FLAME: Adaptive and Reactive Concept Drift Mitigation for Federated Learning Deployments

Ioannis Mavromatis
Digital Catapult
London, UK
ioannis.mavromatis@digicatapult.org.uk

Stefano De Feo
Dept. of Electrical and Electronic Engineering, University of Bristol
Bristol, UK

Aftab Khan
Bristol Research & Innovation Laboratory, Toshiba Europe Ltd.
Bristol, UK
aftab.khan@toshiba-bril.com

Abstract

This paper presents Federated Learning with Adaptive Monitoring and Elimination (FLAME), a novel solution capable of detecting and mitigating concept drift in Federated Learning (FL) Internet of Things (IoT) environments. Concept drift poses significant challenges for FL models deployed in dynamic and real-world settings. FLAME leverages an FL architecture, considers a real-world FL pipeline, and proves capable of maintaining model performance and accuracy while addressing bandwidth and privacy constraints. Introducing various features and extensions on previous works, FLAME offers a robust solution to concept drift, significantly reducing computational load and communication overhead. Compared to well-known lightweight mitigation methods, FLAME demonstrates superior performance in maintaining high F1 scores and reducing resource utilisation in large-scale IoT deployments, making it a promising approach for real-world applications.

CCS Concepts

• **Computing methodologies** → **Distributed artificial intelligence**; *Planning and scheduling*; *Model development and analysis*; • **Security and privacy** → *Intrusion/anomaly detection and malware mitigation*; • **Networks** → Network performance modeling; • **Information systems** → Data analytics.

Keywords

Concept Drift, Federated Learning, IoT, Adaptive Thresholding, Resource-Constrained

1 Introduction

The Internet of Things (IoT) is becoming a standard solution for enhancing and maintaining services and applications in both industrial and non-industrial environments. IoT devices play a critical role in healthcare [14], manufacturing [27], product life cycles and warehouse inventory management [23], among others.

All IoT systems must meet real-time performance requirements while adhering to constraints in power consumption, physical size, installation complexity and more [8]. IoT devices face data privacy concerns [24], communication bandwidth limitations [11], processing limitations, etc. Machine Learning (ML) is very prominent in IoT, used in multiple scenarios to forecast future events and behaviours, optimise various tasks and solves many of the problems and challenges mentioned above [5]. Bandwidth limitations and privacy concerns are prominently addressed by distributed ML architectures, such as Federated Learning (FL) [24].

The dynamic nature of a real-world IoT environment can lead to drastic changes in the input data distributions [15]. From the ML model's perspective, the underlying relationship between the input data and expected output (target) changes over time, leading to an underfitted model. This behaviour is called concept drift [17]. It can occur for several reasons, e.g., long-term changes in the data (e.g., environmental changes), faulty hardware (sensor drift), or even adversarial actions, such as data poisoning attacks. These scenarios can be detrimental to ML predictions' quality when considering large-scale IoT systems with misbehaving "production" models.

This is the problem we consider in this paper. We present Federated Learning with Adaptive Monitoring and Elimination (FLAME) of concept drift. FLAME provides an automated and adaptable way of detecting and handling concept drift within an FL-based learning and inference pipeline. Within such a pipeline and post-deployment, it is important to identify when a model requires retraining without saturating the system resources and the data to be used for high accuracy of the ML predictions over time.

FLAME operates within an FL setup. It captures the entire lifecycle of an ML model, considering the stability of the initial training, its continuous performance and the mitigation strategies against drift, all these implemented in a real-world three-tier architecture, i.e., "cloud-edges-microcontrollers". Due to the nature of the embedded microcontrollers (i.e., minimal processing power), FLAME considers that edge devices installed close to the IoT sensors are used for the training, the cloud is used for the aggregation of a global model, and the model is later converted to its embedded form and deployed to the microcontrollers for inference [10].

As bandwidth and computing limitations are paramount for IoT deployments, we evaluate FLAME not only on the perceived accuracy of the models over time but also on the data exchange introduced and the need for resource-intensive retraining, ensuring it is always kept to the minimum. Moreover, we consider an unlabelled data scenario, representing a realistic case for an IoT system. We compare FLAME against various traditional statistical approaches, e.g., Adaptive Windowing (ADWIN) [6], Kolmogorov-Smirnov Windowing (KSWIN) [22], etc., frequently used from IoT deployments due to their lightweight nature.

