Communication Papers of the 18th Conference on Computer Science and Intelligence Systems pp. 131–139
DOI: 10.15439/2023F7674
ISSN 2300-5963 ACSIS, Vol. 37

FAS-CT: FPGA-Based Acceleration System with Continuous Training

1st Manuel L. González
ITCL
Burgos, SPAIN
manuel.gonzalez@itcl.es

2st Jorge Ruiz
ITCL
Burgos, SPAIN
jorge.ruiz@itcl.es

3st Randy Lozada
ITCL
Burgos, SPAIN
randy.lozada@itcl.es

4st E.S. Skibinsky-Gitlin
ITCL
Burgos, SPAIN
erik.skibinsky@itcl.es

5th Ángel M. García-Vico
Dept. of Communication and Control Systems
UNED, Spain
amgarcia@scc.uned.es

6st Javier Sedano
ITCL
Burgos, SPAIN
javier.sedano@itcl.es

7st José R. Villar
University of Oviedo
Oviedo, SPAIN
villarjose@uniovi.es

***Abstract*—This paper presents FAS-CT a novel approach to a distributed low-latency Deep Learning inference system based on a Field Programmable Gate Array (FPGA). The system incorporates continuous training capabilities based on Concept Drift Detection, where each model prediction is compared with the ground truth to detect a change in the data patterns that the model requires to adapt to. FAS-CT is formed by two main execution pipelines. Firstly, the prediction pipeline is powered with Xilinx® Zynq® UltraScale+™ MPSoC FPGA and where low latency is the target. Secondly the Retraining pipeline aims adapting the model the model when Concept Drift is detected. A complete characterization of FAS-CT is provided in this article using a neural network model and an experimental setup. The latency of the Prediction pipeline achieved was 5.79 ms. The total degradation of the model when continuous training is activated is 57% in contrast to when is deactivated which is 1609%. These results demonstrate that FAS-CT is suited for real-time Deep Learning inference and can be automatically adapted to evolving data environments.**

I. Introduction

IN RECENT times, Deep Learning (DL) has rapidly emerged as a powerful tool, demonstrating unparalleled potential in domains such as computer vision, natural language processing, and predictive analytics, surpassing traditional machine learning techniques [1]. Conventionally, cloud computing has been the favored approach for deploying DL models for such applications, harnessing the vast processing power and storage capabilities of data centers [2]. However, the growth in data traffic coupled with the stringent low-latency requirements of various DL services has begun to challenge this centralized computing approach [3].

The Edge Computing (EC) paradigm has gained prominence, offering a solution to these challenges by processing data closer to its source. EC is a decentralized paradigm that places computational resources, memory, and services closer to the data origination point, thereby accelerating response times and reducing the burden on communication bandwidth. Despite its potential, EC poses its own challenges, particularly in terms of limited computational power and memory resources when compared to cloud-based systems [4]. These EC limitations have a direct impact on DL solutions. Real-time inference or model adaptation to new data could be compromised.

Executing DL models on EC devices in order to achieve real-time inference is a non-trivial task due to the inherent resource constraints. Despite optimization for edge deployment [5], these models continue to demand substantial computational resources. Model adaptation represents another significant challenge within the EC paradigm. DL models necessitate continuous updates to maintain relevance in rapidly evolving data environments or in the presence of Concept Drift (CD) [6]. These widely recognized problems of DL models on EC have been investigated in literature [5] and are typically addressed through an orchestration system architecture or continuous training techniques.

Orchestration architecture, a key theme across several articles, is primarily used to manage and optimize distributed resources. The research in [7] uses it for distributing processing between EC devices and the cloud in a real-time image recognition system. In [8], the authors use an orchestration architecture for continuous training by integrating new data into existing models, whereas in [9], it enables continuous updates to the seizure prediction model. However, it is noteworthy to mention that these approaches typically focus either on orchestration or on continuous training, but rarely integrate both elements effectively. This highlights the novelty of the current work, which aims to bridge this gap by developing a mechanism capable of detecting CD and performing continuous training while managing and optimizing resources through an orchestration system architecture. These studies emphasize the importance of orchestration architecture in managing complex distributed systems and enabling continuous training. This topic is particularly interesting, as highlighted in [8], [9], and [10]. [8] and [9] demonstrate its importance in medical applications, where models are continually updated with new data to improve accuracy. [10] extends this idea to IoT devices, introducing a loss compensation mechanism to improve Federated Incremental Learning, highlighting the

Thematic track: Distributed Edge AI – Risks and Challenges

applicability of continuous training across various fields.

On DL there are different types of models each one with a target application [1]. Those DL architectures have been implemented successfully in different EC devices. Implementations of Fully Connected Networks (FCN) have been explored in [11], [12]. Convolutional Neural Networks (CNN) on [7], [13], [8] and Recurrent Neural Networks (RNN) particularly Gated Recurrent Units (GRU) and Long Short Term Memory (LSTM), are used in [14], [15], [16]. Typically used EC devices for DL implementation have coupled processing technologies like CPU + GPU [17], [18], CPU + FPGA [13], [19] or CPU + TPU [18], [20]. CPU's mean purpose is to manage data and connections with orchestration architecture, meanwhile, GPU, FPGA, or TPU are used to accelerate DL model inference.

The use of FPGA is explicitly discussed in [11], which presents ZyNet to automate FCN implementation on low-cost FPGA platforms. This tool facilitates the deployment of FCN in edge computing devices and is a promising approach for making FPGA-based DL computing more accessible. In [13] an FPGA with LeNet-5 is studied. This article implements an autonomous architecture with continuous training of the model in a Xilinx® Multi-Processing System on Chip (MPSoC) device. The training is performed in MPSoC CPU and the inference is executed on FPGA. This solution leads to inference times of 2.2 ms and training times of 286s. Also in [12], [16] a Xilinx® FPGA is used to implement inferences of FCN and LSTM, their performance is analyzed. These implementations have maximum inference times of 1.09ms for FCN and 2.6ms for LSTM.

