An On-Device Learning-Based Anomaly Detection Approach for Resource-Limited Edge Devices

Mineto Tsukada

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

School of Science for Open and Environmental Systems
Graduate School of Science and Technology
Keio University

March 2023

Preface

To date, a massive number of researches on neural network-based technologies for edge devices have been reported. The boom made several product-level neural network-assisted devices on the market, and new products are still being released. One feature most of these products share in common is to execute only prediction (inference) computations on a device, while training is done in a server machine with high computation power; in contrast to them, the main focus of the thesis is **on-device learning**, where both prediction and training computations are executed all on-device, the most edge-heavy architecture among possible ones in an edge-cloud cooperative system. The aim of this thesis is to explore the advantages of on-device learning in anomaly detection area over prediction-only approaches.

Semi-supervised anomaly detection is a method for identifying anomaly data samples by learning the distribution of normal data. Backpropagation neural network (i.e., BP-NN) based approaches have increasingly attracted wide attention for their high anomaly detection accuracy and robustness. In a typical edge-cloud system, the BP-NN-based models are trained in a batch manner using cloud servers with massive data samples gathered from edge devices, and trained weights are transferred to the prediction-only edge devices. However, the system has the following two problems: (1) BP-NN's batch training often takes a considerably long time, which makes it difficult to follow time-series changes in the distribution of normal data (i.e., concept drift), and (2) data communication between edge and cloud is inevitable and it may impose additional energy consumption and potential risk of data breaches. To address these issues, this thesis proposes a new semi-supervised anomaly detection approach, named **ONLAD (ON-device Learning semi-supervised Anomaly Detection)**, where input data samples are sequentially learned on edge devices in an online manner without server machines. ONLAD allows edge devices to immediately follow concept drift and enables standalone execution where data transfers from/to cloud servers are no more required. The thesis also proposes ONLAD Core, an hardware IP Core implementation of ONLAD, which is able to execute fast

and energy-efficient prediction and training with tiny hardware resources. These technologies, ONLAD and ONLAD Core are the core of the thesis.

The thesis proposes some extended technologies on top of the core technologies towards higher anomaly detection accuracy of the ONLAD algorithm and overflow/underflow-free circuits of ONLAD Core for further stability. Finally, as an application on top of the proposed technologies, ONLAD Sensor, an ONLAD-based wireless sensor for anomaly detection is proposed. The proposed sensor is compared with a prediction-only counterpart. Experiments demonstrate that the on-device learning approach improves anomaly detection accuracy with its fast sequential training functionality at a concept-drifting environment while saving computation and communication costs for low power.

May this thesis contribute to the prosperity of the on-device learning community, sincerely.

Acknowledgments

My doctoral study has been supported by a number of people. I would like to thank all of them.

My supervisor, Professor Hiroki Matsutani, has always supported my study. He has always given valuable suggestions whenever I had a difficulty in my research. He has always reviewed every paper that I have written and given me valuable comments to improve the papers. I would like to thank for his constant support to my study.

I am grateful to my doctoral committee members, Professor Hideharu Amano, Professor Masaaki Kondo, and Assistant Professor Yasushi Narushima for their careful reviews and comments to my thesis.

Finally, I would like to thank my parents and family for their continued and warm support.

Mineto Tsukada
Yokohama, Japan
March 2023

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Anomaly detection is an approach to identify rare data instances (i.e., anomalies) that have different patterns or come from different distributions from that of the majority (i.e., the normal class) [2]. There are mainly three approaches in anomaly detection: (1) supervised anomaly detection, (2) semi-supervised anomaly detection, and (3) unsupervised anomaly detection.

(1) A typical strategy of supervised anomaly detection is to build a binary-classification model for the normal class vs. the anomaly class. It requires labeled normal and anomaly data to train a model; however anomaly instances are basically much rarer than normal ones, which imposes the class-imbalanced problem [3]. Several works have addressed this issue by under-sampling the majority data or oversampling the minority data [4, 5], or assigning more costs on misclassified data to make the classifier concentrate minority classes [6].

(2) Semi-supervised anomaly detection, one of the main topics of this thesis, assumes that all the training data belong to the normal class [2]. A typical strategy of semi-supervised anomaly detection is to learn the distribution of normal data and then identify data samples distant from the distribution as anomalies. Semi-supervised approaches do not require anomalies to train a model, which makes them applicable to a wide range of real-world tasks. Various approaches have been proposed, such as nearest-neighbor based techniques [7, 8], clustering approaches [9, 10], and one-class classification approaches [11, 12].