The remaining of the paper is structured as follows. Sec. 2 presents similar works identified in the literature. A typical system architecture and how FLAME can operate within an FL pipeline is described in Sec. 3. The enhancements introduced within our solution, the model and dataset used and the algorithms compared against FLAME are outlined in Sec. 4. Finally, our results are presented in Sec. 5 and our final remarks are summarised in Sec. 6.

2 Related Work

Based on the data distribution changes, concept drift is classified as abrupt, incremental, gradual or recurring [15]. Many works present centralised mitigation strategies for resource-constrained environments. For example, a lightweight concept drift detector was presented in [25]. The authors calculate the centroids of each data class for old and new data, and finding the distance between the centroids can detect whether drift occurred. For unsupervised detection, such as in our case, the authors rely on a k-means clustering approach to label the data. This approach may work well for simplistic datasets but will underperform in more complex scenarios like the one introduced in this paper. In [1], authors detect drift using Principal Component Analysis and, after removing the outliers, develop a dynamic way of changing the depth of the ML model to adapt to the new data distribution. Even though this approach may lead to great performance over time and tackle problems such as forgetfulness, the model's variable size will make the model deployment in resource-constrained devices rather prohibitive.

Research around distributed IoT environments is very limited. In [17], four well-known drift detection methods were successfully implemented within a distributed architecture, proving that detection is not negatively impacted by distributing a system over a network. Concept drift detection and mitigation become harder in distributed environments due to the high communication and computing costs required for retraining. In distributed architectures where many models are trained separately, drift adaptation can be enabled by models from other systems where the drift was detected (leveraging the asynchronous appearance of concept drifts). For example, in [3], the classifiers on a given node consist of an ensemble of classifiers trained on other nodes after drift is detected. Similar approaches are seen in federated systems such as [7, 9], capitalising on models from other clients through model aggregation. However, both works become rather impractical for a real-world distributed scenario due to the increased volume of resources they consume. The solution in [7] uses continuous learning, constantly exchanging the model parameters and data, and [9], even though it reacts to concept drift better with reduced communications and memory usage, it stores all raw data from previous concepts significantly increasing the storage requirements on the edge devices.

3 System Architecture

We consider a practical application of a smart city scenario. The system comprises various devices such as surveillance cameras, traffic sensors, smart meters, public Wi-Fi hotspots, etc. As discussed, our system operates in a three-tier architecture, i.e., cloud-edge-endpoints, where each tier provides different computing capabilities. All IoT devices (i.e., namely, the endpoints) frequently receive firmware updates that solve bugs or add new functionality. Moreover, these IoT devices can continuously monitor their system logs, network traffic patterns and application behaviours for security breaches, malware, or abnormalities in their operation.

An FL deployment in such a system can utilise the more powerful edge devices to train and validate ML models. The cloud can aggregate a global model, and the trained models are deployed in the existing IoT devices for inference. Such a setup enables data privacy and security as the inference data remain on the local IoT devices, ensures the system's scalability and adaptability, allowing the system to update its detection models continuously, and also achieves reduced latency and bandwidth usage as the data do not need to be exchanged over low-datarate links.

3.1 FedOps for Large Scale IoT Deployments

The above FL deployment needs to consider the entire lifecycle of the ML pipeline, usually divided into 4 phases – scoping, data, modelling, and deployment. Machine Learning Operations (MLOps) refers to the practices, tools and techniques that can realise and manage the ML model lifecycle from the design and training to the evaluation, distribution and deployment. Federated Learning Operations (FedOps) extends the concept of MLOps within the FL space. FedOps considers the distributed nature of an FL deployment and orchestrates the data exchange between the clients, how and when models are aggregated, how models are initialised and monitored and how they can be extended with personalisation to fit heterogeneous deployments of diverse devices [18].

The example FedOps deployment in [18] selects FL clients based on their communication cost and overall model accuracy. This work does not consider model monitoring, making their implementation impractical for a real-world IoT system. We build upon that and introduce FLAME within a realistic FedOps pipeline (Fig. 1). During training, the model weights are initiated from the parameter server. The local datasets across all clients are processed and prepared for training. Each client splits the dataset into three train-test-validate splits (e.g., we use $80\% - 10\% - 10\%$, respectively). In Sec. 5.1, we describe our dataset preprocessing and the features extracted from that. A FedOps pipeline will later incorporate a hyperparameter tuning phase. For our experiments, during this phase, we fine-tuned the ML and statistical models hyperparameters, as described in Secs. 4.2 and 5.2. Following, during training, an ML model is usually trained until a stability point is reached. For our experimentation, we randomly initialised the set of hyperparameters and trained the model until the stability point, introducing a hard-coded epoch threshold to disregard underperforming attempts. Sec. 5 presents the results of a successful pipeline run. The trained model is later deployed for inference to the endpoint nodes. Each endpoint uses its locally collected data as an input for the inference phase.