In the present work we introduce the FPGA-Based Acceleration System with Continuous Training (FAS-CT). This novel EC architecture is designed for the orchestration of DL model inference on FPGA, explicit CD detection, and continuous training. As is exposed, DL model inference on FPGA can yield low-latency responses. The CD is explicitly identified to monitor any degradation in the model's performance. If CD is detected, the retraining stage of the DL model is launched. This process allows continuous training of the model and its automatic update in FPGA. The main contributions of this article are listed as follows:

1) We put forth an architecture that efficiently coordinates various technologies best suited for different tasks. FPGA for DL model inference, GPU for DL model training, and CPU for preprocessing, postprocessing, CD detection, and data communication.
2) A complete description and characterization of FAS-CT is presented. The description of each component and the interaction between them is detailed. The characterization is examined with real-world data concerning response time, model performance, and model updates.
3) CD detection is included in the architecture to perform model retraining only when needed. This stage in the orchestration scheme allows for saving energy because a power-hungry GPU is used only when the model performance is worsening.

The rest of the article is organized as follows. Section II details a complete description of FAS-CT architecture, focusing on different technologies for each component. Section III explains CD detection and its implementation in a module on FAS-CT. Section IV explains the implementation of DL model inference on FPGA and its communication with FAS-CT. Section V focuses on the setup and the experiments performed on FAS-CT to get a complete characterization of the architecture. Section VI exposes the results of metrics defined in the previous section. The article finishs in section VII with conclusions and future research work.

II. Continuous Training System Architecture

FAS-CT architecture is designed around a central orchestration framework that maximizes the benefits of each technology it incorporates. It leverages FPGA for real-time neural network inference, GPU for model training, CPUs for pre and post-processing of data, CD detection, and management of data communication. This collaborative design facilitates high performance and ensures seamless integration of these key processes. FAS-CT has been designed to enable easy adaptation to diverse hardware, software, and data configurations. Each step can be deployed on separate hardware, thus satisfying most latency, performance, or throughput requirements with ease. For instance, feature preprocessing, inference of the neural network, and result postprocessing can be performed near the data source or prediction consumer, while feature storage and model retraining can be executed on more powerful devices like computer servers with GPU.

FAS-CT is composed of different stages or modules. Each of the stages shown in Fig.1 is handled by a different service. Modules are grouped in two pipelines. The first pipeline is responsible for inference in FPGA. The second pipeline is responsible for retraining the model when CD is detected.

To facilitate the service deployment, management, and monitoring, Docker [21] has been used for each module. A description of the task and purpose for each stage is given below.

A. Data Propagation

To communicate the different stages in FAS-CT the Message Queuing Telemetry Transport (MQTT) [22] protocol has been used. MQTT is a lightweight messaging protocol based on the publish-subscribe pattern. The protocol operates on top of the TCP/IP network stack and has support for multiple Quality of Service (QoS) levels to ensure reliable message delivery.

To manage the message queues the open-source Eclipse Mosquitto [23] MQTT Broker has been used. Mosquitto is licensed under EPL2, and it is one of the most suitable MQTT brokers due to its high performance [24], being multi-platform MQTT 5 compliant and having Transport Layer Security (TLS) support. The data has been serialized using Google Protocol Buffers [25].

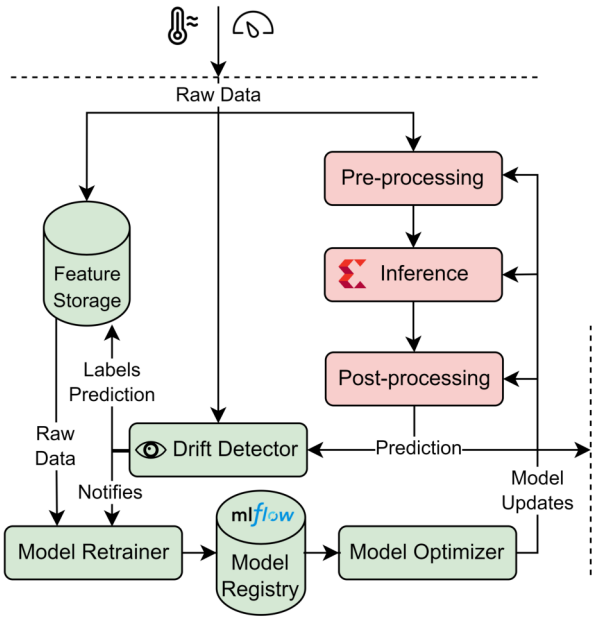

Fig. 1. FAS-CT architecture diagram. There are eight modules grouped in two pipelines. Modules of the Prediction pipeline are highlighted in red and the Retraining pipeline in green. The inference module is executed in FPGA, he Model Retrainer module in GPU and the rest of modules in CPU.

B. Feature Store

A database records each set of model input features with a unique sequential ID, along with the corresponding ground truth, model inference result, and drift level of the model for that prediction. The database can be queried using a set of remote procedure calls (RPC) [25] to retrieve the various features required for the model retraining process.

C. Feature Inference Preprocessing

The feature inference preprocessing stage involves a series of transformations that are applied to the raw data to make it compatible with the machine learning models. This stage may include data cleaning, where inconsistencies, errors, and outliers in the data are identified and corrected or removed. Another common step is data normalization or standardization, which is crucial for algorithms that are sensitive to the scale of the features. This process adjusts the values of numeric features so that they share a common scale, without distorting the differences in the ranges of values or losing information. Feature engineering is another integral part of preprocessing, creating new features based on existing ones, which can enhance the predictive power of the machine learning model.

