# Advanced Adaptive Classifier Methods for Data Streams

**A thesis**

**submitted in fulfilment**

**of the requirements for the Degree**

**of**

**Doctor of Philosophy in Computer Science**

**at**

**The University of Waikato**

by

**Nuwan Amila Gunasekara**

THE UNIVERSITY OF
WAIKATO
*Te Whare Wānanga o Waikato*

**2023**

# Abstract

The exponential growth of the internet has resulted in an overwhelming influx of big data. However, traditional batch learning models face significant obstacles in effectively learning from these vast and constantly evolving data streams and generating up-to-date outcomes. To overcome these limitations, Stream Learning (SL) has emerged as a promising solution that enables continuous learning from evolving data streams and adapts to changes in input distributions.

This thesis focuses on the classification task of SL, specifically investigating streaming gradient-boosted trees and Neural Network (NN)s. Firstly, we introduce Streaming Gradient Boosted Trees (SGBT), a novel gradient-boosted method designed explicitly for SL classification. Next, we propose **C**ontinuously **A**daptive **N**eural Networks for **D**ata Streams (CAND), an architecture agnostic NN approach for evolving data stream classification. Both SGBT and CAND outperform current state-of-the-art bagging and random forest-based SL methods, demonstrating their superiority in handling evolving data stream classification tasks.

Online Continual Learning (OCL) addresses the issue where NN learning from an evolving data stream forgets its past knowledge when confronted with a distribution shift. Online Domain Incremental Continual Learning (ODICL) is a specific variant of OCL where the input data distribution changes from one task to another. We propose two innovative methods: Online Domain Incremental Pool (ODIP) and Online Domain Incremental Networks (ODIN), for ODICL. The proposed methods leverage existing well-researched SL techniques described in Online Streaming Continual Learning (OSCL). ODIP and ODIN outperform current regularization methods without needing a replay buffer. ODIN achieves competitive results compared to replay-based methods. Both methods are ideal candidates for privacy-concerned ODICL scenarios, offering alternatives to regularization-based approaches.

Overall, this thesis explores advancements in SL classification and ODICL, presenting novel techniques that surpass existing approaches in their respective domains. These contributions have significant implications for addressing the challenges posed by evolving data streams in the era of big data.

# Acknowledgements

As numerous individuals have highlighted, the progress of each generation is built upon the achievements and mistakes of those who preceded them, enabling us to learn and grow. In my journey during my PhD, I had the privilege of working alongside remarkable individuals, giants in their fields.

I am sincerely grateful to Prof. Albert Bifet, Prof. Bernhard Pfahringer, and Assistant Prof. Heitor Murilo Gomes for their unwavering guidance and support throughout this transformative experience. Collaborating with such exceptional minds was an absolute pleasure, and I cherished every step of the journey. I sincerely appreciate the remarkable support I received, especially during some challenging times in recent history.

I would also like to express my heartfelt gratitude to my beloved wife, Nadee, whose love, care, and understanding were instrumental in sustaining my perseverance and enabling me to accomplish something of this magnitude. Additionally, I would like to extend my gratitude to my son, Lisara, and my daughter, Meenu, for the joy they bring me each day, keeping me grounded and humble.

To the participants of our engaging lunch chats, Dennis, Jacob, Nick, Yaqian, and Guilherme, I am immensely thankful for your valuable contributions and the stimulating discussions we shared. Furthermore, I would like to thank Prof. Eibe Frank for his insightful perspectives and comments, which proved instrumental in bridging specific gaps in my research. To all those whose support I may have inadvertently omitted, please accept my sincere appreciation for your contributions throughout this journey.

# Contents

# List of Figures

# List of Tables

# Acronyms

**AGI** Artificial General Intelligence. 2

**ARF** Adaptive Random Forest. vii, 5, 7, 15, 17, 32, 36, 40–45, 50, 58, 62, 66–70, 72, 118, 119

**AdIter** Adaptive Iterations. 5, 7, 16, 17, 32–34, 37, 40–42, 45, 118, 119

**Axgb** Adaptive eXtreme Gradient Boosting. 5, 16, 17, 32, 37, 41

**CAND** **C**ontinuously **A**daptive **N**eural Networks for **D**ata Streams. ii, v, vii, x, xi, 7, 10, 57–65, 67–80, 82, 115–117

**CL** Continual Learning. ii, vi, xiii, 3–6, 8, 12, 21–23, 25, 27, 81–83, 89, 96, 102–104, 116

**CNN** Convolutional Neural Network. viii, 82, 84–89, 93, 96–101, 103, 106, 117

**DDM** Drift Detection Method. 14, 36, 50, 52

**DL** Dynamic Learning-Rate. 95, 96, 99–107, 109, 110, 116, 117

**ER** Experience Replay. 8, 23–25, 82, 95, 103–107, 109

**EWC** Elastic Weight Consolidation. 8, 22, 23, 25, 26, 82, 88, 89, 93, 103–105, 107, 116

**FIMT-DD** Fast Incremental Model Trees with Drift Detection. ix–xi, 34, 36, 43, 45, 47–53, 115, 117, 119, 121

**GB** Gradient Boosting. 31

**GDumb** GDumb. 23, 25, 82

**HT** Hoeffding Tree. viii, 15, 17, 30, 34, 42, 43, 45, 66, 84, 87–89, 91, 93, 96, 98, 100, 101, 103–105, 112–114, 117

**IID** Independent and Identically Distributed. 6, 12, 21, 27, 31, 83, 97

**LR** Logistic Regression. viii, 83–88, 92

**LwF** Learning without Forgetting. 8, 22, 23, 25, 26, 82, 88, 89, 93, 103–105, 107, 109, 116

**MIR** Maximally Interfered Retrieval. 8, 23–25, 95, 103, 104, 106, 107, 109

**ML** Machine Learning. 2

**MV** Majority Vote. viii, 86, 88, 89, 93, 97, 99, 103, 105

**NB** Naive Bayes. vii, viii, 15, 84, 87–90, 93, 94, 96, 98, 100, 101, 103–107, 109–112

**NN** Neural Network. ii, 3, 4, 6, 8, 18, 19, 21–25, 29, 57–62, 66, 72, 76, 81, 82, 95, 97, 98, 102, 104, 107, 115–117

**OCICL** Online Class Incremental Continual Learning. 22–24

**OCL** Online Continual Learning. ii, vi, ix, 4–6, 10–12, 21, 22, 24, 26–30, 81, 82

**OC** One Class Classifier. viii, 83–89, 92, 93

**ODICL** Online Domain Incremental Continual Learning. ii, 6, 8, 10, 22–26, 81–83, 85, 89, 95–97, 102, 103, 106, 107, 116, 117

**ODIN** Online Domain Incremental Networks. ii, v, viii, 8, 10, 95–98, 102–107, 109, 110, 116, 117

**ODIP** Online Domain Incremental Pool. ii, v, vii, 8, 10, 81–89, 93, 95, 104–107, 116, 117

**OSCL** Online Streaming Continual Learning. ii, iv, vi, 5, 6, 8, 10–12, 27, 81, 82, 95, 116

**Pht** Page-Hinckley Test. 36

**REMIND** Replay Using Memory Indexing. 24

**SGBT** Streaming Gradient Boosted Trees. ii, iv, vii–xi, 7, 10, 31–56, 115, 117–121

**SGT** Stochastic Gradient Trees. x, 34, 36, 37, 50, 52, 117

**SL** Stream Learning. ii, iv, vi, ix, 3–7, 10–12, 14, 15, 19–21, 27–31, 57, 81, 82, 115

**SRP** Streaming Random Patches. vii, ix, 5, 7, 15, 17, 18, 32, 36, 40–45, 50, 58, 62, 66–70, 72, 118, 119

**SSL** Semi-Supervised Learning. 3, 4

**TD** Task Detection. 82, 83, 85, 87–89, 93, 96, 101–104, 106, 107, 109, 110, 116, 117

**TP** Task Predictor. vii, viii, 8, 82–88, 90–94, 96–104, 106, 107, 110–114, 116, 117

**TR** Tree Replacement. x, 32, 34, 47, 50, 52, 115

**WV** Weighted Voting. 85, 88, 89, 93, 97, 103–107, 109, 110

**XGBoost** eXtreme Gradient Boosting. xii, 5, 15, 16, 31–34

**AdaBoost** AdaBoost. 15, 16, 31

**ADL** Autonomous Deep Learning. v, x, 7, 18, 58, 62, 64, 66, 69, 70

**ADWIN** ADaptive sliding WINdow. 14–17, 60, 66, 67, 82, 84, 85, 87, 95, 96, 98, 100–102, 106

**DJL** Deep Java Library. 68

**LB** Leveraging Bagging. 16, 17

**MLP** Multi-Layer Perceptron. vii, x, 18, 57–62, 64, 65, 69, 70, 72–74

**OB** Online Boosting. 16, 17

**OnlineAdaC2** Online AdaC2. 16

**OnlineBagging** Online Bagging. 16

**OnlineCSB2** Online CSB2. 16

**OnlineRUSBoost** Online RUSBoost. 16

**OnlineSMOTEBagging** Online SMOTEBagging. 16

**OnlineSMOTEBoost** Online SMOTEBoost. 16

**OnlineUnderOverBagging** Online UnderOverBagging. 16

**OSB** Online Smooth Boost. vii, 7, 16, 17, 33, 40–45, 50, 118, 119

**RNN** Recurrent Neural Network. 18

**SGD** Stochastic Gradient Descent. 69

**SmoothBoost** Smooth Boost. 16

**SVM** Support Vector Machine. 18

# Chapter 1

# Introduction and Background

This chapter introduces the dynamic field of evolving data stream classification, encompassing the motivation behind our work and the challenges that arise in this domain. It also emphasizes our contributions in tackling these challenges, which offers a framework for navigating the remainder of the thesis.

## 1.1 Introduction

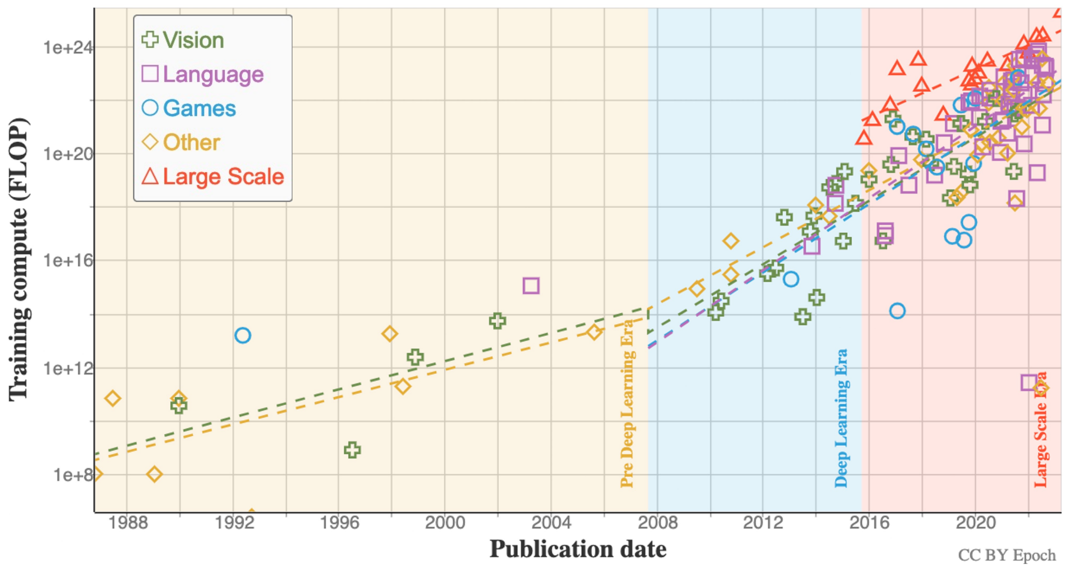


Figure 1.1: Training compute (FLOP) of milestone ML learning systems over time. Source [1].

Over the last decade, the proliferation of the internet has resulted in an unprecedented surge of big data, creating an urgent need for rapid advancements in the field of Machine Learning (ML). As a rapidly evolving field, ML can now extract knowledge and identify patterns from large data sets. It has found widespread applications in various domains, such as healthcare, finance, online marketing, and many others, to solve real-world problems using big data [5, 6].

The progress in computing has facilitated ML systems to keep up with this trend by utilizing larger models [7, 1]. As a result, ML has moved closer to achieving Artificial General Intelligence (AGI). However, this has come at the cost of increased computing power. Figure 1.1 illustrates this trend, with the most recently proposed ML methods requiring a significantly large computing budget.

### 1.1.1 Motivation

Current batch learning methods typically require training models on the entire dataset with multiple epochs, leading to significant energy and computing resource consumption [5]. However, in real-world settings, the underlying data distribution can change over time, causing 'concept drift'[1] to deteriorate the model's performance [8, 9]. As a result, models must be retrained periodically, incorporating data from new distributions to remain relevant. Therefore, there is a pressing need to develop efficient ML algorithms which can evolve.

Online learning methods aim to address this issue, allowing the model to learn from new data as it arrives without access to the entire dataset. These methods enable the model to adapt to new data distributions, making it more robust and capable of handling real-world data with changing patterns. As learning happens online, these models need to be computationally efficient and able to predict at any time.

However, many learning algorithms tend to forget past knowledge when learning from a new data distribution. This phenomenon is well-documented

[1]The idea of 'concept drift' is further explained in section 2.1.1.1.

for popular Neural Network (NN)s and identified as 'catastrophic forgetting' in literature [10, 4]. The naive approach to overcome this is to retrain the model from scratch with old and new data, which is computationally expensive. Thus, Continual Learning without retraining is essential to an online learning agent when learning from an evolving data stream.

### 1.1.2 Learning from evolving data streams

**Stream Learning (SL)** is a machine learning technique that operates on data streams. Stream learning algorithms are designed to handle the dynamic nature of data streams and must continuously adjust to new concepts that emerge over time [9]. Here the main emphasis is to adjust to the current concept efficiently. Similar to batch learning, Stream Learning methods are categorized into supervised, semi-supervised, and clustering [9, 11].

**Supervised SL** assumes that each instance in the dataset has a corresponding target value. There are two main categories of supervised SL: classification and regression. Classification methods aim to predict the category or class of a given instance in the data stream. Section 2.1.1.2 contains an in-depth review of classification methods for supervised SL. On the other hand, regression methods aim to predict a continuous numerical output value for a given instance in the data stream.

**Semi-Supervised SL** relaxes the always-label availability assumption in supervised SL for some instances. For those instances, target values may only become available at a later time or not be available at all. Semi-Supervised SL categorizes label availability into four groups: (i) Immediate and fully labelled, (ii) Delayed and fully labelled, (iii) Immediate and partially labelled, and (iv) Delayed and partially labelled [12]. The majority of data stream Semi-Supervised Learning (SSL) is devoted to understanding (iii). However, [12] highlights the importance of understanding the delayed and partially labelled (iv) setting. Furthermore, the authors categorize streaming SSL methods into: (i) *intrinsically* SSL, (ii) *self-training*, and (iii) *learning by disagreement*. *Intrinsically* SSL methods exploit the unlabelled instances directly as part of their objective function or optimization procedure [12]. *Self-training* methods are based on the idea that a classifier learns from its previous mistakes and then reinforces itself [12]. It can act as a wrapper algorithm that uses any arbitrary classifier. *Learning by disagreement* works by learners teaching other learners. Models are trained with multiple viewpoints of the same data[2], which results in disagreeing models. The key idea behind learning by disagreement is to generate multiple learners and let them collaborate to exploit the unlabelled data [12].

**Clustering Stream Learning** assumes the unavailability of target variables. It can be categorized into: partition clustering, micro-cluster-based clustering, density-based clustering, and hierarchical clustering [11]. Instances from a stream are divided into segments without a class label. The objective of this type of SL is to discover patterns in the stream in an online fashion with a minimum amount of resources. Also, algorithms deployed in this setting should be able to cope with the evolving nature of the stream. The survey [13] contains a recent and extensive study on this field.

**Continual Learning (CL)** aims to preserve already-acquired knowledge while adjusting to new concepts. The main difference between SL and CL is that it contains the additional requirement to preserve old knowledge while adjusting to the new concept. Online Continual Learning (OCL) aims to achieve this dual objective while learning online from an evolving data stream. This allows learning agents to continuously acquire new knowledge without resorting to retraining on the entire data set. The literature identifies three main categories of CL: task-incremental, class-incremental, and domain-incremental [4]. It further identifies three main approaches to avoid catastrophic forgetting in NNs: regularization, replay, and parameter isolation [4]. Section 2.1.2 explains

[2]This could be achieved through techniques such as bootstrapping aggregation.

the above CL settings and current methods to alleviate catastrophic forgetting in detail.

**Online Streaming Continual Learning (OSCL)** is identified as the intersection between SL and OCL in this thesis. It allows well-researched SL fields such as efficient stream learners, concept drift detection, and adaptation strategies to enhance or develop new OCL methods. This emerging field of OSCL is further explored in section 2.1.3. It also compare SL and OCL in depth.

### 1.1.3 Challenges

In Stream Learning, **models must always be available to make predictions** as new data arrives [9]. This requires the model to be updated continuously. Also, the models are expected to **adapt to concept drifts** to ensure the predictions remain accurate and up-to-date [9]. Efficiency is also crucial in SL as the model learns online, and the processing time is limited. Hence, SL **algorithms must be computationally efficient** [9]. OCL adds another requirement for SL models by expecting them to **preserve already acquired knowledge** while adjusting to a new concept [4].

In recent years, bagging-based ensemble learners have emerged as powerful methods for SL, outperforming boosting-based methods. Among these learners, Adaptive Random Forest (ARF) [14] and Streaming Random Patches (SRP)[15] have shown particularly promising results. Even state-of-the-art gradient boosted SL methods like Adaptive eXtreme Gradient Boosting (Axgb) [16] and Adaptive Iterations (AdIter) [17] have failed to surpass the performance of ARF and SRP.

- In this thesis, we propose a gradient-boosted SL classification method that surpasses the current state-of-the-art bagging and random forest-based SL methods.

Neural Network (NN)s have demonstrated remarkable success in processing high-dimensional data sets in batch learning. However, they often require large data sets and multiple epochs of training to learn effectively. Additionally, NNs are vulnerable to hyperparameters, and retraining is necessary to produce an effective model when concept drift occurs. To fully harness the advantages of NNs for SL, it is essential to train and test NNs efficiently without hyperparameter tuning, where predictive NNs are resilient and adaptive to concept drifts.

- Given their significance, this thesis also explores the possibility of utilizing Neural Networks for data stream classification.

When employing NNs for Online Continual Learning, it is crucial to overcome the challenge of catastrophic forgetting [10, 4], where NN forgets its past knowledge when faced with a distribution shift while learning from a non-IID data stream.

- The thesis explores using Online Streaming Continual Learning, the fusion of SL and OCL, to propose novel techniques for Online Domain Incremental Continual Learning. Here, we aim to alleviate catastrophic forgetting of NNs while performing well on current distribution for OD-ICL using SL techniques and methods.

### 1.1.4 Contributions

The thesis presents two novel methods for Stream Learning classification.

- Gradient Boosting is a widely-used machine learning technique highly effective in batch learning. However, its effectiveness in stream learning contexts lags behind bagging-based ensemble methods, which currently dominate the field. One reason for this discrepancy is the challenge of adapting the booster to new concepts following a concept drift. Resetting the entire booster can lead to significant performance degradation as it struggles to learn the new concept. Resetting only some parts of the booster can be more effective, but identifying which parts to reset is difficult, given that each boosting step builds on the previous prediction. To overcome these difficulties, we propose **Streaming Gradient Boosted Trees (SGBT)**, which incorporate trees with a tree replacement strategy to detect drifts and enable the algorithm to adapt without sacrificing performance. An empirical evaluation of SGBT on various streaming datasets with challenging drift scenarios demonstrates that it outperforms current state-of-the-art methods: SRP, AdIter, and Online Smooth Boost (OSB) [18].

- Neural networks have enjoyed tremendous success in many areas over the last decade. They are also receiving more and more attention in learning from data streams, which is inherently incremental. An incremental setting poses challenges for hyperparameter optimization, which is essential for satisfactory network performance. We propose **Continuously Adaptive Neural Networks for Data Streams (CAND)** to overcome this challenge. For every prediction, CAND chooses the current best network from a pool of candidates by continuously monitoring the performance of all candidate networks. The candidates are trained using different optimizers and hyperparameters. An experimental comparison against three state-of-the-art stream learning methods: SRP, ARF and Autonomous Deep Learning (ADL) [19] over various streaming datasets confirms the competitive performance of CAND, especially on high-dimensional data. We also investigate two orthogonal heuristics for accelerating CAND, which trade-off small amounts of accuracy for significant run-time gains. We observe that training on small mini-batches yields similar accuracy to single-instance fully incremental training, even on evolving data streams.

The two novel SL methods: SGBT and CAND, presented in this thesis surpass existing state-of-the-art random forest and bagging-based methods such as ARF and SRP. Some of the ideas from CAND: like the drift in the

NN's loss is analogues to the end-of-the-concept signal, and the estimated loss of NN reveals its current performance, are later used to propose two Online Streaming Continual Learning methods for Online Domain Incremental Continual Learning (ODICL).

- To overcome the challenges (alleviate catastrophic forgetting while performing well on current distribution) of ODICL, we propose **Online Domain Incremental Pool (ODIP)**, a novel method to cope with catastrophic forgetting in ODICL. ODIP also employs automatic concept drift detection and does not require task ids during training. ODIP maintains a pool of learners, freezing and storing the best one after training on each task. An additional Task Predictor (TP) is trained to select the most appropriate NN from the frozen pool for prediction. The empirical evaluation suggests that ODIP outperforms popular regularization methods: Elastic Weight Consolidation (EWC) [10] and Learning without Forgetting (LwF) [20] for ODICL.

- Ideas from ODIP were further extended to propose **Online Domain Incremental Networks (ODIN)**. Compared to ODIP, ODIN only trains a single NN. But, it maintains a pool of NNs, each frozen for further updates at the end of a task. ODIN also employs the same task prediction and detection strategies as ODIP. But it utilizes the incremental or decremental drifts in the loss detected by the drift detector to dynamically increase or decrease the learning rate. With these changes, ODIN surpasses popular regularization methods: EWC [10] and LwF [20] and produces competitive results to replay methods: Experience Replay (ER) [21] and Maximally Interfered Retrieval (MIR) [22] without requiring an instance buffer like in replay methods for ODICL.

Both ODIP and ODIN are able to detect the end of the task signal. As both of them do not require an instance buffer for Online Domain Incremental Continual Learning, they are more suited for privacy-concerned ODICL settings.

### 1.1.5 Publications

- Gunasekara, N., Gomes, H. M., Pfahringer, B., & Bifet, A. (2022, July). Online Hyperparameter Optimization for Streaming Neural Networks. In 2022 International Joint Conference on Neural Networks (IJCNN) (pp. 1-9). IEEE.

- Gunasekara, N., Gomes, H., Bifet, A., & Pfahringer, B. (2022, September). Adaptive Online Domain Incremental Continual Learning. In Artificial Neural Networks and Machine Learning–ICANN 2022: 31st International Conference on Artificial Neural Networks, Bristol, UK, September 6–9, 2022, Proceedings, Part I (pp. 491-502). Cham: Springer International Publishing.

- Gunasekara, N., Gomes, H., Bifet, A., & Pfahringer, B. (2022, November). Adaptive Neural Networks for Online Domain Incremental Continual Learning. In Discovery Science: 25th International Conference, DS 2022, Montpellier, France, October 10–12, 2022, Proceedings (pp. 89-103). Cham: Springer Nature Switzerland.

- Gunasekara, N., Pfahringer, B., Gomes, H. M., & Bifet, A. (2023, August). Survey on Online Streaming Continual Learning. In Proceedings of the Thirty-Second International Joint Conferences on Artificial Intelligence.

- Gunasekara, N., Pfahringer, B., Gomes, H. M., & Bifet, A. (2023, December). Gradient Boosted Trees for Evolving Data Streams. International Conference on Data Mining (under review).

### 1.1.6 Outline

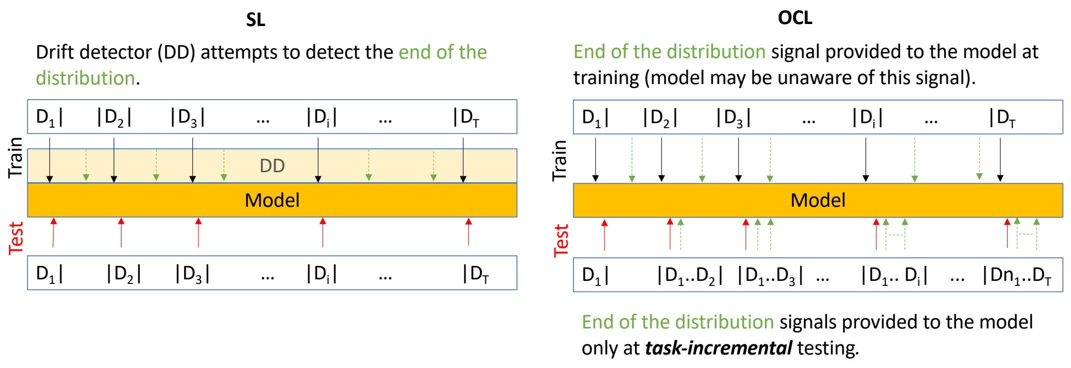
The thesis is composed of seven chapters. The first chapter introduces the basic concept of learning from evolving data streams and provides an overview. Chapter 2 covers preliminary and related work, including Stream Learning, Online Continual Learning, and Online Streaming Continual Learning. Chapters 3 and 4 propose two Stream Learning methods for evolving data streams: SGBT and CAND. Chapters 5 and 6 present two ODICL methods: ODIP and ODIN. Finally, Chapter 7 offers conclusions and future research directions when learning from evolving data streams.

# Chapter 2

# Preliminaries and Related Work

This chapter attempts to understand Online Streaming Continual Learning: the intersection of two closely related fields: SL and OCL, considering their underlying setting, evaluation methods, and applications.

## 2.1 Learning from evolving data streams



SL
Drift detector (DD) attempts to detect the end of the distribution.
D1 | |D2 | |D3 | ... |Di | ... |DT
Train
DD
Model
Test
D1 | |D2 | |D3 | ... |Di | ... |DT
OCL
End of the distribution signal provided to the model at training (model may be unaware of this signal).
D1 | |D2 | |D3 | ... |Di | ... |DT
Train
Model
Test
D1 | |D1..D2 | |D1..D3 | ... |D1.. Di | ... |Dn1..DT
End of the distribution signals provided to the model only at task-incremental testing.

Figure 2.1: Comparison between SL and OCL settings.
SL: uses Drift Detector (DD)s discussed in section 2.1.1.1 to detect distribution shifts, and evaluation is discussed in section 2.1.1.3. OCL: uses evaluation metrics (equations 2.1, 2.2, 2.3, and 2.4) discussed in section 2.1.1.3. The models in this setting do not detect distribution shifts. OSCL proposes some of the techniques used in SL to be used in OCL.

This section provides an overview of supervised classification methods for learning from evolving data streams, covering several key topics and their related work. The section starts by introducing Stream Learning (SL). It then discusses the phenomenon of concept drift and methods to detect concept drifts. The section also discusses several popular supervised SL methods. The evaluation of SL methods, including commonly used metrics for assessing accuracy and efficiency, is also covered in the section. Next, it explores Continual Learning (CL). Several CL methods are discussed in the section. Evaluation methods for CL are also presented, including metrics for measuring performance on both old and new tasks. Finally, the section explores Online Streaming Continual Learning (OSCL): the intersection of SL and Online Continual Learning (OCL). Overall, this section provides a comprehensive overview of the key concepts, methods, and applications of supervised classification methods for learning from evolving data streams. Figure 2.1 gives a general comparison between SL and Online Continual Learning. It also provides a guide to the rest of the subsections.

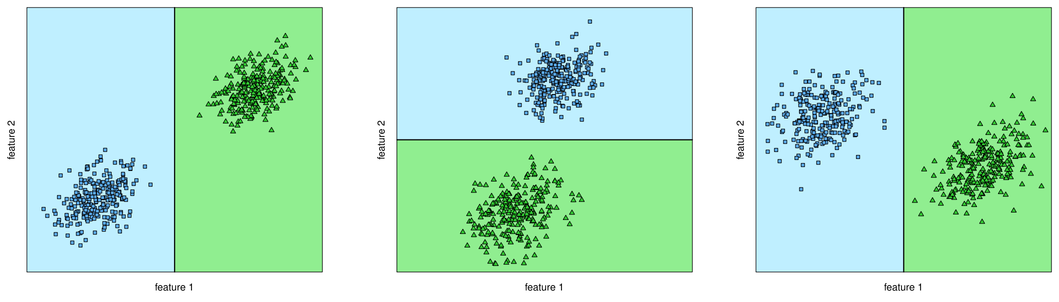
### 2.1.1 Stream Learning

In Stream Learning, a model learns from an evolving data stream (non-IID data), processing one instance at a time. The learner must predict at any given moment using limited processing and memory [9, 8]. Also, it should adjust to distribution changes in the underlying data stream [9, 23]. The literature identifies these distribution shifts as 'concept drifts'[9, 23, 8].

#### 2.1.1.1 Concept Drift

Concept drifts can be categorized according to their impact on the decision boundary, the evolution of the relationship between features and the target, the speed of change, reach, and recurrence [24].

- *Effect on the decision boundary (impact)*: the literature describes real and virtual concept drifts. The former effects the the decision boundary of the model. This affects the performance of the model. The latter does not affect the decision boundary. Hence the model performance is unaffected [2].



(a) Initial distribution. (b) Real concept drift. (c) Virtual concept drift.

Figure 2.2: Different drift types under the "*Effect on the decision boundary (impact)*" category: real and virtual. Source: [2].

- *Evolution of the relationship between features and the target and the speed of change*: in the literature, drifts are categorized into sudden (abrupt), gradual, and incremental drifts, considering the evolution of the relationship between features and the target and the speed of change. With sudden or abrupt drifts, the current data distribution changes to a new one within a short period [2]. This transition happens gradually [24] in

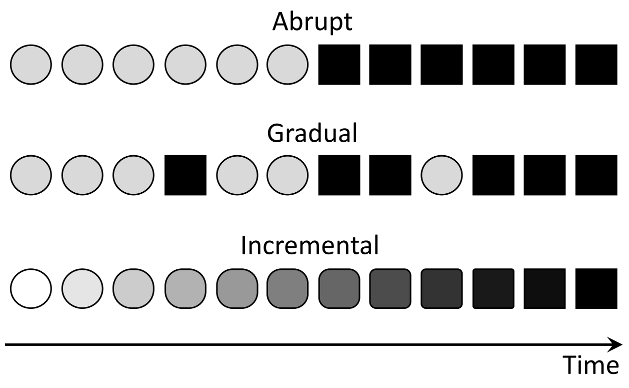


Figure 2.3: Evolution of different drift types under the "*Evolution of relationship between features and the target and the speed of change*" category: abrupt, gradual, and incremental. Source: [3].

the case of gradual drifts. Here for a certain period, one could observe instances from both distributions. The transition time is very long with incremental drifts, and there may not be a statistical difference between adjacent instances [2]. Figure 2.3 shows how the drift types mentioned above evolve.

- *Reach of change*: drifts that affect all of the features are considered global drifts [24], and drifts that affect some of the features are called local drifts [25].
- *Recurrent concept drifts*: if a particular data distribution reoccurs in the stream after a given period, it is considered a recurrent concept drift [24].
- *Random blips/outliers/noise*: are situations where, for a very short time, few instances do not belong to the current distribution popup in the stream [24].

**Drift detectors:** Many types of drift detectors are explained in the literature. [3] describe three types of drift detectors for Stream Learning.

- *Methods based on differences between two distributions*: These methods compare the difference between two data windows. A reference window with old data and a detection window with recent data are compared using a statistical test to discard the null hypothesis that both data belong to the same distribution. Drift detectors based on fixed-size windows usually suffer from a delay in detection [3]. Works such as ADaptive sliding WINdow (ADWIN) [23] use dynamic windows.
- *Methods based on sequential analysis*: These are methods founded on the Sequential Probability Ratio Test (SPRT)[26]. CUSUM and Page–Hinkley [27] are good examples of drift detectors of this type.
- *Methods based on statistical process control*: These methods consider the classification problem a statistical process and monitor the evolution of some performance indicators like error rate to apply heuristics to find change points. For example, DDM [28] has three different states for the classification error evolution: *in-control* when the error is in the control level, *out-of-control* when the error is increasing significantly compared to the recent past, and *warning* when the error is increasing but has not reached the out-of-control level. Where DDM only looks at the

magnitude of the errors, EDDM [29] also considers the distance in time between consecutive errors.

We would like to direct the reader to work by [30] and [31] for a thorough review of drift detectors for Stream Learning.

#### 2.1.1.2 Supervised Classification Methods for Stream Learning

Supervised SL literature explains simple but effective classifiers like Naive Bayes (NB) and Hoeffding Tree (HT) to ensemble learners like ARF and SRP. HT [32] builds a tree using the Hoeffding bound to control its split decisions with a given confidence. Later an adaptive version was introduced to replace the branches when the data stream is evolving [33].

**Ensemble methods** have shown great success in stream learning [8], where it allows one to use currently available efficient stream learning base learners like Hoeffding Tree (HT) in bagging or random forest settings in conjunction with efficient drift detectors like ADaptive sliding WINdow (ADWIN) [23] [15].

**Boosting and bagging** are two popular ensemble learning techniques used in machine learning. Bagging samples instances randomly with replacement to train each item of the ensemble. Boosting, on the other hand, attempts to boost the performance of the next base learner in the ensemble considering the loss of the previous one. It combines the prediction of weak learners addictively to produce a strong learner [34, 35]. AdaBoost [36] highly weights the miss-classified instances by the current base learner to improve the next base learner. Gradient boosting uses the current base learner's gradient information of the loss to improve the next base learner [34]. XGBoost [37] uses this gradient information to derive a particular regression tree that predicts a raw score at the leaf for a given instance.

Data stream boosting is challenging due to the evolving nature of the data. Here the model needs to adjust to the new input distribution of the stream after a concept drift [9, 16]. OnlineBagging and Online Boosting (OB) [38] were inspired by the observation that a binomial distribution $Binomial(p, N)$ can be approximated by a Poisson distribution $Poisson(\lambda)$) with $\lambda = Np$ as $N \rightarrow \infty$. Here, $N$ is the number of instances, and $p$ is the probability of success in the binomial distribution. The $p$ is analogous to uniform sampling with replacement in batch bagging and instance weight in batch AdaBoost [38]. Since the probability of selecting a given example is $1/N$ in the bagging, the uniform sampling with replacement of the bagging algorithm is approximated by $Poisson(1)$ in OnlineBagging. On the other hand, in OB, $\lambda$ is computed by tracking the total weights of correctly classified and misclassified examples for each base learner. The Leveraging Bagging (LB) [39] combines the OnlineBagging technique with the ADWIN. It selectively resets base models whenever their corresponding ADWIN instance flags a drift. An online version of SmoothBoost [40] was proposed in [18]. This Online Smooth Boost (OSB) uses smooth distributions which do not assign too much weight to a single example. When the number of weak learners and examples are sufficiently large, OSB is guaranteed to achieve an arbitrarily small error rate [18, 15]. For imbalanced data streams, a set of online cost-sensitive bagging and boosting algorithms were introduced by [41]: OnlineUnderOverBagging, OnlineSMOTEBagging, OnlineAdaC2, OnlineCSB2, OnlineRUSBoost, and OnlineSMOTEBoost. In these online algorithms, cost sensitivity to class imbalances were introduced by manipulating the parameters of the Poisson distribution discussed in [38]. The main aim of empirical evaluation [41] was to understand the performance gap between online algorithms and their batch counterparts. Evolving data streams were not considered in their evaluation. Recently, two notable approaches were proposed by the stream learning community to leverage gradient boosting for data streams: AXGB [16] and ADITER [17]. AXGB employs mini-batch trained XGBOOST as its base learners and adjusts the ensemble in response to concept drifts, which it detects using ADWIN [23]. ADITER attempts to identify the weak learners in the ensemble and prune them when confronted with concept drift. It then employs multiple training iterations via majority vote among the ensemble to support different drift types. Both AXGB and ADITER only support binary classification. But, the streaming gradient boosting method proposed in this work supports both binary and multi class problems.

ARF [14] and SRP [15] are two popular recently proposed ensemble learning methods. They allow one to use efficient stream learning base learners like Hoeffding Tree (HT) in random forests or bagging set up in conjunction with efficient drift detectors like ADWIN. ARF is an online random forest implementation for data streams that uses effective re-sampling strategies, drift detection, and drift recovery strategies [14]. It simulates re-sampling, as in LB. ARF uses a drift detection and recovery strategy based on detecting warnings and drifts per base tree. After a warning is triggered, a background tree is created and trained without affecting the ensemble's performance. When the warning is escalated to drift, the base tree is replaced with the background tree. Compared to serial implementation, its parallel implementation yields superior computing performance without compromising classification accuracy. SRP is a bagging method that trains base learners on random sub-sets of features and instances identified as sub-spaces[15]. It uses the same drift detection and recovery strategy as ARF but produces superior results compared to ARF [15]. SRP does not have a parallel implementation yet.

OSB performed better compared to OB in the [18] empirical evaluation. Empirical evaluation [15] shows that even with 100 base learners, ARF and SRP outperform OSB by a large margin. In the same evaluation, SRP outperformed ARF. AXGB failed to outperform ARF in the [16] empirical evaluation. In [17] empirical evaluation ADITER also failed to surpass ARF on synthetic data sets with 10000 instances. However, in the same evaluation, ADITER surpassed ARF on real-world data. In that evaluation, all the other data sets had less than 100000 instances apart from airlines. Above empirical evaluations suggest that the latest gradient boosting methods for evolving data streams are yet to surpass current state-of-the-art ensemble methods like SRP.

**Neural Networks** have shown valuable recent advances in various fields; however, to our knowledge, there is still a gap in the research focusing on NNs for evolving data streams. ADL was proposed where the network's depth and width are dynamically managed according to drift detection, the network's generalization power, and the level of mutual information between hidden layers [19]. Network Significance estimates the network generalization power in terms of bias and variance and is used to grow or prune hidden nodes. Once a drift is detected, the model adds a new hidden layer. Based on the mutual information analysis across hidden layers, layers with high correlations are avoided. The network is structured differently than a standard MLP, where each layer has a softmax layer, and the final output is obtained by weighted voting. As this structure limits its use to classification tasks, an MLP-like structure was proposed by keeping the same dynamic network growing and pruning strategy [42]. Later, the same approach was extended to autoencoders [43] and RNNs [44]. All these variants were trained with a single epoch using mini-batches. ADL was compared against Support Vector Machine (SVM)s and distribution-free one-pass learning in [45]. In the experiments, data were fed in batches, and the models were trained for multiple iterations using a given batch. Defining robustness as a relationship between one algorithm's accuracy and the smallest accuracy among all algorithms, the authors found the ADL method to be one of the least robust ones.