During training and inference, the model's performance is continuously monitored for concept drift according to policies introduced by the parameter server. When a model is tagged as "no-longer-optimal," it can be retrained or retired and replaced by a new one. For both mitigation approaches, endpoint data should be sent to the client nodes, verified for validity, and used for training. Our solution enables the above functionality with two schedulers and a "concept"-aware dataset creation as described in Sec. 4.1. Finally, a production FedOps system will usually incorporate various event handlers and result/log collection mechanisms responsible for monitoring requirements and triggering various system events.

4 FLAME: Concept Drift Detection Pipeline

We implement our solution within the above pipeline, extending our previous work Federated LeArning with REactive monitoring of concept drift (FLARE) [10]. FLARE incorporated two scheduling subsystems, one for the FL clients and one for the endpoints. The client's scheduler is responsible for monitoring the model training

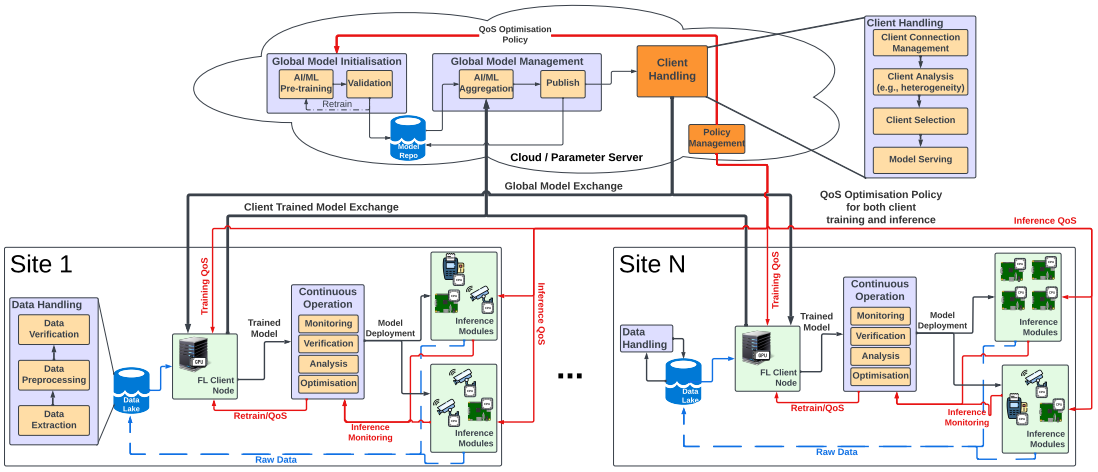


Figure 1: Overview of a typical FL lifecycle in FedOps for our scenario. It is distinguished in the "cloud" (handling the parameter server) and the "on-site" (handling the client and sensor nodes) deployments.

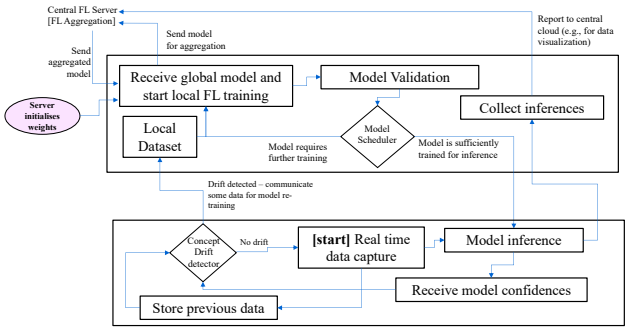


Figure 2: A system diagram showing the operation of FLAME.

and assessing the model's stability, i.e., when it is ready for inference. A stable model is converted into its embedded form and is sent to the endpoint nodes. The stability point is found by measuring the difference between the validation loss (using a validation dataset) and the training loss (using the training dataset) across different time windows. A model is considered stable when the $\sigma_w < \sigma_s \times (1 - \beta)$ holds. In this equation, σ_w is the standard deviation in the current window, and σ_s is the previous stable standard deviation value. β represents a stability coefficient, where a higher β increases the sensitivity to concept drift and the communication cost.

