (Strubell, Ganesh, and McCallum, 2019).

To reduce the training time, researchers have explored several strategies, including Network Pruning (Liu et al., 2018), Batch Normalization (Ioffe and Szegedy, 2015), and Weight Initialization (Kumar, 2017). Additionally, distributing the computation across multiple nodes is a common approach to expedite training. This distribution can be achieved through either model parallelism or data parallelism.

In *model parallelism*, the network architecture is divided among various nodes, each responsible for a segment of the back-propagation process. This method is essential for extremely large models that cannot be accommodated in a single node's memory. However, determining the optimal model partitioning across nodes can be challenging and not feasible in an use-case independent manner (Ouyang et al., 2021). *Data parallelism*, on the other hand, involves initializing distributed models with identical parameters but training them with distinct datasets. During training, these models exchange learned insights to ensure convergence. This method is preferred when the ANN model is small enough to fit within a single node's memory (Ouyang et al., 2021). In this thesis, we delve deeper into data parallelism.

Models in a distributed ANN setup can communicate using either synchronous or asynchronous methods. In a *synchronous setting*, each worker node aligns its computations with every iteration cycle. This method ensures a more uniform and stable convergence of the model, as all nodes update their parameters simultaneously.

However, this synchronized approach can lead to inefficiencies, particularly when faster nodes are forced to remain idle, waiting for the slower ones to complete their part of the computation and the communication process. Additionally, it is also possible to adopt stale-synchronous communication, in which worker nodes align their computations every τ_{max} iterations, which can mitigate the idle time from slower nodes but does not address the idle time from blocking communication processes (Ben-Nun and Hoefler, 2019).

Conversely, in an *asynchronous setting*, worker nodes operate using Asynchronous Stochastic Gradient Descent (ASGD). This method allows each node to proceed with its computations independently, without having to wait for others. Such a setup ensures that faster nodes are not hindered by slower ones nor by communication delays, thereby potentially reducing overall computation time. However, this independence comes with a trade-off. Without the synchronization checkpoints, distributed models might experience slight divergences in their learning paths during optimization phases. Consequently, the model parameters' update in such a setting, particularly in centralized architectures, is represented as $x^{t+1} = x^t - \eta \nabla F(x^{t-\tau}, \xi_i)$. Here, x^t denotes the parameter weights at iteration t, and the gradient used for the update is computed from the model's state at a previous time $t - \tau$, with τ indicating the delay in the gradient application (Ben-Nun and Hoefler, 2019). This delay in gradient application complicates the proof of mathematical convergence for ASGD. Practically, it might lead to less optimal convergence of the model, as the updates are based on slightly outdated information.

The architecture of the network in a data-parallel approach is also a crucial consideration. In a *centralized architecture*, distributed models rely on a central parameter server for updating their parameters. While efficient, this can create bottlenecks due to excessive reliance on the central server (Mayer and Jacobsen, 2020). Conversely, a *decentralized architecture* allows for direct communication between distributed models, eliminating dependency on a central server. While this approach reduces potential bottlenecks, it introduces complexities in ensuring effective and timely communication among all nodes (Mayer and Jacobsen, 2020). Due to this complication, in decentralized architectures it is common to adopt synchronous or stale-synchronous communication. Federated Learning (FL) is a notable application of decentralized SGD, where models are trained without sharing local data to preserve privacy (Nguyen et al., 2022).

1.2.2 Concept Drifts

Concept drift refers to the phenomenon where the statistical properties of target data evolve unpredictably over time. This is particularly prevalent in OL environments where data is continuously received in the form of streams (Lu et al., 2018). A typical real-world example is in Fraud Detection (Dal Pozzolo et al., 2015), where fraudsters constantly modify their tactics to evade detection systems. Consequently, these systems must be adept at recognizing and adapting to these evolving strategies (concept drift identification and adaptation) to effectively counteract fraudulent activities.

From a mathematical standpoint, consider a dataset $\Xi_{0,d} = \xi_0, ..., \xi_d$ spanning the time period $[0, d]$, where each sample ξ_i comprises a pair $\xi_i = (z_i, y_i)$ of features z_i and corresponding label y_i. Suppose $\Xi_{0,d}$ follows the distribution $\mathcal{D}_{0,d}(z, y)$. A concept drift is said to occur at time t when there is a change in the joint probability distribution of z and y, signified by $\mathcal{D}_{0,t}(z, y) \neq \mathcal{D}_{t+1,\infty}(z, y)$ or $\exists t$, $P_t(z, y) \neq P_{t+1}(z, y)$ (Lu et al., 2018).

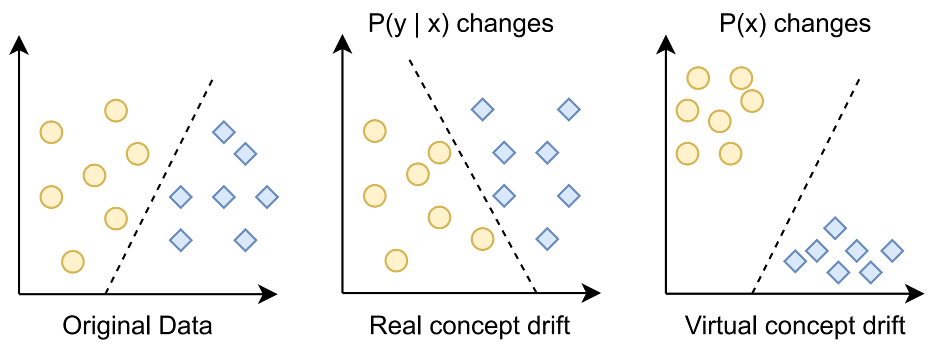


FIGURE 1.2: Sources of concept drift, in which different formats and colors represent different classes.

Diving deeper into concept drift types, we can decompose the disparity $P_t(z, y) \neq P_{t+1}(z, y)$ as $P_t(z, y) = P_t(z) \times P_t(z|y)$ and discern two primary concept drift sources. The first is *real concept drift*, where $P_t(z|y) \neq P_{t+1}(z|y)$. This type of drift alters the decision boundaries of ML algorithms, potentially reducing their accuracy. The second type is *virtual concept drift*, signified by $P_t(z) \neq P_{t+1}(z)$, which does not affect decision boundaries and thus does not impact accuracy. It is also possible for both types of drifts to occur simultaneously (Gama et al., 2014). The different sources of the concept drift can be seen in Figure 1.2.

Identifying concept drifts is achieved through concept drift detectors, which are broadly categorized into distribution-based and error-based types. *Distribution-based detectors* monitor the variance between historical and new data distributions, enabling the detection of both real and virtual drifts. However, this approach is computationally intensive due to the need to process the entire feature space of z. *Error-based detectors*, conversely, track changes in the error rate of a ML algorithm. These detectors simplify drift detection by focusing on a single variable, such as the loss of an ANN model, but are ineffective in identifying virtual drifts (Ben-Nun and Hoefler, 2019).

Finally, concept drift adaptation can be approached either passively or actively. *Passive adaptation* involves continuous training of the ML algorithm, irrespective of concept drift occurrences. While this simplifies the OL process, it can lead to resource wastage if the algorithm has already converged or no concept drift has occurred. *Active adaptation*, on the other hand, involves retraining the ML algorithm only when a concept drift is detected, aiming to conserve resources and tailor the training to the specific nature of the identified drift (Lu et al., 2018).

1.3 Research Problem and Challenges

To the best of our knowledge, the online training of Artificial Neural Network (ANN) models under an Online Learning (OL) setting is an open research problem. This complex challenge spans multiple domains, including Optimization, Distributed Computing, Big Data, Data Science, Machine Learning, and High-Performance Computing (HPC) systems. In light of these considerations, the primary objective of this Ph.D. thesis is to address the following overarching question:

> *How can a distributed system be designed to facilitate online training and prediction on ANN models using incoming data streams as input in a scalable*

manner, with a specific focus on addressing concept drift in an Online Learning setting?

To tackle this interdisciplinary challenge, we investigate the following sub-problems, integrating their solutions into a cohesive and comprehensive approach:

1. **How can a system be designed to enable online training and prediction on ANN models using incoming data streams as input?**

   Training an ANN model with data samples arriving from a data stream poses a unique set of challenges. Unlike traditional training scenarios where multiple epochs over a pre-defined dataset lead to convergence, training from data streams lacks access to pre-defined data and prohibits the reiteration over the same data sample multiple times. This distinction significantly impacts two crucial steps in the training pipeline:

   - Data Pre-processing: The absence of access to the full training data complicates standard tasks essential for training improvement, such as data normalization, noise reduction, undersampling, and oversampling. The real-time nature of data stream processing necessitates adaptations to these pre-processing tasks.
   - Data Management: Handling data is straightforward when dealing with pre-defined data that can be allocated and distributed before the start of the training process. However, in online training from data streams, each data sample needs to be allocated and distributed iteratively. This presents a challenge to ensure low computational, network, and memory impact while maintaining efficiency.

   The development of a framework capable of training ANN models online, in a scalable manner, and not dependent on specific use-cases remains an open research problem. Addressing this challenge involves designing a dataflow that seamlessly integrates with the dynamic nature of data streams, allowing for efficient pre-processing, distribution, and management of data samples for continuous online training. This framework should be versatile enough to accommodate various ANN architectures and training strategies, making it a valuable contribution to the broader field of online machine learning.

2. **How can the training efficiency of ANN models be improved when utilizing incoming data streams as input, considering diverse model types and use-cases?**

   Training ANN models typically involves Stochastic Gradient Descent (SGD), a process known for its time-consuming nature. However, the time-sensitivity inherent in OL tasks necessitates reducing the time it takes for ANN models to converge. Achieving this reduction involves various integrated or mutually exclusive approaches. Many existing approaches such as Network Pruning, Batch Normalization, and Weight Initialization excel in decreasing the training time of ANN models. However, these approaches are often designed for specific model architectures, making it challenging to translate them into a general, use-case-independent setting.

   Differently, the distributed training across multiple computing nodes may be performed in an use-case-independent manner. For relatively smaller models that fit into memory, data parallelism is often employed. Notably, larger models, exceeding memory capacity, are not the focus of this thesis, as their high complexity tends to result in training times ranging from hours to weeks, making them unsuitable for online training. To further speed up the training of ANN models, one could additionally explore ASGD or decentralized architectures in the OL setting.

However, exclusive reliance on ASGD may still introduce computational or communication bottlenecks from the parameter server. Conversely, when relying solely on decentralized SGD, faster workers may experience wait times for slower ones to complete their computations. Hence, there is a need for further research on scaling ANN model training without use-case-dependent solutions or bottlenecks stemming from centralized communication and synchronous implementation wait times. Addressing this challenge could be an important step to develop an universal and efficient solution applicable to diverse ANN architectures and OL use-cases.

3. **How can the high false-positive rate of existing concept-drift detectors be improved to make them reliable for triggering retraining of ANN models in an OL setting?**

   In OL scenarios, preventing indefinite retraining of ML models commonly relies on concept drift detectors (Lu et al., 2018).

   While error-based drift detectors have been studied extensively, many are tailored for classification problems and struggle to detect concept drifts in a stream of real numbers (e.g., the loss of an ANN model) (Bayram, Ahmed, and Kassler, 2022). Among detectors suitable for regression problems, the commonly used ones employ a sliding window W, containing error rates, which is divided into sub-windows. A statistical test is then applied to determine if these sub-windows are statistically different, flagging a concept drift. However, determining the optimal division point for the sliding window remains a challenge. Current approaches either involve multiple divisions (e.g., $\log |W|$ times) or rely on a pre-defined division, negatively impacting drift identification performance.

   In addition to this, despite their excellent recall, existing drift detectors often yield a high number of false positives (Bayram, Ahmed, and Kassler, 2022). Consequently, if retraining an ANN model is triggered by a concept drift detector, a substantial number of false positives would significantly impact the unnecessary time and computational resources spent on retraining an ANN model that did not experience any concept drift.

   Therefore, the development of error-based drift detectors tailored for regression problems that produce low false positive rates is an open research problem (Bayram, Ahmed, and Kassler, 2022). Addressing this challenge is essential for optimizing the utilization of computational resources and preventing unnecessary retraining of ANN models that have already converged and have not experienced concept drift.

4. **How does the asynchronous and decentralized distribution of Stochastic Gradient Descent (SGD) affect its convergence rate?**

   The convergence rate of SGD under both asynchronous and decentralized distributions remains an area requiring further investigation. The primary challenge lies in determining the difference between two models at different time points during training. Given models x_i and x_j at times t and s respectively, measuring $x_i^t - x_j^s$ is crucial to calculate the noise introduced by the asynchrony among decentralized models and to assess its impact on the expected convergence rate of the optimized function.

Many authors rely on assumptions about the optimized function that may not be realistic for most ANN models. Commonly, these assumptions include convexity or strong convexity (Recht et al., 2011; Stich, 2019; Srivastava and Nedic, 2011), which are often not the case of the functions optimized by ANN models (Danilova et al., 2022). Other authors prove the convergence of asynchronous and decentralized SGD under an infinite number of iterations (Srivastava and Nedic, 2011; Ram, Nedić, and Veeravalli, 2009), providing no specific convergence bound. Some studies focus on specific network topologies (Lian et al., 2018) or address Federated Learning (FL) problems (Nguyen et al., 2022), which prioritize privacy over computational performance and often rely on sparse network topologies, leading to higher message exchange delays. Moreover, commonly used communication strategies in decentralized SGD, such as the Gossip strategy (Lian et al., 2017; Koloskova, Stich, and Jaggi, 2019; Koloskova et al., 2020), involve nodes communicating only with immediate neighbors, propagating messages over the network in subsequent iterations. While this mitigates network restrictions, it negatively impacts model convergence due to the high number of iterations required for messages to reach all nodes (Ben-Nun and Hoefler, 2019).

The development, convergence rate proof, and experimentation of a Decentralized and Asynchronous SGD (DASGD) algorithm that leverages a fully-connected network is non-trivial. Overcoming these challenges is essential for understanding the implications of asynchronous and decentralized distributions on the convergence rate of SGD, particularly in the context of ANN models, and can contribute to the development of more efficient and effective distributed training algorithms.

1.4 Research Goal and Approach

The overarching goal of this Ph.D. thesis is to design a system that facilitates online training and prediction on ANN models using incoming data streams as input, with a specific emphasis on addressing concept drift in an OL setting. The key objectives defined to achieve this goal were structured as illustrated in Figure 1.3 and defined below.

1. **Design a dataflow architecture that enables online training and prediction on ANN models using incoming data streams.**

   The initial phase of the investigation involved an in-depth study of prevalent stream processing engines, particularly Apache Spark (Zaharia et al., 2016) and Apache Flink (Carbone et al., 2015), widely utilized in processing data streams for OL problems. Acknowledging their reliance on centralized architectures for data management and fault tolerance, we recognized potential bottlenecks as the throughput of data streams increases.

   Subsequently, our attention shifted to the novel stream processing engine AIR (Venugopal et al., 2020), which uniquely adopts a fully asynchronous and decentralized architecture. Notably, AIR demonstrated superior performance,

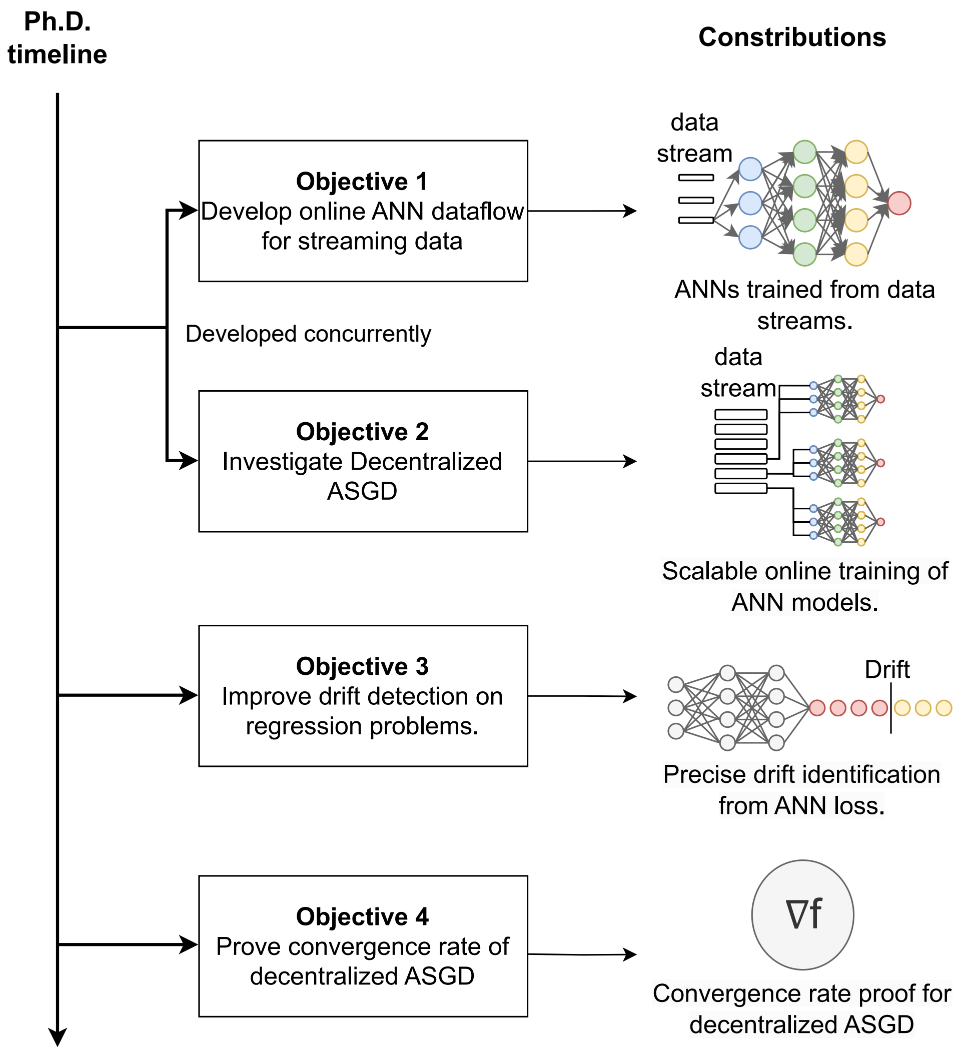

Ph.D. timeline
Constributions
Objective 1
Develop online ANN dataflow for streaming data
data stream
ANNs trained from data streams.
Developed concurrently
Objective 2
Investigate Decentralized ASGD
data stream
Scalable online training of ANN models.
Objective 3
Improve drift detection on regression problems.
Drift
Precise drift identification from ANN loss.
Objective 4
Prove convergence rate of decentralized ASGD
∇f
Convergence rate proof for decentralized ASGD

FIGURE 1.3: Methodology to complete key objective of this thesis, leading to the design of a system for effectively employing ANN training into an Online Learning pipeline.

sustaining up to 15 times more throughput than traditional engines like Flink and Spark (Venugopal et al., 2022).

Chapter 3 delve into the integration of AIR with the training pipeline of an ANN model. We observed the compatibility between the iterative strategy of SGD used in ANN training and the incremental nature of data streams, producing either single data samples or small batches at each instance. The conjecture was that online training of ANN models could be achieved by utilizing these data stream samples.

However, a challenge arose due to the supervised nature of ANN training, where training samples typically consist of pairs (z_i, y_i) representing data (z_i) and corresponding labels (y_i). Data streams, in contrast, primarily provide data samples without explicit labels. To address this, we focused on the online training of self-supervised ANN models, which can derive labels from the provided data samples or other related samples (Liu et al., 2021).

In summary, the investigation into the dataflow architecture involved understanding the limitations of centralized stream processing engines, exploring the potential of a decentralized and asynchronous engine like AIR, and adapting the training pipeline of ANN models to align with the characteristics of data streams. The emphasis on self-supervised learning offers a promising avenue for addressing label-related challenges in the online training of ANN models within an OL setting.

2. **Investigate distribution strategies to scale out the training of ANN models, optimizing their performance by leveraging the asynchronous and decentralized characteristics of our proposed dataflow architecture.**

   In this phase of the research, we explored the potential of harnessing the asynchronous and decentralized characteristics of the AIR dataflow architecture to accelerate the training process of ANN models. Our investigations revealed a gap in the literature, as only a few studies had delved into the asynchronous and decentralized variants of SGD. Furthermore, most of these studies were centered around Federated Learning (FL) problems, which commonly occur in scenarios like the Internet of Things (IoT) (Nguyen et al., 2021a), mobile devices (Kang et al., 2020), and Edge Computing (Abreha, Hayajneh, and Serhani, 2022). These scenarios often exhibit sparse network connectivity and high latency, necessitating centralized architectures or Gossip communication as alternative (Hegedűs, Danner, and Jelasity, 2021).

   Given the absence of solutions leveraging a fully connected network, typical in Cloud and High-Performance Computing (HPC) settings, for distributing the training of SGD in an asynchronous and decentralized manner, we embarked on developing our own solution.

   Chapter 2 presents an empirical exploration of the asynchronous and decentralized implementation of SGD. A key insight emerged: if a group of models initialized with the same weights applies a common set of gradients independently of the order and values of those gradients, the models will remain consistent among themselves. This consistency arises from the commutative nature of subtractions during the gradient application process. Specifically, for an ANN model x^t at time t, the update formula is $x^{t+1} = x^t - \eta \nabla F(x^t, \xi)$, where η is the learning rate and ξ represents the data sample. Consequently, models initialized with the same weights and applying the same set of gradients will converge to the same values.

   Furthermore, our hypothesis posited that with frequent communication among models, the set of gradients applied by each model throughout training would be similar. Therefore, if model x_a is similar to x_b, their gradients should also exhibit similarity, i.e., $\nabla F(x_a, \xi) \simeq \nabla F(x_b, \xi)$. If this holds true, even if x_a applies a gradient calculated by x_b, the small difference between the gradients should have a minimal impact on the convergence rate of decentralized and asynchronous SGD.

   In summary, this investigation focused on developing a distribution strategy that capitalizes on the asynchronous and decentralized characteristics of the proposed AIR dataflow architecture to enhance the performance and scalability of ANN model training. The empirical exploration provided valuable insights into the consistency of models and the potential benefits of leveraging a fully connected network for asynchronous and decentralized SGD in Cloud and HPC settings.

3. **Develop an error-based drift detector algorithm specifically designed for regression and classification problems, aiming to achieve a smaller false positive rate than existing baselines while maintaining an equivalent level of recall.**

   Existing error-based drift detectors suitable for both classification and regression problems often rely on monitoring statistical differences between the means of current (W_{new}) and historical (W_{hist}) errors of a ML algorithm (Lu et al., 2018). Typically, if the mean error surpasses a predetermined threshold, a concept drift is signaled. However, we identified a limitation in this approach, where changes in the standard deviation of error may go unnoticed, potentially leading to undetected concept drifts.

   To illustrate this concern, consider two sets of errors, W_{hist} and W_{new}:

   $$W_{hist} = \langle 0.3; 0.7; 0.7; 0.3; 0.3; 0.7; 0.3; 0.7 \rangle$$
   $$W_{new} = \langle 0.0; 1.0; 1.0; 0.0; 1.0; 0.0; 0.0; 1.0 \rangle$$

   Current drift detectors may fail to identify a concept drift because the means of W_{hist} and W_{new} are both 0.5, and there is no statistical difference between them. However, we hypothesize that a concept drift could be responsible for the change in the standard deviation of the ML algorithm's error.

   In Chapter 4, our investigation focuses on developing an algorithm that considers changes in both the mean and the standard deviation of ML algorithm errors to detect concept drifts. We proposed a novel approach combining two statistical tests, the t-test and the f-test, to identify drifts caused by differences in the mean and standard deviation of errors, respectively. Additionally, we explored the feasibility of automatically detecting the optimal splitting point of W into W_{hist} and W_{new} by combining both statistical tests. This enhancement would significantly improve the current splitting point detection complexity from $\mathcal{O}(\log |W|)$ (Bifet and Gavalda, 2007) to $\mathcal{O}(1)$.

   The ultimate goal of this investigation is to develop a drift detection algorithm that offers a smaller false positive rate compared to existing baselines while maintaining an equivalent level of recall. By addressing the limitations of current approaches and incorporating both mean and standard deviation considerations, the proposed algorithm aims to enhance the accuracy and reliability of concept drift detection in both regression and classification problems.

4. **Formulate a rigorous mathematical proof establishing the convergence rate of decentralized and asynchronous Stochastic Gradient Descent (SGD), contributing to a theoretical understanding of their convergence properties.**

   In this phase of the research, the objective was to develop a rigorous mathematical proof to establish the convergence rate of decentralized and asynchronous SGD. Existing convergence rate proofs for decentralized and asynchronous SGD often rely on gossip communication between distributed models (Nadiradze et al., 2021). However, the empirical investigations in the previous stages of the research indicated a potential for almost linear speed-up in the convergence rate when gradients are broadcasted to other models instead of being propagated via gossip.

   Inspired by the asynchronous SGD (ASGD) convergence rate proofs developed by (Koloskova, Stich, and Jaggi, 2022), the goal was to explore the feasibility of extending their proof to account for the case where there is no parameter server, and communication is decentralized.

In Chapter 5, the approach considers the convergence rate of each decentralized model separately, treating it as if it were the parameter server in its own centralized approach. This allowed the construction of a convergence proof analogous to the one presented in (Koloskova, Stich, and Jaggi, 2022). The key insight in this decentralized setting was to estimate the difference $x_i^t - x_j^s$ based on the difference in the sets of gradients applied by each model.

For example, given models x_i and x_j, both initialized with the same weights, and considering that x_i applied to itself a set of gradients $G_i = \nabla_a, \nabla_b, \nabla_c$ while x_j applied $G_j = \nabla_a, \nabla_c, \nabla_d$, the difference between those models was determined to be smaller or equal than the difference between the sets of gradients G_i and G_j. This insight was formalized as $x_i^t - x_j^s \leq |\nabla_b| + |\nabla_d|$. Leveraging this observation, the convergence rate proof for decentralized and asynchronous SGD (DASGD) was further elaborated.

This theoretical contribution aimed to provide a deep understanding of the convergence properties of decentralized and asynchronous SGD, shedding light on their efficiency and speed-up potential. The mathematical proof contributed to the foundational knowledge of these optimization algorithms, paving the way for their more informed and confident application in various contexts.

By achieving these key objectives, the thesis aims not only to advance the theoretical understanding of online training and prediction for ANN models in OL settings but also to provide practical tools and solutions that empower developers to address complex and time-sensitive problems previously deemed challenging with existing OL approaches. Ultimately, the goal is to contribute to the broader field of online machine learning and enable the application of ANN models in a wide range of dynamic and evolving scenarios.

1.5 Contributions

The main contributions of this Ph.D. thesis are summarized as follows:

1. TensAIR Framework (Chapter 3), TensAIR is a framework designed for the online training and prediction of ANN models using data streams. Its key feature is a fully asynchronous and decentralized architecture that enables distributed training across multiple nodes, with or without GPUs. TensAIR was compared with state-of-the-art approaches for employing ANN models in an Online Learning (OL) setting, demonstrating sustainable throughput improvements ranging from 6 to 116 times over available baselines.

2. OPTWIN Drift Detector (Chapter 4), OPTWIN is an error-based drift detector designed for classification and regression problems, featuring a low false positive rate. It identifies concept drifts by considering statistical changes in both the mean and standard deviation of errors from a ML algorithm. OPTWIN optimally divides a sliding window of errors in constant time, improving splitting point detection. The detector was evaluated against baselines on 12 real-world and synthetic datasets, demonstrating superior F1-scores and saving 21% of time compared to the baseline when retraining an ANN model based on flagged concept drifts.

3. DASGD Algorithm (Chapter 5), DASGD is an algorithm for training ANN models in an asynchronous and decentralized manner, accompanied by proven convergence bounds. It introduces a novel *staleness* metric S to quantify dissimilarity between decentralized models at different time points. DASGD was proven to converge under mild assumptions when optimizing non-convex functions with or without bonded gradients. Specifically, DASGD converges in $\mathcal{O}(\sigma\epsilon^{-2}) + \mathcal{O}(QS_{avg}\epsilon^{\frac{-3}{2}}) + \mathcal{O}(S_{avg}\epsilon^{-1})$ iterations when gradients are bounded by Q and $\mathcal{O}(\sigma^2\epsilon^{-2}) + \mathcal{O}(\sqrt{\hat{S}_{avg}\hat{S}_{max}}\epsilon^{-1})$ iterations when the gradients are not bounded. We note that when gradients are bounded, DASGD eliminates the convergence dependency on the maximum *staleness*. The algorithm demonstrated excellent scale-out performance across various models, including LSTM, Auto Encoders, CNN, and Skip-gram, and performed well on multiple datasets.

4. TensAIR Library[1] (Chapters 3 and 6), The TensAIR library is the implementation of the TensAIR framework, providing a technological contribution for online training and prediction of ANN models in an asynchronous and decentralized manner using data streams. In addition, TensAIR also makes it possible to identify concept drifts using our OPTWIN drift detector and adapt the trained model accordingly. The library extends the AIR data-stream processing engine and supports training and predicting on TensorFlow (Abadi et al., 2016) models. Implemented in C++ and utilizing MPI for communication, TensAIR includes a Python interface for ease of use and real-time visualization of the training process.

These contributions collectively address the challenges associated with online training of ANN models in OL settings, providing tools and insights to enable the resolution of complex problems that were once considered unfeasible under traditional OL approaches.

1.6 Organization

This doctoral thesis is organized as a “cumulative thesis”, in which the following chapters are reprints of academic articles published or under review for publication, as described below.

Chapter 2 corresponds to the extended abstract “Convergence time analysis of Asynchronous Distributed Artificial Neural Networks” (Tosi, Venugopal, and Theobald, 2022), which was published and presented on the 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD). It presents our exploratory analysis on the behavior of ANN models when trained under an asynchronous and decentralized setting.

Chapter 3 corresponds to the paper “TensAIR: Online Learning from Data Streams via Asynchronous Iterative Routing” (Tosi, Venugopal, and Theobald, 2024), which was published and presented on the 8th International Conference on Machine Learning and Soft Computing (ICMLSC 2024). It introduces the TensAIR framework, which enables the asynchronous and decentralized training of ANN models from data streams in real-time.

Chapter 4 corresponds to the extended version of the paper “OPTWIN: Drift Identification with Optimal Sub-Windows” (Tosi and Theobald, 2024), which will be

[1] `https://github.com/maurodlt/tensair`

presented on the 3rd International Workshop on Databases and Machine Learning (DBML 2024) colocated with the 40th IEEE International Conference on Data Engineering (ICDE 2024). This work proposes OPTWIN, an error-based drift detector algorithm that detects concept drifts from classification and regression problems while producing a low false positive rate.

Chapter 5 corresponds to the paper "Convergence Analysis of Decentralized ASGD" (Tosi and Theobald, 2023), which is under review for publication. This paper presents our formal convergence rate analysis of the decentralized and asynchronous SGD strategy adopted by TensAIR.

Chapter 6 corresponds to the demonstration paper "TensAIR: Real-Time Training of Distributed Artificial Neural Networks" (Tosi and Theobald, n.d.), which is under review for publication. This paper demonstrates how to use the TensAIR's Python API to adapt an ANN model in an OL setting when concept drifts are detected by OPTWIN.

Chapter 7 contains a detailed discussion over each research object covered in this thesis. In addition, we discuss about the limitations of each of our proposed approaches and the thesis as a whole. At last, we summarize our findings and present how those are used to answer the main objective of this thesis.

Chapter 8 concludes by presenting a summary of the contributions we produced in this doctoral thesis and how they can be extended in future works. It additionally lists the methods used to disseminate our findings. At last, it concludes this thesis with our final considerations.

1.7 Disclaimer

During the preparation of this thesis, the AI-based language model ChatGPT (OpenAI, 2023) (versions 3.5 and 4.0) was employed as a tool to assist in enhancing the grammar, structure, and clarity of the text. This usage was specifically confined to the "Abstract" and Chapters 1, 6, 7, and 8. For each chapter, the original text was input into ChatGPT with the prompt: "Considering a PhD thesis, how can the following section of the chapter be improved for better clarity and coherence with minimal interference?" The section in question was then provided for review.

It is important to note that while ChatGPT contributed to the linguistic refinement of these chapters, it did not partake in any intellectual or conceptual development of the thesis. The ideas, arguments, and conclusions presented remain entirely those of the author. Each suggestion provided by ChatGPT was rigorously evaluated and, where necessary, adapted or discarded to ensure that the final content accurately represents the author's original thoughts and adheres to academic standards.

In utilizing ChatGPT, we have been mindful of the ethical implications and potential limitations associated with AI tools in academic writing. We have taken diligent care to mitigate any inaccuracies or biases that might arise from the use of this technology, maintaining the integrity and originality of the scholarly work.

Chapter 2

Convergence Time Analysis of Asynchronous Distributed Artificial Neural Networks

2.1 Contribution Statement

This chapter corresponds to the reprint of the extended abstract “Convergence time analysis of Asynchronous Distributed Artificial Neural Networks” (Tosi, Venugopal, and Theobald, 2022) authored by Mauro Dalle Lucca Tosi, Vinu Ellampallil Venugopal, and Martin Theobald. Below, we provide the CRediT (Contributor Roles Taxonomy) [1] statements correspondent to each author contribution.

- **Mauro Dalle Lucca Tosi:**
  - Conceptualization;
  - Methodology;
  - Software;
  - Validation;
  - Formal analysis;
  - Investigation;
  - Data Curation;
  - Writing - Original Draft;
  - Visualization;
- **Vinu Ellampallil Venugopal**
  - Conceptualization;
  - Formal analysis;
  - Writing - Review & Editing;
- **Martin Theobald**
  - Conceptualization;
  - Resources;
  - Writing - Review & Editing;
  - Supervision;
  - Project administration;
  - Funding acquisition

[1] https://credit.niso.org/

2.2 Chapter Contextualization

This chapter corresponds to the extended abstract "Convergence time analysis of Asynchronous Distributed Artificial Neural Networks" (Tosi, Venugopal, and Theobald, 2022). It presents our first exploratory analysis on the behavior of ANN models when trained under an asynchronous and decentralized setting. In this chapter, we start our research on scaling ANN model training without bottlenecks stemming from centralized communication and synchronous implementation wait times. Additionally, this chapter presents our first formal definition of staleness, which later served as key metric for our convergence proof of Decentralized and Asynchronous Stochastic Gradient (DASGD) in Chapter 5. Addressing the challenge of reducing bottlenecks from synchronization and centralization is crucial for developing a universal and efficient solution to scale-out diverse ANN architectures and OL use cases.

2.3 Introduction

Artificial Neural Networks (ANNs) have had tremendous success in the recent past, in particular, due to their capability to solve complex problems. Those problems are generally solved using large models coupled with a high volume of training data. Researchers have achieved excellent speedups on the training of large models by considering data parallelism, where a subset of the training data is distributed among multiple compute nodes. However, common model optimizers as Stochastic Gradient Descent (SGD) are sequential in nature. Prior approaches such as (Sergeev and Del Balso, 2018) perform a *synchronous* parallel computation of SGD on large mini-batches. Nevertheless, a larger mini-batch size can negatively impact the model's ability of generalization (Keskar et al., 2017); and the synchronous SGD distribution increases the idle time of the compute nodes due to the necessity of synchronization barriers (Recht et al., 2011).

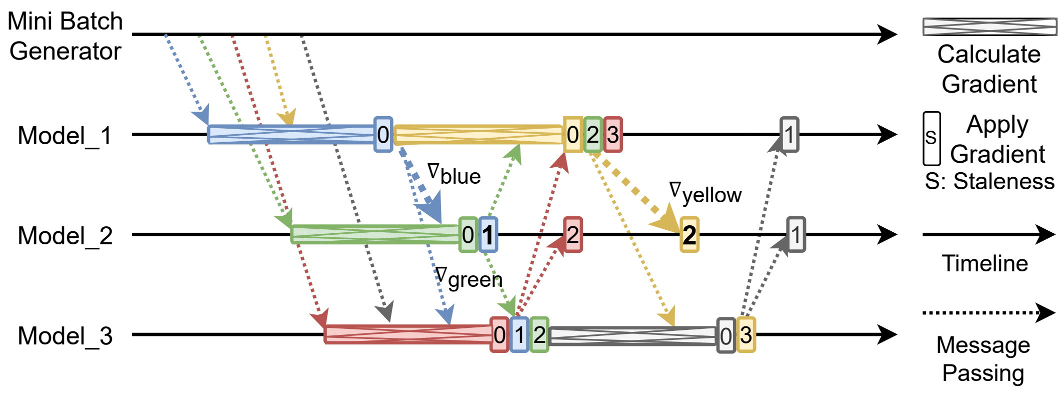


FIGURE 2.1: **Asynchronous SGD distribution.**

Another line of research in this direction is to explore *asynchronous* approaches where nodes would asynchronously update themselves, ignoring the potential *staleness* of their updates. Following this notion, Niu *et al.* (Recht et al., 2011) developed Hogwild, which used asynchronous application of SGD on multicore machines. Later, Zhang, Hsieh, and Akella (Zhang, Hsieh, and Akella, 2016) developed Hogwild++, which explores how to overcome Hogwild's bottleneck of having all worker nodes reading/updating a single central weight matrix. However, these approaches achieve a nearly optimal rate of convergence only when the associated optimization problem is *sparse*. In addition, the impact of updating an ANN with stale gradients on the model's convergence time and loss value is not well studied experimentally.

In this paper, we consider a distributed asynchronous architecture for training an ANN model that helps us to systematically study the *asynchronicity* and *staleness* notions effectively. This setting allows models to learn from sharded input data simultaneously. It also lets them loosely synchronise themselves without synchronisation barriers, making the training process much faster than the synchronous approach. However, theoretically, the convergence of a model in an asynchronous SGD approach depends on the staleness of model updates. Within the scope of this paper, we define *staleness* in this setting. Then we analyse its impact on the convergence time and loss value of the model and report our observations on this topic based on our experiments with Word2Vec.

2.4 Asynchronicity and Staleness

We perform the asynchronous distributed training of an ANN model by training n models with different batches of data across N computing nodes. Each model represented by a weight matrix x, would asynchronously exchange the gradient $\nabla_{x,\xi}$, calculated from the batch of data (ξ), with the other $n-1$ models. All models are initially assigned with the same weight matrix x^0. Considering that the model updates are gradient summations, thus commutative operations, all the n models would eventually get synchronized. However, the convergence of these models largely relies on the following factors: the degree of staleness of the incoming gradients; and how the models consider those gradients while updating themselves (Recht et al., 2011).

In an asynchronous setting, we end up updating each model with gradients that were calculated based on a different weight matrix. For example, if x_a is the current weight matrix of Model A, then on receiving two new gradients the model is updated as $x'_a = x_a + \nabla_{x_b,\xi_i} + \nabla_{x_c,\xi_j}$, where x_b and x_c are weight matrices of two remote models, B & C, respectively, and ξ_i & ξ_j are the respective training data they considered. Consequently, the SGD guarantees of convergence to a local minimum are loosened accordingly to the staleness of the gradients used for updating the model.

At Model A, the staleness of a gradient $\nabla_{x,\xi}$ calculated by Model B can be represented as $S(\nabla_{x,\xi}, B, A)$. We can define this staleness in terms of two gradient sets: (1) $G_{B\rightsquigarrow\nabla_{x,\xi}}$, the set of gradients processed by Model B at the time that it calculated the gradient using the batch ξ; and (2) $G_{\nabla x,\xi\rightsquigarrow A}$, the set of gradients processed by Model A at the time it applies the gradient. Intuitively, we define the staleness here as the symmetric difference of both gradient sets. That is, $G_{B\rightsquigarrow\nabla_{x,\xi}} \triangle G_{\nabla_{x,\xi}\rightsquigarrow A}$. The arrow, $\rightsquigarrow$, denotes if the gradient is an incoming or an outgoing gradient. Fig. 2.1 shows the distribution of the calculation and application of the gradients, along with their staleness values.