D. FPGA Edge Inference

An FPGA receives a set of tensors from the preprocessing stage which serves as inputs for the neural network. These tensors are essentially multi-dimensional arrays of data, prepared and structured to be ingested by the model for making predictions.

Simultaneously, the device also listens for model updates from the continuous training pipeline. Upon receiving these model updates, the FPGA substitutes the current model with the new model. Essentially the model is evolving its ability to make accurate predictions in line with the most recent trends in the data. This process of continuous listening and updating ensures that the model deployed on the FPGA is always synchronized with the most recent version and maintains accuracy even in the face of changing data landscapes.

E. Result Postprocessing

Certain models may need a postprocessing step to enable an accurate comparison between the ground truth and the prediction. For instance, if the preprocessing stage involved scaling or standardization of features, an inverse transformation might be necessary for the postprocessing stage to convert predictions back to the original scale. In this way, the predictions can be compared with the ground truth. Another common postprocessing step involves the treatment of probability outputs. Many machine learning models, especially in classification tasks, output probabilities of each class. A thresholding operation might be necessary to convert these probabilities into discrete class labels. The choice of threshold can significantly affect the model's performance metrics and can be fine-tuned based on the requirements of the specific task.

F. Drift Detector

This module is continuously monitoring the error that the neural network is generating. If the error is between some limits or thresholds, the model is considered to be providing a correct prediction. If the error increases, CD is detected and the Retraining pipeline is executed.

There are several Drift Detectors that can be placed at this stage. For FAS-CT we choose Drift Detector Method. This is an algorithm developed by J. Gama et al. [26]. It is computationally lightweight and has low memory requirements, in line with the two main constraints in EC. A description of this algorithm is detailed in Section III.

G. Model Retraining, Validation and Registry

Upon a Drift Detector notification, the Model Retraining stage updates the scalers and the neural network to fit the newest data. The data has been stored properly in the Feature Storage module and is served to perform new training on the model.

The validation process of the updated model entails a comparison between the Drift Detector error metric of both old and new models. If the error metric of the new model is less than the current drifted mean, the updated model is stored in the model registry and publishes a model update on FAS-CT. However, if the error metric of the new model is assessed as an improvement, the model is stored.

The experiments, models, and scalers are tracked by MLFlow [27]. MLFlow is an open-source platform for machine learning workflows that includes features such as experiment and model registry, allowing for efficient management

of models. Furthermore, MLFlow's experiment tracker stores and organizes all the models, data, and metrics of a retraining process.

H. Model Optimization and Update

Upon completion of the model's retraining and validation process, the subsequent step involves its conversion and optimization into a specific format that can be consumed by edge devices like FPGA. The new model is serialized in JSON format and it is sent to the FPGA as detailed above.

The model update could also involve changes in Preprocessing and Post-processing stages. In this case, some functions like scalers must be updated. Finally, if the model is retrained, the Drift Detector is also reset with new parameters. This process is explained in the following section.

III. Concept Drift Detection

CD refers to the phenomenon where the statistical properties of data, on which the model has been trained, change over time in unforeseen ways, causing the model's performance to degrade. This happens because most predictive models are designed and trained under the assumption that future patterns will remain consistent with historical ones, which is often not the case in real-world scenarios. Real-world data is continually evolving and changing, and so too is the context in which DL models operate. Changes can occur in various forms e.g. gradual, abrupt, incremental, or recurring changes [6]. Different types of CD exist, such as real, virtual, and dual CD [6], [28]. In this work, only real CD will be considered and will be referred as CD for the sake of simplicity.

DL models, though powerful and highly accurate, have a significant weakness when it comes to CD. Detecting CD and subsequently retraining the models can be a solution to mitigate this impact. The process typically involves monitoring the model's performance metrics over time, and if a significant decline is detected, it's an indication that CD might be occurring. Once identified, the model can be retrained with the latest data, which reflects the new patterns. By applying this continuous training approach, DL models can become more adaptable to the evolving nature of real-world data.

The CD detection method implemented in the Drift Detector module, shown in Fig. 1, is similar to the Drift Detection Method (DDM) proposed by [26]. This method is based on the error signal produced by a binary classifier. The error signal is the probability of misclassifying an instance plus the standard deviation. The error signal fits a Bernoulli distribution because a binary classifier is assumed in [26]. DDM can be extended to forecasting or regression models by monitoring the error in prediction. On these models, DDM studies error signal mean and standard deviation based on a Gaussian distribution:

$$e_n = y_n^{true} - y_n^{pred} \tag{1}$$

$$\mu_n = \frac{n-1}{n} \cdot \mu_{n-1} + \frac{1}{n} e_n \tag{2}$$

$$\sigma_n = \sqrt{\frac{n-1}{n}\sigma_{n-1}^2 + \frac{1}{n-1}\left(e_n - \mu_n\right)^2} \tag{3}$$

$$ddm\_e_n = \mu_n + \sigma_n \tag{4}$$

Where $\mu_0(e) = \sigma_0(e) = 0$ and n is the number of monitored predictions. Recurrent formulas for μ_n and σ_n are used to avoid storing previous values of the error and satisfy the memory requirements of the processing system. These formulas are derived in detail in Appendix A. The value ddm_e_n is known as the DDM error metric and is used to determine when CD is detected. Notice that for DDM it is necessary to have the y^{true} value. Particularly, this is possible for a forecasting task on time-series data because y^{true} will be available at a certain time. Two configuration parameters are needed μ_{min} and σ_{min}. These parameters are the minimum mean and the minimum standard deviation calculated during the training process. After that, DDM is configured and starts monitoring the model. The warning level is triggered if:

$$ddm\_e_n >= \mu_{min} + 2 \cdot \sigma_{min} \tag{5}$$

At the warning level, the performance of the DL model is starting to worsen and the CD may arise. To adapt the model input and target data are stored to retrain the model. The drift level is triggered if:

$$ddm\_e_n >= \mu_{min} + 3 \cdot \sigma_{min} \tag{6}$$

At the drift level, CD is detected, retrain is performed with stored data, DL model adaptation is executed and DDM parameters are restored. In FAS-CT the new model is changed on FPGA and the inference is executed with the new adapted model. This is an endless loop of inferring, monitoring, retraining, and adapting the DL model that could generate an updated response in an evolving environment.