The endpoint scheduler runs a confidence validation test using the Kolmogorov-Smirnov (KS) test [13], calculating values between 0-1 and comparing the confidence from the client's validation set against the current dataset on the endpoint. When the similarity is low (KS-test result close to 1), this indicates a change in the sensor data distribution and the latest data are sent to the client for further training. On the other hand, a high similarity (KS-test result close to 0) indicates the deployed model is still effective on the current dataset. A static threshold ϕ was used to determine whether the similarity was high or low. More information can be found in [10].

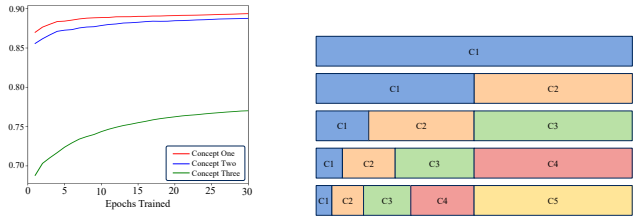
4.1 FLAME Enhancements

FLAME considered the intricacies of a real-world IoT system, such as the limited communication bandwidth, the resource-sparse nature of the devices, etc., and the long-term continuous operation within an IoT platform. Fig. 2 shows the interactions across the various system components. FLAME extends FLARE's functionality in three ways. First, we introduce a dynamic threshold ϕ_a to compare the KS test. The adaptive thresholding technique takes all KS statistics' mean μ_a and standard deviation σ_a within a variable-sized window w_a and sets $\phi_a = 3\sigma_a + \mu_a$. The window w_a is defined as the last n KS test values.

The variable-sized window is designed to ensure the threshold is based only on relevant previous KS statistics, allowing for the detection of different types of drift. To achieve this, only the KS statistics recorded since the last drift detection are considered, as earlier concepts should not influence the current threshold. Additionally, even within the same concept, older KS statistics can distort the threshold and should eventually be marked as stale and excluded. In the proposed method, the oldest 1/3 of the confidence values are discarded from the window before the KS test calculation. This windowing strategy is effective for long-term deployments as the increasing window size enhances the threshold's reliability.

Moreover, the static threshold β used for the model's stability is enhanced with an adaptive solution. The stability measurements across different concepts may vary significantly, with some concepts never reaching the threshold β, thus making the method very sensitive (Fig. 3a). Instead of β, we use the gradient of the stability curve for a given concept (as opposed to an absolute value), leading to a robust and adaptive method. When the gradient increases above a specific value, the model can be marked as stable. This allows us to work with imbalanced classes and datasets where one class may underperform compared to the rest.

As a final improvement, we introduce the retention of training data from previous concepts (Fig 3b). FLARE, when N new samples were received for training, the N oldest samples were removed



(a) Model stability. (b) Concept Examples.

Figure 3: a) Stability example for a model training on three different concepts in the MNIST-C dataset [19], b) an example of five concepts and the split of the newly created dataset.

from the training dataset before training, resulting in a dataset with a constant size. However, this can be catastrophic over time, with the model forgetting older concepts. Our new approach picks random samples from all concepts based on the following. Let $C \triangleq \{C_1, \ldots, C_n\}$ with $n \in \mathbb{N}^*$ denoting all concepts that have appeared in the system where C_n is the newest concept. Also, let $\mathcal{S}_{C_n}$ be all the samples of the concept C_n. Then, the total dataset D consists of:

$$D = D_n \cup D_{n-1} \cup \ldots \cup D_1, \tag{1}$$

$$D_n \subseteq \mathcal{S}_{C_n}, \quad \text{where } |D_n| = \left\lfloor \frac{1}{2} |\mathcal{S}_{C_n}| \right\rfloor, \tag{2}$$

$$D_x \subseteq \mathcal{S}_{C_x} \quad \text{where } |D_x| = \left\lfloor \frac{1}{2} \frac{x\,|\mathcal{S}_{C_x}|}{\sum_{i=1}^{n-1} i} \right\rfloor, \forall x \in [1, n-1] \tag{3}$$

resulting in 50% of the samples coming from the newest concept and smaller subsets from the older ones. Following such an approach, we can ensure that the model will perform best on the newest data without forgetting the older ones. Moreover, the training, validation, and test datasets extracted are always kept constant in size without draining the resources of the edge and sensor nodes.