In Fig. 2.1, we depict a 4 node infrastructure in which the *Mini Batch Generator* (MBG) is a node that generates and asynchronously sends mini batches of training examples to Models 1 to 3. All models are initialised with the same weights. Each model calculates new gradients from the incoming batches and applies the gradients to itself, and incorporate the gradients it receives from the remote models. Take as example $Model_2$ that calculates the green gradient, ∇_{green}, apply it to itself and sends a copy of it to the other models. Then, $Model_2$ applies ∇_{blue} that it received from $Model_1$. Before calculating the blue gradient, $Model_1$ did not apply

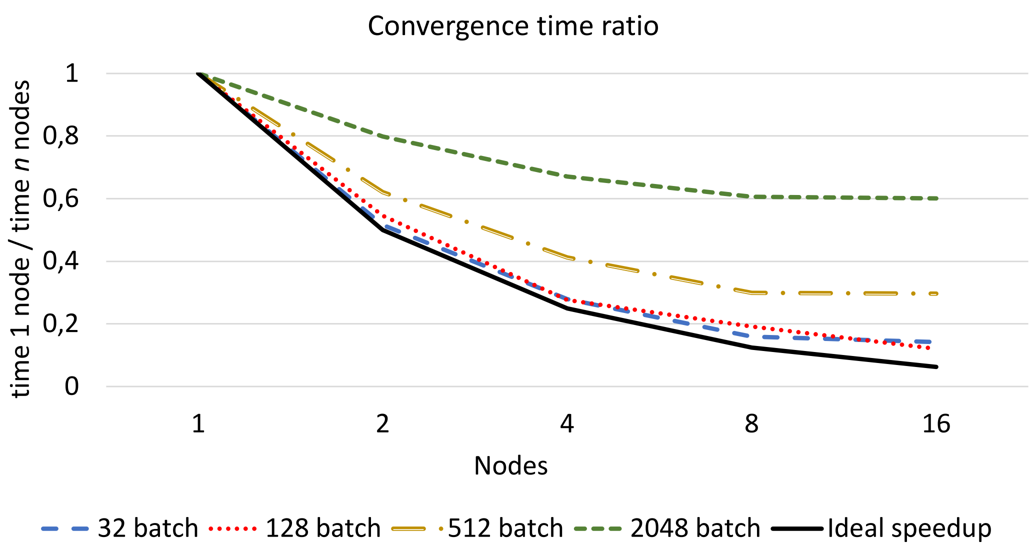


FIGURE 2.2: Convergence time ratio using 4 Word2Vec models per node varying the mini-batch sizes on 1 and n nodes.

any gradient to itself, thus, $G_{1 \rightsquigarrow blue} = \{\}$. Also, $Model_2$, immediately before applying ∇_{blue}, had already applied the green gradient to itself, thus $G_{blue \rightsquigarrow 2} = \{\nabla_{green}\}$. Therefore, the staleness of ∇_{blue} when applied to $Model_2$ is 1 because $S(blue, 1, 2) = G_{1 \rightsquigarrow blue} \triangle G_{blue \rightsquigarrow 2} = 1$.

2.5 Convergence Analysis & Directions

Experimental setting. By following the above-mentioned asynchronicity and staleness notions, we trained a Word2Vec model[2] using a fixed learning rate and 1% of Wikipedia (85% training and 15% test) on an HPC (Varrette et al., 2014) environment programmed using AIR (Venugopal et al., 2020), and Tensorflow C API as the NN framework.

Convergence analysis. Fig. 2.2 shows the reduction in the convergence time of the Word2Vec use case when asynchronously distributing its training. We achieved a near-optimal reduction in the convergence time for smaller mini-batches, whereas the training with larger ones is less time-consuming—the margin of speed up has reduced. In Fig. 2.3, we show that the losses on the *training* set and the *test* set remain the same while varying the number of nodes. This behaviour is evident from overlapping plots in the figure. While using 1 node (4 models), we had an average staleness of 3. And, while using 16 nodes (64 models), we had an average staleness of 55, which did not impact the model convergence rate. We also ran this same experiment using 32, 512, and 2048 mini-batches, and we observed this same pattern of overlap of the losses. However, while increasing the staleness further, even for sparse problems as Word2Vec, we eventually expect a difference in this trend. In general, these results indicate that the impact of staleness is negligible and can be ignored for sparse problems as Word2Vec considering the speed up achieved by allowing the staleness.

Future directions. We plan to investigate the behaviour of training a Word2Vec model with more nodes until observing the negative effects of the staled gradients.

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

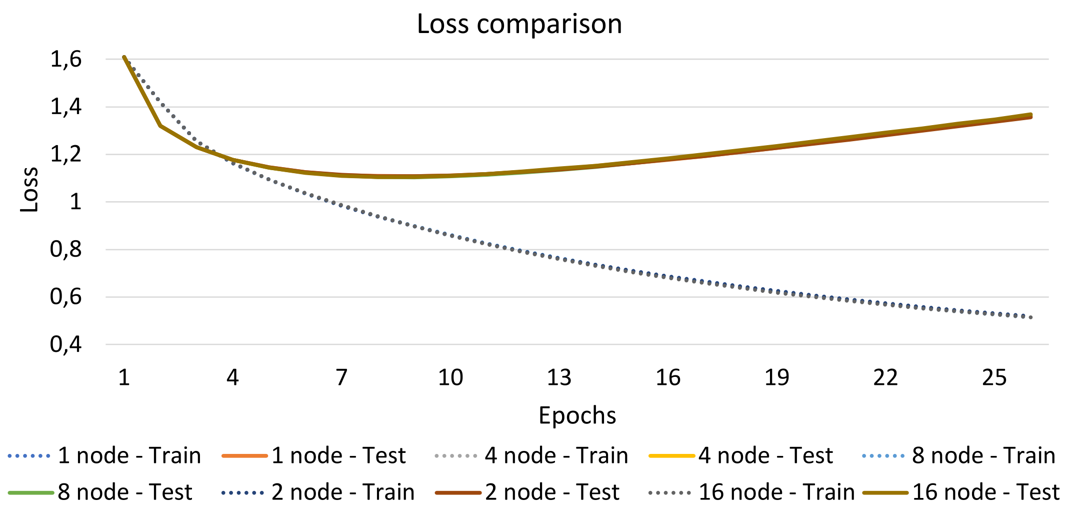


FIGURE 2.3: Train and test losses of the distributed Word2Vec usecase using 128 minibatch size on 4 models per node.

Moreover, we intend to explore how dense networks behave in an asynchronous distributed scenario. At last, we plan to perform these future studies using GPUs to approximate our experiments to current industry implementations.

Chapter 3

TensAIR: Online Learning from Data Streams via Asynchronous Iterative Routing

3.1 Contribution Statement

This chapter corresponds to the reprint of the paper "TensAIR: Online Learning from Data Streams via Asynchronous Iterative Routing" (Tosi, Venugopal, and Theobald, 2024) authored by Mauro Dalle Lucca Tosi, Vinu Ellampallil Venugopal, and Martin Theobald. Below, we provide the CRediT (Contributor Roles Taxonomy) [1] statements correspondent to each author contribution.

- **Mauro Dalle Lucca Tosi:**
  - Conceptualization;
  - Methodology;
  - Software;
  - Validation;
  - Formal analysis;
  - Investigation;
  - Data Curation;
  - Writing - Original Draft;
  - Visualization;
- **Vinu Ellampallil Venugopal**
  - Conceptualization;
  - Writing - Review & Editing;
- **Martin Theobald**
  - Conceptualization;
  - Resources;
  - Writing - Review & Editing;
  - Supervision;
  - Project administration;
  - Funding acquisition

[1] `https://credit.niso.org/`

3.2 Chapter Contextualization

This chapter corresponds to the paper "TensAIR: Online Learning from Data Streams via Asynchronous Iterative Routing" (Tosi, Venugopal, and Theobald, 2024). It introduces the TensAIR framework, which enables the asynchronous and decentralized training of ANN models from data streams in real-time. The development of a framework for training ANN models in an OL scenario that is versatile enough to accommodate various ANN architectures and training strategies, is a valuable contribution to the broader field of online ML.

3.3 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* (Hoi et al., 2021). A common characteristic of data streams is the presence of *concept drifts* (Iwashita and Papa, 2018), i.e., changes in the statistical properties among the incoming data objects over time (Lu et al., 2018). Consequently, pre-trained ML models tend to be inadequate in OL scenarios as their performance usually decreases after concept drifts (Lu et al., 2018). 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 (Hoi et al., 2021).

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, complex OL problems cannot rely on robust solutions common to other ML problems due to their long training time (Hoi et al., 2021), like those involving Artificial Neural Networks (ANNs) (Goodfellow, Bengio, and Courville, 2016).

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 (Lu et al., 2018; Hu, Kantardzic, and Sethi, 2020; Priya and Uthra, 2021; Barros and Santos, 2018; Sethi and Kantardzic, 2017) 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 (*Deep Learning on Flink* 2022; *Robust machine learning on streaming data using Kafka and Tensorflow-IO* 2022) for Apache Flink (Carbone et al., 2015) and Kafka (Kreps, Narkhede, Rao, et al., 2011) 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 3.6). 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) *high throughput data stream:* The model must process a higher throughput (data samples per second) than the data stream produces. This is achieved by processing data samples (for training and for inference/prediction) faster than they arrive from the data stream.

In this paper, we present the architecture of TensAIR, the first OL system 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) (Tosi and Theobald, 2023) to update the ANN models.

To assess TensAIR, we performed experiments on sparse (Word2Vec) 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 (*Robust machine learning on streaming data using Kafka and Tensorflow-IO* 2022) and from 6 to 120 times higher than Apache Flink extension (*Deep Learning on Flink* 2022). We additionally compared TensAIR to Horovod (Sergeev and Del Balso, 2018), 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 and the major limitations of TensAIR.

Contributions

1. Design and implementation of TensAIR, the first system for real-time training and prediction in ANN models. Instead of relying on buffered datasets for training, TensAIR updates its ANN model from every data sample or data batch from the data streams;

2. Consistency and convergence analysis of asynchronous and decentralized training of ANN models.

3. Experimental evaluation of TensAIR showing almost linear training time speed-up in terms of the worker nodes deployed (with no degradation of models quality);

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 best baselines;

5. Depiction of real-time Sentiment Analysis use case that would not be feasible with standard OL approaches.

Limitations

1. TensAIR currently relies on passive drift adaptation (cf. Section 3.4), which tends to be computationally more demanding than active drift adaptation;

2. TensAIR uses labels for training the ANN models in real-time. However, labels are rarely available in data streams. Therefore, TensAIR is mostly suited for training unsupervised or semi-supervised problems such as Anomaly Detection (Pang et al., 2021) and Generative Modeling (Ruthotto and Haber, 2021).

3.4 Background & Related Work

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.

3.4.1 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 (Gomes et al., 2019). 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 Ω consists of ordered *events* e with *timestamps* ω, i.e., $(e_1, \omega_1), \ldots, (e_\infty, \omega_\infty)$, where the ω_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 of fixed size b, as follows:

$$\begin{aligned} B_1 &= (e_1, \omega_1), \ldots, (e_b, \omega_b) \\ B_2 &= (e_{b+1}, \omega_{b+1}), \ldots, (e_{2b}, \omega_{2b}) \\ &\ldots \end{aligned}$$

Batches B 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 ω_i were ordered. In common stream-processing architectures, such as Apache Flink (Carbone et al., 2015), Spark (Zaharia et al., 2016) and Samza (Noghabi et al., 2017), batches are distinguished into *sliding windows*, *tumbling windows* and (per-user) *sessions* (Akidau et al., 2015).

Latency vs. Throughput. When analyzing systems that process data streams, one typically benchmarks them by their latency and throughput (Karimov et al., 2018).

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" (Kulkarni et al., 2015), or even a crash).

Passive & Active Drift Adaptation. To adapt to concept drifts, one may rely on either passive or active adaptation strategies (Heusinger, Raab, and Schleif, 2020). 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.

3.4.2 Artificial Neural Networks

ANNs denote a family of supervised ML algorithms which are designed to be trained on a pre-defined dataset (Goodfellow, Bengio, and Courville, 2016). A training dataset is composed of multiple (z, y) pairs, in which z is a training example and y is its corresponding label. ANNs are usually trained using *mini-batches* θ, which are sets of (z, y) pairs of fixed size c that are iteratively (randomly) sampled from the training dataset, thus $\theta = (z_i, y_i), \ldots, (z_{i+c}, y_{i+c})$.

BGD vs. SGD. An ANN model is represented by the weights and biases of the network, described together by x. We train the network using Batch Gradient Descent (BGD) (Ruder, 2016) which is based on Stochastic Gradient Descent (SGD) (Robbins and Monro, 1951). BGD updates x by considering $\nabla F(x, \theta)$, which is the gradient of a pre-defined *loss function* F with respect to x when taking θ as input. Thus, we can represent the update rule of x^t as in Equation 3.1, in which t is the iteration in BGD, and η is a pre-defined learning rate.

$$x^{t+1} = x^t - \eta \nabla F(x^t, \theta) \tag{3.1}$$

Based on Equation 3.1, x^{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 x^{t+1}.

Convergence. Despite the differences between OL and ANNs, they have a similar training pipeline. Both approaches are trained based on analogous sets of data points, which are the batches for OL and the mini-batches for ANNs. A seminal result for training ANNs (Goodfellow, Bengio, and Courville, 2016) is that SGD converges to a (local) minimum of the loss function as the number of batches approaches *infinity*—even when not all of the actually available training examples fall into these batches. So-called *epochs* were introduced to ANN training in order to compensate for the lack of labeled training examples by performing repeated iterations over the available examples. A main conjecture we follow with TensAIR thus is that, instead of using epochs, we may train the underlying ANN by using new batches from a data stream as input and hence converge as long as the principal distribution of the (z, y) pairs in the data stream remains unchanged (e.g., between two major concept drifts).

3.4.3 Distributed Artificial Neural Networks

Over the last years, ANN models have 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 (Sergeev and Del Balso, 2018). 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 synchronization settings (Mayer and Jacobsen, 2020).

3.4.3.1 Parallelization methods

There are many possible *parallelization strategies* for ANNs (Mayer and Jacobsen, 2020). We describe below the two most common ones: *model* and *data parallelism*.

Model Parallelism. In model parallelism (Ouyang et al., 2021), the ANN model is split into different parts which are distributed among the worker nodes. The major challenge when using model parallelism is to determine how to partition the model and keep the computation balanced among the workers (Mayer and Jacobsen, 2020). Considering the inherent difficulty of developing a splitting method that is generic enough to be used on different ANN models and scalable enough to be distributed across multiple nodes, we omit the consideration of model parallelism in this work.

Data Parallelism. In data parallelism (Ouyang et al., 2021), the c training pairs within a mini-batch are assumed to be independent of each other (Ben-Nun and Hoefler, 2019). Thus, workers are initialized as replicas and trained with different splits of those c pairs. The replicas perform the GC steps over the data received and then synchronize their parameters among themselves. This method of parallelism is the most common one and has existed since the first implementations of ANNs (Mayer and Jacobsen, 2020). Frameworks like TensorFlow (Abadi et al., 2016) or PyTorch (Paszke et al., 2019) use data parallelism by default when deployed on multi-core compute nodes or GPUs.

3.4.3.2 System Architectures

The synchronization among the workers' parameters in a data-parallel ANN setting is either *centralized* or *decentralized* (Mayer and Jacobsen, 2020), as described below.

Centralized. In a centralized architecture (Ouyang et al., 2021), 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 centralized model (Mayer and Jacobsen, 2020). Then, the workers use the centralized model on the next iterations of BGD. Despite being easier to manage compared to decentralized architectures, the scalability when using parameter servers is limited. Thus, by relying on parameter servers to aggregate updates and broadcast them to all workers, the parameter servers may become the actual bottleneck of such an architecture (Chen, Wang, and Li, 2019).

Decentralized. In a decentralized architecture (Ouyang et al., 2021), the workers synchronize themselves using a broadcasting-like form of communication (Ouyang et al., 2021) which eliminates the bottleneck of the parameter servers. An *AllReduce* operation may be performed on a fully connected network, generating a communication cost of $\mathbf{O}(N^2)$ over N workers; or, on ring-like topologies, an *Ring-AllReduce* operation may reduce the communication to $\mathbf{O}(N)$ (Mayer and Jacobsen, 2020) but typically increases the time it takes to propagate the parameters through the network.

3.4.3.3 Synchronization Settings

The aggregation and application of the model updates in a data-parallel ANN system can be *synchronous* or *asynchronous*. Those settings are described below.

Synchronous. In a synchronous setting (Ouyang et al., 2021), workers have to synchronize themselves after each iteration. Therefore, they can only initialize the next *GC* step if their models are synchronized. This synchronization barrier directly facilitates the proof of convergence of SGD (as in a centralized setting) but wastes computational resources at idle times (i.e., when workers have to wait for others to resume their computation) (Mayer and Jacobsen, 2020).

Asynchronous. In an asynchronous SGD (ASGD) setting (Ouyang et al., 2021), workers are allowed to compute GC steps also on stale model parameters. This behavior obviously minimizes idle times but makes it harder to mathematically prove SGD convergence. Recent developments on ASGD (Bornstein et al., 2023; Tosi and Theobald, 2023; Lian et al., 2018; Koloskova, Stich, and Jaggi, 2022; Dean et al., 2012; Jiang et al., 2017; Dai et al., 2015; Liu et al., 2014; Liu and Wright, 2015; Sun, Hannah, and Yin, 2017; Lian et al., 2015; Zhang, Liu, and Zhu, 2020), however, have tackled exactly this issue under different assumptions. In special, we highlight (Bornstein et al., 2023; Lian et al., 2018; Tosi and Theobald, 2023), which proved the convergence of asynchronous and decentralized SGD altogether. On one hand, Bornstein et al. (Bornstein et al., 2023) and Lian et al. (Lian et al., 2018) rely on averaging local models with their neighbors. On the other hand, Tosi and Theobald (Tosi and Theobald, 2023) propose DASGD, which broadcasts locally calculated gradients across the whole network.

3.4.4 AIR – Distributed Dataflow Engine

Asynchronous Iterative Routing (AIR) is a distributed dataflow engine which implements a light-weight iterative sharding protocol (Venugopal et al., 2020) on top of the Message Passing Interface (MPI). It is a native stream-processing engine that processes complex dataflows, consisting of direct acyclic graphs (DAGs) of logical dataflow operators. AIR's main features are that it uses an asynchronous MPI protocol and does not rely on a master-client architecture (but follows a pure client-client pattern) for communication. Both of these characteristics differ AIR both from existing bulk-synchronous processing (BSP) systems, such as Apache Spark (Zaharia et al., 2016), and asynchronous stream-processing (ASP) engines, such as Apache Flink (Carbone et al., 2015) and Samza (Noghabi et al., 2017). AIR is implemented in C++ using POSIX threads (pthreads) for multi-threading and MPI for communication among nodes. Due to its light-weight and robust architecture, AIR's scale-out performance has been shown to be up to 15 times better than Spark and up to 9 times better than Flink (Venugopal et al., 2022) over a variety of HPC settings.

3.5 TensAIR Architecture

We now introduce the architecture of *TensAIR*, the first stream-processing system for training and predicting in ANNs models in real-time.

3.5.1 TensAIR

TensAIR is a native stream-processing system that processes complex dataflows consisting of logical AIR (Venugopal et al., 2020; Venugopal et al., 2022) operators. TensAIR augments AIR with 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 3.1), which differently than AIR, is not a DAG because it allows cycles among the `Model` operators. Note that, throughout this paper, we use the terms `prediction` and `inference` interchangeably.

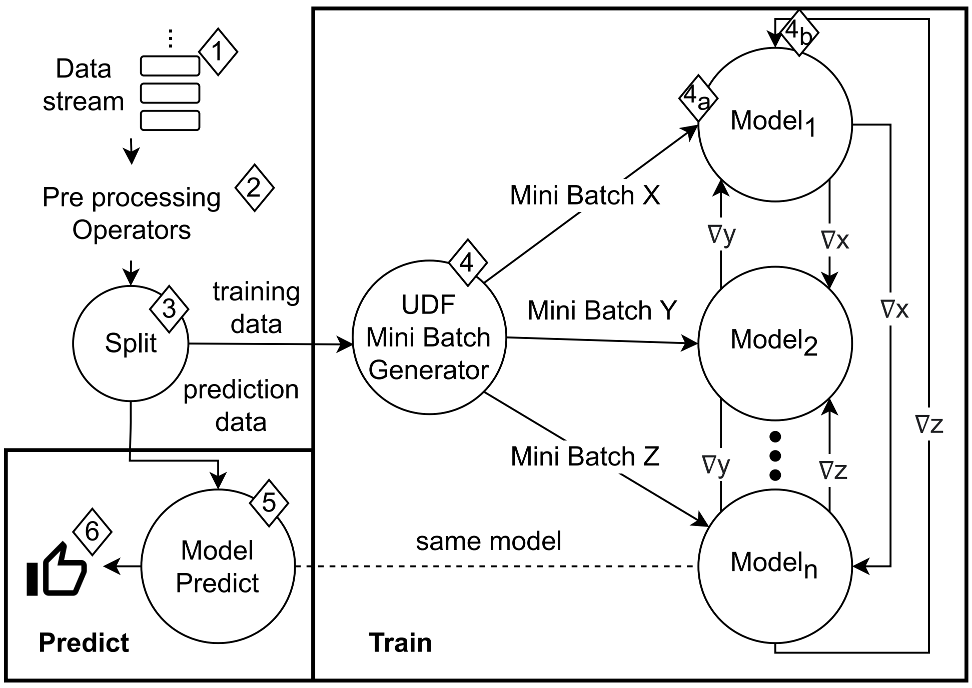


FIGURE 3.1: TensAIR generic dataflow with n distributed `Model` instances.

TensAIR Dataflows. Figure 3.1 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 buffered until a desired buffer size (*maxGradBuffer*) for the outgoing gradients is reached, upon which the buffer then is broadcast to all other `Model` instances.

Algorithm 1 TensAIR `Model` class with additional OL functions `train` and `predict`

```
1:  Constructor Model (tfModel, maxBuffer) extends Vertex:
2:      model = tfModel
3:      maxGradBuffer = maxBuffer
4:      gradients = $\emptyset$
5:      ALIVE = true
6:  procedure STREAMPROCESS(msg)
7:      while ALIVE do
8:          if msg.mode == TRAIN then
9:              train(msg)
10:         else
11:             predict(msg)
12: procedure TRAIN(msg)
13:     if msg.isGradient then
14:         model = apply_gradient(model, msg)
15:     else
16:         gradient = calculate_gradient(model, msg)
17:         model = apply_gradient(model, gradient)
18:         gradients = gradients $\cup$ {gradient}
19:         if $|$gradients$| \geq$ maxGradBuffer then
20:             send_gradients(gradients)
21: procedure PREDICT(msg)
22:     predictions = model.make_predictions(msg)
23:     send_results(predictions)
```

3.5.2 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* (Tosi, Venugopal, and Theobald, 2022). This $staleness_{i,j}(\nabla_a)$ metric is the symmetric distance between G_i and G_j with respect to the times at

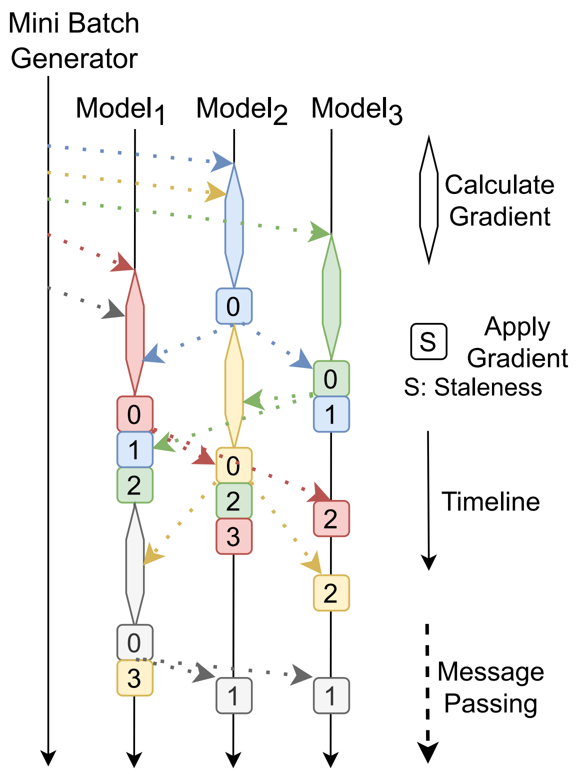


FIGURE 3.2: ASGD staleness on a distributed framework.

which a new gradient ∇_a was computed by a model i and is applied to model j, respectively. This phenomenon and the staleness metric are illustrated in Figure 3.2.

Figure 3.2 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, immediately 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 3.2. 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 $staleness_{1,2}(\nabla_{red}) = 3$.

3.5.3 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 (Zhang, Liu, and Zhu, 2020; Sun, Hannah, and

Yin, 2017; Jiang et al., 2017) 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 (Goodfellow, Bengio, and Courville, 2016), each of the local model instances will eventually converge.

Second, considering TensAIR's asynchronous and distributed SGD (DASGD), model updates can be staled. Nevertheless, as proven by Tosi and Theobald (Tosi and Theobald, 2023), the model will converge in up to $\mathcal{O}(\frac{\sigma}{\epsilon^2}) + \mathcal{O}(\frac{QS_{avg}}{\epsilon^{\frac{3}{2}}}) + \mathcal{O}(\frac{S_{avg}}{\epsilon})$ 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 is proven to converge in $\mathcal{O}(\frac{\sigma}{\epsilon^2}) + \mathcal{O}(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{\epsilon})$ iterations, with $\hat{S}_{max}$ and $\hat{S}_{avg}$ representing the maximum and average staleness, that also takes into consideration a recursive factor during their calculation.

Note that, by relying on DASGD, instead of local models sending their current weights to other nodes, local models only broadcast their gradients. This characteristic reduces the size of the messages exchanged among nodes if gradients are sparse.

3.5.4 Implementation

TensAIR dataflow operators extend a basic `Vertex` superclass in AIR (Venugopal et al., 2020). `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 3.1.

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[2].

3.6 Experiments & Discussion

To assess TensAIR, we performed experiments to measure its performance on solving prototypical ML problems such as Word2Vec (*word embeddings*) and image classification (CIFAR-10). 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

[2] `https://github.com/maurodlt/tensair`

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 (Varrette et al., 2022). 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[3]) and dense (image classification[4]) models based on English Wikipedia articles and images from CIFAR-10 (Krizhevsky, Hinton, et al., 2009), 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 3.1) which generates events e_i with timestamps ω_i, groups them into mini-batches θ_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).

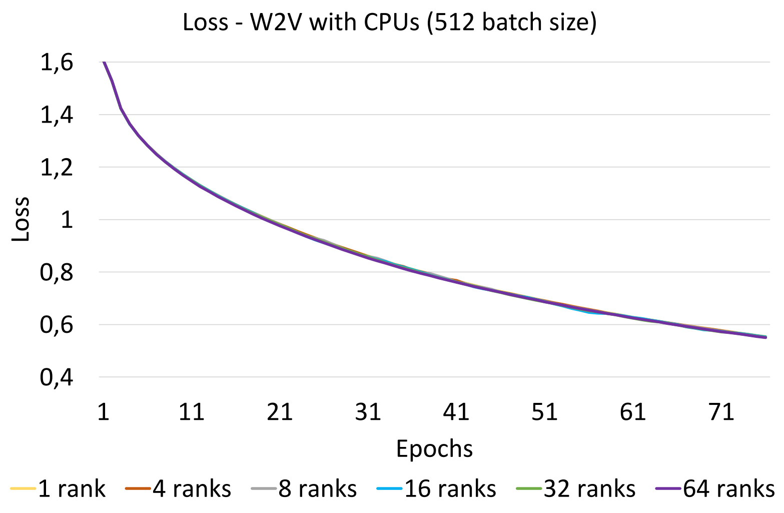
Sparse vs. Dense Models. We chose Word2Vec and CIFAR-10 because they represent prototypical ML problems with *sparse* and *dense* model updates, respectively. Sparse updates mean that only a small portion of the neural network variables actually become updated per mini-batch (Recht et al., 2011). Hence, sparseness should assist the models' convergence when using DASGD, as observed also in Hogwild! (Recht et al., 2011). 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.

3.6.1 Convergence Analysis

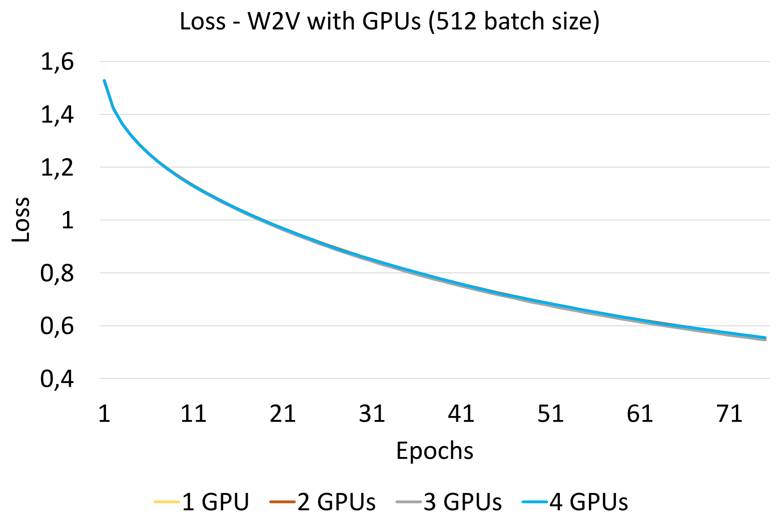
We first explored TensAIR's ability to converge by determining if and how DASGD might degrade the quality of the trained model (Figures 3.3 and 3.4). We compared the training loss curve of Word2Vec and CIFAR-10 by distributing TensAIR models from 1 to 4 GPUs using 1 TensAIR rank per GPU (Figures 3.3b & 3.4b). We additionally explored the models convergence when trained with distributed CPU nodes (Figures 3.3a & 3.4a). 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 3.3a and 3.3b demonstrates that our asynchronous and distributed 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

[3]`https://www.tensorflow.org/tutorials/text/word2vec`
[4]`https://www.tensorflow.org/tutorials/images/cnn`



(A)

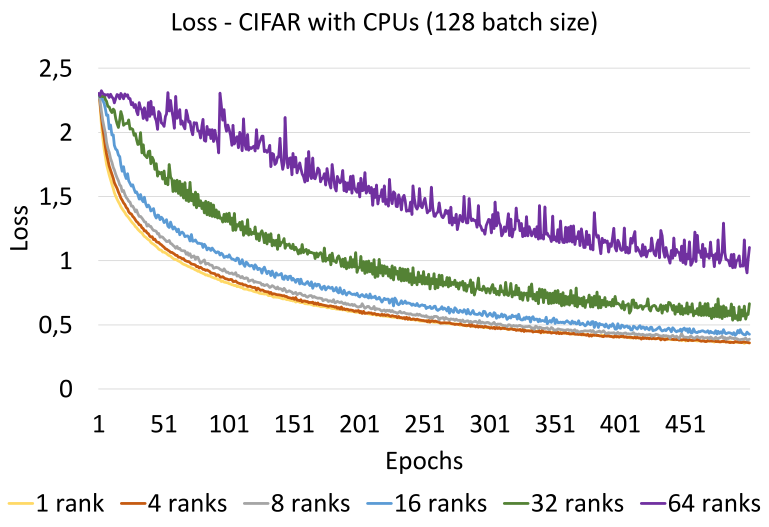


(B)

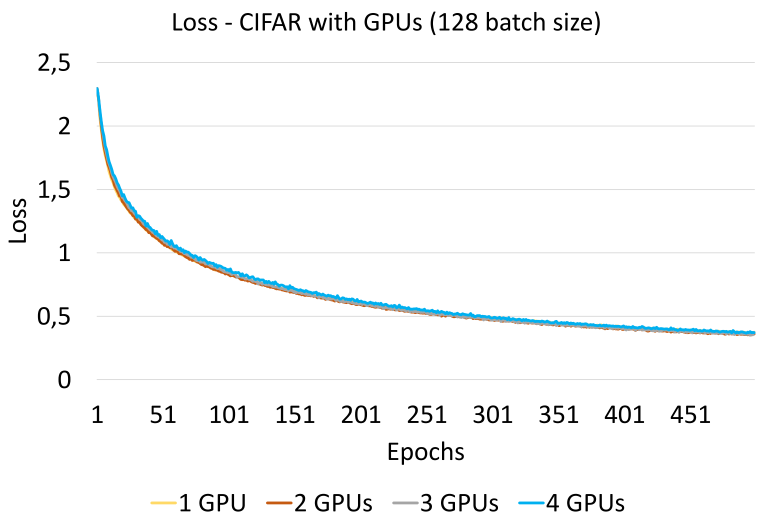
FIGURE 3.3: Convergence analysis of TensAIR on the Word2Vec use-case.

are relatively inexpensive to compute and apply for Word2Vec). The low staleness indicates a fast exchange of gradients among models.

In Figure 3.4a, 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 c is similar to training 1 model with mini-batches of size $n \cdot c$. To mitigate this issue, Horovod recommends to increase the learning rate α by the number of ranks used to distribute the model (*Horovod with Keras* 2019), i.e., $\alpha_{new} = \alpha \cdot n$. Accordingly, in Figure 3.4b, we again do not see any degradation of the loss when distributing CIFAR-10 across multiple GPUs because we use a maximum of 4 GPUs.



(A)

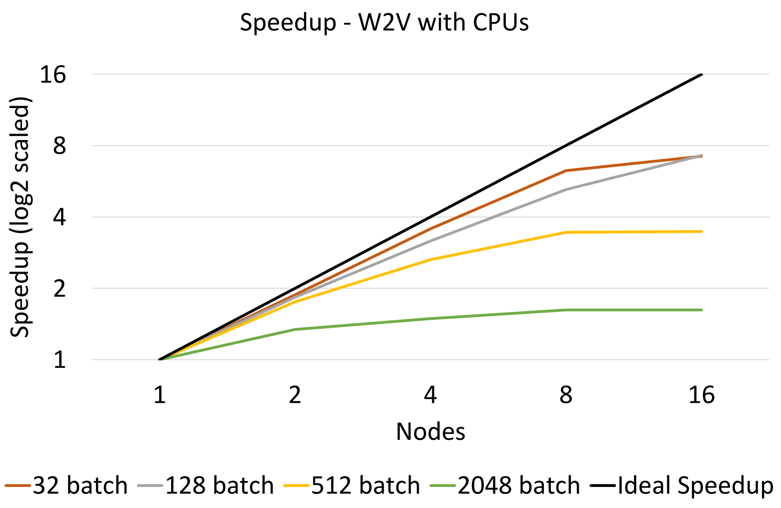


(B)

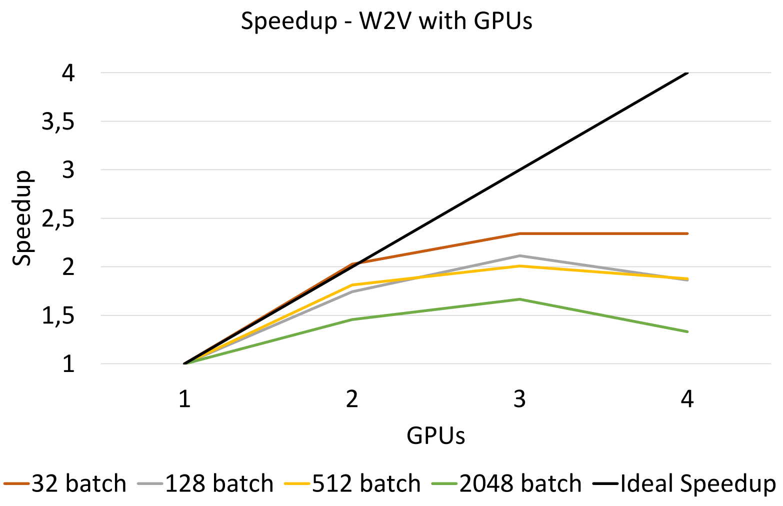
FIGURE 3.4: Convergence analysis of TensAIR on the CIFAR-10 use-case.

3.6.2 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 Figures 3.5 and 3.6 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 the mini-batch size is inversely proportional to the training time. Hence, the smaller the training time, the bigger is the fraction of the computation that is not distributed. Thus, models with expensive gradient computations will have a better scale-out performance.