Previous work on NNs for data stream learning was mainly focused on using mini-batches to train and test the algorithms. It is reasonable to assume that drift could occur in the middle of a mini-batch. The effect of mini-batch size has been investigated for ensemble learners in a streaming setting [46]. However, to the best of our knowledge, it is yet to be scrutinized for NNs. Also, the previous work on NNs for data streams mainly focused on dynamically adjusting the network structure. In contrast, there is less focus on investigating the effects of learning rate and optimization methods. Exploring those fields involves the challenging aspect of hyperparameter tuning for data streams [47]. Self hyperparameter tuning for data streams, which requires a double pass over the data during the exploration phase, was proposed by [48]. Later, it was further improved to use a single pass [49]. During its exploration phase, triggered by a concept drift detection, for $n$ hyperparameters, the method creates $n + 1$ models plus seven experimental models proposed by the Nelder–Mead algorithm, using shallow copies of the best model. The exploration starts by randomly selecting the $n + 1$ models, and it stops when the best, good, and worst models converge. It was compared against the algorithms with their default values, offline grid search, and offline random search for various machine learning tasks, including classification, regression and recommendation. The authors point out that the results for classification were fairly similar to those obtained by the model with the default hyperparameters while promising accuracy was obtained for recommendation problems. Also, using this method to optimize NN's width or depth could be difficult, as weights and biases need to be compressed or expanded to match the new setting.

#### 2.1.1.3 Evaluation

Several methods are explained in the SL literature for evaluating a model. The most popular one is the *test-then-train* approach [9, 50]. As the name suggests, the evaluation uses the incoming instance to test the model first and later train the model. Here the current predictive evaluation is affected by the previous evaluations. This may be desirable when one is interested in the model's overall performance. Test-then-train is also known as prequential evaluation in the literature. The prequential evaluation may not be reliable in conveying the current predictive performance of the model. Therefore prequential evaluation can be equipped with a sliding window, or a fading factor, to gracefully forget the performance of instances from the distant past [9, 50]. Prequential evaluation is still applicable for partly labelled data, as the loss can be calculated on just the labelled subset of instances [12]. Data stream *cross-validation* was introduced by [51], where models are trained and tested in parallel on different folds of the data. *Continuous re-evaluation* considers the verification latency in the streaming setting with partially delayed labels [52, 53]. This evaluation attempts to evaluate how fast a model can transform from an initial, possibly incorrect prediction to a correct prediction before the availability of the true label.

There are several metrics explained in the literature to measure the performance of an SL classification algorithm. The most popular one is accuracy. If the data stream is imbalanced, accuracy can be misleading; sensitivity and specificity are better measurement alternatives [11]. The *kappa* statistic compares the model's prequential accuracy against the chance classifier (one that randomly assigns to each class the same number of instances as the model under consideration) [9]. On the other hand, the *kappa M* compares the current model's performance against the majority class classifier [9]. *Kappa temporal* attempts to capture the temporal dependencies in a data stream by comparing the model performance against a "no-change" model, which predicts the next instance using the current instance's label [9][1]. For delayed label situations, when multiple predictions are made for a single instance, accuracy and kappa values can be aggregated to produce immediate measures until the true label is available [53, 12].

Regression SL uses two main evaluation metrics: (i) Root mean squared error (RMSE) and (ii) Mean absolute error (MAE) [11]. We direct the reader to [9, 11] for thorough reviews of regression evaluation methods and [54] for clustering evaluation methods. Furthermore, data stream evaluation also considers computing and memory usage [9].

[1]These measurements are thoroughly explained in [9]

#### 2.1.1.4 Application

SL has been used in many situations where learning happens from an evolving data stream. [3] used SL on data generated by optical sensors, which measure the flying behaviour of insects to identify disease vector insects. Also, [55] used SL methods for online crude oil price prediction. SL was used to predict power production considering environmental conditions [56]. The study by [57] contains some interesting applications of SL for monitoring and control problems. It includes application tasks such as traffic management, activity recognition, communication monitoring, controlling robots, intelligent appliances, intrusion detection, fraud detection, and insider trading. The study also contains some interesting areas where SL could provide solutions. We like to direct the reader to [57] for a broader understanding of SL applications.

### 2.1.2 Continual Learning

The literature has thoroughly documented that an NN receiving non-IID data forgets past knowledge when confronted with a concept shift [10, 4]. CL attempts to learn with minimal forgetting of past concepts [10, 4]. In OCL, this learning happens online. Three main continual learning settings are described in the literature: task-incremental, class-incremental, and domain-incremental.

- *Task-incremental*: In this setting, output distributions are demarked by external task ids, available for training and testing. In this setting, the model can use the external task-id signal at test time [4].
- *Class-incremental*: Each distribution consists of classes that are unavailable in other distributions (tasks). This setting adapts a single-head NN configuration. Here, output distributions differ from task to task [4].
- *Domain-incremental*, on the other hand, assumes output distribution from one task to the other to be the same while having different input distributions [4].

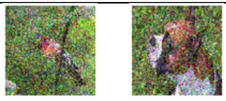
| Task | Task Incremental | Class Incremental | Domain Incremental |
|---|---|---|---|
| $D_{i-1}$ | x: y: Bird Dog | x: y: Bird Dog | x: y: Bird Dog |
| task-ID(test) | **i-1** | **Unknown** | **Unknown** |
| $D_i$ | x: y: Ship Guitar | x: y: Ship Guitar | x:  y: Bird Dog |
| task-ID(test) | **i** | **Unknown** | **Unknown** |

Figure 2.4: Three main CL settings discussed by [4]. *Task-incremental*: tasks are demarked by task id. Task id is available at the test time. *Class-incremental*: different classes are present at each task. Task id is not available at the test time. *Domain-incremental*: each task contains the same set of classes, but the input distribution changes from one task to another, e.g., blur vs. noise. Task id is not available at test time. Source: [4].

In both class-incremental and domain-incremental settings, an external task-id that separates one task from another is assumed to be unavailable at test time [4]. The availability of this signal at training is optional [4]. However, some CL methods rely on this signal during training. Online Class Incremental Continual Learning (OCICL) and Online Domain Incremental Continual Learning (ODICL) assume class-incremental and domain-incremental Online Continual Learning settings, respectively.

#### 2.1.2.1 Methods

CL algorithms use three popular approaches to avoid catastrophic forgetting in NNs: regularization, replay, and parameter isolation.

**Regularization methods:** algorithms like EWC [10] and LwF [20] adjust the weights of the network in such a way that it minimizes the overwriting of the weights for the old concept. EWC uses a quadratic penalty to regularize updating the network parameters related to the past concept. It uses the Fisher Information Matrix's diagonal to approximate the importance of the parameters [10]. EWC has some shortcomings: 1) the Fisher Information

Matrix needs to be stored for each task, and 2) it requires an extra pass over each task's data at the end of the training [4]. Though different versions of EWC address these concerns [4], [58] seems suitable for online CL by keeping a single Fisher Information Matrix calculated by a moving average. LwF uses knowledge distillation to preserve knowledge from past tasks. Here, the model related to the old task is kept separate, and a separate model is trained on the current task. When the LwF receives data for a new task ($X_{new}$, $Y_{new}$), it computes the output ($Y_{old}$) from the old model for the new data $X_{new}$. During training, assuming that $\hat{Y_{old}}$ and $\hat{Y_{new}}$ are predicted values for $X_{new}$ from the old model and new model, LwF attempts to minimize the loss: $\alpha L_{KD}(Y_{old}, \hat{Y_{old}}) + L_{CE}(Y_{new}, \hat{Y_{new}}) + R$ [4]. Here $L_{KD}$ is the distillation loss for the old model, and $\alpha$ is the hyper-parameter controlling the strength of the old model against the new one. $L_{CE}$ is the cross-entropy loss for the new task. $R$ is the general regularization term. Due to this strong relation between old and new tasks, it may perform poorly in situations where there is a huge difference between the old and new task distributions [4].

**Replay methods** present a mix of old and current concepts instances to the NN based on a given policy while training. This reduces forgetting as the training instances from the old concepts avoid complete overwriting of past concept's weights. GDumb [59], ER [21], and MIR [22] are some of the most popular CL replay methods. GDumb attempts to maintain a class-balanced memory buffer using instances from the stream. At the end of the task, it trains the model using the buffered instances. ER uses reservoir sampling to sample instances from the stream to fill the buffer. Reservoir sampling ensures that every instance in the stream has the same probability of being selected to fill the buffer. ER uses random sampling to retrieve instances from the memory buffer. Despite its simplicity, ER has shown competitive performance in ODICL[4]. Five (three buffer and two non-buffer) tricks have been proposed by [60] to improve the accuracy of ER in the OCICL setting. The buffer tricks are independent buffer augmentation, balanced reservoir sampling, and loss-aware reservoir sampling. The two non-buffer tricks are bias control and exponential learning rate decay. Except for bias control which controls the bias of newly learned classes, these tricks can be used in ODICL to improve the performance of a replay method. MIR uses the same reservoir sampling as ER to fill the memory buffer. However, when retrieving instances from the buffer, it first does a virtual parameter update using the incoming mini-batch. Then it selects the top $k$ randomly sampled instances with the most significant loss increases by the virtual parameter update for training. In the online implementation in [4], this virtual update is done on a copy of the NN. Replay Using Memory Indexing (REMIND) [61] takes this approach to another level by storing the internal representations of the instances by the initial frozen part of the network and using a randomly selected set of these internal representations to train the last unfrozen layers of the network. REMIND can store more instance representations using internal low-dimensional features. In general, these replay approaches are motivated by how the hippocampus in the brain stores and replays high-level representations of the memories to the neocortex to learn from them [61]. The empirical survey by [4] suggests that ER and MIR perform better on OCICL and ODICL than other OCL methods. More recently, [62] has proposed repeated augmented rehearsal to improve replay methods. The method utilize data argumentation for replayed instances to avoid over-fitting on replay buffer data[2]. The approach seems to improve all replay methods in general.

**Parameter-isolation:** The intuition behind parameter-isolation methods is to avoid interference by allocating separate parameters for each task [4]. There are two types of parameter-isolation-based methods: *fixed architecture* and *dynamic architecture*. Fixed architecture only activates the relevant part of the network without changing the NN architecture [4]. On the other hand,

[2]A well-documented issue in replay methods [62].

dynamic architecture adds new parameters for the new task while keeping the old parameters [63, 4]. Continual Neural Dirichlet Process Mixture (CN-DPM) [64] trains a new model for each new task and leaves the existing models untouched so that at a later point, it can retain the knowledge of the past tasks. It comprises a group of experts where each expert contains a discriminative and a generative model. Each expert is responsible for a subset of the data. The group is expanded based on the Dirichlet Process Mixture using Sequential Variational Approximation [4].

Most current ODICL methods rely on an explicit end-of-task signal during training. EWC and LwF use this signal to optimize weights, while replay methods can use it to update their replay buffer. However, GDumb, ER, and MIR do not rely on this signal for replay buffer updates. Though [4] defines ODICL as training without the end of the task signal. Implementations such as [65] and [66] use the end of the task signal to employ CL methods such as EWC and LwF. However, on the other hand, the implementation in [67] assumes a gradual distribution shift in the input data distribution where instances from both the new and old tasks can appear in the stream for a certain period. We would like to direct the reader to a survey by [4] for in-depth detail about those methods.

#### 2.1.2.2 Evaluation

There are many evaluation metrics defined in the CL literature. On a stream with $T$ tasks, after training the NN from tasks 1 to $i$, let $a_{i,j}$ be the accuracy on the held-out test set for task $j$. *Average accuracy* ($A_i$) at task $i$ is defined as:

$$A_i = \frac{1}{i} \sum_{j=1}^{i} a_{i,j} \tag{2.1}$$

[58]. Average forgetting ($F_i$) at task $i$ is defined as:

$$F_i = \frac{1}{i-1} \sum_{j=1}^{i-1} f_{i,j} \tag{2.2}$$

, where

$$f_{k,j} = \max_{l \in \{1,\ldots,k-1\}} (a_{l,j}) - a_{k,j} \forall j < k$$

Here $f_{k,j}$ is the best test accuracy the model has ever achieved on task $j$ before learning task $k$. $a_{k,j}$ is the test accuracy on task $j$ after learning task $k$ [58]. The positive influence of learning a new task on previous tasks' performance is measured by *Positive Backward Transfer* (BWT):

$$BWT = \max\left(\frac{\sum_{i=2}^{T}\sum_{j=1}^{i-1} a_{i,j} - a_{j,j}}{\frac{T(T-1)}{2}}, 0\right) \tag{2.3}$$

[4]. The positive influence of learning a given task on future tasks' performance is defined as *Forward Transfer* (FWT):

$$FWT = \frac{\sum_{i=1}^{T-1}\sum_{j=2}^{T} a_{i,j}}{\frac{T(T-1)}{2}} \forall i < j \tag{2.4}$$

[4]. Further to the above metrics, run-time and memory usage are also considered when evaluating OCL methods [4].

#### 2.1.2.3 Applications

Recent research has focused on using ODICL methods to avoid costly retraining in practical situations where the model is confronted with a concept shift. ODICL has been used in X-ray image classification to avoid costly retraining on distribution shifts due to unforeseen shifts in hardware's physical properties [67]. Also, it has been used to mitigate bias in facial expression and action unit recognition across different demographic groups [65]. Furthermore, ODICL was used to counter retraining on concept shifts for multi-variate sequential data of critical care patient recordings [66]. The authors highlight some replay method's infeasibility due to strong privacy requirements in clinical settings. This concern is further highlighted in the empirical study by [4]. Practical implementations such as [65] and [66] use the end of the task signal to employ OCL methods such as EWC and LwF. However, on the other hand, practical implementation in [67] assumes a gradual distribution shift in the input data distribution where instances from both the new and old tasks could appear in the stream for a certain period.

### 2.1.3 Online Streaming Continual Learning (OSCL)

In Stream Learning, the objective is to adjust to the current concept in the stream efficiently. On the other hand, OCL has dual learning objectives: adapt to the present concept while preserving knowledge about previous concepts. Both settings assume data is non-IID. In Stream Learning, it is assumed that model should detect distribution changes and adapt accordingly. However, in OCL, the end of the concept signal is provided at training time, even though some replay methods may not use it. This end-of-the-concept signal is only provided for task-incremental CL at test time. Figure 2.1 also shows the differences between these two settings. Contrary to the differences, these two fields have many intersection points. We identify these intersection points as Online Streaming Continual Learning (OSCL). In OSCL, we mainly identify how well-researched Stream Learning techniques and methods could be used to enhance OCL.

SL on recurrent concept drifts attempts to adjust to an evolving data stream where some concepts could reemerge later in the stream [24]. The setting is similar to OCL but without the additional learning objective of preserving old knowledge explicitly. Hence evaluation in this setting does not consider measuring forgetting of past knowledge. Most methods explored in this setting keep a fixed-size pool of classifiers [24]. Various mechanisms are explored in the literature on maintaining this pool [24, 68] and using it for prediction [24, 69]. This ”pool of classifiers” is also known as ”concept history”, ”concept list”, and ”concept repository” in the literature [24]. Measures like *concept equivalence* and *concept similarity* were introduced to identify the current concept in the data stream from the concept pool.

- *Conceptual equivalence* assumes that when two classifiers behave similarly on a given time window, both describe the same concept [70].

- *Concept similarity*: recognizes similar concepts using Euclidean distances between concept clusters [71]. Thus it can detect recurring drifts in unlabelled data.

Measures such as *concept equivalence* and *concept similarity* could be used for model selection and data retrieval from the replay buffer in OCL. When handling recurrent concepts, predicting the following concept is helpful so the learner can adjust to the incoming concept ahead of time. A probabilistic network was proposed to predict future changes by [72]. The patterns acquired during previous drifts to predict the time of the next drift was proposed by [73]. The method assumed a Gaussian distribution for the duration of the concepts. A recent survey by [24] discusses the above and many more exciting topics on SL for recurrent concept drifts.

Most of the OCL methods rely on externally provided end-of-concept signals (task ids) at training[3]. It is critical for an autonomous learning agent to detect these concept shifts and adjust accordingly. While OCL research has explored different methods to preserve old knowledge when adjusting to new concepts, SL has done an excellent job of understanding how to detect distribution shifts, especially through different drift detection methods on streams with varying types of drift (abrupt, gradual, incremental, and recurrent) and different label conditions (available for all instances/ delayed label/ no label). OCL could utilize well-researched drift detectors to detect the end-of-concept signal. Recent work by [74] has explored the use of drift detectors to identify when to use the data from the instance buffer and how to use it in an OCL setting. Having well-researched SL knowledge on different distribution shifts and different drift detectors would allow OCL algorithms to be more effective in practical OCL scenarios, like the gradual distribution shifts in x-ray images [67].

Furthermore, semi-supervised SL could make OCL to be more practical when label data is only sometimes immediately available. Works like ORDisCo

[3]Due to internal instance buffers, some OCL models may not need task ids at train time. The performance of these models is heavily dependent upon the size of this instance buffer [4]

Table 2.1: Synergies and differences between SL and OCL.

| Topic | SL | OCL |
|---|---|---|
| Setting | Single learning objective: adjust to current concept efficiently. | Dual learning objective: adjust to current concept and preserve old knowledge. |
| Drift detection | Thoroughly studied | Can be used for task detection Some recent OCL work: [74]. |
| Drift prediction. | Used when dealing with recurrent concept drifts. | Can be used for task prediction. Some SL work: [72], [24]. |
| Missing labels | Some methods have been proposed to tackle this [12]. | Yet to be fully explored. Can employ some of the SL approaches discussed in [12]. |
| Recurrent concept drifts | Similar to OCL, without explicit learning objective to preserve old knowledge. For latest research refer to [24]. | SL concept pool maintenance techniques [24] can be useful in maintaining references to different NN structures in OCL parameter-isolation methods. Concept equivalence and concept similarity can be used to retrieve relevant instances or NN structures. Many more techniques are discussed in [24]. |
| Evaluation | Frameworks can employ OCL dual learning objective and metrics discussed in section 2.1.2.2. So SL methods and techniques can be evaluated under OCL setting. | Employs dual learning objective. |
| Application | Suitable for applications which needs to adjust to current concept very quickly. | Suitable for applications which needs to adapt to current concept very quickly while preserving old knowledge. |

[75] and CURL [76] have started exploring this research area. Semi-supervised SL methods under *Self-training* and *learning by disagreement* categories [12]

could easily be deployed in real-world OCL settings where labels are not always present. Many more opportunities exist, considering the breadth of semi-supervised SL methods discussed by [12].

Streaming clustering is another exciting area to explore in future OCL work. Here, well-established streaming clustering algorithms [13] could solve interesting OCL problems as streaming clustering algorithms extract patterns from evolving data. A streaming clustering algorithm could extract tasks from an unsupervised OCL setting. The extracted information, such as task information, could be used for model selection and data retrieval from the replay buffer. The most exciting aspect of the streaming clustering algorithms for OCL is that they are well-studied for evolving data streams.

On the other hand, SL could adapt the dual learning objective in OCL (adjust to the current concept while preserving knowledge about previous concepts). This would allow one to evaluate the breadth of well-studied SL methods for OCL. There is some emerging work in this area where catastrophic forgetting is explored in HTs [77]. Implementing the OCL evaluations discussed in [4] on popular SL platforms such as MOA [78] and River [79] would speed up this area of research.

Table 2.1 summarizes the above-discussed synergies and differences between Stream Learning and OCL. It points out the differences in the settings and lists some future research directions in Online Streaming Continual Learning.

# Chapter 3

# Streaming Gradient Boosted Trees

Gradient boosting methods are less performant than bagging and random forest based methods in SL (see section 2.1.1.2). This chapter introduces a novel gradient-boosted trees algorithm for evolving data streams to address this limitation.

## 3.1 Introduction

Boosting methods have become increasingly successful in machine learning over the past decade. While early weighed boosting algorithms such as AdaBoost showed promise [36], they were later surpassed by gradient boosting methods [35, 80]. Gradient Boosting leverages the previous base learner's gradient information to boost the performance of the next learner in an ensemble. The eXtreme Gradient Boosting (XGBoost) [37] takes this approach to another level, achieving high efficiency and superior performance on various time-critical real-world problems. However, in many real-world scenarios, traditional batch learning with the IID assumption cannot keep pace with the evolving nature of the underlying data stream [8, 9]. On the other hand, SL accounts for the possibility of change in underlying data distribution (concept drift) [9]. Here, a model should respond efficiently in real-time when learning from an evolving data stream [9]. Streaming gradient boosting poses challenges in terms of handling concept drift and resetting specific components of the booster. This difficulty arises from the additive nature of the ensemble setup. While the research community has proposed methods such as Adaptive eXtreme Gradient Boosting (AXGB) [16] and Adaptive Iterations (ADITER) [17] to enable gradient boosting for evolving data streams, they failed to outperform state-of-the-art ensemble learners like Adaptive Random Forest (ARF) [14] and Streaming Random Patches (SRP)[15]. This chapter introduces a novel approach to setting up gradient-boosted trees for evolving data streams.

This chapter utilizes streaming regression trees with inbuilt drift detectors in a gradient-boosted setting. Overall, this work proposes a promising new gradient-boosted tree ensemble approach for evolving data streams that outperforms existing techniques. We outline following contributions:

1. To our knowledge, proposed Streaming Gradient Boosted Trees (SGBT) is the first instance where the weighted squared loss derived in [37] with *hessian* as the *weight* and *gradient* over *hessian* as the *target* considering the previous boosting step's loss, is used to develop a streaming gradient-boosted method for evolving data streams.

2. SGBT utilises trees with an internal Tree Replacement (TR) mechanism instead of externally monitoring each item in the boosting ensemble for drifts and adjusting each item like AXGB [16] or resetting some parts as in ADITER[17]. This Tree Replacement mechanism in SGBT allows the trees in the booster to adapt dynamically to concept drifts. Unlike binary-class gradient-boosted streaming implementations: AXGB and ADITER, SGBT can solve multi class problems using a committee of trees at each boosting step or a committee of SGBTs.

3. The chapter presents an extensive empirical evaluation of SGBT against current state-of-the-art streaming bagging (SRP), random forest (ARF),

boosting (OSB), and gradient boosting (AdIter) methods on 14 datasets with different drift types.

## 3.2 Streaming Gradient Boosted Trees (SGBT)

In gradient boosting, a model $\phi$ can be represented as $S$ additive functions:

$$\hat{y} = \phi(x_i) = \sum_{s=1}^{S} f_s(x_i), f_s \in \mathcal{F} \tag{3.1}$$

[80, 37]. Here, $\mathcal{F}$ is the space of regression trees. In XGBoost [37], each $f_s$ corresponds to an independent tree structure with leaf weights $\omega$. Unlike other trees, each regression tree contains a continuous score $\omega_i$ at the leaf for $i$-th instance. The authors proposed to sum up the corresponding scores at the leaves of each tree for prediction. For a dataset with $n$ instances, the learning objective is to minimize the regularized objective:

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{s=1}^{S} \Omega(f_s) \tag{3.2}$$

Where $\Omega$ penalizes the complexity of the tree $f$:

$$\Omega(f) = \gamma T + \frac{1}{2}\beta \left\| \omega \right\|$$

$T$ is the number of leaves in the tree. $l$ is a differentiable convex loss function that measures the difference between the prediction $\hat{y}_i$ and the target $y_i$. Furthermore, the loss at the $s$-th step is the loss incurred by the previous $(s-1)$ step and the loss incurred by tree $f_s$ plus the regularization term:

$$\mathcal{L}^{(s)} = \sum_{i=1}^{n} l(y_i, \hat{y}^{(s-1)} + f_s(x_i)) + \Omega(f_s) \tag{3.3}$$

This loss could be approximated using second-order Taylor approximation to:

$$\mathcal{L}^{(s)} \simeq \sum_{i=1}^{n} \left[ l(y_i, \hat{y}^{(s-1)}) + g_i f_s(x_i) + \frac{1}{2} h_i f_s^2(x_i) \right] + \Omega(f_s) \tag{3.4}$$

[37]. Here $g_i = \partial_{\hat{y}^{(s-1)}} l(y_i, \hat{y}^{(s-1)})$ and $h_i = \partial^2_{\hat{y}^{(s-1)}} l(y_i, \hat{y}^{(s-1)})$ are the first and second-order gradient statistics of the loss considering $s-1$-th prediction. Though the authors [37] use a simplified version of the above loss function by

removing constants to derive raw score values at the leaves, the below version was derived to explain it as a weighted squared loss with weight $h_i$ and target $g_i/h_i$:

$$\sum_{i=1}^{n} \frac{1}{2} h_i(f_t(x_i) - g_i/h_i)^2 + \Omega(f_s) + constant. \tag{3.5}$$

At the $s^{th}$ boosting step, the loss function consists of a weighted squared loss, a regularization term specific to tree $s$, and a constant term. Equation 3.5 provides the flexibility to utilize various streaming regression trees instead of the one employed in XGBoost. Moreover, depending on the implementation, the regularization term in the streaming regression tree implementation can diverge from that employed in XGBoost [1].

In data stream learning, $n$ could be infinite, and learning happens online, where a model $\phi_{i-1}$ learned at $i-1^{\text{th}}$ instance is used to predict the $i^{\text{th}}$ instance. Also, from any $i^{\text{th}}$ instance underlying distribution of $x$ could change (concept drift). To adjust to the new distribution at $i$, model $\phi_i$ should adjust it's regression trees. Instead of externally monitor and reset each $f_s$ tree like in AdIter [17], in SGBT the trees internally monitor their standardized absolute error and train an alternate tree if it goes above a warning level. The tree $f_s$ switches to its alternate tree once the error reaches a danger zone. To trigger these warning and danger signals, $f_s$ tree employs a drift detector to monitor its standardized absolute error. The rest of the thesis identifies this strategy of replacing the active tree with an alternate tree on the drift detection signal as Tree Replacement (TR). In the experiments, we used two regression trees for data streams: FIMT-DD [81][2] and SGT [82] with in-

[1]The streaming regression trees used in the experiments do not have an explicit regularization constraint. However, they utilize a Tree Replacement strategy to control tree growth, which is explained later in this chapter. An alternative approach would be to incorporate an explicit memory constraint similar to the one used in HT as a regularization strategy. There could be many other efficient approaches to constrain tree growth. Thus, to avoid scope expansion, we did not dedicate time to explicitly implement this constraint on each regression tree used in the experiments.

[2]FIMT-DD does not support nominal features. It was changed to pass the index of

**Algorithm 1** TRAINING SGBT

```
Input: $y_i$: label as a one-hot vector for $x_i$, $lr$: learning rate, $S$: # boosting steps, $\hat{y}^{(0)}$: initial prediction, $l$: loss function, $m$: % of features for training, $C$: # classes.
1: if start of training then
2:     Initialize $M$ using $m$.    ▷ $M$={sets of randomly picked $m$% of features for each boosting step}
3: end if
4: for $s := 0$ to $S$ do
5:     Generate instance $x_{i,s}$ using feature set $m_s$ ($\in M$) from instance $x_i$.
6:     $g_i \leftarrow \partial_{\hat{y}^{(s-1)}} l(y_i, \hat{y}^{(s-1)})$    ▷ gradient for committee
7:     $h_i \leftarrow \partial^2_{\hat{y}^{(s-1)}} l(y_i, \hat{y}^{(s-1)})$    ▷ hessian for committee
8:     for $c := 0$ to $C - 1$ do
9:         TRAIN $f_{s,c}(x_{i,s}, g_{i,c}/h_{i,c})$    ▷ train $f_{s,c}$ using instance $x_{i,s}$ and label $g_{i,c}/h_{i,c}$
10:    end for
11:    $\hat{y}_i \leftarrow \hat{y}_i + lr * f_s(x_i)$    ▷ scale $f_s(x_{i,s})$ and add to $\hat{y}_i$
12: end for
```

built drift detectors: Page-Hinckley Test (PHT) [83] and DDM [28]. The implementation of SGT with DDM is generic, and one could replace SGT with any other regression tree for data streams.

The loss function in equation 3.5 requires the regression trees to support fractional weights, as $h_i$ could be a fractional value for some loss functions. Streaming regression trees (SGT and FIMT-DD) considered in this work only support integer weights. Supporting fractional weights for them is not trivial. For example, SGT and FIMT-DD require the incremental calculation of variance and co-variance for fractional weights. Though recent work by [84] and [85] suggests this is possible, this itself is a separate research topic. Hence, even though SGBT calculates these weights (hessians), it does not pass them to the underlying trees for this practical reason. Alternatively, it passes a weight of 1 to the trees.

Instead of using all the features to train at each boosting step, SGBT uses a subset of features based on a predefined feature percentage. This approach of using a subset of features to train each ensemble is also used in ARF and SRP [14, 15] to leverage diversity among the base learners. Algorithm 1 explains the training procedure of SGBT.

Two approaches are used to support multi class problems: SGBT and SGBT$^{MC}$.

- **SGBT** uses a committee of regression trees in a given boosting step $s$. Here a single tree is trained for each class. The committee is composed of a softmax function, so the probability that an instance, $x_i$, belongs to class $c$ is given by:
$$\hat{y}_{i,c} = \frac{exp(f_{s,c}(x_i))}{\sum_{c=1}^{C} exp(f_{s,c}(x_i))} \tag{3.6}$$
Here $f_{s,c}$ is the regression tree trained to predict a real-valued score for class $c$ at $s$-th boosting step, and $C$ is the number of classes. In practice, hard-wiring $f_{s,C}(x_i) = 0$ allows SGBT to reduce the number of trees

the nominal value to the split criterion. This allows one to avoid one-hot encoding of those nominal features, as often it requires more computing due to increased dimensionality.

being trained[3]. The categorical cross-entropy loss ($l^{CE}$) is used to train the model:

$$l^{CE}(y, \hat{y}) = -\sum_{c=1}^{C} y_c log(\hat{y}_c) \tag{3.7}$$

Here, $y$ is the ground truth encoded as a one-hot vector. For $l^{CE}$, gradient ($g$) is $y_c - \hat{y}_c$, and hessian ($h$) is $\hat{y}_c(1-\hat{y}_c)$. The regression tree committee (composing $C-1$ items) at the $s$-th boosting step represents the base learner for $s$-th boosting step. This approach is also used in SGT[82] to support multi class classification.

- **Sgbt$^{\mathbf{MC}}$** uses the same loss function ($l^{CE}$) as in SGBT. But uses a wrapper classifier to invoke a binary SGBT classifier for each class. The task of the binary SGBT classifier is to distinguish a given class from all the other classes. All $C$ classifier votes for the positive outcome are collected and normalized at prediction. The class associated with the classifier that predicted the positive outcome most confidently is considered the final class for the instance. This approach is very popular in batch learning and is commonly known as *one-vs-rest* or *one-vs-all* in literature [86]. SGBT$^{MC}$ reverts to SGBT for binary class problems to avoid any computing overhead.

Unlike Axgb and AdIter, the above two approaches allow SGBT to support gradient boosting for evolving data streams on multi class problems.

To improve the computing performance and to utilize already calculated hessian weights, two variants of SGBT are proposed.

- **Sgbt$^{\mathbf{SK}}$**: In most streaming regression trees, the computation and memory complexity is affected by the number of instances they process. Some computation and memory savings could be achieved via skip training on random instances. SGBT could randomly skip $1/k$-th of instances ($k \geq 1, \in \mathbb{N}$) if provided. By default, $k$ is set to 1, so all instances are

[3]This practice is used in [82] as well.

**Algorithm 2** TRAINING $\text{SGBT}_{MI}^{SK}$

```
Input: $y_i$: label as a one-hot vector for $x_i$, $lr$: learning rate, $S$: # boosting
    steps, $\hat{y}^{(0)}$: initial prediction, $l$: loss function, $m$: % of features for training,
    $C$: # classes, $k$: randomly skip $1/k$ instances ($k \geq 1, \in \mathbb{N}$).
 1: if start of training then
 2:     Initialize $M$ using $m$.   ▷ $M$={sets of randomly picked $m$% of features
    for each boosting step}
 3: end if
 4: $train\_int \leftarrow RandInt(0, k)$                    ▷ $0 \leq train\_int < k, \in \mathbb{N}$
 5: if $k = 1$ OR $train\_int \neq 0$ then                    ▷ skip $1/k$ th for $k > 1$
 6:     for $s := 0$ to $S$ do
 7:         Generate instance $x_{i,s}$ using feature set $m_s$ ($\in M$) from instance $x_i$.
 8:         $g_i \leftarrow \partial_{\hat{y}^{(s-1)}} l(y_i, \hat{y}^{(s-1)})$          ▷ gradient for committee
 9:         $h_i \leftarrow \partial^2_{\hat{y}^{(s-1)}} l(y_i, \hat{y}^{(s-1)})$          ▷ hessian for committee
10:         for $c := 0$ to $C - 1$ do
11:             $T \leftarrow ceiling(h_{i,c} * 10)$                    ▷ $h_{i,c} < 1$ for $l^{CE}$
12:             for $t := 0$ to $T$ do                    ▷ train $f_{s,c}$ $T$ times
13:                 TRAIN $f_{s,c}(x_{i,s},\ g_{i,c}/h_{i,c})$  ▷ train $f_{s,c}$ using instance $x_{i,s}$ and
    label $g_{i,c}/h_{i,c}$
14:             end for
15:         end for
16:         $\hat{y}_i \leftarrow \hat{y}_i + lr * f_s(x_i)$          ▷ scale $f_s(x_{i,s})$ and add to $\hat{y}_i$
17:     end for
18: end if
```

processed. Work by [87, 88] also exploited skip training for learning from evolving data. Line 5 in algorithm 2 highlights this skip training.

- **SGBT$_{\textbf{MI}}$**: Even though current base learners do not support fractional weights, utilizing already calculated hessian weights is helpful. For $l^{CE}$, hessian for class $c$ at $i$-th instance is always less than 1 ($h_{i,c} < 1$). Even if one passes $h_{i,c}$ to a ceiling[4] function, it will always return 1. For all instances, one could multiply $h_{i,c}$ by 10 and pass that to a ceiling function to get a positive integer weight greater than 1 for some instances. For all the other instances, the weight would be 1. If $ceiling(h_{i,c} * 10) = T$, SGBT can train $f_{s,c}$ base learner $T$ times using instance $x_{i,s}$ with label $g_{i,c}/h_{i,c}$. This approach of training a base learner multiple times based on a calculated integer weight for an instance is quite common in stream learning [38, 15]. Line 12 in algorithm 2 highlights this multiple training iteration approach.

Algorithm 2 explains the above two variants of SGBT in detail. In the experiments, we evaluate the effectiveness of these SGBT variants.

As SGBT allows different streaming regression trees for its base learners, its final time and memory complexity is influenced by the base learner's time and memory complexity. Assuming base learner's time and memory complexity as $\mathcal{O}(f)$, SGBT's time complexity can be derived as $\mathcal{O}(\alpha(C-1)Sf)$, and its memory complexity is $\mathcal{O}(\alpha(C-1)Sf)$. Here, $\alpha$ ($1 \leq \alpha \leq 2$) is the tree generation factor controlled by the drift detector, $S$ is the number of boosting steps, and $C$ is the number of classes. On the other hand, SGBT$^{MC}$'s time and memory complexities are $\mathcal{O}(\alpha CSf)$ and $\mathcal{O}(\alpha CSf)$, respectively for multi class problems. It has the same time and memory complexity as SGBT for binary class problems. Time complexity could be further improved by parallel training each tree in the tree committee at each boosting step for SGBT, and each binary SGBT for SGBT$^{MC}$. This allows Time complexity to get closer

[4]Similar to *Java lang.Math.ceil(v)* that returns an integer value greater than or equal to the passed-in value $v$.

to $\mathcal{O}(\alpha S f)$, in both situations. If the underlying base learner trains its alternate tree in parallel, then the time complexity could be further improved to approach $\mathcal{O}(S f)$.

## 3.3 Experiments

We begin our experiments by comparing SGBT against current state-of-the-art streaming bagging (SRP), random forest (ARF), boosting (OSB), and gradient boosting (AdIter) methods on 14 datasets. We also conducted a parameter exploration to illustrate the effects of different SGBT components. Finally, we show an in-depth analysis concerning the computational requirements of SGBT.

Datasets: $\text{AGR}_\text{a}$, $\text{AGR}_\text{g}$, electricity, airlines, $\text{LED}_\text{a}$, $\text{LED}_\text{g}$, $\text{RBF}_\text{f}$, $\text{RBF}_\text{m}$ and covtype are from [15]. The synthetic datasets without drifts: RandomTree, LED, and RBF5 were generated using MOA *RandomTreeGenerator*, *LEDGenerator*, and *RandomRBFGenerator*. $\text{RBF\_B}_\text{m}$ and $\text{RBF\_B}_\text{f}$ were generated using MOA *RandomRBFGeneratorDrift*. Like in $\text{RBF}_\text{m}$, the speed of change for the centroids was set to 0.0001 for $\text{RBF\_B}_\text{m}$. For $\text{RBF\_B}_\text{f}$, it is set to 0.001, like in $\text{RBF}_\text{f}$. The synthetic datasets with drifts simulate different types of concept drifts, i.e., abrupt ($\text{AGR}_\text{a}$, $\text{LED}_\text{a}$), gradual ($\text{AGR}_\text{g}$, $\text{LED}_\text{g}$), fast incremental changes ($\text{RBF\_B}_\text{f}$, $\text{RBF}_\text{f}$), and moderate incremental changes ($\text{RBF\_B}_\text{m}$, $\text{RBF}_\text{m}$). $\text{AGR}_\text{a}$, $\text{AGR}_\text{g}$, $\text{LED}_\text{a}$, and $\text{LED}_\text{g}$ had concept drifts occurring after every 250000 instances, with a drift width of 50 for abrupt ($\text{AGR}_\text{a}$, $\text{LED}_\text{a}$) and 50000 for gradual ($\text{AGR}_\text{g}$, $\text{LED}_\text{g}$). Table 3.1 summarizes the characteristics of the datasets. Each algorithm was executed multiple times with different random seeds, and the average accuracy was considered in the evaluation process[5].