IV. FPGA Inference

The edge inference module in the prediction pipeline shown in Fig. 1 is implemented using an FPGA device. These devices are highly versatile integrated circuits that can be reconfigured and programmed to perform specific tasks, making them ideal for application acceleration, including neural network inference [11]. FPGA devices offer several advantages for such tasks. First, their parallel processing architecture allows multiple operations to be performed efficiently and simultaneously, resulting in high throughput and low latency [13], [16]. This is particularly beneficial for neural network inference, which involves intensive matrix calculations. In addition, FPGAs offer the flexibility to customize hardware designs, enabling the implementation of highly optimized neural network architectures tailored to specific application requirements. The ability to fine-tune hardware resources at the circuit level enables efficient utilization of FPGA resources, resulting in improved power efficiency [12]. In addition, FPGAs can be integrated with existing systems, including CPUs and GPUs, to leverage their respective strengths in a heterogeneous computing environment. Overall, the programmability, parallelism, customization, and integration capabilities of FPGAs make them a compelling choice for accelerating neural network

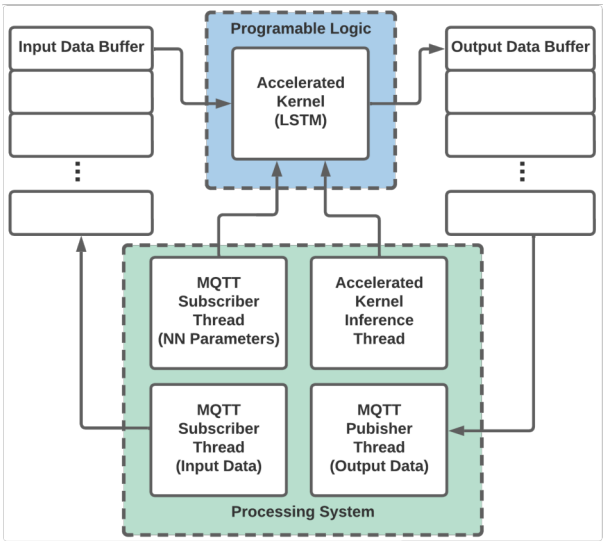

Fig. 2. FPGA inference diagram

inference, offering significant performance gains and energy efficiency for a wide range of applications.

The inference stage must be able to process various data sent by the FAST-CT using the MQTT protocol. The transmitted data are in JSON format. The FPGA processing system must be in charge of extracting the necessary information to carry out the inference on the input data and the configuration of the corresponding neural network parameters. Likewise, the FPGA device transmits the result of the inference of each set of data received. The neural network is implemented on an acceleration kernel in the programmable logic side of the FPGA. This kernel communicates with the processing system and accelerates the inference task.

For the management of all these tasks, the architecture shown in Fig. 2 has been implemented. Four processing threads are used for data processing, control of the inference process, and configuration of the acceleration kernel. Three threads manage the data received and sent by MQTT. The other one is in charge of the inference execution. As can be seen in Fig. 2, two buffers are used for the synchronization between the tasks of reading input data, inference, and writing output data.

The inference process in the prediction pipeline on FAS-CT works as follows. The MQTT subscriber thread for input data writes the inference data to the input buffer. The accelerated kernel inference thread orders the execution of the acceleration kernel and stores the result in the output data buffer. Finally, the MQTT publisher thread for output data sends the results of the inference to continue the prediction pipeline. The FPGA device is also integrated into the retraining pipeline. In the fourth thread, the MQTT subscriber for neural network parameters is listening for updates in neural network weights. These updates are codified in JSON format. This thread manages the configuration of the acceleration kernel parameters and sets it for the next execution.

V. Materials and Methods

A. Setup

In this study, we employed FAS-CT with an LSTM neural network for time series forecasting. Due to client confidentiality, the data and results of the experiment have been anonymized. The purpose of the network is to forecast the value of a single sensor with a prediction horizon of 10 minutes. The model has been trained with a 32-minute sliding window of 4 different correlated sensors with a sample rate of 1 minute. All data is stored in a database so it is available for any experiment.

The LSTM network is a sequential model with an input LSTM layer of 32 units followed by a fully connected layer of 16 neurons with $tanh$ activation function and a single neuron as output with a $linear$ activation function. The neural network training uses the mean squared error loss function and Adam as the optimizer. The training process is limited to 100 epochs, with an early stopping of 10 epochs of patience.

The implementation of all the modules specified in Section II, with the exception of the inference kernel, are implemented in Python using common libraries like Paho-MQTT, Scikit Learn, GRPC, SQL Alchemy, and Tensorflow. The Preprocessing module receives data from 4 different sensors and appends them into a First-in-First-out queue of size 32 working as a sliding window. The data is then scaled into the interval $[0, 1]$ using the MinMaxScaler algorithm. The Post-Processing module receives the network result and implements the inverse scale transformation.

The description of the acceleration kernel has been performed using HLS in the Xilinx® Vitis™ HLS development environment. This acceleration kernel describes the LSTM neural network with a 32-cell LSTM layer, 4 input features, and a 32-sample time window. This LSTM network also has two dense layers, the first of 16 neurons and the second of one neuron. For the implementation of the MQTT communication protocol, the MQTT-C [29] library has been used.