4.2 Concept Drift Detection Algorithms

Many lightweight concept drift detectors are found in the literature. We chose three detectors to compare our method with, these being ADWIN [6], PHT [12], and KSWIN [22]. These detectors were chosen as they have been frequently used in the literature for similar concept drift detection activities [17] and operate comparably well on unlabelled data distributions [15].

4.2.1 ADaptive WINdowing (ADWIN) algorithm. ADWIN can detect distribution changes and drifts in data that vary with time. It uses an adaptive sliding window recalculated online according to the rate of change observed from the data. The window is discretised in two sub-windows without overlap. When a new sample is received, ADWIN examines all possible window splits, calculating the mean values for both windows and their absolute difference. The optimal lengths for the two sub-windows are found based on a threshold compared against all the calculated values.

Once a drift is detected, all the old data samples within the first window are discarded. ADWIN can effectively detect gradual drift since the sliding window can be extended to a large-sized window and identify long-term changes. Abrupt changes can again be identified with a small number of samples due to the big difference introduced in the mean values.

4.2.2 Page-Hinkley Test (PHT) algorithm. PHT is a variant of the CUmulative SUM (CUSUM) test. It has optimal properties in detecting changes in the mean value of a normal process. PHT's two-sided extension was considered in this work [15]. PHT recalculates the mean value and cumulative sum for every sample received based on a user-defined value.

PHT compares the mean values against a change detection threshold. The user-defined value's magnitude controls PHT's tolerance, while the change detection threshold tunes the false alarm rate. Larger thresholds entail fewer detections of false positives while increasing the number of false negatives. PHT easily identifies abrupt drifts due to the sudden change in the mean value. In contrast, incremental drift can be identified by sporadically sampling the time-series data stream.

4.2.3 Kolmogorov-Smirnov Windowing (KSWIN) algorithm. KSWIN is based on a KS test that accepts one-dimensional data and operates without assuming the underlying data distribution. KSWIN maintains a fixed-size sliding window discretised in two sub-windows without overlap. A two-sampled KS test is performed on both sub-windows. It compares the absolute distance between two empirical cumulative data distributions.

The test result is compared against the square root of the log of a sensitivity parameter over the length of the window. Data with increased periodicity and a large window make KSWIN too sensitive and return many false positives. Relatively small window sizes and an optimised sensitivity parameter significantly improve the performance. As described in [22], KSWIN can detect gradual and abrupt drifts but falsely classifies many samples as false positives. However, considering the criticality of an error, false positives are not as critical and can be removed in post-processing.

5 Results

Our experimental setup consists of a "cloud" node, which is used as the model aggregator, 8 "edge" nodes that act as the FL clients, and 32 "endpoint" nodes, the embedded microcontrollers used for inference and "raw data collection". The training dataset (Sec. 5.1) was split equally across the different clients, randomly allocating $\sim$ 42k applications to each. Each endpoint was assigned $\sim$ 70k for inference, splitting the available applications equally. Our experiment runs chronologically (monthly), so a sensor infers or a client trains/validates, respectively, on the sub-dataset of the given month. Finally, out of the 92 months of the entire dataset's length, the first 12 months (year 2014) are used for the initial model training and the remaining 80 months (Jan. 2015 - Sep. 2021) for inference.

For the fine-tuning of the different detectors, we initially ran a grid search to estimate the value for all hyperparameters. The detectors' sensitivity was later fixed for the rest of the experiment. We chose the values that did not return candidate drift within the "training" and "validation" sets used for the initial 12 months of training. Finally, for our ML model, we used the FedAvg aggregation method, cross-entropy loss, ADAM optimisation, a learning rate of

0.003 and a batch size of 4. Finally, some results will be described only in text due to the limited space.

5.1 Malware Detection Dataset

Our experimentation is based on the Androzoo dataset [2]. Androzoo contains millions of malware scans for different Android apps since 2010. The dataset was chosen due to the real and virtual concept drifts presented – the types of malware change over time (virtual drift) and the efficacy of virus scanners (real drift).