[5]Table 3.2, figure 3.1 and figure A.1 used ten iterations with random seeds: 5, 9, 17, 13, 19, 23, 29, 31, 37 and 121. All the other experiments used three iterations with random seeds: 9, 17, and 121. Code and data are available at https://anonymous.4open.science/r/SGBT-9D43. Experiments were run on an a) Ubuntu 18.04 LST system with AMD EPYC 7702

Table 3.1: Dataset properties: has (**D**)rifts, (**R**)eal, (**S**)ynthetic, dense($^d$), sparse($^s$).

| name | type | instances | features | # nominal features | # classes | class dist max(%) | class dist min(%) |
|---|---|---|---|---|---|---|---|
| binary class | | | | | | | |
| $\text{AGR}_\text{a}$ | $\text{DS}^\text{d}$ | 1M | 9 | 3 | 2 | 52.83 | 47.17 |
| $\text{AGR}_\text{g}$ | $\text{DS}^\text{d}$ | 1M | 9 | 3 | 2 | 52.83 | 47.17 |
| $\text{RBF\_B}_\text{f}$ | $\text{DS}^\text{d}$ | 1M | 10 | 0 | 2 | 51.75 | 48.25 |
| $\text{RBF\_B}_\text{m}$ | $\text{DS}^\text{d}$ | 1M | 10 | 0 | 2 | 51.75 | 48.25 |
| RandomTree | $\text{S}^\text{d}$ | 100K | 10 | 5 | 2 | 57.84 | 42.16 |
| electricity | $\text{R}^\text{d}$ | 45310 | 8 | 1 | 2 | 57.55 | 42.45 |
| airlines | $\text{R}^\text{d}$ | 539382 | 7 | 4 | 2 | 55.46 | 44.54 |
| multi class | | | | | | | |
| $\text{LED}_\text{a}$ | $\text{DS}^\text{s}$ | 1M | 24 | 0 | 10 | 10.08 | 9.94 |
| $\text{LED}_\text{g}$ | $\text{DS}^\text{s}$ | 1M | 24 | 0 | 10 | 10.08 | 9.94 |
| $\text{RBF}_\text{f}$ | $\text{DS}^\text{d}$ | 1M | 10 | 0 | 5 | 30.01 | 9.27 |
| $\text{RBF}_\text{m}$ | $\text{DS}^\text{d}$ | 1M | 10 | 0 | 5 | 30.01 | 9.27 |
| RBF5 | $\text{S}^\text{d}$ | 100K | 10 | 0 | 5 | 32.17 | 8.10 |
| LED | $\text{S}^\text{s}$ | 100K | 24 | 0 | 10 | 10.00 | 9.96 |
| covtype | $\text{R}^\text{d}$ | 581010 | 54 | 0 | 7 | 48.76 | 0.47 |

SGBT was compared against the current state-of-the-art stream learning baseline SRP, streaming random forest method ARF, the latest gradient-boosted method for data streams AdIter, and the stream-boosting method OSB. Axgb was not considered in the evaluation as it failed to outperform ARF [16]. SRP used the same parameter configurations explained in [15]. As 100 base learners produced the best results for SRP in [15], all the baselines

64-Core Processor at 4.00GHz, and with 1000GB RAM and on b)Ubuntu 20.04.3 system with an Intel(R) Core(TM) i7-6700K CPU at 4.00GHz, and with 64GB RAM. All CPU Time experiments were done on the system a. The OpenJDK version was 11.0.11, and the JVM configurations were: -Xmx96g, -Xms50m, and -Xss1g.

Table 3.2: Accuracy: SGBT$^{MC}$ against other baselines (values rounded to 2 decimals).

| | SGBT$^{MC}$ | SRP | ARF | OSB | AdIter |
|---|---|---|---|---|---|
| binary class | | | | | |
| AGR$_a$ | **94.45** ± **0.01** | 92.81 ± 0.19 | 87.87 ± 0.08 | 90.39 ± 0.01 | 90.73 ± 0.18 |
| AGR$_g$ | **91.91** ± **0.01** | 89.68 ± 0.19 | 82.45 ± 0.11 | 87.87 ± 0.03 | 87.66 ± 0.34 |
| RBF_B$_m$ | 92.10 ± 0.66 | 90.76 ± 0.67 | **92.10** ± **0.63** | 89.27 ± 0.84 | 76.85 ± 1.30 |
| RBF_B$_f$ | 84.33 ± 1.22 | 82.15 ± 1.46 | **85.61** ± **1.31** | 78.14 ± 1.25 | 72.16 ± 1.17 |
| RandomTree | 86.19 ± 8.21 | 87.58 ± 2.78 | 90.15 ± 3.38 | **92.09** ± **2.59** | 68.55 ± 10.79 |
| electricity | 88.50 ± 0.06 | 89.68 ± 0.14 | **90.62** ± **0.05** | 89.51 ± 0.00 | 78.77 ± 0.08 |
| airlines | **68.79** ± **0.03** | 68.54 ± 0.05 | 66.68 ± 0.03 | 64.56 ± 0.00 | 62.72 ± 0.07 |
| avg | **86.61** | 85.89 | 85.07 | 84.55 | 76.78 |
| rank | **2.14** | 2.43 | 2.57 | 3.29 | 4.57 |
| multi class | | | | | |
| LED$_a$ | **74.04** ± **0.01** | **74.04** ± **0.01** | 73.95 ± 0.01 | 72.48 ± 0.00 | |
| LED$_g$ | **73.32** ± **0.01** | 73.25 ± 0.01 | 73.12 ± 0.01 | 72.11 ± 0.01 | |
| RBF$_m$ | **88.00** ± **0.76** | 86.60 ± 0.84 | 87.82 ± 0.75 | 76.81 ± 0.99 | |
| RBF$_f$ | 76.98 ± 1.34 | 76.91 ± 1.21 | **77.69** ± **1.44** | 50.71 ± 1.06 | |
| LED | 73.82 ± 0.14 | **73.87** ± **0.12** | 73.75 ± 0.15 | 73.86 ± 0.18 | |
| RBF5 | 90.13 ± 0.84 | 90.56 ± 0.96 | **90.60** ± **0.99** | 85.67 ± 1.18 | |
| covtype | 94.29 ± 0.03 | **95.34** ± **0.01** | 94.72 ± 0.02 | 92.69 ± 0.00 | |
| avg | 81.51 | 81.51 | **81.66** | 74.90 | |
| rank | **2.00** | **2.00** | 2.29 | 3.71 | |
| avg (both) | **84.06** | 83.70 | 83.37 | 79.73 | |
| rank (both) | **2.07** | 2.21 | 2.43 | 3.50 | |

KappaM [9] results in appendix A.1 also aligns with accuracy rankings.

used 100 base learners keeping all the other parameters to default values. OSB used the same base learner (HT) in SRP with the same hyperparameters as in SRP.

We collected votes for each class on each instance from AdIter's Python implementation and ran it through dummy MOA classifiers to yield the same evaluation as the other methods. SGBT was implemented as an MOA classifier, and it used 100 boosting steps ($S$) to match other baselines 100 base learners. The SGBT$^{MC}$ variant was compared against the above baselines. Here *one-vs-rest* wrapper classifier was also implemented in MOA. SGBT used

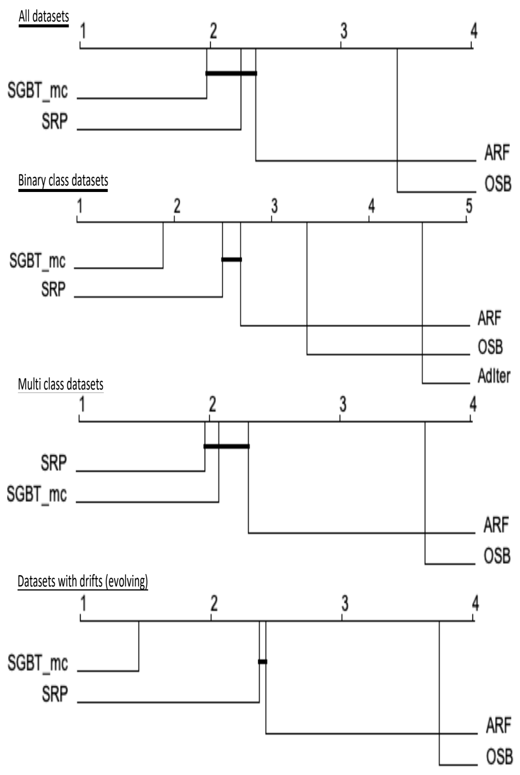


Figure 3.1: Nemenyi Post-hoc test with p-value 0.05 for **all, binary class, multi class, and evolving** ($AGR_a$, $AGR_g$, $LED_a$, $LED_g$, $RBF\_B_m$, $RBF\_B_f$, $RBF_m$, $RBF_f$) datasets (*accuracy*): $SGBT^{MC}$ against other baselines (10 iterations with different random seeds)

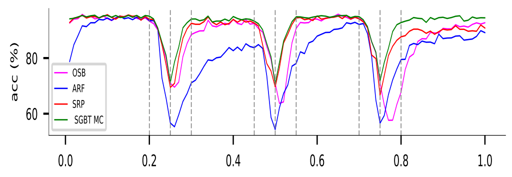


Figure 3.2: Accuracy over time: $SGBT^{MC}$ against SRP, ARF, and OSB on $\mathbf{AGR_g}$. x axis is the number of instances seen so far. Vertical dotted lines mark a concept drift's start, center, and end. $SGBT^{MC}[S = 100, m = 75, lr = 1.25e\text{-}2]$.

a learning rate of 0.0125 and 75% of the features at each boosting step. As SGBT requires streaming regression trees as its base learners, the streaming classifier tree HT can not be used as a base learner. Therefore streaming regression tree FIMT-DD [81] was chosen as its base learner. FIMT-DD used a *variance reduction* split criterion, a *grace period* of 25, a *split confidence* interval of 0.05, a *constant learning rate at the leaves*, and the *regression tree*

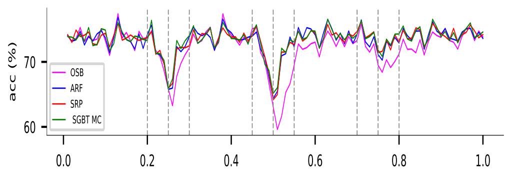


Figure 3.3: Accuracy over time: SGBT$^{MC}$ against SRP, ARF, and OSB on **LED$_\mathbf{g}$**. x axis is the number of instances seen so far. Vertical dotted lines mark a concept drift's start, center, and end. SGBT$^{MC}$[$S$ = 100, $m$ = 75, $lr$ =1.25e-2].

Table 3.3: Time(s): SGBT$^{MC}$ against SRP (values rounded to 0 decimals, except ranks).

| | SGBT$^{MC}_{ST}$ | SGBT$^{MC}$ | SRP |
|---|---|---|---|
| binary class | | | |
| AGR$_a$ | 1423 ± 34 | **1187 ± 32** | 3208 ± 143 |
| AGR$_g$ | 1401 ± 100 | **1160 ± 37** | 3838 ± 200 |
| RBF_B$_m$ | 2278 ± 134 | **1756 ± 190** | 3697 ± 293 |
| RBF_B$_f$ | 2027 ± 27 | **1475 ± 130** | 4728 ± 102 |
| RandomTree | 156 ± 7 | **128 ± 13** | 334 ± 90 |
| electricity | **44 ± 3** | 48 ± 5 | 138 ± 10 |
| airlines | 606 ± 27 | **507 ± 41** | 2892 ± 225 |
| avg | 1134 | **894** | 2691 |
| rank | 1.86 | **1.14** | 3.00 |
| multi class | | | |
| LED$_a$ | 17489 ± 2191 | **1667 ± 7** | 2920 ± 311 |
| LED$_g$ | 16802 ± 1715 | **1669 ± 16** | 2901 ± 315 |
| RBF$_m$ | 14511 ± 1058 | **2767 ± 102** | 3228 ± 114 |
| RBF$_f$ | 13580 ± 1550 | **2399 ± 46** | 3624 ± 540 |
| LED | 1503 ± 104 | **163 ± 2** | 295 ± 20 |
| RandomRBF5 | 1473 ± 119 | 268 ± 5 | **163 ± 18** |
| covtype | 20067 ± 1869 | **1789 ± 22** | 3801 ± 26 |
| avg | 12203 | **1532** | 2419 |
| rank | 3.00 | **1.14** | 1.86 |
| avg (both) | 6669 | **1213** | 2555 |
| rank (both) | 2.43 | **1.14** | 2.43 |

*option.*

Table 3.2 compares SGBT$^{MC}$'s accuracy against the baselines mentioned above. As one can see, SGBT$^{MC}$ outperforms all the baselines on binary class problems considering average accuracy and rank. It also performs equally well on multi class problems. It is also evident that SGBT$^{MC}$ outperforms other methods on datasets with drifts: AGR$_{\mathrm{a}}$, AGR$_{\mathrm{g}}$, LED$_{\mathrm{a}}$, LED$_{\mathrm{g}}$, and RBF$_{\mathrm{m}}$. This suggests that SGBT$^{MC}$ is a good candidate for evolving data. It also performed well on the airlines dataset. On the other hand, SRP yielded good results on LED$_{\mathrm{a}}$, LED, and covtye datasets, while ARF performed well on RBF_B$_{\mathrm{m}}$, RBF_B$_{\mathrm{f}}$, electricity, RBF$_{\mathrm{f}}$, and RBF5. OSB performed well on RandomTree dataset. The streaming gradient boosting method AdIter was the least performant among all methods. As it is a binary classifier, AdIter was only evaluated on binary class problems [6]. Figure 3.1 shows the Nemenyi Post-hoc test results with a p-value of 0.05 for: all, binary class, multi class, and evolving (AGR$_{\mathrm{a}}$, AGR$_{\mathrm{g}}$, LED$_{\mathrm{a}}$, LED$_{\mathrm{g}}$, RBF_B$_{\mathrm{m}}$, RBF_B$_{\mathrm{f}}$, RBF$_{\mathrm{m}}$, RBF$_{\mathrm{f}}$) datasets considering accuracy. It further highlights the fact that SGBT$^{MC}$ outperforms other methods on binary and evolving datasets. We investigate each algorithm's performance on evolving data further in figure 3.2 and figure 3.3 by comparing accuracy over time for SGBT$^{MC}$, SRP, ARF, and OSB on AGR$_{\mathrm{g}}$ and LED$_{\mathrm{g}}$. From the figures, it is clearly evident that SGBT$^{MC}$ had the lowest decrease in performance around drift points.

Table 3.3 compares the evaluation time in seconds reported by MOA among single-threaded SGBT$^{MC}$ (SGBT$^{MC}_{ST}$), multi-threaded SGBT$^{MC}$, and SRP. For binary class problems, both SGBT$^{MC}$ variants perform faster than SRP. Maybe FIMT-DD in SGBT$^{MC}$ is a faster base learner than HT in SRP. Compared to SGBT$^{MC}_{ST}$, SRP performs well on multi class problems. However, SGBT$^{MC}$ performed the fastest on multi class problems leveraging parallel processing at training and prediction.

[6]Considering AdIter's weak performance on binary class problems and it's Python implementation, it was not evaluated on multi class problems using MOA *one-vs-rest* wrapper classifier.

### 3.3.1 Multiple Steps and Multi Class Support

Table 3.4: Accuracy: Different variants of SGBT (values rounded to 2 decimals).

| | SGBT | SGBT$^{MC}$ | SGBT$_{MI}$ | SGBT$^{MC}_{MI}$ |
|---|---|---|---|---|
| binary class | | | | |
| AGR$_\mathrm{a}$ | **94.45 ± 0.01** | **94.45 ± 0.01** | 94.30 ± 0.01 | 94.30 ± 0.01 |
| AGR$_\mathrm{g}$ | **91.92 ± 0.01** | **91.92 ± 0.01** | 91.75 ± 0.01 | 91.75 ± 0.01 |
| RBF_B$_\mathrm{m}$ | 91.91 ± 0.19 | 91.85 ± 0.11 | **92.58 ± 0.13** | **92.58 ± 0.13** |
| RBF_B$_\mathrm{f}$ | 84.54 ± 0.67 | 84.12 ± 0.13 | **87.36 ± 0.07** | **87.36 ± 0.07** |
| RandomTree | **85.72 ± 9.40** | **85.72 ± 9.40** | 84.05 ± 8.36 | 84.05 ± 8.36 |
| electricity | 88.54 ± 0.03 | 88.54 ± 0.03 | **90.64 ± 0.06** | **90.64 ± 0.06** |
| airlines | **68.77 ± 0.03** | **68.77 ± 0.03** | 67.85 ± 0.03 | 67.85 ± 0.03 |
| avg | 86.55 | 86.48 | **86.93** | **86.93** |
| rank | **2.21** | 2.50 | 2.64 | 2.64 |
| multi class | | | | |
| LED$_\mathrm{a}$ | 73.96 ± 0.01 | **74.05 ± 0.01** | 73.71 ± 0.01 | 73.99 ± 0.01 |
| LED$_\mathrm{g}$ | 73.22 ± 0.00 | **73.32 ± 0.01** | 72.91 ± 0.01 | 73.18 ± 0.01 |
| RBF$_\mathrm{m}$ | 87.13 ± 0.71 | 87.96 ± 0.63 | 88.18 ± 0.91 | **88.92 ± 0.57** |
| RBF$_\mathrm{f}$ | 75.40 ± 1.84 | 77.03 ± 1.39 | 79.28 ± 1.62 | **81.14 ± 1.19** |
| LED | 73.81 ± 0.17 | 73.81 ± 0.19 | 73.56 ± 0.19 | **73.82 ± 0.18** |
| RBF5 | 88.80 ± 0.91 | 89.76 ± 0.72 | 90.05 ± 0.81 | **90.68 ± 0.58** |
| covtype | 94.31 ± 0.01 | 94.29 ± 0.02 | **95.18 ± 0.02** | 94.73 ± 0.02 |
| avg | 80.95 | 81.46 | 81.84 | **82.35** |
| rank | 3.29 | 2.43 | 2.71 | **1.57** |
| avg (both) | 83.75 | 83.97 | 84.39 | **84.64** |
| rank (both) | 2.75 | 2.46 | 2.68 | **2.11** |

Another study was conducted to understand the performance of different SGBT variants: SGBT, SGBT$^{MC}$, SGBT$_{MI}$, and SGBT$^{MC}_{MI}$. SGBT$^{MC}$ supports multi class problems using binary SGBTs, and SGBT$_{MI}$ employs multiple iterations by *hessian* weights. Both SGBT$^{MC}$ and SGBT$_{MI}$ are orthogonal, so they can be fused to yield SGBT$^{MC}_{MI}$. All SGBT variants used the same hyperparameter configurations as in the previous experiments. Table 3.4 shows the results of the study. Since SGBT$^{MC}$ reverts to SGBT and SGBT$^{MC}_{MI}$ reverts to SGBT$_{MI}$ on binary class problems, if one ignores

$\mathrm{SGBT}_{MI}^{MC}$ and $\mathrm{SGBT}^{MC}$ for binary class problems, SGBT performs well on most of the binary class datasets compared to $\mathrm{SGBT}_{MI}$. However, $\mathrm{SGBT}_{MI}$ has a higher average accuracy for that category. This suggests it performs exceptionally well on certain datasets such as RBF_B$_\mathrm{f}$ and electricity. This results on RBF_B$_\mathrm{f}$, which has fast-evolving drifts, is interesting, as it suggests that multiple training iterations by hessian in $\mathrm{SGBT}_{MI}$ improve SGBT's performance on fast-evolving data. For multi class problems, $\mathrm{SGBT}_{MI}^{MC}$ is the clear winner. When one compares $\mathrm{SGBT}^{MC}$ with SGBT, it is clear that multi class support using binary SGBTs performs better than SGBT with multi class support. On the other hand, multi class results on SGBT and $\mathrm{SGBT}_{MI}$ suggest that multiple iterations by hessian improve SGBT's accuracy on multi class problems. This explains why $\mathrm{SGBT}_{MI}^{MC}$ performs best on multi class problems, as it includes multi class support using binary SGBTs, and multiple iterations by hessian approaches. Overall performance by $\mathrm{SGBT}_{MI}^{MC}$ exceeds the performance of $\mathrm{SGBT}^{MC}$, which is compared against other baselines in table 3.2. But $\mathrm{SGBT}^{MC}$ was used in Table 3.2 evaluation considering its computation efficiency compared to $\mathrm{SGBT}_{MI}^{MC}$. On the other hand, $\mathrm{SGBT}_{MI}^{MC}$ is a good candidate for evolving data stream applications that prioritize predictive performance over computation efficiency.

### 3.3.2 Parameter Exploration

A parameter exploration was conducted to understand the impact of learning rate ($lr$), boosting steps ($S$), weight ($h_i$) transfer methods, percentage of features ($m$), and the independent TR mechanism at each tree via drift detection on $\mathrm{SGBT}^{MC}$'s predictive performance[7]. Furthermore, another study was conducted to understand the effect of one-hot encoding on the predictive performance of $\mathrm{SGBT}^{MC}$ with FIMT-DD. The results for all these analyses are shown in tables 3.5, 3.6, 3.7, 3.8, and 3.9.

[7]Experimental results for different loss functions effect (categorical cross-entropy and squared) on $\mathrm{SGBT}^{MC}$'s predictive performance are explained in section A.2

Table 3.5: Test then train accuracy of SGBT$^{MC}$ [$S$ = 100, $m$ = 75, $lr$ ={6.25e-3, 1.25e-2, 2.50e-2}, FIMT-DD] for different learning rates ($lr$) (values rounded to 2 decimals, 4 decimals were considered to select the winner).

| | 6.25e-3 | 1.25e-2 | 2.50e-2 |
|---|---|---|---|
| binary class | | | |
| AGR$_a$ | 94.42+0.00 | **94.45+0.01** | 94.44+0.01 |
| AGR$_g$ | 91.90+0.01 | **91.92+0.01** | 91.91+0.01 |
| RBF_B$_m$ | 91.69+0.09 | 91.85+0.11 | **92.04+0.15** |
| RBF_B$_f$ | 83.42+0.39 | 84.12+0.13 | **84.82+0.05** |
| RandomTree | 85.14+9.96 | 85.72+9.40 | **86.36+9.10** |
| electricity | 87.90+0.07 | 88.54+0.03 | **89.27+0.04** |
| airlines | 68.70+0.00 | 68.77+0.03 | **68.83+0.02** |
| avg | 86.17 | 86.48 | **86.81** |
| rank | 3.00 | 1.71 | **1.29** |
| multi class | | | |
| LED$_a$ | 74.04+0.01 | **74.05+0.01** | 74.02+0.01 |
| LED$_g$ | 73.32+0.02 | 73.32+0.01 | **73.33+0.01** |
| RBF$_m$ | 87.70+0.64 | 87.96+0.63 | **88.24+0.64** |
| RBF$_f$ | 76.12+1.45 | 77.03+1.39 | **78.05+1.38** |
| LED | **73.84+0.22** | 73.81+0.19 | 73.79+0.22 |
| RBF5 | 89.58+0.70 | 89.76+0.72 | **89.97+0.78** |
| covtype | 93.86+0.02 | 94.31+0.02 | **94.78+0.01** |
| avg | 81.21 | 81.46 | **81.74** |
| rank | 2.57 | 1.86 | **1.57** |
| avg (both) | 83.69 | 83.97 | **84.27** |
| rank (both) | 2.79 | 1.79 | **1.43** |

Three learning rates: 6.25e-3, 1.25e-2, and 2.50e-2, were used in the study to understand the effect of learning rate ($lr$) on SGBT$^{MC}$'s performance. All the other configurations: FIMT-DD base learner, 75% of features($m$), and

100 boosting steps($S$) were kept unchanged. As per table 3.5, considering SGBT$^{MC}$ [$S$ = 100, $m$ = 75, $lr$ ={6.25e-3, 1.25e-2, 2.50e-2}, FIMT-DD] configurations, in general, larger learning rates ($lr$) seem to favour both binary and multi class problems.

Table 3.6: Test then train accuracy of SGBT$^{MC}$ [$S = 20, 40, 60, 80, 100$, $m =$ 75, $lr$ =1.25e-2, FIMT-DD] for different boosting steps ($S$) (values rounded to 2 decimals, 4 decimals were considered to select the winner).

| | 20 | 40 | 60 | 80 | 100 |
|---|---|---|---|---|---|
| binary class | | | | | |
| AGR$_a$ | 94.08+0.01 | 94.42+0.01 | 94.42+0.01 | **94.45+0.01** | 94.45+0.01 |
| AGR$_g$ | 91.51+0.04 | 91.87+0.02 | 91.88+0.01 | **91.93+0.01** | 91.92+0.01 |
| RBF_B$_m$ | 90.60+0.22 | 91.31+0.15 | 91.59+0.14 | 91.74+0.13 | **91.85+0.11** |
| RBF_B$_f$ | 80.45+0.14 | 82.32+0.10 | 83.14+0.16 | 83.71+0.13 | **84.12+0.13** |
| RandomTree | 84.35+10.61 | 84.49+11.00 | 85.27+9.67 | 85.10+9.86 | **85.72+9.40** |
| electricity | 87.01+0.17 | 87.68+0.05 | 87.96+0.07 | 88.36+0.03 | **88.54+0.03** |
| airlines | 68.01+0.06 | 68.51+0.01 | 68.63+0.02 | 68.71+0.03 | **68.77+0.03** |
| avg | 85.14 | 85.80 | 86.13 | 86.29 | **86.48** |
| rank | 5.00 | 4.00 | 2.86 | 1.86 | **1.29** |
| multi class | | | | | |
| LED$_a$ | 74.00+0.00 | 73.96+0.00 | 74.02+0.01 | 74.04+0.02 | **74.05+0.01** |
| LED$_g$ | 73.23+0.02 | 73.26+0.01 | 73.29+0.01 | 73.29+0.01 | **73.32+0.01** |
| RBF$_m$ | 86.42+0.72 | 87.25+0.70 | 87.59+0.65 | 87.81+0.65 | **87.96+0.63** |
| RBF$_f$ | 72.39+1.46 | 74.75+1.49 | 75.78+1.44 | 76.54+1.44 | **77.03+1.39** |
| LED | 73.81+0.21 | 73.82+0.22 | 73.83+0.16 | **73.84+0.17** | 73.81+0.19 |
| RBF5 | 89.16+0.60 | 89.49+0.65 | 89.59+0.66 | 89.67+0.74 | **89.76+0.72** |
| covtype | 93.17+0.01 | 93.61+0.02 | 93.92+0.01 | 94.07+0.06 | **94.31+0.02** |
| avg | 80.31 | 80.88 | 81.15 | 81.32 | **81.46** |
| rank | 4.86 | 4.00 | 2.86 | 1.86 | **1.43** |
| avg (both) | 82.73 | 83.34 | 83.64 | 83.80 | **83.97** |
| rank (both) | 4.93 | 4.00 | 2.86 | 1.86 | **1.36** |

In a separate study to understand the effect of boosting steps on SGBT$^{MC}$'s performance, five boosting steps (20, 40, 60, 80, 100) were considered. In this study, base learner (FIMT-DD), feature percentage ($m$=75%), and learning rate ($lr$=1.25e-2) were kept unchanged. According to table 3.6, when considering SGBT$^{MC}$ [$S = 20, 40, 60, 80, 100$, $m = 75$, $lr$ =1.25e-2, FIMT-DD] configurations, 100 boosting steps yields better results than the smaller boosting steps for both binary and multi class problems. This aligns with OSB results in [15], where more boosting iterations performed better than fewer boosting iterations.

In another study to analyze the impact of varying feature percentages ($m$) on SGBT$^{MC}$'s performance, except $m$, all the other SGBT$^{MC}$ configurations, including the base learner (FIMT-DD), learning rate ($lr$=1.25e-2), and boosting steps ($S$=100) were kept constant. According to table 3.7, among SGBT$^{MC}$ [$S = 100$, $m = 45, 60, 75, 100$, $lr$ =1.25e-2] configurations, 75% of features yield good accuracy on most datasets. Not having 100% of the features helps to increase the diversity of the ensemble, which avoids overfitting to data. These results match [14, 15] findings where ARF and SRP perform best with 60% of the features.

A separate study examines the effect of independent TR mechanisms by each base learner on SGBT$^{MC}$'s performance. For this study, SGT was selected as the base learner since FIMT-DD has a built-in TR mechanism. Hence a generic regressor with an inbuilt TR mechanism based on DDM's *warning* and *out-of-control* signals was introduced into MOA. This allows us to enable or disable the underlying TR strategy using a generic regressor with SGT and DDM or just using SGT. The DDM settings were: *minimum number of instances before permitting a change detection* = 250, *warning level* = 2.0, and *out-of-control level* = 2.5. SGT used the same default configurations used in [82]. From table 3.8 results, one can see that having an internal TR mechanism often improves performance. Also, all the SGBT$^{MC}$ configurations with SGT perform poorly on RBF$_{\mathrm{f}}$. Maybe SGT's default *warmStart* (number of instances used to estimate bin boundaries for numeric values) 1000

Table 3.7: Test then train accuracy of SGBT$^{MC}$ [$S = 100$, $m = 45, 60, 75, 100$, $lr$ =1.25e-2, FIMT-DD] for different feature percentages ($m$) (values rounded to 2 decimals, 4 decimals were considered to select the winner).

| | 45 | 60 | 75 | 100 |
|---|---|---|---|---|
| binary class | | | | |
| $\mathrm{AGR_a}$ | 92.76+0.01 | 93.73+0.01 | **94.45+0.01** | 94.40+0.00 |
| $\mathrm{AGR_g}$ | 90.29+0.02 | 91.20+0.00 | 91.92+0.01 | **91.97+0.01** |
| $\mathrm{RBF\_B_m}$ | 90.96+0.00 | 91.48+0.04 | **91.85+0.11** | 91.66+0.20 |
| $\mathrm{RBF\_B_f}$ | 82.75+0.41 | 83.54+0.25 | **84.12+0.13** | 83.98+0.23 |
| RandomTree | 81.89+5.95 | 83.99+6.63 | **85.72+9.40** | 81.38+14.50 |
| electricity | 88.42+0.09 | **88.57+0.06** | 88.54+0.03 | 88.06+0.00 |
| airlines | **68.85+0.03** | 68.83+0.04 | 68.77+0.03 | 68.47+0.00 |
| avg | 85.13 | 85.91 | **86.48** | 85.70 |
| rank | 3.29 | 2.43 | **1.57** | 2.71 |
| multi class | | | | |
| $\mathrm{LED_a}$ | 73.87+0.00 | 73.99+0.01 | **74.05+0.01** | 73.98+0.01 |
| $\mathrm{LED_g}$ | 72.91+0.01 | 73.26+0.00 | **73.32+0.01** | 73.27+0.01 |
| $\mathrm{RBF_m}$ | 86.88+0.65 | 87.53+0.64 | **87.96+0.63** | 87.81+0.66 |
| $\mathrm{RBF_f}$ | 75.64+1.52 | 76.47+1.48 | **77.03+1.39** | 76.81+1.38 |
| LED | 73.80+0.20 | **73.89+0.25** | 73.81+0.19 | 73.56+0.16 |
| RBF5 | 89.24+0.34 | **89.84+0.55** | 89.76+0.72 | 85.92+1.23 |
| covtype | 94.29+0.07 | **94.39+0.04** | 94.31+0.02 | 93.85+0.00 |
| avg | 80.95 | 81.34 | **81.46** | 80.74 |
| rank | 3.57 | 2.00 | **1.43** | 3.00 |
| avg (both) | 83.04 | 83.62 | **83.97** | 83.22 |
| rank (both) | 3.43 | 2.21 | **1.50** | 2.86 |

is too large for $\mathrm{RBF_f}$ with fast-moving drifts.

The following study examines the impact of one-hot encoding on the predictive performance of SGBT$^{MC}$ when used in conjunction with FIMT-DD. In this study, all nominal features within a specified ($m$) percentage of selected

Table 3.8: Test then train accuracy of SGBT$^{MC}$ [$S = 100$, $m = 75$, $lr$ =1.25e-2, SGT] for different Tree Replacement (TR) mechanisms (values rounded to 2 decimals, 4 decimals were considered to select the winner).

| | TR via DDM | no TR |
|---|---|---|
| binary class | | |
| AGR$_a$ | **93.86+0.00** | 91.40+0.02 |
| AGR$_g$ | **91.60+0.00** | 86.39+0.11 |
| RBF_B$_m$ | **78.85+0.92** | 59.77+1.20 |
| RBF_B$_f$ | **57.41+1.67** | 57.25+1.50 |
| RandomTree | 86.07+2.15 | **87.08+2.08** |
| electricity | **76.83+0.08** | 73.41+0.22 |
| airlines | **64.70+0.05** | 62.94+0.26 |
| avg | **78.48** | 74.03 |
| rank | **1.14** | 1.86 |
| multi class | | |
| LED$_a$ | **73.68+0.00** | 71.73+0.00 |
| LED$_g$ | **72.85+0.01** | 71.42+0.01 |
| RBF$_m$ | **66.86+1.27** | 36.64+0.65 |
| RBF$_f$ | **28.57+1.50** | 27.79+2.44 |
| LED | **72.67+0.16** | 72.67+0.33 |
| RBF5 | 82.01+0.51 | **83.20+0.39** |
| covtype | **83.08+0.02** | 69.78+4.71 |
| avg | **68.53** | 61.89 |
| rank | **1.14** | 1.86 |
| avg (both) | **73.50** | 67.96 |
| rank (both) | **1.14** | 1.86 |

features are one-hot encoded before being passed to FIMT-DD. The results presented in table 3.9 demonstrate that, except for RandomTree, one-hot encoding does not appear to enhance accuracy for binary class problems. Only for RandomTree, one-hot encoding improves accuracy and reduces the stan-

Table 3.9: Effect of one-hot encoding on test then train accuracy of SGBT$^{MC}$ [$S$ = 100, $m$ = 75, $lr$ =1.25e-2, FIMT-DD] (values rounded to 2 decimals, 4 decimals were considered to select the winner).

| Encoding | one-hot | no one-hot* |
|---|---|---|
| binary class | | |
| AGR$_a$ | 94.41+0.00 | **94.45+0.01** |
| AGR$_g$ | 91.89+0.01 | **91.92+0.01** |
| RBF_B$_m$ | **91.85+0.11** | **91.85+0.11** |
| RBF_B$_f$ | **84.12+0.13** | **84.12+0.13** |
| RandomTree | **87.84+7.33** | 85.72+9.40 |
| electricity | 88.44+0.08 | **88.54+0.03** |
| airlines | 67.93+0.02 | **68.77+0.03** |
| avg | **86.64** | 86.48 |
| rank | 1.71 | **1.29** |
| multi class | | |
| LED$_a$ | **74.05+0.01** | **74.05+0.01** |
| LED$_g$ | **73.32+0.01** | **73.32+0.01** |
| RBF$_m$ | **87.96+0.63** | **87.96+0.63** |
| RBF$_m$ | **77.03+1.39** | **77.03+1.39** |
| LED | **73.81+0.19** | **73.81+0.19** |
| RBF5 | **89.76+0.72** | **89.76+0.72** |
| covtype | **94.31+0.02** | **94.31+0.02** |
| avg | **81.46** | **81.46** |
| rank | **1.50** | **1.50** |
| avg (both) | **84.05** | 83.97 |
| rank (both) | 1.61 | **1.39** |

*: FIMT-DD use value of the index for nominal features.

dard deviation. Interestingly, passing the index value to the split criterion for nominal attributes seem to encode dependencies between different values for a given feature. For example, in the electricity dataset, the index values for 'day' (only nominal feature) contain weekend information. However, none of the multi-class datasets analyzed in this study included any nominal features. As a result, there was no significant difference between using one-hot encoding versus passing value index for multi-class problems in this particular study.

### 3.3.3 Skip training on instances

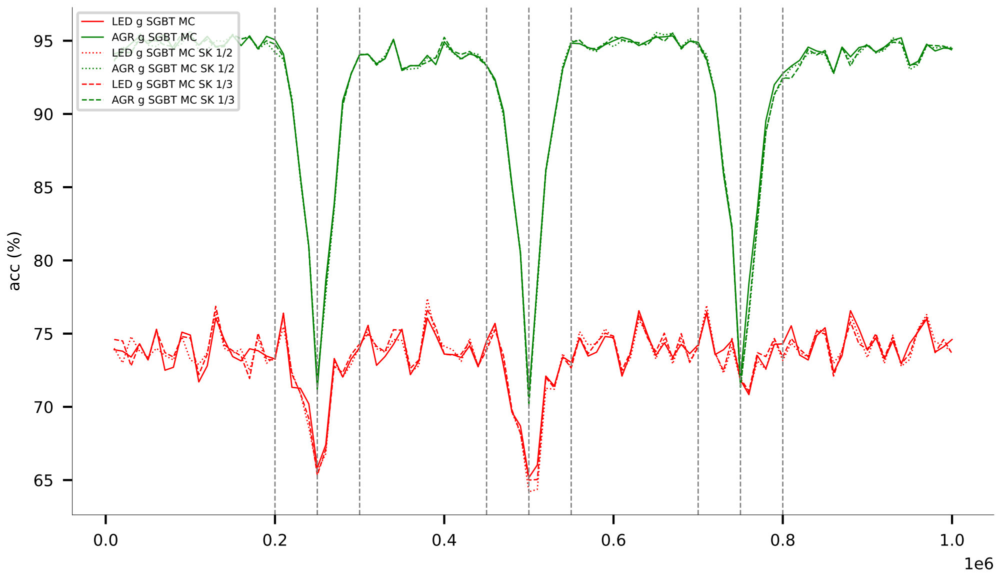


Figure 3.4: Accuracy over time: Different SGBT$^{MC}$ versions on AGR$_g$ and LED$_g$. x axis is the number of instances seen so far. Vertical dotted lines mark a concept drift's start, center, and end. SGBT$^{MC}_{SK\_1/k}$[$S = 100$, $m = 75$, $lr$ =1.25e-2].