All the modules, with the exception of the inference, are running on a local host PC inside Docker containers on top of a Linux OS on a CPU Intel Core i7-13700k, a GPU Nvidia RTX 3060-12GB, and a memory RAM of 32 GB. The Model Retrainer uses the power of the GPU to accelerate the training of the neural network. The FPGA is connected to the local LAN network via Ethernet. The FPGA used for the LSTM implementation is the Ultra96v2 evaluation board which contains a Xilinx® Zynq® UltraScale+™ MPSoC device.

B. Experiments and Characterization

To evaluate the performance of the FAS-CT, backtesting experiments have been executed. Different metrics have been monitored during experiments. To characterize the prediction pipeline, the latency of each process, and communications are measured. To characterize the retraining pipeline, the error of the model and the number of retrains are monitored. On the retraining pipeline, the focus is on studying the DDM Drift Detector because it is the module in charge of executing the retraining.

A base model was trained offline with the first 5% of the available data whereas the rest was used for backtesting experiments. All the experiments have been performed using a configuration of 1000 input samples per minute. Three distinct experiments were conducted. The initial test use static scalers, which were initially fitted with the feasible range of values that each sensor can detect to prevent any bias from the training set. The second test involved employing dynamic scalers, during the retrain step the scalers are fitted with the updated train dataset. The final test examined the behavior of the system without any retraining, serving as a baseline. The three tests were conducted using the same base model and initial scalers.

VI. Results and Discussion

A. Latency

The latency of different modules and communications has been measured in the prediction pipeline. The method was to calculate the difference between input and output message timestamps for each module. As a control for communication, the latency between the FAS-CT host and the FPGA was measured using a ping command. The ping package yielded an average latency of 1.253 ± 0.614 milliseconds.

TABLE I
Latency of each process for prediction pipeline in FAS-CT

	Process Name	Process Type	Technology	Latency (ms)
1	Pre-processing	Execution	CPU	0.856 ± 0.457
2	Pre-P → Infer	Communication	CPU	1.442 ± 0.834
3	Inference	Execution	FPGA	1.894 ± 0.085
4	Infer → Post-P	Communication	CPU	1.372 ± 0.890
5	Post-processing	Execution	CPU	0.336 ± 0.150
6	Prediction	Orchestration	FAS-CT	5.792 ± 1.396

The latency between Pre-processing and inference measures the time that it takes for a tensor to arrive at the FPGA (Table I row 2). Similarly, the latency between inference and Post-processing measures the time required for a prediction to reach the Post-processing module (Table I row 4). None of both measurements include the run-time of the involved stages. The execution latency of Pre-processing, Inference, and Post-processing modules has also been measured in Table I rows 1, 3, and 5. Finally, latency measurements have been conducted to determine the time required for generating a prediction from the moment the sensors are polled (Table I row 6). This measurement includes communication and execution of all modules.

B. DDM Backtesting Results

This section focuses on the behavior of the DDM algorithm on the backtest dataset and the consequent start of the re-training pipeline. In table II the experiment results are shown. DDM error metric is calculated in backtests using Eq. 4. The optimal continuous training configuration involves having the least mean DDM error and maximizing the number of samples in the No Drift region.

TABLE II
Backtesting results on retraining pipeline

Scaler Type	Retrain		No Retrain
	Static	Dynamic	Static
Level No Drift %	**65,23%**	35,77%	1,11%
Level Warning %	23,50%	**21,22%**	0,00%
Level Drift %	**11,26%**	43,01%	98,89%
N Retrains	8	24	0
Initial DDM Error	0,1053		
Last DDM Error	0,1653	**0,1475**	1,8
Mean DDM Error	**0,391 ± 1,36**	0,464 ± 1,37	1,335 ± 2,03

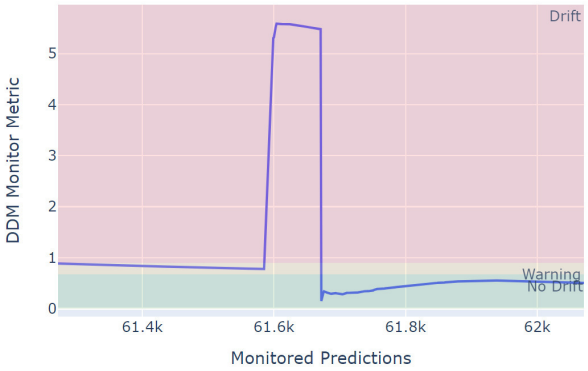

Fig. 3. Outlier in data that causes a sudden increment on DDM error metric, plotted in blue. The green area is the no drift region, the yellow area is the warning region and the red area is the drift region.

Among the three different test configurations, the configuration that yields the best results is the one that retrains the base model and keeps the scalers static. This configuration labels 66.2% of the predictions as No Drift with an average error of 0.391 ± 1.36. These statistics have been achieved with 8 retrains during backtesting.

The configuration with dynamic scalers updates them after each retrain stage. On this configuration, there are 43.01% predictions labelled as Drift. This is 281% more than the previous configuration with static scalers. In contrast with the previous experiment, the final DDM error is lower but with a higher mean DDM error of 0.464 ± 1.37. This is an increase of 18.6% in the DDM error metric with also an increase in the number of retrains. This behaviour is due to the dynamic scalers altering the data distribution after each retraining, worsening the model generalization.

Lastly, the no retraining configuration results in 99% of drifted predictions with an average error of 1.335. Furthermore, in this particular scenario, no prediction was within the warning region as the model encounters an outlier among the first 1.1% of the data that, drastically increments the DDM error metric, as seen in Fig.3. This is the worst configuration meaning that continuous training is needed for this neural network to operate with real-world data.