(A)

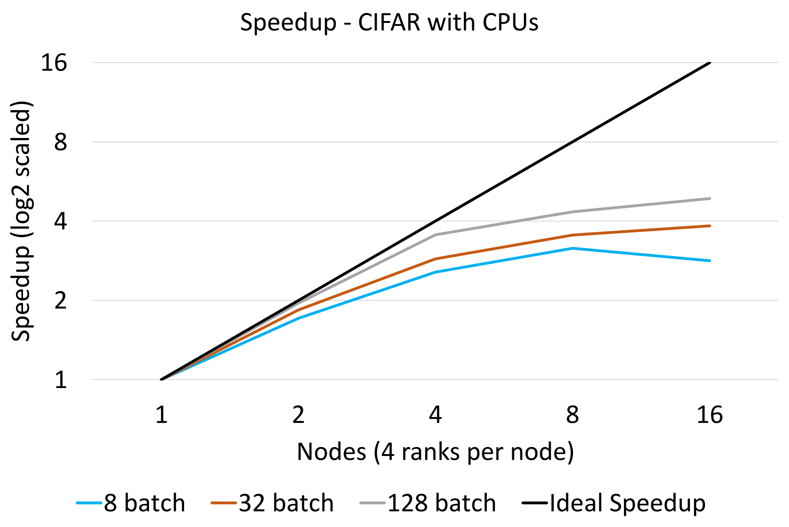


(B)

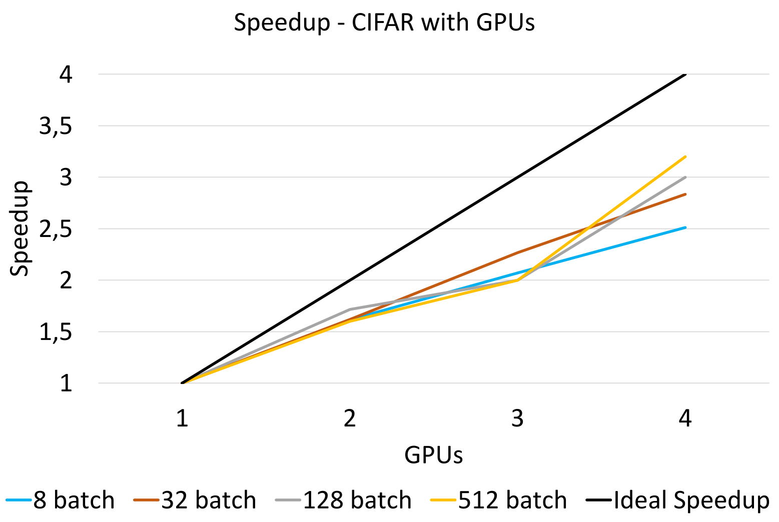
FIGURE 3.5: **Speedup analysis of TensAIR on the Word2Vec use-case.**

3.6.3 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 (*Robust machine learning on streaming data using Kafka and Tensorflow-IO* 2022). 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 (*Deep Learning on Flink* 2022) even after various attempts on our HPC setup. We therefore report the direct deployment of our Word2Vec and CIFAR-10 use-cases (Figures 3.7a & 3.7b) 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



(A)

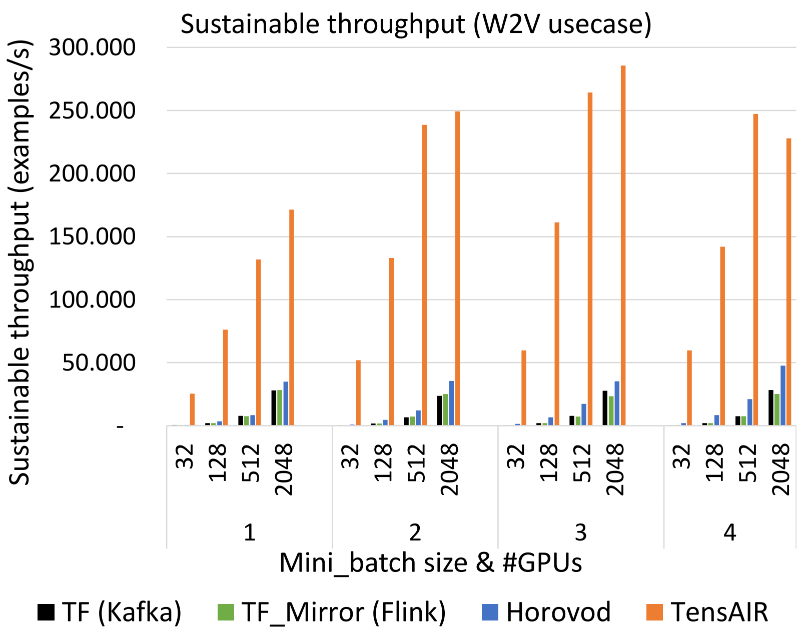


(B)

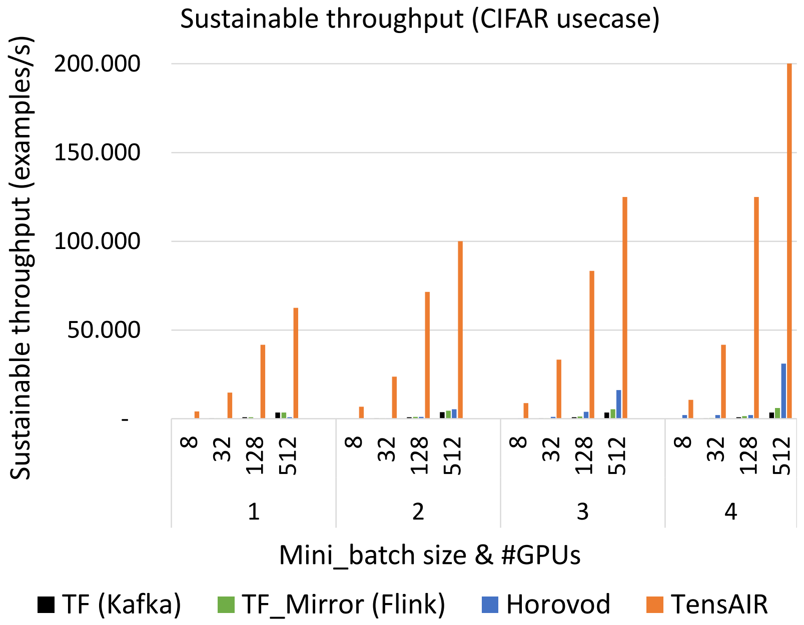
FIGURE 3.6: Speedup analysis of TensAIR on the CIFAR-10 use-case.

upper-bound for the maximum throughput that Kafka and Flink could achieve. In a similar manner, we also determined the maximum throughput of Horovod (Sergeev and Del Balso, 2018), which is however not a streaming engine by default.

In Figures 3.7a and 3.7b, 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 is an end-to-end dataflow engine prepared to train ANNs from streaming data. Thus, the transfer of training data overhead is mitigated by the asynchronous iterative routing protocol. This allows TensAIR to (1) reduce both computational resources and idle times while the data is being transferred, and (2)



(A) W2V use-case.



(B) CIFAR-10 use-case.

FIGURE 3.7: Throughput comparison between TensAIR, TensorFlow (standard and distributed), and Horovod.

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 CIFAR-10.

3.6.4 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 (Go, Bhayani, and Huang, 2009) from Stanford. Then, we trained an LSTM model (*TensorFlow* 2022b) 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.

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

TABLE 3.1: Cosine differences after updating word embeddings.

As shown in Table 3.1, 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 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 3.8) 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 if 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 3.8 depicts the dataflow for a *Sentiment Analysis* (SA) use-case on a Twitter data stream. This dataflow predicts the sentiments of live tweets using a pre-trained ANN model ($\texttt{Model}^{SA}$). However, it does not rely on pre-defined word embeddings. The dataflow constantly improves its embeddings on a second *Word2Vec* (W2V) ANN model ($\texttt{Model}^{W2V}$), 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 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 (z, y) directly from the input tweets.

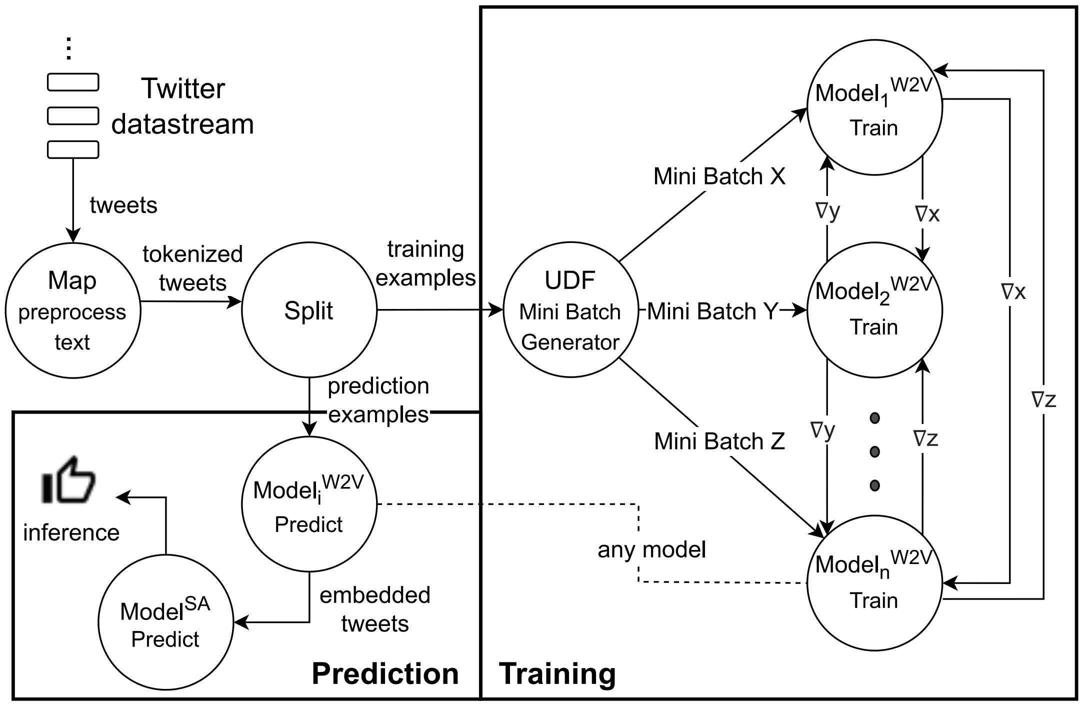


FIGURE 3.8: TensAIR dataflow with n distributed `Model`^{W2V} instances and a single instance of `Map`, `Split`, `UDF` and `Model`^{SA}.

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 `Model`^{W2V} and `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 (z, y) word pairs in each mini-batch θ are sharded across `Model`$_1^{W2V}$, ..., `Model`$_n^{W2V}$ with a standard hash-partitioner using words z as keys. `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 `Model`^{W2V} instances and sent to the pre-trained `Model`^{SA} which then predicts the tweets' sentiments.

3.7 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 distributed 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 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 implementation 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 the throughput of Apache Kafka and Flink extensions, respectively).

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 scale-out performance 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.

Chapter 4

OPTWIN: Drift Identification with Optimal Sub-Windows

4.1 Contribution Statement

This chapter corresponds to the extended version of the paper “OPTWIN: Drift Identification with Optimal Sub-Windows” (Tosi and Theobald, 2024) authored by Mauro Dalle Lucca Tosi and Martin Theobald - © 2024 IEEE. Below, we provide the CRediT (Contributor Roles Taxonomy) [1] statements correspondent to each author contribution.

- **Mauro Dalle Lucca Tosi:**
  - Conceptualization;
  - Methodology;
  - Software;
  - Validation;
  - Formal analysis;
  - Investigation;
  - Data Curation;
  - Writing - Original Draft;
  - Visualization;
- **Martin Theobald**
  - Formal analysis;
  - Resources;
  - Writing - Review & Editing;
  - Supervision;
  - Project administration;
  - Funding acquisition

4.2 Chapter Contextualization

This chapter corresponds to the extended version of the paper “OPTWIN: Drift Identification with Optimal Sub-Windows” (Tosi and Theobald, 2024). It proposes

[1] https://credit.niso.org/

OPTWIN, an error-based drift detector algorithm that detects concept drifts from classification and regression problems while producing a low false positive rate. The development of such drift detectors is essential for optimizing the utilization of computational resources and preventing unnecessary retraining of ANN models that have already converged and have not experienced concept drift.

4.3 Introduction

Online Learning (OL) is an area of research that gained an increasing amount of attention in academia and industry over the past years. OL is a sub-field of Machine Learning (ML) in which an underlying ML technique aims to constantly update its model's parameters from an incoming data stream under limited time and space constraints. Ideally, OL methods are trained in real-time and thereby aim to maximize the prediction accuracy of their models based on bounded windows of data instances they have previously seen and which they may see in the near future (Hoi et al., 2021). Use-cases of OL are varied and include online video segmentation (Koner et al., 2023), spam detection (Fernández Riverola et al., 2007), oral hygiene rewarder (Trella et al., 2023), and many others (Wang et al., 2023; Li et al., 2023; Mandal et al., 2023).

The presence of *concept drifts* is one of the numerous challenges when processing data streams in real-time. Concept drifts are commonly defined as unforeseeable changes in the statistical properties of the incoming data stream over time (Lu et al., 2018). Such a change may have different sources and types, and it may involve different adaptation strategies (cf. Section 4.4.1). In practice, concept drift is the key phenomenon that impairs the performance of pre-trained (and hence static) ML models over data streams.

The most common form of drift detection is the *error rate-based* one. This type of drift detection uses the difference between the number or magnitude of historical prediction errors and new prediction errors to determine if a concept drift occurred (Lu et al., 2018). The most popular error rate-based algorithms are ADWIN (Bifet and Gavalda, 2007), DDM (Gama et al., 2004), and their variations as (Baena-Garcıa et al., 2006; Barros et al., 2017). Those and newer drift detectors (Wu et al., 2021; Hidalgo, Mariño, and Barros, 2019) are known for being lightweight and identifying concept drifts with an outstanding true-positive rate. However, those algorithms have a high false-positive rate. Additionally, most of them work with classification problems only, with few being suited for regression problems. Therefore, as concluded by Bayram et al. (Bayram, Ahmed, and Kassler, 2022), the development of drift detectors which return a low false-positive rate and are suited for both classification and regression are open research problems in this area.

As opposed to previous approaches, which track concept drifts based on the *means* of past prediction errors, we argue that also changes in the *standard deviations* of those errors should trigger concept drifts. Take, as a simple example, a regressor that outputs the following errors at window W_0:

$$W_0 = \langle 0.3; 0.7; 0.7; 0.3; 0.3; 0.7; 0.5; 0.5 \rangle$$

While, at window W_1, it outputs the following errors:

$$W_1 = \langle 0.0; 1.0; 1.0; 0.0; 1.0; 0.0; 0.0; 1.0 \rangle$$

Current drift detectors like ADWIN would not consider this as a concept drift, as the error means μ_{W_0}, μ_{W_1} over both windows are equal to 0.5, whereas we—intuitively—understand that something changed in the error stream and, therefore, a concept drift should be flagged.

In this paper, we propose the "OPTimal WINdow" drift detector (OPTWIN). It is a sliding-window algorithm that analyzes the error rates (either binary or real-valued) produced by an underlying ML algorithm. It calculates the *optimal cut* of a sliding window W to divide it into two sub-windows W_{hist} and W_{new}. It then performs the well-known t- (for means) and f-tests (for standard deviations) to determine whether the two sub-windows exhibit a statistically significant difference in either their means or standard deviations, respectively. OPTWIN has the following two main features: (1) it can calculate the *optimal cut* based only on the length of W in $\mathcal{O}(1)$; (2) it uses both the errors' means and standard deviations to flag drifts. Furthermore, we provide detailed and rigorous guarantees of OPTWIN's performance in terms of its true-positive (TP), false-positive (FP), and false-negative (FN) rates, as well as in its drift-identification delay.

To assess OPTWIN, we used the popular MOA framework (Bifet et al., 2010) and compared OPTWIN to the most studied drift-detection frameworks in the literature: ADWIN (Bifet and Gavalda, 2007), DDM (Gama et al., 2004), EDDM (Baena-Garcıa et al., 2006), STEPD (Nishida and Yamauchi, 2007) and ECDD (Ross et al., 2012). Using MOA, we compared the *precision*, *recall* and *F1-score*, as well as the *delay* of all drift detectors over both sudden and gradual drifts. Furthermore, we also trained a Naïve Bayes (NB) classifier on MOA that adapts itself based on the drift detectors and compared its results on various synthetic datasets (STAGGER (Schlimmer and Granger, 1986), RANDOM RBF (Bifet et al., 2009), and AGRAWAL (Agrawal, Imielinski, and Swami, 1993)) and real-world ones (Covertype and Electricity) (Blackard and Dean, 1999; Harries, 1999). Additionally, we likewise assessed OPTWIN on a Neural Network (NN) use-case. Specifically, we pre-trained a Convolutional NN (CNN) with the CIFAR-10 (Krizhevsky, Hinton, et al., 2009) image dataset and simulated an end-to-end OL scenario with concept drifts by using OPTWIN and ADWIN as drift detectors.

Our results show that OPTWIN reliably identifies both sudden and gradual drifts in classification and regression problems. Furthermore, OPTWIN is the drift detector with the best F1-score when compared to the baselines. This is due to its higher precision, which indicates a low FP rate. With respect to the drift-identification delay, there was no drift detector that was superior to the others in a majority of the datasets. In terms of run-time, OPTWIN in combination with the CNN training pipeline is 21% faster than a similar ADWIN pipeline due to OPTWIN's lower FP rate, which leads to a significantly reduced amount of re-training iterations. Therefore, our experiments indicate that OPTWIN identifies concept drifts with a similar delay as other drift detectors but maintains a higher precision and recall in the drift detection, which reduces the overall run-time of OL pipelines that trigger a re-training of their models upon each detected concept drift.

4.4 Background & Related Works

We next formally introduce the problem of concept drift and review its principal characteristics (cf. Section 4.4.1). We also discuss the most important related approaches for concept-drift detection from the literature, which serve as inspiration and baselines for our work (cf. Section 4.4.2).

4.4.1 Background

A concept drift may exhibit different characteristics, whose understanding is essential for a fast and reliable drift detection. Consider an unbounded data stream that receives *features* $z_i \in Z$ and *labels* $y_i \in Y$ in the form of pairs of instances (z_i, y_i) used for training an underlying ML model. At *time* t, these instances follow a certain distribution $P_t(Z, Y)$. A concept drift is the change of this distribution over time. Formally, a concept drift occurs at time $t+1$ iff $P_t(Z, Y) \neq P_{t+1}(Z, Y)$ (Lu et al., 2016; Lu et al., 2018; Gama et al., 2014). The underlying data distribution $P_t(Z, Y)$ may also be decomposed and expressed as a product of probabilities $P_t(Z, Y) = P_t(Z) \times P_t(Y|Z)$ via the well-known *chain rule of probability*. Thus, a concept drift may come from two *sources*: (1) $P_t(Z) \neq P_{t+1}(Z)$, which is known as a *virtual drift* because it does not impact the decision boundaries of the learner; and (2) $P_t(Y|Z) \neq P_{t+1}(Y|Z)$, which is known as an *actual drift* because it directly impacts the learner's accuracy (Lu et al., 2016; Lu et al., 2018; Gama et al., 2014). It is also possible to have both drift sources simultaneously (Lu et al., 2018).

Another important characteristic of a concept drift is its *type*. Concept drifts can be (1) *sudden*; (2) *incremental*; (3) *gradual*; and (4) *reoccurring* (see Figure 4.1) (Lu et al., 2018; Gama et al., 2014). Sudden drifts occur when the probability distribution changes completely within a single step. Incremental drifts occur when the distribution $P_t(Z, Y)$ changes incrementally until its convergence. Gradual drifts occur when the new distribution gradually replaces the old one. Reoccurring drifts occur when distributions can reoccur after some time.

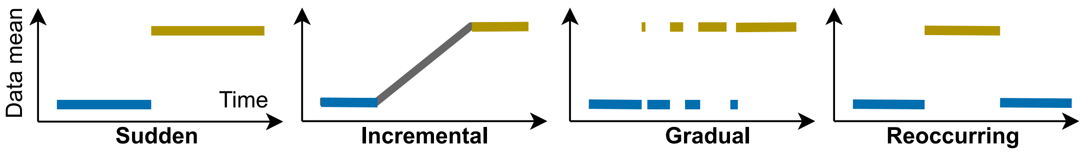


FIGURE 4.1: **Concept drift types.**

To recover from concept drifts, one may adapt the learner according to different strategies. The selection of the appropriate strategy depends not only on the type and source of the drifts but also on the use-case and the adopted learner. A common strategy is to train a new model with the latest data points whenever a drift is identified. Another strategy is to adjust the current learner instead of training a new model from scratch. Moreover, it is also possible to have an ensemble of learners, which is primarily useful for adapting to reoccurring drifts (Lu et al., 2018).

The most popular procedure to evaluate learning algorithms that handle concept drifts is the *prequential procedure*. Prequential is an evaluation scheme in which each data point is used for testing before training the learning algorithm. Therefore, it is not necessary to know when a drift occurred to perform the evaluation, which is useful when using real-world datasets that do not have labeled drifts (Lu et al., 2018). Regarding the metrics used to evaluate the drift detector, one may refer to the following ones: TPs, FPs and FNs (and hence *precision*, *recall* and *F1-score*) in the detection rates, as well as the *delay* of the detection (Lu et al., 2018; Dasu et al., 2006).

4.4.2 Related Works

ADWIN (Bifet and Gavalda, 2007) (for "ADaptive WINdowing") is an algorithm with a sliding window W that takes as input a confidence level $\delta \in (0, 1)$ and an unbounded sequence of real values $X = \langle r_1, r_2, ..., r_i, ... \rangle$ from a data stream. Each input r_i is expected to be in $[0, 1]$ and is stored in a finite window $W \subset X$. The idea of

the algorithm is that a concept drift occurs whenever there are two subsequent sub-windows of W with an average difference in their mean above ϵ_{cut}, which is guaranteed to yield a *confidence level* of δ. If a drift is detected, a part of W is dropped, and the algorithm is ready to resume the drift detection. ADWIN has guaranteed bounds on all error distributions. However, the observed errors usually tend to follow a normal distribution. Thus, they tightened their ϵ_{cut} accordingly and evaluated the algorithm taking those new bounds. Furthermore, as ADWIN checks multiple sub-windows of W, its complexity is $\mathcal{O}(\log |W|)$ per iteration (where an iteration is ideally triggered for each new element that arrives from the stream).
DDM (Gama et al., 2004) (for "Drift Detection Method") also receives as parameter a value δ related to the confidence level of the algorithm, and a sequence of data instances X which are available one by one from a data stream. However, the data stream is expected to follow a binomial distribution, as its values represent the number of errors in a sample of n instances. Therefore, it is primarily suited for classification problems rather than for regression. The main idea of DDM is that, as the number of samples increases, the number of errors from a learner will decrease. Therefore, if the error rate from the learner increases above a certain threshold, it means that a concept drift occurred. Thus, DDM tracks the minimal error rate p_{min} from X and its respective standard deviation s_{min}. Then, DDM flags a drift at step $i \geq 30$ iff $p_i + s_i \geq p_{min} + \delta s_{min}$. When a drift is detected, the algorithm is reset. One advantage of DDM is its lightweight nature due to storing only p_{min} and s_{min}. However, DDM is known for its decreased performance when detecting slow gradual changes.
EDDM (Baena-Garcıa et al., 2006) (for "Early Drift Detection Method") is similar to DDM but uses the distance between errors to identify concept drifts. Therefore, EDDM is suited only for binary input data, which does not allow it to be used for regression problems. They assume that, as the number of examples increases, the distance between errors will increase accordingly. Therefore, if this distance decreases more than a threshold, a concept drift is flagged. Thus, EDDM tracks the maximum distance between errors p'_{max} and its respective standard deviation s'_{max}. Then, after analyzing at least 30 errors, EDDM flags a drift at step t if $(p'_t + 2\,s'_t)/(p'_{max} + 2\,s'_{max} < \delta)$, with δ again representing a simple form of confidence level for the detected concept drifts. According to the authors, EDDM improves DDM's performance when identifying gradual concept drifts while maintaining good performance on sudden drifts.
STEPD (Nishida and Yamauchi, 2007) (for using the "Statistical Test of Equal Proportions") is a lightweight drift detector that also considers a sequence of data instances X as input. The idea of STEPD is that the recent accuracy of a learner shall be similar to its overall accuracy, otherwise a concept drift occurred. As the name suggests, STEPD checks the equality-of-proportions hypothesis test (Massey and Miller, 2006) by comparing the most recent values from X (kept in a sliding window W) with the other values in X, thus comparing $X_{0:W}$ and $X_{W:|X|-1}$. It also considers that the data follows a normal distribution, and with a confidence level of δ, it can determine if the null hypothesis is rejected and a concept drift is flagged. In this case, the algorithm is reset. The value of W selected by the authors was 30.
ECDD (Ross et al., 2012) is the acronym for "Exponentially Weighted Moving Average" (EWMA) for concept-drift detection. It is a drift-detection method based on EWMA that checks the misclassification rate of its underlying learners. ECDD receives a sequence of data instances X as input, which it assumes to follow a Bernoulli distribution. ECDD is suited only for classification problems, as it only accepts binary data from the data stream. Other than X, ECDD takes as inputs ARL_0, which

is the time one expects between detecting false positives; and λ, which dictates how much weight new data has compared to the old one. Despite being computationally expensive to calculate, ARL_0 can be approximated before starting to process the data stream. ECDD then uses pre-calculated polynomials, ARL_0 and λ to update its estimators. With its updated estimators, ECDD detects if a concept drift occurred or not. Finally, if a drift is detected, the algorithm is reset.

We note that all algorithms described above additionally have a form of warning detection which can be used to assist in the concept drift adaptation. The warning is usually a simple relaxation of the drift threshold which can be used to start training a new learner before the actual concept drift is flagged.

4.5 OPTWIN – Optimal Window Concept Drift Detector

OPTWIN is the new concept-drift detector which we propose in this paper. It is an error rate-based drift detector, which tracks the error rates produced by an OL learner in a sliding window W. Its name stands for "OPTimal WINdow" due to its calculation of the optimal split of the sliding window. We consider this split "optimal" because it is detected for the first element in W at which a statistically significant difference between either the means or the variances of the error rates (based on the common t- and the f-tests) among two sub-windows of W occurs. Specifically, W keeps growing until either a concept drift is detected or a maximum user-defined size w_{max} is reached. Based on the sliding-window size $|W|$ and a predefined confidence level δ, OPTWIN calculates the optimal point to divide W into "historical" (W_{hist}) and "new" (W_{new}) data points.

4.5.1 Setup & Parameters

Without loss of generality, we consider a typical OL scenario, in which OPTWIN receives as input a sequence of real numbers $r_1, r_2, ..., r_i, ...$ from a possibly unbounded data-stream X. Thus, the algorithm is assumed to access only one element r_i at time i from the stream, and it buffers previously seen elements in a *sliding window* $W \subset X$ of consecutive events. Moreover, OPTWIN requires as parameters (1) δ, the *confidence level* for the concept-drift detection; (2) w_{max}, the *maximum size* of the sliding window W; and (3) ρ, a parameter we refer to as *robustness*, which we define as the minimum ratio by which $\mu_{W_{new}}$ has to vary in relation to $\sigma_{W_{hist}}$ in order for OPTWIN to consider this variation as a concept drift.

4.5.1.1 Assumptions

- There is no concept drift within the first w_{min} data instances from X (which is needed to initialize OPTWIN).
- A concept drift occurs when the means or the standard deviations within two sub-windows of W are statistically different.
- Within any two sub-windows of W, the values produced by the test statistic of the unequal-variance t-test (Ruxton, 2006) (henceforth called "*t_value*") follow a t-distribution.
- Within any two sub-windows of W, the values produced by the test statistic of the f-test (Mahbobi and Tiemann, 2016) (henceforth called "*f_value*") follow an f-distribution.

Our first assumption mitigates the negative impact of outliers when only a few data points are available. The second assumption indicates when we expect OPTWIN to identify concept drifts. The third and the fourth assumptions are generally required to guarantee the validity of the t- and f-tests, respectively, but do not impose limitations on the values ingested from the data stream in practice.

4.5.2 Algorithm

Algorithm 2 OPTWIN

```
Input parameters:                                   Global variables:
• $\delta$ – confidence level                       $W = \langle \rangle$ – sliding window
• $\rho$ – robustness                               $w_{min} = 30$ – min window size
• $w_{max}$– max window size                        $\beta = 1e^{-5}$ – avoids division by 0

 1: procedure AddElement($r_i$)
 2:     $W \leftarrow W \cup r_i$
 3:     if $|W| < w_{min}$ then
 4:         return False
 5:     else if $|W| \geq max\_lenght$ then
 6:         $W \leftarrow W - W_0$
 7:     $\nu \leftarrow$ OptimalCut($|W|, \rho, \delta^{\frac{1}{4}}$)                cf. Equation (4.1)
 8:     $\nu_{split} \leftarrow \lfloor \nu\,|W| \rfloor$
 9:     $W_{hist} \leftarrow W_{0:\nu_{split}}$
10:     $W_{new} \leftarrow W_{\nu_{split}:|W|-1}$
                                                                // f-test
11:     if $\frac{(\sigma_{W_{new}}+\beta)^2}{(\sigma_{W_{hist}}+\beta)^2} > f\_ppf(\delta^{\frac{1}{4}}, \nu|W|-1, (1-\nu)|W|-1)$ then
12:         reset()
13:         return True
                                                                // t-test
14:     else if $t\_value(W_{hist}, W_{new}) > t\_ppf(\delta^{\frac{1}{4}}, df)$ then
15:         reset()
16:         return True
```

As seen in Algorithm 2, OPTWIN first needs to be initialized by creating an empty sliding window W. Analogously to other drift detectors, it collects w_{min} many first elements from the stream (usually within 30 to 50) to guarantee that it has a minimum amount of data to output statistically relevant drift detections. It also defines a small constant $\beta = 1e^{-5}$, which avoids a division by 0 when added to the standard deviations during the calculation of the f-test.

Second, the AddElement procedure is called once to process each data element received from the data stream. The procedure starts by inserting the most recent data element r_i into W and by checking whether W has already reached the minimum amount of elements to detect a possible concept drift. If so, it also checks if W increased above w_{max} and removes the oldest element from W if necessary. Up to this point, OPTWIN performs standard queuing operations to maintain its sliding window bounded between w_{min} and w_{max} elements.

Then, OPTWIN calculates ν, which is the optimal splitting point of W into W_{hist} and W_{new}, by solving Equation 4.1 for the highest value of ν in terms of δ', ρ and $|W|$:

$$\rho = t\_ppf(\delta', df)\sqrt{\frac{1}{\nu|W|} + \frac{f\_ppf(\delta', \nu\,|W|-1, (1-\nu)\,|W|-1)}{(1-\nu)\,|W|}} \tag{4.1}$$

where ρ is the aforementioned, user-defined robustness parameter. During this calculation, OPTWIN uses t_ppf and f_ppf, which are the Probability Point Functions (PPF) of the t- and the f-distributions, respectively. To calculate t_ppf, we apply a confidence of $\delta' = \delta^{\frac{1}{4}}$, which considers the application of the two tests to calculate ν in Equation 4.1 and the two tests on Lines 11 and 14 in Algorithm 2.

Finally, OPTWIN performs the t- and f-tests (in any order) to compare both sub-windows and determine if their means and standard deviations, respectively, belong to the same distribution. If not, a concept drift is flagged and the algorithm is reset. Otherwise, the AddElement method is called for the next element from the data stream in an iterative manner.

The usage of the t- and f-tests to identify significant changes in the means and standard deviations among series of data values is well-established. However, the novelty of OPTWIN comes from the combination of both tests to calculate ν, and thereby identify where to optimally divide this series of values to perform both tests. In short, by solving Equation 4.1 in terms of ν, we determine the minimum size of W_{new} (the optimal splitting point of W) that statistically guarantees the identification of any concept drift with a robustness of at least ρ when using the t- and f-tests, thereby achieving lower drift-detection delays than other approaches. To calculate ν, other than the confidence level δ, the only values that shall be inputted by the user are w_{max} and ρ. Regarding ρ, when selecting a small value, one may expect to identify smaller drifts in exchange of a higher drift-detection delay. On the other hand, with higher values of ρ, one can expect a smaller detection delay in exchange of missing smaller drifts. In practice, ρ shall be set based on the magnitude and frequency of drifts expected. However, determining the correct ρ to each use case should not be a difficult task, since different ρ's tend to produce similar results (as seen in Section 4.6). Regarding w_{max}, with a higher value, one may expect smaller drift-detection delays in exchange for more memory usage. In practice, when using $\rho = 0.1$, we did not observe a significant variation in $|W_{new}|$ even when increasing w_{max} to more than 25,000 elements.

One important point to note is that we can only calculate ν as the optimal splitting point if W is large enough. Otherwise, we set ν to the middle of W until it grows to the minimum size needed to solve Equation 4.1. Thus, if it exists, it is defined as the highest root of Equation 4.1. Otherwise, it is set to $\nu = 0.5$.

Below, we present Theorem 4.5.1, our main theoretical result. It gives guarantees in terms of OPTWIN's false positive (FP) and false negative (FN) bounds for identifying concept drifts. Theorem's 4.5.1 proof is available in Section 4.8.1.2.

Theorem 4.5.1.

- ***False Positive Bound.*** *At every step, if μ_W and σ_W^2 remain constant within W, OPTWIN will flag a concept drift at this step with a confidence of at most 1-δ.*
- ***False Negative Bound*** *(for mean drift with large enough W). For any partitioning of W into two sub-windows $W_{hist}\,W_{new}$, with $|W| \geq w_{proof}$ and W_{new} containing the most recent elements, if $\mu_{hist} - \mu_{new} > \rho\,\sigma_{hist}$, then, with confidence δ, OPTWIN flags a concept drift in at most $|W| - \nu_{split}$ steps.*
- ***False Negative Bound*** *(for mean drift with small W). For any partitioning of W into two sub-windows $W_{hist}\,W_{new}$, with $w_{min} \leq |W| < w_{proof}$ and W_{new} containing the*

most recent elements, if $\mu_{hist} - \mu_{new} > \rho_{temp}\,\sigma_{hist}$, then, with confidence δ, OPTWIN flags a concept drift in at most $\frac{|W|}{2}$ steps.

- ***False Negative Bound*** *(for standard-deviation drift with any W). For any partitioning of W into two sub-windows W_{hist} W_{new}, with $|W| \geq w_{min}$ and W_{new} containing the most recent elements, if $\frac{\sigma^2_{new}}{\sigma^2_{hist}} > f\_ppf(\delta', \nu\,|W| - 1, (1-\nu)\,|W| - 1)$, then, with confidence δ, OPTWIN flags a concept drift in at most ν_{split} steps.*

4.5.3 Implementation & Analysis

We implemented OPTWIN via a combination of Java and Python scripts as extensions of the MOA (Bifet et al., 2010) (Java) and the River (Montiel et al., 2021) (Python) libraries (both available on Github[2]). We pre-calculated the values of ν, *t_ppf*, and *f_ppf* based on a fixed confidence value of $\delta = 0.99$ and for $w_{max} = 25{,}000$, thus $30 \leq |W| \leq 25{,}000$. All pre-calculated variables were stored in lists indexed by the window sizes $|W|$ from which they were calculated. This is possible because those variables, as seen in Equation 4.1, do not depend on the actual error distribution. Thus, it is not necessary to calculate them in real-time. On the other hand, we need to store the sliding window W plus the other three lists of floating point values of size up to w_{max} in memory, which takes a maximum of $w_{max} * 4 * 4$ bytes. Thus, for $w_{max} = 25{,}000$, OPTWIN would require only around 390 KB of memory.

Furthermore, the full calculation of the means and standard deviations from W_{hist} and W_{new} can be avoided. Instead of calculating them from scratch, one only needs to update them incrementally. Moreover, as W is bounded by w_{max}, we can use a circular array to make insertions at the end of the array, deletions from the beginning of the array, and then look up each value in $\mathcal{O}(1)$ time. Therefore, the ADDELEMENT procedure has an overall computational complexity of $\mathcal{O}(1)$ per element ingested from the data stream (assuming a constant cost for the numerical operations involved in resolving Equation 4.1 to ν). Our analytical approach thus provides great potential runtime gains over the iterative search procedure applied, e.g., by ADWIN which requires $\mathcal{O}(\log|W|)$ computations per iteration.

By default, OPTWIN's Algorithm 2 tracks concept drifts that either increase or decrease the means and standard deviations of the variables tracked. However, in an OL scenario, we usually want to update the learner only when the number of errors increases. Therefore, in our implementation, we check if also $\mu_{new} \geq \mu_{hist}$ along with the statistical tests on Lines 11 and 14 of Algorithm 2. In doing so, we consider that a drift occurred only if μ_{new} is higher than μ_{hist} (i.e., when the learner decreased in performance).

4.6 Experiments & Results

In this section, we report our detailed experiments to asses OPTWIN. We compared OPTWIN to the most commonly used baselines for concept-drift detection: ADWIN, DDM, EDDM, STEPD, and ECDD. We additionally investigated more recent techniques (Hidalgo, Mariño, and Barros, 2019; Wu et al., 2021; Barros et al., 2017) but found them transitively compared to the classical techniques mentioned above. That is because they address specific frailties of the classical drift detectors like recurrent drifts and decreased sensitivity when concepts are long, often without statistical improvements over the classical techniques.

[2]`https://github.com/maurodlt/optwin`

We performed most of the experiments using the common MOA (Bifet et al., 2010) framework which is a Java-based data stream simulator. In addition, we performed experiments on Python to asses OPTWIN on a Neural Network (NN) use case, which would not be possible via MOA.

4.6.1 MOA Experiments

We compared 3 configurations of OPTWIN with the default configurations of our baselines. All OPTWIN configurations had $\delta = 0.99$, $w_{max} = 25{,}000$, and pre-computed values for ν, t_ppf, and f_ppf (as described on Section 4.5.3). The difference among the OPTWIN configurations is only on ρ, which we varied between 0.1, 0.5, and 1.0 to better understand how our robustness parameter affects OPTWIN in practice.

We performed two types of experiments on MOA. The first one uses the “Concept Drift” interface, in which MOA creates a stream of data points (either binary or non-binary) and produces a concept drift that can be sudden or gradual. We later refer to those experiments according to their data input and their drift type.

The second type of experiment uses the “Classification” interface. It generates data streams based on both synthetic datasets (STAGGER (Schlimmer and Granger, 1986), RANDOM RBF (Bifet et al., 2009), and AGRAWL (Agrawal, Imielinski, and Swami, 1993)) and real-world ones (Electricity (Blackard and Dean, 1999) and Covertype (Harries, 1999)). The idea of these experiments is to train a classifier that is reset every time a concept drift is detected by a drift detector. We chose MOA’s built-in Naïve Bayes (NB) classifier for its simplicity, which facilitates the analysis of the drift-detection results. For the synthetic datasets, we generate data streams with 100,000 data points with drifts occurring every 20,000 data points (either sudden or gradual). For the real-world data sets, the drifts are already present and have an unknown location on the stream.

Tables 4.1 and 4.2 present the comparison among drift detectors on the “Concept Drift” and the “Classification” interfaces respectively. We repeated each experiment 30 times and compared their average TP, FP, and FN rates to compute their micro-average precision, recall, and F1-score, along with their average drift-detection delay (in terms of the number of streamed elements between the occurrence of a known concept drift and its identification by the detector). Note that we did not include in Table 4.2 real-world datasets because it is not possible to calculate the above-mentioned metrics without knowing the drift’s locations, which are only given on the synthetic datasets. Moreover, when using the “Classification” interface, we observed a divergence between the starting and ending points of gradual concept drifts in the MOA documentation and in practice. Therefore, we did not include those configurations in Table 4.2.

Table 4.3 shows the average metrics obtained by each drift detector on Tables 4.1 and 4.2. Based on the results aggregated in Table 4.3, we can observe a high F1-score for OPTWIN when compared to the baselines. In fact, with $\rho \leq 0.5$, OPTWIN’s average F1-score is over 95%, followed by OPTWIN with $\rho = 1.0$ achieving 89.4%, DDM with 85.6%, ADWIN reaching 67% and the other detectors less than 50%. We assert this due to OPTWIN’s low FP rate which produces a high precision and F1-score, respectively. We further compared the F1-scores of all configurations of OPTWIN with the two drift detectors that can be used for regression problems (ADWIN and STEPD), which showed OPTWIN to be superior based on a one-tailed Wilcoxon signed-rank test with $\alpha = 0.05$ in a statistically significant manner.

		gradual drift			sudden drift		
Setting	**Detector**	**Delay**	**FP**	**F1**	**Delay**	**FP**	**F1**
binary data	ADWIN	280	16.33	60%	46.33	1.87	52%
	DDM	365	0.37	93%	64.50	0.20	91%
	EDDM	148	6.33	71%	26.77	6.57	23%
	STEPD	180	8.00	37%	**0.40**	6.67	83%
	ECDD	**117**	5.37	42%	5.47	2.67	42%
	OPTWIN$_{\rho=0.1}$	328	0.30	94%	75.13	**0.00**	**100**%
	OPTWIN$_{\rho=0.5}$	278	0.10	98%	28.17	**0.00**	**100**%
	OPTWIN$_{\rho=1.0}$	317	**0.07**	**99**%	18.33	0.33	86%
non-binary data	ADWIN	145	**0**	**100**%	25	2	50%
	DDM	-	-	-	-	-	-
	EDDM	-	-	-	-	-	-
	STEPD	41	150	18%	21	32	6%
	ECDD	-	-	-	-	-	-
	OPTWIN$_{\rho=0.1}$	**2**	**0**	**100**%	**1**	**0**	**100**%
	OPTWIN$_{\rho=0.5}$	**2**	**0**	**100**%	**1**	**0**	**100**%
	OPTWIN$_{\rho=1.0}$	**2**	**0**	**100**%	**1**	**0**	**100**%

TABLE 4.1: Statistics of drift identification in classification experiments on synthetic datasets.