Another study was conducted using SGBT$^{MC}_{SK\_1/k}$[$S = 100$, $m = 75$, $lr$ =1.25e-2] with different $k$ values to understand the effect of random skip training. Here $k$ was set to 1, 2, and 3 so that SGBT$^{MC}_{SK\_1/k}$ would not skip, skipping 1/2 and 1/3 of instances. As per table 3.10, apart from RBF_B$_f$ and RBF$_f$ SGBT$^{MC}_{SK\_1/3}$, produced good results even with 1/3-rd of instances skipped. Here, slight poor accuracy in those two datasets may be because both RBF_B$_f$ and RBF$_f$ have fast-moving drifts. On the other hand, 1/3-rd of skipping ac-

Table 3.10: Accuracy and evaluation time(s) of $\text{SGBT}^{MC}_{SK\_1/k}[S = 100, m = 75, lr = 1.25\text{e-}2]$

| | Accuracy (%) | | | Time (s) | | |
|---|---|---|---|---|---|---|
| | $\text{SGBT}^{MC}_{SK\_1/k}$ | | $\text{SGBT}^{MC}$ | $\text{SGBT}^{MC}_{SK\_1/k}$ | | $\text{SGBT}^{MC}$ |
| $k$ | 2 (skip 1/2) | 3 (skip 1/3) | 1 (no skip) | 2 (skip 1/2) | 3 (skip 1/3) | 1 (no skip) |
| binary class | | | | | | |
| $\text{AGR}_\text{a}$ | 94.32 | 94.36 | **94.45** | **719.16** | 818.16 | 1386.55 |
| $\text{AGR}_\text{g}$ | 91.81 | 91.83 | **91.92** | **663.56** | 810.28 | 1359.81 |
| $\text{RBF\_B}_\text{m}$ | 90.55 | 91.20 | **91.85** | **983.85** | 1218.42 | 2126.22 |
| $\text{RBF\_B}_\text{f}$ | 78.11 | 80.89 | **84.12** | **891.39** | 1061.86 | 1749 |
| RandomTree | 84.73 | 85.62 | **85.72** | **74.62** | 93.09 | 148.69 |
| electricity | 85.88 | 87.10 | **88.54** | **31.43** | 37.43 | 54.75 |
| airlines | 67.99 | 68.31 | **68.77** | **309.32** | 378.3 | 584.59 |
| avg | 84.77 | 85.62 | **86.48** | **524.76** | 631.07 | 1058.52 |
| rank | 3.00 | 2.00 | **1.00** | **1** | 2 | 3 |
| multi class | | | | | | |
| $\text{LED}_\text{a}$ | 73.91 | 73.97 | **74.05** | **1002.87** | 1197.72 | 1747.52 |
| $\text{LED}_\text{g}$ | 73.18 | 73.22 | **73.32** | **977.22** | 1217.96 | 1718.98 |
| $\text{RBF}_\text{m}$ | 86.17 | 87.01 | **87.96** | **1441.83** | 1875.82 | 2773.24 |
| $\text{RBF}_\text{f}$ | 68.75 | 72.62 | **77.03** | **1395** | 1683.68 | 2439.11 |
| LED | 73.80 | 73.76 | **73.81** | **91.99** | 121.11 | 162.38 |
| RBF5 | 88.51 | 89.06 | **89.76** | **153.97** | 201.3 | 279.21 |
| covtype | 92.23 | 93.15 | **94.28** | **1005.95** | 1318.55 | 1944.42 |
| avg | 79.51 | 80.40 | **81.46** | **866.97** | 1088.02 | 1580.69 |
| rank | 2.86 | 2.14 | **1.00** | **1** | 2 | 3 |
| avg (both) | 82.14 | 83.01 | **83.97** | **695.87** | 859.55 | 1319.61 |
| rank (both) | 2.93 | 2.07 | **1.00** | **1** | 2 | 3 |

Standard deviations are available in table A.2.

tually helps on RandomTree dataset. This could be due to simple tree-like rules used to generate this dataset, and it does not have any drifts. So random skipping of some instances may have helped avoid overfitting the model to the data.

To further illustrate the influence of random skipping, another study was conducted using $\text{SGBT}^{MC}_{SK\_1/k}[S = 100,\ m = 75,\ lr = 1.25\text{e-}2]$ with different $k$ values: 1, 2, 3 on $\text{AGR}_\text{g}$ and $\text{LED}_\text{g}$ datasets. The idea here is to understand

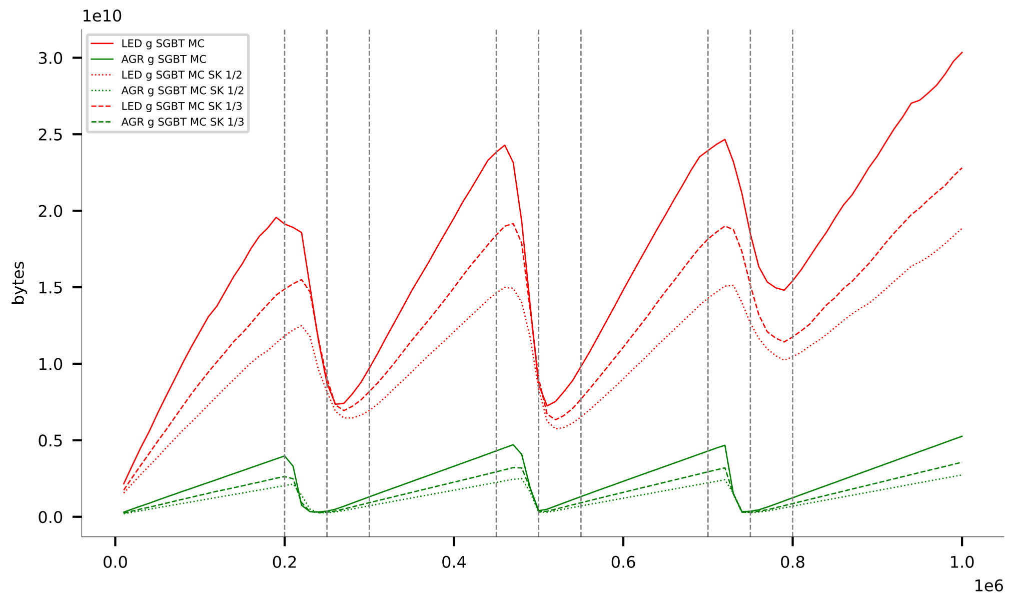


Figure 3.5: Model size over time: Different SGBT$^{MC}$ versions on AGR$_\text{g}$ and LED$_\text{g}$. x axis is the number of instances seen so far. Vertical dotted lines mark a concept drift's start, center, and end. SGBT$^{MC}_{SK\_1/k}$[$S$ = 100, $m$ = 75, $lr$ =1.25e-2].

the effect of skip training instances on SGBT$^{MC}_{SK\_1/k}$'s performance for binary and multi class problems. Both AGR$_\text{g}$ and LED$_\text{g}$ had drifts happening at the same time intervals. However, AGR$_\text{g}$ is a binary problem, and LED$_\text{g}$ is a multi class problem with 10 classes. Accuracy and model size statistics were collected every 10000 instances. When one considers the classification accuracy in figure 3.4, skipping instances for training does not significantly hinder the accuracy on both AGR$_\text{g}$ and LED$_\text{g}$. On the other hand, skipping instances result in significant memory savings on both datasets in figure 3.5. These savings are much more prevalent in LED$_\text{g}$ as SGBT$^{MC}_{SK\_1/k}$ needs 10 trees per boosting step compared to 1 tree for AGR$_\text{g}$.

# Chapter 4

# Continuously Adaptive Neural Networks for Data Streams

In the previous chapter, we discussed a novel streaming gradient-boosted tree classifier ideal for binary classification and low-dimensional data. While Neural Networks have proven to excel in high-dimensional data domains when operating in batch settings, their effectiveness in SL is hindered by the difficulties of hyperparameter optimization and large batch training. This chapter presents a novel approach that enables Neural Networks to be applied to diverse, continuously evolving streaming datasets overcoming these challenges.

## 4.1 Introduction

Neural Networks have greatly succeeded in high-dimensional data fields such as image classification and natural language processing (NLP). NNs need larger datasets and multiple iterations of epoch training to learn effectively. Furthermore, NNs can be susceptible to hyperparameters and need to be re-trained to produce a good model after a concept drift.

Hence, to reap the benefits of NNs for data streams, it is required to efficiently train and test NNs without any hyperparameter tuning, where predictive NNs are resilient and adaptive to concept drifts. This work addresses this issue by employing an efficient per-instance training of a pool of Multi-Layer

Perceptron (MLP)s, and choosing the best MLP according to its estimated loss for every prediction.

The main contributions of this work are the following:

1. **C**ontinuously **A**daptive **N**eural Networks for **D**ata Streams (CAND): We introduce training a pool of NNs per instance and selecting the best NN considering their estimated loss for prediction to overcome the NN hyperparameter selection for evolving data streams.

2. We propose two novel orthogonal approaches 1) selecting a sub-pool for training, and 2) skipping backpropagation below a certain threshold of the loss, to increase the training efficiency of CAND without compromising accuracy too much.

3. We compare CAND against three current state-of-the-art stream learning methods: ADL, ARF and SRP on 17 datasets with low-dimensional, evolving, and high-dimensional data. The results give a clear overview of the performance and resource usage of the compared methods. Also, an extensive empirical analysis is done to understand the effect of mini-batch size, sub-pool size, and skip backpropagation threshold on CAND's predictive and computing performance. Furthermore, an in-depth investigation is done on one of CAND's efficient variants to understand its NN selection behaviour under multiple concept drifts.

In our method, we employ per-instance NN and mini-batch training in contrast to the recent literature focusing mainly on mini-batch training [19]. Per-instance training allows the model to dynamically adapt to concept drifts with as little delay as possible. One drawback of per-instance training is that it is less efficient than mini-batch training, as mini-batch training makes better use of the computational resources. In this work, we also explore the possibility of using mini-batch training without compromising too much accuracy. An extensive empirical study is done to understand the effect of mini-batch size on accuracy and wall-time. To further improve the computational performance, we propose two orthogonal approaches: training a sub-pool of NNs at a given moment and skipping backpropagation for some instances. Those two approaches are orthogonal, so they could be combined for further run-time savings.

## 4.2 Continuously Adaptive Neural Networks for Data Streams (CAND)

We propose **C**ontinuously **A**daptive **N**eural Networks for **D**ata Streams (CAND), which trains a pool of MLPs to overcome the hyperparameter selection issue on NNs for evolving data streams. The MLPs are tested and trained per instance (i.e. incrementally). Incoming instances are pre-processed in an online

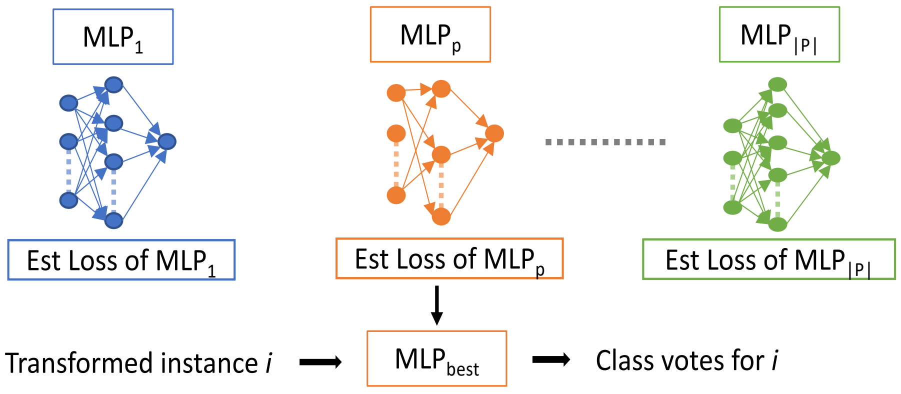


Figure 4.1: CAND prediction Algorithm 3

fashion (normalized and one-hot encoded) before being fed into the MLPs. For efficiency, two orthogonal variants of CAND are proposed: 1) CAND with a selected smaller pool for per-instance training and 2) CAND which skips backpropagation in some instances. The following sections explain the CAND algorithm and its two orthogonal variants in detail.

Online normalization used in this work was inspired by [81] work, where the sum of the values and the sum of squared values are calculated for each attribute per instance and used to compute the standard deviation and mean for a given attribute. Those point-wise standard deviations and means are used to normalize the incoming $i^{th}$ instance's attributes. As no fading factor is used in the normalization, most recent instances have the same weight as past instances. The nominal attributes are one-hot encoded per instance. Assuming a test-then-train setting, instance $i$ can be transformed via the above-mentioned online normalization and one-hot encoding scheme for testing and later be used for training.

### 4.2.1 Training a pool of MLPs and using it for Prediction

CAND's prediction algorithm is quite simple. For a given instance $i$, CAND chooses MLP$_{best}$ considering the estimated loss of each MLP in the $P$ pool. This MLP$_{best}$ is used to predict the class label for transformed $i$ before training. Algorithm 3, along with figure 4.1 explains the method CAND use to find the MLP$_{best}$ using estimated loss to predict the class label for transformed $i$. The loss estimation is done using the ADWIN estimator. ADWIN adapts

**Algorithm 3** CAND PREDICT

```
Input: instance $i$, pool $P$ of MLPs
1: $MLP_{best} = \arg\min_{p \in P}$ estimated loss at $i$ (excluding loss for $i$)
Output: PREDICT($i$, MLP$_{best}$)
```

the size of its data windows very efficiently by using exponential histograms according to the underlying data distribution changes, discarding older parts of the data window that relate to the old distribution. This allows it to adapt to distribution changes dynamically. A distribution change in the MLP's loss function is assumed to be due to a concept drift in the incoming data. This allows us to avoid a fading factor in the normalization process as ADWIN discards the loss related to the old distribution.

For the same instance $i$, all the MLPs in the $P$ pool are trained in parallel using the transformed instance $i$. Before training, each NN updates its estimated loss using the loss for instance $i$. The $P$ pool is defined so that it

**Algorithm 4** CAND$_{sub}^{SB}$ TRAIN

```
Input: instance $i$, pool $P$ of MLPs, number of instances for warm-up $W$, $|M|$
    pool size, skip backpropagation threshold $b$.
 1: if the current instance count $< W$ then
 2:     $M$=$P$
 3: else
 4:     $M$={Half of $M$ is selected from $P$ by lowest estimated loss, other half
    random}
 5: end if
 6: for all $m \in M$ do
 7:     TRAIN(i,m)                          ▷ invoked in $|M|$ separate threads
 8:                               ▷ will skip backpropagation, if the loss $< b$
 9:                                       ▷ will update loss estimate for $m$
10: end for
11: Wait for all $|M|$ training threads to finish.
```

consists of MLPs with different optimizers, learning rates, and hidden layer widths. This allows the prediction algorithm to choose the best MLP for the current data distribution from a diverse pool of MLPs.

### 4.2.2 Enhancements to CAND's Training

Training a bigger $P$ pool with complex NNs may not be computationally efficient. CAND$_{sub}$ is proposed to overcome this. CAND with these enhancements is explained in algorithm 4, where a smaller $M(\subset P)$ pool is chosen for per-instance training. This $M$ pool is chosen such that the currently best-performing MLPs represent half of it, and the other half is randomly selected from the remaining MLPs. This approach allows more performant MLPs to be trained more frequently while also allowing all the other MLPs to be trained at least occasionally. Thus even low-performance MLPs can improve over time and become more dominant, especially after concept drifts. Here, the performance of an MLP is assessed considering its estimated loss at that time. In CAND$_{sub}$, there is a warm-up period where all the MLPs in the $P$ pool are trained. This allows all the MLPs to have a reasonable estimated loss after the warm-up. In our experiments, this warm-up period is either 1% of the dataset or 1000 instances, whichever is smaller. If $M$=$P$ for all the dataset instances, then CAND$_{sub}$ reverts to CAND.

The backpropagation procedure takes more computing resources than the forward pass of the network. Avoiding backpropagation can lead to more significant savings in computing resources. As an orthogonal efficient variant of CAND, backpropagation of the loss is skipped for instances where their loss falls below a pre-specified threshold. In the initial experiments, this threshold was determined considering empirical results on low-dimensional datasets. Later, an in-depth empirical investigation is done to understand this effect further. This orthogonal variant is identified as CAND$^{SB}$ in the rest of the thesis. As both CAND$_{sub}$ and CAND$^{SB}$ are orthogonal, one can combine those two CAND variants for greater savings in computing resources. This combined CAND variant is identified as CAND$^{SB}_{sub}$ in the rest of the thesis.

The time complexity of CAND is linear in the size of $M$: $\mathcal{O}(|M|)$. The memory complexity is linear in the size of $P$: $\mathcal{O}(|P|)$.

## 4.3 Experiments

In the experiments, multiple variants of CAND were compared against the streaming NN method ADL and two state-of-the-art stream learning methods: ARF and SRP, on 17 datasets in various sets of experiments. First, we compare CAND against ADL and different configurations of ARF and SRP using 10 learners and 30 learners. Second, we compare the best possible baseline configurations from the first experiment against different variants of CAND. The second experiment aims to understand the computational cost associated with each model. To clarify the effect of mini-batch size and GPU

Table 4.1: Dataset properties and data type: (**L**)ow dimensional, has (**D**)rifts, (**H**)igh dimensional, (**R**)eal world, (**S**)ynthetic, dense($^d$), sparse($^s$). * 1.00E-04.

| name | type | instances | features | | # | class distribution | |
|---|---|---|---|---|---|---|---|
| | | | before one-hot | after one-hot | cla-sses | max(%) | min(%) |
| airlines | LR$^d$ | 539382 | 7 | 614 | 2 | 55.46 | 44.54 |
| electricity | LR$^d$ | 45310 | 8 | 14 | 2 | 57.55 | 42.45 |
| kdd99 | LR$^d$ | 4898430 | 41 | 122 | 23 | 56.24 | 0.00* |
| WISDM | LR$^d$ | 5417 | 45 | 80 | 6 | 38.43 | 4.53 |
| covtype | LR$^d$ | 581010 | 54 | 54 | 7 | 48.76 | 0.47 |
| nomao | LR$^d$ | 34464 | 118 | 172 | 2 | 71.44 | 28.56 |
| AGR$_a$ | LDS$^d$ | 1000000 | 9 | 40 | 2 | 52.83 | 47.17 |
| AGR$_g$ | LDS$^d$ | 1000000 | 9 | 40 | 2 | 52.83 | 47.17 |
| RBF$_f$ | LDS$^d$ | 1000000 | 10 | 10 | 5 | 30.01 | 9.27 |
| RBF$_m$ | LDS$^d$ | 1000000 | 10 | 10 | 5 | 30.01 | 9.27 |
| LED$_a$ | LDS$^s$ | 1000000 | 24 | 24 | 10 | 10.08 | 9.94 |
| LED$_g$ | LDS$^s$ | 1000000 | 24 | 24 | 10 | 10.08 | 9.94 |
| epsilon | HR$^d$ | 100000 | 2000 | 2000 | 2 | 50.05 | 49.95 |
| SVHN | HR$^d$ | 26032 | 3072 | 3072 | 10 | 19.59 | 6.13 |
| gisette | HR$^d$ | 6000 | 5000 | 5000 | 2 | 50.00 | 50.00 |
| spam | HR$^s$ | 9323 | 39916 | 39916 | 2 | 74.40 | 25.60 |
| sector | HR$^s$ | 6412 | 55197 | 55197 | 105 | 1.25 | 0.14 |

training on CAND, the third set of experiments compare per-instance-trained CAND against GPU-trained CAND with different mini-batch sizes. Forth and the fifth set of experiments were performed to identify the effects of smaller pools size ($|M|$) and backpropagation skip threshold on CAND's accuracy and wall-time. The sixth set of experiments compares the selected efficient versions of CAND against per-instance trained CAND.

The datasets included low-dimensional($<$ 2000 features) and high-dimensional($\geq$ 2000 features) data. Except for WISDM [89][1], all the other low-dimensional ones were used in [15] as classification benchmarks. The synthetic data streams

[1]available at: https://www.cis.fordham.edu/wisdm/dataset.php

Table 4.2: CAND($|P|$=10) against ADL and ensemble learners with 10 base learners. * hypothetical MLP selection criteria. ! single best MLP for the given dataset. # at least one MLP predicted the correct label.

| dataset | ADL | Ensemble learners | | | | CAND$\|P\|$=10 | | | |
|---|---|---|---|---|---|---|---|---|---|
| dataset | | ARF 10, 60% | SRP 10, 60% | ARF 10, 10% | SRP 10, 10% | Min Estd Loss | Majority Vote | Best MLP*! | At Least One*# |
| airlines | 61.06 | 65.86 | **66.74** | 61.18 | 64.90 | 61.14 | 61.24 | 61.14 | 83.27 |
| electricity | 74.20 | **89.87** | 89.14 | 58.00 | 83.47 | 84.98 | 82.48 | 85.24 | 95.59 |
| kdd99 | 99.96 | 99.96 | **99.97** | 99.94 | **99.97** | 99.92 | 99.91 | 99.91 | 99.97 |
| WISDM | 56.37 | 85.28 | **85.36** | 77.77 | 83.54 | 72.96 | 71.71 | 74.35 | 90.70 |
| covtype | 87.91 | 94.49 | **95.28** | 85.34 | 89.46 | 88.91 | 86.01 | 89.90 | 97.41 |
| nomao | **97.58** | 97.08 | 97.23 | 97.00 | 97.06 | 97.02 | 96.96 | 97.27 | 98.99 |
| Avg Acc | 79.51 | 88.76 | **88.95** | 79.87 | 86.40 | 84.16 | 83.05 | 84.64 | 94.32 |
| Avg Rank | 4.92 | 2.25 | **1.42** | 5.67 | 3.08 | 4.83 | 5.83 | | |
| AGR$_{a}$ | 63.56 | 85.91 | **92.44** | 64.45 | 77.21 | 89.59 | 87.34 | 89.40 | 97.66 |
| AGR$_{g}$ | 61.30 | 79.99 | **87.55** | 62.98 | 76.19 | 87.14 | 85.20 | 86.98 | 96.60 |
| RBF$_{f}$ | 31.93 | **71.20** | 70.84 | 30.01 | 46.88 | 67.26 | 64.76 | 67.67 | 88.43 |
| RBF$_{m}$ | 46.26 | 84.70 | 83.19 | 30.01 | 62.07 | **85.75** | 84.75 | 85.62 | 95.91 |
| LED$_{a}$ | 73.66 | 73.92 | 73.49 | 56.24 | 37.24 | **74.02** | 73.91 | 73.89 | 84.38 |
| LED$_{g}$ | 73.02 | 73.06 | 72.73 | 57.80 | 38.09 | **73.28** | 73.22 | 73.23 | 83.76 |
| Avg Acc | 58.29 | 78.13 | **80.04** | 50.25 | 56.28 | 79.51 | 78.20 | 79.47 | 91.12 |
| Avg Rank | 5.67 | 2.83 | 3.00 | 6.33 | 5.67 | **1.67** | 2.83 | | |
| epsilon | 77.54 | 50.26 | 50.21 | 58.06 | 55.87 | **85.89** | 79.69 | 85.73 | 96.41 |
| SVHN | 22.59 | 19.52 | 20.35 | 20.57 | 22.25 | **55.68** | 38.58 | 55.94 | 76.02 |
| gisette | 76.27 | 82.36 | 82.17 | 88.04 | 89.64 | **96.23** | 93.74 | 96.25 | 99.48 |
| spam | **98.34** | 96.12 | 96.22 | 96.87 | 96.80 | 97.99 | 97.56 | 98.13 | 99.74 |
| sector | 0.25 | 0.74 | 0.80 | 4.39 | 12.73 | **67.13** | 46.84 | 66.21 | 74.47 |
| Avg Acc | 55.00 | 49.80 | 49.95 | 53.59 | 55.46 | **80.58** | 71.28 | 80.45 | 89.22 |
| Avg Rank | 4.20 | 6.20 | 6.00 | 4.20 | 4.00 | **1.20** | 2.20 | | |
| Overall Acc | 64.81 | 73.55 | 74.34 | 61.69 | 66.67 | **81.46** | 77.88 | 81.58 | 91.69 |
| Overall Rank | 4.97 | 3.62 | 3.32 | 5.47 | 4.26 | **2.65** | 3.71 | | |

simulate different types of concept drifts, i.e. abrupt (AGR$_{a}$, LED$_{a}$), gradual (AGR$_{g}$, LED$_{g}$), fast incremental changes (RBF$_{f}$), and moderate incremen-

Table 4.3: CAND($|P|$=30) against ensemble learners with 30 base learners. $^*$ hypothetical MLP selection criteria. $^!$ single best MLP for the given dataset. $^\#$ at least one MLP predicted the correct label. $\underline{\textit{underline}}$ value is worse (smaller accuracy) than the 10 learner counterpart in table 4.2 (ranks are not considered in this comparison).

| dataset | ARF | SRP | ARF | SRP | CAND $\|P\|$=30 | | | |
|---|---|---|---|---|---|---|---|---|
| | 30, 60% | 30, 60% | 30, 10% | 30, 10% | Min Estd Loss | Majo-rity Vote | Best MLP$^{*!}$ | At Least One$^{*\#}$ |
| airlines | 66.46 | **67.97** | 61.18 | 66.77 | 61.44 | 61.87 | 61.63 | 86.77 |
| electricity | 90.57 | 89.48 | 58.09 | 84.57 | **91.55** | 88.15 | 92.16 | 98.64 |
| kdd99 | 99.96 | 99.97 | 99.94 | **99.98** | 99.96 | 99.94 | 99.97 | 99.99 |
| WISDM | 85.59 | 86.43 | 78.90 | 85.86 | **88.85** | 80.26 | 90.81 | 97.96 |
| covtype | 94.79 | **95.60** | 86.09 | 91.35 | 93.77 | 89.00 | 94.54 | 99.14 |
| nomao | 97.20 | 97.37 | 97.16 | 97.33 | **97.56** | 97.27 | 97.72 | 99.37 |
| Avg Acc | 89.10 | **89.47** | 80.23 | 87.64 | 88.86 | 86.08 | 89.47 | 96.98 |
| Avg Rank | 3.25 | **1.83** | 5.92 | 3.00 | 2.42 | 4.58 | | |
| $AGR_a$ | 87.48 | **92.96** | 64.45 | 79.71 | 89.70 | 87.37 | $\underline{89.38}$ | 98.77 |
| $AGR_g$ | 81.96 | **89.14** | 62.98 | $\underline{75.99}$ | 87.23 | $\underline{84.84}$ | 87.03 | 98.31 |
| $RBF_f$ | 75.02 | **75.43** | 30.01 | 52.17 | $\underline{66.98}$ | $\underline{62.34}$ | 67.67 | 93.02 |
| $RBF_m$ | **86.63** | 85.68 | 30.01 | 68.07 | 85.78 | $\underline{84.31}$ | 85.63 | 97.80 |
| $LED_a$ | 73.96 | 73.97 | 62.55 | 53.26 | **74.02** | 73.94 | 73.99 | 87.71 |
| $LED_g$ | 73.10 | 73.14 | 62.79 | 51.81 | **73.30** | 73.22 | 73.26 | 87.35 |
| Avg Acc | 79.69 | **81.72** | 52.13 | 63.50 | $\underline{79.50}$ | $\underline{77.67}$ | 79.49 | 93.83 |
| Avg Rank | 2.83 | **1.83** | 5.67 | 5.33 | **1.83** | 3.50 | | |
| epsilon | $\underline{\text{NA}}$ | $\underline{\text{NA}}$ | 61.21 | 60.20 | **85.89** | $\underline{50.04}$ | 85.76 | 99.76 |
| SVHN | 19.89 | 20.72 | 21.76 | 23.59 | **57.02** | $\underline{18.79}$ | 57.38 | 87.12 |
| gisette | $\underline{74.72}$ | $\underline{75.80}$ | 89.25 | 90.93 | **96.26** | $\underline{50.84}$ | 96.36 | 99.93 |
| spam | $\underline{95.40}$ | $\underline{\text{NA}}$ | 97.15 | 97.29 | **98.30** | 97.60 | 98.54 | 99.90 |
| sector | $\underline{\text{NA}}$ | $\underline{\text{NA}}$ | 4.72 | 16.95 | **73.56** | $\underline{1.71}$ | 73.19 | 82.62 |
| Avg Acc | $\underline{38.00}$ | $\underline{19.30}$ | 54.82 | 57.79 | **82.21** | $\underline{43.80}$ | 82.25 | 93.87 |
| Avg Rank | 5.20 | 5.00 | 3.00 | 2.40 | **1.00** | 4.40 | | |
| Overall Acc | $\underline{70.75}$ | $\underline{66.10}$ | 62.84 | 70.34 | **83.60** | $\underline{70.68}$ | 83.82 | 94.95 |
| Overall Rank | 3.68 | 2.76 | 4.97 | 3.65 | **1.79** | 4.15 | | |

tal changes ($RBF_m$). They were also from [15]. $AGR_a$, $AGR_g$, $LED_a$ and $LED_g$ had concept drifts occurring after every 250000 instances, with drift width size 50 for abrupt ($\text{AGR}_\text{a}$, $\text{LED}_\text{a}$) and 50000 for gradual ($\text{AGR}_\text{g}$, $\text{LED}_\text{g}$). The high-dimensional datasets are commonly used in the NNs literature [2]. Table 4.1 summarizes the characteristics of the datasets and the number of features after online one-hot encoding. Each algorithm was executed three times. The average accuracy and wall-time were considered in the evaluation process. Standard deviations were omitted from the tables due to space constraints. But are mentioned in the tables if necessary. CPU experiments were run on an Ubuntu 20.04.3 system with an Intel(R) Core(TM) i7-6700K CPU at 4.00GHz, and with 64GB RAM. The OpenJDK version was 11.0.11, and the JVM configurations were: -Xmx32g, -Xms50m, and -Xss1g. The GPU experiments were run on the same system using an NVIDIA Quadro GV100 GPU with 32508 MiB memory.

ADL was selected as it is the currently available streaming NN method. As per the authors of [49], their online hyperparameter tuning method did not produce significantly better results for classification tasks than a model with the default hyperparameters. Hence it was not selected as a baseline in the experiments. SRP and parallel ARF were also selected as the comparison baselines as they are the current state-of-the-art streaming algorithms. For ADL, the Python implementation was used in the experiments [3] and the same parameter configurations as in [19] were used as the thresholds ($\alpha_D = 0.0001$, $\alpha_W = 0.0005$, $\delta = 0.05$, $\zeta = 0.001$). For SRP, HT (50 as the grace period and 0.01 as the split confidence) was selected as the base learner, with 10 and 30 as the ensemble sizes, 60% and 10% as the subspace sizes. For ARF, Adaptive Random Forest HT (same grace period and split confidence as in SRP) was selected as the base learner, with 10 and 30 as the ensemble sizes, 60%, and 10% were selected as the feature percentages considered for each split, 6.0 for the lambda hyperparameter and 10 (in 10 ensembles setting) and 30 (in 30 ensembles setting) jobs to run in parallel. ADWIN was selected as

[2]from: https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/

[3]available at: https://github.com/ContinualAL/ADL_Pytorch

Table 4.4: Accuracy (%) for selected ensemble settings against CAND variants. C=CAND(CPU trained).

| dataset | ARF | SRP | ARF | SRP | C | C$_{sub}$ | C$_{sub}^{SB=0.6}$ | C |
|---|---|---|---|---|---|---|---|---|
| | 10, | 10, | 30, | 30, | $\|P\|$=10 | $\|P\|$=30 | $\|P\|$=30 | $\|P\|$=30 |
| | 60% | 60% | 10% | 10% | | $\|M\|$=10 | $\|M\|$=10 | |
| airlines | 65.86 | 66.74 | 61.18 | **66.77** | 61.13 | 61.40 | 61.10 | 61.44 |
| electricity | 89.87 | 89.14 | 58.09 | 84.57 | 84.97 | 90.21 | 90.59 | **91.55** |
| kdd99 | 99.96 | 99.97 | 99.94 | **99.98** | 99.92 | 99.96 | 99.96 | 99.96 |
| WISDM | 85.28 | 85.36 | 78.90 | 85.86 | 72.72 | 89.68 | **90.54** | 88.85 |
| covtype | 94.49 | **95.28** | 86.09 | 91.35 | 88.91 | 94.37 | 93.79 | 93.77 |
| nomao | 97.08 | 97.23 | 97.16 | 97.33 | 97.15 | **97.61** | 97.13 | 97.56 |
| Avg Acc | 88.76 | **88.95** | 80.23 | 87.64 | 84.13 | 88.87 | 88.85 | 88.86 |
| Avg Rank | 4.58 | 3.17 | 6.83 | 3.67 | 7.00 | **3.08** | 4.42 | 3.25 |
| AGR$_a$ | 85.91 | **92.44** | 64.45 | 79.71 | 89.64 | 89.31 | 88.16 | 89.70 |
| AGR$_g$ | 79.99 | **87.55** | 62.98 | 75.99 | 87.19 | 87.17 | 86.83 | 87.23 |
| RBF$_f$ | **71.20** | 70.84 | 30.01 | 52.17 | 67.20 | 67.06 | 66.84 | 66.98 |
| RBF$_m$ | 84.70 | 83.19 | 30.01 | 68.07 | 85.77 | 85.18 | 84.69 | **85.78** |
| LED$_a$ | 73.92 | 73.49 | 62.55 | 53.26 | **74.03** | **74.03** | 73.89 | 74.02 |
| LED$_g$ | 73.06 | 72.73 | 62.79 | 51.81 | 73.27 | 73.28 | 73.10 | **73.30** |
| Avg Acc | 78.13 | **80.04** | 52.13 | 63.50 | 79.52 | 79.34 | 78.92 | 79.50 |
| Avg Rank | 4.33 | 3.67 | 7.67 | 7.33 | 2.58 | 3.08 | 5.00 | **2.33** |
| epsilon | 50.26 | 50.21 | 61.21 | 60.20 | 85.86 | **85.89** | 83.11 | **85.89** |
| SVHN | 19.52 | 20.35 | 21.76 | 23.59 | 55.60 | 55.08 | 54.12 | **57.02** |
| gisette | 82.36 | 82.17 | 89.25 | 90.93 | **96.38** | 96.28 | 94.40 | 96.26 |
| spam | 96.12 | 96.22 | 97.15 | 97.29 | 98.03 | **98.42** | 97.64 | 98.30 |
| sector | 0.74 | 0.80 | 4.72 | 16.95 | 67.51 | 71.61 | 72.20 | **73.56** |
| Avg Acc | 49.80 | 49.95 | 54.82 | 57.79 | 80.68 | 81.46 | 80.29 | **82.21** |
| Avg Rank | 7.60 | 7.40 | 5.80 | 5.20 | 2.60 | 2.10 | 3.60 | **1.70** |
| Overall Acc | 73.55 | 74.34 | 62.84 | 70.34 | 81.49 | 83.33 | 82.83 | **83.60** |
| Overall Rank | 5.38 | 4.59 | 6.82 | 5.41 | 4.15 | 2.79 | 4.38 | **2.47** |

the drift and warning detector for both SRP and ARF. For SRP, the delta value for ADWIN was 0.00001 for the drift detector and 0.0001 for the warning detector. In the ARF case, it was 0.001 for the drift detector and 0.01 for the warning detector. As online one-hot encoding and normalization are in-built in CAND, the same data were fed into all algorithms. For both ARF and SRP, different random seeds (9,19,121) were used in each iteration.

Table 4.5: Wall time (s) for selected ensemble settings against CAND variants. C=CAND(CPU trained).

| dataset | ARF 10, 60% | SRP 10, 60% | ARF 30, 10% | SRP 30, 10% | C $\lvert P\rvert$=10 | $C_{sub}$ $\lvert P\rvert$=30 $\lvert M\rvert$=10 | $C_{sub}^{SB=0.6}$ $\lvert P\rvert$=30 $\lvert M\rvert$=10 | C $\lvert P\rvert$=30 |
|---|---|---|---|---|---|---|---|---|
| airlines | 127.56 | 208.50 | **22.46** | 381.44 | 15888.01 | 9784.03 | 7626.94 | 28999.15 |
| electricity | 6.10 | 9.95 | **3.47** | 24.17 | 120.99 | 81.10 | 67.75 | 196.99 |
| kdd99 | **49.67** | 111.36 | 100.27 | 151.76 | 1853.02 | 2201.20 | 844.08 | 5311.52 |
| WISDM | 3.07 | 3.92 | **2.77** | 3.57 | 20.21 | 15.76 | 11.17 | 41.64 |
| covtype | 140.12 | 252.40 | **123.21** | 339.30 | 2769.45 | 1682.24 | 949.38 | 4981.91 |
| nomao | 13.59 | 26.15 | **10.01** | 18.89 | 185.05 | 140.53 | 42.49 | 384.48 |
| Avg | 56.69 | 102.05 | **43.70** | 153.19 | 3472.79 | 2317.48 | 1590.30 | 6652.62 |
| Avg Rank | 1.83 | 3.33 | **1.17** | 3.67 | 6.83 | 6.17 | 5.00 | 8.00 |
| $AGR_a$ | 113.89 | 253.63 | **45.16** | 3137.58 | 4165.67 | 3286.07 | 2025.02 | 7680.91 |
| $AGR_g$ | 125.14 | 298.59 | **46.13** | 2688.19 | 4161.60 | 3410.06 | 2131.33 | 7679.61 |
| $RBF_f$ | 105.93 | 249.15 | **54.87** | 363.33 | 2852.29 | 2331.26 | 1992.29 | 4850.71 |
| $RBF_m$ | 102.48 | 216.62 | **54.08** | 352.31 | 2855.15 | 2121.74 | 1691.46 | 4858.58 |
| $LED_a$ | **73.50** | 168.49 | 135.07 | 675.34 | 4395.22 | 3213.00 | 2335.20 | 6599.98 |
| $LED_g$ | **74.15** | 171.34 | 137.01 | 634.07 | 4452.85 | 3044.41 | 2246.57 | 6593.23 |
| Avg | 99.18 | 226.30 | **78.72** | 1308.47 | 3813.80 | 2901.09 | 2070.31 | 6377.17 |
| Avg Rank | 1.67 | 3.00 | **1.33** | 4.33 | 7.00 | 6.00 | 4.67 | 8.00 |
| epsilon | 2006.44 | 2298.30 | **1170.11** | 1464.35 | 4987.39 | 5059.79 | 2481.79 | 11576.26 |
| SVHN | 1033.51 | 2392.79 | **403.31** | 1053.32 | 2739.18 | 2044.38 | 1787.48 | 5549.00 |
| gisette | 246.95 | 272.70 | 178.46 | **153.06** | 560.66 | 678.26 | 187.56 | 1332.19 |
| spam | 3144.16 | 3742.61 | 5163.25 | **1759.17** | 10414.31 | 8410.42 | 2174.42 | 25601.24 |
| sector | 112244.68 | 88245.67 | 38638.48 | 20272.19 | 15046.42 | 10573.59 | **8981.35** | 30843.94 |
| Avg | 23735.15 | 19390.41 | 9110.72 | 4940.42 | 6749.59 | 5353.29 | **3122.52** | 14980.53 |
| Avg Rank | 4.00 | 5.20 | 3.00 | **2.20** | 5.80 | 5.40 | 3.00 | 7.40 |
| Overall | 7035.94 | 5818.95 | 2722.83 | **1968.94** | 4556.91 | 3416.34 | 2210.37 | 9004.78 |
| Overall Rank | 2.41 | 3.76 | **1.76** | 3.47 | 6.59 | 5.88 | 4.29 | 7.82 |

CAND was implemented in MOA[78] using the Deep Java Library (DJL) as an interface to call the PyTorch. MOA was chosen because it had an efficient implementation of SRP and ARF. The MLPs in the $P$ pool used two types of optimizers: Adam and SGD, each with five learning rates: 5e-1, 5e-2, 5e-3, 5e-4,and 5e-5. All the MLPs were single-layer, with either $2^8$, $2^9$, or $2^{10}$ neurons in the hidden layer. All the above 30 configurations of MLPs were considered for the $|P|$=30 setups, while only the $2^9$ neurons in the hidden layer configurations were considered for the $|P|$=10 setups. Other parameters for Adam and SGD were set to Pytorch defaults (betas = [0.9, 0.999], eps = 1e−8, weight decay = 0, amsgrad = False for Adam, and momentum = 0, dampening = 0, weight decay = 0, nesterov = False for SGD). Pytorch CrossEntropyLoss was used for the loss calculation. Along with CAND, efficient CAND$_{sub}$ and CAND$_{sub}^{SB}$ were used in the experiments. In the initial experiments, for CAND$_{sub}$ smaller pool size ($|M|$) was set to 10, to match the smallest ensemble size. For CAND$_{sub}^{SB}$, the backpropagation threshold was empirically determined using low-dimensional datasets and set to 0.6 in the initial experiments. In the later experiments, these hyperparameters were further explored.