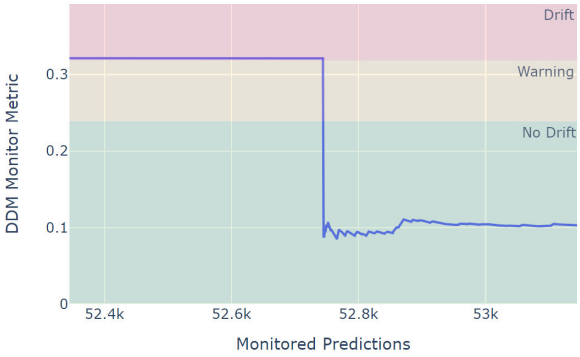

Fig. 4. Retrain number 5 in Table III where after 23333 predictions reach drift level, the retrain pipeline is executed and DDM error metric (plotted in blue) goes again to No Drift region. The green area is the no drift region, the yellow area is the warning region and the red area is the drift region.

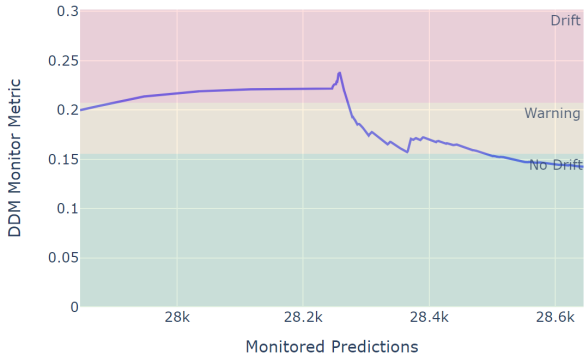

Fig. 5. Retrain number 3 in Table III where the new model could only make 1166 predictions before being substituted. After the retrain number 3 DDM error metric (plotted in blue) does not reach immediately the no drift region. The green area is the no drift region, the yellow area is the warning region and the red area is the drift region.

C. Retraining

After establishing that static scalers are the optimal configuration for FAS-CT with this data, we will now delve closer. We will focus on this experiment, studying how retraining improves the model. The self-retraining process using the static scalers configuration has been detailed in table III. The table includes the model DDM error improvements between each retraining step, as well as the count of predicted samples by the model prior to it being updated.

A successful retrain of the model greatly improves the error metric over its predecessor and generalizes enough that it can make reliable predictions for a substantial portion of samples without triggering another retrain. An example of a successful model retrain would be the fourth retrain, which predicts 23333 samples and presents minimal drift, as seen in the left part of Fig. 4.

TABLE III
DETAILED IMPROVEMENT ON EACH RETRAIN WITH STATIC SCALERS.

Retrain	Previous DDM Error	New DDM Error	Improvement in DDM Error	New Model Predictions
1	12,532	12,085	3,57%	876
2	1,5295	0,1783	88,34%	28248
3	0,2421	0,2217	8,43%	1166
4	0,4559	0,2832	37,89%	23333
5	0,3212	0,0883	72,50%	6035
6	0,2032	0,1368	32,66%	1557
7	0,4283	0,4187	2,24%	1335
8	0,9455	0,1491	84,23%	8053

Furthermore, other retrain attempts are not as successful, such as the first or the third retrain. These retrains happen far from the drift level Eq. 6. Fig. 5 shows the third retrain that only has a slight improvement. This model can only predict 1166 samples before being replaced with a more accurate model.

Lastly, the speed of the retraining process affects the number of predictions beyond the drift level. By reducing the retraining time of the model, the number of predictions in the drift region can be reduced. A comparison between the training performance on the CPU and GPU of the system using Tensorflow Keras is presented in Table IV. The training time depends on various factors such as the number of retrain samples or early stopping configuration. Because of that milliseconds per training batch of 64 samples has been used as a comparative metric.

TABLE IV
TRAINING PERFORMANCE COMPARISON IN DIFFERENT DEVICES

Batch	RTX 3060 12GB Mean (ms)	RTX 3060 12GB Std (ms)	i7 13700k Mean (ms)	i7 13700k Std (ms)
32	2.81	8.05	3.59	4.40
64	2.86	5.22	3.96	5.07
128	2.94	6.95	4.93	7.55
256	3.07	10.02	10.006	11.04
512	3.46	12.97	14.99	14.04

GPU is faster per training batch than CPU as expected. Also, training on GPU leverages CPU that can perform better in other modules like in the prediction pipeline, where low latency is required.

VII. CONCLUSION AND FUTURE WORK

This article introduces FAS-CT, a distributed DL inference architecture with FPGA acceleration and continuous training based on CD detection. This architecture is focused on enhancing the performance and reliability of deep learning predictions in changing or difficult-to-predict environments. To achieve that purpose, FAS-CT is composed of two execution pipelines. First is the prediction pipeline that orchestrates model inference in FPGA. Second is the retraining pipeline which monitors the error metric of the model and manages the actualization of the model.

One of the components of FAS-CT is the CD Detector, an algorithm that labels the model predictions with three possible values, No Drift, Warning, and Drift. Once a prediction is

labelled as Drift, FAS-CT retraining pipeline is launched using two possible configurations, static or dynamic scalers.

The reliability of FAS-CT has been backtested using an LSTM neural network trained for a forecasting task. The results in table II showed that both retraining configurations outperform the default behaviour of a simple prediction pipeline without continuous training. In addition, the implementation with static scalers stands out by labelling 75% fewer predictions as drift and having a lower mean DDM error metric than the implementation using dynamic scalers.

Additionally, the article also studies the latency of each module involved in the prediction pipeline. FPGA has an inference latency of 1.9ms whereas the complete pipeline has an average latency of 5.8ms, with communication between different components accounting for over 2.5ms of the total latency.

Overall, the article demonstrates that FAS-CT is a reliable low-latency DL inference system that adapts over time. This system is suitable for real-time complex tasks that must be executed on the edge. Also, this article demonstrates that is completely feasible the coordination between a drift detector and an FPGA as an accelerator.

A. Future Work

Regression Outlier Resilience: The presence of outliers can significantly influence the efficacy of CD detection. In scenarios where a model fails to accurately regress an outlier value, the DDM update process may erroneously identify CD, triggering the retraining of a stable model.