Setting	**Detector**	**Delay**	**FP**	**F1**
sudden STAGGER	ADWIN	31.00	0.30	96%
	DDM	6.33	0.17	98%
	EDDM	40.01	**0.00**	**100**%
	STEPD	17.39	31.40	20%
	ECDD	0.72	0.47	94%
	OPTWIN$_{\rho=0.1}$	0.76	0.27	97%
	OPTWIN$_{\rho=0.5}$	**0.72**	0.43	95%
	OPTWIN$_{\rho=1.0}$	**0.72**	0.60	93%
sudden RANDOM RBF	ADWIN	169	6	46%
	DDM	536	2	67%
	EDDM	**53**	11	33%
	STEPD	281	25	24%
	ECDD	71	174	4%
	OPTWIN$_{\rho=0.1}$	315	**0**	**86**%
	OPTWIN$_{\rho=0.5}$	187	**0**	**86**%
	OPTWIN$_{\rho=1.0}$	1,931	1	57%
sudden AGRAWAL	ADWIN	**229**	4.23	65%
	DDM	1,875	0.90	79%
	EDDM	5,370	17.20	20%
	STEPD	838	23.47	25%
	ECDD	465	153.6	5%
	OPTWIN$_{\rho=0.1}$	371	0.63	**93**%
	OPTWIN$_{\rho=0.5}$	350	0.77	88%
	OPTWIN$_{\rho=1.0}$	630	**0.27**	91%

TABLE 4.2: Statistics of drift identification in regression experiments on synthetic datasets.

	Average		
Detector	**Delay**	**FP**	**F1**
ADWIN	132.19	4.39	67%
DDM	569.37	0.73	86%
EDDM	1127.56	8.22	49%
STEPD	196.97	34.93	30%
ECDD	131.84	67.22	37%
OPTWIN$_{\rho=0.1}$	156.13	**0.17**	**95%**
OPTWIN$_{\rho=0.5}$	**120.98**	0.19	**95%**
OPTWIN$_{\rho=1.0}$	414.29	0.32	89%

TABLE 4.3: Average statistics of drift identification on synthetic datasets.

Regarding the drift-detection delay, OPTWIN with $\rho = 0.5$ is the drift detector with the smallest delay, taking on average 121 iterations to detect a concept drift; followed by ECDD, ADWIN, and OPTWIN at $\rho = 0.1$ taking 131, 132 and 156 iterations each; the other detectors took on average between 197 and 1,128 iterations to detect a drift, with OPTWIN at $\rho = 1.0$ taking 414 on average. Observe that, the higher the FP rate of a drift detector, the higher are also its chances of identifying a drift earlier. Therefore, when analyzing ECDD's average drift delay of 131 iterations, we have to consider that it had an average of 67 FPs per run, in contrast to an average of less than 0.4 for any of the OPTWIN configurations. Furthermore, considering that DDM, EDDM, and ECDD depend on binary data, they were not included in the experiments using "non-binary" datasets.

We can better visualize the difference between the drift detectors in Figure 4.2, which represents one of the 30 runs of the "sudden binary drift" configuration (we selected the run with results closest to the ones on the observed average). In Figure 4.2, we can see the relatively high FP rate of the EDDM, ADWIN and ECDD detectors when compared to OPTWIN, DDM and STEPD. Moreover, we can see that OPTWIN's detection delay decreases as ρ increases.

In Figure 4.3, we can once again see OPTWIN and DDM identifying all the drifts with a low FP rate. However, in this scenario, ADWIN also produces only a few FPs, which we noticed to occur immediately after correctly identifying a concept drift. This is due to the time it takes to adapt its window size and remove data points from before the concept drift. In this use case, ECDD and STEPD produce almost "random guesses" on the location of the concept drifts.

Additionally, we provide illustrations of the remaining results described in Table 4.4 in Section 4.8.2.

4.6.2 Classification Experiments

In the "Classification" experiments (still using MOA as platform), we can analyze the average accuracy achieved by the NB classifier when varying the drift detectors (cf. Table 4.4). The idea is that the better a drift detector's performance, the better the classifier can adapt to the concept drifts, thus generating a higher prediction accuracy. However, the fast and accurate detection of the drifts did not always traverse to a better accuracy for the classifier. We can observe this by comparing Tables 4.3 and 4.4, in which experiments using drift detectors that produced a low F1-score in Table 4.3 achieved a good accuracy in Table 4.4. Moreover, the drift detectors with the best accuracy on real-world data sets were the ones that detected more

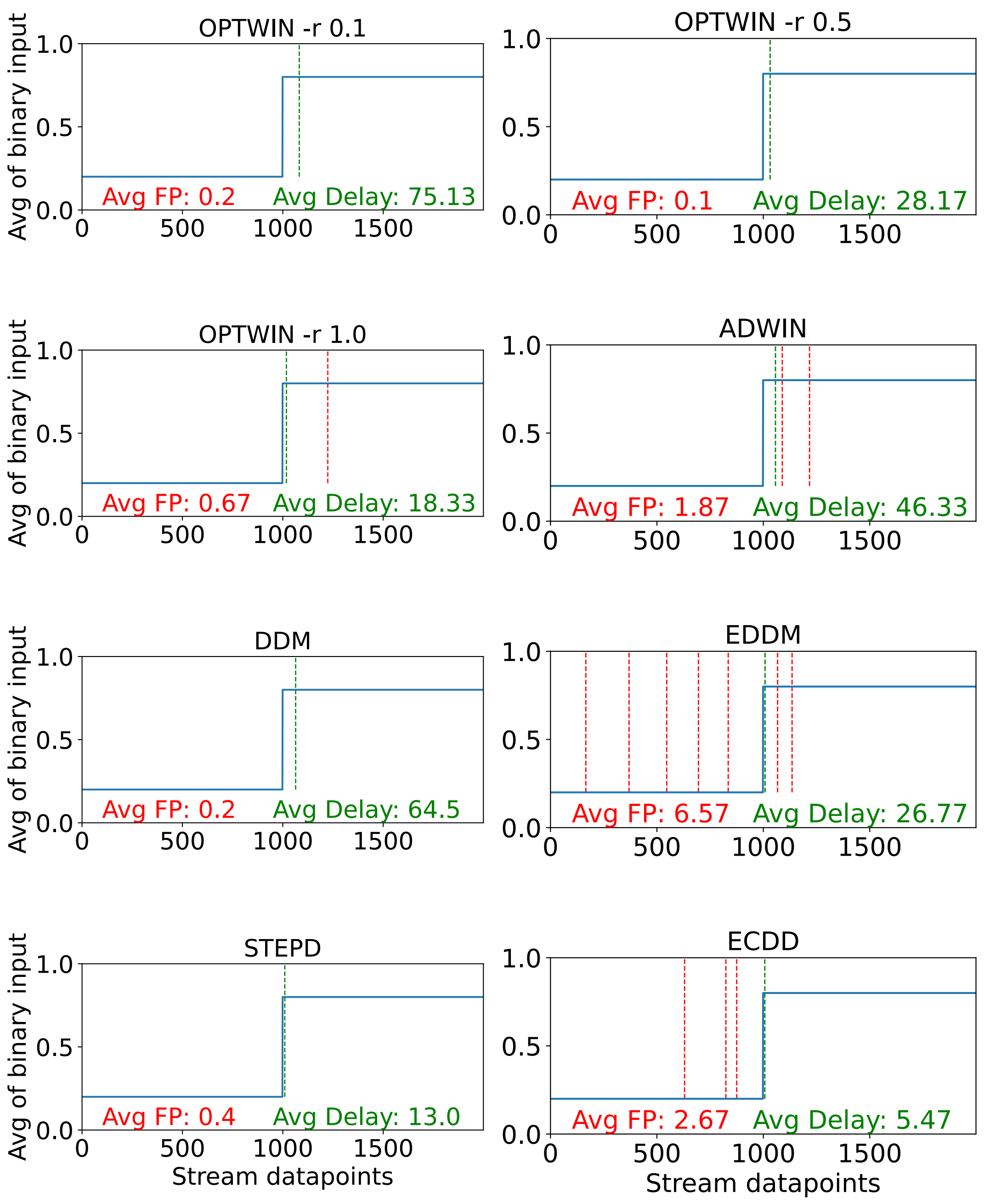


FIGURE 4.2: Sudden binary drift detection with average FP rates compared to drift-detection delays.

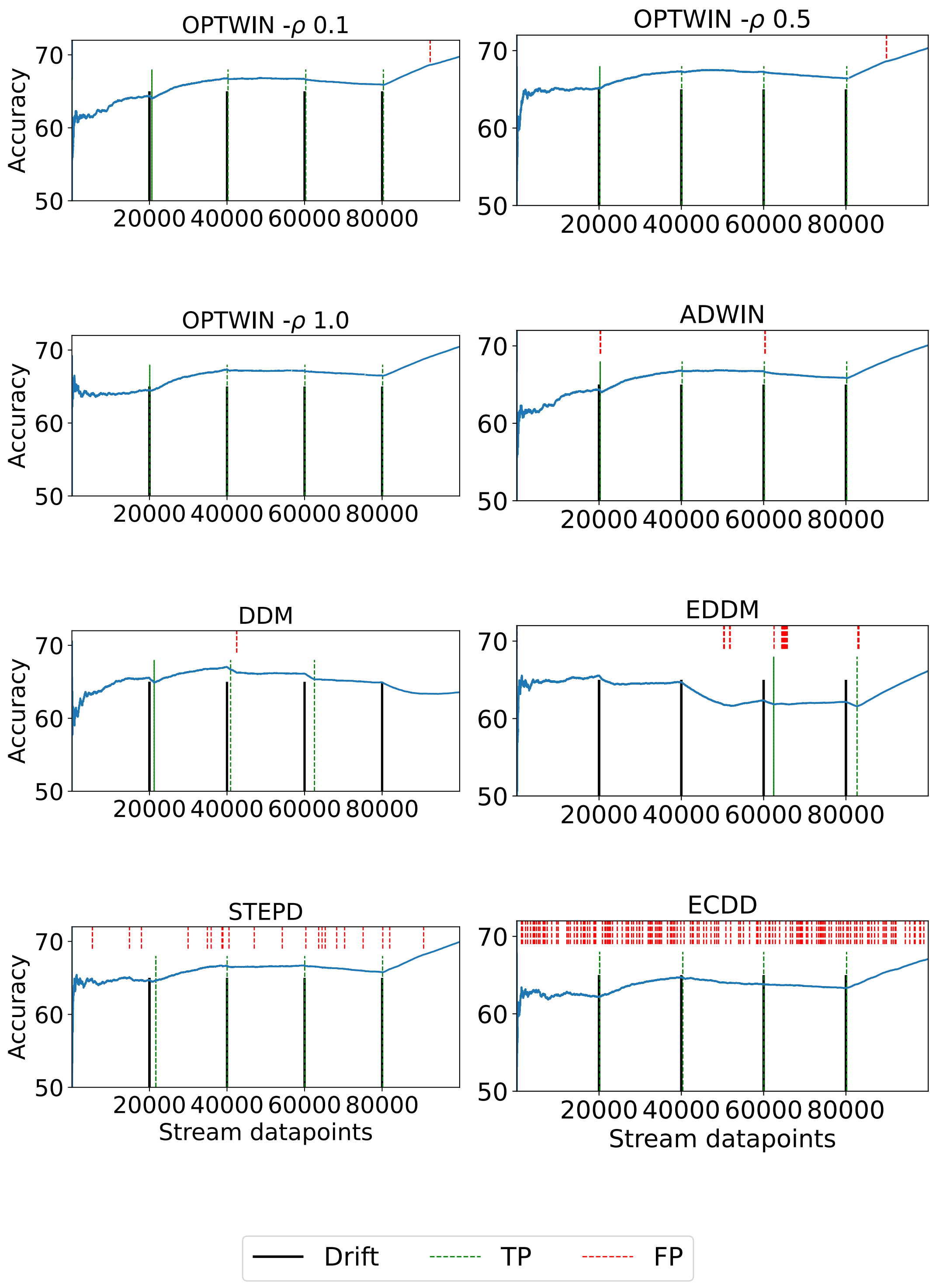


Figure 4.3: TP and FP rates for drift detection on the AGRAWAL dataset with sudden concept drifts.

drifts, which is why ECDD achieved such good accuracy (being the detector with the higher amount of FPs). For example, ECDD detected 426 drifts on the Electricity dataset—over twice the amount of other algorithms.

Drift	STAGGER		Random RBF		AGRAWL		Elec	Cove
Detector	Sud	Grad	Sud	Grad	Sed	Grad		
None	66.31	66.31	58.69	58.69	57.94	57.93	73.36	60.52
ADWIN	99.89	98.78	69.74	69.48	**70.22**	69.89	80.01	82.49
DDM	**99.96**	98.67	68.18	68.19	65.53	65.80	81.18	88.03
EDDM	99.87	**98.83**	65.99	66.92	65.46	65.31	84.83	86.08
STEPD	98.74	98.78	**70.92**	**70.49**	69.48	69.13	84.49	87.53
ECDD	**99.96**	98.71	68.64	68.52	67.03	66.80	**86.76**	**90.16**
OPTWIN$_{\rho=0.1}$	**99.96**	98.72	69.65	69.54	70.11	**69.94**	79.99	83.29
OPTWIN$_{\rho=0.5}$	**99.96**	98.51	69.77	69.48	69.91	69.27	83.30	85.63
OPTWIN$_{\rho=1.0}$	**99.96**	98.36	69.67	68.48	69.44	68.61	82.96	86.31

TABLE 4.4: Accuracy of NB on synthetic and real-world datasets. "Sud"and "Grad" representing sudden and gradual concept drifts respectively. "Elec" and "Cove" representing the Electricity and the Covertype datasets.

4.6.3 Regression Experiments on Neural Networks

To further explore OPTWIN's behavior in a regression scenario, we compared it with ADWIN for identifying drifts from the loss of a CNN. We chose ADWIN as our baseline because it was the drift detector with the best F1-score and smallest drift-identification delay among the ones that do not require binary inputs (thus excluding DDM, EDDM, and ECDD). To simplify the reader's understanding of the generation of the concept drift, instead of training a regression problem on a CNN, we here focused on image classification by swapping the labels of the images. Thus, we provoked 4 concept drifts by swapping the labels of two classes of images every 20% of the simulated data stream. For example, after 62,480 iterations, we swapped the labels between images from "cats" to "horses". Nevertheless, as the drift detectors track the loss of the CNN, the type of the problem or the network architecture should not affect the experiment.

We pre-trained an image classification model (TensorFlow, 2022a) with the CIFAR-10 (Krizhevsky, Hinton, et al., 2009) data set during 100 epochs, achieving an average of 89% training accuracy over 3 different runs. Then, we simulated an OL scenario with concept drifts. Our data stream was formed by batches of 32 images from the CIFAR-10 dataset; with a total of 312,400 data points (equivalent to 100 epochs). We simulated our 4 concept drifts by swapping the labels of two classes every 62,480 iterations (the equivalent of 20 epochs). At every iteration, the model classified the 32 images and outputted the loss of the batch. We inputted this loss into the drift-detection algorithm. If a drift was detected, the next 9,372 batches of images (the equivalent of 3 epochs) were used for fine-tuning the model (thus adapting it to the concept drift). Therefore, the goal was for the drift detector to identify the 4 concept drifts that we simulated, triggering the fine-tuning of the model for a total of 12 epochs (3 epochs per drift).

In Figure 4.4, we compare OPTWIN and ADWIN under the setting described above. First, we note that ADWIN's high FP rates made it re-train the model for much longer than OPTWIN. In comparison, ADWIN identified 15 concept drifts (with 11 FPs), thus triggering model fine-tuning for 61,562 iterations which took in total 945 seconds. In contrast, OPTWIN identified just 5 drifts (with 1 FP), thus triggering the model fine-tuning for 23,430 iterations which took 781 seconds. We note that OPTWIN's running time per iteration is superior to ADWIN, $1e^{-5}$ against $6e^{-6}$ seconds. However, OPTWIN still ends up speeding up the OL pipeline whenever re-training is triggered by the concept-drift detection (21% faster in this use case). This is because the training of a learner is usually computationally more expensive than the drift detection. Thus, with fewer FPs, the total training time can be reduced substantially.

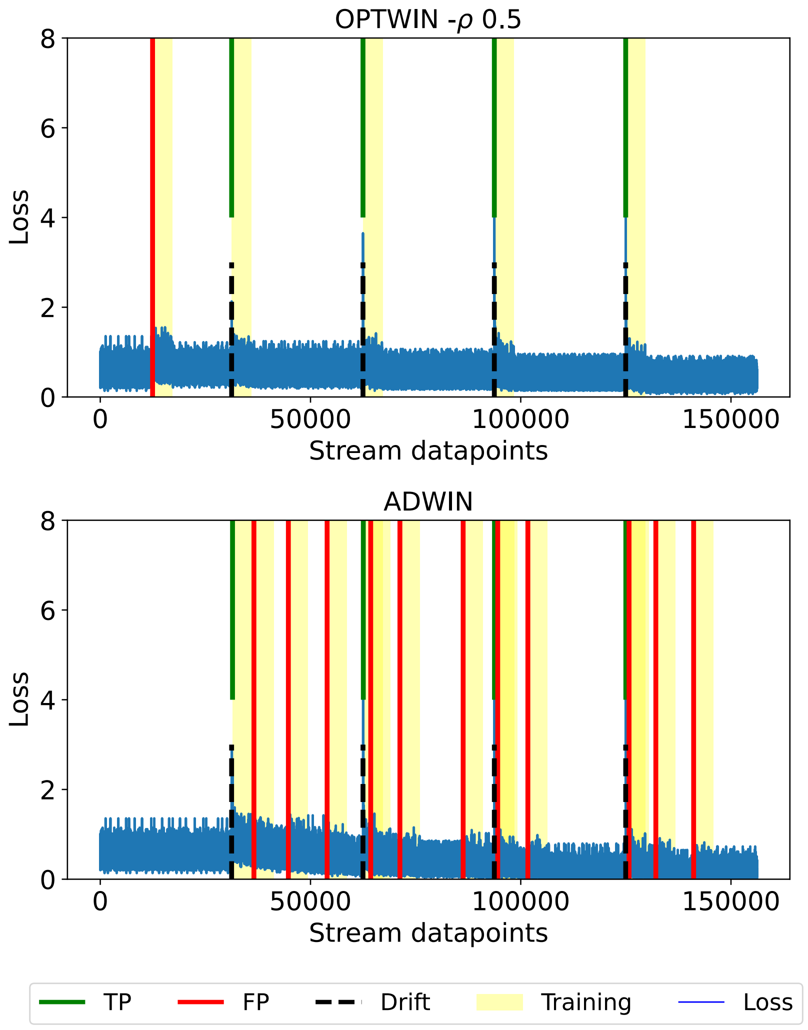


FIGURE 4.4: Sudden drift detection over the loss of a CNN.

4.7 Conclusion

In this paper, we presented OPTWIN, a novel concept-drift detector that uses a sliding window of errors to identify concept drifts in both classification and regression problems with a low FP rate. OPTWIN's novelty relies on the assumption that changes also in the variances of the elements ingested from a data stream can be an indication of concept drift. This assumption lets it optimally divide its sliding window in $\mathcal{O}(1)$ time by applying the t- and f-tests to determine whether a concept drift occurred. To assess OPTWIN, we compared it with 5 popular drift detectors in

11 different experiments. As a result, OPTWIN achieved higher F1-scores in most of our experiments. In fact, OPTWIN had the best F1-score (with statistical significance) while maintaining a similar drift-detection delay compared to the drift detectors suited for both classification and regression problems. Moreover, OPTWIN could speed up the overall OL pipeline of a CNN by 21% (compared to ADWIN) due to its low FP rate. To conclude, we conjecture that OPTWIN's characteristics can enable even higher speed-ups in scenarios where the drift identification triggers the re-training of complex models.

4.8 Supplementary Material

4.8.1 Proof

4.8.1.1 Definition of concepts

- δ: confidence level defined by the user, $\delta \in (0, 1)$.
- W: sliding window of error rates.
- w_{proof}: minimum size of W to identify drifts of ρ magnitude.
- w_{min}: minimum W size, with $w_{min} = 30$.
- w_{max}: maximum W size, with $w_{max} \in [w_{min}, \infty)$.
- μ_W: mean of W.
- σ_W: standard deviation of W.
- ν: optimal splitting percentage of W, with $\nu \in (0, 1)$.
- ν_{split}: optimal splitting point of W, with $\nu_{split} = \lfloor \nu \, |W| \rfloor$
- W_{hist}: historical errors from W, defined as $W_{0:\nu_{split}}$.
- W_{new}: new errors from W, defined as $W_{\nu_{split}:|W|-1}$.
- ρ: robustness parameter, i.e., the ratio that $\mu_{W_{new}}$ has to vary in relation to $\sigma_{W_{hist}}$ to count as a concept drift, with $\rho\ in(0, \infty)$.
- ρ_{temp}: temporary ρ, defined by solving Equation 4.1, with $\nu = 0.5$.
- $X = r_1, r_2, ..., r_i, ...$: data-stream of real numbers r_i.

4.8.1.2 Proof of Theorem 4.5.1

First, let us recall Theorem 4.5.1.

Theorem 4.5.1.

- ***False Positive Bound.*** *At every step, if μ_W and σ^2_W remain constant within W, OPTWIN will flag a concept drift at this step with a confidence of at most 1-δ.*
- ***False Negative Bound*** *(for mean drift with large enough W). For any partitioning of W into two sub-windows W_{hist} W_{new}, with $|W| \geq w_{proof}$ and W_{new} containing the most recent elements, if $\mu_{hist} - \mu_{new} > \rho\, \sigma_{hist}$, then, with confidence δ, OPTWIN flags a concept drift in at most $|W| - \nu_{split}$ steps.*

- ***False Negative Bound*** *(for mean drift with small W). For any partitioning of W into two sub-windows $W_{hist}\, W_{new}$, with $w_{min} \leq |W| < w_{proof}$ and W_{new} containing the most recent elements, if $\mu_{hist} - \mu_{new} > \rho_{temp}\, \sigma_{hist}$, then, with confidence δ, OPTWIN flags a concept drift in at most $\frac{|W|}{2}$ steps.*
- ***False Negative Bound*** *(for standard-deviation drift with any W). For any partitioning of W into two sub-windows $W_{hist}\, W_{new}$, with $|W| \geq w_{min}$ and W_{new} containing the most recent elements, if $\frac{\sigma^2_{new}}{\sigma^2_{hist}} > f\_ppf(\delta', \nu\, |W| - 1, (1-\nu)\, |W| - 1)$, then, with confidence δ, OPTWIN flags a concept drift in at most ν_{split} steps.*

As seen above, Theorem 4.5.1 is based on ρ, which we defined in Equation 4.1 and we recall it below.

$$\rho = t\_ppf(\delta', df)\sqrt{\frac{1}{\nu|W|} + \frac{f\_ppf(\delta', \nu\,|W|-1, (1-\nu)\,|W|-1)}{(1-\nu)\,|W|}} \tag{4.1}$$

Proof.

- **Part 1:** In each iteration of OPTWIN, we apply the t- and the f-tests to identify the concept drifts. Therefore, the false-positive bounds of OPTWIN are directly drawn from those tests. Thus, with $\delta' = \delta^{\frac{1}{4}}$ as the confidence for each of the four tests, we identify a false-positive drift with a confidence of at most $1-\delta$.

- **Part 2:** Considering an unequal-variance t-test (Ruxton, 2006) by comparing W_{hist} and W_{new}, we have:

$$t\_value = \frac{\mu_{hist} - \mu_{new}}{\sqrt{\frac{\sigma^2_{hist}}{|W_{hist}|} + \frac{\sigma^2_{new}}{|W_{new}|}}} \tag{4.2}$$

The nominator on the right represents the difference observed between the means of W_{hist} and W_{new}. If this difference is large enough, we can statistically guarantee that both sets have distributions with different means. To simplify the equation, we represent this difference in terms of σ_{hist}. Therefore, we set $\mu_{hist} - \mu_{new} = \rho\, \sigma_{hist}$ with ρ being a user-defined robustness parameter. Thus, by applying this substitution and passing the denominator to the other side of the equation, we have:

$$\rho\, \sigma_{hist} = t\_value\sqrt{\frac{\sigma^2_{hist}}{|W_{hist}|} + \frac{\sigma^2_{new}}{|W_{new}|}} \tag{4.3}$$

Considering ν as the splitting point of W into W_{hist} and W_{new}, we have $|W| = \nu\, |W_{hist}| + (1-\nu)\, |W_{new}|$, and therefore:

$$\rho\, \sigma_{hist} = t\_value\sqrt{\frac{\sigma^2_{hist}}{\nu\, |W|} + \frac{\sigma^2_{new}}{(1-\nu)\, |W|}} \tag{4.4}$$

By our definition of a concept drift, σ_{new} and σ_{hist} must follow the same distribution (otherwise a drift occurred). Furthermore, we also consider that they follow the f-distribution, which is usually the case for standard deviations.

Thus, consider the definition of the test statistic for the f-test (Mahbobi and Tiemann, 2016) as shown below:

$$f\_value = \frac{\sigma_{new}^2}{\sigma_{hist}^2} \tag{4.5}$$

We can now determine σ_{new}'s upper bound in terms of σ_{hist} and by defining f_factor as the highest value that the f_value can have while maintaining the null hypothesis of the f-test with confidence δ'.

$$\sigma_{hist}^2 f\_factor \geq \sigma_{new}^2 \tag{4.6}$$

In this case, the f_factor can be calculated using the f_ppf, the PPF of the F-curve, by taking a confidence of δ' and $|W_{new}| - 1$ and $|W_{hist}| - 1$ as the degrees of freedom for the numerator and the denominator, respectively (cf. Equation 4.5). Thus, we obtain:

$$f\_factor = f\_ppf(\delta', \nu\,|W| - 1, (1-\nu)\,|W| - 1) \tag{4.7}$$

Therefore, to later simplify Equation 4.4, we substitute σ_{new} with its upper bound as calculated by the f-test on Equation 4.6:

$$\rho\,\sigma_{hist} = t\_value\sqrt{\frac{\sigma_{hist}^2}{\nu\,|W|} + \frac{\sigma_{hist}^2 f\_factor}{(1-\nu)\,|W|}} \tag{4.8}$$

Next, we can simplify the equation by removing σ_{hist} from both sides as follows:

$$\rho = t\_value\sqrt{\frac{1}{\nu\,|W|} + \frac{f\_factor}{(1-\nu)\,|W|}} \tag{4.9}$$

Then, we calculate the t_value in Equation 4.10 analogously by using the PPF of the T-curve.

$$t\_value = t\_ppf(\delta', df) \tag{4.10}$$

We consider a confidence level of δ' and the degrees of freedom df for the unequal-variance t-test (cf. Equation 4.11).

$$df = \frac{\left(\frac{\sigma_{hist}^2}{|W_{hist}|} + \frac{\sigma_{new}^2}{|W_{new}|}\right)^2}{\frac{\left(\frac{\sigma_{new}^2}{|W_{hist}|}\right)^2}{|W_{hist}|-1} + \frac{\left(\frac{\sigma_{new}^2}{|W_{new}|}\right)^2}{(|W_{new}|)-1}} \tag{4.11}$$

By considering Equation 4.11 in our setting with $|W_{hist}| = \nu\,|W|$ and $|W_{new}| = (1-\nu)\,|W|$ and our upper bound for $\sigma_{new} \leq \sigma_{hist} f\_factor$ (cf. Equation 4.6), we reach Equation 4.12, which depicts our calculation of the degrees of freedom df.

$$df = \frac{\left(\frac{1}{\nu|W|} + \frac{f\_ppf}{(1-\nu)|W|}\right)^2}{\frac{\left(\frac{1}{\nu|W|}\right)^2}{(\nu|W|)-1} + \frac{\left(\frac{f\_ppf}{(1-\nu)|W|}\right)^2}{((1-\nu)|W|)-1}} \tag{4.12}$$

Finally, by replacing t_value on Equation 4.9 by the one calculated on Equation 4.10, we reach Equation 4.1 introduced in Section 4.5.2. $\square$

We remark that Equation 4.1 only depends on δ', ρ, $|W|$ and ν. Recall that δ' is $\delta^{\frac{1}{4}}$ which is provided by the user; ρ is also given by the user; and $|W|$ increases linearly until w_{max}. We can then calculate the highest value ν based on each $|W|$ that solves Equation 4.1.

Therefore, considering the split $W = W_{hist}\, W_{new}$ with $W_{hist} = W_{0:\nu\,|W|}$ and $W_{new} = W_{\nu\,|W|:|W|-1}$, by applying the t-test, we guarantee that if μ_{hist} and μ_{new} diverge more than $\rho\,\sigma_{hist}$, we identify this divergence with a confidence of δ. Moreover, considering our OL setting, it takes no more than $(1-\nu)|W|$ iterations for W to be divided into $W_{hist}\, W_{new}$ after the drift occurred.

- **Part 3:** By setting $\nu = 0.5$ in Equation 4.9, we have:

$$\rho_{temp} = t\_value\sqrt{\frac{1}{0.5\,|W|} + \frac{f\_factor}{0.5\,|W|}} \tag{4.13}$$

  With $\nu = 0.5$ and ρ_{temp} as calculated above, we can guarantee that if μ_{hist} and μ_{new} diverge more than $\rho_{temp}\,\sigma_{hist}$, we identify the divergence with a confidence of δ (as guaranteed by the test procedure). Similarly to Part 2 of the proof, we achieve this in at most $(1-\nu)\,|W|$ iterations, in this case, $\frac{|W|}{2}$ iterations. $\square$

- **Part 4:** This bound is directly obtained from the f-test. By simply applying the f-test to compare two sub-windows of W, we guarantee the given confidence value, in this case δ. Again, it takes no more than $(1-\nu)\,|W|$ iterations for W to be divided into $W_{hist}\, W_{new}$ after the drift occurred. $\square$

4.8.2 Additional Plots

This section presents the remaining plots related to Tables 4.1, 4.2, and 4.3.

In Figure 4.5, we cannot observe the same effect of ρ in OPTWIN's delay detection that was observed in Figure 4.2. Nevertheless, we can still observe the high FP rate of EDDM, ECDD , and ADWIN compared to the other drift detectors. We note that STEPD's high FP rate comes from a few runs with dozens of FPs.

Figures 4.6 and 4.7 illustrate, respectively, the sudden and gradual drift detection experiments using non-binary data. In these experiments, we can clearly observe the excellent results obtained by OPTWIN in terms of drifts detected and the delay of those detections, independently of ρ. When analyzing ADWIN, we can see in Figure 4.6 how it correctly identifies the concept drifts but ends up producing FPs while its sliding window is not fully adapted to the new distribution. On the other hand, STEPD produced almost random guesses throughout both experiments, with multiple FPs.

Next, Figures 4.8 and 4.9 illustrate the results over the STAGGER and the Random RBF datasets, respectively. In Figure 4.8 we see that, besides STEPD, all drift detectors could successfully identify the concept drifts with few (or none) FPs. Differently, the Random RBF dataset was more challenging to the drift detectors. Consequently, in Figure 4.9 we can visualize that STEPD and ECDD once more produced almost random guesses on the location of the concept drifts, with multiple FPs. Besides those detectors, none of the others could successfully identify the third concept drift, which we assume to be due to an extremely subtle change in the data distribution. At last, we highlight that in the Random RBF experiment, the accuracy of the learner sometimes decreased even after the correct identification of the concept drift due to the new distribution being more challenging for the NB classifier to represent.

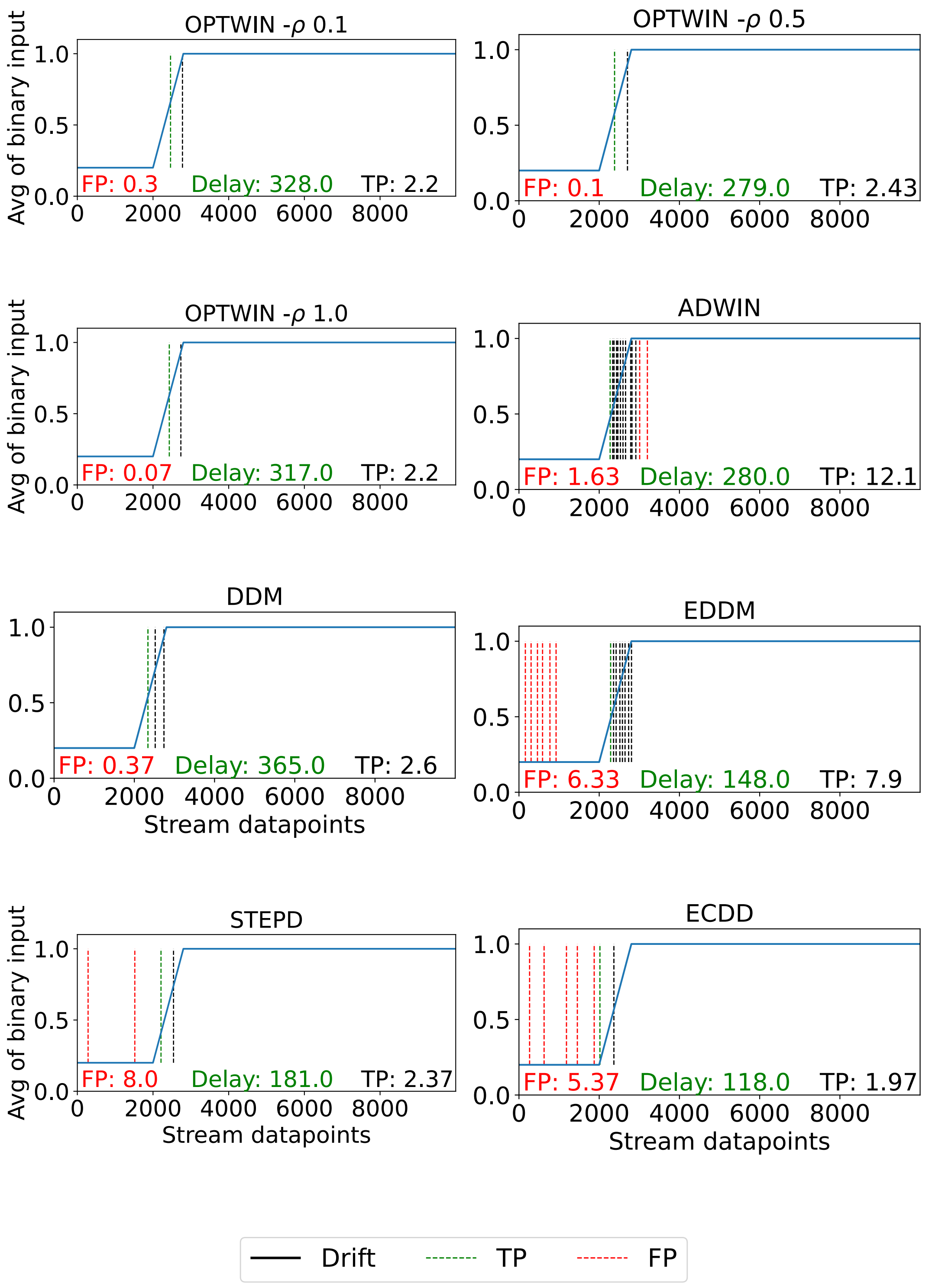


FIGURE 4.5: Gradual binary drift detection with average FP rates compared to drift-detection delays.

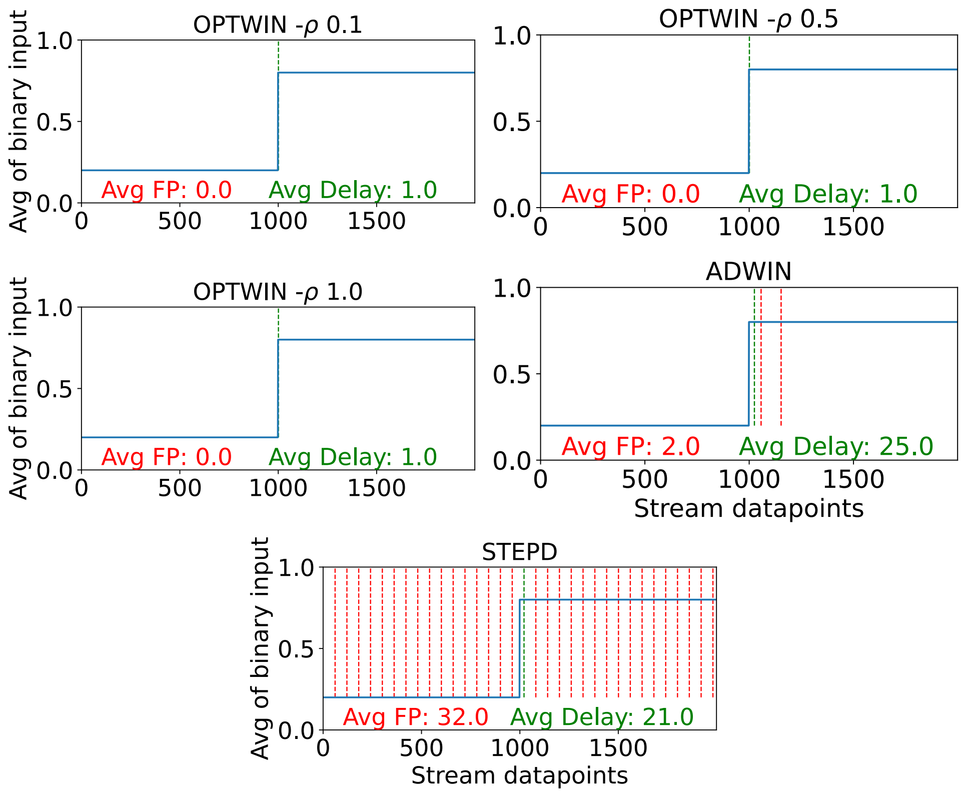


FIGURE 4.6: Sudden non-binary drift detection with average FP rates compared to drift-detection delays.

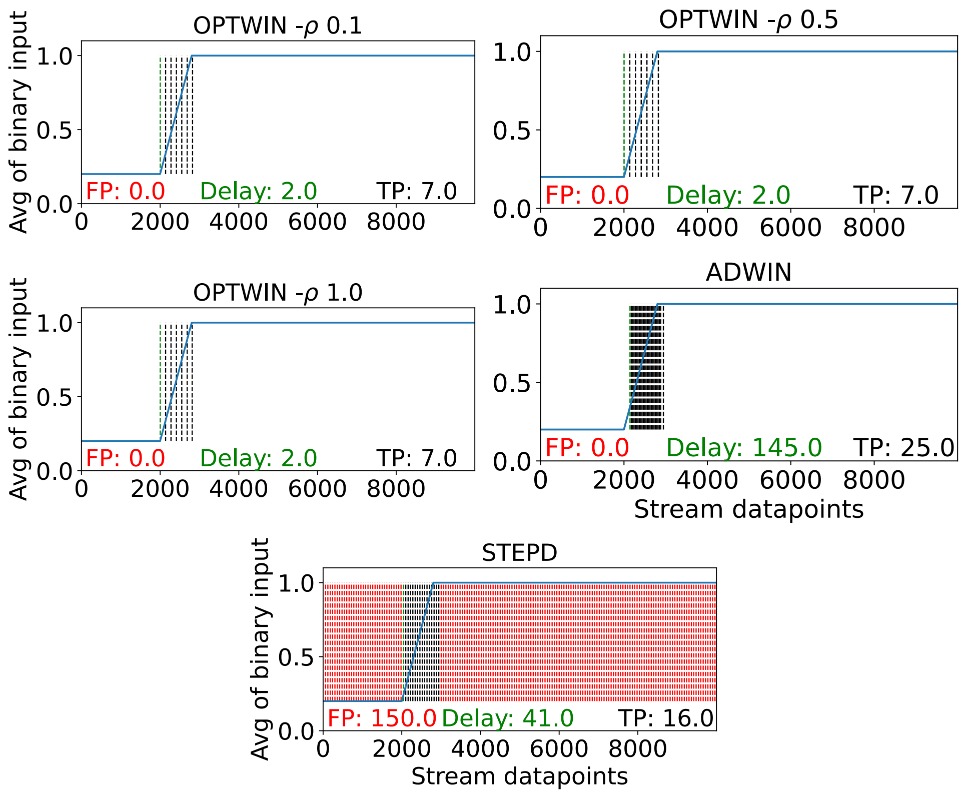


FIGURE 4.7: Gradual non-binary drift detection with average FP rates compared to drift-detection delays.