### 4.3.1 CAND against ADL and ensemble learners

These experiments compare CAND against ADL and different ARF and SRP configurations, using both 10 and 30 base learners. We also evaluated two vote aggregation methods for CAND: 1) majority vote and 2) minimum estimated loss: a special case of vote aggregation which discards all MLPs except the one with the least estimated loss at a given point. We also include two hypothetical baselines: 1) the accuracy of the single best MLP, as determined at the end of an experiment (Best MLP), and 2) an accuracy estimate based on how often at least one MLP had predicted the correct label and assuming it would have been selected (AtLeastOne). For these experiments, except for ADL, all the other methods were run using CPUs. ADL experiments were run using GPUs.

As per table 4.2, considering average accuracy and average rank, one can see that CAND($|P|$=10) with minimum estimated loss as the vote aggregation method performs better than ADL, different ensemble settings and CAND with the majority vote. Also, on average, it outperforms the hypothetical Best MLP. It only lags behind the hypothetical AtLeastOne. In general, CAND variants seem to perform well on high-dimensional data. Also, on high-dimensional data, ARF and SRP with the smaller split percentage (10%) and subspace size (10%) perform better compared to higher values (60%). Another notable aspect for both ARF and SRP is that they perform poorly on high-dimensional data with more than two classes (SVHN: 3072 features and 10 classes, sector: 55197 features and 105 classes). The higher ARF split percentage(60%) and higher SRP subspace size(60%) hinder the performance in this case, which is worth investigating. But beyond the scope of this research. ADL also performs poorly on high-dimensional data with more than two classes. Overall, ADL is one of the worst performers in these experiments.

Experimental results for CAND($|P|$=30) against ensemble learners with 30 base learners are summarized in table 4.3. The table also compares the 30 learner values against 10 learner values in table 4.2. On epsilon and sector datasets, both ARF with 60% split percentage and SRP with 60% subspace size failed to complete. Also, the same happened on the spam dataset for SRP with the same settings. On spam, only one run of ARF with the same settings was completed with random seed 19. The re-runs of those failed experiments with higher -Xmx, which is greater than the initial 32GB, also failed. For the failed parameter combinations, 0.00% accuracy was assumed when computing the average accuracy and ranks. One can observe the same phenomenon where CAND with minimum estimated loss outperforms all the other methods in the 30 learner setting as well. If one excludes CAND with the majority vote and all the failed ARF and SRP cases, a higher number of learners generally have slightly improved the accuracy of all the methods. If one compares the average accuracy of the hypothetical AtLeastOne method to its 10 learner counterpart, an increase of 3% can be seen. This suggests that the prediction

Table 4.6: Chosen percentage(%) of each hyperparameter type: network width, optimizer and learning rate for CAND$_{sub}$($|M|$=10, $|P|$=30). Learning rate 5e-1 was chosen <0.05% for all except on kdd99 (8.71%).

| dataset | # neurons | | | optimizer | | learning rate | | | |
|---|---|---|---|---|---|---|---|---|---|
| | $2^8$ | $2^9$ | $2^{10}$ | Adam | SGD | 5e-2 | 5e-3 | 5e-4 | 5e-5 |
| airlines | 31.92 | **38.29** | 29.79 | **78.04** | 21.96 | 9.89 | 31.00 | 25.45 | **33.66** |
| electricity | 39.25 | **44.45** | 16.30 | **91.59** | 8.41 | 1.41 | 30.26 | 7.27 | **61.06** |
| kdd99 | 29.49 | **52.19** | 18.32 | **88.17** | 11.83 | 12.88 | 23.17 | 19.54 | **35.71** |
| WISDM | 19.92 | **67.34** | 12.74 | **99.66** | 0.34 | 0.05 | 24.39 | 0.39 | **75.15** |
| covtype | 32.77 | **52.07** | 15.16 | **98.67** | 1.33 | 0.01 | 2.33 | 0.92 | **96.72** |
| nomao | 31.32 | **45.54** | 23.13 | **59.75** | 40.25 | 6.96 | 21.67 | 9.33 | **61.65** |
| Avg(%) | 30.78 | **49.98** | 19.24 | **85.98** | 14.02 | 5.20 | 22.14 | 10.48 | **60.66** |
| AGR$_a$ | 12.14 | **80.38** | 7.47 | **88.34** | 11.66 | 0.64 | 21.21 | 7.84 | **70.31** |
| AGR$_g$ | 21.95 | **58.06** | 20.00 | **98.14** | 1.86 | 0.37 | 3.89 | 1.42 | **94.32** |
| RBF$_f$ | 16.37 | **70.74** | 12.89 | **97.50** | 2.50 | 0.00 | 27.16 | 2.29 | **70.55** |
| RBF$_m$ | 24.73 | **56.07** | 19.19 | **89.12** | 10.88 | 0.00 | 11.70 | 1.00 | **87.30** |
| LED$_a$ | 26.64 | **55.27** | 18.09 | **68.49** | 31.51 | 1.00 | 31.26 | 10.33 | **57.42** |
| LED$_g$ | 34.02 | **54.73** | 11.25 | **83.65** | 16.35 | 0.04 | 16.09 | 7.17 | **76.70** |
| Avg(%) | 22.64 | **62.54** | 14.82 | **87.54** | 12.46 | 0.34 | 18.55 | 5.01 | **76.10** |
| epsilon | 22.63 | **71.33** | 6.04 | **71.98** | 28.02 | 0.01 | 27.73 | 0.25 | **72.01** |
| SVHN | 9.62 | **88.40** | 1.98 | **91.05** | 8.95 | 0.00 | 7.85 | 0.81 | **91.34** |
| gisette | 26.76 | **72.95** | 0.29 | **70.61** | 29.39 | 0.64 | 27.03 | 6.65 | **65.65** |
| spam | 21.11 | **64.30** | 14.58 | **98.56** | 1.44 | 0.01 | 23.44 | 1.42 | **75.13** |
| sector | 15.19 | **73.77** | 11.03 | **96.11** | 3.89 | 0.06 | 12.28 | 3.31 | **84.32** |
| Avg(%) | 19.06 | **74.15** | 6.78 | **85.66** | 14.34 | 0.14 | 19.67 | 2.49 | **77.69** |
| Overall(%) | 24.46 | **61.52** | 14.01 | **86.44** | 13.56 | 2.00 | 20.14 | 6.20 | **71.12** |

potential of CAND can increase with the number of learners.

### 4.3.2 Different Variants of CAND against best ensembles

To further investigate the cost associated with the number of learners, ensemble learner configurations which yielded good results on all datasets were compared against different CAND configurations: CAND($|P|$=10), CAND$_{sub}$($|M|$=10, $|P|$=30), CAND$_{sub}^{SB=0.6}$($|M|$=10, $|P|$=30), and CAND($|P|$=30). Here, all experiments were run using CPUs.

Table 4.4 summarizes the average accuracy for each method on each dataset. In general, CAND variants yield good results on all datasets. They perform particularly well on high-dimensional data. Also, they yield good results on all the synthetic datasets with concept drifts. On airlines, CAND variants lag slightly behind ARF and SRP. This could be since the tree-based algorithms converge quicker on low-dimensional data. When considering average accuracy and average rank, CAND$_{sub}$($|M|$=10, $|P|$=30) is very close to the top-performing CAND($|P|$=30). Also, CAND$_{sub}^{SB=0.6}$($|M|$=10, $|P|$=30) is not far behind compared to the CAND$_{sub}$($|M|$=10, $|P|$=30). The summary of the average wall-time for each algorithm on each dataset is shown in table 4.5. Both ARF and SRP have a smaller average wall-time on low-dimensional datasets. This is well in line with the general understanding of tree-based algorithms performing efficiently on such data compared to NNs. ARF variants seem to be faster than the other algorithms on low-dimensional data. But in sector, ARF is slower compared to its SRP counterparts. Among all CAND variants, CAND($|P|$=30) seems to take a lot of time. It is the slowest among all the methods. This aligns with the general understanding that training more NNs consume more resources. Except for CAND$_{sub}^{SB=0.6}$($|M|$=10, $|P|$=30), all CAND variants seem to take more time on all datasets other than sector, which is high-dimensional and multi-class. Interestingly, CAND($|P|$=10) takes more time than CAND$_{sub}$($|M|$=10, $|P|$=30). This could be due to some of the smaller ($2^8$) predictive MLPs

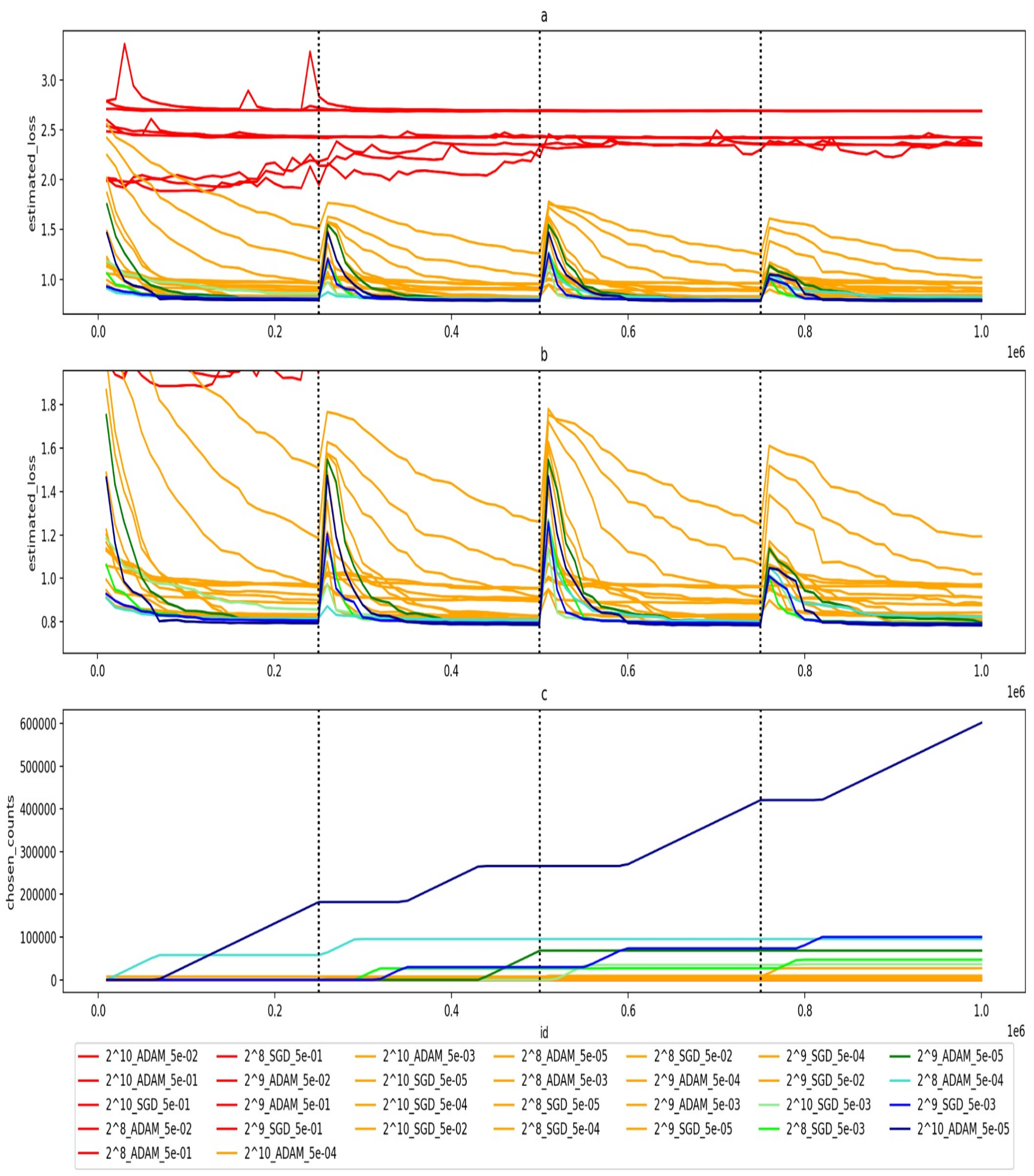


Figure 4.2: a): MLPs **estimated losses**, b): MLPs **estimated losses** (zoomed-in), c): **chosen counts** of MLPs. **MLP colour scheme**: cooler colours for the most chosen, and warmer colours for the least chosen MLPs. **Vertical dotted lines**: start of concept drift.

missing in CAND($|P|$=10). CAND$_{sub}^{SB=0.6}$($|M|$=10, $|P|$=30) is the fastest CAND version, which is expected, as it performs the lowest number of network updates among all the CAND variants. Considering both accuracy and wall-time, CAND$_{sub}^{SB=0.6}$($|M|$=10, $|P|$=30) strikes a good compromise, yielding good accuracy and being reasonably computationally efficient, especially so for high-dimensional data.

### 4.3.3 Discussion on CAND$_{sub}$($|M|$=10, $|P|$=30)

To further understand the behaviour of CAND, we computed the chosen percentage of MLPs (for prediction) by different hyperparameter types: network width, optimizer, and learning rate. Table 4.6 summarises those results. MLPs with $2^9$ hidden neurons were selected most often. But MLPs with $2^8$ and $2^{10}$ hidden neurons have contributed differently for different datasets. Unsurprisingly, most chosen MLPs have Adam as the optimizer. As expected in per-instance stream learning, most chosen MLPs have the lowest learning rate (5e-5).

To further understand the CAND$_{sub}$($|M|$=10, $|P|$=30) behaviour on concept drifts, an in-depth analysis was done on LED$_\mathrm{a}$ dataset. Figure 4.2.a and figure 4.2.b illustrate the estimated loss of each MLP on the same dataset with time. Figure 4.2.b is a zoomed-in version of figure 4.2.a. Also, figure 4.2.c depicts the chosen count of each MLP with time. Figure 4.2.a and figure 4.2.b clearly show that MLPs estimated losses increase immediately after a concept drift. This suggests that, when learning from an evolving data stream, the estimated loss gives us a good indication of the predictive performance of an MLP at a given moment. As expected, according to figure 4.2.a, figure 4.2.b, and figure 4.2.c CAND correctly chooses the MLP with the least estimated loss most of the time. This is evident in figure 4.2.c, with different MLPs contributing to prediction at a given moment.

### 4.3.4 Effect of Mini-batch Size and GPU Training

Another set of experiments was carried out to understand the effect of mini-batch size and GPU training on CAND's predictive and computing performance. Here, CAND$_{sub}$($|M|$=10, $|P|$=30, mini-batch size=1, CPU training) was compared against CAND$_{sub}$($|M|$=10, $|P|$=30, GPU training) with different mini-batch sizes: 1, 4, 16 and 32. The evaluation was done per-instance as usual for all the experiments.

As per table 4.7, per-instance CPU and GPU training yield similar predic-

Table 4.7: Effect of mini-batch size on accuracy and wall time (s) for CAND$_{sub}$($|M|$=10, $|P|$=30).

| | Acc | | | | | Wall Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| device | CPU | GPU | | | | CPU | GPU | | | |
| batch size | 1 | 1 | 4 | 16 | 32 | 1 | 1 | 4 | 16 | 32 |
| airlines | 61.4 | **61.44** | 61.21 | 61.21 | 61 | 9784.03 | 5090.33 | 1104.13 | 405.24 | **291.46** |
| electricity | 90.21 | **91.51** | 86.94 | 79.28 | 76.11 | 81.1 | 341.76 | 95.15 | 37.43 | **27.69** |
| kdd99 | 99.96 | **99.96** | 99.93 | 99.85 | 99.78 | 2201.2 | 6833.39 | 1950.46 | 979.91 | **816.73** |
| WISDM | **89.68** | 89.55 | 78.64 | 64.36 | 60.39 | 15.76 | 41.55 | 14.89 | 7.8 | **6.57** |
| covtype | **94.37** | 93.74 | 89.52 | 88.15 | 88.26 | 1682.24 | 5027.07 | 1254.86 | 491.59 | **364.15** |
| nomao | 97.61 | **97.62** | 97.46 | 96.31 | 95.85 | 140.53 | 248.85 | 72.4 | 30.15 | **23.02** |
| Avg | 88.87 | **88.97** | 85.62 | 81.53 | 80.23 | 2317.48 | 2930.49 | 748.65 | 325.35 | **254.94** |
| Avg rank | 1.67 | **1.33** | 3 | 4.17 | 4.83 | 4 | 4.83 | 3.17 | 2 | **1** |
| $\text{AGR}_\text{a}$ | 89.31 | **89.35** | 89.13 | 89.05 | 88.91 | 3286.07 | 8428.79 | 2178.76 | 749.77 | **534.47** |
| $\text{AGR}_\text{g}$ | 87.17 | 87.11 | 87.11 | **87.31** | 87.23 | 3410.06 | 9227.59 | 2213.58 | 761.32 | **537.64** |
| $\text{RBF}_\text{f}$ | 67.06 | 67.05 | **67.28** | 60.71 | 59.71 | 2331.26 | 9412.32 | 2176.72 | 802.27 | **584.56** |
| $\text{RBF}_\text{m}$ | 85.18 | 85.22 | 85.61 | **86.13** | 86.03 | 2121.74 | 10264.88 | 2226.64 | 800.08 | **582.6** |
| $\text{LED}_\text{a}$ | 74.03 | **74.07** | 74.03 | 74.03 | 74 | 3213 | 10514.25 | 2192.45 | 878.36 | **656.89** |
| $\text{LED}_\text{g}$ | 73.28 | **73.3** | 73.29 | 73.3 | 73.3 | 3044.41 | 9421.42 | 2198.36 | 881.17 | **660.23** |
| Avg | 79.34 | 79.35 | **79.41** | 78.42 | 78.2 | 2901.09 | 9544.88 | 2197.75 | 812.16 | **592.73** |
| Avg rank | 3.5 | **2.33** | 3 | 2.5 | 3.67 | 3.83 | 5 | 3.17 | 2 | **1** |
| epsilon | 85.89 | 85.74 | **85.92** | 85.52 | 85.62 | 5059.79 | 807.52 | 332 | 205.65 | **180.77** |
| SVHN | 55.08 | **55.34** | 54.6 | 49.54 | 53.11 | 2044.38 | 241.18 | 109.96 | 75.3 | **69.67** |
| gisette | **96.28** | 96.1 | 96.09 | 96.14 | 95.66 | 678.26 | 61.72 | 34.53 | 26.3 | **24.75** |
| spam | **98.42** | 98.38 | 97.52 | 96.93 | 94.85 | 8410.42 | 401.02 | 284.44 | 244.99 | **237.41** |
| sector | 71.61 | 68.69 | 73.29 | 72.72 | **73.37** | 10573.59 | 429.81 | 213.59 | 152.86 | **140.67** |
| Avg | 81.46 | 80.85 | **81.48** | 80.17 | 80.52 | 5353.29 | 388.25 | 194.9 | 141.02 | **130.66** |
| Avg rank | **2** | 2.8 | 2.6 | 3.8 | 3.8 | 5 | 4 | 3 | 2 | **1** |
| Overall | **83.33** | 83.19 | 82.21 | 80.03 | 79.6 | 3416.34 | 4517.26 | 1097.23 | 442.95 | **337.61** |
| Overall rank | 2.41 | **2.12** | 2.88 | 3.47 | 4.12 | 4.24 | 4.65 | 3.12 | 2 | **1** |

tive performance as expected. For instance, in the sector dataset, per-instance CPU and GPU are fairly identical in terms of accuracy when we take into account the standard deviations (i.e. CPU batch size std 1: 1.81, GPU batch sizes std 1: 1.41, 4: 0.55, 16: 0.05, 32: 0.68). However, for low dimensional data, per-instance GPU training is slower. This could be due to per-instance data transfer happening between CPU and GPU. But for high-dimensional data, even with batch size 1, GPU training seems to be faster. Nevertheless, GPU training with larger mini-batch sizes is faster. However, this improved efficiency comes with a decay in prediction accuracy. The bigger the mini-batch size, the lower the accuracy compared to per-instance training. Overall, mini-batch size 4 yields good accuracies for all data types. Also, it performs faster compared to per-instance CPU or GPU-trained CAND. Furthermore, these results verify our initial assumption that a larger mini-batch size could inversely affect NN's quick adaptability to concept drifts.

These experiments were run using a single training iteration of a given mini-batch. Experiments on multiple (mini-batch size) training iterations using the same mini-batch did not reveal any significant results compared to per-instance training. Hence they are not included in this work.

### 4.3.5 Effect of smaller pool size ($|M|$)

In the previous experiments, smaller pool size ($|M|$) was set to 10 for CAND$_{sub}$($M$, $|P|$=30) to align with the smallest ensemble size. These experiments explore the effect of $|M|$ on CAND$_{sub}$($M$, $|P|$=30)'s accuracy and wall time using mini-batch size 1 and GPU training. We experiment with smaller pool sizes: 2, 4, 6, 8, and 10.

As shown in table 4.8, on average CAND$_{sub}$($M$, $|P|$=30) with larger $M$ pool sizes performs slightly better than the smaller pool sizes. But this comes with an increased computation cost. On average, even the smallest $M$ pool size 2, is not so far behind compared to $|M|$=10 setting. Its only considerable performance decreases are only on high-dimensional data with more than two classes (SVHN: 3072 features and 10 classes, sector: 55197 features and 105 classes). Overall CAND$_{sub}$($|M|$=2, $|P|$=30) is at lest 1.6 faster than CAND$_{sub}$($|M|$=10, $|P|$=30). Generally, $M$ pool sizes 6 and 8 yield competitive results considering their computation costs.

Table 4.8: Effect of $|M|$ on accuracy and wall time (s) for CAND$_{sub}$($M$, $|P|$=30, mini-batch size=1, GPU-trained)

| | Acc | | | | | Wall Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\|M\|$ | 2 | 4 | 6 | 8 | 10 | 2 | 4 | 6 | 8 | 10 |
| airlines | 61.2 | 61.34 | 61.36 | **61.45** | 61.44 | **3160.08** | 4325.67 | 3802.3 | 4266.72 | 5090.33 |
| electricity | 90.42 | 91.23 | 91.51 | **91.53** | 91.51 | 268.44 | **230.25** | 298.64 | 292.05 | 341.76 |
| kdd99 | 99.95 | 99.96 | 99.96 | 99.96 | **99.96** | **5239.57** | 6145.11 | 7269.31 | 6575.8 | 6833.39 |
| WISDM | 88.37 | **90.05** | 89.67 | 89.69 | 89.55 | **27.39** | 31.02 | 34.56 | 38.02 | 41.55 |
| covtype | 92.87 | 93.46 | 93.62 | 93.73 | **93.74** | 4111.38 | **3409.18** | 4114.78 | 5481.37 | 5027.07 |
| nomao | 97.18 | 97.44 | 97.54 | 97.5 | **97.62** | **149.03** | 166.27 | 267.65 | 297.18 | 248.85 |
| Avg | 88.33 | 88.91 | 88.94 | **88.98** | 88.97 | **2159.31** | 2384.58 | 2631.2 | 2825.19 | 2930.49 |
| Avg rank | 5 | 3.5 | 2.83 | **1.83** | **1.83** | **1.33** | 2 | 3.5 | 3.83 | 4.33 |
| AGR$_a$ | 87.7 | 88.62 | 89.2 | 89.27 | **89.35** | **5148.26** | 6423.45 | 8107.53 | 8705.5 | 8428.79 |
| AGR$_g$ | 86.64 | 86.9 | 86.99 | **87.13** | 87.11 | **6164.74** | 6559.55 | 7992.71 | 8733.13 | 9227.59 |
| RBF$_f$ | 66.09 | 66.35 | 66.5 | **67.13** | 67.05 | **5729.67** | 6746.42 | 7325.99 | 9727.68 | 9412.32 |
| RBF$_m$ | 84.65 | 84.63 | 84.73 | 84.7 | **85.22** | **6006.38** | 8089.97 | 8139.33 | 9401.23 | 10264.88 |
| LED$_a$ | 73.85 | 73.97 | 74.01 | 74.03 | **74.07** | **5081.26** | 6321.63 | 7547.84 | 9247.49 | 10514.25 |
| LED$_g$ | 73.13 | 73.25 | **73.31** | 73.28 | 73.3 | **5765.69** | 6868.67 | 8526 | 8297.06 | 9421.42 |
| Avg | 78.68 | 78.95 | 79.12 | 79.26 | **79.35** | **5649.33** | 6834.95 | 7939.9 | 9018.68 | 9544.88 |
| Avg rank | 4.83 | 4.17 | 2.5 | 2 | **1.5** | **1** | 2 | 3.17 | 4.17 | 4.67 |
| epsilon | 85.21 | 85.71 | **85.82** | 85.74 | 85.74 | **577.48** | 630.77 | 700.2 | 740.24 | 807.52 |
| SVHN | 45.24 | 45.43 | 52.03 | **55.45** | 55.34 | **173.19** | 188.86 | 207.82 | 224.1 | 241.18 |
| gisette | 95.64 | 95.66 | 95.77 | **96.12** | 96.1 | **50.43** | 52.96 | 56.39 | 59.28 | 61.72 |
| spam | 98.31 | 98.29 | 98.31 | 98.34 | **98.38** | **317.62** | 322.72 | 334.1 | 349.55 | 401.02 |
| sector | 57.51 | 62.68 | 65 | 68.21 | **68.69** | **205.13** | 218.29 | 267.81 | 323.11 | 429.81 |
| Avg | 76.38 | 77.55 | 79.38 | 80.77 | **80.85** | **264.77** | 282.72 | 313.27 | 339.26 | 388.25 |
| Avg rank | 4.6 | 4.2 | 2.8 | 1.8 | **1.6** | **1** | 2 | 3 | 4 | 5 |
| Overall | 81.41 | 82.06 | 82.67 | 83.13 | **83.19** | **2833.87** | 3337.1 | 3823.12 | 4279.97 | 4517.26 |
| Overall rank | 4.82 | 3.94 | 2.71 | 1.88 | **1.65** | **1.12** | 2 | 3.24 | 4 | 4.65 |

### 4.3.6 Effect of skip backpropagation threshold

In the previous experiments skip backpropagation threshold was set to 0.0 for all CAND variants except for CAND$_{sub}^{SB=0.6}$($|M|$=10, $|P|$=30), where it was set to 0.6. These experiments further explore the effect of skip backpropagation threshold on CAND$_{sub}^{SB}$($|M|$=10, $|P|$=30)'s accuracy and wall time using mini-batch size 1 and GPU training. For that, we experiment with skip backpropagation thresholds: 0.0, 0.3, 0.6, and 0.9.

Table 4.9 shows the accuracy and wall time for different skip backpropagation thresholds. Generally, higher thresholds result in more significant wall time savings. This behaviour is expected as larger thresholds allow more instances to avoid costly backpropagation. Typically, this wall time savings comes with a decrease in predictive performance. But, interestingly, a smaller threshold of 0.3, causes an increase in the accuracy with a less computing cost compared to no skip backpropagation (threshold of 0.0). Maybe the skip backpropagation threshold act as a regularizer and no skip backpropagation (0.0) allows the model to overfit.

### 4.3.7 Efficient CAND Variants

Considering previous experiments on mini-batch sizes, mini-batch size 4 yields competitive results with less wall-time. Also, experiments on the effect of $M$ pool size suggest that $M$ pool sizes 6 and 8 yield competitive results while being wall-time efficient. Furthermore, experiments on the effect of the skip backpropagation threshold reveal that 0.3 yields the best results compared to not using skip backpropagation. Hence, the next set of experiments attempts to find the most efficient variant of CAND, considering the above empirical outcomes. Here we compare CAND with the above parameter combinations against $\text{CAND}_{sub}^{SB=0.0}(|M|=10,\ |P|=30,\ \text{mini-batch size}=1)$. Here all the CAND variants were trained using GPUs.

As per table 4.10, on average $\text{CAND}_{sub}^{SB=0.3}(|M|=10,\ |P|=30,\ \text{mini-batch size}=4)$ and $\text{CAND}_{sub}^{SB=0.3}(|M|=8,\ |P|=30,\ \text{mini-batch size}=4)$ seem to yield very competitive results with less than a quarter of the wall-time used by $\text{CAND}_{sub}^{SB=0.0}(|M|=10,\ |P|=30,\ \text{mini-batch size}=1)$. For high-dimensional data, $\text{CAND}_{sub}^{SB=0.3}(|M|=10,\ |P|=30,\ \text{mini-batch size}=4)$ yields the best results. Considering all the above, $\text{CAND}_{sub}^{SB=0.3}(|M|=10,\ |P|=30,\ \text{mini-batch size}=4)$ seems to be the best CAND variant with good computation efficiencies.

Table 4.9: Effect of skip backpropagation threshold on accuracy and wall time (s) for $\text{CAND}^{SB}_{sub}(|M|=10, |P|=30$, mini-batch size=1, GPU-trained)

| | Acc | | | | Wall Time | | | |
|---|---|---|---|---|---|---|---|---|
| SB | 0 | 0.3 | 0.6 | 0.9 | 0 | 0.3 | 0.6 | 0.9 |
| airlines | 61.44 | **61.49** | 60.64 | 60.11 | 5090.33 | 4766.42 | 4608.48 | **2321.72** |
| electricity | 91.51 | **91.94** | 91.65 | 90.06 | 341.76 | 279.75 | **213.11** | 230.6 |
| kdd99 | 99.96 | **99.97** | 99.96 | 99.96 | 6833.39 | 4591.33 | 4477.32 | **4055.24** |
| WISDM | 89.55 | 89.42 | **90.16** | 89.4 | 41.55 | 33.87 | 32.4 | **31.29** |
| covtype | **93.74** | 93.66 | 93 | 80.96 | 5027.07 | 3796.15 | 2958.63 | **2911.57** |
| nomao | **97.62** | 97.36 | 97.16 | 96.38 | 248.85 | 139.07 | 133.69 | **126.54** |
| Avg | **88.97** | 88.97 | 88.76 | 86.15 | 2930.49 | 2267.76 | 2070.6 | **1612.83** |
| Avg rank | 1.83 | **1.67** | 2.67 | 3.83 | 4 | 3 | 1.83 | **1.17** |
| $\text{AGR}_\text{a}$ | **89.35** | 89.34 | 87.89 | 64.87 | 8428.79 | 7369.06 | 6736.86 | **4547.55** |
| $\text{AGR}_\text{g}$ | 87.11 | **87.21** | 86.65 | 62.56 | 9227.59 | 7254.49 | 7120.22 | **4202.88** |
| $\text{RBF}_\text{f}$ | 67.05 | **67.47** | 66.89 | 64.19 | 9412.32 | 9185.03 | **8323.73** | 9024.21 |
| $\text{RBF}_\text{m}$ | 85.22 | **85.34** | 84.68 | 82.81 | 10264.88 | 10063.59 | **7064.63** | 8166.7 |
| $\text{LED}_\text{a}$ | **74.07** | 74.06 | 73.87 | 72.95 | 10514.25 | 8394.46 | **7671.93** | 7726.84 |
| $\text{LED}_\text{g}$ | **73.3** | 73.28 | 73.13 | 72.23 | 9421.42 | 7975.14 | 7633.01 | **6979.15** |
| Avg | 79.35 | **79.45** | 78.85 | 69.93 | 9544.88 | 8373.63 | 7425.06 | **6774.56** |
| Avg rank | **1.5** | **1.5** | 3 | 4 | 4 | 3 | **1.5** | **1.5** |
| epsilon | **85.74** | 85.24 | 82.89 | 73.93 | 807.52 | 661.79 | 665.37 | **534.69** |
| SVHN | 55.34 | **55.54** | 54.64 | 54.07 | 241.18 | 234.22 | 230.48 | **227.24** |
| gisette | **96.1** | 95.72 | 94.16 | 93.07 | 61.72 | 50.14 | 48.55 | **47.19** |
| spam | 98.38 | **98.5** | 97.05 | 89.06 | 401.02 | 317.46 | 313.54 | **310.94** |
| sector | 68.69 | **71.16** | 69.54 | 70.26 | 429.81 | 372.4 | 359.39 | **354.87** |
| Avg | 80.85 | **81.23** | 79.66 | 76.08 | 388.25 | 327.2 | 323.47 | **294.98** |
| Avg rank | 2 | **1.4** | 3 | 3.6 | 4 | 2.8 | 2.2 | **1** |
| Overall | 83.19 | **83.33** | 82.59 | 77.46 | 4517.26 | 3852.02 | 3446.55 | **3047.01** |
| Overall rank | 1.76 | **1.53** | 2.88 | 3.82 | 4 | 2.94 | 1.82 | **1.24** |

Table 4.10: Accuracy and wall time (s) for efficient $\text{CAND}_{sub}^{SB}(M,\ |P|{=}30$, GPU-trained) variants.

| | Acc | | | | Wall Time | | | |
|---|---|---|---|---|---|---|---|---|
| $\lvert M\rvert$ | 10 | 10 | 8 | 6 | 10 | 10 | 8 | 6 |
| batch size | 1 | 4 | 4 | 4 | 1 | 4 | 4 | 4 |
| SB | 0 | 0.3 | 0.3 | 0.3 | 0 | 0.3 | 0.3 | 0.3 |
| airlines | **61.44** | 61.21 | 61.21 | 61 | 5090.33 | 1104.13 | 405.24 | **291.46** |
| electricity | **91.51** | 86.94 | 79.28 | 76.11 | 341.76 | 95.15 | 37.43 | **27.69** |
| kdd99 | **99.96** | 99.93 | 99.85 | 99.78 | 6833.39 | 1950.46 | 979.91 | **816.73** |
| WISDM | **89.55** | 78.64 | 64.36 | 60.39 | 41.55 | 14.89 | 7.8 | **6.57** |
| covtype | **93.74** | 89.52 | 88.15 | 88.26 | 5027.07 | 1254.86 | 491.59 | **364.15** |
| nomao | **97.62** | 97.46 | 96.31 | 95.85 | 248.85 | 72.4 | 30.15 | **23.02** |
| Avg | **88.97** | 85.62 | 81.53 | 80.23 | 2930.49 | 748.65 | 325.35 | **254.94** |
| Avg rank | **1** | 2 | 3.17 | 3.83 | 4 | 3 | 2 | **1** |
| $\text{AGR}_\text{a}$ | **89.35** | 89.13 | 89.05 | 88.91 | 8428.79 | 2178.76 | 749.77 | **534.47** |
| $\text{AGR}_\text{g}$ | 87.11 | 87.11 | **87.31** | 87.23 | 9227.59 | 2213.58 | 761.32 | **537.64** |
| $\text{RBF}_\text{f}$ | 67.05 | **67.28** | 60.71 | 59.71 | 9412.32 | 2176.72 | 802.27 | **584.56** |
| $\text{RBF}_\text{m}$ | 85.22 | 85.61 | **86.13** | 86.03 | 10264.88 | 2226.64 | 800.08 | **582.6** |
| $\text{LED}_\text{a}$ | **74.07** | 74.03 | 74.03 | 74 | 10514.25 | 2192.45 | 878.36 | **656.89** |
| $\text{LED}_\text{g}$ | **73.3** | 73.29 | 73.3 | 73.3 | 9421.42 | 2198.36 | 881.17 | **660.23** |
| Avg | 79.35 | **79.41** | 78.42 | 78.2 | 9544.88 | 2197.75 | 812.16 | **592.73** |
| Avg rank | **2** | 2.67 | 2.17 | 3.17 | 4 | 3 | 2 | **1** |
| epsilon | 85.74 | **85.92** | 85.52 | 85.62 | 807.52 | 332 | 205.65 | **180.77** |
| SVHN | **55.34** | 54.6 | 49.54 | 53.11 | 241.18 | 109.96 | 75.3 | **69.67** |
| gisette | 96.1 | 96.09 | **96.14** | 95.66 | 61.72 | 34.53 | 26.3 | **24.75** |
| spam | **98.38** | 97.52 | 96.93 | 94.85 | 401.02 | 284.44 | 244.99 | **237.41** |
| sector | 68.69 | 73.29 | 72.72 | **73.37** | 429.81 | 213.59 | 152.86 | **140.67** |
| Avg | 80.85 | **81.48** | 80.17 | 80.52 | 388.25 | 194.9 | 141.02 | **130.66** |
| Avg rank | **2** | **2** | 3 | 3 | 4 | 3 | 2 | **1** |
| Overall | **83.19** | 82.21 | 80.03 | 79.6 | 4517.26 | 1097.23 | 442.95 | **337.61** |
| Overall rank | **1.65** | 2.24 | 2.76 | 3.35 | 4 | 3 | 2 | **1** |

# Chapter 5

# Online Domain Incremental Pool

In chapter 4, estimated loss of a NN was used to identify the best-performing NN from a pool of networks for SL. Section 2.1.2 introduces Online Domain Incremental Continual Learning, which pertains to a specific form of OCL. ODICL involves NN learning from a data stream where the input data distribution changes when different tasks are encountered. Compared to SL, a learning algorithm in this setting has the additional requirement of preserving past knowledge. Additionally, section 2.1.3 discuses OSCL, the fusion of SL and OCL to derive new solutions to Online Continual Learning. This chapter presents a novel approach for ODICL, leveraging some of the SL techniques outlined in section 2.1.3.

## 5.1 Introduction

In recent years, ODICL has been applied to various domains, including healthcare, facial expression and action unit recognition among different demographics (refer to section 2.1.2.3). However, some replay-based ODICL methods raise privacy concerns due to their reliance on storing and replaying sensitive data [4, 66]. On the other hand, regularisation methods do not perform as well as replay-based methods for ODICL [4].

Many existing ODICL methods use the explicit end-of-task signal during the training process. For instance, methods like EWC [10] and LwF [20] optimize their weights by incorporating this signal, while replay-based approaches such as ER [21] employ it to update their replay buffer. However, the GDumb [59] does not require this signal for replay buffer updates. Furthermore, as discussed in section 2.1.3, SL techniques such as drift detectors and task prediction in recurrent concepts can be employed to detect the end-of-task signal and concept prediction in ODICL. Also, as per CAND experiments in section 4.3.3, NN's loss gives a good indication of the current performance of the NN. It also closely resembles the underlying input distribution shifts. Using a drift detector, this idea could be leveraged to detect end-of-the-task signals in ODICL.

Considering the practical importance of non-replay Online Continual Learning, this work proposes a non-replay-based method that alleviates catastrophic forgetting in NNs for ODICL using OSCL techniques. The main contributions of this work are the following:

1. Online Domain Incremental Pool (ODIP): we introduce a novel method to alleviate catastrophic forgetting for Online Domain Incremental Continual Learning without using instance replay. Here, a small pool of tiny CNNs is trained, and the best one is frozen at the end of each task. Task Predictor is trained to predict the best frozen CNN for evaluation for a given instance. The experiment results reveal that ODIP yields superior accuracy than regularization baselines. Furthermore, an in-depth investigation is done to understand better the effectiveness of different TPs on three ODICL datasets.

2. Instead of relying on an external task id signal during prediction, ODIP uses an automatic Task Detection mechanism to detect tasks in the incoming data. This allows ODIP to select the most appropriate frozen network to produce predictions for each instance. ADWIN detects drifts in CNN's loss to determine a new task. To the best of our knowledge,

this automatic Task Detection for Online Domain Incremental Continual Learning has not been proposed before.