Synchronous Model Update: Updating a model while ensuring consistent distributions across different components can be a complex task as communication is asynchronous. Updating the model, the DDM parameters or scalers can happen in different timestamps, resulting in incoherent distributions until all the models are updated.

Monolithic Scaling and Inference block: The communication between the Pre-processing, Inference, and Post-processing modules introduces latency to the inference task, as highlighted in Table I. It is possible to consolidate the three blocks into a single monolithic block executed on the FPGA.

Enhancing Dynamic Scalers Configuration: Not all scenarios can be deployed using the static scalers configuration, as the working data interval might be unknown or can drastically change over time. An algorithm that detects when the scalers are outdated so they can be dynamically updated could be developed.

Early CD detection and model adaptation: Since the DDM error metric reaches the drift level until the model is updated, the system makes some predictions in the drift region. These predictions have been minimized using GPU for training. However, it is necessary to reduce them as much as possible. To achieve this, the proposal is to use methods that detect CD early, such as the Early Drift Detector Method [30]. Further research is needed in this area.

ACKNOWLEDGMENT

This work has been founded by: the Ministry of Science and Innovation under CERVERA Excellence Network CER-20211003 (IBERUS) and CER-20211022 (CEL.IA), Missions Science and Innovation MIG-20211008 (INMERBOT), CDTI (Centro para el Desarrollo Tecnológico Industrial) under project CER-20211022, ICE (Junta de Castilla y León) under project CCTT3/20/BU/0002 and CCTT3/20/0003, the SEDIA of Spanish Ministry of Economic and Digital Transformation under the program R&D Missions in Artificial Intelligence MIA.2021.M01.0004, the Spanish Ministry of Economics and Industry under the grant PID2020-112726RB-I00, the Spanish Ministry of Science, Innovation and Universities with code PID2019-107793GB-I00 and TED2021-131983B-I00, the Principado de Asturias under the grant SV-PA-21-AYUD/2021/50994 and the FEDER funds, ref. 1380734.

APPENDIX A: MATHEMATICAL DERIVATION OF MEAN AND STANDARD DEVIATION RECURRENT FORMULAS

Definitions of mean and standard deviation over a set of items $\{e_i\}_{i=1,\dots,n}$ are:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} e_i; \qquad \sigma_n = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(e_i-\mu_n)^2}$$

These definitions imply that all items of e_i must be available in order to calculate μ_n and σ_n for any n. This could be a problem for a limited memory process system if the set is too big or infinite. Due to this limitation, it is necessary to rewrite the mean and the standard deviation definitions as recurrent formulas where only a few values must be stored. Starting with the mean:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} e_i = \frac{n-1}{n}\mu_{n-1} + \frac{1}{n}e_n$$

For the calculation of the mean, it is only necessary to store three values: the previous mean μ_{i-1}, the number of items n, and the last item e_i. Now deriving the same recurrent formula for standard deviation:

$$\sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n}(e_i-\mu_n)^2 = \frac{1}{n}\sum_{i=1}^{n-1}(e_i-\mu_n)^2 + \frac{1}{n}(e_n-\mu_n)^2$$

Notice that the first term is not σ_{n-1}^2 because the mean is the updated mean μ_n and not μ_{n-1}. It is necessary to substitute μ_n with the recurrent formula:

$$\sigma_n^2 = \frac{1}{n^3}\sum_{i=1}^{n-1}\left[n(e_i-\mu_{n-1}) - (e_n-\mu_{n-1})\right]^2 + \frac{1}{n}(e_n-\mu_n)^2$$

Developing the square of the binomial, applying the definition of μ_{n-1} and σ_{n-1}^2 and arranging all the terms:

$$\sigma_n^2 = \frac{n-1}{n}\sigma_{n-1}^2 + \frac{n-1}{n^3}(e_n-\mu_{n-1})^2 + \frac{1}{n}(e_n-\mu_n)^2$$

Now it is possible to derive the second or the third term depending if the final formula is μ_n or μ_{n-1} dependent.

Developing the second term by substituting the expression of μ_{n-1} in terms of μ_n:

$$\frac{n-1}{n^3}\left(e_n-\mu_{n-1}\right)^2=\frac{1}{n\left(n-1\right)}\left(e_n-\mu_n\right)^2$$

Now the second term has the same dependence as the third term. Summing those terms, the recurrent formula for standard deviation is:

$$\sigma_n=\sqrt{\frac{n-1}{n}\sigma_{n-1}^2+\frac{1}{n-1}\left(e_n-\mu_n\right)^2}$$

For the calculus of the standard deviation, it is only necessary to store four values: the previous standard deviation σ_{n-1}, the current mean μ_n, the number of items n, and the last item e_n.

These recurrent formulas for mean and standard deviation can satisfy the memory requirements in a process system where data is continuously arriving like in FAS-CT or any other system that deals with data streams.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. doi: 10.1038/nature14539

[2] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, Feb. 2018. doi: 10.1016/j.future.2017.09.020

[3] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch, "Software Engineering Challenges of Deep Learning," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2018. doi: 10.1109/SEAA.2018.00018 pp. 50–59.

[4] M. P. Véstias, R. P. Duarte, J. T. de Sousa, and H. C. Neto, "Moving Deep Learning to the Edge," *Algorithms*, vol. 13, no. 5, p. 125, May 2020. doi: 10.3390/a13050125

[5] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," Jun. 2021, arXiv:2103.13630 [cs].