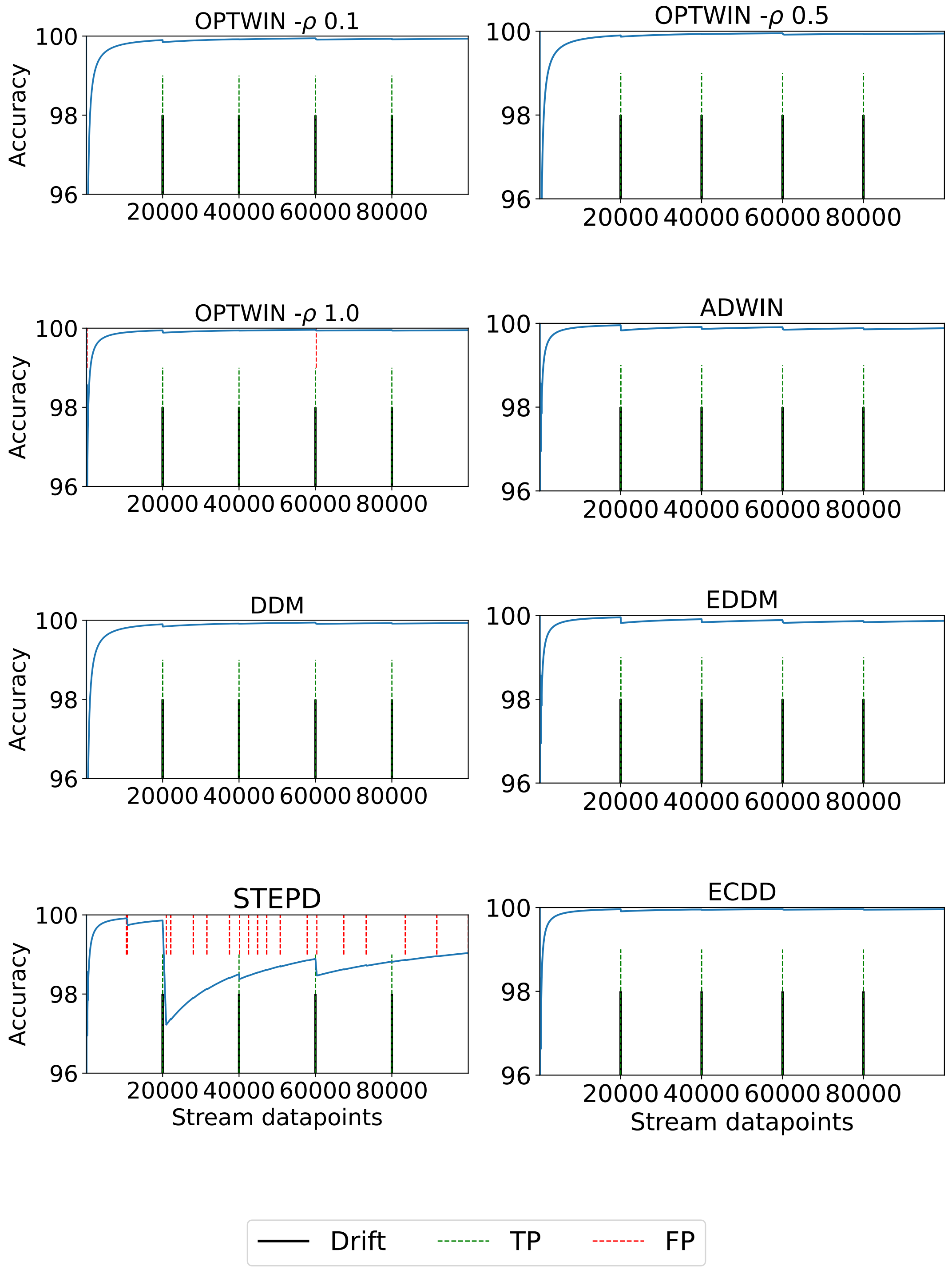


FIGURE 4.8: TP and FP rates for drift detection on the STAGGER dataset with sudden concept drifts.

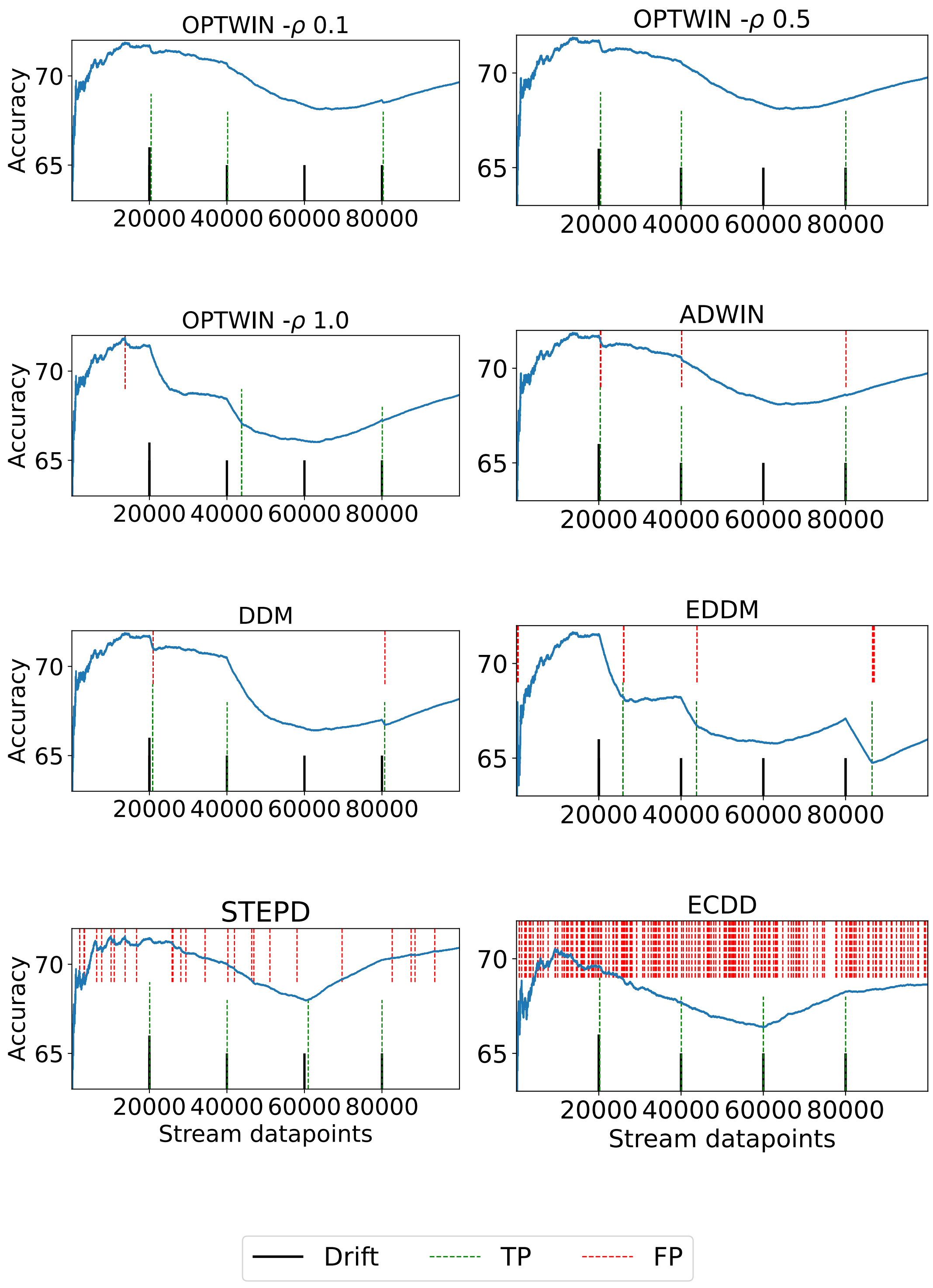


FIGURE 4.9: TP and FP rates for drift detection on the RandomRBF dataset with sudden concept drifts.

Chapter 5

Convergence Analysis of Decentralized ASGD

5.1 Contribution Statement

This chapter corresponds to the reprint of the paper "Convergence Analysis of Decentralized ASGD" (Tosi and Theobald, 2023) authored by Mauro Dalle Lucca Tosi and Martin Theobald. Below, we provide the CRediT (Contributor Roles Taxonomy) [1] statements correspondent to each author contribution.

- **Mauro Dalle Lucca Tosi:**
  - Conceptualization;
  - Methodology;
  - Software;
  - Validation;
  - Formal analysis;
  - Investigation;
  - Data Curation;
  - Writing - Original Draft;
  - Visualization;
- **Martin Theobald**
  - Formal analysis;
  - Resources;
  - Writing - Review & Editing;
  - Supervision;
  - Project administration;
  - Funding acquisition

5.2 Chapter Contextualization

This chapter corresponds to the paper "Convergence Analysis of Decentralized ASGD" (Tosi and Theobald, 2023). It presents our formal convergence rate analysis of the

[1] https://credit.niso.org/

decentralized and asynchronous SGD strategy adopted by TensAIR. The development of this convergence rate proof is essential for understanding the implications of asynchronous and decentralized distributions on the convergence rate of SGD, particularly in the context of ANN models, and can provide theoretical guarantees for the efficient and effective distributed training employed in TensAIR.

5.3 Introduction

Over the last decades, Stochastic Gradient Descent (SGD) (Robbins and Monro, 1951) has been intensively studied by the Machine Learning community. SGD and its many variants (such as mini-batch SGD (LeCun et al., 2002), ADAM (Kingma and Ba, 2014), AdaGrad (Duchi, Hazan, and Singer, 2011), etc.) have demonstrated their robustness by frequently achieving state-of-the-art results in diverse problems. In particular, Large Language Models (LLMs) such as GPT-4 (OpenAI, 2023), Generative Models such as Stable Diffusion (Rombach et al., 2021), and other Neural Network (NN) models such as (Chen et al., 2023; Schäfl et al., 2022; Zhang et al., 2020), involving billions of parameters, could not have been trained without the usage of SGD-based optimizers.

Despite its versatility and excellent performance, SGD demands a substantial amount of iterations to converge when solving complex problems over a large number of parameters. Consequently, training a large model with SGD may be a time-consuming task (Cheng et al., 2018). To mitigate this issue, it is common to distribute the computations performed by the SGD optimizer across multiple CPUs, GPUs, or even across multiple compute nodes. Various settings used to distribute the SGD optimizations are described in the literature (Ouyang et al., 2021; Mayer and Jacobsen, 2020); in this paper, we will focus on *data parallelism*, which is the most common form of distributed training for SGD-based optimizers (Mayer and Jacobsen, 2020).

When performing data-parallel training, multiple workers are initialized with the same parameters (i.e., weights), and they simultaneously calculate their models' gradients based on different data samples obtained from the training data (Ouyang et al., 2021). In a *distributed* (but synchronous) *SGD* setting, the gradients of all workers are aggregated periodically and used to update a centralized model, which is stored in a single parameter server and periodically broadcast to all workers (Mayer and Jacobsen, 2020). This setting however has two main disadvantages: (1) synchronization idle times, which occur whenever a worker finishes calculating its gradient faster than other workers and it spends resources waiting for delayed workers (Mayer and Jacobsen, 2020); and (2) bottlenecks in the parameter server, which may occur due to all workers communicating with the same centralized server simultaneously (Chen, Wang, and Li, 2019). To reduce the training time when using SGD-based optimizers, researchers actively studied how to overcome these issues arising from synchronization and centralization.

To avoid synchronization barriers among workers, current efforts are mostly focused on *asynchronous SGD* (ASGD). In an ASGD setting, workers do not wait for updates from the parameter server to resume computing new gradients (Ouyang et al., 2021). Therefore, they eliminate the workers' synchronization barriers and, consequently, idle times. However, as a consequence, the parameter server may receive also delayed gradients calculated during past iterations, which complicates the ASGD convergence proof. Recently, Koloskova, Stich, and Jaggi (Koloskova, Stich, and Jaggi, 2022) proved an improved ASGD convergence rate of $\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{\sqrt{\tau_{avg}\tau_{max}}}{\epsilon})$ to an ϵ-small error with τ representing the gradient delay; and $\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) +$

$\mathcal{O}(\frac{\tau_{avg}G}{\epsilon^{3/2}}) + \mathcal{O}(\frac{\tau_{avg}}{\epsilon})$ when the norm of the gradients is additionally bounded by G. These results are very broadly applicable because, based on them, the authors could also prove that ASGD always converges faster than mini-batch SGD.

To avoid these bottlenecks arising from centralized parameter servers, one may choose a decentralized setting, in which it is not necessary to aggregate gradients within a single parameter server. Instead, worker nodes exchange gradients directly among themselves, aggregate them locally, and update themselves accordingly (Mayer and Jacobsen, 2020). The exchange of gradients among workers may be performed based on different network topologies. Common such topologies are the *fully connected topology*, in which all workers are pairwisely connected; and the *ring topology*, in which the workers are connected in the form of a closed loop or ring, with each worker communicating to two adjacent workers (Mayer and Jacobsen, 2020).

Algorithms that consider ASGD in a decentralized topology are still rare due to the complexity of proving its convergence under both of these relaxations simultaneously. Most authors have studied the convergence rate of decentralized ASGD for convex functions (Wu et al., 2017; Assran and Rabbat, 2020; Jiang et al., 2021), while a few others proved convergence rates also for non-convex ones (Lian et al., 2018; Assran et al., 2019). Nevertheless, most of those proofs still rely on partial synchronization during models communication and specific network topologies.

5.3.1 Contributions

- We formally prove a convergence rate for DASGD of $\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{QS_{avg}}{\epsilon^{3/2}}) + \mathcal{O}(\frac{S_{avg}}{\epsilon})$ to an ϵ-small error, with Q bounding the norm of the gradients, and S_{avg} representing the *average staleness* among all models. Here, staleness is defined as the pairwise symmetric difference between the sets of gradients calculated by one model and applied by another model (cf. Section 5.6). This convergence rate is guaranteed for any non-convex, L-smooth, and homogeneous objective function with bounded gradients, and for a constant stepsize of $\eta \leq (4LS_{avg})^{-1}$.
- With no assumption over the size of the gradients, we additionally prove a convergence rate for DASGD of $\mathcal{O}(\frac{\sigma}{\epsilon^2}) + \mathcal{O}(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{\epsilon})$ to an ϵ-small error, using a constant stepsize of $\eta \leq (4L\sqrt{\hat{S}_{avg}\hat{S}_{max}})^{-1}$. Here, $\hat{S}_{avg}$ and $\hat{S}_{max}$ represent a generalized version of the *average* and *maximum staleness* among the distributed models, respectively.
- Finally, we also empirically demonstrate how staleness impacts the convergence rate of a logistic regression, a quadratic function, and a Convolutional Neural Network (CNN) model when optimized with DASGD.

5.3.2 Limitations

- Our proofs depend on a fixed stepsize (aka. "learning rate") that is defined based on the models' staleness, which may not be known before the training.
- Our proof does not currently contemplate heterogeneous functions, commonly observed in federated learning problems. Nevertheless, the effect of the heterogeneity should be orthogonal to the effect of our staleness measure.

5.4 Related Works

Decentralized SGD. The idea that decentralized SGD (DSGD) can outperform centralized SGD has been shown in (Lian et al., 2017). However, depending on the topology of the underlying network, the communication cost when not relying on a centralized parameter server can increase quadratically with the number of distributed worker nodes n (Mayer and Jacobsen, 2020). One approach to lower the communication cost is to use the so-called "gossip algorithms" (Lian et al., 2017; Koloskova, Stich, and Jaggi, 2019; Koloskova et al., 2020), which allow workers to exchange and aggregate their gradients only with their immediate neighbors (usually 2 to 3), instead of all worker nodes. Then, those neighbors disseminate the messages received across the network iteratively, and after approximately 1.639 $\log_2(n)$ communication steps all workers will receive the initial message (Ben-Nun and Hoefler, 2019). However, networks with more than 32 workers may still suffer in terms of convergence and performance due to this relatively high delay in the exchange of messages (Ben-Nun and Hoefler, 2019).

Asynchronous SGD. Introduced already decades ago, the study of asynchronous SGD (ASGD) has gained an increasing amount of attention again recently. At first, works such as Hogwild! (Recht et al., 2011) focused on proving the convergence of ASGD under the assumption of the sparseness of the optimized models. Recently, less restricted convergence proves were developed, which also contemplate dense models as in (Koloskova, Stich, and Jaggi, 2022; Mishchenko et al., 2022; Cohen et al., 2021; Arjevani, Shamir, and Srebro, 2020; Xie, Koyejo, and Gupta, 2019; Nguyen et al., 2022; Toghani and Uribe, 2022; Stich and Karimireddy, 2020; Stich, 2019). A main aspect to facilitate the proof of convergence used by (Mishchenko et al., 2022) is to scale the gradients based on their delay, such that delayed gradients have less impact when updating the model. Another technique seen in (Cohen et al., 2021) is to simply discard gradients which are too delayed. Both techniques described above are easy to implement in a centralized setting, in which a parameter server can adapt its learning rate based on the delay of the gradient that it applies. However, this adaptation is not straightforward in a decentralized setting, in which there is no server to decide how delayed the gradients are. Furthermore, part of these ASGD convergence proofs do not cover the optimization of non-convex functions (Arjevani, Shamir, and Srebro, 2020; Xie, Koyejo, and Gupta, 2019; Stich, 2019). Thus, only very few convergence proofs provided in the literature (Koloskova, Stich, and Jaggi, 2022; Nguyen et al., 2022; Toghani and Uribe, 2022; Stich and Karimireddy, 2020) cover ASGD on non-convex objective functions while using learning rates which are invariant to the gradients' delays. We highlight here the recent proof by Koloskova, Stich and Jaggi (Koloskova, Stich, and Jaggi, 2022) which provides the currently best convergence rate for ASGD under mild assumptions, while considering a fixed learning rate and non-convex functions. Specifically, they were able to prove that ASGD convergence does not depend on the maximum delay of gradients when those gradients are bounded.

Decentralized and Asynchronous SGD. To mitigate the bottleneck of using a centralized parameter server and to avoid idle times from synchronizing distributed models, some authors focused their research on decentralized and asynchronous SGD (Wu et al., 2023; Ram, Nedić, and Veeravalli, 2010; Sirb and Ye, 2016; Assran and Rabbat, 2020; Wu et al., 2017; Even, Hendrikx, and Massoulié, 2021; Srivastava and Nedic, 2011; Ram, Nedić, and Veeravalli, 2009; Lian et al., 2018; Assran et al., 2019). However, most of those studies make strong assumptions as in(Wu et al., 2023; Ram, Nedić, and Veeravalli, 2010; Sirb and Ye, 2016; Wu et al., 2017; Assran and Rabbat, 2020; Even, Hendrikx, and Massoulié, 2021; Srivastava and Nedic, 2011), which either do not prove convergence for non-convex functions, or they prove the convergence of their models only when the number of iterations goes to infinity (Wu et al., 2023; Srivastava and Nedic, 2011; Ram, Nedić, and Veeravalli, 2009). Contrarily, (Lian et al., 2018) proved the convergence rate of decentralized and asynchronous SGD for non-convex functions by relying on specific network topologies and atomic communication steps among nodes, which requires a partial synchronization between any two nodes during the optimization process, thus hindering the convergence time in practice (Jiang et al., 2021; Luo et al., 2020; Miao et al., 2021). Differently, (Assran et al., 2019) does not assume atomic communication steps but blocks its workers to ensure that a maximum delay τ is maintained during the optimization process.

Recently, (Bornstein et al., 2023) reached a similar convergence rate to the one we present in this paper by also using a wait-free approach that eliminates the partial synchronization and special network architecture requirements. Their optimization process relies on a novel client-communication matrix that governs the models averaging. As opposed to this approach, we freely exchange gradients among models and introduce the new *staleness* metric to measure the difference between those models based on the sets of gradients they exchanged. At last, both our approach and the one presented in (Bornstein et al., 2023) were able to eliminate the dependency on the slowest model (the "maximum delay") in the network from their convergence rate proofs. We eliminate this dependency by assuming that gradients are bounded, while they eliminate this dependency by defining a minimum amount of iterations until convergence and choosing their learning rate accordingly.

A detailed comparison of all papers discussed above under these aspects can be found in Table 5.1.

5.5 Optimization Objective & DASGD Algorithm

In this section, we present the setup under which we prove an upper bound of the convergence rate of *decentralized and asynchronous SGD* (DASGD). First, we formally describe the optimization problem we solve and the assumptions we make. Then, we present the characteristics expected from the network topology and communication strategy on which our convergence analysis is based. Finally, we present the algorithm which captures the protocol that is followed by each of our decentralized worker nodes.

Our setting is based on the one presented in (Koloskova, Stich, and Jaggi, 2022), which provides tight convergence rates for ASGD under various centralized settings. We in particular generalize their analysis of ASGD under a homogeneous setting with a fixed stepsize (their Theorem 6), and extend this to a decentralized setting which no longer relies on a centralized parameter server. The main intuition behind our approach is that, if we can assume that any gradient computed by any of the local models also eventually reaches all the other models, then each of the local models effectively works like a parameters server itself, and therefore all local models will converge to the same global model. This allows us to develop analogous constructions to the proofs provided in (Koloskova, Stich, and Jaggi, 2022) and (Nguyen et al., 2022) for our decentralized ASGD setting.

Reference	**Async.**	**Decentr.**	**Non-convex**	**Convergence rate**
(Lian et al., 2017)	No	Yes	Yes	Yes
(Koloskova et al., 2020)	No	Yes	Yes	Yes
(Koloskova, Stich, and Jaggi, 2019)	No	Yes	Yes	Yes
(Cohen et al., 2021)	No	Yes	Yes	Yes
(Koloskova, Stich, and Jaggi, 2022)	Yes	No	Yes	Yes
(Nguyen et al., 2022)	Yes	No	Yes	Yes
(Stich and Karimireddy, 2020)	Yes	No	Yes	Yes
(Mishchenko et al., 2022)	Yes	No	Yes	Yes
(Toghani and Uribe, 2022)	Yes	No	Yes	Yes
(Recht et al., 2011)	Yes	No	No	Yes
(Xie, Koyejo, and Gupta, 2019)	Yes	No	No	Yes
(Stich, 2019)	Yes	No	No	Yes
(Arjevani, Shamir, and Srebro, 2020)	Yes	No	No	Yes
(Wu et al., 2023)	Yes	Yes	No	Infinity
(Srivastava and Nedic, 2011)	Yes	Yes	No	Infinity
(Ram, Nedić, and Veeravalli, 2010)	Yes	Yes	No	Yes
(Sirb and Ye, 2016)	Yes	Yes	No	Yes
(Wu et al., 2017)	Yes	Yes	No	Yes
(Assran and Rabbat, 2020)	Yes	Yes	No	Yes
(Even, Hendrikx, and Massoulié, 2021)	Yes	Yes	No	Yes
(Ram, Nedić, and Veeravalli, 2009)	Yes	Yes	Yes	Infinity
(Lian et al., 2018)	**Yes**	**Yes**	**Yes**	**Yes**
(Assran et al., 2019)	**Yes**	**Yes**	**Yes**	**Yes**
(Bornstein et al., 2023)	**Yes**	**Yes**	**Yes**	**Yes**
Theorem 5.6.1	**Yes**	**Yes**	**Yes**	**Yes**

TABLE 5.1: Comparison between SGD convergence proofs.

5.5.1 Optimization Objective

Following (Koloskova, Stich, and Jaggi, 2022), we consider the *optimization objective* shown below:

$$\min_{x\in\mathbb{R}^d}\left[f(x) := \frac{1}{n}\sum_{i=1}^{n}\left[f_i(x) = \mathbb{E}_{\xi\sim\mathcal{D}} F_i(x,\xi)\right]\right] \tag{5.1}$$

Here, F_i represents a local loss function which is accessed by node i with parameters (i.e., weights) x on data samples $\xi \sim \mathcal{D}$, considering $i \in \{1,\dots,n\}$ (in the following abbreviated as $i \in [n]$). As in (Koloskova, Stich, and Jaggi, 2022), each $f_i(x)$ with $f_i : \mathbb{R}^d \to \mathbb{R}$ is assumed to be a stochastic function, i.e., $f_i(x) = \mathbb{E}_{\xi\sim\mathcal{D}_i} F_i(x,\xi)$, and is accessed only via its local gradients $\nabla F_i(x,\xi)$.

We remark that this is a very generic setting, which captures any data distribution $\mathcal{D}$ and applies to any (smooth, but possibly non-convex) objective function. In case the optimization problem, for example, is deterministic, we may set $f_i(x) = F_i(x,\xi), \forall\xi$; if, on the other hand, $\mathcal{D}$ is a uniform distribution with local samples $\{\xi_i^1,\dots,\xi_i^{m_i}\}$, we have $f_i(x) = \frac{1}{m_i}\sum_{t=1}^{m_i} F_i(x,\xi_i^t)$. However, as opposed to (Koloskova, Stich, and Jaggi, 2022), we assume all samples ξ to come from the same global distribution $\mathcal{D}$ rather than allowing different local distributions $\mathcal{D}_i$ (which, together with fixed stepsizes, matches the setting of Theorem 6 in (Koloskova, Stich, and Jaggi, 2022)). Consequently, we also only look at homogeneous functions $f_i(x)$, as stated in Assumption 2 in Section 5.5.1.2.

5.5.1.1 Notation

Below we summarize the notations we use throughout the paper.

- η, fixed learning rate;
- n, number of distributed worker nodes and models;
- x^0, initial model (distributed across all worker nodes);
- x_i^t, local model i at iteration t, with $x_i^0 = x^0$ and $i \in [n]$;
- x^t, any model $x_i^t, \forall i \in [n]$;
- g_i^t, gradient calculated using model x_i^t and sample ξ_i^t; thus, $g_i^t = \nabla F(x_i^t, \xi_i^t)$;
- G_i^t, set of gradients applied from x^0 to x_i^t; thus, $x_i^t = x_i^0 - \eta \sum_{g \in G_i^t} g$;
- $S_{i,j}^{t,s}$, staleness between set of gradients G_i^t and G_j^s (cf. Definition 1);
- $\hat{S}_{i,j}^{t,s}$, generalized version of staleness between G_i^t and G_j^s (cf. Definition 1);
- $\delta_{i,j}^{t,s}$, sum of gradients in $S_{i,j}^{t,s}$ weighted by η; thus, $\delta_{i,j}^{t,s} = \eta \sum_{g \in S_{i,j}^{t,s}} g$, with $\delta_{i,j}^{t,s} \in \mathbb{R}^d$;
- $\Delta_{i,j}^{t,s}$, upper bound of $\delta_{i,j}^{t,s}$; $\Delta_{i,j}^{t,s} = \eta \sum_{g \in S_{i,j}^{t,s}} |g|$, with $\Delta_{i,j}^{t,s} \in \mathbb{R}^d$.

Throughout this paper, we refer to L$_2$-norm $\| \cdot \|_2$ as our default norm for vectors and thus simplify our notation by writing $\| \cdot \|$. Moreover, with $|v|$, for a vector $v \in \mathbb{R}^d$, we denote a corresponding vector consisting of the absolute values along v's dimensions, i.e., $|v| = \langle |v_1|, |v_2|, \ldots, |v_d| \rangle$.

5.5.1.2 Assumptions

The following assumptions are considered throughout the description of our setting and the convergence analysis provided in the Supplementary Material (Section 5.9.2).

Assumption 1 (Bounded variance)**.** There exists a constant σ, such that:

$$\mathbb{E}_{\xi \sim \mathcal{D}} \|\nabla F_i(x, \xi) - \nabla f_i(x)\|^2 \leq \sigma^2 \quad \forall i \in [n], \forall x \in \mathbb{R}^d \tag{5.2}$$

Assumption 2 (Function homogeneity)**.** The functions f_i are homogeneous, thus:

$$\|\nabla f_i(x) - \nabla f_j(x)\| = 0 \quad \forall i, j \in [n], \forall x \in \mathbb{R}^d \tag{5.3}$$

Assumption 3 (Lipschitz gradient)**.** The gradient is L-smooth and there exists a constant $L \geq 1$, such that:

$$\|\nabla f_i(y) - \nabla f_i(x)\| \leq L\|x - y\| \quad \forall i \in [n], \forall x, y \in \mathbb{R}^d \tag{5.4}$$

Assumption 4 (Bounded gradient)**.** There exists a constant $Q \geq 0$, such that:

$$\|\nabla f_i(x)\|^2 \leq Q^2 \quad \forall i \in [n], \forall x \in \mathbb{R}^d \tag{5.5}$$

The above assumptions are very common in the context of SGD and have been adopted by various classical works. Specifically, Assumptions 1 and 3 are commonly used for most SGD proofs. The only restriction we make when compared to other papers is to consider $L \geq 1$ (instead of the commonly used $L \geq 0$), which allows us to perform simplifications during our proof that guarantee the provided convergence rate without further theoretical or practical implications. For Assumption 4, we adopt the same strategy as (Koloskova, Stich, and Jaggi, 2022) by providing two different bounds, one considering this assumption and one not considering it. At last, considering Assumption 2 (also explored by (Arjevani, Shamir, and Srebro, 2020; Stich and Karimireddy, 2020; Agarwal and Duchi, 2011; Feyzmahdavian, Aytekin, and Johansson, 2016; Lian et al., 2015; Sra et al., 2016)), we focus our analysis on distributed gradient computations as they are typically performed in cloud or HPC-based settings (Dean et al., 2012), and in which all worker nodes (using CPUs or GPUs) process data from the same distribution $\mathcal{D}$ (thus guaranteeing Assumption 2).

5.5.2 Network Topology & Communication Protocol

Our convergence proof for DASGD is flexible enough to allow any network topology or communication protocol between the worker nodes (e.g., fully connected, ring, and mesh topologies (Mayer and Jacobsen, 2020)) as long as the following characteristics are respected.

- *connected graph*: there must exist a communication path between any two nodes in the network;
- *no lost messages*: a message (encoding gradients) that is sent by a worker node shall eventually be received by all other nodes in the network;
- *no repeated messages*: each message will be received at most once by each worker node.

We remark that neither our DASGD algorithm (cf. Section 5.5.3) nor our convergence analysis (cf. Section 5.6) depend on the order of the sent messages to be preserved among the worker nodes. We see this as a strong feature of our approach which enhances the flexibility of the communication protocol that may be adopted. In practice, the longer it takes for two nodes to exchange their local gradients, the larger the staleness among their models will become and the worse the bounds for the convergence rate will be (cf. Section 5.6). Moreover, as our messages are timestamped (using local step counters t only), our staleness measure immediately complies also with unordered messages, since we can exactly determine the amount of lagging messages from these step counters. However, in practice, one will observe better convergence rates when using denser network topologies with a frequent and ordered communication between the worker nodes.

5.5.3 Decentralized & Asynchronous SGD Algorithm

We next introduce our *DASGD algorithm*, as summarized in Algorithm 3. From a practical perspective, the idea behind it is to calculate new gradients only if no updates (i.e., gradients received from other models) are available. In doing so, we reduce the dissimilarity between the asynchronous models throughout the training. Specifically, we eliminate idle times coming from synchronization barriers between nodes (common on synchronous SGD) and prevent bottlenecks on parameter servers (common on centralized SGD).

Algorithm 3 DASGD Algortihm.

```
1:  Constructor DASGD ($x^0$, $i$)
2:      $t = 0$
3:      $x_i^t = x^0$
4:  procedure TRAINMODEL
5:      while true do
6:          $g =$ receiveGradient()          ▷ NULL if there is no incoming gradient
7:          if $g \neq$ NULL then
8:              $x_i^{t+1} = x_i^t - \eta g$
9:              $t = t + 1$
10:         else if $\exists\, \xi_i^t$ then                    ▷ compute a new gradient
11:             $g_i^t = \nabla F(x_i^t, \xi_i^t)$
12:             $x_i^{t+1} = x_i^t - \eta g_i^t$
13:             sendGradient($g_i^t$)                          ▷ non-blocking
14:             $t = t + 1$
```

In Algorithm 3, we show the protocol performed by all $i \in [n]$ worker nodes. First, all nodes are initialized with the same model parameters x^0, their node id i, and a local iteration counter $t = 0$.

During training, on Lines 6 and 7, node i checks if any gradient g computed by another node is available. If so, it immediately (i.e., asynchronously) updates itself based on the received gradient and by using a fixed learning rate η (Line 8). It then also increments its iteration counter t by 1 (Line 9) and resumes its training back on Line 5.

If, on the other hand, no gradients were received, worker i checks if a new training sample ξ_i^t is available (Line 10). It then calculates a new gradient g_i^t based on its current model's weights x_i^t and iteration counter t (Line 11). After calculating g_i^t, worker i updates itself based on this gradient and the learning rate η (Line 12). It then sends (i.e., in the simplest case "broadcasts") this gradient to the other workers it is connected to in the network (Line 13). At last, it increments its iteration counter t by 1 (Line 14); and it resumes its training back on Line 5.

The training is performed until $\nexists\, \xi_i^t, \forall i \in [n]$ and all gradients g_i^t were applied by all nodes $i, \forall i \in [n]$, which assumes only a single (and final) synchronization point among all worker nodes.

5.6 Convergence Rate Analysis

Throughout the execution of Algorithm 3, a local model x_i will update itself using gradients calculated by a model x_j, with $i, j \in [n]$. As opposed to current works, considering our asynchronous and decentralized setting, we cannot guarantee that x_i and x_j represent the same model at different iteration points. Thus, neither $x_i^t = x_j^{t-\tau}$ nor $x_j^t = x_i^{t-\tau}, \forall \tau \in \mathbb{N}$ are assured, considering t as the model's iteration and τ a possible delay. Strictly speaking, we cannot even assume that the iteration counters t at each model are synchronized. Therefore, we rely on a different approach to calculate the difference between models. First, we represent a model x_i^t via a set of gradients G_i^t which it either received or computed itself at iteration t.

Remark 1. *The set of gradients G_i^t uniquely represents the evolution of model x^0 into x_i^t when using a fixed learning rate η. Thus, we can describe x_i^t as:*

$$x_i^t = x^0 - \eta \sum_{g \in G_i^t} g \tag{5.6}$$

Therefore, we can quantify the dissimilarity between models based on the set of gradients which they applied to themselves since they were initialized, provided that these were initialized with the same parameters x^0 and updated with the same learning rate η. More specifically, we calculate the symmetric difference between these two sets, which produces the *staleness* of gradients (Tosi, Venugopal, and Theobald, 2022), represented here as $S_{i,j}^{t,s}$, between pairs of models i, j and iterations t, s (cf. Figure 5.1). Our convergence bounds depend on the *average* and *maximum* sizes of these staleness sets, represented as S_{avg} and S_{max}, respectively.

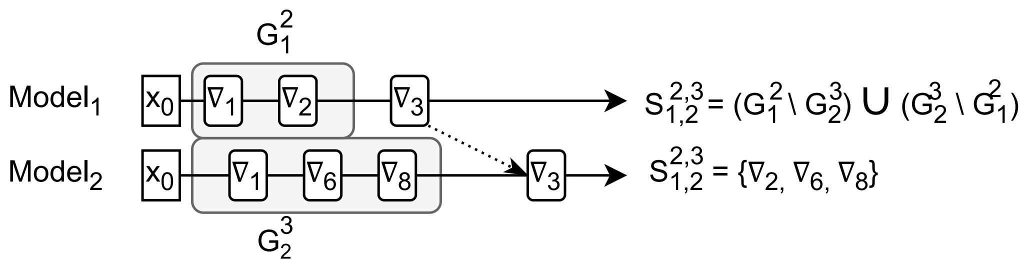

G_1^2
Model$_1$
x_0
∇_1
∇_2
∇_3
$S_{1,2}^{2,3} = (G_1^2 \setminus G_2^3) \cup (G_2^3 \setminus G_1^2)$
Model$_2$
x_0
∇_1
∇_6
∇_8
∇_3
$S_{1,2}^{2,3} = \{\nabla_2, \nabla_6, \nabla_8\}$
G_2^3

FIGURE 5.1: *Staleness* $S_{1,2}^{2,3}$ at the time when Model$_2$ applies gradient ∇_3 calculated by Model$_1$. The grey areas determine the sets of gradients G_1^2 and G_2^3 used to calculate the symmetric difference.

For the following definition, let x_j^s represent model j that calculated a gradient g_j^s at iteration s, and let x_i^t represent the model i that applies g_j^s at iteration $t+1$. We then represent x_j^s and x_i^t using G_j^s and G_i^t respectively, assuming that both models were initialized with the same parameters x^0 and updated at the same constant learning rate η.

Definition 1 (Staleness). *Under Assumption 4 (bounded gradients), we define* staleness *$S_{i,j}^{t,s}$ as the symmetric difference between the sets of gradients G_i^t and G_j^s applied by models i and j at steps t and s, respectively:*

$$S_{i,j}^{t,s} = (G_i^t \setminus G_j^s) \cup (G_j^s \setminus G_i^t) \\ (\textit{when Assumption 4 holds}) \tag{5.7}$$

Moreover, when Assumption 4 (bounded gradients) does not hold, we also define a generalized (i.e., "looser") version of staleness $\hat{S}_{i,j}^{t,s}$. This version is defined as the union of (1) the symmetric difference between the sets of gradients G_i^t and G_j^s (as before); and (2) the staleness $\hat{S}_{i,k}^{t,u}$, which recursively also includes gradients G_k^u calculated by another model k at step u (and which belong to G_j^s but not to G_i^t):

$$\hat{S}_{i,j}^{t,s} = (G_i^t \setminus G_j^s) \cup (G_j^s \setminus G_i^t) \cup_{g_k^u \in (G_j^s \setminus G_i^t)} \hat{S}_{i,k}^{t,u} \\ (\textit{when Assumption 4 does not hold}) \tag{5.8}$$

We highlight here that, when calculating the staleness between models x_i^t and x_j^s (using G_i^t and G_j^s), each model requires only a local iteration counter, i.e., $0 \leq t \leq T_i$ and $0 \leq s \leq T_j$ (cf. Algorithm 3). To simplify further notations, we thus refer to T as the maximum step counter T_i of the model that is currently under consideration. When analyzing the staleness $S_{i,j}^{t,s}$ between pairs of models i, j, then by convention T will represent the step counter of the left-hand model (the one of model i throughout the rest of the paper).

Definition 2 (Average & Maximum Staleness). *Let g_j^s be the gradient applied to model i at step $t+1$. Then, $|S_{i,j}^{t,s}|$ and $|\hat{S}_{i,j}^{t,s}|$ denote the sizes of the* staleness *sets between model i at step t and model j at step s. Thus, we define the* maximum *and* average staleness *among all models in the same manner for S and $\hat{S}$:*

$$\begin{aligned} S_{max} &:= \max_{0<i\leq n, 0\leq t\leq T} \{|S_{i,j}^{t,s}|\} \\ S_{avg} &:= \max_{0<i\leq n} \{\frac{1}{(T+1)} \sum_{t=0}^{T} |S_{i,j}^{t,s}|\} \end{aligned} \tag{5.9}$$

$$\begin{aligned} \hat{S}_{max} &:= \max_{0<i\leq n, 0\leq t\leq T} \{|\hat{S}_{i,j}^{t,s}|\} \\ \hat{S}_{avg} &:= \max_{0<i\leq n} \{\frac{1}{(T+1)} \sum_{t=0}^{T} |\hat{S}_{i,j}^{t,s}|\} \end{aligned} \tag{5.10}$$

The key idea behind our DASGD approach comes from the observation that gradient applications with a fixed learning rate are both *associative* and *commutative*. Therefore, we can guarantee convergence among multiple decentralized models as long as (1) the models are initialized with the same weights; (2) the models apply the same gradients (independently of their order); (3) and the same gradients are applied with the same learning rate. Moreover, by considering our definition of staleness, we can estimate the dissimilarity between models and thereby determine the expected convergence rate of the global model.