(3) Unsupervised anomaly detection does not require labeled training data [2], thus its constraint is the least restrictive among the three approaches. Many semi-supervised

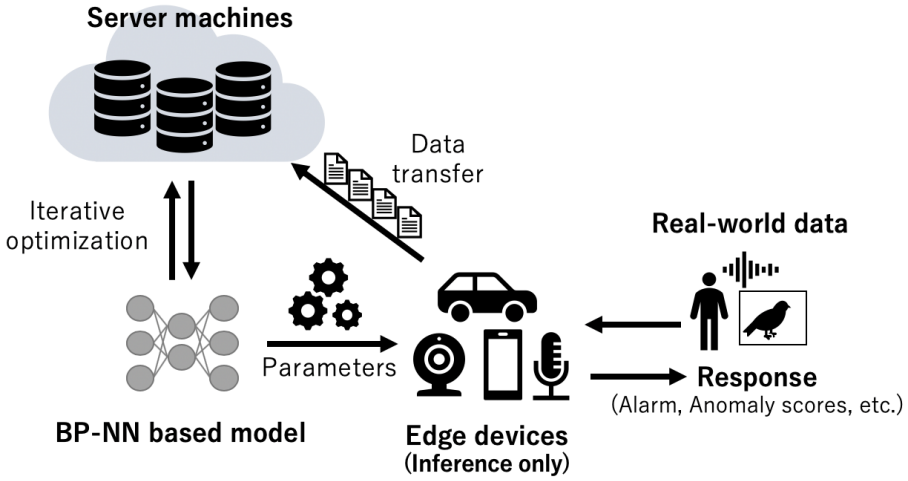

Figure 1.1: Typical Edge-Cloud System Leveraging BP-NN-based Semi-Supervised Anomaly Detection

methods can be used in an unsupervised manner by using unlabeled data to train a model because most unlabeled data belong to the normal class. Sometimes unsupervised anomaly detection and semi-supervised anomaly detection are not distinguished explicitly.

The focus of this thesis is semi-supervised anomaly detection. Recently, neural network-based approaches [13–15] have been drawing attention because in many cases neural networks can achieve relatively higher generalization performance than traditional approaches for a wide range of real-world data such as images, natural languages, and audio data, by stacking a number of layers. Although there are several variants of neural networks, backpropagation-based neural networks (i.e., BP-NNs) are now the mainstream and widely used.

Figure 1.1 illustrates a typical edge-cloud system using BP-NN-based semi-supervised anomaly detection models. The system shown in the figure is designed for edge devices that implement their own models to detect anomalies of incoming real-world data. In this system, the edge devices are supposed to perform only prediction computations (e.g., calculating anomaly scores), and training computations are offloaded to server machines. The models are iteratively trained in the server machines with a large amount of input data gathered from the edge devices. Once the training loop completes, the parameters of edge devices are updated with the optimized ones. However, there are two issues with this approach: (1) BP-NNs' iterative optimization approach often takes considerable computation time, which makes it difficult to follow time-series changes in the distribution of normal data (i.e., concept drift). (2) Data transfers to the server machines may impose several problems on the edge devices such as additional latency and energy consumption

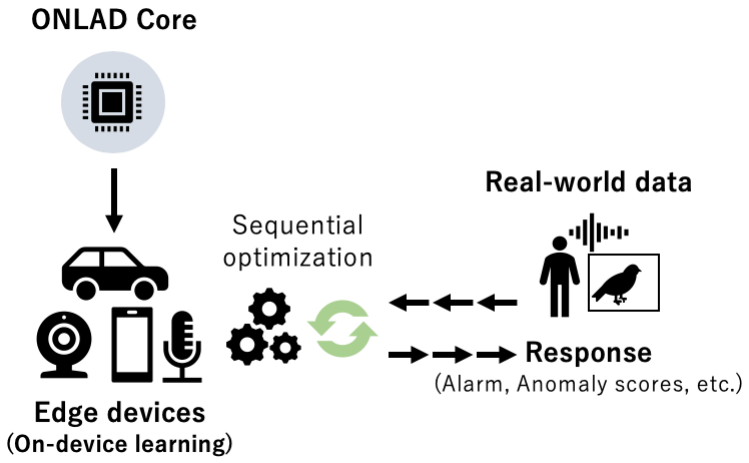

Figure 1.2: On-device Sequential Learning Approach

for communication.

(1) As mentioned before, learning the distribution of normal data is a key feature of semi-supervised anomaly detection approaches. However, the distribution may change over time. This phenomenon is referred to as concept drift. Concept drift is a serious problem when there are frequent changes in the surrounding environment of data [16] or behavioral state changes in data sources [17]. A semi-supervised anomaly detection model should learn new normal data to follow the changes; however BP-NNs' iterative optimization approach often introduces a considerable delay, which widens a gap between the latest true distribution of normal data and the one learned by the model [18]. This gap makes identifying anomalies more difficult gradually.

(2) Usually, edge devices implementing machine learning models are specialized only for prediction computations because the backpropagation method often requires a large amount of computational power. This is why training computations of BP-NNs are typically offloaded to server machines with high computational power. In this case, data transfers to the server machines are inevitable, which imposes additional energy consumption for communication and potential risks of data breaches on the edge devices.

One practical solution to these two issues is the on-device sequential learning approach illustrated in Figure 1.2. In this approach, incoming input data are sequentially learned on edge devices themselves. This approach allows the edge devices to sequentially follow changes in the distribution of normal data and makes possible standalone execution where no data transfers are required. However, it poses challenges in regard to how to construct such a sequential learning algorithm and how to implement it on edge devices with limited resources.