Each app is scanned by over 70 antivirus scanners using VirusTotal[1], and Androzoo reports the number of times it is classified as malware. This score is used as the ground truth label for each app. As discussed in [26], no definitive threshold classifies an app as malware; therefore, for our experimentation, a threshold of two detections was chosen to avoid single-case outliers. This threshold has also been used in previous works [4]. We used around 2.6M applications for our experiment, with the percentage of malware at around 10%. This is considered realistic in terms of the proportion of real-world apps that contain malware, according to [20].

The Androzoo dataset consists of the full APK file for every app, so all APK files were pre-processed to extract the features required. Two main types of raw features can be extracted from the apps, i.e., raw opcode sequences or bytecode [21]. Opcode sequences were chosen due to their successful prior use in [16]. They were extracted using Apktool[2], similarly with [16]. As this work presents, a high F1 score can be achieved by classifying these opcode sequences, so this was considered a reasonable method for classification.

5.2 ML Model

We chose a simple Convolutional Neural Network (CNN) architecture that can be easily converted and deployed on embedded devices while retaining high accuracy. The model consists of an embedding layer, one convolutional layer followed by a max pooling layer, one hidden connected layer (fully connected) and a softmax classification layer (fully connected). ReLU activation functions were used on the convolutional layer, and the fully connected layer and a sigmoid function were applied to the softmax layer to produce the positive class probability.

The opcodes are projected in the embedding space, and each vector is multiplied by a weight matrix that is initially randomised and updated by back-propagation during training. By doing so, the semantic information of each opcode can be encoded in the embedded space, and the network can discover certain opcodes with similar meanings, allowing them to be treated comparably by deeper layers in the network. The kernel size chosen for the convolutional layer was 64, which means it can form features for sequences of up to 64 opcodes; this is a reasonable choice given that the opcode sequence length was 357. Moreover, before applying the convolutional filters, we zero-pad the start and end of the input to ensure that the length of the output matrix from the convolutional layer is the same as the length of its input. Finally, the max pooling layer takes a maximum over the sequence length dimension, which allows sequences of different lengths to be represented by feature vectors of the same length. For more details, refer to [16].

[1]VirusTotal: https://www.virustotal.com/
[2]Apktool: https://ibotpeaches.github.io/Apktool/

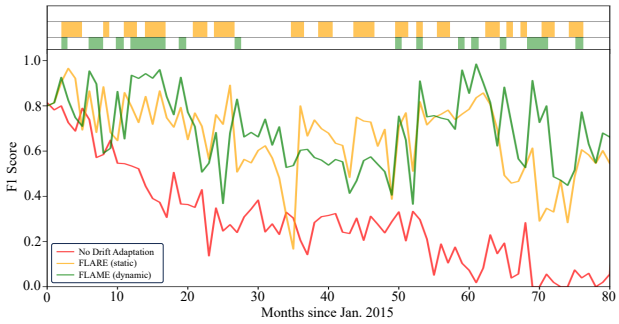


Figure 4: Small-factor experiment comparing static and dynamic thresholds.

The class imbalance (roughly 90% benign - 10% malware), was tackled with a weighted loss function to ensure no bias towards the benign (majority) class. Binary cross entropy was used with an increased weight to the malware class, with the weights calculated as the inverse of the square root of the number of samples per class.

5.3 Preliminary FL Experiment

We initially conducted a small-scale experiment with a single FL client and a single endpoint, using both FLARE (static thresholds) and FLAME (dynamic thresholds) methods. Around 12k applications were selected initially for training, and 80k applications were used for inference and retraining. Our results can be found in Fig. 4. The F1 score significantly decreases after around ten months when drift is present. In contrast, it stays relatively high when introducing a drift detection and mitigation strategy (similar to the initial model). Comparing FLARE and FLAME, we see that they achieve roughly equal performance over time and similar trends regarding how the F1 score increases or decreases.

Running the experiment across many small-factor datasets, we observe that the F1 score significantly deviates over time. This is due to the different subsets of the applications for every experiment (thus, the big dip in the performance between months 35-50 in Fig. 4). Comparing the retraining required and the data exchange, the top of Fig. 4 shows the timeframes (in months) that retraining was executed. A month is considered a "retraining month" when retraining was triggered from the previous month's drift detection. As seen, with the dynamic thresholding and the retention of the data introduced, this time is significantly reduced, thus reducing the computational load in the system without affecting the F1 score. This is also reflected in the total volume of data exchanged as well, where FLAME exchanged around 13 GB with FLARE exchanging almost 22 GB.