Below, we present the central results of our convergence analysis.

Theorem 5.6.1. *Considering Assumptions 1, 2, 3, 4, and a constant stepsize $\eta \leq \frac{1}{4LS_{avg}}$, Algorithm 3 reaches $\frac{1}{T+1}\sum_{t=0}^{T}(\|\nabla f(x^t)\|^2) \leq \epsilon$ after*

$$\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{QS_{avg}}{\epsilon^{\frac{3}{2}}}) + \mathcal{O}(\frac{S_{avg}}{\epsilon}) \quad \textit{iterations.} \tag{5.11}$$

Moreover, without Assumption 4 and using $\eta \leq \frac{1}{4L\sqrt{\hat{S}_{avg}\hat{S}_{max}}}$, Algorithm 3 reaches $\frac{1}{T+1}\sum_{t=0}^{T}(\|\nabla f(x^t)\|^2) \leq \epsilon$ after

$$\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{\epsilon}) \quad \textit{iterations.} \tag{5.12}$$

Our detailed proof of Theorem 5.6.1 is available in the Supplementary Material (Section 5.9.2) which accompanies this submission.

5.6.1 Discussion

DASGD vs. SGD. As seen in Theorem 5.6.1, the importance of the staleness decreases over time. Therefore, we can conclude that if $S_{max} \leq \sqrt{T}$ or $\hat{S}_{max} \leq \sqrt{T}$, DASGD converges at a similar rate as synchronous SGD. Nevertheless, DASGD eliminates idle time coming from slower nodes, which makes it calculate gradients at a higher pace and, consequently, converging faster than synchronous approaches.

Network topology. The convergence of DASGD is directly impacted by S_{avg} or by $\hat{S}_{avg}$ and $\hat{S}_{max}$ (when Assumption 4 does not hold). These depend on three factors: (1) the network topology; (2) the communication latency; and (3) the computational resources available among nodes. Assuming a scenario with computational resources being equally distributed and the latency being smaller than the computation time it takes to calculate gradients, we can directly measure the impact of the network topology chosen. For example, under the above-mentioned conditions, in a fully connected topology with n nodes, one can expect $S_{avg} = \frac{n+1}{2}$ and $S_{max} = n$. Differently, in a ring topology, messages will take n times longer to reach their destinations, with $S_{avg} = \frac{n^2+1}{2}$ and $S_{max} = n^2$.

Tightness. When compared to a centralized setting, the sizes of our *staleness* sets behave analogously to the *delay* used in works such as (Koloskova, Stich, and Jaggi, 2022; Nguyen et al., 2022). Therefore, we can also interpret the delay as a special case of our staleness, the latter representing the size of the symmetric difference of gradient sets obtained from the same model (i.e., the one at the parameter server) across different iterations. Considering this, we can view the convergence proof provided in this paper as a generalization of Theorem 6 presented in (Koloskova, Stich, and Jaggi, 2022), which in turn also extends traditional mini-batch SGD with constant stepsizes. This indicates the tightness of the given convergence rate, coinciding with the known lower bound for mini-batch SGD of $\Theta(\frac{\sigma^2}{n\epsilon^2} + \frac{1}{\epsilon})$ batches which are necessary to reach ϵ as stationary point (when setting $S_{max} = n$ and $S_{avg} = \frac{n}{2}$ in Equation (5.12)).

Communication complexity. To guarantee a smaller staleness during the optimization, each model from Algorithm 3 broadcasts the gradients it calculated to all other models in the network, resulting in a communication complexity of $\mathcal{O}(n^2)$. To reduce the communication complexity, we propose to locally aggregate gradients and broadcast them every n iteration as seen in Algorithm 4 in the Supplementary Material (Section 5.9.3). This relaxation reduces the communication complexity to $\mathcal{O}(n)$ while increasing the staleness during the optimization. We note that Theorem 5.6.1 and its proofs still hold when using Algorithm 4.

Practical limitations. Despite the simplicity of Algorithm 3, there still are practical limitations when choosing the learning rate η. First, η shall be fixed throughout the training, which is known not to be optimal in practice. Second, η shall be bounded by S_{avg} or by $\hat{S}_{max}$ and $\hat{S}_{avg}$, which may not be known beforehand. Nevertheless, in practice, one can estimate those values and use varying step sizes to reach convergence under the provided bounds without further concerns, as seen in our experiments in Section 5.7 and in the Supplementary Material (Section 5.9.4).

5.7 Experiments

In this section, we show how the theoretical bounds for DASGD introduced in Theorem 5.6.1 traverse to practical experiments. All experiments described below are available on our Gitlab repository[2]. The experiments were run on a DELL PowerEdge R840 server with 192 cores using an MPI environment which simulates multiple ranks with no shared memory.

Inspired by (Koloskova, Stich, and Jaggi, 2022), we assessed DASGD by performing our first two experiments in a scenario with no stochastic noise, i.e., $\sigma = 0$. In doing so, the convergence rate of the model being optimized is reduced to $\mathcal{O}(\sqrt{\hat{S}_{avg}\hat{S}_{max}}\ \epsilon^{-1})$, thus depending only on the *staleness* observed during the optimization. In both experiments, we fixed $n = 2$, $\epsilon = 1e^{-12}$, $\eta = 0.002$ and varied $\hat{S}_{max}$ from 0 to 100, which in particular maintains $\eta < (4L\sqrt{\hat{S}_{avg}\hat{S}_{max}})^{-1}$.

To artificially control the staleness during the optimization, we followed two strategies: (1) we reduced the gradient calculation speed of one of the models, making it x times slower; (2) we artificially increased the time it took to calculate each gradient by 0.05 seconds. The first technique guarantees that one of the workers outperforms the other one, thereby increasing the *maximum staleness*. The second technique guarantees that the gradient calculation takes orders of times more than the actual gradient application, which gives the slower model the chance to calculate new gradients instead of being overloaded by the application of the gradients calculated by the fastest model (which is the case in most real-world use-cases). Thus, with both techniques described above and the other parameters fixed, we can expect $\hat{S}_{max} = x$ during the optimization. Furthermore, we estimated the error of the optimized functions based on the average of the L$_2$-norm of their gradients over the last 30 iterations. We chose to optimize the same functions as in (Koloskova, Stich, and Jaggi, 2022):

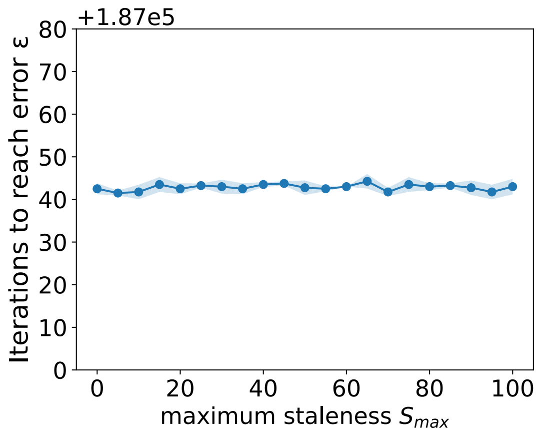
- a *quadratic function* $f(x) = \frac{1}{2}\|Ax - b\|^2$, with $x, b \in \mathbb{R}^{10}$, $i \in [1, 10]$, $b_i \sim \mathcal{N}(0, 1)$, and $A \in \mathbb{R}^{10x10}$ being a random matrix with $\lambda_{min}(A) = 1$, $\lambda_{max}(A) = 2$ (cf. Figure 5.2a);
- a *logistic-regression function* $f(x) = \frac{1}{m}\sum_{j=1}^{m}\log(1 + \exp(-b_j\ a_j^T\ x))$, with $a_j \sim \mathcal{N}(0, 1)^{20}$, $x \in \mathbb{R}^{20}$, $m = 100$, and b_j is sampled uniformly at random from $\{-1, 1\}$ (cf. Figure 5.2b).

In addition, we analyzed the impact of staleness on the convergence rate and time of a Convolutional Neural Network (CNN) for image classification (*Convolutional Neural Network (CNN): Tensorflow Core* 2022), which we trained using the CIFAR-10 dataset (Krizhevsky, Hinton, et al., 2009). In this experiment, we set $\epsilon = 1$, $\eta = 1e^{-4}$, and varied n between 1 and 25. By varying n, we ended up increasing $\hat{S}_{max}$ indirectly—close to what we may expect in a real-world scenario. The results can be seen in Figure 5.2c.

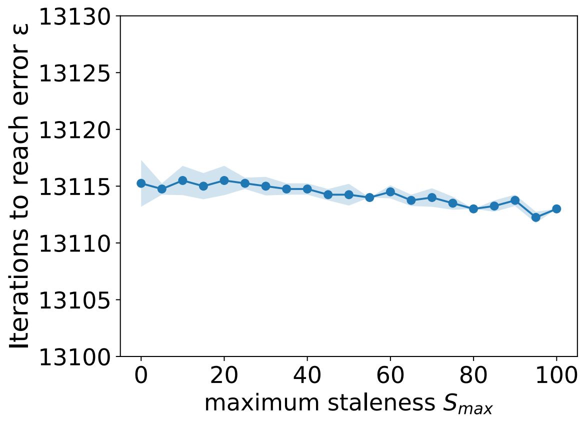
When analyzing Figure 5.2, we observe no significant impact of $\hat{S}_{max}$ on the number of iterations necessary to reach ϵ. This confirms the very good convergence of DASGD in a homogeneous setting with a fixed learning rate, which is even largely invariant of $\hat{S}_{max}$ and thus much better in practice than predicted by our bounds. Consequently, in Figure 5.2c, we observe a considerable decrease in the amount of time necessary to reach ϵ, which exemplifies the practical viability of DASGD.

In the Supplementary Material (Section 5.9.4), we further compare DASGD with state-of-the-art baselines in a real-world use-case training VGG-16 (Simonyan and Zisserman, 2015) using the Tiny ImageNet (Le and Yang, 2015) dataset.

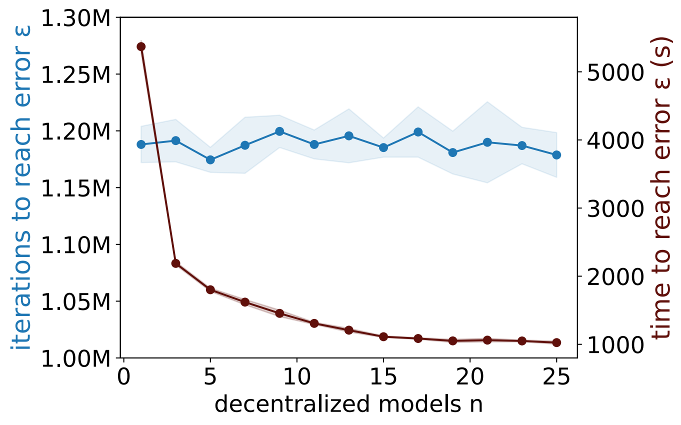
[2]https://github.com/maurodlt/dasgd



(A) Logistic regression



(B) Quadratic function



(C) CIFAR

FIGURE 5.2: Numbers of iterations and runtimes needed to reach an error of ϵ (each averaged over 4 runs, the shaded areas denote one standard deviation).

5.8 Conclusion

In this paper, we prove the convergence rate of decentralized asynchronous SGD (DASGD). Our proof does not depend on partial synchronization among models, complex network topologies, nor on unrealistic assumptions about the objective function. We introduce a generic staleness measure to quantify the differences between decentralized models over time. Moreover, our resulting bounds show that, over time, the impact of the stochastic noise σ on the convergence of the function is higher than the impact of the staleness components S_{avg}, $\hat{S}_{max}$, and $\hat{S}_{avg}$, respectively. We demonstrated that our theoretical results are even outperformed by our empirical evaluation over various optimization objectives, including a logistic regression, a quadratic function, and a CNN.

5.9 Supplementary Material

5.9.1 Useful Inequalities & Remarks

5.9.1.1 Inequalities

Below, we list a number of useful inequalities to which we will refer in our proof of Theorem 5.6.1.

Lemma 5.9.1. *In analogy to (Koloskova, Stich, and Jaggi, 2022), we establish the following inequality for any set of n vectors* $\{a_i\}_{i=1}^n$ *with* $a_i \in \mathbb{R}^d$*:*

$$\|\sum_{i=1}^{n} a_i\|^2 \leq n \sum_{i=1}^{n} \|a_i\|^2 \tag{5.13}$$

Lemma 5.9.2. *For a vector* $a \in \mathbb{R}^d$ *and a multiplier* $m \in \mathbb{R}$*, it holds that:*

$$\|ma\|^2 \leq m^2 \|a\|^2 \tag{5.14}$$

Lemma 5.9.3. *Considering Assumption 3, for any function f, it holds that:*

$$\|\nabla f(y)\| \leq \|\nabla f(x)\| + L\|x - y\| \qquad \forall x, y \in \mathbb{R}^d \tag{5.15}$$

Lemma 5.9.4. *From the polarization identity, we have:*

$$\langle a, b \rangle = \frac{\|a\|^2}{2} + \frac{\|b\|^2}{2} - \frac{\|a - b\|^2}{2} \tag{5.16}$$

Lemma 5.9.5. *For any* $a, b \in \mathbb{R}$*, it holds that:*

$$(a + b)^2 \leq 2a^2 + 2b^2 \tag{5.17}$$

5.9.1.2 Remarks

Here, we provide the following useful remarks.

Remark 2. *$\Delta_{i,j}^{t,s}$ is larger or equal to the sum of any subset $A \subseteq \hat{S}_{i,j}^{t,s}$ (scaled with η), that is:*

$$\eta \sum_{g \in A} |g| \leq \Delta_{i,j}^{t,s}$$

We can guarantee this because $\Delta_{i,j}^{t,s}$ is the sum of the absolute values of all gradients in $\hat{S}_{i,j}^{t,s}$ (also scaled with η). Take as an example a set $\hat{S}_{i,j}^{t,s} = \{g_\alpha, g_\beta, g_\gamma\}$. Thus, $\Delta_{i,j}^{t,s} = \eta(|g_\alpha| + |g_\beta| + |g_\gamma|)$. Evidently, $\Delta_{i,j}^{t,s} \geq \eta|g_\alpha|$. Consequently, we can also reach the following remark.

Remark 3. *Considering $|x_\alpha| = x_i^t - \eta \sum_{g \in A} |g|$ and $A \subseteq \hat{S}_{i,j}^{t,s}$, we can guarantee that:*

$$x_i^t - |x_\alpha| \leq \Delta_{i,j}^{t,s} \tag{5.18}$$

Remark 2 follows the same idea as Remark 3. By definition, $\Delta_{i,j}^{t,s} = x_i^t - |x_\alpha| - |x_\beta|$, considering $|x_\beta| = x_i^t - \eta \sum_{g \in B} |g|$ and B being the complementary set of A and $\hat{S}_{i,j}^{t,s}$, that is $B = \overline{(A \cup \hat{S}_{i,j}^{t,s})}$. Thus, $x_i^t - |x_\alpha| \leq \Delta_{i,j}^{t,s}$.

5.9.2 Proof of Theorem 5.6.1

Our proof for Theorem 5.6.1 closely follows the structure of the proofs provided in Koloskova, Stich, and Jaggi in (Koloskova, Stich, and Jaggi, 2022) (specifically the ones leading to Theorem 6).

First, let us recall our Theorem 5.6.1 from Section 5.6.

Theorem 1. *Considering Assumptions 1, 2, 3, 4, and a constant stepsize $\eta \leq \frac{1}{4LS_{avg}}$, Algorithm 3 reaches $\frac{1}{T+1}\sum_{t=0}^{T}(\|\nabla f(x^t)\|^2) \leq \epsilon$ after*

$$\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{QS_{avg}}{\epsilon^{\frac{3}{2}}}) + \mathcal{O}(\frac{S_{avg}}{\epsilon}) \qquad \textit{iterations.} \tag{5.11}$$

Moreover, without Assumption 4 and $\eta \leq \frac{1}{4L\sqrt{\hat{S}_{avg}\hat{S}_{max}}}$, Algorithm 3 reaches $\frac{1}{T+1}\sum_{t=0}^{T}(\|\nabla f(x^t)\|^2) \leq \epsilon$ after

$$\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{\epsilon}) \qquad \textit{iterations.} \tag{5.12}$$

First, we define $\delta_{i,j}^{t,s} \in \mathbb{R}^d$ as the difference between two models x_i^t and x_j^s, which can also be represented as the sum of the gradients in the staleness set $S_{i,j}^{t,s}$ scaled by η, as follows:

$$\delta_{i,j}^{t,s} = \sum_{g \in S_{i,j}^{t,s}} \eta g \tag{5.19}$$

Then, we represent its upper bound $\Delta_{i,j}^{t,s}, \in \mathbb{R}^d$ as the summation of the absolute values of the gradients in $S_{i,j}^{t,s}$ (also scaled by η), expressed as follows:

$$\Delta_{i,j}^{t,s} = \sum_{g \in S_{i,j}^{t,s}} \eta|g| \geq \delta_{i,j}^{t,s} \tag{5.20}$$

Now, we define Lemma 5.9.7 which will be used to reach the convergence rates of Equations (5.11) and (5.12).

Lemma 5.9.7 (Descent Lemma)**.** *Considering Assumptions 1, 2 and 3 with a stepsize $\eta \le \frac{1}{2L}$, we have:*

$$\mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] \le f(x_i^t) - \frac{\eta}{2}\|\nabla f(x_i^t)\|^2 + L\eta^2\sigma^2 + \frac{\eta L^2}{2}\|\Delta_{i,j}^{t,s}\|^2 \tag{5.21}$$

Proof.

Following (Nguyen et al., 2022; Koloskova, Stich, and Jaggi, 2022) and due to the L-smoothness of f, when model x_i^t updates itself with a gradient computed by model x_j^s, it then holds that:

$$\begin{aligned}\mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] &= \mathbb{E}_{t+1}\left[f(x_i^t - \eta\nabla F(x_i^t + \delta_{i,j}^{t,s}, \xi_j^s))\right] \\ &\le f(x^{(t)}) - \underbrace{\eta\,\mathbb{E}_{t+1}\left[\langle\nabla f(x_i^t), \nabla F(x_i^t + \delta_{i,j}^{t,s}, \xi_j^s)\rangle\right]}_{T_1} \\ &\quad + \mathbb{E}_{t+1}\left[\frac{L\eta^2}{2}\underbrace{\|\nabla F(x_i^t + \delta_{i,j}^{t,s}, \xi_j^s)\|^2}_{T_2}\right]\end{aligned}$$

Note that, due to the function homogeneity assumption (Assumption 2), we omit the indices of F and f throughput our proofs to simplify their notations.

We first transform T_1 as follows:

$$\begin{aligned}T_1 &= -\eta\,\mathbb{E}_{t+1}\left[\langle\nabla f(x_i^t), \nabla F(x_i^t + \delta_{i,j}^{t,s}, \xi_j^s)\rangle\right] \\ &= -\eta\,\langle\nabla f(x_i^t), \nabla f(x_i^t + \delta_{i,j}^{t,s})\rangle \\ &\overset{(5.16)}{=} -\frac{\eta}{2}\|\nabla f(x_i^t)\|^2 - \frac{\eta}{2}\|\nabla f(x_i^t + \delta_{i,j}^{t,s})\|^2 + \frac{\eta}{2}\|\nabla f(x_i^t) - \nabla f(x_i^t + \delta_{i,j}^{t,s})\|^2\end{aligned}$$

Next, we transform T_2 as follows:

$$\begin{aligned}T_2 &= \mathbb{E}_{t+1}\left[\|\nabla F(x_i^t + \delta_{i,j}^{t,s}, \xi_j^s)\|^2\right] \\ &\overset{(5.2)}{\le} \sigma^2 + \|\nabla f(x_i^t + \delta_{i,j}^{t,s})\|^2\end{aligned}$$

Then, we combine again T_1 and T_2:

$$\begin{aligned}\mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] &\le f(x_i^t) - \frac{\eta}{2}\|\nabla f(x_i^t)\|^2 - \frac{\eta}{2}(1 - L\eta)\|\nabla f(x_i^t + \delta_{i,j}^{t,s})\|^2 \\ &\quad + \frac{\eta}{2}\|\nabla f(x_i^t) - \nabla f(x_i^t + \delta_{i,j}^{t,s})\|^2 + \frac{L\eta^2\sigma^2}{2}\end{aligned}$$

By exploiting the L-smoothness to estimate $\|\nabla f(x_i^t) - \nabla f(x_i^t + \delta_{i,j}^{t,s})\|^2 \le L^2\|x_i^t - (x_i^t + \delta_{i,j}^{t,s})\|^2$, we obtain:

$$\mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] \overset{(5.4)}{\leq} f(x_i^t) - \frac{\eta}{2}\|\nabla f(x_i^t)\|^2 - \frac{\eta}{2}(1-L\eta)\|\nabla f(x_i^t+\delta_{i,j}^{t,s})\|^2 + \frac{\eta L^2}{2}\|x_i^t-(x_i^t+\delta_{i,j}^{t,s})\|^2 + L\eta^2\sigma^2$$

Simplifying the fourth term, we get:

$$\mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] \leq f(x_i^t) - \frac{\eta}{2}\|\nabla f(x_i^t)\|^2 - \frac{\eta}{2}(1-L\eta)\|\nabla f(x_i^t+\delta_{i,j}^{t,s})\|^2 + \frac{\eta L^2}{2}\|\delta_{i,j}^{t,s}\|^2 + L\eta^2\sigma^2$$

By applying $\eta \leq \frac{1}{2L}$, we then obtain:

$$\leq f(x_i^t) - \frac{\eta}{2}\|\nabla f(x_i^t)\|^2 - \frac{\eta}{4}\|\nabla f(x_i^t+\delta_{i,j}^{t,s})\|^2 + L\eta^2\sigma^2 + \frac{\eta L^2}{2}\|\delta_{i,j}^{t,s}\|^2$$

By discarding $-\frac{\eta}{4}\|\nabla f(x_i^t+\delta_{i,j}^{t,s})\|^2$ from the right-hand side of the inequality and considering $\Delta_{i,j}^{t,s} \geq \delta_{i,j}^{t,s}$, we reach Lemma 5.9.7. □

5.9.2.1 Preliminaries for the Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.12)

Lemma 5.9.8 (Estimation of the residual). *By considering Assumptions 1, 2 3 and a constant stepsize $\eta \leq \frac{1}{4L\sqrt{\hat{S}_{avg}\hat{S}_{max}}}$, we have:*

$$\frac{1}{(T+1)}\sum_{t=0}^{T}\mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq \frac{1}{7L^2(T+1)}\sum_{t=0}^{T}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] + \frac{2\eta\sigma^2}{7L}$$

Proof.
First, we unroll $\Delta_{i,j}^{t,s}$ as follows:

$$\mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] = \mathbb{E}\left[\left\|\sum_{g\in\hat{S}_{i,j}^{t,s}}\eta|g|\right\|^2\right]$$

Let $g_k^u \in \hat{S}_{i,j}^{t,s}$ be a gradient calculated by any model k at any iteration u, then we have:

$$= \mathbb{E}\left[\left\|\sum_{g_k^u\in\hat{S}_{i,j}^{t,s}}\eta|\nabla F(x_k^u,\xi_k^u)|\right\|^2\right]$$

Now, by considering Lemmas 5.9.1 and 5.9.2 as well as Assumption 1, we have:

$$\overset{(5.2)}{\leq} \mathbb{E}\left[\left\|\sum_{g_k^u \in \hat{S}_{i,j}^{t,s}} \eta |\nabla f(x_k^u)|\right\|^2\right] + |\hat{S}_{i,j}^{t,s}|\eta^2\sigma^2 \tag{5.22}$$

$$\overset{(5.13)}{\leq} |\hat{S}_{i,j}^{t,s}|\, \mathbb{E}\left[\sum_{g_k^u \in \hat{S}_{i,j}^{t,s}} \eta \|\nabla f(x_k^u)\|^2\right] + |\hat{S}_{i,j}^{t,s}|\eta^2\sigma^2 \tag{5.23}$$

$$\overset{(5.14)}{\leq} |\hat{S}_{i,j}^{t,s}|\eta^2\, \mathbb{E}\left[\sum_{g_k^u \in \hat{S}_{i,j}^{t,s}} \|\nabla f(x_k^u)\|^2\right] + |\hat{S}_{i,j}^{t,s}|\eta^2\sigma^2 \tag{5.24}$$

By considering the L-smoothness to estimate $\nabla f(x_k^u)$, we obtain:

$$\overset{(5.15)}{\leq} |\hat{S}_{i,j}^{t,s}|\eta^2\, \mathbb{E}\left[\sum_{g_k^u \in \hat{S}_{i,j}^{t,s}} (\|\nabla f(x_i^t)\| + L\|x_i^t - x_k^u\|)^2\right] + |\hat{S}_{i,j}^{t,s}|\eta^2\sigma^2$$

Next, consider Remark 3.

$$\overset{(5.18)}{\leq} |\hat{S}_{i,j}^{t,s}|\eta^2\, \mathbb{E}\left[\sum_{g \in \hat{S}_{i,j}^{t,s}} (\|\nabla f(x_i^t)\| + L\|\Delta_{i,j}^{t,s}\|)^2\right] + |\hat{S}_{i,j}^{t,s}|\eta^2\sigma^2$$

Let us assume $\eta \leq \frac{1}{4L\sqrt{\hat{S}_{avg}\hat{S}_{max}}}$ in the first term of the right-hand side of the inequality and $|\hat{S}_{i,j}^{t,s}| \leq \hat{S}_{max}$. We then obtain:

$$\leq \frac{1}{16L^2\hat{S}_{avg}}\, \mathbb{E}\left[\sum_{g \in \hat{S}_{i,j}^{t,s}} (\|\nabla f(x_i^t)\| + L\|\Delta_{i,j}^{t,s}\|)^2\right] + |\hat{S}_{i,j}^{t,s}|\eta^2\sigma^2$$

By summing over all steps $t \in T$, we obtain:

$$\sum_{t=0}^{T} \mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \overset{(5.10)}{\leq} \frac{1}{16L^2\hat{S}_{avg}} \sum_{t=0}^{T} \sum_{g \in \hat{S}_{i,j}^{t,s}} \mathbb{E}\left[\|\nabla f(x_i^t)\| + L\|\Delta_{i,j}^{t,s}\|\right]^2 + (T+1)\hat{S}_{avg}\eta^2\sigma^2$$

By considering that the inner summation from the right side of the equation is bounded by $|\hat{S}_{i,j}^{t,s}|$ and is executed T times, we can simplify the equation by using Definition 2. This simplification upper bounds the number of iterations over $\nabla f(x_i^t)$ and $\Delta_{i,j}^{t,s}$ to $\hat{S}_{avg}$ times, thus leading us to:

$$\overset{(5.10)}{\leq} \frac{1}{16L^2} \sum_{t=0}^{T} \mathbb{E}\left[\|\nabla f(x_i^t)\| + L\|\Delta_{i,j}^{t,s}\|\right]^2 + (T+1)\hat{S}_{avg}\eta^2\sigma^2$$

From Lemma 5.9.5, we can further consider that $\mathbb{E}\left[\|\nabla f(x_i^t)\| + L\|\Delta_{i,j}^{t,s}\|\right]^2 \leq \mathbb{E}\left[2\|\nabla f(x_i^t)\|^2 + 2L^2\|\Delta_{i,j}^{t,s}\|^2\right]$, thus:

$$\sum_{t=0}^{T} \mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq \frac{1}{16L^2} \sum_{t=0}^{T} \mathbb{E}\left[2\|\nabla f(x_i^t)\|^2 + 2L^2\|\Delta_{i,j}^{t,s}\|^2\right] + (T+1)\hat{S}_{avg}\eta^2\sigma^2$$

Then, by taking all terms based on $\|\Delta_{i,j}^{t,s}\|$ to the left side of the inequality:

$$(1-\frac{2}{16}) \sum_{t=0}^{T} \mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq \frac{1}{16L^2} \sum_{t=0}^{T} \mathbb{E}\left[2\|\nabla f(x_i^t)\|^2\right] + (T+1)\hat{S}_{avg}\eta^2\sigma^2$$

Later simplifying the inequality, we reach:

$$\frac{7}{8} \sum_{t=0}^{T} \mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq \frac{1}{8L^2} \sum_{t=0}^{T} \mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] + (T+1)\hat{S}_{avg}\eta^2\sigma^2$$

Then, by multiplying the inequality by 8/7, we have:

$$\sum_{t=0}^{T} \mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq \frac{1}{7L^2} \sum_{t=0}^{T} \mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] + \frac{8(T+1)}{7}\hat{S}_{avg}\eta^2\sigma^2$$

At last, by assuming $\eta \leq \frac{1}{4L\sqrt{\hat{S}_{max}\hat{S}_{avg}}}$, and $\hat{S}_{max} \geq \hat{S}_{avg}$ such that $\sqrt{\hat{S}_{max}S_{avg}} \geq \hat{S}_{avg}$. Then, it holds that:

$$\sum_{t=0}^{T} \mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq \frac{1}{7L^2} \sum_{t=0}^{T} \mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] + \frac{2(T+1)}{7}L\eta\sigma^2$$

After dividing by $(T+1)$, we reach the statement of the lemma. □

5.9.2.2 Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.12)

We are now ready to give the proof of Theorem 5.6.1 with the convergence rate of Equation (5.12). We start by passing $f(x_i^{t+1})$ to the right-hand side of the inequality of Lemma 5.9.7 and $f(x_i^t)$ to the left-hand side. Then, for every model $i \in [n]$, it holds that:

$$\frac{\eta}{2}\|\nabla f(x_i^t)\|^2 \leq f(x_i^t) - \mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] + L\eta^2\sigma^2 + \frac{\eta L^2}{2}\|\Delta_{i,j}^{t,s}\|^2$$

Then, we average over all $t \in T$ and divide by η. In the following, let $f*$ denote the value of our objective function at a local minimum ϵ.

$$\frac{1}{T+1}\sum_{t=0}^{T}\frac{1}{2}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{1}{\eta(T+1)}(f(x_i^0)-f^*) + L\eta\sigma^2 + \frac{1}{(T+1)}\frac{L^2}{2}\sum_{t=0}^{T}\mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right]$$

We next apply Lemma 5.9.8 to the last term in order to obtain:

$$\frac{1}{T+1}\sum_{t=0}^{T}\frac{1}{2}\,\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{1}{\eta(T+1)}(f(x_i^0)-f^*)+L\eta\sigma^2 + \frac{L\eta\sigma^2}{7}+\frac{1}{14(T+1)}\sum_{t=0}^{T}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right]$$

By setting $f(x_i^0)-f^* =: r_0$ and considering that $\frac{L\eta\sigma^2}{7} < L\eta\sigma^2$, we get:

$$\frac{1}{T+1}\sum_{t=0}^{T}(\frac{1}{2}\,\mathbb{E}\left[\|\nabla f(x_i^t)\|^2)\right] \leq \frac{r_0}{\eta(T+1)}+2L\eta\sigma^2+\frac{1}{14(T+1)}\sum_{t=0}^{T}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right]$$

We next pass $\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right]$ to the left-hand side of the inequality.

$$\frac{1}{T+1}\sum_{t=0}^{T}(\frac{1}{2}-\frac{1}{14})\,\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{r_0}{\eta(T+1)}+2L\eta\sigma^2$$
$$\frac{1}{T+1}\sum_{t=0}^{T}\frac{3}{7}\,\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{r_0}{\eta(T+1)}+2L\eta\sigma^2$$

We then multiply the inequality by $\frac{7}{3}$ as follows:

$$\frac{1}{T+1}\sum_{t=0}^{T}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{7r_0}{3\eta(T+1)}+\frac{14}{3}L\eta\sigma^2$$

Using $\eta \leq \frac{1}{4L\sqrt{\hat{S}_{avg}\hat{S}_{max}}}$ together with Lemma 17 from (Koloskova et al., 2020).

$$c\Psi_T \leq 2(\frac{14L\sigma^2 r_0}{3(T+1)})^{\frac{1}{2}}+\frac{4Lr_0\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{T+1}$$

This finally yields the bound of Theorem 5.6.1 with the convergence rate of Equation (5.12).

$$\mathcal{O}(\frac{\sigma}{\sqrt{T}})+\mathcal{O}(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{T})$$

□

5.9.2.3 Preliminaries for the Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.11)

Lemma 5.9.9 (Estimation of the residual – bounded gradients)**.** *Considering Assumptions 1, 2, 3 and 4 with a constant stepsize $\eta \leq \frac{1}{4LS_{avg}}$, we have:*

$$\frac{1}{T+1}\sum_{i=0}^{T}\mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \leq S_{avg}^2\eta^2 Q^2 + S_{avg}\eta^2\sigma^2$$

Proof.
From Equation (5.24), when using the tighter representation of staleness $|S_{i,j}^{t,s}|$, we obtain:

$$\leq |S_{i,j}^{t,s}|\eta^2 \mathbb{E}\left[\sum_{g_k^u \in S_{i,j}^{t,s}} \|\nabla f(x_k^u)\|^2\right] + |S_{i,j}^{t,s}|\eta^2\sigma^2$$

By considering Assumption 4, in which $\|\nabla f(x_k^u)\|^2 \leq Q^2$, we have:

$$\mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right] \overset{(5.5)}{\leq} |S_{i,j}^{t,s}|^2\eta^2 Q^2 + |S_{i,j}^{t,s}|\eta^2\sigma^2$$

Then, by averaging over all steps $t \in T$, we already obtain the statement of the lemma. □

5.9.2.4 Proof of Theorem 5.6.1 with the Convergence Rate of Equation (5.11)

Finally, we give the proof of Theorem 5.6.1 with the convergence rate of Equation (5.11). We start by passing $f(x_i^{t+1})$ to the right-hand side of the inequality of Lemma 5.9.7 and $f(x_i^t)$ to the left-hand side, respectively.

$$\frac{\eta}{2}\|\nabla f(x_i^t)\|^2 \leq f(x_i^t) - \mathbb{E}_{t+1}\left[f(x_i^{t+1})\right] + L\eta^2\sigma^2 + \frac{\eta L^2}{2}\|\Delta_{i,j}^{t,s}\|^2$$

Then, we average over all steps $t \in T$ and divide by η.

$$\frac{1}{T+1}\sum_{t=0}^{T}\frac{1}{2}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{1}{\eta(T+1)}(f(x_i^0 - f^*) + L\eta\sigma^2 + \frac{1}{T+1}\frac{L^2}{2}\sum_{t=0}^{T}\mathbb{E}\left[\|\Delta_{i,j}^{t,s}\|^2\right]$$

We next apply Lemma 3 to the last term.

$$\frac{1}{T+1}\sum_{t=0}^{T}\frac{1}{2}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{1}{\eta(T+1)}(f(x_i^0 - f^*) + L\eta\sigma^2 + \frac{L^2 S_{avg}^2\eta^2 Q^2}{2} + \frac{L^2 S_{avg}\eta^2\sigma^2}{2}$$

Let again $f(x_i^0) - f^* =: r_0$. We simplify the inequality with the following conditions:

- we multiply the inequality by 2;
- let $\eta \leq \frac{1}{4LS_{avg}}$ on the last term of the inequality;
- we consider that $\frac{2L^2 S_{avg}\eta^2\sigma^2}{2} \leq \frac{L\eta\sigma^2}{4} \leq L\eta\sigma^2$.

We therefore obtain:

$$\frac{1}{T+1}\sum_{t=0}^{T}\mathbb{E}\left[\|\nabla f(x_i^t)\|^2\right] \leq \frac{2r_0}{\eta(T+1)} + 3L\eta\sigma^2 + L^2 S_{avg}^2\eta^2 Q^2$$

Let us assume $\eta \leq \frac{1}{4LS_{avg}}$, then together with Lemma 17 from (Koloskova et al., 2020), we obtain:

$$\Psi_T \leq 2(\frac{3L\sigma^2 r_0}{T+1})^{\frac{1}{2}} + 2(L^2 S_{avg}^2 Q^2)^{\frac{1}{3}}(\frac{r_0}{T+1})^{\frac{2}{3}} + \frac{4Lr_0 S_{avg}}{T+1}$$

This finally yields the bound of Theorem 5.6.1 with the convergence rate of Equation (5.11).

$$\mathcal{O}(\frac{\sigma}{\sqrt{T}}) + \mathcal{O}(\frac{QS_{avg}}{T^{\frac{2}{3}}}) + \mathcal{O}(\frac{S_{avg}}{T})$$

□

5.9.3 DASGD - reduced communication

Below, we introduce Algorithm 4, which bounds the frequency of communication among models during the optimization process. When compared to Algorithm 3, this adjustment results in reduced communication complexity while producing a higher *staleness* during the optimization.

Algorithm 4 DASGD - reduced communication.

```
1:  Constructor DASGD ($x^0$, $i$, $broadcast\_frequancy$)
2:      $t = 0$
3:      $x_i^t = x^0$
4:      $\Delta = \varnothing$
5:      $delta\_count = 0$
6:      $bf = broadcast\_frequancy$
7:  procedure TRAINMODEL
8:      while true do
9:          $\Delta_{apply} =$ receiveGradient()   ▷ //NULL if there is no incoming gradient
10:         if $\Delta_{apply} \neq$ NULL then
11:             $x_i^{t+bf} = x_i^t - \Delta_{apply}$
12:             $t = t + bf$
13:         else if $\exists\, \xi_i^t$ then                ▷ //compute a new gradient
14:             $g_i^t = \nabla F(x_i^t, \xi_i^t)$
15:             $x_i^{t+1} = x_i^t - \eta g_i^t$
16:             $\Delta = \Delta + \eta g_i^t$
17:             $delta\_count = delta\_count + 1$
18:             if $delta\_count = bf$ then
19:                 sendGradient($\Delta$)              ▷ //non-blocking
20:                 $\Delta = \varnothing$
21:                 $delta\_count = 0$
22:             $t = t + 1$
```

In Algorithm 4, analogously to Algorithm 3, we show the protocol performed by all $i \in [n]$ worker nodes. We also initialize the local iteration counter t and all nodes with the same model parameters x^0. Additionally, we define the frequency bf in which the models will broadcast their local gradients. At last, we initialize another two variables: (1) Δ, representing the sum of local gradients calculated by model x_i until they are broadcasted; and (2) *delta_count*, a counter describing the number of gradients summed into Δ.

During training, from lines 9 to 12, node i checks if any sum of gradients Δ_{apply} calculated by other models is available (line 9). If so, it promptly updates itself based on it (line 11). Then, it increments its iteration counter by bf (line 12), as this is the number of gradients aggregated in each message received from other nodes.

If no gradients were available in line 10, node i checks if new training samples ξ_i^t are available (line 13). Then, it calculates a local gradient g_i^t based on its current model weights x_i^t (line 14). After calculating g_i^t, the local model x_i^t updates itself based on it (line 15) and sums it into Δ (line 16). Later, it also increments the counter *delta_count* (line 17). Then, the algorithm checks if Δ is composed of bf gradients (line 18). If so, Δ is broadcasted (line 19) and the variables Δ and *delta_count* are reinitialized (lines 20 and 21). At last, the local iteration counter t is incremented, and Algorithm 4 resumes the training on line 8.

Analogously to Algorithm 3, the training is performed until $\nexists \xi_i^t, \forall i \in [n]$ and all Δ were applied by all nodes $i \in [n]$.