[6] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under Concept Drift: A Review," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12, pp. 2346–2363, Dec. 2019. doi: 10.1109/TKDE.2018.2876857

[7] E. A. Castillo and A. Ahmadinia, "A Distributed Smart Camera System Based on an Edge Orchestration Architecture," *Journal of Circuits, Systems and Computers*, vol. 30, no. 04, p. 2150059, Mar. 2021. doi: 10.1142/S0218126621500596

[8] L. D. Biasi, A. A. Citarella, M. Risi, and G. Tortora, "A Cloud Approach for Melanoma Detection Based on Deep Learning Networks," *IEEE Journal of Biomedical and Health Informatics*, vol. 26, no. 3, pp. 962–972, Mar. 2022. doi: 10.1109/JBHI.2021.3113609

[9] H. K. Fatlawi and A. Kiss, "An Adaptive Classification Model for Predicting Epileptic Seizures Using Cloud Computing Service Architecture," *Applied Sciences*, vol. 12, no. 7, p. 3408, Mar. 2022. doi: 10.3390/app12073408

[10] B. Cao, W. Wu, and J. Zhou, "LCFIL: A Loss Compensation Mechanism for Latest Data in Federated Incremental Learning," in *2022 IEEE 19th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. Denver, CO, USA: IEEE, Oct. 2022. doi: 10.1109/MASS56207.2022.00055. ISBN 978-1-66547-180-0 pp. 332–338.

[11] K. Vipin, "ZyNet: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. Tianjin, China: IEEE, Dec. 2019. doi: 10.1109/ICFPT47387.2019.00058. ISBN 978-1-72812-943-3 pp. 323–326.

[12] R. Lozada, J. Ruiz, M. L. González, J. Sedano, J. R. Villar, A. M. García-Vico, and E. S. Skibinsky-Gitlin, "Performance/Resources Comparison of Hardware Implementations on Fully Connected Network Inference," in *Intelligent Data Engineering and Automated Learning – IDEAL 2022*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022. doi: 10.1007/978-3-031-21753-1_34. ISBN 978-3-031-21753-1 pp. 348–358.

[13] Y. Zheng, B. He, and T. Li, "Research on the Lightweight Deployment Method of Integration of Training and Inference in Artificial Intelligence," *Applied Sciences*, vol. 12, no. 13, p. 6616, Jun. 2022. doi: 10.3390/app12136616

[14] J. Violos, S. Tsanakas, T. Theodoropoulos, A. Leivadeas, K. Tserpes, and T. Varvarigou, "Hypertuning GRU Neural Networks for Edge Resource Usage Prediction," in *2021 IEEE Symposium on Computers and Communications (ISCC)*. Athens, Greece: IEEE, Sep. 2021. doi: 10.1109/ISCC53001.2021.9631548. ISBN 978-1-66542-744-9 pp. 1–8.

[15] X. Wang, A. Khan, J. Wang, A. Gangopadhyay, C. Busart, and J. Freeman, "An edge–cloud integrated framework for flexible and dynamic stream analytics," *Future Generation Computer Systems*, vol. 137, pp. 323–335, Dec. 2022. doi: 10.1016/j.future.2022.07.023

[16] M. L. González, R. Lozada, J. Ruiz, E. S. Skibinsky-Gitlin, A. M. García-Vico, J. Sedano, and J. R. Villar, "Exploring the implementation of LSTM inference on FPGA [Manuscript submitted for publication]," *Proc. of the International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME 2023)*, 2023.

[17] S. An and U. Y. Ogras, "MARS: mmWave-based Assistive Rehabilitation System for Smart Healthcare," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5s, pp. 72:1–72:22, Sep. 2021. doi: 10.1145/3477003

[18] A. Pardos, A. Menychtas, and I. Maglogiannis, "On unifying deep learning and edge computing for human motion analysis in exergames development," *Neural Computing and Applications*, vol. 34, no. 2, pp. 951–967, Jan. 2022. doi: 10.1007/s00521-021-06181-6

[19] W. Jiang, X. Ye, R. Chen, F. Su, M. Lin, Y. Ma, Y. Zhu, and S. Huang, "Wearable on-device deep learning system for hand gesture recognition based on FPGA accelerator," *Mathematical Biosciences and Engineering*, vol. 18, no. 1, pp. 132–153, 2021. doi: 10.3934/mbe.2021007

[20] B. C. Dos Santos Melício, G. Baranyi, Z. Gaál, S. Zidan, and A. Lorincz, "DeepRehab: Real Time Pose Estimation on the Edge for Knee Injury Rehabilitation," in *Artificial Neural Networks and Machine Learning – ICANN 2021*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021. doi: 10.1007/978-3-030-86365-4_31. ISBN 978-3-030-86365-4 pp. 380–391.

[21] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[22] "Mqtt specification," https://mqtt.org/mqtt-specification/.

[23] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," *Journal of Open Source Software*, vol. 2, no. 13, p. 265, 2017. doi: 10.21105/joss.00265

[24] B. Mishra, B. Mishra, and A. Kertesz, "Stress-testing mqtt brokers: A comparative analysis of performance measurements," *Energies*, vol. 14, no. 18, 2021. doi: 10.3390/en14185817

[25] "grpc," https://grpc.io/.

[26] J. Gama, P. Medas, G. Castillo, and P. Rodrigues, "Learning with Drift Detection," in *Advances in Artificial Intelligence – SBIA 2004*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004. doi: 10.1007/978-3-540-28645-5_29. ISBN 978-3-540-28645-5 pp. 286–295.

[27] "Mlflow - a platform for the machine learning lifecycle | mlflow," https://mlflow.org/.

[28] M. L. González, J. Sedano, A. M. García-Vico, and J. R. Víllar, "A Comparison of Techniques for Virtual Concept Drift Detection," in *16th International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2021)*, ser. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2022. doi: 10.1007/978-3-030-87869-6_1. ISBN 978-3-030-87869-6 pp. 3–13.

[29] "Liambindle/mqtt-c: A portable mqtt c client for embedded systems and pcs alike." https://github.com/LiamBindle/MQTT-C.

[30] M. Baena-Garcıa, R. Gavalda, and R. Morales-Bueno, "Early Drift Detection Method."