The experimental results demonstrate that ODIP, both with and without automatic TD, outperforms existing popular regularization methods. Since ODIP does not rely on a replay buffer for ODICL, it emerges as a promising choice for such settings with heightened privacy requirements.

## 5.2 Online Domain Incremental Pool (ODIP)

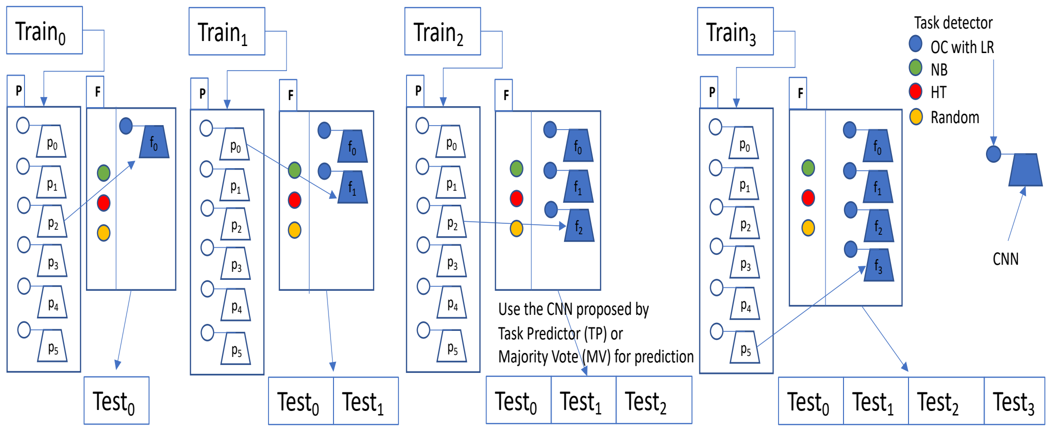


Figure 5.1: Proposed Online Domain Incremental Pool (ODIP)

The ODICL is defined as the training set composed of multiple concepts of non-IID data, where each concept has a different input distribution with the same label distribution [4]. The goal of the learning algorithm is to min-

```
Algorithm 5 TRAIN OC WITH LR
Input: Task Predictor $TP$: One Class Classifier with Logistic Regression , $z$:
    extracted features
1: $score, in\_class \leftarrow$ TRAIN $OC(z)$
2: TRAIN $LR(score, in\_class)$
```

imize catastrophic forgetting of the past concepts while performing well on the current concept [4, 61]. Initially, at training, we assume that the task id that signals the end of a concept is available to the learning model. However, this information is not available to the model during evaluation. Later, the proposed method(ODIP) is extended to discard this external task id signal.

**Algorithm 6** ODIP TRAINING ALGORITHM

```
Input: $P$: pool of training CNNs, $F$: pool of frozen CNNs, $T$: task set, $X_t$:
    training set for task $t$, $TP$: Task Predictor
 1: Initialize pool $F = \{\}$
 2: for all task $t \in T$ do
 3:     for all mini-batch $b_t$ in training set $X_t$ for task $t$ do
 4:         $z \leftarrow$ features from mini-batch $b_t$ for task $t$
 5:         for all learner $p \in P$ do
 6:             Compute loss $L_p$ of mini-batch $b_t$ and train CNN$_p$
 7:             Update ADWIN$_p$ with $L_p$
 8:             if task predictor $TP_p$ is One Class Classifier with LR then
 9:                 TRAIN OC WITH LR($TP_p$, $z$)
10:             end if
11:         end for
12:         if task predictor $TP$ is Naive Bayes or Hoeffding Tree then
13:             TRAIN $TP(z, t)$
14:         end if
15:     end for
16:     Append the CNN with lowest loss estimated using ADWIN to F
17: end for
```

We propose an Online Domain Incremental Pool (ODIP), where $P$ pool of tiny CNNs are trained for each concept $t$ with a given Task Predictor. The Task Predictors could be None, Naive Bayes (NB), Hoeffding Tree (HT), and One Class Classifier (OC) with Logistic Regression (LR). The Task Predictor is trained for mini-batch $b_t$ using extracted features from a static feature extractor. At the end of each task's training, CNN with the lowest estimated loss is frozen and added into the frozen pool $F$. In the special case of OC with LR, the relevant OC with the LR is also part of the frozen CNN. Algorithm 6, along with figure 5.1, further explains this training approach.

**Algorithm 7** Predict OC with LR

```
Input: Task Predictor $TP$: One Class Classifier with Logistic Regression, $z$:
    extracted features.
1: $score$, $in\_class \leftarrow$ PREDICT $OC(z)$
Output: PREDICT $LR(score)$
```

ODIP has two vote aggregation methods for prediction: Weighted Voting (WV) or votes from the best CNN (CNN$_{best}$). For Weighted Voting, the probabilities of the Task Predictor are used as weights. In the CNN$_{best}$ case, it is either selected randomly from the $F$ pool or the one predicted by Task Predictor. Algorithm 8 further explains this. Recently proposed ODICL algorithms rely on an explicit end of the task signal (task id) to identify the start of a new task. ODIP is also relying on these explicit task ids to distinguish different tasks. This reliance on an explicit task id may preclude one from employing current ODICL algorithms in real-life settings where it may be challenging to identify such a signal explicitly.

ODIP is extended to identify concept drifts in the incoming stream automatically. ADaptive sliding WINdow (ADWIN) [23] is used as a task detector. ADWIN has nice properties where it uses exponential histograms for memory efficiency and discards the buffer related to the previous concept once confronted with a drift. Every CNN in $P$ pool has its ADWIN. They are updated with each CNN's loss after training. Once updated, a new task is identified if any ADWIN detects a drift in the loss. Here a drift in the loss is assumed to be related to the drift in the input stream. Algorithm 9 explains this training with automatic Task Detection in detail. In the experiments, the effectiveness of ODIP was compared against popular regularization baselines.

**Algorithm 8** ODIP prediction algorithm

```
Input: $x_t$: instance of task $t$, $F$: pool of frozen CNNs, $TP$: Task Predictor,
    useWeightedVoting
1: $z \leftarrow$ features from instance $x_t$ of task $t$
2: if useWeightedVoting then
3:     if $TP$ is Majority Vote then
4:         $votes \leftarrow 1/|F| \sum_{f=1}^{|F|} \textsc{Predict}(f, x_t)$
5:     else if $TP$ is One Class Classifier with LR then
6:         $votes \leftarrow 1/|F| \sum_{f=1}^{|F|} \textsc{Predict OC with LR}(TP_f, z) \times \textsc{Predict}(f, x_t)$
7:     else
8:         $votes \leftarrow 1/|F| \sum_{f=1}^{|F|} \textsc{Predict}(TP, z) \times \textsc{Predict}(f, x_t)$
9:     end if
10: else
11:     if $TP$ is Random then
12:         Select CNN$_{selected}$ randomly from pool $F$
13:     else if $TP$ One Class Classifier with LR then
14:         CNN$_{selected} \leftarrow \arg\max_{f \in F} \textsc{Predict OC with LR}(TP_f, z)$
15:     else
16:         CNN$_{selected} \leftarrow \arg\max_{f \in F} \textsc{Predict}(TP_f, z)$
17:     end if
18:     $votes \leftarrow \textsc{predict}(\text{CNN}_{selected}, x_t)$
19: end if
Output: $votes$
```

**Algorithm 9** ODIP training algorithm with Auto Task Detection

```
Input: $P$: pool of training CNNs, $F$: pool of frozen CNNs, $T$: task set, $X_t$: training set for task $t$, $TP$: Task Predictor
 1: Initialize pool $F = \{\}$
 2: Initialize $taskId = 0$
 3: for all task $t \in T$ do
 4:     for all mini-batch $b_t$ in training set $X_t$ for task $t$ do
 5:         $taskEnd \leftarrow false$
 6:         $z \leftarrow$ features from mini-batch $b_t$ for task $t$
 7:         for all learner $p \in P$ do
 8:             Compute the loss $L_p$ of mini-batch $b_t$ and train CNN$_p$
 9:             Update ADWIN$_p$ with $L_p$
10:             if ADWIN$_p$ detects change then
11:                 $taskEnd \leftarrow true$
12:             end if
13:             if task predictor $TP_p$ is One Class Classifier with LR then
14:                 TRAIN OC WITH LR($TP_p$, $z$)
15:             end if
16:         end for
17:         if task predictor $TP$ is Naive Bayes or Hoeffding Tree then
18:             TRAIN $TP(z, taskId)$
19:         end if
20:         if $taskEnd$ then
21:             $taskId \leftarrow taskId + 1$
22:             Append the CNN with lowest loss estimated using ADWIN to F
23:         end if
24:     end for
25: end for
```

Table 5.1: Datasets

| Dataset | # tasks | instances per train/test task | # classes | Channels, H, W |
|---|---|---|---|---|
| CORe50 | 11 | 2000/1000 | 10 | 3, 32, 32 |
| RotatedCIFAR10 | 4 | 50000/10000 | 10 | 3, 32, 32 |
| RotatedMNIST | 4 | 60000/10000 | 10 | 1, 28, 28 |

## 5.3 Experiments

The experiments attempt to understand the effectiveness of ODIP against popular regularization baselines. They also try to identify the efficacy of Task Predictors. Lastly, they attempt to determine the effectiveness of ODIP with automatic Task Detection against regularization baselines.

Different versions of ODIP were compared against regularization baselines: LwF and EWC. The replay methods were not considered as a baseline for these experiments[1]. The baselines use CNNs with 4.3 times the parameters (144234) than in ODIP experiments (33450). For ODIP, ResNet-18 was used as the static feature extractor and flattened last-layer features were used to train the TPs. Five types of TPs were used in the experiments: random, Majority Vote (MV), NB, HT[2], and OC [3] with LR. Also, two types of vote aggregation methods were considered in the experiments: WV and the use of votes from CNN$_{best}$. Furthermore, two variants of automatic Task Predictor were considered in the experiments: $\alpha$) include the best training CNN for prediction when the frozen pool is empty, $\beta$) include the best training CNN for prediction when the frozen pool is empty OR when the predicted network is related to the current concept. $P$ pool size for ODIP was set to 6 CNNs. In the experiments, we also considered a hypothetical scenario of ODIP, where

[1]Chapter 6 compares ODIP against replay methods.

[2]Use skmultiflow[90] online versions of NB and HT

[3]Online One-Class SVM: https://scikit-learn.org/stable/modules/sgd.html#sgd-online-one-class-svm.

the task id is available at evaluation, and it is used to determine the correct frozen CNN. This is presented as the "Tid known" in the results. This allows one to determine the hypothetical upper bound of ODIP.

The experiments were done on three datasets: CORe50 [91], Rotated-CIFAR10, and RotatedMNIST. With RotatedCIFAR10 and RotatedMNIST, 90°rotations (0°, 90°, 180°, -90°) of the original images from CIFAR10 [92] and MNIST [93] were considered separate tasks. There were four tasks in each of those two datasets. With CORe50, 11 distinct sessions (8 indoor and 3 outdoor) of the same object were considered as separate tasks: tasks 0-2,4-8 indoor, 3,9, and 10 outdoor. Here the 10 object categories were considered as the class labels. Though it uses the same dataset as in [91] for ODICL, task separation is more natural than the random separation in [91]. Also, our CORe50 version had a separate evaluation set for each task rather than a mixed evaluation set, as in [91]. This allows one to better understand forgetting in the ODICL setting. In the above datasets, all classes were presented in all the tasks. Such rearranging was done to the original datasets to adhere to the ODICL definition described in [4].

Table 5.2: Average accuracy after training on the last task

| dataset | Baselines | | ODIP | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | EWC | LwF | **Tid known** | Random | MV | $HT_{WV}$ | $OC_{WV}$ | $NB_{WV}$ | $NB_{NoWV}$ | $NB_{TD\alpha}$ | $NB_{TD\beta}$ |
| CORe50 | 0.41 | 0.41 | 0.69 | 0.42 | 0.53 | 0.63 | 0.56 | **0.66** | 0.61 | 0.44 | 0.47 |
| RotatedCIFAR10 | 0.44 | **0.48** | 0.48 | 0.38 | 0.45 | 0.44 | 0.46 | 0.43 | 0.40 | 0.40 | 0.42 |
| RotatedMNIST | 0.51 | 0.72 | 0.97 | 0.48 | 0.66 | 0.53 | 0.65 | **0.79** | 0.78 | **0.79** | **0.79** |
| **Avg** | 0.45 | 0.54 | 0.72 | 0.42 | 0.55 | 0.53 | 0.56 | **0.63** | 0.60 | 0.54 | 0.56 |

All experiments were run using Avalanche [94] Continual Learning platform [4]. Average accuracy and forgetting defined in [4] are used in the evaluation. All experiments were run three times, and relevant averages and standard deviations were considered in the evaluation. The standard deviations were omitted from this manuscript due to space constraints.

Table 5.2 contains the average accuracy of each method after training on

[4]ODIP source code available at: https://github.com/nuwangunasekara/ODIP.

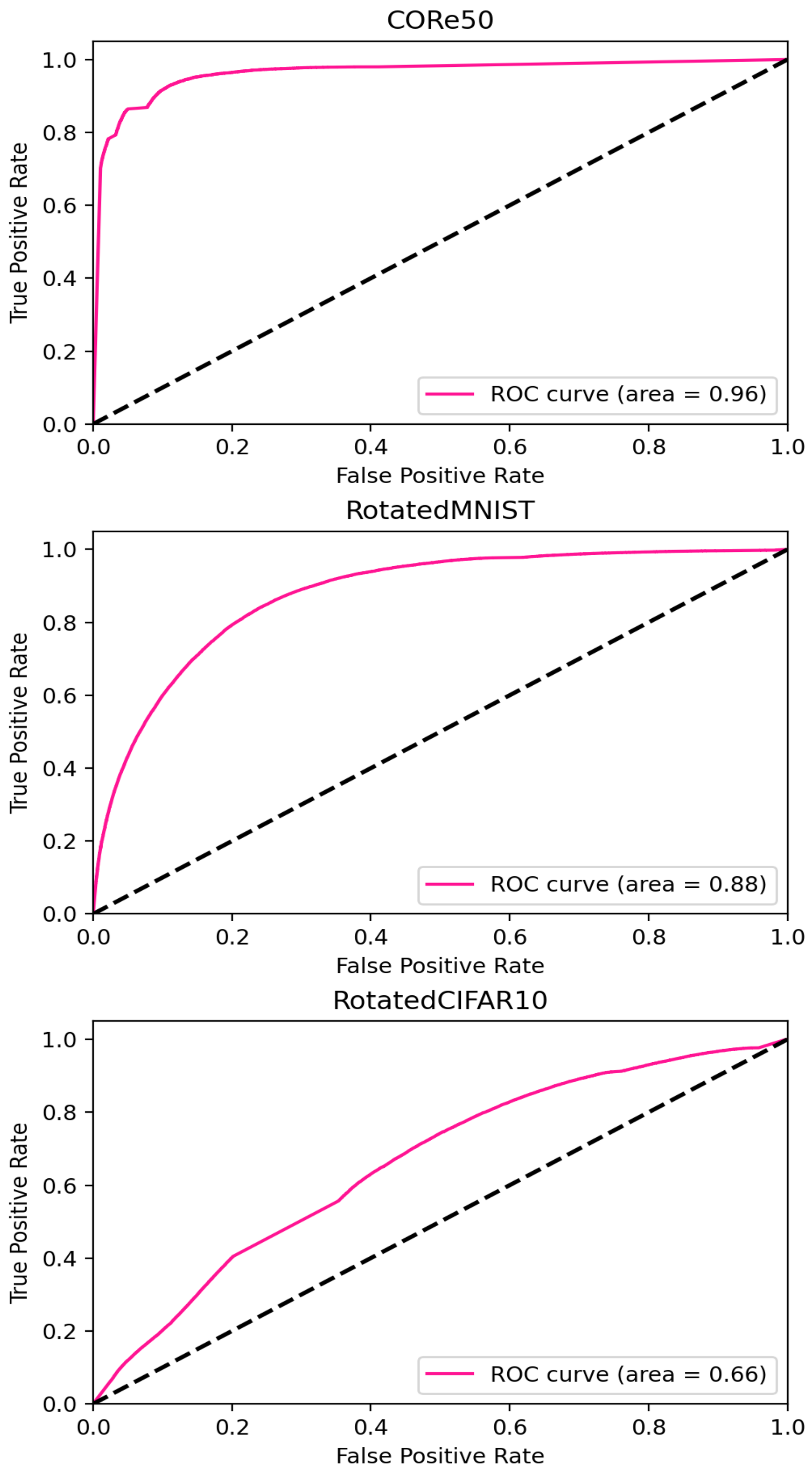


Figure 5.2: Effectiveness of Task Predictor: Naive Bayes. ROC curves for the predicted task id and AUC scores for the same.

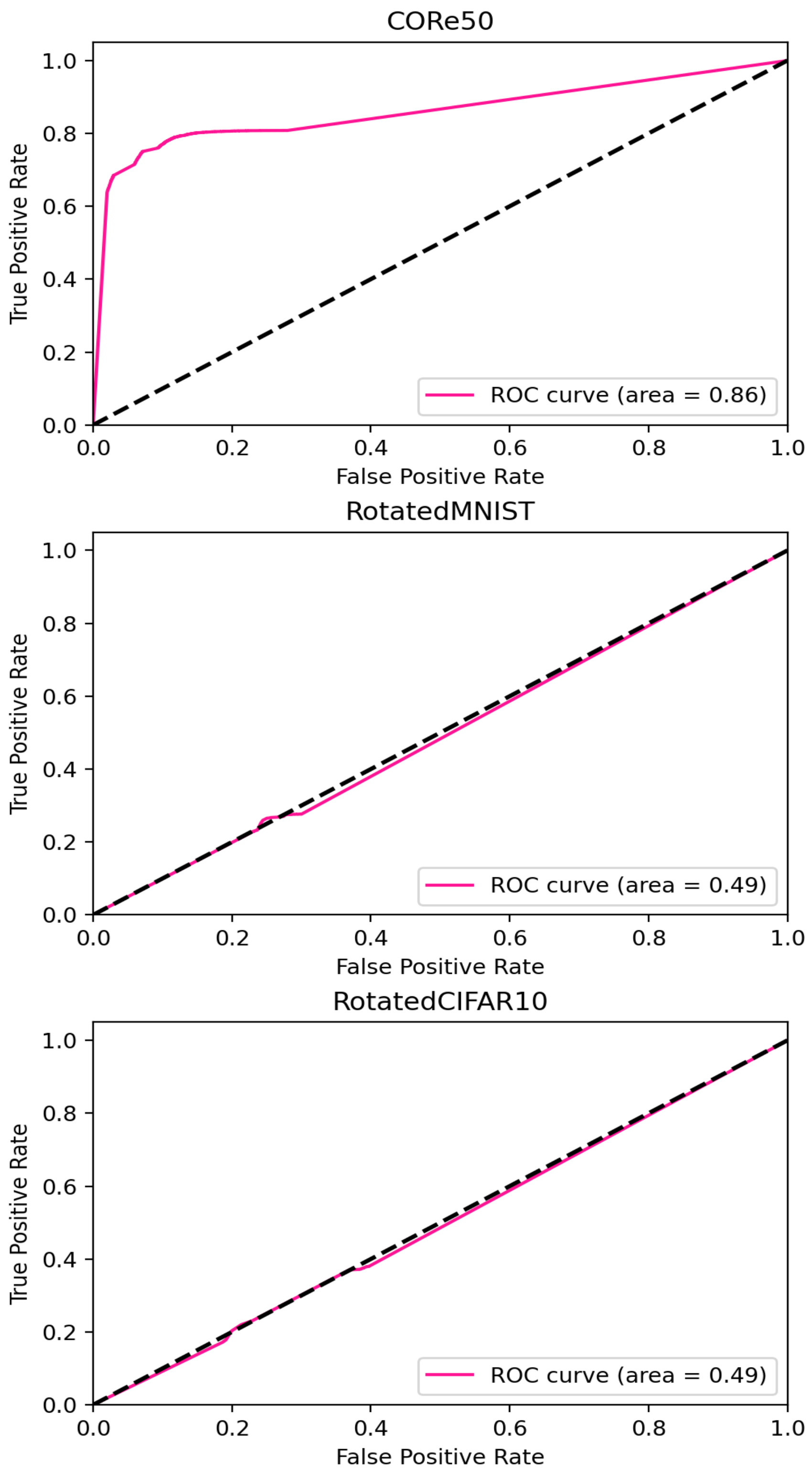


Figure 5.3: Effectiveness of Task Predictor: HT. ROC curves for the predicted task id and AUC scores for the same.

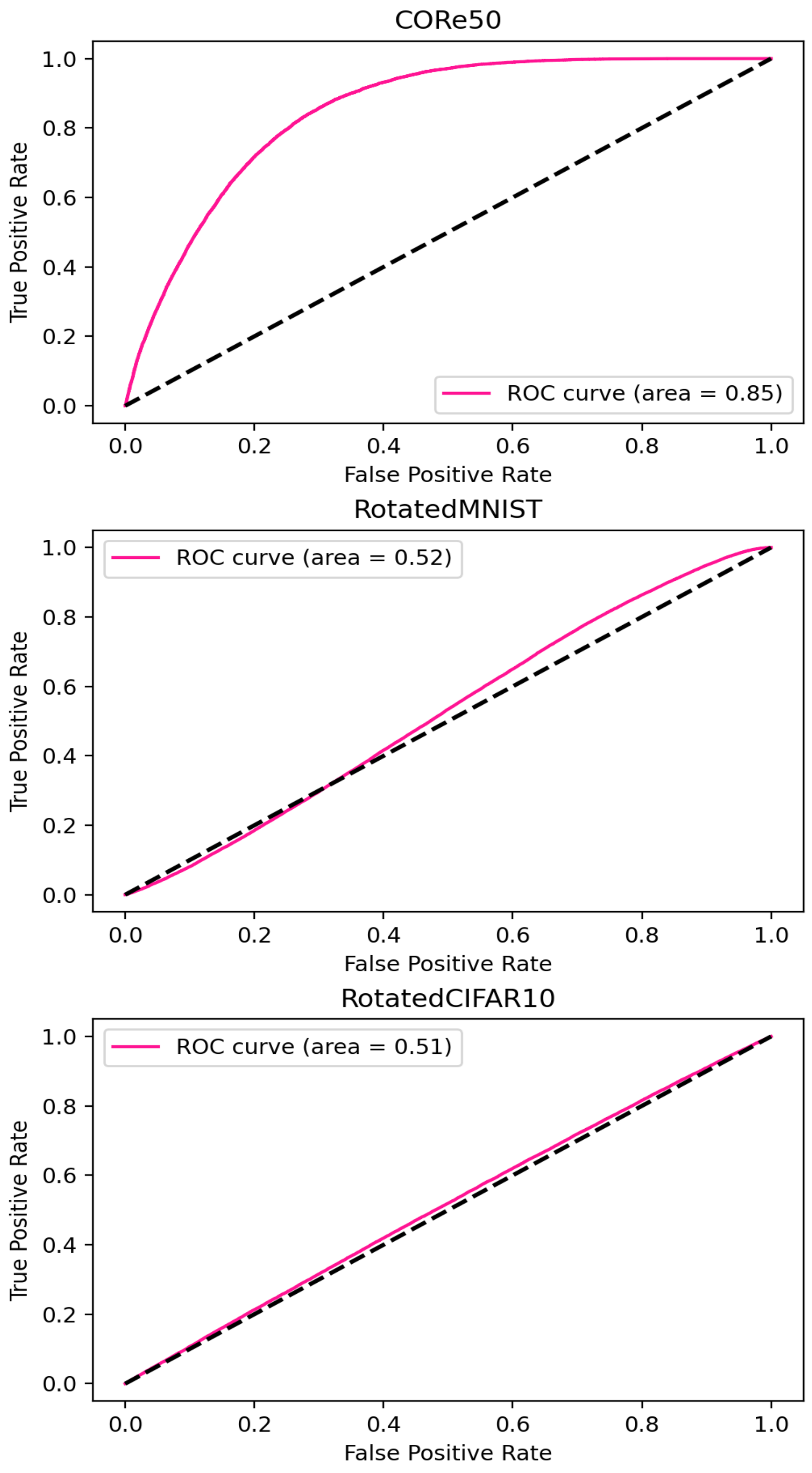


Figure 5.4: Effectiveness of Task Predictor: OC with LR. ROC curves for the predicted task id and AUC scores for the same.

Table 5.3: Average forgetting after training on the last task

| dataset | Baselines | | ODIP | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | EWC | LwF | **Tid known** | Random | MV | $\mathrm{HT}_{\mathrm{WV}}$ | $\mathrm{OC}_{\mathrm{WV}}$ | $\mathrm{NB}_{\mathrm{WV}}$ | $\mathrm{NB}_{No\mathrm{WV}}$ | $\mathrm{NB}_{\mathrm{TD}\alpha}$ | $\mathrm{NB}_{\mathrm{TD}\beta}$ |
| CORe50 | 0.10 | 0.07 | 0.00 | 0.01 | -0.02 | 0.02 | **-0.03** | 0.01 | 0.00 | 0.00 | 0.20 |
| RotatedCIFAR10 | 0.10 | 0.00 | 0.00 | 0.01 | **-0.03** | 0.05 | -0.03 | -0.01 | 0.00 | -0.03 | -0.01 |
| RotatedMNIST | 0.63 | 0.24 | 0.00 | 0.20 | 0.16 | 0.59 | **0.12** | **0.12** | **0.12** | **0.12** | **0.12** |
| **Avg** | 0.28 | 0.11 | 0.00 | 0.07 | 0.04 | 0.22 | **0.02** | 0.04 | 0.04 | 0.03 | 0.10 |

the last task. As one can see from the table, ODIP $\mathrm{NB}_{\mathrm{WV}}$ produces the best results. ODIP Random and EWC yield poor results. In general, all methods with Weighted Voting produced good results compared to the two baselines. However, weights from a good Task Predictor seem to boost the performance significantly. Also, ODIP $\mathrm{NB}_{\mathrm{TD}\beta}$, which has automatic Task Detection, yields better results than regularization baselines. It is also on par or better than the other ODIP methods, which use task ids, except for NB. Considering the hypothetical "Tid known" scenario, it is evident that just selecting the correct frozen CNN is sufficient to outperform current baselines by a considerable margin. This is further evident in table 5.3, with "Tid known" having a zero average forgetting across all datasets after training on the last task. Note here that a smaller average forgetting is better.

To further understand the effectiveness of the Task Predictors, the predicted task id was compared against the true task id in non-auto-TD mode against all datasets. This comparison was made for all evaluation instances after training on the last task. Figures 5.2, 5.3 and 5.4 show the ROC curves for the predicted task id and the relevant AUC scores for each TP on each dataset. According to the figure, it is clear that NB is a better Task Predictor for all datasets. This further strengthens the overall strong NB results in table 5.2. Figure 5.5 further explains the effectiveness of NB as a Task Predictor when predicting each task in a given dataset. From the per-task ROC curves and AUC scores, it is clear that NB performs similarly on all the tasks for a given dataset. Nevertheless, it does perform slightly better on specific tasks. This is evident in CORe50, with NB performing somewhat better for tasks

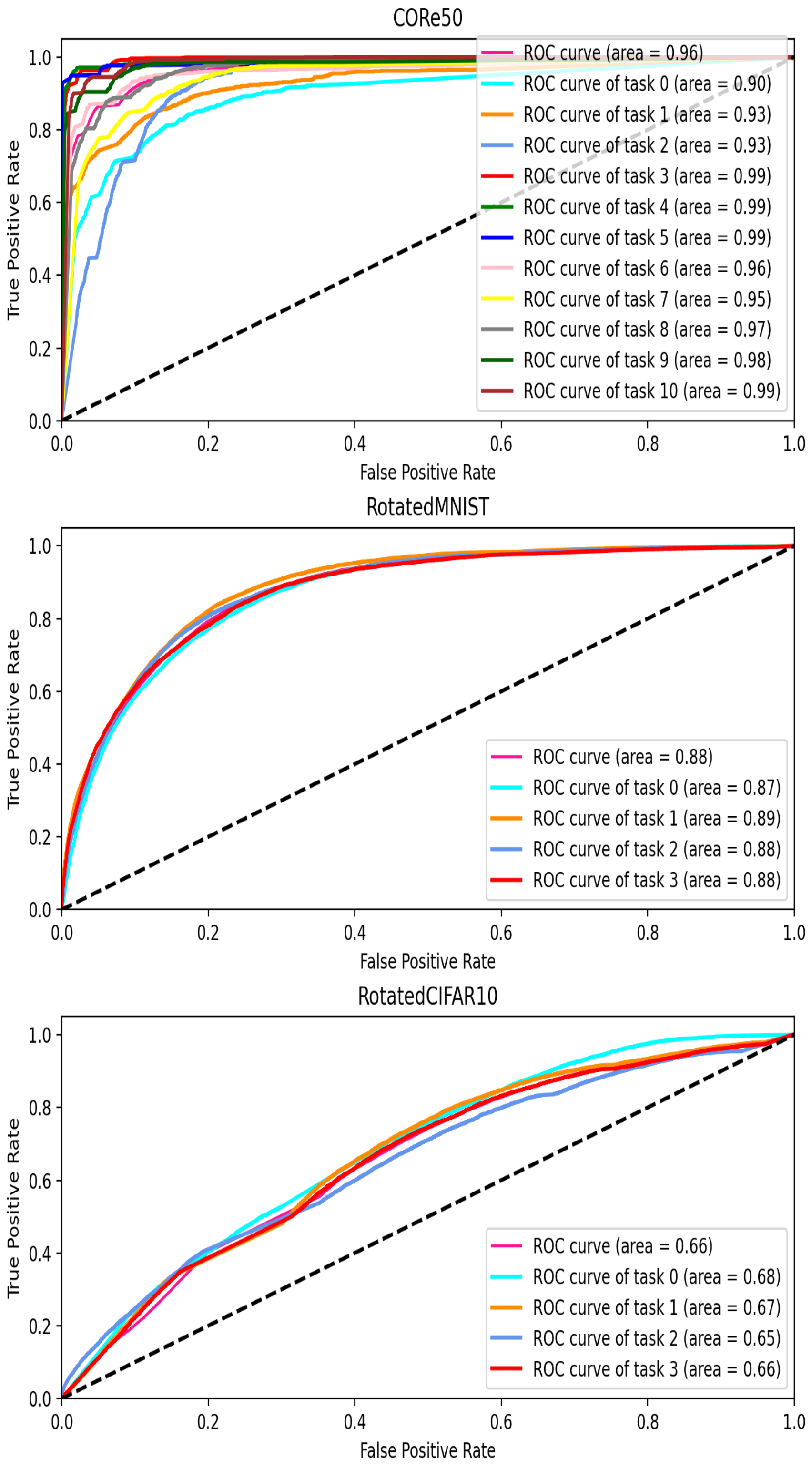


Figure 5.5: Effectiveness of Naive Bayes as a TP. ROC curves and AUC scores for predicted task id for each task.

3,4,5,9, and 10. This suggests, in general, that NB is a good Task Predictor.

# Chapter 6

# Online Domain Incremental Networks

This chapter extends the work discussed in chapter 5 by presenting an enhanced algorithm for ODICL. The proposed approach further leverages OSCL to achieve improved results. Additionally, the method incorporates constraints to ensure a fairer comparison with the baselines, including replay-based methods.

## 6.1 Introduction

In the previous chapter, having a pool of training NNs allowed ODIP to select the best one to freeze at the end of the concept. But this makes it unfair for other baselines as they only use a single NN configuration throughout training on all the tasks. The baselines used a larger NN with 4.3 times the parameters than the ones used by ODIP to overcome this. We only train a single NN in Online Domain Incremental Networks (ODIN) to enable all methods to use a single NN architecture. It also employs the same task prediction and detection strategies as ODIP. But it utilizes the incremental or decremental drifts in the loss detected by ADWIN to increase or decrease the learning rate dynamically. Furthermore, ODIN was compared against popular replay methods: ER and MIR. With this Dynamic Learning-Rate, ODIN surpasses popular regularization methods and produces competitive results to replay methods without requiring an instance buffer for ODICL.

The main contributions of this work are the following:

1. Online Domain Incremental Networks (ODIN): we introduce a novel method to alleviate catastrophic forgetting for Online Domain Incremental Continual Learning without using instance replay. Here, a frozen copy of the training CNN is saved in a pool at the end of each task. A Task Predictor is trained to predict the best frozen CNN for evaluation for a given instance. The experiment results reveal that ODIN yields better accuracy than regularization and competitive performance to replay baselines. Furthermore, an in-depth investigation is done to understand better the effectiveness of different TPs on three ODICL datasets.

2. Instead of relying on an external task id signal during training, ODIN uses an automatic Task Detection mechanism to detect tasks in the incoming data. ADaptive sliding WINdow (ADWIN) is used to detect drifts in CNN's loss. An incremental drift in the loss is determined as the end of a task. Furthermore, incremental or decremental drifts in CNN's loss detected by ADWIN allow ODIN to increase or decrease the learning rate dynamically. To the best of our knowledge, this automatic Task Detection with Dynamic Learning-Rate (DL) adjustment for ODICL has not been proposed before.

## 6.2 Online Domain Incremental Networks (ODIN)

We propose an Online Domain Incremental Networks (ODIN), where CNN $p$ is trained on each concept $t$ with a given Task Predictor (TP). The TPs could be Naive Bayes or Hoeffding Tree. The TP is trained on mini-batch $b_t$ using extracted features from a feature extractor. Feature extractors extract features from high-dimensional data. Hence, it allows one to use simple learn-

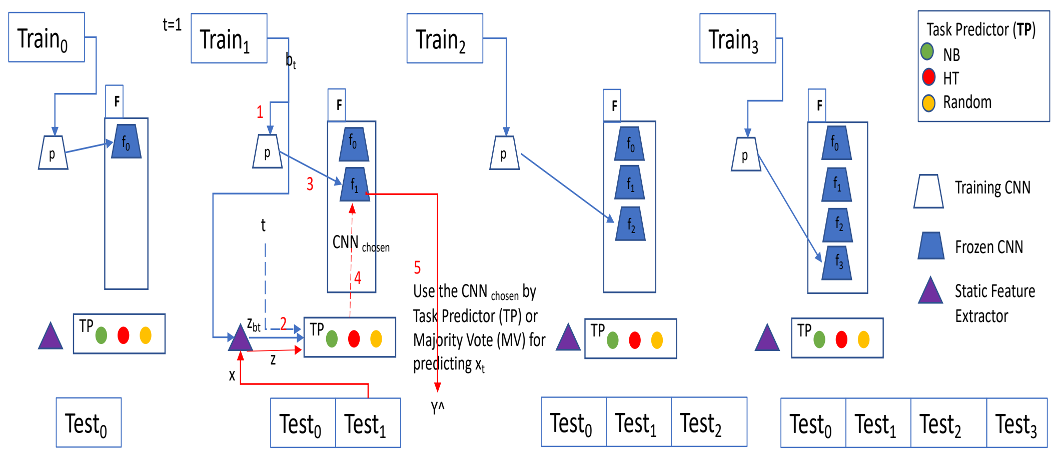


Figure 6.1: Proposed ODIN: 1) train network $p$ with incoming mini-batch $b_t$ for $t^{th}$ task, 2) train TP using extracted features and task id, 3) freeze a copy of $p$ at the end of task $t$ 4) at prediction, if enabled, TP predicts CNN$_{chosen}$ via extracted $x$ features 5) predict using CNN$_{chosen}$ or Majority Vote.

ing algorithms on high-dimensional data [95]. Usually, a pre-trained network is used as a feature extractor [95], and its last layer features are used to train the TP. At the end of each task's training, a copy of $p$ is frozen and added to the frozen pool $F$. Algorithm 10, along with figure 6.1, further explains this training approach. In ODIN, there are two vote aggregation methods for prediction: Weighted Voting (WV) or votes from the best CNN (CNN$_{best}$). Weighted Voting uses the TP's probabilities for each frozen CNN as weights. In the CNN$_{best}$ case, it is either selected randomly from the $F$ pool or the one predicted by TP. Algorithm 11 further explains this.

Generally, NN's loss distribution changes when the underlying input distribution changes as the network weights need to be readjusted to match the new distribution. Once a drift in the loss is detected, 1) it would be helpful to learn following the direction of the loss, where the network learns faster if there is an upward drift in the loss, and it learns slower if the drift in the loss decreases. Usually, in ODICL, NN's loss gradually decreases for a given task with non-IID training instances. Hence, 2) it would be helpful to reduce the magnitude of the learning for the incoming instances further away from the task's start so that the later instances of the same task do not disturb

**Algorithm 10** Training algorithm

```
Input: $p$: training CNN, $F$: frozen CNN pool, $T$: task set, $X_t$: training set
    for task $t$, $TP$: Task Predictor
1: Initialize: $F = \{\}$
2: for all task $t \in T$ do
3:     for all mini-batch $b_t$ in training set $X_t$ for task $t$ do
4:         Train $p$ with the computed the loss $L_{b_t}$ for mini-batch $b_t$
5:         if task predictor $TP$ is Naive Bayes or Hoeffding Tree then
6:             $z \leftarrow$ extract features from mini-batch $b_t$ via feature extractor
7:             TRAIN $TP(z, t)$
8:         end if
9:     end for
10:    Append a copy of $p$ to $F$
11: end for
```

the learned weights too much. Here we use the drift detector ADWIN [23] to monitor the loss of $p$. ADWIN uses exponential histograms for memory efficiency and discards the buffer related to the previous concept once a drift is detected. It also estimates the mean of the current items in the buffer. Once ADWIN detects a drift in the loss, one can compare the current estimated loss against the previous estimated loss to identify the direction of the loss. This helps determine whether to increase or decrease the learning rate (point 1). Also, the number of instances seen after the drift can be used to decrease the magnitude of the learning rate (point 2).