5.4 Large Scale FL Experiment

As it was shown that FLAME performs better compared to FLARE, our large-scale experiment compares FLAME with a high-frequency update method (every month, the model is retrained and redeployed regardless of the performance drop), a low-frequency update method (the retraining and redeployment happen every three months) and the three concept drift detectors introduced in Sec. 4.2. The random seed was fixed to ensure we allocate the same applications to each client and sensor across the entire experiment

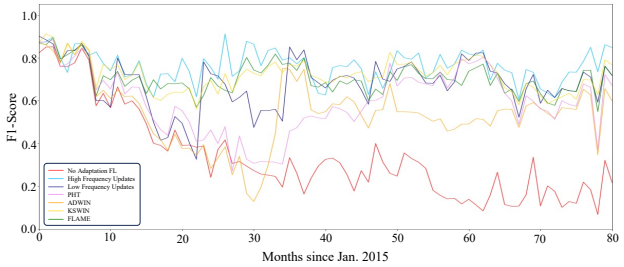


Figure 5: Comparison of F1 scores for all methods.

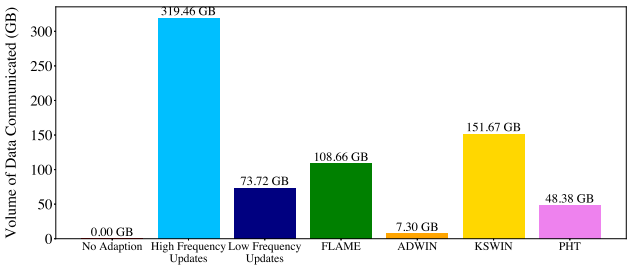


Figure 6: Volume of data exchanged across all methods.

length and different runs. Fig. 5 summarises the F1 scores perceived from all methods. The no-adaptation method is our baseline, as no retraining is considered throughout the experiment.

As in the small-scale experiment, we see a significant drop of about 20% in the model's performance after a few months (starting from month 6) (as seen in the baseline measurement). Moreover, we see that FLAME achieved similar performance with the high-frequency method and KSWIN while outperforming PHT, ADWIN and the low-frequency method. Each method has its own sensitivity, reflected in the number of times throughout the 80 months that drift is detected on any of the sensors. ADWIN reported few detections; thus, the retraining was very sparse. We believe this is due to the absolute values of the confidence intervals fed to the detectors (always between 0-1), making ADWIN significantly underperform in such a scenario. PHT and KSWIN perform better in such a distribution, and this is reflected in the number of detections they reported and, of course, in the perceived model quality as well.

Regarding resource utilisation, FLAME, compared to KSWIN, showed similar performance and triggered a retraining cycle around 15% fewer times, achieving the same result with fewer computing resources. A similar result is also shown in the data exchanged (Fig. 6). FLAME reduced the communication overhead by 33%, compared to KSWIN and by almost 66% compared to the high-frequency adaptation method. Overall, FLAME has managed to maintain the same performance using less computing and network resources, making it a promising solution for resource-sparse IoT environments.

6 Conclusions

This paper presented FLAME, which effectively addresses the challenges of concept drift in large-scale IoT environments. Operating in an FL fashion, it tightly integrates within a FedOps pipeline, leveraging monitoring capabilities, intelligent data handling and various adaptations for smarter real-time operation. The framework's dynamic thresholding and data retention capabilities ensure sustained model accuracy and performance over time, even in the face of evolving data distributions. Comparative experiments highlight FLAME's ability to outperform traditional drift detection methods while minimising computational and communication resources. These findings underscore FLAME's potential as a scalable and efficient solution for maintaining the reliability of ML models in resource-constrained IoT systems. Future work will explore further optimisations and extensions of FLAME to enhance its applicability across diverse IoT applications and environments.

References

[1] Omar Abdel Wahab. 2022. Intrusion Detection in the IoT Under Data and Concept Drifts: Online Deep Learning Approach. *IEEE Int. Things J.* 9, 20 (2022), 19706–19716.

[2] Kevin Allix et al. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proc. of IEEE/ACM MSR*. 468–471.

[3] Hock Hee Ang et al. 2013. Predictive Handling of Asynchronous Concept Drifts in Distributed Environments. *IEEE Trans. Knowl. Data Eng.* 25, 10 (2013), 2343–2355.

[4] Daniel Arp et al. 2014. Drebin: Effective and Explainable Detection of Android Malware in your Pocket.. In *Proc. of Ndss*, Vol. 14. 23–26.