Discussion. Algorithm 4 is a simple relaxation of Algorithm 3 that allows gradients to be broadcasted every bf iterations. Therefore, instead of performing a broadcast at every iteration, which results in a communication complexity of $\mathcal{O}(n^2)$, we can define $bf = n$ and reduce the communication complexity to $\mathcal{O}(n)$. This relaxation greatly alleviates the network burden. Additionally, it also reduces in bf times the time taken in the gradient application process (line 11 in Algorithm 3) because each local model applies a single Δ_{apply} instead of bf gradients g (line 11 in Algorithm 4). The gradient application, despite not dominating the time taken at each optimization step, still represents a considerable portion of the training time. Nevertheless, this relaxation also increases the model *stalaness* during the optimization, which can hinder its convergence when too high. In practice, we recommend setting $bf = n$ to avoid communication bottlenecks while maintaining a reasonably small *staleness*, producing good model convergence as seen in Section 5.9.4. At last, we note that Algorithm 3 is a special case of Algorithm 4, in which *broadcast_frequency* $= 1$ and, therefore, both algorithms have their convergence proven by Theorem 5.6.1.

5.9.4 VGG-16 Experiments

5.9.4.1 Experimental setting

In this Section, we further assess how DASGD behaves in a real-world use-case. We carried out our experiments on the HPC facilities of the University of Luxembourg (Varrette et al., 2022). We distributed the training process over up to 8 NVIDIA Tesla V100 GPUs divided in two nodes with 4GPUs, 28 CPUs, and 768 GB RAM each. All our experiments were performed simulating an all-connected network topology using an MPI environment, launching one rank per GPU available with no shared memory among ranks.

We trained the VGG-16 model (Simonyan and Zisserman, 2015) using the Tiny ImageNet (Le and Yang, 2015) dataset. The idea behind this experiment is to identify how DASGD behaves when optimizing a large model such as the VGG-16, with more than 40 million parameters and 150MB. Considering that DASGD is decentralized and does not follow a gossip strategy, such large models are challenging as they may cause bottlenecks in the communication process. Considering this, we used Algorithm 4 to train the model and defined $bf = n$. As hyperparameters, we defined $\eta = 0.001$ and used a mini-batch of size 128.

5.9.4.2 Results and Discussion

The results of the VGG-16 experiments can be visualized in Figure 5.3.

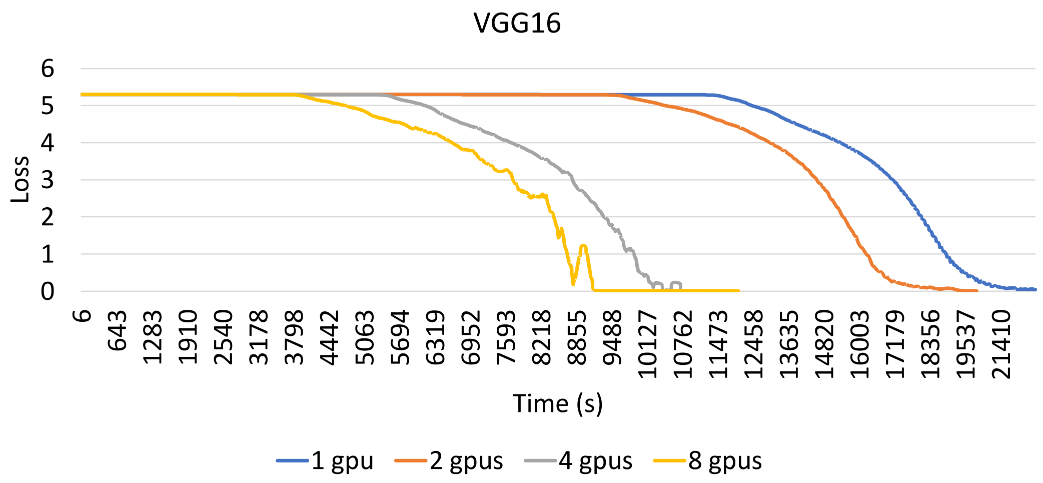


FIGURE 5.3: VGG-16 model trained with DASGD (Algorithm 4).

In Figure 5.3, we illustrate the convergence rate of the VGG-16 model when trained over 1, 2, 4, and 8 GPUs following Algorithm 4. We can see that, as expected, the higher the number of GPUs used during the optimization process, the lower the time it takes the model to converge. Still, differently from the experiments in Section 5.7, when training the VGG-16 model, we can verify the impact of the *staleness* during the optimization process, as given by Theorem 5.6.1. To reach $\epsilon = 1$ in the VGG-16 experiment, DASGD took: 125,034 iterations when training with 1 GPU; 127,475 iterations when training with 2 GPUs; 137,641 when training with 4 GPUs; and 169,801 when trained with 8 GPUs. The difference between the impact of *staleness* in the experiments in Section 5.7 and the ones presented here are mostly due to: (1) our usage of Algorithm 4 in the VGG-16 experiment, which increases the model *staleness* during the optimization process; and (2) the complexity of the VGG-16 model, which made it suffer more from the "noise" generated by staled gradients. Nevertheless, as seen in Figure 5.3, the speedup achieved when increasing the number of GPUs surpasses the increased number of iterations necessary to reach convergence.

Chapter 6

Implementation Aspects & Demonstration Setting

6.1 Contribution Statement

This chapter corresponds to the reprint of the demonstration paper "TensAIR: Real-Time Training of Distributed Artificial Neural Networks" (Tosi and Theobald, n.d.) authored by Mauro Dalle Lucca Tosi and Martin Theobald. Below, we provide the CRediT (Contributor Roles Taxonomy) [1] statements correspondent to each author contribution.

- **Mauro Dalle Lucca Tosi:**
  - Conceptualization;
  - Methodology;
  - Software;
  - Validation;
  - Formal analysis;
  - Investigation;
  - Data Curation;
  - Writing - Original Draft;
  - Visualization;
- **Martin Theobald**
  - Formal analysis;
  - Resources;
  - Writing - Review & Editing;
  - Supervision;
  - Project administration;
  - Funding acquisition

6.2 Chapter Contextualization

This chapter corresponds to the demonstration paper entitled "TensAIR: Real-Time Training of Distributed Artificial Neural Networks" (Tosi and Theobald, n.d.). It

[1] https://credit.niso.org/

demonstrates how to use TensAIR to train ANN models within an OL setting. This is a progression from the original TensAIR paper (Tosi, Venugopal, and Theobald, 2024), as it incorporates the active identification and adaptation of concept drifts via OPTWIN. Additionally, it introduces an intuitive Python interface, which simplifies the integration of TensAIR into existing ML and data science pipelines.

6.3 Introduction

Online Learning (OL), a subset of Machine Learning (ML), focuses on solving time-sensitive problems through sequential learning from individual data samples (Hoi et al., 2021). Unlike traditional ML tasks, OL operates on data streams, thus lacking prior access to the complete training data. This approach necessitates OL solutions to rely on the currently available data to yield the best-possible results. A critical aspect of data streams is their susceptibility to concept drifts, i.e., unexpected changes in the data's statistical properties (Lu et al., 2018). This variability renders standard pretrained ML models ineffective in OL scenarios, as concept drifts can significantly reduce the performance of their predictions over time. We therefore argue that OL requires flexible solutions which are in particular also capable of a real-time adaptation to concept drifts. Given the time-sensitive nature of OL, researchers and developers often resort to computationally efficient solutions in order to remain capable of an online adaptation to concept drifts. These are particularly suitable for simpler tasks, such as univariate time-series analysis. However, most contemporary methods fall short when complexity escalates, e.g., in case of non-linear, multivariate time series (Hoi et al., 2021), or when training complex neural networks with millions to billions of parameters (Devlin et al., 2019).

Artificial Neural Networks (ANNs) are acclaimed for their ability to manage high-dimensional problems, attributed to their advanced generalization capabilities (Goodfellow, Bengio, and Courville, 2016). However, their effectiveness is often constrained in environments experiencing concept drifts, primarily due to their reliance on offline training methods (like those presently integrated into Apache Kafka (*Robust machine learning on streaming data using Kafka and Tensorflow-IO* 2022) or Flink (*Deep Learning on Flink* 2022)). In such scenarios, once a concept drift is identified, traditional approaches necessitate the creation of a new training dataset and a subsequent retraining of the ANN model. This process can be time-consuming, and it critically depends on the volume of data and the duration of the retraining period during which the ANN model may yield sub-optimal predictions.

In contrast, if adapted to an OL setting, the inherently iterative nature of training ANNs via Stochastic Gradient Descent (SGD) offers a more dynamic solution. It allows for the immediate incorporation of new data samples from the stream into the ANN model and thereby streamlines the retraining process. This approach not only minimizes the time lag inherent in offline training but also enhances the quality of the predictions during the retraining phase. In this demonstration, we address the still prevalent lack of universal solutions for complex ML problems in an online setting, such as multivariate time series analysis and ANNs. To this end, we explore TensAIR (Tosi, Venugopal, and Theobald, 2024), a system designed for the real-time training of ANNs from data streams. By scaling-out the batching and SGD updates across multiple distributed models, we advocate that online training presents a great opportunity for maintaining high-quality predictions, even in the midst of ongoing model adjustments and/or in the presence of concept drifts.

6.3.1 TensAIR Extensions for the Demonstration

For this demonstration, TensAIR's core functionality has been enhanced with an active drift-detection strategy. Rather than training ANN models with every data sample, the system retrains models only upon detecting concept drifts. An accurate concept-drift detection spares computational resources and also avoids an unnecessary retraining of models which may actually be unaffected by concept drifts.

Additionally, recognizing the widespread use of Python in ML and Data Science (DS), we have developed a Python interface for TensAIR for this demonstration. This interface allows users to conveniently deploy the TensAIR system directly from an Jupyter notebook (Kluyver et al., 2016). It also provides real-time visualization of the prediction accuracy and the training progress during concept-drift adaptations.

6.4 Online Training ANN Models

TensAIR has been designed from scratch for training ANNs via a form of *decentralized and asynchronous SGD* (DASGD) (Tosi and Theobald, 2023) and is based on our previous AIR (Venugopal et al., 2020) stream processing engine and TensorFlow (Abadi et al., 2016) (both openly available as native C/C++ code). AIR offers a faster communication and a sustainable throughput that has been shown to be up to 14 times higher than conventional stream-processing engines like Apache Spark (Zaharia et al., 2016) and Flink (Carbone et al., 2015). This remarkable performance boost is attributed to its decentralized architecture and an efficient non-blocking communication strategy.

To effectively illustrate TensAIR's operation, one can conceptualize its dataflow using a graph, where vertices symbolize the various operators and directed edges depict the communication pathways between them. For our OL setting, the configuration of TensAIR involves two primary operators: the "Mini Batch Generator" and the "Model". This setup enables the ANN models to continuously update themselves through a form of passive drift adaptation, whereby the distributed model instances are continuously trained with the latest data samples without being explicitly aware of concept-drifts (Tosi, Venugopal, and Theobald, 2024). We further advanced this framework to incorporate an active drift-adaptation mechanism (as shown in Figure 3.1), where the models are retrained exclusively in response to detected concept drifts, thereby optimizing their computational resources by avoiding a retraining of the model instance when there is either no concept drift at all, or when they are not actually affected by a recent concept drift (i.e., when their prediction accuracy does not decrease).

In Figure 3.1, we illustrate our enhanced TensAIR dataflow. This setup involves three key components: the "Mini Batch Generator", the "Model", and the "Drift Detector". Each plays a crucial role in the system's operation. Specifically, the "Mini Batch Generator" acts as the entry point to this workflow. It takes data samples from the data stream and compiles them into mini-batches. These mini-batches are then used for either prediction or training purposes, depending on the model's current state. Next in the sequence is the "Model" operator. This component is responsible for initializing and managing n instances of ANNs, distributed across n ranks. These "Model" instances receive mini-batches from the "Mini Batch Generator". When the models are in a stable state, having converged and not having encountered any concept drift since their last training iteration, they use the incoming mini-batches for making predictions. The outcomes of these predictions, particularly the losses incurred, are then forwarded to the "Drift Detector".

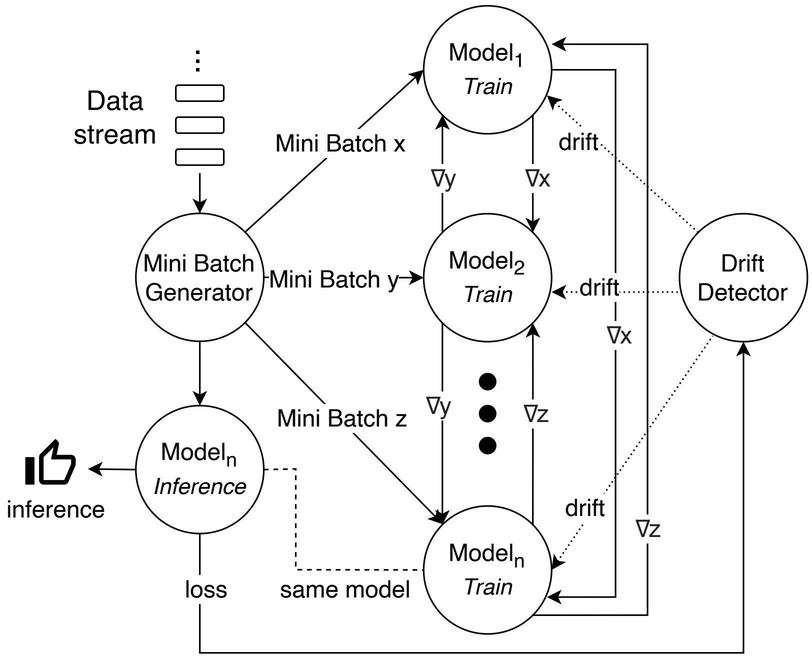


FIGURE 6.1: TensAIR dataflow with active drift adaptation.

However, if the "Model" instances are still adapting to the most recent concept drift (i.e., they have not converged yet), the mini-batches serve a different purpose. In this case, they are used for further training. During training sessions, the "Model" instances process the mini-batches, thereby calculating and applying the necessary SGD updates. These updates are broadcast across all model instances to ensure uniform learning and adaptation. This training continues until all "Model" instances have converged (Tosi and Theobald, 2023), at which point it transitions back to using the mini-batches for inference, as it is detailed in Algorithm 5.

The final and novel piece in this process is the "Drift Detector". This operator is critical for identifying changes in the data stream that might affect model performance. It examines the prediction losses received from the "Model" instances and determines, via our novel OPTWIN (Tosi and Theobald, 2024) drift-detection algorithm, whether there have been statistically significant changes in either the mean or the standard deviation of these losses. If such changes are detected, which indicates a concept drift, the "Drift Detector" signals the "Model" to shift its focus back to training mode using the next set of mini-batches.

6.4.1 Implementation

TensAIR is primarily developed in C++, utilizing the Message Passing Interface (MPI) (Message Passing Interface Forum, 2021) for efficient communication across its distributed architecture. This design allows for scalability; users can simply increase the number of MPI ranks to expand the system's training capacity. Notably, TensAIR supports training ANN models on nodes equipped with or without GPUs, offering also versatility in the hardware utilization (Tosi, Venugopal, and Theobald, 2024).

At its core, TensAIR integrates TensorFlow's native C API, enabling seamless loading of TensorFlow models, originally designed in Python, into the TensAIR environment. Nevertheless, the construction of TensAIR's dataflow and the data preprocessing pipeline is still handled in C++. This approach ensures a best-possible

Algorithm 5 TensAIR `Model` dataflow, including active drift adaptation

```
1:  training = True
2:  procedure StreamProcess($msg$)
3:      if $msg$ from Mini Batch Generator then
4:          if $training$ then
5:              gradient = calculateGradient($msg$)
6:              apply_gradient_locally(gradient)
7:              broadcast(gradient)
8:              converged = checkConvergence()
9:              if converged then
10:                 training = False
11:         else
12:             prediction, loss = predict($msg$)
13:             send_to_drift_detector(loss)
14:     else if $msg$ from Model then
15:         apply_gradient_locally($msg$)
16:         converged = checkConvergence()
17:         if converged then
18:             training = False
19:     else if $msg$ from Drift Detector then
20:         training = True
```

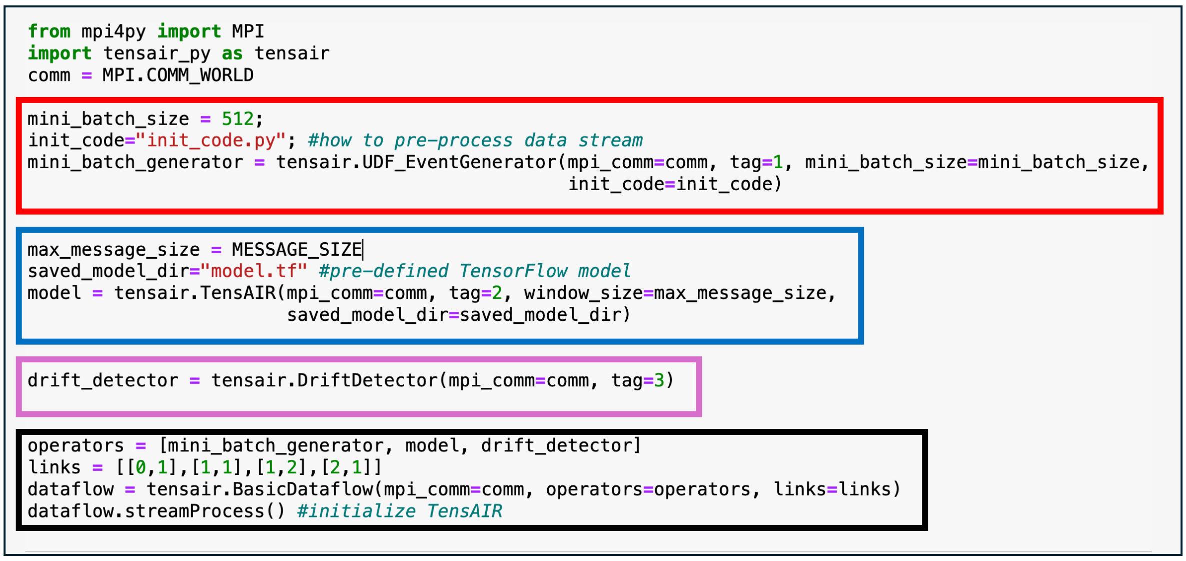
performance but traditionally poses challenges in integrating with ML and DS workflows, which predominantly use Python.

In response to this integration challenge, we introduce TensAIR's Python interface in this demonstration. This interface establishes a communication link with TensAIR's C++ backend, allowing users to construct TensAIR pipelines using Python. This significant enhancement alleviates the complexity typically associated with memory management and multithreading in C++ (tasks traditionally known for their propensity for errors and difficulty in debugging within asynchronous systems). Our Python interface streamlines the development process and thus reduces the time and complexity involved in programming custom use-cases. We anticipate that this user-friendly approach will foster a wider adoption of TensAIR within the ML community.

In Figure 6.2, we illustrate the ease of implementing a TensAIR dataflow using this Python interface. The code is structured into four main blocks: the first three blocks are dedicated to initializing the "Mini Batch Generator", the "Model", and the "Drift Detector" operators, respectively, while the fourth block orchestrates the overall TensAIR dataflow. Each operator requires a communication variable, instantiated earlier in the code using MPI, and a unique "tag" value, which is incremented for each operator.

Specifically, the "Mini Batch Generator" block is configured with parameters such as "mini batch size" and "init code". The "init code" is a Python script that outlines the methodology for reading and preprocessing data from the stream. This script includes a "next message" function, which processes the "mini batch size" input and outputs a serialized mini-batch of the specified size. The serialization format comprises the mini batch size, the number of input tensors, the size of each tensor, and the serialized tensors themselves, as depicted in Figure 6.3.

Next, the "Model" block is instantiated with inputs like the "saved model dir"



```python
from mpi4py import MPI
import tensair_py as tensair
comm = MPI.COMM_WORLD

mini_batch_size = 512;
init_code="init_code.py"; #how to pre-process data stream
mini_batch_generator = tensair.UDF_EventGenerator(mpi_comm=comm, tag=1, mini_batch_size=mini_batch_size,
                                                  init_code=init_code)

max_message_size = MESSAGE_SIZE
saved_model_dir="model.tf" #pre-defined TensorFlow model
model = tensair.TensAIR(mpi_comm=comm, tag=2, window_size=max_message_size,
                        saved_model_dir=saved_model_dir)

drift_detector = tensair.DriftDetector(mpi_comm=comm, tag=3)

operators = [mini_batch_generator, model, drift_detector]
links = [[0,1],[1,1],[1,2],[2,1]]
dataflow = tensair.BasicDataflow(mpi_comm=comm, operators=operators, links=links)
dataflow.streamProcess() #initialize TensAIR
```

FIGURE 6.2: Code exemplifying implementation of generic TensAIR dataflow.

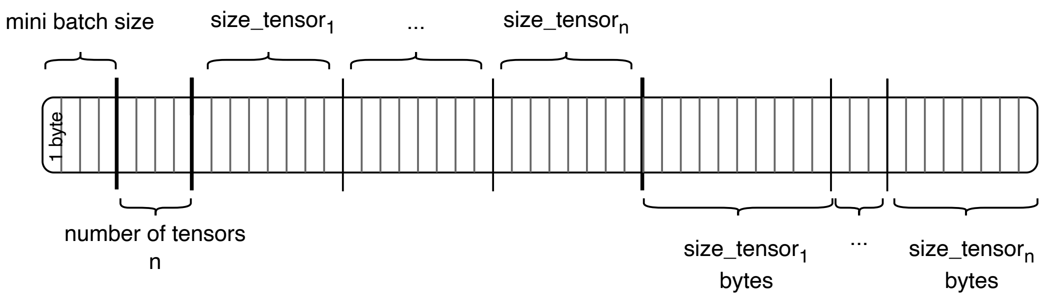


FIGURE 6.3: Encoding of message returned by *next message* method.

and “max message size”. The “saved model dir” specifies the desired directory of the TensorFlow model, while “max message size” determines the largest possible message size this operator can handle. This size is calculated based on the larger of two values: the output from the “next message” method and the combined size of the ANN’s trainable weights plus an additional 28 bytes for meta-data.

The “Drift Detector” block, comparatively straightforward, requires no extra parameters. Finally, the last block creates two lists — one for the operators and another one for the links between them — thus effectively defining TensAIR’s dataflow.

6.5 Demonstration Scenario

To effectively showcase TensAIR’s Python interface in an OL environment, while ensuring reproducibility, we adopt a simulation approach. Instead of using live data streams, we iterate over a predefined dataset in a sequential manner, processing one data sample per iteration. This demonstration tackles an image classification task, utilizing a Convolutional Neural Network (CNN) model (TensorFlow, 2022a) trained on the CIFAR-10 dataset (Krizhevsky, Hinton, et al., 2009). To simulate concept drifts, we introduce a label-swapping mechanism: every 10 epochs, the labels of two classes are interchanged. For example, after the first 10 epochs, images labeled as "dogs" are reclassified as "horses", and vice versa. This problem, while seemingly straightforward compared to those solved by advanced ANN architectures, presents a significant challenge for many ML algorithms, particularly those designed for online adaptation. Further options for switching labels and inducing concept drifts will be provided for the interested attendees of the demonstration.

To enhance the ease of use and interaction with TensAIR, we implement the entire process within an Jupyter notebook, rather than using a traditional Python script (which is executed via the command-line interface of MPI). This is achieved by incorporating the "ipyparallel" library, which allows for the asynchronous deployment of a specified number of ranks directly within the notebook environment. This setup not only streamlines the development process but also enables real-time monitoring of the training progress of distributed models in the same notebook.

A key feature of this demonstration is the ability to visually track and animate the model's performance using the "animation" module from the "matplotlib" library. This functionality is crucial for observing the dynamics of the model's loss metrics over time. By monitoring these metrics, we can directly discern the model's convergence patterns during the training phases and identify instances of concept drifts during the prediction phases. The visual representation of this data, including the evolution of the loss metrics and the model's response to concept drifts, is depicted in Figure 6.4.

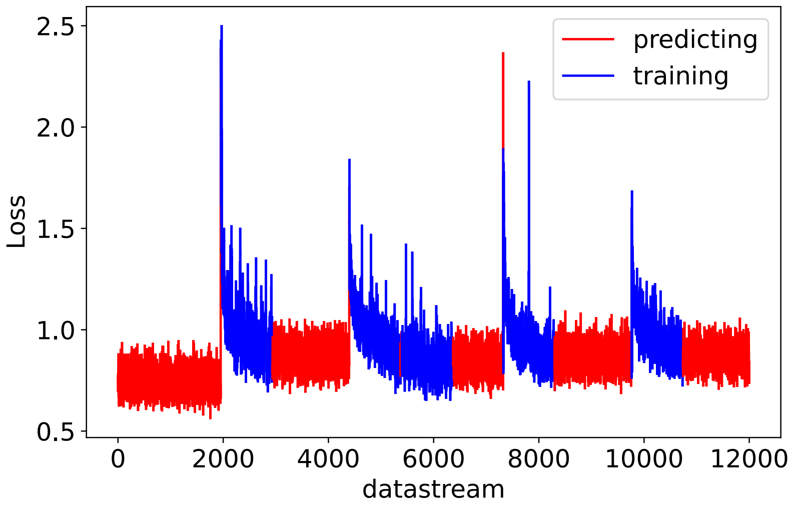


FIGURE 6.4: Prediction-loss curve of a CNN model over the CIFAR-10 dataset with simulated drifts. Each point from the data stream corresponds to a batch of 256 images.

In Figure 6.4, we observe a distinctive pattern in the model's performance. Initially, the model exhibits low loss metrics, thus indicating a high accuracy. However, upon the occurrence of a concept drift, there is a marked deterioration in performance. Notably, due to TensAIR's capability for online adaptation, the model quickly recalibrates itself in response to the drift, thus regaining its initial accuracy level for the predictions. This cycle of performance dip and recovery post-drift is a recurring phenomenon throughout the demonstration, underscoring the effectiveness of TensAIR's online learning and adaptation mechanisms in a dynamic streaming environment.

6.6 Conclusion

In this demonstration, we showcase the use of TensAIR's novel Python interface for training ANN models in an OL environment, emphasizing its capability for active concept-drift adaptation. This approach is particularly promising for multivariate time series analysis, where the adaptability and robustness of ANNs can address complex challenges more effectively than traditional methods. We believe that the integration of TensAIR offers great opportunities for advancements in handling dynamic data streams also across a wide range of other ML and DS domains.

Chapter 7

Discussion

In this chapter, we discuss the results achieved during the development of this Ph.D. thesis and how they attain the objectives we described in Chapter 1. First, let us recall that the main objective of this thesis is to design a distributed system that facilitates online training and prediction on Artificial Neural Network (ANN) models using incoming data streams as input in a scalable manner, with a specific emphasis on addressing concept drift in an Online Learning (OL) setting. To reach such objective we broke it down into four key objectives, which we addressed in the last four chapters of this thesis. In Section 7.1, we discuss how the results from the last four chapters reach the key objectives we formerly defined. In Section 7.2, we present and discuss the main limitations from the approaches we presented during the development of this thesis. Finally, in Section 7.3 we present a synthesis of our results and how their combination enable us to fulfill the main objective of this thesis.

7.1 Discussion on the Individual Research Objectives

In this section, we delve into the achievement of the four defined key objectives outlined in Section 1.4 through the results presented in the last five chapters of this thesis.

1. **Design a dataflow architecture that enables online training and prediction on ANN models using incoming data streams.**

   In Chapter 3, we introduced TensAIR, an innovative OL framework facilitating real-time training and prediction on ANN models. TensAIR's groundbreaking asynchronous and decentralized dataflow architecture, built on AIR, enables seamless processing of incoming data streams in real-time. The system's lightweight design and impressive scalability allow its sustainable throughput to increase nearly linearly with the deployment of additional computational nodes.

   Notably, TensAIR's versatility shines through its general dataflow system, designed to accommodate various use-cases independently of the chosen network type or architecture (e.g., CNN, LSTM, Transformers, RNN, DNN). While TensAIR empowers users to implement diverse use-cases, it places the responsibility of pre-processing tasks on the user due to the use-case-dependent nature of data stream pre-processing. Although this decision offers flexibility, allowing users to tailor pre-processing based on their needs, it requires users to determine the necessary pre-processing steps, a limitation discussed in Section 7.2.

The continuous training nature of TensAIR, outlined in Chapter 3, positions it well for detecting concept drifts independently of their characteristics. However, to optimize computational resource usage, Chapter 6 provided guidance on incorporating a concept drift detector into the TensAIR dataflow, ensuring that the ANN model will be updated only after a concept drift is detected.

Further illustrating TensAIR's real-world applicability, we explore its use in the *Sentiment Analysis of COVID19* use-case (cf. Chapter 3.6), showcasing its ability to track sentiment evolution in real-time tweets. Further, we demonstrate the straightforward process of employing TensAIR's Python interface for online adaptation of a CNN model. This modification is crucial for addressing concept drifts, which are effectively identified using our OPTWIN method.

The discussed enhancements aim to provide a clearer and more comprehensive overview of the contributions and capabilities of TensAIR in achieving the outlined objectives.

2. **Investigate distribution strategies to scale out the training of ANN models, optimizing their performance by leveraging the asynchronous and decentralized characteristics of our proposed dataflow architecture.**

   Our exploration of distribution strategies for scaling out the training of ANN models is detailed in Chapter 2. The initial analysis focused on the asynchronous and decentralized distribution of Stochastic Gradient Descent (SGD). Notably, our observations revealed that the impact of asynchronicity on model convergence is minimal when gradients reach their destination with low delay. This preliminary study served as a crucial step in identifying potential bottlenecks in our proposed approach.

   An important insight emerged during this analysis, particularly regarding the gradient application process. Despite being less time-intensive, this step can become a bottleneck if not managed effectively. For instance, distributing 1,000 gradient calculation steps across multiple models highlights a potential challenge. In the scenario with two distributed models, each computes an average of 500 gradients while applying all 1,000 gradients. Conversely, with 10 distributed models, each calculates an average of 100 gradients while applying the same 1,000 gradients. The linear growth in the time spent on the gradient application process, independent of the number of distributed models, became more apparent in Chapter 3, especially when leveraging GPU acceleration. This emphasized the significance of the ratio between the time taken to calculate and apply gradients in determining the potential speedup with an increasing number of distributed nodes.

   Building on these insights, we proposed the TensAIR training procedure in Algorithm 1. This procedure harnesses the asynchronous and decentralized setting while addressing the challenge of applying an excessive number of gradients in each local model. The solution involves locally summing a specified number of gradients (*maxBuffer*) before broadcasting them to other models. By setting $maxBuffer = n$ (where n is the number of distributed models), with each model calculating and applying an equal number of gradients. While this introduces a delay in the gradients exchange and may result in higher staleness, Chapter 5 demonstrated that the impact on the convergence time is outweighed by the benefits of reducing the number of applied gradients.

3. **Develop an error-based drift detector algorithm specifically designed for regression and classification problems, aiming to achieve a smaller false positive rate than existing baselines while maintaining an equivalent level of recall.**

   The active adaptation to concept drifts can reduce drastically the amount of computational resources spent unnecessarily retraining an ANN model that did not suffer from concept drift since the last time they were trained. Moreover, by identifying that a concept drift occurred, one may adapt the retraining of the model accordingly. For instance, one could increase the learning rate when concept drifts are identified.

   In Chapter 4, we introduced OPTWIN, an error-base drift detector that produces low amount of false-positives while maintaining excellent recall. Different from other drift detectors, OPTWIN assumes that concept drifts occur when not just the mean but also the standard deviation of the errors of a ML algorithm statistically change. Based on this assumption, we could additionally detect the optimal splitting point of the sliding window of errors used by OPTWIN. In our experiments, we could observe how OPTWIN achieves similar recall as the best drift detectors analyzed while achieving increased precision.

   We note that, despite identifying if a concept drift occurred at every new error received from the ML algorithm in $\mathcal{O}(1)$ (improving the state-of-the-art $\mathcal{O}(\log |W|)$), OPTWIN has a higher running time per iteration than the baselines. This occurs simply because OPTWIN performs both the t and the f-test at every iteration, which despite not computationally expensive, involves more computation than the baselines. Nevertheless, OPTWIN still is a lightweight algorithm, which took in our experiments approximately $1e^{-5}$ seconds per iteration. Therefore, considering the time taken to train or predict on ANN models, this running time is negligible. Moreover, due to the lower number of false-positive concept drifts flagged by OPTWIN, less unnecessary re-training of the ANN model will be needed. Therefore, even on a small ANN model like the one presented in Chapter 4.6.3, by using OPTWIN, it was possible to save 21% of time. Consequently, when analyzing errors from more complex models, which take longer to compute, we expect OPTWIN to save a higher percentage of the time taken when using other concept drift detectors.

   When compared to other concept drift detectors, we consider OPTWIN to be easy to be implemented. This is due to the fact that it relies only on two variables that should be defined by the user: δ, the confidence level of the algorithm; and ρ, its robustness. The definition of ρ is perhaps one of the main insights that enabled the successful implementation of OPTWIN. By defining ρ as the ratio by which the mean of the most recent error rates has to vary in terms of the standard deviation of the older errors, we can simplify the equation used to define the optimal splitting point of the sliding window, and remove its dependency on all other variables related to the data distribution. Moreover, as discussed in Chapter 4.6, in practice, the variation of ρ did not drastically impacted its performance. In any case, for most usecases we recommend a confidence level $\delta = 0.99$ and robustness $\rho = 0.5$.

4. **Formulate a rigorous mathematical proof establishing the convergence rate of decentralized and asynchronous Stochastic Gradient Descent (SGD), contributing to a theoretical understanding of their convergence properties.**

In Chapter 5, we delved into the convergence rate of decentralized and asynchronous Stochastic Gradient Descent (DASGD), presenting a comprehensive proof demonstrating its convergence properties. Specifically, our analysis establishes that DASGD converges in $\mathcal{O}(\frac{\sigma}{\epsilon^2}) + \mathcal{O}(\frac{QS_{avg}}{\epsilon^{\frac{3}{2}}}) + \mathcal{O}(\frac{S_{avg}}{\epsilon})$ iterations. This convergence is contingent on bounded gradients (Q), variance of gradients (σ), and average staleness of gradients (S_{avg}). The novel definition of staleness, representing the symmetric difference between sets of gradients applied by different models, proves to be a key innovation in this context.

Our proof generalizes existing results from (Koloskova, Stich, and Jaggi, 2022) by incorporating the concept of staleness, showcasing its versatility. We establish the tightness of our proof, demonstrating that under scenarios with no staleness, the bounds align with standard SGD. Notably, our bounds are independent of the maximum staleness during optimization, a departure from other asynchronous SGD proofs. This reliance solely on average staleness (S_{avg}) enhances the robustness of our approach, particularly in heterogeneous settings with varying computing power among distributed nodes and unreliable networks susceptible to communication delays.

Furthermore, we additionally extend our analysis to the case where gradients are not bounded by Q, providing a more general convergence rate. In this scenario, DASGD is expected to converge in $\mathcal{O}(\frac{\sigma^2}{\epsilon^2}) + \mathcal{O}(\frac{\sqrt{\hat{S}_{avg}\hat{S}_{max}}}{\epsilon})$ iterations, where $\hat{S}_{max}$ and $\hat{S}_{avg}$ represent less tight versions of the maximum and average staleness observed during optimization. This proof, while less tight, eliminates the assumption of bounded gradients, enhancing its applicability.

Notably, DASGD is proven to converge under minimal network constraints, requiring only that a gradient sent by a model eventually reaches all other models in the network. Although developed with fully connected networks in mind, DASGD's convergence properties are expected to hold in various practical scenarios, making it suitable for Online Learning (OL) use-cases.

Finally, we introduce DASGD2, an extension that addresses network capacity constraints. By introducing a broadcast frequency parameter (bf), DASGD2 allows models to broadcast gradients after every bf iterations, reducing both the time taken by local models to apply gradients and network usage by bf times. While this modification increases expected staleness during optimization, our experiments, such as optimizing the VGG-16 network, showcase its favorable scalability properties.

7.2 Limitations

In this section, we identify and elucidate the primary limitations encountered in this doctoral thesis:

- **Limited Investigation of Use-Case Specific Data Pre-processing Challenges.** One of the main limitations of this thesis is that it does not delve into the specific challenges and their corresponding solutions for performing real-time data pre-processing. Data pre-processing challenges often vary depending on the nature of the data and the specific use-case being addressed. However, the thesis aims to propose a general solution applicable across a wide range of use-cases that require online training of ANN models. As a result, the decision was made to focus on solutions that could be universally applied in a use-case independent manner. Consequently, the thesis did not explore the intricacies of data pre-processing challenges that are specific to particular use-cases. For example, commonly employed techniques such as feature scaling and data normalization may not be directly applicable in use-cases with features of unknown magnitude, whereas use-cases involving images with a fixed encoding format should not encounter this issue. While this approach provides versatility, it also underscores the need for further research in addressing data pre-processing challenges in real-time, tailored to specific use-cases.

- **Limitation in Handling Unlabeled Data.** TensAIR is constrained by its inability to train without labeled data, a limitation that poses challenges in real-time learning scenarios where labeled data is either scarce, arrives with significant delay, or is absent altogether. ANN models typically require labeled data for training, and in OL settings, labels may not be readily available. This limitation hinders the applicability of TensAIR to scenarios that demand the training of ANNs with annotated data. TensAIR was primarily designed to address self-supervised problems, where data labels can be automatically derived from the data itself, eliminating the need for manual annotation. While this design choice aligns with the OL context, it means that TensAIR may not be suitable for training ANN models that rely on annotated data for their training. However, in cases where training labels eventually become available after a delay, TensAIR can still be employed. Nevertheless, this workaround may not be ideal, as it involves postponing the training of ANN models until labels are accessible, which may not align with real-time learning objectives.

- **Limitation in Adapting to Frequent Concept Drifts with Large ANN Models.** In practical scenarios where concept drifts occur frequently, TensAIR may face limitations in adapting large ANN models in real-time. ANN models, especially those with a large number of parameters, typically require a substantial number of training steps to converge effectively. As the size of the ANN model increases, more training steps are generally needed to achieve convergence. Consequently, a higher volume of training data and computational resources are necessary to facilitate this convergence.

  In today's landscape, some ANN models comprise billions of parameters and can take months to complete their training process (as evidenced in recent research (OpenAI, 2023)). Given the time and computational resources required for such large models, TensAIR's ability to train them in real-time and adapt them promptly to frequent concept drifts may be constrained. In scenarios characterized by both frequent concept drifts and large ANN models, achieving real-time adaptation may prove challenging, and alternative strategies may need to be considered to address this limitation.

- **Limitation in Handling ANN Models That Do not Fit into Memory.** TensAIR employs a "data parallelism" strategy to distribute the online training of ANN models. This strategy relies on the assumption that copies of the chosen ANN model must fit into memory during training. Consequently, if an ANN model is too large and cannot fit into memory during training, it becomes impractical or impossible to use TensAIR for distributed training.

  It is important to note that if an ANN model is so large that it cannot fit into memory, it may not be well-suited for online training in the first place. As highlighted in the limitation above, such large models often require numerous iterations to converge, which can translate to extended training times, contrary to the time-sensitive nature of many online learning problems. Therefore, this limitation emphasizes the need to consider the practical constraints of model size when applying TensAIR for online training.

- **User Responsibility in Selecting ANN Model Adaptation Strategy.** A limitation in TensAIR is that it leaves the responsibility of deciding the adaptation strategy for ANN models after a concept drift is detected to the user. There exist various options for adapting ANN models when concept drift occurs, including fine-tuning the model with recent data, training a new model, fine-tuning a pre-trained model, or adjusting the learning rate for faster adaptation. However, there is no one-size-fits-all solution, and the most appropriate approach depends on the specific use-case being implemented.

  For instance, in scenarios involving the training of large CNN models like VGG-16, it may not be advisable to train an entirely new model from scratch after every concept drift. Instead, it might be more efficient to fine-tune a pre-trained model. Conversely, with smaller DNN models that require fewer training steps for convergence, retraining a new model after each concept drift might be a feasible approach.