The easiest way to manage the learning of a NN is to adjust the learning rate. ODIN increases the learning rate to learn faster for the upward drifts in the loss detected by ADWIN. For the downward drifts, it decreases the learning rate to prevent against large changes of presumably already well-adjusted weights. Also, to reduce the magnitude of the learning in either direction, it uses a decaying factor $d^n$ where $d$ is $0 < d < 1$ and $n$ is the number of instances seen since the last drift. This decaying factor is discussed

**Algorithm 11** PREDICTION ALGORITHM

```
Input: $x_t$: instance of task $t$, $F$: frozen CNN pool, $TP$: Task Predictor,
    useWeightedVoting
 1: $z \leftarrow$ features from instance $x_t$ of task $t$
 2: if useWeightedVoting then
 3:     if $TP$ is Majority Vote then
 4:         $votes \leftarrow 1/|F| \sum_{f=1}^{|F|} \textsc{Predict}_f(x_t)$
 5:     else
 6:         $votes \leftarrow 1/|F| \sum_{f=1}^{|F|} \textsc{Predict}_{TP}(z)_f \times \textsc{Predict}_f(x_t)$
 7:     end if
 8: else
 9:     if $TP$ is Random then
10:         Select CNN$_{chosen}$ randomly from pool $F$
11:     else
12:         CNN$_{chosen} \leftarrow \arg\max_{f \in F} \textsc{Predict}_{TP}(z)$
13:     end if
14:     $votes \leftarrow \textsc{Predict}_{\text{CNN}_{chosen}}(x_t)$
15: end if
Output: $votes$
```

**Algorithm 12** DYNAMIC LEARNING-RATE

```
Input: $lr_0$: learning rate at start, $d$: decay factor$(0 < d < 1)$, $n$: instances
    seen since last drift, $upwardDrift$: whether the estimated loss going up
 1: if $upwardDrift$ then
 2:     $lr \leftarrow lr_0 * (1 + d^n)$
 3: else
 4:     $lr \leftarrow lr_0 * (d^n)$
 5: end if
Output: $lr$
```

**Algorithm 13** TRAINING ALGORITHM WITH DL

```
Input: $p$: training CNN, $F$: frozen CNN pool, $T$: task set, $X_t$: training set
    for task $t$, $TP$: Task Predictor, $lr_0$: learning rate at start, $d$: decay factor
 1: Initialize: $F = \{\}$, $n = 0$, $upwardDrift = true$
 2: for all task $t \in T$ do
 3:     for all mini-batch $b_t$ in training set $X_t$ for task $t$ do
 4:         Compute the loss $L_{b_t}$ of mini-batch $b_t$ for $p$
 5:         Update ADWIN$_p$ with $L_{b_t}$
 6:         if ADWIN$_p$ detects change then
 7:             $n \leftarrow 0$
 8:             if change is upward then
 9:                 $upwardDrift \leftarrow true$
10:             else
11:                 $upwardDrift \leftarrow false$
12:             end if
13:         else
14:             $n \leftarrow n + 1$
15:         end if
16:         if task predictor $TP$ is Naive Bayes or Hoeffding Tree then
17:             $z \leftarrow$ extract features from mini-batch $b_t$ via feature extractor
18:             TRAIN $TP(z, t)$
19:         end if
20:         $lr \leftarrow$ DYNAMIC LEARNING-RATE($lr_0$, $d$, $n$, $upwardDrift$)
21:         train $p$ with loss $L_{b_t}$ and $lr$
22:     end for
23:     Append a copy of $p$ to $F$
24: end for
```

**Algorithm 14** TRAINING ALGORITHM WITH DL AND AUTOMATIC TD

```
Input: $p$: training CNN, $F$: frozen CNN pool, $T$: task set, $X_t$: training set
    for task $t$, $TP$: Task Predictor, $lr_0$: learning rate at start, $d$: decay factor
1: Initialize: $F = \{\}$, $n = 0$, $upwardDrift = true$, $taskId = 0$
2: for all task $t \in T$ do
3:     for all mini-batch $b_t$ in training set $X_t$ for task $t$ do
4:         Compute the loss $L_{b_t}$ of mini-batch $b_t$ for $p$
5:         Update ADWIN$_p$ with $L_{b_t}$
6:         if ADWIN$_p$ detects change then
7:             $n \leftarrow 0$
8:             if change is upward then
9:                 $upwardDrift \leftarrow true$
10:                $taskId \leftarrow taskId + 1$
11:                Append a copy of $p$ to $F$
12:            else
13:                $upwardDrift \leftarrow false$
14:            end if
15:        else
16:            $n \leftarrow n + 1$
17:        end if
18:        if task predictor $TP$ is Naive Bayes or Hoeffding Tree then
19:            $z \leftarrow$ extract features from mini-batch $b_t$ via feature extractor
20:            TRAIN $TP(z, taskId)$
21:        end if
22:        $lr \leftarrow$ DYNAMIC LEARNING-RATE($lr_0$, $d$, $n$, $upwardDrift$)
23:        train $p$ with loss $L_{b_t}$ and $lr$
24:    end for
25: end for
```

in [60]. However, they continually decrease the learning rate from the start of learning in their work. Hence it forces NN not to learn too much from instances of later tasks. On the other hand, this Dynamic Learning-Rate (DL) in ODIN allows $p$ to best adjust to the current task. Algorithm 12 and algorithm 13 explain ODIN's Dynamic Learning-Rate adjustment mechanism.

Some of the proposed ODICL algorithms rely on an explicit end of the task signal (task id) to identify the start of a new task. The initial ODIN version also relies on explicit task ids to distinguish different tasks for training. This reliance on an explicit task id may preclude one from employing current ODICL algorithms in real-life settings where it may be challenging to identify such a signal explicitly.

One can assume the upward drift in $p$'s loss detected by ADWIN is due to the distribution shift in the underlying input features. Hence, ODIN determines the end of the task when an upward drift is detected. Line 10 in algorithm 14 explains this automatic Task Detection (TD) in ODIN. Line 22 of algorithm 14 further integrates DL with this automatic TD. With automatic TD, if the new task is similar to the past task, ADWIN might not detect an upward drift in the loss, as the learning on the new task can improve the prediction of the previous task due to backward knowledge transfer[4]. Hence, detected task ids may not align with actual task ids. Therefore in automatic task detection, the current training network $p$ is included in the $F$ pool only for prediction. In the experiments, the effectiveness of different versions of ODIN were compared against popular regularization and replay baselines.

## 6.3 Experiments

The experiments attempt to understand the effectiveness of ODIN against popular online CL baselines. Also, they try to identify the efficacy of Dynamic Learning-Rate adjustment. Furthermore, experiments attempt to determine the effectiveness of the Task Predictor. Lastly, they attempt to identify the efficacy of ODIN with automatic Task Detection against online CL baselines that do not use the external end-of-task signal.

We used the same ODICL datasets used in section 5.3 experiments: CORe50, RotatedCIFAR10, and RotatedMNIST[1]. Different versions of ODIN were compared against regularization baselines: LwF, EWC, and replay baselines: ER and MIR. For ER, an extended buffer version was considered in the experiments. Instead of randomly replacing an item from the buffer, we replace an instance from the most represented task's most represented class. It is referenced as $\mathrm{ER}_{TbCb}$ in this work. This $\mathrm{ER}_{TbCb}$ is a further extension of [60], where we attempt to balance the buffer concerning both task and class. This extended $\mathrm{ER}_{TbCb}$ was considered so that ODIN without automatic TD can be compared against a good replay method that utilizes the external end of task signal. All the replay methods used a 1k instance buffer. Also, all the methods use a simple CNN (33450 parameters) with four convolution layers. Two types of TPs were used in the experiments for ODIN: NB[2], and HT[2]. Quantized ResNet-18 was used as the feature extractor, and flattened last-layer features were used to train the TPs: NB and HT. ResNet was chosen considering [95], where HT was trained on extracted features by the ResNet feature extractor for images. Also, three types of vote aggregation methods were considered in the experiments: Majority Vote (MV), Weighted Voting (WV), and $\mathrm{CNN}_{best}$. In the experiments, we also considered a hypothetical scenario of ODIN, where the task id is available at evaluation and is used to determine the correct frozen CNN. This is presented as the "$\mathrm{known}_{tid}$" in the results. It indicates achievable performance if task prediction is perfect. In the results for ODIN, $\mathrm{TP}_{\mathrm{WV}}$ represents Task Predictor with Weighted Voting, $\mathrm{TP}_{No\mathrm{WV}}$ represents: Task Predictor without Weighted Voting, $\mathrm{TP}^{\mathrm{DL}}_{\mathrm{WV}}$ represents: Task Predictor with Weighted Voting and Dynamic Learning-Rate, $\mathrm{TP}^{\mathrm{DL}}_{No\mathrm{WV}}$ represents: Task Predictor without Weighted Voting and with Dynamic Learning-

[1]Please refer to table 5.1 on section 5.3 for dataset information.

[2]Source code is available at: https://github.com/nuwangunasekara/ODIN and it uses online NB and HT[90]

Rate, $\mathrm{TP}_{\mathrm{WV}}^{\mathrm{TD}}$ represents: Task Predictor with Weighted Voting and automatic Task Detection and, $\mathrm{TP}_{\mathrm{WV}}^{\mathrm{DLTD}}$ represents: Task Predictor with Weighted Voting, Dynamic Learning-Rate and automatic Task Detection[3]. ODIN Dynamic Learning-Rate used the same learning rate decay factor (0.999995) as in [60]'s continuous learning rate decay.

All experiments were run using the Avalanche [94] CL platform. The online buffer implementations of ER and MIR were from [4]. Best performing ODIP from chapter 5 without and with Task Detection: ODIP $\mathrm{NB}_{\mathrm{WV}}$ and ODIP $\mathrm{NB}_{\mathrm{TD}\beta}$ are also considered in the evaluation. But as pointed out in section 6.1, training a pool of NNs, makes it difficult to compare ODIP against other methods which train a single NN. Nonetheless, ODIP is considered to understand the advantages or disadvantages of training a pool of NNs. Average accuracy and forgetting, defined in [4] were used in the evaluation. All experiments were run three times, and relevant averages and standard deviations were considered in the evaluation.

Table 6.1 contains the average accuracy and forgetting after training on the last task for each method that uses task ids. Considering the average accuracy ranks, ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DL}}$ produces the best results. It also has very little forgetting, considering average forgetting ranks. Extended $\mathrm{ER}_{TbCb}$ has better average accuracy but lags a bit behind ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DL}}$ when considering the average accuracy ranks. All ODIN versions achieve better accuracy than the regularization baselines EWC and LwF except for ODIN random. Both of the regularization baselines have quite a high forgetting rate. ODIN $\mathrm{NB}_{\mathrm{WV}}$ yields better results than ODIN $\mathrm{NB}_{No\mathrm{WV}}$ with less average forgetting. This shows that Weighted Voting boosts accuracy. Also, ODIN $\mathrm{NB}_{\mathrm{WV}}$ produces better accuracy than ODIN $\mathrm{HT}_{\mathrm{WV}}$ with less forgetting. This indicates that NB is a better Task Predictor compared to HT. This is further explored in later experiments. Both $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DL}}$ and $\mathrm{NB}_{No\mathrm{WV}}^{\mathrm{DL}}$ yield superior accuracy compared to $\mathrm{NB}_{\mathrm{WV}}$ and $\mathrm{NB}_{No\mathrm{WV}}$.

[3]In the legend of the plots, superscripts and subscripts are in lowercase letters

Table 6.1: Average accuracy and forgetting after training on the last task (use end of the task signal)

| dataset | LwF | EWC | ER$_{TbCb}$ | ODIN | | | | | | | | ODIP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | known$^{*}_{tid}$ | random | MV | HT$_{\mathrm{WV}}$ | NB$_{\mathrm{WV}}$ | NB$^{\mathrm{DL}}_{\mathrm{WV}}$ | NB$_{No\mathrm{WV}}$ | NB$^{\mathrm{DL}}_{No\mathrm{WV}}$ | NB$_{\mathrm{WV}}$ |
| Accuracy | | | | | | | | | | | | |
| CORe50 | 0.44 ± 0.02 | 0.47 ± 0.03 | 0.65 ± 0.01 | 0.69 ± 0.01 | 0.44 ± 0.00 | 0.54 ± 0.02 | 0.63 ± 0.05 | 0.62 ± 0.01 | **0.66 ± 0.01** | 0.62 ± 0.01 | 0.65 ± 0.01 | **0.66 ± 0.01** |
| Rotated | | | | | | | | | | | | |
| CIFAR10 | 0.44 ± 0.01 | 0.42 ± 0.01 | 0.42 ± 0.02 | 0.49 ± 0.01 | 0.37 ± 0.02 | **0.45 ± 0.01** | **0.45 ± 0.01** | 0.43 ± 0.02 | 0.44 ± 0.01 | 0.40 ± 0.00 | 0.41 ± 0.00 | 0.43 ± 0.02 |
| Rotated | | | | | | | | | | | | |
| MNIST | 0.66 ± 0.01 | 0.52 ± 0.01 | **0.84 ± 0.01** | 0.97 ± 0.00 | 0.48 ± 0.01 | 0.65 ± 0.02 | 0.52 ± 0.01 | 0.79 ± 0.00 | 0.80 ± 0.00 | 0.78 ± 0.01 | 0.78 ± 0.00 | 0.79 ± 0.00 |
| Avg | 0.51 ± 0.01 | 0.47 ± 0.02 | **0.64 ± 0.01** | 0.72 ± 0.01 | 0.43 ± 0.01 | 0.55 ± 0.02 | 0.53 ± 0.02 | 0.62 ± 0.01 | 0.63 ± 0.01 | 0.60 ± 0.00 | 0.62 ± 0.00 | 0.63 ± 0.01 |
| Avg Rank | 6.67 | 8.67 | 4.33 | | 11.00 | 5.67 | 5.33 | 4.67 | **2.67** | 7.67 | 5.67 | 3.67 |
| Forgetting | | | | | | | | | | | | |
| CORe50 | 0.19 ± 0.04 | 0.15 ± 0.05 | -0.01 ± 0.02 | 0.00 ± 0.00 | -0.01 ± 0.01 | **-0.03 ± 0.01** | 0.03 ± 0.04 | 0.01 ± 0.01 | 0.01 ± 0.00 | 0.00 ± 0.01 | 0.01 ± 0.00 | 0.01 ± 0.01 |
| Rotated | | | | | | | | | | | | |
| CIFAR10 | 0.01 ± 0.02 | 0.07 ± 0.02 | 0.04 ± 0.00 | 0.00 ± 0.00 | 0.02 ± 0.01 | **-0.03 ± 0.01** | 0.04 ± 0.02 | -0.02 ± 0.00 | -0.02 ± 0.01 | 0.00 ± 0.00 | 0.00 ± 0.00 | -0.01 ± 0.01 |
| Rotated | | | | | | | | | | | | |
| MNIST | 0.24 ± 0.01 | 0.60 ± 0.02 | 0.16 ± 0.01 | 0.00 ± 0.00 | 0.20 ± 0.02 | 0.12 ± 0.02 | 0.60 ± 0.02 | **0.12 ± 0.00** | 0.12 ± 0.01 | **0.12 ± 0.00** | 0.13 ± 0.00 | **0.12 ± 0.00** |
| Avg | 0.14 ± 0.02 | 0.27 ± 0.03 | 0.06 ± 0.01 | 0.00 ± 0.00 | 0.07 ± 0.01 | **0.02 ± 0.02** | 0.22 ± 0.02 | 0.04 ± 0.01 | 0.03 ± 0.00 | 0.04 ± 0.00 | 0.04 ± 0.00 | 0.04 ± 0.01 |
| Avg Rank | 9.00 | 10.67 | 6.33 | | 6.00 | **2.00** | 9.67 | 4.00 | 2.67 | 5.00 | 5.67 | 5.00 |

!ER$_{TbCb}$ is ER with an extended task-balanced and class-balanced online buffer.

*known$_{tid}$ is a hypothetical scenario where task id is known at evaluation.

This suggests that Dynamic Learning-Rate improves accuracy. In general, a good Task Predictor, Weighted Voting, and Dynamic Learning-Rate improve ODIN's accuracy. Considering the hypothetical ODIN known$_{tid}$ scenario, it is evident that just selecting the correct frozen CNN is sufficient to outperform current baselines by a considerable margin. ODIN known$_{tid}$ also has zero average forgetting across all datasets after training on the last task. This suggests further improvements to the Task Predictors can result in good accuracy gains. Finally, when one considers ODIP NB$_{\mathrm{WV}}$ results training a pool of networks does not make much difference when task id is available at the training time.

Table 6.2: Average accuracy and forgetting after training on the last task (do not use end of the task signal)

| dataset | ODIN | | ER | MIR | ODIP |
|---|---|---|---|---|---|
| | NB$_{\mathrm{WV}}^{\mathrm{TD}}$ | NB$_{\mathrm{WV}}^{\mathrm{DLTD}}$ | | | NB$_{\mathrm{TD}\beta}$ |
| Accuracy | | | | | |
| CORe50 | 0.47 ± 0.04 | 0.52 ± 0.03 | 0.61 ± 0.03 | **0.62 ± 0.03** | 0.47 ± 0.04 |
| RotatedCIFAR10 | 0.42 ± 0.01 | **0.44 ± 0.00** | 0.27 ± 0.01 | 0.28 ± 0.01 | 0.42 ± 0.01 |
| RotatedMNIST | 0.71 ± 0.03 | 0.69 ± 0.04 | 0.78 ± 0.03 | 0.77 ± 0.01 | **0.79 ± 0.00** |
| Avg | 0.53 ± 0.02 | 0.55 ± 0.02 | 0.56 ± 0.02 | 0.56 ± 0.02 | **0.56 ± 0.02** |
| Avg Rank | 4.00 | 3.00 | 3.00 | 2.67 | **2.33** |
| Forgetting | | | | | |
| CORe50 | 0.18 ± 0.03 | 0.17 ± 0.03 | -0.09 ± 0.01 | **-0.10 ± 0.04** | 0.20 ± 0.03 |
| RotatedCIFAR10 | 0.00 ± 0.01 | 0.00 ± 0.00 | 0.01 ± 0.01 | 0.01 ± 0.01 | **-0.01 ± 0.00** |
| RotatedMNIST | 0.22 ± 0.04 | 0.25 ± 0.04 | **0.06 ± 0.02** | 0.07 ± 0.02 | 0.12 ± 0.00 |
| Avg | 0.14 ± 0.03 | 0.14 ± 0.03 | -0.01 ± 0.02 | **-0.01 ± 0.02** | 0.10 ± 0.01 |
| Avg Rank | 3.33 | 3.67 | 2.67 | **2.33** | 3.00 |

Table 6.2 only compares ODICL methods that do not use task ids: ODIN NB$_{\mathrm{WV}}^{\mathrm{TD}}$, ODIN NB$_{\mathrm{WV}}^{\mathrm{DLTD}}$, ER, MIR and ODIP NB$_{\mathrm{TD}\beta}$, for a fairer comparison. Here, the ODIN versions and ODIP NB$_{\mathrm{TD}\beta}$ use ADWIN as a Task Detection. Also, ER and MIR can be included in the same category as they do not rely on an external task id signal. Contrary to the previous setting, having a pool of training networks in ODIP NB$_{\mathrm{TD}\beta}$ seems to help with slightly

better average accuracy when the task id is unavailable. If one ignores ODIP $\mathrm{NB}_{\mathrm{TD}\beta}$ from the results for this category, it is evident that replay methods slightly outperform ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DLTD}}$. Also, replay methods have better forgetting in this category. However, compared to the ODIN methods, they perform quite badly on RotatedCIFAR10. Maybe being task aware gives ODIN methods an edge against replay methods on RotatedCIFAR10. Here also, one can see the positive effect of DL on ODIN's accuracy when comparing $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DLTD}}$ with $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{TD}}$. Considering the results in tables 6.1 and 6.2, it is evident that ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DL}}$ performs better than ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DLTD}}$. This highlights the importance of good Task Detection. Furthermore, when comparing the two tables, $\mathrm{ER}_{TbCb}$ performs better than ER. This highlights the importance of ER to be task aware. In general, when one considers the results of both tables, it is evident that being task aware gives an edge to an ODICL method. Considering the results in both tables, one can conclude that $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DLTD}}$ performs well in all the datasets.

To get a deeper understanding of each method's predictive performance on old tasks after training on a new task, figure 6.2, figure 6.3, and figure 6.4 plot the accuracy of old tasks after training on a new task for selected methods on CORe50, RotatedCIFAR10, and RotatedMNIST datasets. Here top two ODIN methods from each category (with and without the end of the task signal): ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DL}}$, ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DLTD}}$ were compared against the baselines that use the end of the task signal: EWC, LwF, $\mathrm{ER}_{TbCb}$, and baselines that do not need the end of the task signal: ER and MIR. Hypothetical ODIN $\mathrm{known}_{tid}$ is also included in the plots to better understand the upper bound of ODIN's Task Predictor. ODIP variants were not considered in this in-depth study, due the incompatibility caused by using a pool of NNs at training. From figure 6.2, it is evident that replay methods perform quite well on past tasks. Especially task-aware $\mathrm{ER}_{TbCb}$. However, their performance has degraded for recent tasks. On the other hand, ODIN $\mathrm{NB}_{\mathrm{WV}}^{\mathrm{DL}}$ has relatively stable performance across all tasks.

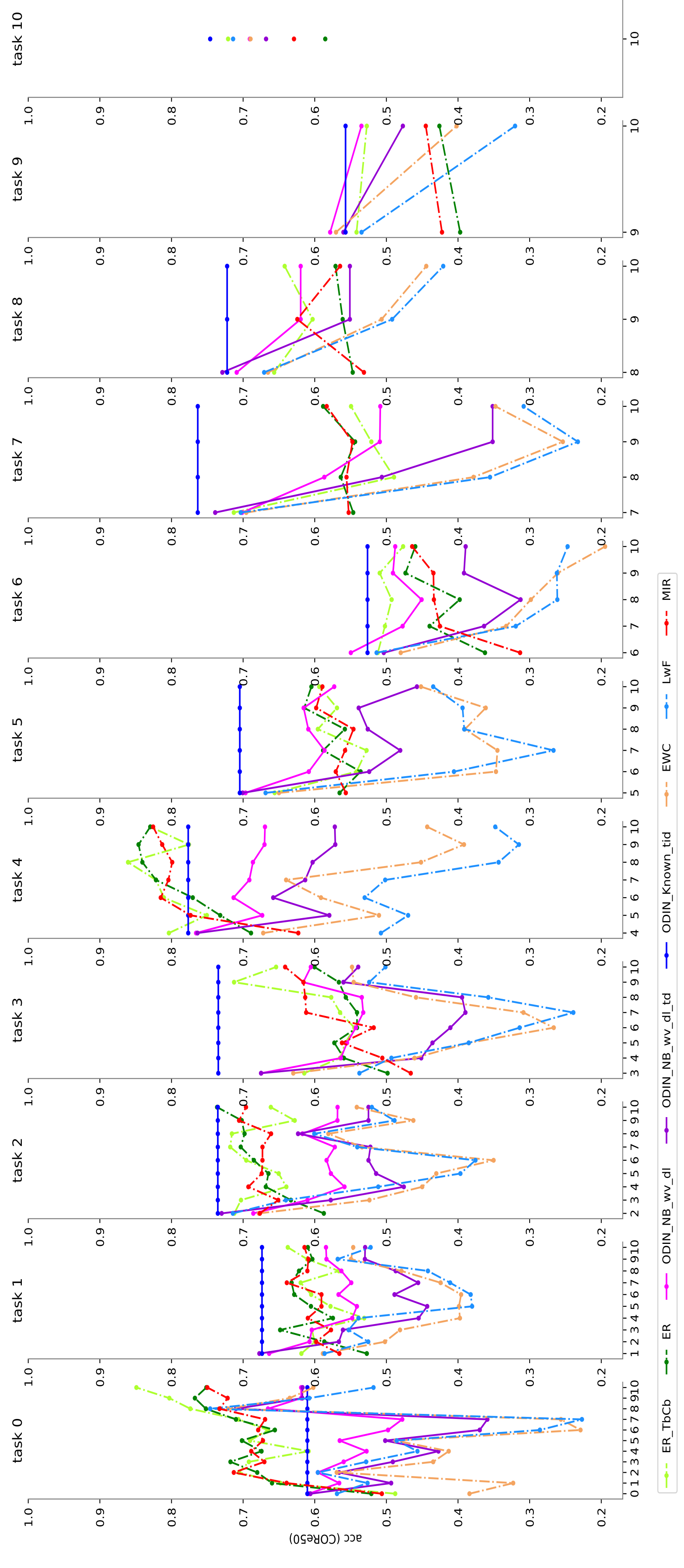


Figure 6.2: Evaluation accuracy for each task after training on a given task for CORe50

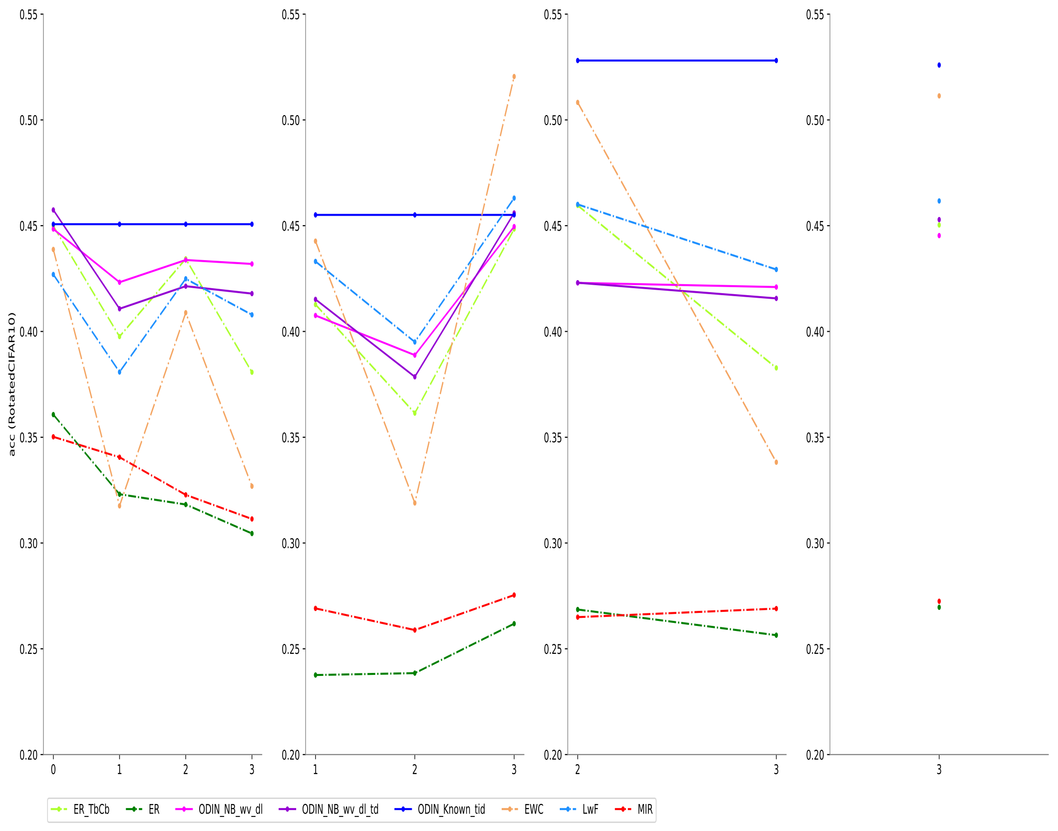


Figure 6.3: Evaluation accuracy for each task after training on a given task for RotatedCIFAR10

Hence on average, ODIN $NB_{WV}^{DL}$ performs well on CORe50. This explains its good average accuracy on CORe50 in table 6.1. Also, ODIN $NB_{WV}^{DLTD}$ has a similar accuracy pattern to ODIN $NB_{WV}^{DL}$. But with less performance. Regularization baselines are quite poor on this dataset. They also have a very high variance. However, as per figure 6.3, LwF performs well as ODIN versions on RotatedCIFAR10. Nevertheless, the replay methods ER and MIR perform poorly on that dataset except for $ER_{TbCb}$. It seems that the learning model needs to be aware of the task identities to perform well on RotatedCIFAR10. As per figure 6.4, replay baselines generally perform well on RotatedMNIST. However, except for $ER_{TbCb}$, the performance gap between ODIN $NB_{WV}^{DL}$ and other replay methods (ER and MIR) seem to narrow for recent tasks. ODIN $NB_{WV}^{DL}$ performed better on the last task than ER and MIR on this dataset.

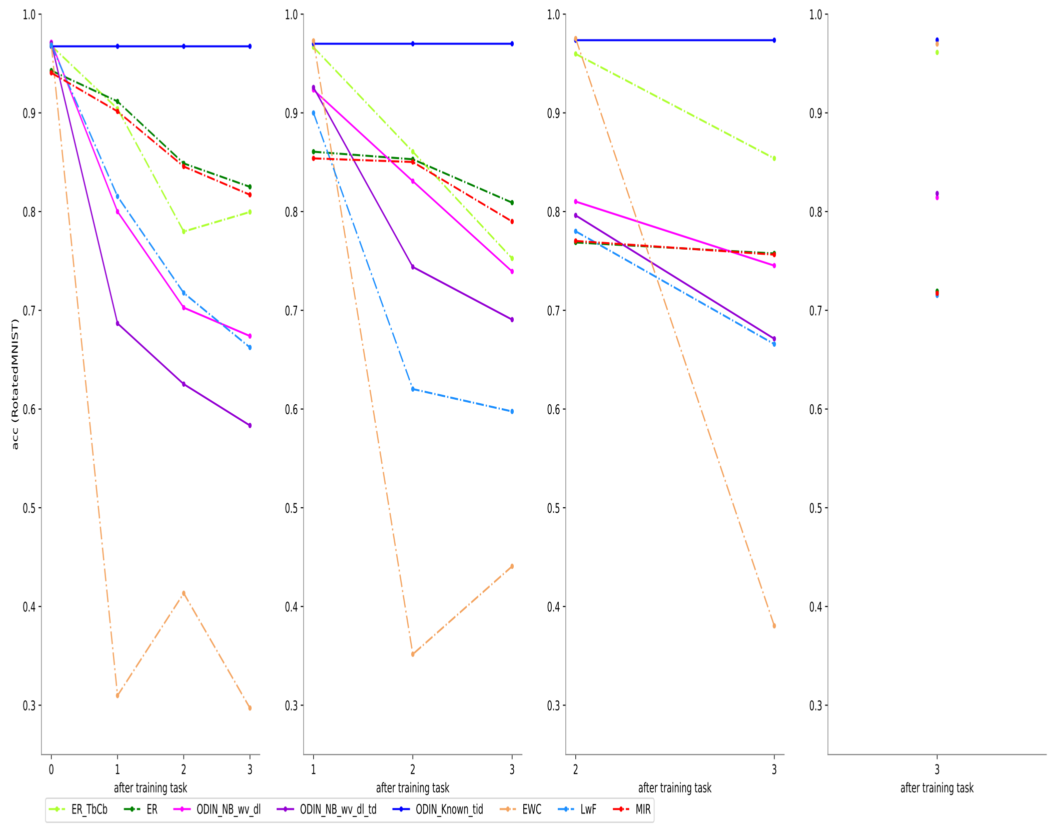


Figure 6.4: Evaluation accuracy for each task after training on a given task for RotatedMNIST

This shows ODIN $\text{NB}_{\text{WV}}^{\text{DL}}$'s ability to perform well on current tasks as well as on past tasks. In all three plots, ODIN with hypothetical TP $\text{known}_{tid}$ never forgets after training on a new task. However, it does not improve as well. This explains 0.0 average forgetting for ODIN $\text{known}_{tid}$ in table 6.1.

To further understand the TP's effectiveness, the predicted task id by each TP was compared against the actual task id in non-auto-TD mode against all datasets. This comparison was made for all evaluation instances after training on the last task. Figures 6.5, 6.6 , 6.7 and 6.8 show the ROC curves for the predicted task id and the relevant AUC scores for each TP on each dataset. According to the figures, it is clear that NB is a better Task Predictor for all datasets. This further strengthens the overall strong NB results in table 6.1. Figure 6.6 further explains the effectiveness of NB as a TP when predicting

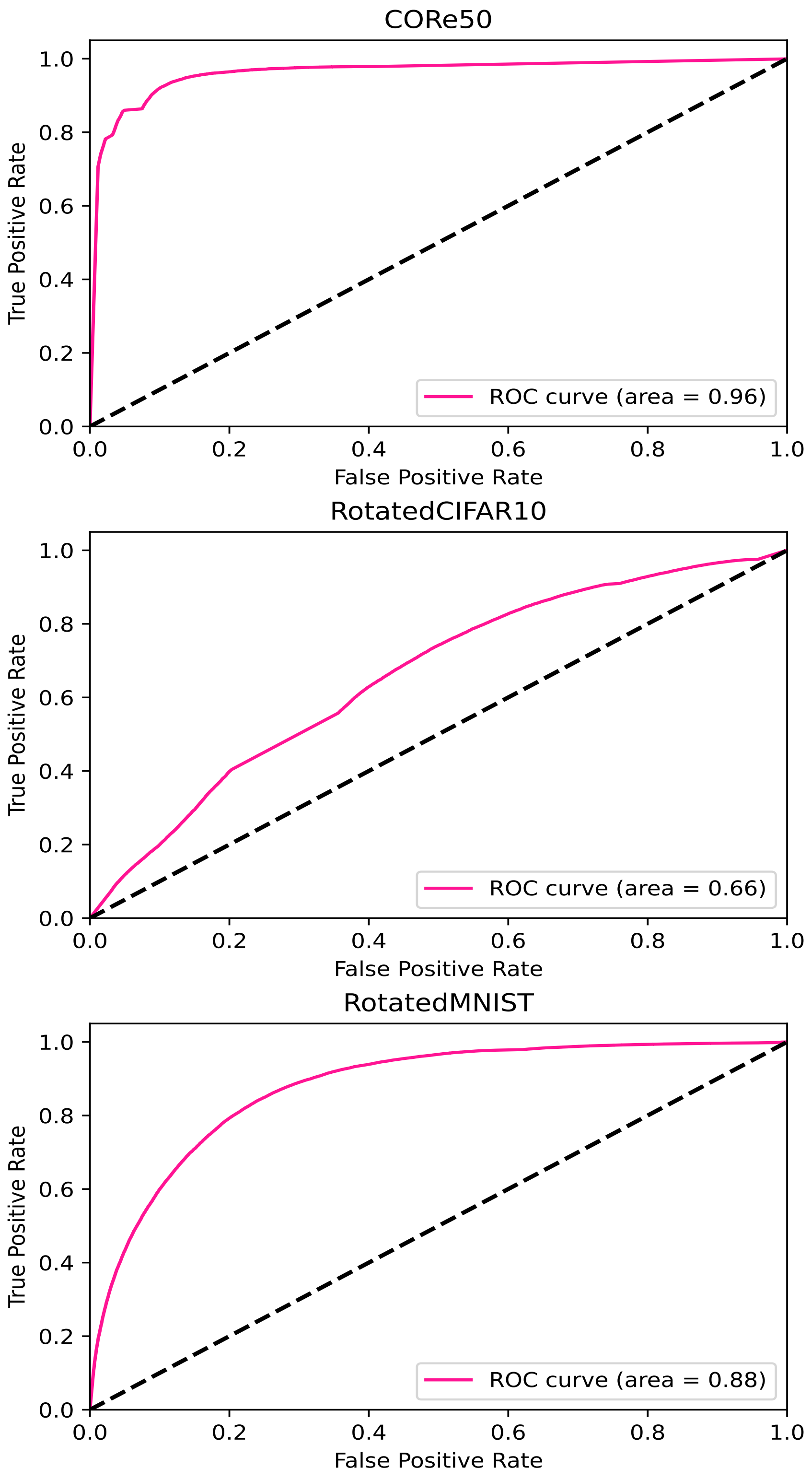


Figure 6.5: Effectiveness of NB as a TP considering predicted task id. Micro-average ROC curves and AUC scores.

each task for a given dataset. From the per-task ROC curves and AUC scores in figures 6.6 and 6.8, it is clear that NB performs similarly on all the tasks for a given dataset. Nevertheless, it does perform slightly better on certain

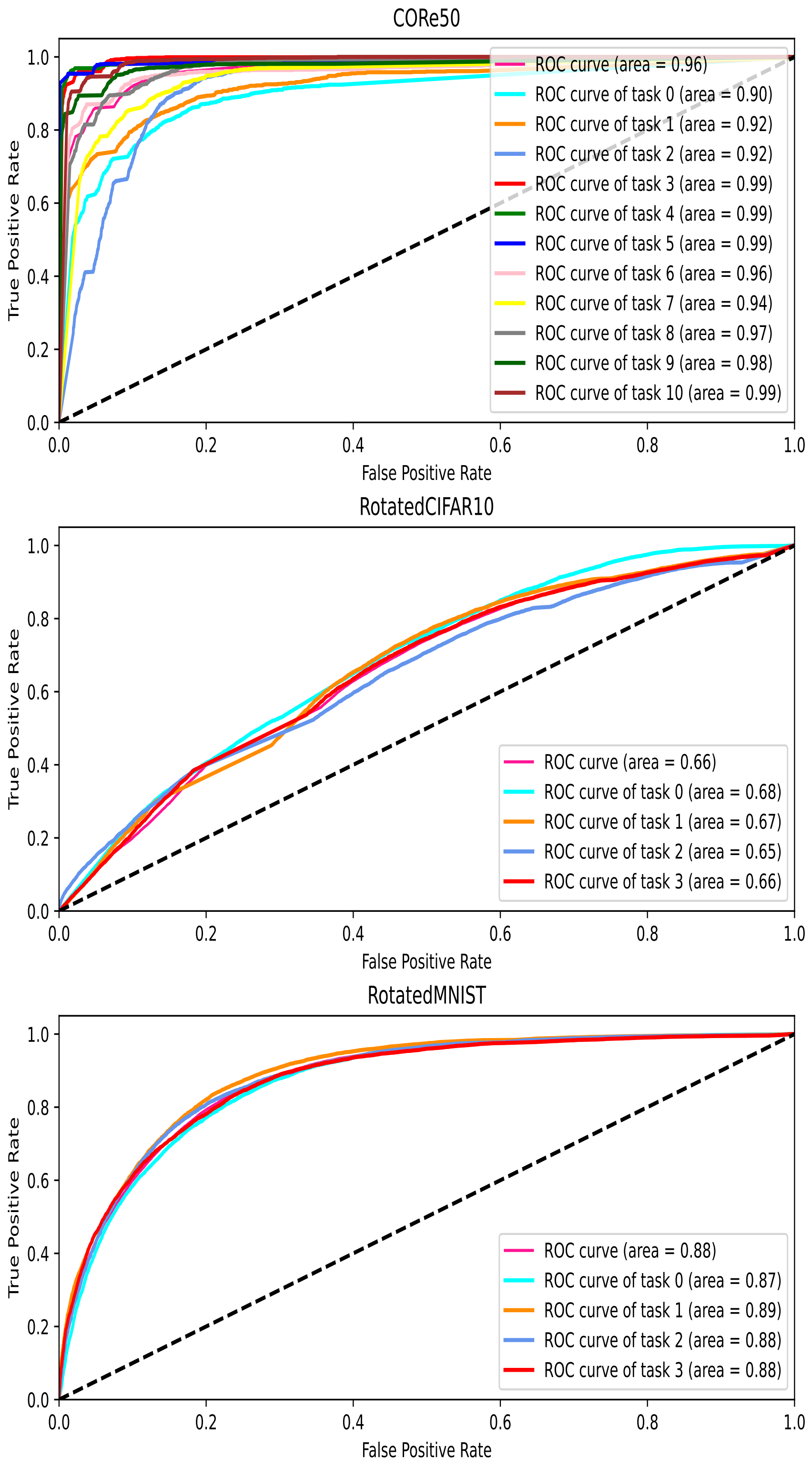


Figure 6.6: Effectiveness of NB as a TP considering predicted task id. Per-task ROC curves and AUC scores.

tasks. This is evident in CORe50, with NB performing slightly better for tasks 3,4,5,9 and 10. NB's generally uniform predictive capability makes it a better Task Predictor than HT.