[5] Jiang Bian et al. 2022. Machine Learning in Real-Time Internet of Things (IoT) Systems: A Survey. *IEEE Internet Things J.* 9, 11 (2022), 8364–8386.

[6] Albert Bifet and Ricard Gavaldà. 2007. Learning from Time-Changing Data with Adaptive Windowing. In *Proc. of Int. Conf. SDM 2007*.

[7] Giuseppe Canonaco, Alex Bergamasco, Alessio Mongelluzzo, and Manuel Roveri. 2021. Adaptive Federated Learning in Presence of Concept Drift. In *Proc. of IEEE IJCNN*.

[8] Maurizio Capra et al. 2019. Edge Computing: A Survey on the Hardware Requirements in the Internet of Things World. *Future Internet* 11, 4 (2019), 100.

[9] Fernando E Casado et al. 2021. Concept Drift Detection and Adaptation for Federated and Continual Learning. *Multimedia Tools and Applications* 81, 3 (2021), 3397–3419.

[10] T. Chow et al. 2023. FLARE: Detection and Mitigation of Concept Drift for Federated Learning based IoT Deployments. In *Proc. of IWCMC*.

[11] Alexander Herzog et al. 2024. Selective Updates and Adaptive Masking for Communication-Efficient Federated Learning. *IEEE Trans. Green Commun. Netw.* (2024), 1–1.

[12] D. V. Hinkley. 1971. Inference about the Change-point from Cumulative Sum Tests. *Biometrika* 58, 3 (1971), 509–523.

[13] Frank J. Massey Jr. 1951. The Kolmogorov-Smirnov Test for Goodness of Fit. *J. Amer. Statist. Assoc.* 46, 253 (1951), 68–78.

[14] Ravi Kishore Kodali, Govinda Swamy, and Boppana Lakshmi. 2015. An Implementation of IoT for Healthcare. In *Proc. of IEEE RAICS*. 411–416.

[15] I. Mavromatis et al. 2023. LE3D: A Lightweight Ensemble Framework of Data Drift Detectors for Resource-Constrained Devices. In *Proc. of IEEE CCNC*.

[16] Niall McLaughlin et al. 2017. Deep Android Malware Detection. In *Proc. of ACM CODASPY*. 301–308.

[17] Hassan Mehmood et al. 2021. Concept Drift Adaptation Techniques in Distributed Environment for Real-World Data Streams. *Smart Cities* 4, 1 (2021), 349–371.

[18] JiHwan Moon, SeMo Yang, and KangYoon Lee. 2024. FedOps: A Platform of Federated Learning Operations with Heterogeneity Management. *IEEE Access* (2024).

[19] Norman Mu and Justin Gilmer. 2019. MNIST-C: A Robustness Benchmark for Computer Vision. *ArXiv* abs/1906.02337 (2019).

[20] Feargus Pendlebury et al. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proc. of USENIX Sec. Symp.* 729–746.

[21] Junyang Qiu et al. 2020. A Survey of Android Malware Detection with Deep Neural Models. *ACM Comput. Surv.* 53, 6, Article 126 (2020), 36 pages.

[22] Christoph Raab, Moritz Heusinger, and Frank-Michael Schleif. 2020. Reactive Soft Prototype Computing for Concept Drift Streams. *Neurocomputing* 416 (2020).

[23] B Sai Subrahmanya Tejesh and SJAEJ Neeraja. 2018. Warehouse Inventory Management System Using IoT and Open Source Framework. *Alex. Eng. J* 57, 4 (2018), 3817–3823.

[24] Runhua Xu, Nathalie Baracaldo, and James Joshi. 2021. Privacy-Preserving Machine Learning: Methods, Challenges and Directions. *CoRR* abs/2108.04417 (2021).

[25] Takeya Yamada and Hiroki Matsutani. 2023. A Lightweight Concept Drift Detection Method for On-Device Learning on Resource-Limited Edge Devices. In *Proc. of IEEE IPDPSW*. 761–768.

[26] Namrud Zakeya et al. 2021. Probing AndroVul dataset for studies on Android malware classification. *J. King Saud Univ., Comp. & Info. Sci.* (2021).

[27] Ray Y Zhong and Wenbo Ge. 2018. Internet of Things Enabled Manufacturing: A Review. *IJASM* 11, 2 (2018), 126–154.