  While this flexibility allows users to tailor the adaptation strategy to their specific needs, it also places a decision-making burden on them, requiring careful consideration of the trade-offs between different strategies. Therefore, TensAIR's limitation lies in the absence of an automated mechanism for selecting the most suitable adaptation strategy, leaving it up to the user's discretion.

- **Limited Baseline Comparisons with Other Real-Time ANN Training Systems.** One of the limitations of this thesis is the absence of direct comparisons with other systems that enable the real-time training of ANN models. In scientific research, it is customary to evaluate a proposed method against baseline systems developed to address the same problem, highlighting the improvements achieved. However, as of the knowledge cutoff date, TensAIR represents a pioneering system in the realm of training ANN models in real-time from data streams.

  Given the absence of existing baselines specifically designed for real-time ANN training from data streams, the thesis adapts similar methods to perform the same task as TensAIR. While this approach allows for some level of comparison, it is not an optimal solution, as these adapted methods may not be directly comparable to TensAIR. Nevertheless, this limitation is a result of the nascent nature of real-time ANN training systems, emphasizing the novelty and uniqueness of TensAIR in the current state-of-the-art landscape.

- **Complexity in Defining the Right ANN Model Architecture for Online Learning.** Defining the appropriate architecture for ANN models is a well-known challenge in machine learning. However, in the context of OL addressed in this thesis, the task of defining the right ANN model architecture when training in real-time becomes even more complex.

  In conventional machine learning, model architectures are often defined based on the complexity of representing and solving the problem at hand, which is closely related to the characteristics of the training data. In OL settings, ANN models should be capable of representing and solving problems even when the complexity of these problems changes due to shifts in the statistical properties of the training data (a.k.a. concept drifts). This dynamic nature of data in OL settings introduces an additional layer of complexity when defining ANN model architectures.

In OL, developers must consider not only the initial data distribution but also potential changes in data distribution and, consequently, the representation of the problem as it evolves over time. Additionally, given the time-sensitive nature of OL, the choice of smaller models that require less time to converge becomes crucial. These factors underscore the challenges in defining the right ANN model architecture for OL scenarios, requiring careful consideration of adaptability, scalability, and the capacity to handle evolving data distributions.

- **Fixed Network Architecture in TensAIR.** TensAIR employs a fixed network architecture, which means that the architecture is defined beforehand and remains static throughout the training process. In the context of OL where concept drifts can lead to unexpected changes in the statistical properties of data, this limitation poses challenges.

  Concept drifts in OL can involve various changes, including the emergence or disappearance of classes in classification problems. However, TensAIR does not currently address the challenge of adapting the architecture of ANN models in real-time to accommodate these changes. As a result, TensAIR may not be able to consider classes that were not originally defined in the model architecture. Furthermore, it does not have the capability to dynamically adjust its hidden layers, depth, or length to better adapt to concept drifts.

  While the fixed architecture serves the scope of the existing research, it is worth noting that real-time adaptation of ANN model architectures represents a promising extension. Enabling TensAIR to adapt its architecture during training could enhance its functionality and applicability, particularly in OL scenarios characterized by evolving data distributions and changing problem complexities.

- **Fixed Stepsize in DASGD.** DASGD's convergence is guaranteed when using a fixed stepsize. In practice, this contrasts with other SGD-based optimizers, which often achieve faster convergence through the implementation of stepsize schedulers that progressively reduce the stepsize. While it is conceivable that future extensions of DASGD's could accommodate diminishing stepsizes, the current model presents this limitation.

  Moreover, unlike other ASGD algorithms, DASGD's architecture requires uniform application of the same gradient and stepsize across all distributed models. This design choice makes it impossible to adjust the stepsize in response to gradient staleness, a technique commonly employed in other models to mitigate the adverse effects of delayed gradients.

7.3 Synthesis of Findings and Recommendations

In the course of this PhD thesis, several key findings and insights have emerged, shedding light on the realm of real-time training of Artificial Neural Networks (ANN) in Online Learning (OL) settings. These findings and the resulting recommendations can help guide future research and applications in this field.

7.3.1 Findings

DASGD for Accelerating ANN Training: The research demonstrates that Decentralized and Asynchronous Stochastic Gradient Descent (DASGD) is a highly effective solution for accelerating the training of ANN models. This finding underscores the significance of DASGD as a viable approach for enhancing the efficiency of online training ANN models.

Computation vs. Network Bottleneck: In OL scenarios with relatively few computational nodes (e.g., fewer than 64 CPU ranks or 16 GPUs), the research suggests that the bottleneck in real-time ANN training typically lies in the computation phase, rather than in network-related issues, especially when utilizing high-performance network technologies like InfinityBand.

Optimal GPU Utilization: The research indicates that using more than 16 GPUs in OL settings may not be beneficial for most use-cases. This limitation is attributed to the fact that the amount of data generated in real-time may not require the computational power provided by a large number of GPUs. For example, in our experiments, utilizing 4 GPUs allowed us to process image data at a rate of 585 MB/s. This throughput is remarkably 11 times greater than that typically required by 4K cameras, which usually generate data at rates below 50 MB/s.

Importance of Standard Deviation for Concept-Drift Detection: A significant insight is the importance of considering the standard deviation of errors as an indicator of concept drift. This insight is particularly valuable in scenarios where concept drifts can cause a notable impact on the standard deviation of the data distribution, even before affecting the mean. This early detection can be critical for timely adaptation and indirect reduction of false-positives.

Use-Case Specific Challenges: The research recognizes that the OL setting is characterized by numerous use-case-specific challenges that cannot be comprehensively covered in a single study. These challenges often demand individual investigation and tailored solutions based on the specific problem domain.

Self-Supervised Online Training: TensAIR's primary utility lies in online training self-supervised problems, where labels can be defined automatically from the data stream. Additionally, the thesis outlines a strategy, as demonstrated in the Sentiment Analysis use-case (Chapter 3.6.4), where real-time adaptation to changing data distributions can be achieved without the need for pre-defined labels. This approach has the potential to extend to other use-cases, such as audio and video data streams, where real-time embeddings can be used to adapt to data distribution changes.

7.3.2 Recommendations

In light of the findings described above, several recommendations emerge:

- Researchers and practitioners in the field of online training ANN models should consider DASGD as a valuable method for accelerating training processes.
- When designing OL systems, it is crucial to assess whether the computational or network resources are the primary bottleneck and allocate resources accordingly.
- Careful consideration should be given to the number of GPUs used in OL settings, as excessive computational power may not yield substantial benefits.

- Concept-drift detection methods should include the monitoring of standard deviation, particularly in scenarios where it can signal concept drift earlier than changes in the mean.
- Future research in online training ANN models should address specific use-case challenges individually, tailoring solutions to unique problem domains.
- The potential of self-supervised online training approaches, as demonstrated in the Sentiment Analysis use-case, should be explored further in various OL scenarios where labels can be defined in real-time.

These findings and recommendations provide a foundation for further advancements in the field of real-time ANN training within the context of Online Learning.

Chapter 8

Conclusion

8.1 Summary of Contributions

In this doctoral thesis, several significant contributions have been made to the study of training Artificial Neural Network (ANN) models in real-time within the context of Online Learning (OL). The following is a summary of the main contributions:

- **TensAIR Dataflow**: This research introduced the TensAIR dataflow, a novel approach for real-time ANN model training in OL settings. TensAIR enables the training of ANN models using data streams asynchronously and in a decentralized manner. Notably, TensAIR exhibited impressive scalability, achieving up to 116 times higher sustainable throughput compared to state-of-the-art approaches for online training of ANN models.
- **OPTWIN Drift Detector**: The development of OPTWIN, a sliding window drift detection algorithm, represents another key contribution. Unlike existing drift detectors, OPTWIN considers concept drifts to occur when either the mean or the standard deviation of errors statistically changes in a ML algorithm. OPTWIN offers advantages such as the ability to detect concept drifts in both classification and regression problems, as well as lower false positives while maintaining comparable recall. OPTWIN is particularly suitable for identifying concept drifts using the loss of ANN models.
- **DASGD Algorithm and Convergence Proof**: This research introduced the decentralized and asynchronous Stochastic Gradient Descent (DASGD) algorithm. DASGD facilitates direct gradient update exchanges among distributed models without synchronization requirements. A significant contribution lies in the proof of convergence, demonstrating that DASGD converges to an ϵ-small error in a bounded number of steps under mild assumptions. The convergence rate is defined as $\mathcal{O}(\frac{\sigma}{\epsilon^2}) + \mathcal{O}(\frac{QS_{avg}}{\epsilon^{3/2}}) + \mathcal{O}(\frac{S_{avg}}{\epsilon})$, where gradient norms are bounded by Q, and S_{avg} represents the average staleness. Additionally, DASGD2, an extension of DASGD, was introduced to address potential bottlenecks by locally aggregating gradients before broadcasting them, thereby mitigating network congestion and reducing the number of gradients applied per distributed model.
- **TensAIR Implementation**: TensAIR, the system developed in this thesis, was implemented using the AIR stream processing engine, providing efficient asynchronous and decentralized communication among nodes. The system's backend was coded in C++ and relied on MPI for communication, ensuring high performance and low latency. To enhance usability, a Python interface for TensAIR was created, enabling users to define, execute, and visualize the training of their TensorFlow models directly in Python.

These contributions collectively advance the field of online training ANN models in OL settings, offering novel approaches, algorithms, and systems that have the potential to significantly impact various domains of machine learning and data analysis.

8.2 Disseminating our findings

- **Colloquium Presentation (Oral and Poster):**
  - Title: Online Learning using Distributed Neural Networks.
  - Colloquium: Doctoral Training Unit (DTU) on Data-driven Computational Modelling and Applications (DRIVEN) Colloquium.
  - Date: May 21, 2021.
  - Location: Esch-sur-Alzette, Luxembourg.
- **Seminar Presentation (Oral):**
  - Title: TensAIR: an asynchronous and decentralized framework to distribute artificial neural networks training.
  - Seminar: Machine Learning Seminar affiliated with the University of Luxembourg.
  - Date: September 15, 2021.
  - Location: online.
- **Conference Publication and Presentation (Oral and Poster):**
  - Title: Convergence time analysis of Asynchronous Distributed Artificial Neural Networks.
  - Conference: 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD).
  - Date: January 8, 2022.
  - Location: online.
  - Publication: *Tosi, Mauro DL, Vinu Ellampallil Venugopal, and Martin Theobald. "Convergence time analysis of Asynchronous Distributed Artificial Neural Networks." 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD). 2022.*
- **Workshop presentation (Oral):**
  - Title: TensAIR: Online Learning from Data Streams via Asynchronous Iterative Routing.
  - Workshop: Stream Reasoning Workshop.
  - Date: December 05, 2022.
  - Location: Amsterdam, Netherlands.
- **Colloquium Presentation (Oral):**
  - Title: Online Learning using Distributed Neural Networks.
  - Colloquium: DTU Driven Colloquium.

  - Date: June 21, 2023.
  - Location: Kanzem, Germany.

- **Workshop Presentation (Oral and Poster):**
  - Title: Online Learning using Distributed Neural Networks.
  - Workshop: The Dutch-Belgian DataBase Day 2023.
  - Date: December 21, 2023.
  - Location: Ghent, Belgium.

- **Conference Publication and Presentation (Oral) - Best presentation award:**
  - Title: TensAIR: Real-Time Training of Neural Networks from Data-streams.
  - Conference: The 8th International Conference on Machine Learning and Soft Computing (ICMLSC 2023).
  - Date: January 27, 2024.
  - Location: Singapore.
  - Publication: *Tosi, Mauro DL, Vinu Ellampallil Venugopal, and Martin Theobald. "TensAIR: Real-Time Training of Neural Networks from Data-streams." Proceedings of the 2024 8th International Conference on Machine Learning and Soft Computing. 2024.*

- **Conference Publication:**
  - Title: Convergence Analysis of Decentralized ASGD.
  - Conference: Submitted to international conference.

- **Workshop Publication:**
  - Title: OPTWIN: Drift Identification with Optimal Sub-Windows.
  - Workshop: Submitted to workshop in an international conference.

- **Workshop Publication:**
  - Title: TensAIR: Real-Time Training of Distributed Artificial Neural Networks.
  - Workshop: Submitted to demonstration track of a workshop in an international conference.

8.3 Future Work

In this section, we outline potential avenues for extending our research, envisioning opportunities to build upon the contributions presented in this thesis.

- **Investigate Online Data Pre-processing:** Exploring the impact of online data pre-processing on various use-cases would provide valuable insights and guidelines for selecting appropriate data pre-processing techniques. This research could help users make informed decisions about data pre-processing strategies based on their specific use-cases.

- **Extend DASGD Convergence Proof to Mini-Batch SGD:** Extending the current DASGD convergence rate proof to consider mini-batch Stochastic Gradient Descent (SGD) would bridge the gap between the theoretical analysis and practical implementation of the algorithm. This extension could provide insights into how the mini-batch size affects DASGD's convergence properties.

- **Extend DASGD Convergence Proof to Heterogeneous Functions:** Considering the optimization of heterogeneous functions within the context of DASGD would make the algorithm applicable to a broader range of machine learning scenarios. Addressing this extension could benefit settings like Federated Learning, where functions across distributed nodes may vary significantly.

- **Develop Additional Use-cases for TensAIR:** Exploring and implementing new use-cases for TensAIR, especially those involving complex data types like audio and video, would demonstrate its versatility and applicability across different domains. This expansion could open up opportunities for real-time training of ANN models in previously challenging scenarios.

- **Investigate the Use of Remote Direct Memory Access (RDMA):** Exploring how the utilization of RDMA technology can decrease data transfer times from GPUs and improve TensAIR's performance would be a valuable research direction. By optimizing data transfer mechanisms, we can enhance the system's efficiency in real-time ANN model training.

- **Create Container that Supports Distributed GPUs:** Developing a container (eg. Docker (Merkel, 2014), Singularity (Kurtzer, Sochat, and Bauer, 2017), etc.) that supports TensAIR execution on multiple GPU nodes would expand its scalability and performance capabilities. This enhancement could enable users to leverage more powerful hardware resources for real-time ANN model training.

These future research directions reflect the potential for further advancements and applications in the field of online ANN training within Online Learning settings. Exploring these extensions would contribute to the ongoing development of cutting-edge machine learning techniques and systems.

8.4 Final Considerations

In this doctoral thesis, we investigated online training of ANN models in an Online Learning (OL) setting. Our research led to significant advancements across various scientific fields, enhancing the state-of-the-art. We contributed on multiple levels, from detailed implementation aspects to complex theoretical insights. The results obtained are not only promising but also provide a roadmap for tackling this intricate task. Additionally, our work has illuminated several potential future research directions, building upon the findings of this thesis. Our efforts have culminated in three academic articles, one extended abstract, and one demonstration paper, which have been accepted or are under consideration at international conferences and workshops. In conclusion, this research underscores the practicality of real-time ANN model training in OL, fostering further exploration and potential applications in this flourishing field.

Bibliography

Abadi, Martín et al. (2016). "TensorFlow: a system for Large-Scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283.

Abreha, Haftay Gebreslasie, Mohammad Hayajneh, and Mohamed Adel Serhani (2022). "Federated learning in edge computing: a systematic survey". In: *Sensors* 22.2, p. 450.

Agarwal, Alekh and John C Duchi (2011). "Distributed delayed stochastic optimization". In: *Advances in neural information processing systems* 24.

Agrawal, Rakesh, Tomasz Imielinski, and Arun Swami (1993). "Database mining: A performance perspective". In: *IEEE transactions on knowledge and data engineering* 5.6, pp. 914–925.

Akidau, Tyler et al. (2015). "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *Proceedings of the VLDB Endowment* 8.12, pp. 1792–1803.

Arjevani, Yossi, Ohad Shamir, and Nathan Srebro (2020). "A tight convergence analysis for stochastic gradient descent with delayed updates". In: *Algorithmic Learning Theory*, pp. 111–132.

Assran, Mahmoud et al. (2019). "Stochastic gradient push for distributed deep learning". In: *International Conference on Machine Learning*. PMLR, pp. 344–353.

Assran, Mahmoud S and Michael G Rabbat (2020). "Asynchronous gradient push". In: *IEEE Transactions on Automatic Control* 66.1, pp. 168–183.

Baena-Garcıa, Manuel et al. (2006). "Early drift detection method". In: *Fourth international workshop on knowledge discovery from data streams*. Vol. 6, pp. 77–86.

Barros, Roberto SM et al. (2017). "RDDM: Reactive drift detection method". In: *Expert Systems with Applications* 90, pp. 344–355.

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

Basterrech, Sebastián et al. (2023). "A Continual Learning System with Self Domain Shift Adaptation for Fake News Detection". In: *2023 IEEE 10th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, pp. 1–10.

Bayram, Firas, Bestoun S Ahmed, and Andreas Kassler (2022). "From concept drift to model degradation: An overview on performance-aware drift detectors". In: *Knowledge-Based Systems*, p. 108632.

Ben-Nun, Tal and Torsten Hoefler (2019). "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis". In: *ACM Computing Surveys (CSUR)* 52.4, pp. 1–43.

Bifet, Albert and Ricard Gavalda (2007). "Learning from time-changing data with adaptive windowing". In: *Proceedings of the 2007 SIAM international conference on data mining*. SIAM, pp. 443–448.

Bifet, Albert et al. (2009). "New ensemble methods for evolving data streams". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 139–148.

Bifet, Albert et al. (2010). "MOA: Massive Online Analysis". In: *J. Mach. Learn. Res.* 11, pp. 1601–1604. DOI: 10.5555/1756006.1859903. URL: https://dl.acm.org/doi/10.5555/1756006.1859903.

Blackard, Jock A and Denis J Dean (1999). "Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables". In: *Computers and electronics in agriculture* 24.3, pp. 131–151.

Bornstein, Marco et al. (2023). "SWIFT: Rapid Decentralized Federated Learning via Wait-Free Model Communication". In: *The Eleventh International Conference on Learning Representations*. URL: https://openreview.net/forum?id=jh1nCir1R3d.

Breiman, Leo (2001). "Random forests". In: *Machine learning* 45, pp. 5–32.

Brown, Tom et al. (2020). "Language models are few-shot learners". In: *Advances in neural information processing systems* 33, pp. 1877–1901.

Carbone, Paris et al. (2015). "Apache Flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4.

Chai, Junyi et al. (2021). "Deep learning in computer vision: A critical review of emerging techniques and application scenarios". In: *Machine Learning with Applications* 6, p. 100134.

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, pp. 532–540.

Chen, Xiangning et al. (2023). "Symbolic discovery of optimization algorithms". In: *arXiv preprint arXiv:2302.06675*.

Chen, Xinyun, Yunan Liu, and Guiyu Hong (2023). "An online learning approach to dynamic pricing and capacity sizing in service systems". In: *Operations Research*.

Cheng, Yu et al. (2018). "Model compression and acceleration for deep neural networks: The principles, progress, and challenges". In: *IEEE Signal Processing Magazine* 35.1, pp. 126–136.

Cohen, Alon et al. (2021). "Asynchronous stochastic optimization robust to arbitrary delays". In: *Advances in Neural Information Processing Systems* 34, pp. 9024–9035.

Convolutional Neural Network (CNN): Tensorflow Core (2022). Accessed on May 10, 2022. URL: https://www.tensorflow.org/tutorials/images/cnn.

Cottrell, Marie et al. (2012). "Neural networks for complex data". In: *KI-Künstliche Intelligenz* 26, pp. 373–380.

Dai, Wei et al. (2015). "High-performance distributed ML at scale through parameter server consistency models". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29.

Dal Pozzolo, Andrea et al. (2015). "Credit card fraud detection and concept-drift adaptation with delayed supervised information". In: *2015 international joint conference on Neural networks (IJCNN)*. IEEE, pp. 1–8.

Danilova, Marina et al. (2022). "Recent theoretical advances in non-convex optimization". In: *High-Dimensional Optimization and Probability: With a View Towards Data Science*. Springer, pp. 79–163.

Dasu, Tamraparni et al. (2006). "An information-theoretic approach to detecting changes in multi-dimensional data streams". In: *In Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*. Citeseer, pp. 1–24.

Dean, Jeffrey et al. (2012). "Large scale distributed deep networks". In: *Advances in neural information processing systems* 25.

Deep Learning on Flink (2022). Accessed: 2022-08-05. URL: https://github.com/flink-extended/dl-on-flink.

Devlin, Jacob et al. (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: `10.18653/v1/N19-1423`. URL: `https://aclanthology.org/N19-1423`.

Dhariwal, Prafulla and Alexander Nichol (2021). "Diffusion models beat gans on image synthesis". In: *Advances in neural information processing systems* 34, pp. 8780–8794.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7.

Even, Mathieu, Hadrien Hendrikx, and Laurent Massoulié (2021). "Asynchronous speedup in decentralized optimization". In: *arXiv preprint arXiv:2106.03585*.

Fernández Riverola, Florentino et al. (2007). "Applying lazy learning algorithms to tackle concept drift in spam filtering". In.

Feyzmahdavian, Hamid Reza, Arda Aytekin, and Mikael Johansson (2016). "An asynchronous mini-batch algorithm for regularized stochastic optimization". In: *IEEE Transactions on Automatic Control* 61.12, pp. 3740–3754.

Gama, Joao et al. (2004). "Learning with drift detection". In: *Brazilian symposium on artificial intelligence*. Springer, pp. 286–295.

Gama, João et al. (2014). "A survey on concept drift adaptation". In: *ACM computing surveys (CSUR)* 46.4, pp. 1–37.

Go, Alec, Richa Bhayani, and Lei Huang (2009). "Twitter sentiment classification using distant supervision". In: *CS224N project report, Stanford* 1.12.

Gomes, Heitor Murilo et al. (2019). "Machine learning for streaming data: state of the art, challenges, and opportunities". In: *ACM SIGKDD Explorations Newsletter* 21.2, pp. 6–22.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Goodfellow, Ian et al. (2014). "Generative adversarial nets". In: *Advances in neural information processing systems* 27.

Hadsell, Raia et al. (2020). "Embracing change: Continual learning in deep neural networks". In: *Trends in cognitive sciences* 24.12, pp. 1028–1040.

Hariri, Reihaneh H, Erik M Fredericks, and Kate M Bowers (2019). "Uncertainty in big data analytics: survey, opportunities, and challenges". In: *Journal of Big Data* 6.1, pp. 1–16.

Harries, M (1999). "Splice-2 comparative evaluation: electricity pricing (Technical Report UNSW-CSE-TR-9905)". In: *Artificial Intelligence Group, School of Computer Science and Engineering, The University of New South Wales, Sydney* 2052.

Hassabis, Demis et al. (2017). "Neuroscience-inspired artificial intelligence". In: *Neuron* 95.2, pp. 245–258.

Hegedűs, István, Gábor Danner, and Márk Jelasity (2021). "Decentralized learning works: An empirical comparison of gossip learning and federated learning". In: *Journal of Parallel and Distributed Computing* 148, pp. 109–124.

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

Hidalgo, Juan Isidro González, Laura Maria Palomino Mariño, and Roberto Souto Maior de Barros (2019). "Cosine similarity drift detector". In: *Artificial Neural Networks and Machine Learning–ICANN 2019: Text and Time Series: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings, Part IV*. Springer, pp. 669–685.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Hoi, Steven CH et al. (2021). "Online learning: A comprehensive survey". In: *Neurocomputing* 459, pp. 249–289.

Horovod with Keras (2019). Accessed: 2022-05-18. URL: https://horovod.readthedocs.io/en/stable/keras.html.

Hu, Hanqing, Mehmed Kantardzic, and Tegjyot S Sethi (2020). "No Free Lunch Theorem for concept drift detection in streaming data classification: A review". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10.2, e1327.

Hulten, Geoff, Laurie Spencer, and Pedro Domingos (2001). "Mining time-changing data streams". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 97–106.

Ioffe, Sergey and Christian Szegedy (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr, pp. 448–456.

Iwashita, Adriana Sayuri and Joao Paulo Papa (2018). "An overview on concept drift learning". In: *IEEE access* 7, pp. 1532–1547.

Jiang, Jiawei et al. (2017). "Heterogeneity-aware distributed parameter servers". In: *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 463–478.

Jiang, Jiyan et al. (2021). "Asynchronous decentralized online learning". In: *Advances in Neural Information Processing Systems* 34, pp. 20185–20196.

Kang, Jiawen et al. (2020). "Reliable federated learning for mobile networks". In: *IEEE Wireless Communications* 27.2, pp. 72–80.

Karimov, Jeyhun et al. (2018). "Benchmarking distributed stream data processing systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518.

Keskar, Nitish Shirish et al. (2017). "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France*. OpenReview.net.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Kluyver, Thomas et al. (2016). "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *Elpub* 2016, pp. 87–90.

Koloskova, Anastasia, Sebastian Stich, and Martin Jaggi (2019). "Decentralized stochastic optimization and gossip algorithms with compressed communication". In: *International Conference on Machine Learning*, pp. 3478–3487.

Koloskova, Anastasia, Sebastian U Stich, and Martin Jaggi (2022). "Sharper Convergence Guarantees for Asynchronous SGD for Distributed and Federated Learning". In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al.

Koloskova, Anastasia et al. (2020). "A unified theory of decentralized SGD with changing topology and local updates". In: *International Conference on Machine Learning*, pp. 5381–5393.

Koner, Rajat et al. (2023). "Instanceformer: An online video instance segmentation framework". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 1, pp. 1188–1195.

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

Krizhevsky, Alex, Geoffrey Hinton, et al. (2009). *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto, Department of Computer Science.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25.

Kulkarni, Sanjeev et al. (2015). "Twitter heron: Stream processing at scale". In: *Proceedings of the 2015 ACM SIGMOD international conference on Management of data*, pp. 239–250.

Kumar, Siddharth Krishna (2017). "On weight initialization in deep neural networks". In: *arXiv preprint arXiv:1704.08863*.

Kurtzer, Gregory M, Vanessa Sochat, and Michael W Bauer (2017). "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5, e0177459.

Labrèche, Georges et al. (2022). "OPS-SAT Spacecraft Autonomy with TensorFlow Lite, Unsupervised Learning, and Online Machine Learning". In: *2022 IEEE Aerospace Conference (AERO)*. IEEE, pp. 1–17.

Le, Ya and Xuan Yang (2015). "Tiny imagenet visual recognition challenge". In: *CS 231N* 7.7, p. 3.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *nature* 521.7553, pp. 436–444.

LeCun, Yann et al. (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551.

LeCun, Yann et al. (2002). "Efficient backprop". In: *Neural networks: Tricks of the trade*, pp. 9–50.

Li, Guozheng et al. (2023). "Online noisy continual relation learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 11, pp. 13059–13066.

Lian, Xiangru et al. (2015). "Asynchronous parallel stochastic gradient for nonconvex optimization". In: *Advances in Neural Information Processing Systems* 28.

Lian, Xiangru et al. (2017). "Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent". In: *Advances in neural information processing systems* 30.

Lian, Xiangru et al. (2018). "Asynchronous decentralized parallel stochastic gradient descent". In: *International Conference on Machine Learning*. PMLR, pp. 3043–3052.

Liu, Ji and Stephen J Wright (2015). "Asynchronous stochastic coordinate descent: Parallelism and convergence properties". In: *SIAM Journal on Optimization* 25.1, pp. 351–376.

Liu, Ji et al. (2014). "An asynchronous parallel stochastic coordinate descent algorithm". In: *International Conference on Machine Learning*. PMLR, pp. 469–477.

Liu, Xiao et al. (2021). "Self-supervised learning: Generative or contrastive". In: *IEEE transactions on knowledge and data engineering* 35.1, pp. 857–876.

Liu, Zhuang et al. (2018). "Rethinking the Value of Network Pruning". In: *International Conference on Learning Representations*.

Lu, Jie et al. (2018). "Learning under concept drift: A review". In: *IEEE Transactions on Knowledge and Data Engineering* 31.12, pp. 2346–2363.

Lu, Ning et al. (2016). "A concept drift-tolerant case-base editing technique". In: *Artificial Intelligence* 230, pp. 108–133.

Luo, Qinyi et al. (2020). "Prague: High-performance heterogeneity-aware asynchronous decentralized training". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 401–416.

Mahbobi, Mohammad and Thomas K Tiemann (2016). *Introductory business statistics with interactive spreadsheets-1st Canadian edition*. Campus Manitoba, pp. 55–63.

Mandal, Debmalya et al. (2023). "Online Reinforcement Learning with Uncertain Episode Lengths". In: *37th AAAI Conference on Artificial Intelligence*. AAAI.

Massey, Adam and Steven J Miller (2006). "Tests of hypotheses using statistics". In: *Mathematics Department, Brown University, Providence, RI* 2912, pp. 1–32.

Mayer, Ruben and Hans-Arno Jacobsen (2020). "Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools". In: *ACM Computing Surveys (CSUR)* 53.1, pp. 1–37.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5, pp. 115–133.

Merkel, Dirk (2014). "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239, p. 2.

Message Passing Interface Forum (June 2021). *MPI: A Message-Passing Interface Standard Version 4.0*. URL: `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

Miao, Xupeng et al. (2021). "Heterogeneity-aware distributed machine learning training via partial reduce". In: *Proceedings of the 2021 International Conference on Management of Data*, pp. 2262–2270.

Mishchenko, Konstantin et al. (2022). "Asynchronous SGD beats minibatch SGD under arbitrary delays". In: *Advances in Neural Information Processing Systems* 35, pp. 420–433.

Montiel, Jacob et al. (2021). "River: machine learning for streaming data in python". In: *The Journal of Machine Learning Research* 22.1, pp. 4945–4952.

Nadiradze, Giorgi et al. (2021). "Asynchronous decentralized SGD with quantized and local updates". In: *Advances in Neural Information Processing Systems* 34, pp. 6829–6842.

Najafabadi, Maryam M et al. (2015). "Deep learning applications and challenges in big data analytics". In: *Journal of big data* 2.1, pp. 1–21.

Nguyen, Dinh C et al. (2021a). "Federated learning for internet of things: A comprehensive survey". In: *IEEE Communications Surveys & Tutorials* 23.3, pp. 1622–1658.

Nguyen, John et al. (2022). "Federated learning with buffered asynchronous aggregation". In: *International Conference on Artificial Intelligence and Statistics*, pp. 3581–3607.

Nguyen, Quang Hung et al. (2021b). "Influence of data splitting on performance of machine learning models in prediction of shear strength of soil". In: *Mathematical Problems in Engineering* 2021, pp. 1–15.

Nishida, Kyosuke and Koichiro Yamauchi (2007). "Detecting concept drift using statistical testing". In: *International conference on discovery science*. Springer, pp. 264–269.

Noghabi, Shadi A et al. (2017). "Samza: stateful scalable stream processing at LinkedIn". In: *Proceedings of the VLDB Endowment* 10.12, pp. 1634–1645.

Oord, Aaron van den et al. (2016). "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499*.

OpenAI (2023). *GPT-4 Technical Report*. arXiv: `2303.08774 [cs.CL]`.

Ouyang, Shuo et al. (2021). “Communication optimization strategies for distributed deep neural network training: A survey”. In: *Journal of Parallel and Distributed Computing* 149, pp. 52–65.

Oza, Nikunj C and Stuart J Russell (2001). “Online bagging and boosting”. In: *International Workshop on Artificial Intelligence and Statistics*. PMLR, pp. 229–236.

Paladini, Tommaso et al. (2023). “Advancing Fraud Detection Systems Through Online Learning”. In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, pp. 275–292.

Pang, Guansong et al. (2021). “Deep learning for anomaly detection: A review”. In: *ACM computing surveys (CSUR)* 54.2, pp. 1–38.

Paszke, Adam et al. (2019). “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pelossof, Raphael et al. (2009). “Online coordinate boosting”. In: *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. IEEE, pp. 1354–1361.

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

Ram, S Sundhar, A Nedić, and Venugopal V Veeravalli (2009). “Asynchronous gossip algorithms for stochastic optimization”. In: *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*. IEEE, pp. 3581–3586.

Ram, S Sundhar, Angelia Nedić, and Venu V Veeravalli (2010). “Asynchronous gossip algorithm for stochastic optimization: Constant stepsize analysis”. In: *Recent Advances in Optimization and its Applications in Engineering: The 14th Belgian-French-German Conference on Optimization*, pp. 51–60.

Recht, Benjamin et al. (2011). “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent”. In: *Advances in neural information processing systems* 24.

Rettig, Laura et al. (2019). “Online anomaly detection over big data streams”. In: *Applied Data Science: Lessons Learned for the Data-Driven Business*, pp. 289–312.

Robbins, Herbert and Sutton Monro (1951). “A stochastic approximation method”. In: *The annals of mathematical statistics*, pp. 400–407.

Robust machine learning on streaming data using Kafka and Tensorflow-IO (2022). Accessed: 2022-08-05. URL: https://www.tensorflow.org/io/tutorials/kafka.

Rombach, Robin et al. (2021). *High-Resolution Image Synthesis with Latent Diffusion Models*. arXiv: 2112.10752 [cs.CV].

Ross, Gordon J et al. (2012). “Exponentially weighted moving average charts for detecting concept drift”. In: *Pattern recognition letters* 33.2, pp. 191–198.

Ruder, Sebastian (2016). “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747*.

Ruthotto, Lars and Eldad Haber (2021). “An introduction to deep generative modeling”. In: *GAMM-Mitteilungen* 44.2, e202100008.

Ruxton, Graeme D (2006). “The unequal variance t-test is an underused alternative to Student’s t-test and the Mann–Whitney U test”. In: *Behavioral Ecology* 17.4, pp. 688–690.

Schäfl, Bernhard et al. (2022). “Hopular: Modern Hopfield Networks for Tabular Data”. In: *arXiv preprint arXiv:2206.00664*.

Schlimmer, Jeffrey C and Richard H Granger (1986). "Incremental learning from noisy data". In: *Machine learning* 1.3, pp. 317–354.
Sergeev, Alexander and Mike Del Balso (2018). "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799*.
Sethi, Tegjyot Singh and Mehmed Kantardzic (2017). "On the reliable detection of concept drift from streaming unlabeled data". In: *Expert Systems with Applications* 82, pp. 77–99.
Shahraki, Amin et al. (2022). "A comparative study on online machine learning techniques for network traffic streams analysis". In: *Computer Networks* 207, p. 108836.
Simonyan, Karen and Andrew Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1409.1556.
Sinha, Harsh, Vinayak Awasthi, and Pawan K Ajmera (2020). "Audio classification using braided convolutional neural networks". In: *IET Signal Processing* 14.7, pp. 448–454.
Sirb, Benjamin and Xiaojing Ye (2016). "Consensus optimization with delayed and stochastic gradients on decentralized networks". In: *2016 IEEE International Conference on Big Data (Big Data)*, pp. 76–85.
Sra, Suvrit et al. (2016). "Adadelay: Delay adaptive distributed stochastic optimization". In: *Artificial Intelligence and Statistics*, pp. 957–965.
Srivastava, Kunal and Angelia Nedic (2011). "Distributed asynchronous constrained stochastic optimization". In: *IEEE journal of selected topics in signal processing* 5.4, pp. 772–790.
Stich, Sebastian U. (2019). "Local SGD Converges Fast and Communicates Little". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=S1g2JnRcFX.
Stich, Sebastian U and Sai Praneeth Karimireddy (2020). "The error-feedback framework: Better rates for SGD with delayed gradients and compressed updates". In: *The Journal of Machine Learning Research* 21.1, pp. 9613–9648.
Strubell, Emma, Ananya Ganesh, and Andrew McCallum (2019). "Energy and Policy Considerations for Deep Learning in NLP". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 3645–3650.
Sun, Tao, Robert Hannah, and Wotao Yin (2017). "Asynchronous coordinate descent under more realistic assumptions". In: *Advances in Neural Information Processing Systems* 30.
Tantisripreecha, Tanapon and Nuanwan Soonthomphisaj (2018). "Stock market movement prediction using LDA-online learning model". In: *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, pp. 135–139.
TensorFlow (2022a). *Convolutional Neural Network (CNN); Tensorflow Core*. URL: https://www.tensorflow.org/tutorials/images/cnn.
TensorFlow (2022b). Accessed: 2022-05-27. URL: https://www.tensorflow.org/text/tutorials/text_classification_rnn.
Thompson, Neil C, Shuning Ge, and Gabriel F Manso (2022). "The importance of (exponentially more) computing power". In: *arXiv preprint arXiv:2206.14007*.
Toghani, Mohammad Taha and César A Uribe (2022). "Unbounded Gradients in Federated Learning with Buffered Asynchronous Aggregation". In: *2022 58th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1–8.

Tosi, Mauro DL and Martin Theobald (n.d.). “TensAIR: Real-Time Training of Distributed Artificial Neural Networks”. Under review.

— (2023). “Convergence Analysis of Decentralized ASGD”. In: *arXiv e-prints*, arXiv–2309.

— (2024). “OPTWIN: Drift identification with optimal sub-windows”. In: *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW 2024)*. IEEE, in press.

Tosi, Mauro DL, Vinu E. 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)*, pp. 314–315. DOI: 10.1145/3493700.3493758.

— (2024). “TensAIR: Real-Time Training of Neural Networks from Data-streams”. In: *Proceedings of the 8th international conference on machine learning and soft computing*. DOI: 10.1145/3647750.3647762.

Trella, Anna L et al. (2023). “Reward design for an online reinforcement learning algorithm supporting oral self-care”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 13, pp. 15724–15730.

Varrette, S. et al. (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)*. Fuzhou, China: Association for Computing Machinery (ACM). ISBN: 978-1-4503-9664-6.

Varrette, Sébastien et al. (2014). “Management of an academic HPC cluster: The UL experience”. In: *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, pp. 959–967.

Vaswani, Ashish et al. (2017). “Attention is all you need”. In: *Advances in neural information processing systems* 30.

Venugopal, Vinu E. et al. (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)*, pp. 51–58.

Venugopal, Vinu E. et al. (2022). “Targeting a Light-Weight and Multi-Channel Approach for Distributed Stream Processing”. In: *Journal of Parallel and Distributed Computing*.

Wang, Jian et al. (2022). “A review on extreme learning machine”. In: *Multimedia Tools and Applications* 81.29, pp. 41611–41660.

Wang, Kai et al. (2023). “Optimistic whittle index policy: Online learning for restless bandits”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 8, pp. 10131–10139.

Webb, Geoffrey I. (2010). “Naïve Bayes”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, pp. 713–714. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_576. URL: https://doi.org/10.1007/978-0-387-30164-8_576.

Wu, Ocean et al. (2021). “Nacre: Proactive recurrent concept drift detection in data streams”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, pp. 1–8.

Wu, Tianyu et al. (2017). “Decentralized consensus optimization with asynchrony and delays”. In: *IEEE Transactions on Signal and Information Processing over Networks* 4.2, pp. 293–307.

Wu, Xuyang et al. (2023). “Delay-agnostic Asynchronous Distributed Optimization”. In: *arXiv preprint arXiv:2303.18034*.

Xie, Cong, Sanmi Koyejo, and Indranil Gupta (2019). “Asynchronous federated optimization”. In: *arXiv preprint arXiv:1903.03934*.

Zaharia, Matei et al. (2016). “Apache Spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11, pp. 56–65.

Zhang, Huan, Cho-Jui Hsieh, and Venkatesh Akella (2016). “Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent”. In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, pp. 629–638.

Zhang, Senyue, Wenan Tan, and Yibo Li (2018). “A survey of online sequential extreme learning machine”. In: *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, pp. 45–50.

Zhang, Xin, 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)*, pp. 3580–3585.

Zhang, Yu et al. (2020). “Pushing the limits of semi-supervised learning for automatic speech recognition”. In: *arXiv preprint arXiv:2010.10504*.