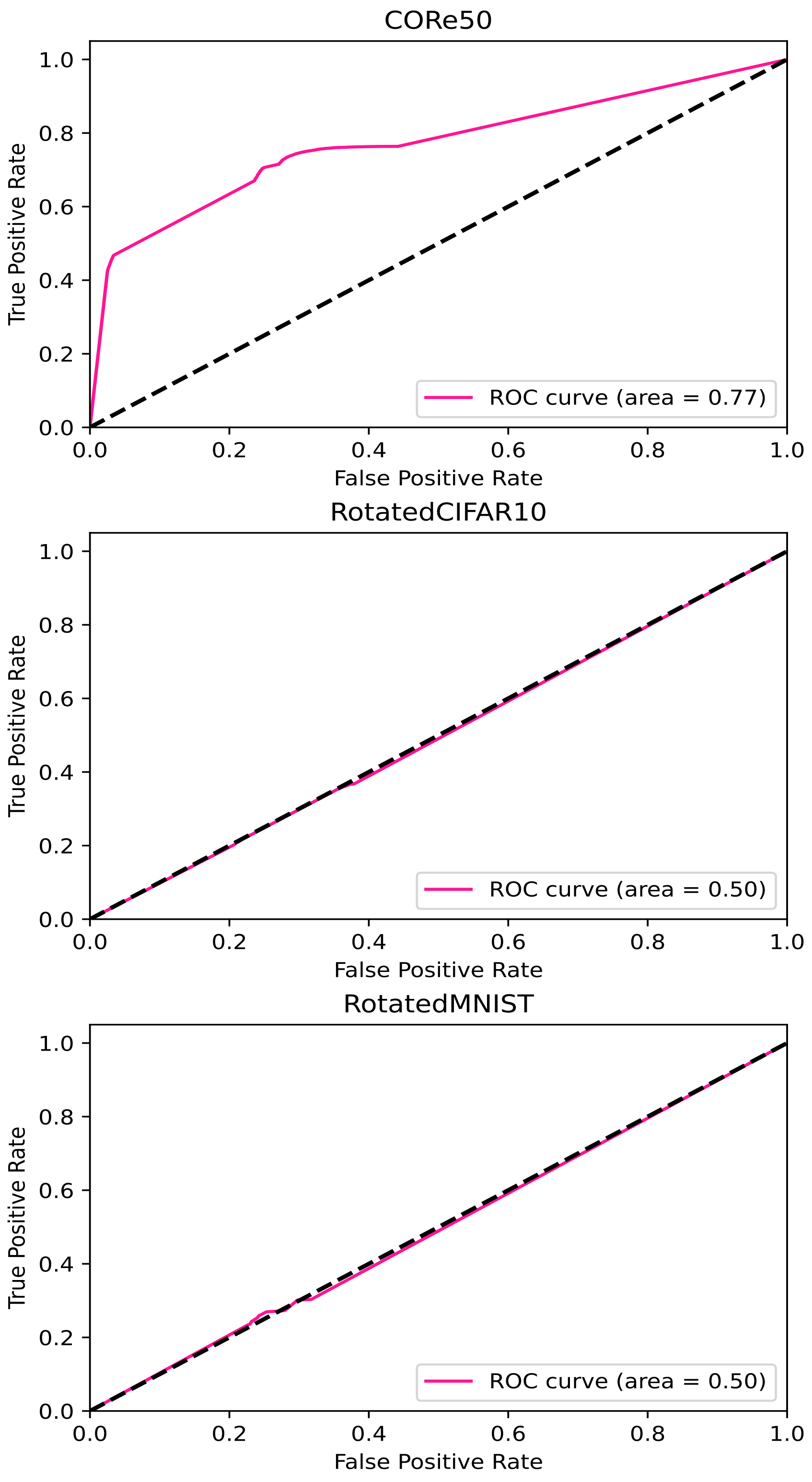


Figure 6.7: Effectiveness of HT as a TP considering predicted task id. Micro-average ROC curves and AUC scores.

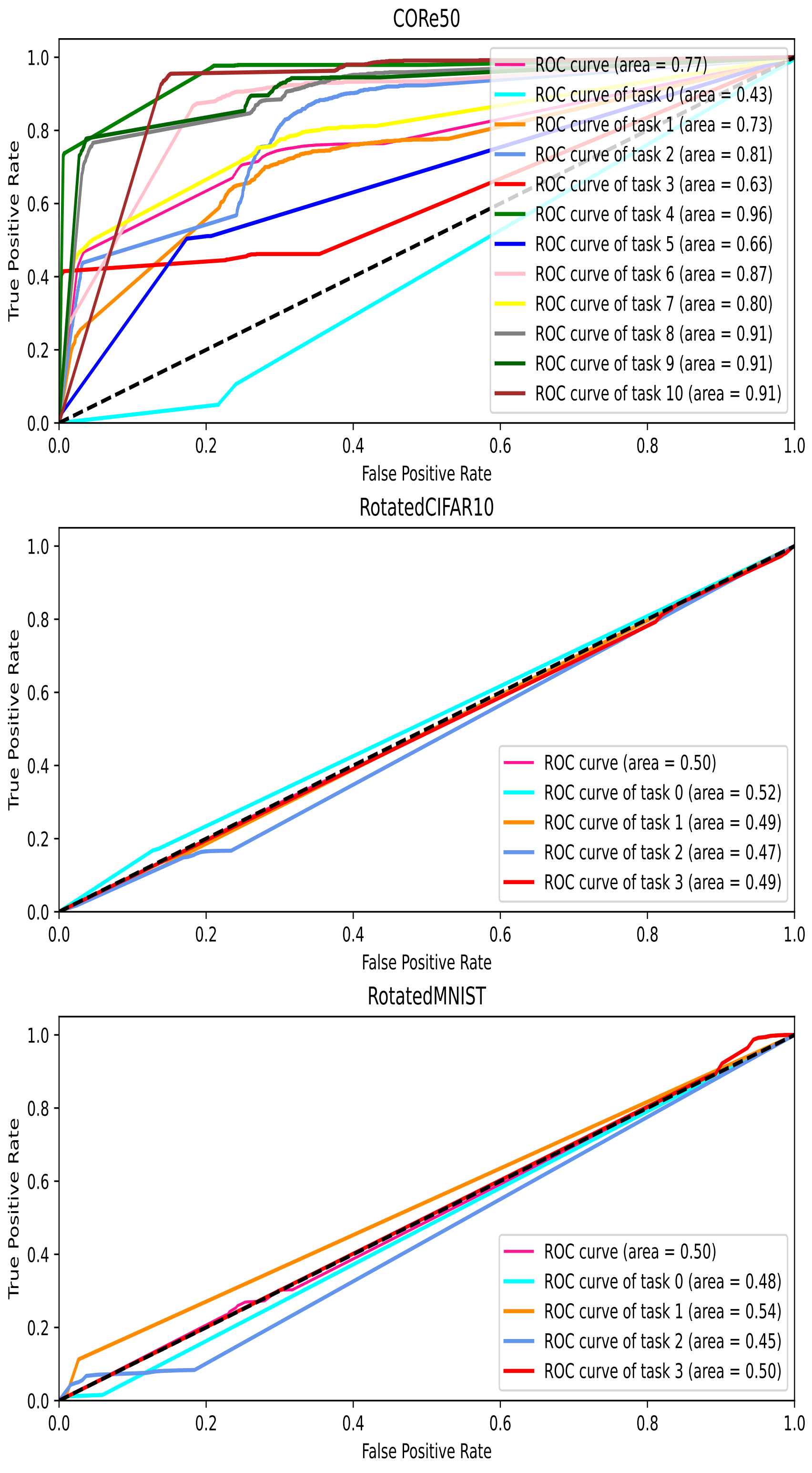


Figure 6.8: Effectiveness of HT as a TP considering predicted task id. Per-task ROC curves and AUC scores.

# Chapter 7

# Conclusions and Future Work

The thesis introduces a novel gradient-boosted tree algorithm, called Streaming Gradient Boosted Trees, for SL classification that outperforms existing bagging and random forest methods. Streaming gradient boosting is challenging due to adapting the booster after a concept drift without sacrificing performance. SGBT addresses this challenge by using streaming regression trees with an internal Tree Replacement strategy, enabling them to adapt to drifts dynamically. Experimental results demonstrate that SGBT, specifically the SGBT$^{MC}$ variant with FIMT-DD as the base learner, achieves superior performance compared to state-of-the-art methods when applied to large evolving datasets with multiple drifts.

Recognizing the significance of Neural Networks in batch learning, the thesis explores their integration into evolving data stream classification. However, incorporating NN into the SL poses the challenge of hyperparameter optimization, which is crucial for achieving satisfactory network performance. The thesis proposes **C**ontinuously **A**daptive **N**eural Networks for **D**ata Streams to address this challenge. CAND dynamically selects the best NN from a pool of NNs for Stream Learning classification, employing online hyperparameter selection. Experimental results demonstrate that CAND outperforms current state-of-the-art Stream Learning methods, particularly on high-dimensional data. Notably, CAND$^{SB}_{sub}$, a variant that selectively updates only some networks, balances computation efficiency and accuracy. Additionally, the experiments indicate that small mini-batches achieve similar accuracy to single-instance fully incremental training, even on evolving data streams. $\text{CAND}_{sub}^{SB}$ effectively mitigates overfitting by avoiding costly backpropagation for instances that yield minimal loss. These promising results set the stage for exploring more complex NN architectures in future CAND variants.

Certain concepts from CAND, such as estimating the loss of an NN to reveals its current performance and NN's loss resembles drifts in input data, are leveraged in chapters 5 and 6 to develop two novel methods for Online Domain Incremental Continual Learning: ODIP and ODIN. Both methods build upon Online Streaming Continual Learning techniques outlined in Section 2.1.3.

ODIP maintains a pool of NNs and freezes the best-performing one after training on each task. It also trains a Task Predictor as an additional stream learner to select the most suitable NN from the frozen pool for prediction. ODIP delivers competitive results in ODICL compared to regularization-based approaches. An extended version of ODIP incorporates drift detectors to automatically detect tasks in ODICL based on each NN's loss. ODIP with and without automatic Task Detection achieve competitive results compared to popular regularization baselines, such as LwF and EWC. Consequently, ODIP emerges as a promising alternative to regularization methods in the ODICL setting.

In contrast, ODIN focuses on training a single NN instead of a pool of NNs like ODIP. It employs similar Task Detection and Task Predictor mechanisms as ODIP. But, extends the Online Streaming Continual Learning techniques used in ODIP to incorporate Dynamic Learning-Rate, considering upward and downward drifts detected by the Task Detection component. ODIN surpasses current regularization-based approaches and achieves competitive results comparable to popular replay-based methods for ODICL without requiring an instance buffer. As a result, ODIN is an appropriate choice for privacy-concerned ODICL scenarios.

## 7.1 Future Work

SGBT-calculated *hessian* weights often result in fractions for most loss functions. To our knowledge, none of the existing streaming regression trees supports non-integer weights. SGBT assigns a weight of 1 or a transformed weight that yields a positive integer to the base learner as a workaround. However, as a future direction, it would be valuable to investigate the incremental calculation of variance and covariance, as proposed in [85] and [84], to support fractional weights for SGT and FIMT-DD base learners. Furthermore, one could include regularization constraints in the base learners to avoid large trees. The easiest method would be to have a memory constraint like in Hoeffding Tree. Another avenue for future SGBT research involves selectively skipping the training of certain instances based on the loss exceeding a certain threshold rather than randomly skipping instances. This approach would lead to computational and memory savings without significantly compromising accuracy.

We could consider looking into more intricate Neural Network architectures for future CAND variants, as this would open up opportunities for application in diverse domains. Also, only the smaller $M$ pool could be loaded into the GPU memory when using GPUs for training and prediction to reduce the GPU memory usage.

It could be beneficial to incorporate a fixed-size frozen pool size to avoid an ever-growing frozen pool for both ODIP and ODIN. For that, information from the Task Predictor could be leveraged to identify the CNN to train or replace when the frozen pool is full. Better Task Predictors and more responsive Task Detection mechanisms also could improve the performance of ODIP and ODIN. More effective Dynamic Learning-Rate mechanisms could improve ODIN's performance in ODICL.

# Appendix A

# Chapter 3 Additional Results

## A.1 KappaM results

Table A.1 and figure A.1 contains KappM results for learners: SGBT$^{MC}$, SRP, ARF, OSB and AdIter on all datasets discussed in sect3.3. KappaM measures learner's performance against a majority class classifier [9]. It is used to evaluate learner's performance on an imbalanced dataset [9]. Here learner rankings in table A.1 and figure A.1 align with rankings in table 3.2 and figure 3.1.

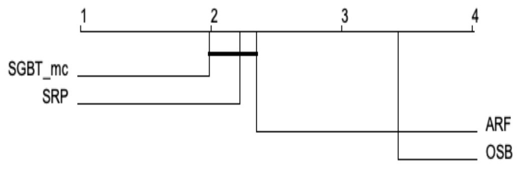


Figure A.1: Nemenyi Post-hoc test with p-value 0.05 for **all datasets** (*KappaM*): SGBT$^{MC}$ against other baselines (10 iterations with different random seeds).

## A.2 Results for different loss functions

Squared loss:

$$l^{SL}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{A.1}$$

Table A.1: KappaM (percentage): SGBT$^{MC}$ against other baselines (values rounded to 2 decimals).

| | SGBT$^{MC}$ | SRP | ARF | OSB | AdIter |
|---|---|---|---|---|---|
| binary class | | | | | |
| AGR$_a$ | **88.23 ± 0.02** | 84.76 ± 0.40 | 74.28 ± 0.16 | 79.62 ± 0.02 | 80.35 ± 0.37 |
| AGR$_g$ | **82.85 ± 0.02** | 78.11 ± 0.39 | 62.80 ± 0.23 | 74.29 ± 0.06 | 73.83 ± 0.72 |
| RBF_B$_m$ | 81.28 ± 2.96 | 78.41 ± 3.52 | **81.55 ± 2.98** | 74.63 ± 3.21 | 46.33 ± 3.03 |
| RBF_B$_f$ | 62.76 ± 6.67 | 58.13 ± 8.04 | **66.22 ± 6.91** | 48.39 ± 5.26 | 35.39 ± 4.54 |
| RandomTree | 67.72 ± 17.18 | 69.50 ± 7.06 | 74.98 ± 10.45 | **80.31 ± 7.28** | 25.36 ± 19.02 |
| electricity | 72.91 ± 0.14 | 75.69 ± 0.33 | **77.91 ± 0.12** | 75.30 ± 0.00 | 49.99 ± 0.19 |
| airlines | **29.93 ± 0.07** | 29.37 ± 0.11 | 25.20 ± 0.06 | 20.43 ± 0.00 | 16.31 ± 0.17 |
| avg | **69.38** | 67.71 | 66.13 | 64.71 | 46.79 |
| rank | **2.14** | 2.43 | 2.57 | 3.29 | 4.57 |
| multi class | | | | | |
| LED$_a$ | **71.11 ± 0.01** | **71.11 ± 0.01** | 71.01 ± 0.01 | 69.37 ± 0.01 | |
| LED$_g$ | **70.31 ± 0.01** | 70.23 ± 0.01 | 70.08 ± 0.01 | 68.96 ± 0.01 | |
| RBF$_m$ | **83.12 ± 1.56** | 81.14 ± 1.79 | 82.85 ± 1.58 | 67.40 ± 2.13 | |
| RBF$_f$ | 67.61 ± 2.94 | 67.51 ± 2.85 | **68.59 ± 3.04** | 30.73 ± 3.08 | |
| LED | 70.82 ± 0.15 | **70.87 ± 0.13** | 70.73 ± 0.16 | 70.86 ± 0.20 | |
| RBF5 | 86.10 ± 1.55 | 86.70 ± 1.74 | **86.76 ± 1.77** | 79.84 ± 2.03 | |
| covtype | 88.85 ± 0.05 | **90.91 ± 0.03** | 89.69 ± 0.04 | 85.74 ± 0.00 | |
| avg | 76.84 | 76.92 | **77.10** | 67.56 | |
| rank | **2.00** | **2.00** | 2.29 | 3.71 | |
| avg (both) | **73.11** | 72.32 | 71.62 | 66.13 | |
| rank (both) | **2.07** | 2.21 | 2.43 | 3.50 | |

For $l^{SL}$, gradient ($g$) is $y - \hat{y}$, and hessian ($h$) is 1.

Table A.3 summarizes the results of the study aimed at investigating the impact of the loss function. The loss functions evaluated in this study were the squared loss and the categorical cross-entropy loss. For squared loss, no special weight handling was required as the hessian is 1. All the other parameters: base learner (FIMT-DD), learning rate ($lr$=1.25e-02), feature percentage ($m$=60%), and boosting iterations ($s$=80), were set unchanged. From the results, it is evident that categorical cross-entropy loss outperforms squared loss as a loss function for SGBT$^{MC}$ across all data sets.

Table A.2: Average Accuracy and evaluation time(s) with Standard Deviation for $\text{SGBT}^{MC}_{SK\_1/k}[S = 100, m = 75, lr = 1.25\text{e-}2]$
(values rounded to 2 decimals, 4 decimals were considered to select the winner)

| | Accuracy (%) | | | Time (s) | | |
|---|---|---|---|---|---|---|
| | $\text{SGBT}^{MC}_{SK\_1/k}$ | | $\text{SGBT}^{MC}$ | $\text{SGBT}^{MC}_{SK\_1/k}$ | | $\text{SGBT}^{MC}$ |
| $k$ | 2 (skip 1/2) | 3 (skip 1/3) | 1 (no skip) | 2 (skip 1/2) | 3 (skip 1/3) | 1 (no skip) |
| binary class | | | | | | |
| $\text{AGR}_\text{a}$ | 94.32+0.02 | 94.36+0.01 | **94.45+0.01** | **719.16+64.47** | 818.16+7.11 | 1,386.55+51.40 |
| $\text{AGR}_\text{g}$ | 91.81+0.01 | 91.83+0.03 | **91.92+0.01** | **663.56+5.70** | 810.28+27.69 | 1,359.81+74.20 |
| $\text{RBF\_B}_\text{m}$ | 90.55+0.17 | 91.20+0.15 | **91.85+0.11** | **983.85+18.69** | 1,218.42+97.26 | 2,126.22+299.86 |
| $\text{RBF\_B}_\text{f}$ | 78.11+0.31 | 80.89+0.20 | **84.12+0.13** | **891.39+23.25** | 1,061.86+67.40 | 1,749.00+251.29 |
| RandomTree | 84.73+8.10 | 85.62+8.99 | **85.72+9.40** | **74.62+3.95** | 93.09+8.07 | 148.69+6.50 |
| electricity | 85.88+0.05 | 87.10+0.26 | **88.54+0.03** | **31.43+2.72** | 37.43+0.67 | 54.75+4.68 |
| airlines | 67.99+0.03 | 68.31+0.07 | **68.77+0.03** | **309.32+12.37** | 378.30+12.75 | 584.59+47.24 |
| avg | 84.77 | 85.62 | **86.48** | **524.76** | 631.07 | 1,058.52 |
| rank | 3.00 | 2.00 | **1.00** | **1.00** | 2.00 | 3.00 |
| multi class | | | | | | |
| $\text{LED}_\text{a}$ | 73.91+0.01 | 73.97+0.03 | **74.05+0.01** | **1,002.87+17.89** | 1,197.72+101.18 | 1,747.52+149.75 |
| $\text{LED}_\text{g}$ | 73.18+0.01 | 73.22+0.01 | **73.32+0.01** | **977.22+25.89** | 1,217.96+57.35 | 1,718.98+128.82 |
| $\text{RBF}_\text{m}$ | 86.17+0.82 | 87.01+0.73 | **87.96+0.63** | **1,441.83+210.95** | 1,875.82+227.84 | 2,773.24+164.02 |
| $\text{RBF}_\text{f}$ | 68.75+1.97 | 72.62+1.58 | **77.03+1.39** | **1,395.00+92.88** | 1,683.68+38.21 | 2,439.11+200.90 |
| LED | 73.80+0.20 | 73.76+0.17 | **73.81+0.19** | **91.99+6.92** | 121.11+9.15 | 162.38+15.80 |
| RBF5 | 88.51+0.82 | 89.06+0.79 | **89.76+0.72** | **153.97+2.43** | 201.30+13.50 | 279.21+32.25 |
| covtype | 92.23+0.00 | 93.15+0.00 | **94.28+0.00** | **1,005.95+49.29** | 1,318.55+48.46 | 1,944.42+262.60 |
| avg | 79.51 | 80.40 | **81.46** | **866.97** | 1,088.02 | 1,580.69 |
| rank | 2.86 | 2.14 | **1.00** | **1.00** | 2.00 | 3.00 |
| avg (both) | 82.14 | 83.01 | **83.97** | **695.87** | 859.55 | 1,319.61 |
| rank (both) | 2.93 | 2.07 | **1.00** | **1.00** | 2.00 | 3.00 |

Table A.3: Test then train accuracy of SGBT$^{MC}$ [$S = 80$, $m = 60$, $lr$ =1.25e-2, FIMT-DD] for different loss functions.Values rounded to 2 decimals, and 4 decimals were considered to select the winner.

| loss | categorical cross-entropy $l^{CE}$ | squared $l^{SL}$ |
|---|---|---|
| binary class | | |
| AGR$_a$ | **93.63+0.02** | 93.45+0.00 |
| AGR$_g$ | **91.09+0.02** | 90.96+0.02 |
| RBF_B$_m$ | **91.45+0.25** | 91.22+0.28 |
| RBF_B$_f$ | **83.62+0.89** | 82.85+1.31 |
| RandomTree | **83.10+6.95** | 82.09+7.45 |
| electricity | **88.39+0.04** | 87.57+0.09 |
| airlines | **68.76+0.04** | 68.63+0.02 |
| avg | **85.72** | 85.25 |
| rank | **1.00** | 2.00 |
| multi class | | |
| LED$_a$ | **73.75+0.01** | 69.28+0.01 |
| LED$_g$ | **72.95+0.01** | 68.64+0.01 |
| RBF$_m$ | **86.57+0.88** | 83.76+2.00 |
| RBF$_f$ | **74.47+2.15** | 68.72+4.68 |
| LED | **73.46+0.13** | 69.23+0.21 |
| RBF5 | **88.99+0.89** | 87.77+1.56 |
| covtype | **94.08+0.05** | 91.49+0.05 |
| avg | **80.61** | 76.98 |
| rank | **1.00** | 2.00 |
| avg (both) | **83.16** | 81.12 |
| rank (both) | **1.00** | 2.00 |

# References

[1] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning, 2022.

[2] Sergio Ramírez-Gallego, B Krawczyk, Salvador García, Michał Woźniak, and F Herrera. A survey on data preprocessing for data stream mining: Current status and future directions. *Neurocomputing*, 2017.

[3] Vinicius MA Souza, Denis M dos Reis, Andre G Maletzke, and Gustavo EAPA Batista. Challenges in benchmarking stream learning algorithms with real-world data. *Data Min. Knowl. Discov.*, 2020.

[4] Zheda Mai, Ruiwen Li, Jihwan Jeong, David Quispe, Hyunwoo Kim, and Scott Sanner. Online continual learning in image classification: An empirical survey. *Neurocomputing*, 2022.

[5] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga, Jinshi Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta Chauhan, Benjamin Lee, Hsien-Hsin Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. Sustainable ai: Environmental implications, challenges and opportunities. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 795–813, 2022.

[6] Zhenghua Chen, Min Wu, Alvin Chan, Xiaoli Li, and Yew-Soon Ong. Survey on ai sustainability: Emerging trends on learning algorithms and research challenges. *IEEE Computational Intelligence Magazine*, 18(2):60–77, 2023.

[7] Pablo Villalobos Jaime Sevilla and Juan Felipe Cerón. Parameter counts in machine learning, 2021. Accessed: 2023-5-10.

[8] Heitor M Gomes, J P Barddal, F Enembreck, and Albert Bifet. A survey on ensemble learning for data stream classification. *ACM (CSUR)*, 2017.

[9] Albert Bifet, Ricard Gavaldà, G Holmes, and Bernhard Pfahringer. *Machine learning for data streams: with practical examples in MOA*. MIT, 2018.

[10] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, G Desjardins, Andrei A Rusu, K Milan, J Quan, T Ramalho, Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *PNAS*, 2017.

[11] Maroua Bahri, Albert Bifet, João Gama, Heitor M Gomes, and Silviu Maniu. Data stream analysis: Foundations, major tasks and tools. *WIRE: Data Min. Knowl. Discov.*, 2021.

[12] Heitor M Gomes, M Grzenda, Rodrigo Mello, Jesse Read, Minh Huong Le Nguyen, and Albert Bifet. A survey on semi-supervised learning for delayed partially labelled data streams. *ACM Comput. Surv.*, 2022.

[13] Alaettin Zubaroğlu and Volkan Atalay. Data stream clustering: a review. *AI Review*, 2021.

[14] Heitor M Gomes, Albert Bifet, Jesse Read, J P Barddal, F Enembreck, Bernhard Pfahringer, G Holmes, and T Abdessalem. Adaptive random forests for evolving data stream classification. *ML*, 2017.

[15] Heitor M Gomes, Jesse Read, and Albert Bifet. Streaming random patches for evolving data stream classification. In *IEEE (ICDM)*. IEEE, 2019.

[16] Jacob Montiel, Rory Mitchell, Eibe Frank, Bernhard Pfahringer, Talel Abdessalem, and Albert Bifet. Adaptive xgboost for evolving data streams. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[17] Kun Wang, Jie Lu, Anjin Liu, Yiliao Song, Li Xiong, and Guangquan Zhang. Elastic gradient boosting decision tree with adaptive iterations for concept drift adaptation. *Neurocomputing*, 491:288–304, 2022.

[18] Shang-Tse Chen, Hsuan-Tien Lin, and Chi-Jen Lu. An online boosting algorithm with theoretical justifications. *arXiv preprint arXiv:1206.6422*, 2012.

[19] Andri Ashfahani and Mahardhika Pratama. Autonomous deep learning: Continual learning approach for dynamic environments. In *Proceedings of the 2019 SIAM International Conference on Data Mining*, pages 666–674. SIAM, 2019.

[20] Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2017.

[21] Arslan Chaudhry, Marcus Rohrbach, M Elhoseiny, T Ajanthan, P K Dokania, P H S Torr, and M Ranzato. On tiny episodic memories in continual learning. *arXiv:1902.10486*, 2019.

[22] Rahaf Aljundi, Lucas Caccia, Eugene Belilovsky, Massimo Caccia, Min Lin, Laurent Charlin, and Tinne Tuytelaars. Online continual learning with maximally interfered retrieval. *NeurIPS*, 2019.

[23] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *SIAM (SDM)*. SIAM, 2007.

[24] Andrés L. Suárez-Cetrulo, David Quintana, and Alejandro Cervantes. A survey on machine learning for recurring concept drifting data streams. *Expert Syst. Appl.*, 2023.

[25] Imen Khamassi, Moamar Sayed-Mouchaweh, Moez Hammami, and Khaled Ghédira. Self-adaptive windowing approach for handling complex concept drift. *Cogn. Comput.*, 2015.

[26] Abraham Wald. *Sequential analysis.* John Wiley, Oxford, England, 1947.

[27] Ewan S Page. Continuous inspection schemes. *Biometrika*, 1954.

[28] Joao Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with drift detection. In *SBIA*. Springer, 2004.

[29] Manuel Baena-Garcıa, José del Campo-Ávila, Raul Fidalgo, Albert Bifet, Ricard Gavalda, and Rafael Morales-Bueno. Early drift detection method. In *IWKDDS*. Citeseer, 2006.

[30] Imen Khamassi, Moamar Sayed-Mouchaweh, Moez Hammami, and Khaled Ghédira. Discussion and review on evolving data streams and concept drift adapting. *Evol. Syst.*, 2018.

[31] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM (CSUR)*, 2014.

[32] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *ACM SIGKDD*, 2001.

[33] Albert Bifet and Ricard Gavalda. Adaptive learning from evolving data streams. In *IDA*. Springer, 2009.

[34] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[35] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[36] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[37] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[38] Nikunj C Oza and Stuart J Russell. Online bagging and boosting. In *International Workshop on Artificial Intelligence and Statistics*, pages 229–236. PMLR, 2001.

[39] Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. Leveraging bagging for evolving data streams. In José Luis Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 135–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[40] Rocco A Servedio. Smooth boosting and learning with malicious noise. *The Journal of Machine Learning Research*, 4:633–648, 2003.

[41] Boyu Wang and Joelle Pineau. Online bagging and boosting for imbalanced data streams. *IEEE Transactions on Knowledge and Data Engineering*, 28(12):3353–3366, 2016.

[42] Mahardhika Pratama, Choiru Za'in, Andri Ashfahani, Yew Soon Ong, and Weiping Ding. Automatic construction of multi-layer perceptron network from streaming examples. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1171–1180, 2019.

[43] Andri Ashfahani, Mahardhika Pratama, Edwin Lughofer, and Yew-Soon Ong. Devdan: Deep evolving denoising autoencoder. *Neurocomputing*, 390:297–314, 2020.

[44] Monidipa Das, Mahardhika Pratama, Septiviana Savitri, and Jie Zhang. Muse-rnn: A multilayer self-evolving recurrent neural network for data

stream classification. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 110–119. IEEE, 2019.

[45] Peng Zhao, Xinqiang Wang, Siyu Xie, Lei Guo, and Zhi-Hua Zhou. Distribution-free one-pass learning. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[46] G. Cassales, H. Gomes, A. Bifet, B. Pfahringer, and H. Senger. Improving parallel performance of ensemble learners for streaming data through data locality with mini-batching. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2020.

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

[48] Bruno Veloso and João Gama. Self hyper-parameter tuning for stream classification algorithms. In *IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning*, pages 3–13. Springer, 2020.

[49] Bruno Veloso, João Gama, Benedita Malheiro, and João Vinagre. Hyperparameter self-tuning for data streams. *Information Fusion*, 76:75–86, 2021.

[50] Joao Gama, Raquel Sebastiao, and Pedro Pereira Rodrigues. On evaluating stream learning algorithms. *Mach. Learn*, 2013.

[51] Albert Bifet, Gianmarco de Francisci Morales, Jesse Read, Geoff Holmes, and Bernhard Pfahringer. Efficient online evaluation of big data stream classifiers. In *21th ACM SIGKDD*, 2015.

[52] Maciej Grzenda, Heitor M Gomes, and Albert Bifet. Delayed labelling evaluation for data streams. *Data Min. Knowl. Discov.*, 2020.

[53] Maciej Grzenda, H M Gomes, and A Bifet. Performance measures for evolving predictions under delayed labelling classification. In *IJCNN*, 2020.

[54] Hardy Kremer, Philipp Kranen, Timm Jansen, T Seidl, Albert Bifet, G Holmes, and Bernhard Pfahringer. An effective evaluation measure for clustering on evolving data streams. In *17th ACM SIGKDD*, 2011.

[55] Shuang Gao and Yalin Lei. A new approach for crude oil price prediction based on stream learning. *GSF*, 2017.

[56] Jesus L Lobo, Igor Ballesteros, Izaskun Oregi, Javier Del Ser, and Sancho Salcedo-Sanz. Stream learning in energy iot systems: a case study in combined cycle power plants. *Energies*, 2020.

[57] Indrė Žliobaitė, Mykola Pechenizkiy, and Joao Gama. An overview of concept drift applications. *Big data analysis: new algorithms for a new society*, 2016.

[58] Arslan Chaudhry, P K Dokania, T Ajanthan, and P H S Torr. Riemannian walk for incremental learning: Understanding forgetting and intransigence. In *ECCV*, 2018.

[59] Ameya Prabhu, P H S Torr, and P K Dokania. Gdumb: A simple approach that questions our progress in continual learning. In *ECCV*. Springer, 2020.

[60] Pietro Buzzega, M Boschini, A Porrello, and S Calderara. Rethinking experience replay: a bag of tricks for continual learning. In *ICPR*. IEEE, 2021.

[61] Tyler L Hayes, Kushal Kafle, Robik Shrestha, Manoj Acharya, and Christopher Kanan. Remind your neural network to prevent catastrophic forgetting. In *ECCV*. Springer, 2020.

[62] Yaqian Zhang, Bernhard Pfahringer, Eibe Frank, Albert Bifet, Nick Jin Sean Lim, and Yunzhe Jia. A simple but strong baseline for online continual learning: Repeated augmented rehearsal. *NeurIPS*, 2022.

[63] Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv:1708.01547*, 2017.

[64] Dahua Lin. Online learning of nonparametric mixture models via sequential variational approximation. *NeurIPS*, 2013.

[65] Ozgur Kara, N Churamani, and H Gunes. Towards fair affective robotics: Continual learning for mitigating bias in facial expression and action unit recognition. *arXiv:2103.09233*, 2021.

[66] Jacob Armstrong and D Clifton. Continual learning of longitudinal health records. *IEEE EMBS (ITAB)*, 2021.

[67] Shikhar Srivastava, M Yaqub, K Nandakumar, Zongyuan Ge, and D Mahapatra. Continual domain incremental learning for chest x-ray classification in low-resource clinical settings. In *DART, FAIR*. Springer, 2021.

[68] Robert Anderson, Yun Sing Koh, G Dobbie, and A Bifet. Recurring concept meta-learning for evolving data streams. *Expert Syst. Appl.*, 2019.

[69] Paulo RL Almeida, Luiz S Oliveira, Alceu S Britto Jr, and R Sabourin. Adapting dynamic classifier selection for concept drift. *Expert Syst. Appl.*, 2018.

[70] Ying Yang, X Wu, and X Zhu. Mining in anticipation for concept change: Proactive-reactive prediction in data streams. *Data Min. Knowl. Discov.*, 2006.

[71] Peipei Li, Xindong Wu, and Xuegang Hu. Mining recurring concept drifts with limited labeled streaming data. *ACM (TIST)*, 2012.

[72] Kylie Chen, Yun S Koh, and P Riddle. Proactive drift detection: Predicting concept drifts in data streams using probabilistic networks. In *IJCNN*, 2016.

[73] Alexandr Maslov, Mykola Pechenizkiy, Indreė Žliobaitė, and Tommi Kärkkäinen. Modelling recurrent events for improving online change detection. In *SIAM (SDM)*. SIAM, 2016.

[74] Charalampos Davalas, Dimitrios Michail, Christos Diou, Iraklis Varlamis, and Konstantinos Tserpes. Computationally efficient rehearsal for online continual learning. In *ICIAP*. Springer, 2022.

[75] Liyuan Wang, Kuo Yang, Chongxuan Li, Lanqing Hong, Zhenguo Li, and Jun Zhu. Ordisco: Effective and efficient usage of incremental unlabeled data for semi-supervised continual learning. In *CVPR*, 2021.

[76] Dushyant Rao, Francesco Visin, A Rusu, Razvan Pascanu, Yee Whye Teh, and R Hadsell. Continual unsupervised representation learning. *NeurIPS*, 2019.

[77] Łukasz Korycki and Bartosz Krawczyk. Streaming decision trees for lifelong learning. In *ECML PKDD*. Springer, 2021.

[78] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: massive online analysis. *J. Mach. Learn. Res.*, 2010.

[79] Jacob Montiel, Max Halford, S M Mastelini, G Bolmier, R Sourty, R Vaysse, A Zouitine, Heitor M Gomes, Jesse Read, T Abdessalem, et al. River: machine learning for streaming data in python. *JMLR*, 22(1):4945–4952, 2021.

[80] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[81] Elena Ikonomovska, Joao Gama, and Sašo Džeroski. Learning model trees from evolving data streams. *Data Min. Knowl. Discov.*, 2011.

[82] Henry Gouk, Bernhard Pfahringer, and Eibe Frank. Stochastic gradient trees. In *Asian Conference on Machine Learning*, pages 1094–1109. PMLR, 2019.

[83] Hayet Mouss, D Mouss, N Mouss, and L Sefouhi. Test of page-hinckley, an approach for fault detection in an agro-alimentary production system. In *2004 5th Asian control conference (IEEE Cat. No. 04EX904)*, volume 2, pages 815–818. IEEE, 2004.

[84] Philippe Pébay, Timothy B Terriberry, Hemanth Kolla, and Janine Bennett. Numerically stable, scalable formulas for parallel and online computation of higher-order multivariate central moments with arbitrary weights. *Computational Statistics*, 31(4):1305–1325, 2016.

[85] Erich Schubert and Michael Gertz. Numerically stable parallel computation of (co-) variance. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2018.

[86] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical Machine Learning Tools and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Elsevier Science & Technology, San Francisco, 2016.

[87] Nuwan Gunasekara, Heitor Murilo Gomes, Bernhard Pfahringer, and Albert Bifet. Online hyperparameter optimization for streaming neural networks. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9. IEEE, 2022.

[88] Martin Pavlovski, Fang Zhou, Ivan Stojkovic, Ljupco Kocarev, and Zoran Obradovic. Adaptive skip-train structured regression for temporal networks. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2017, Skopje, Macedonia, September 18–22, 2017, Proceedings, Part II 10*, pages 305–321. Springer, 2017.

[89] Jennifer Kwapisz, Gary Weiss, and Samuel Moore. Activity recognition using cell phone accelerometers. *SIGKDD Explorations*, 12:74–82, 11 2010.

[90] Jacob Montiel, Jesse Read, Albert Bifet, and Talel Abdessalem. Scikit-multiflow: A multi-output streaming framework. *Journal of Machine Learning Research*, 19(72):1–5, 2018.

[91] Vincenzo Lomonaco and Davide Maltoni. Core50: a new dataset and benchmark for continuous object recognition. In *Conference on Robot Learning*, pages 17–26. PMLR, 2017.

[92] Alex Krizhevsky. *Learning multiple layers of features from tiny images.* PhD thesis, University of Toronto, 2009.

[93] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[94] Vincenzo Lomonaco, Lorenzo Pellegrini, Andrea Cossu, Antonio Carta, Gabriele Graffieti, Tyler L. Hayes, Matthias De Lange, Marc Masana, Jary Pomponi, Gido van de Ven, Martin Mundt, Qi She, Keiland Cooper, Jeremy Forest, Eden Belouadah, Simone Calderara, German I. Parisi, Fabio Cuzzolin, Andreas Tolias, Simone Scardapane, Luca Antiga, Subutai Amhad, Adrian Popescu, Christopher Kanan, Joost van de Weijer, Tinne Tuytelaars, Davide Bacciu, and Davide Maltoni. Avalanche: an end-to-end library for continual learning. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2nd Continual Learning in Computer Vision Workshop, 2021.

[95] Łukasz Korycki and Bartosz Krawczyk. Streaming decision trees for lifelong learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 502–518. Springer, 2021.