To deal with the underlying challenges, we propose an ON-device sequential Learning semi-supervised Anomaly Detector called **ONLAD** and its IP core, named **ONLAD Core**. The algorithm of ONLAD is designed to perform fast sequential learning to follow concept drift in less than one millisecond. ONLAD Core realizes on-device learning for resource-limited edge devices at low power consumption.

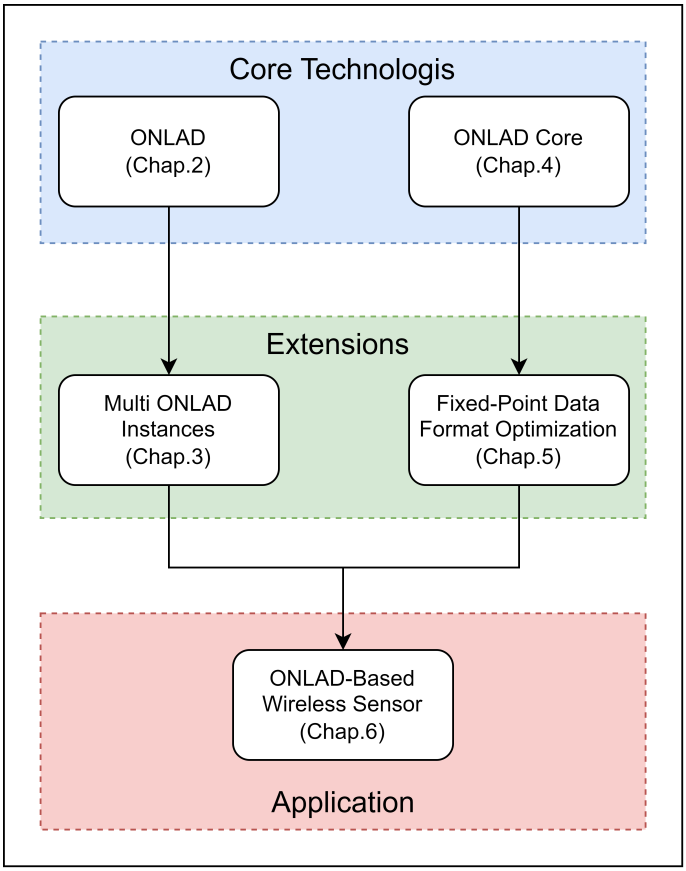

Figure 1.3: Relationships Between Proposed Technologies of Thesis

1.2 Thesis Organization

Figure 1.3 illustrates the relationships between the proposed technologies. The ONLAD algorithm and its IP core implementation ONLAD Core positioned as core technologies of this thesis are introduced in Chapter 2 and Chapter 4, respectively.

As extensions of the core technologies, Chapter 3 introduces an ensemble approach leveraging multiple ONLAD instances to extend the representation capability of ONLAD and achieve more anomaly detection accuracy. Also Chapter 5 proposes a fixed-point data format optimization method towards overflow/underflow-free OS-ELM digital circuits for

more stability. The proposed optimization method is for general OS-ELM digital circuits; it can be applied to ONLAD Core since OS-ELM, a light-weight neural-network variant, is a core component of the ONLAD algorithm.

Then Chapter 6 introduces an ONLAD-based wireless sensor node for anomaly detection, called **ONLAD Sensor**. ONLAD Sensor is an all-in-one wireless sensor where sensing, prediction, and training are executed all on-device. ONLAD Sensor can quickly adapt to a given environment where concept-drift may occur, leveraging its fast on-device sequential training functionality based on the ONLAD algorithm, while saving computation and communication costs for low power.

Chapter 7 describes related works of the technologies proposed in this thesis. The chapter clarifies the novelties and positions of the proposed technologies.

Chapter 8 clarifies the contributions of the proposed technologies then concludes this thesis.

Chapter 2

ONLAD

This chapter derives the algorithm of ONLAD, the origin of this thesis. Section 2.1 gives a brief introduction of base technologies behind ONLAD for ease of understanding. Section 2.2 makes a cost analysis of OS-ELM, a building block of the ONLAD algorithm, and derives a surprisingly simple technique to minimize computational and space complexities of the algorithm. Then the ONLAD algorithm is formulated. Section 2.3 evaluates the anomaly detection performance of ONLAD in comparison with the other BP-NN-based anomaly detection models assuming a static environment and a dynamic environment where concept drift occurs.

2.1 Preliminaries

In this section, three base technologies behind ONLAD: (1) ELM (Extreme Learning Machine), (2) OS-ELM (Online Sequential Extreme Learning Machine), and (3) Autoencoder, are briefly described.

2.1.1 ELM

ELM (Extreme Learning Machine) [19] illustrated in Figure 2.1 is a variant of single hidden layer feed-forward networks (i.e., SLFNs) which consist of an input layer, a hidden layer, and an output layer. n, $\tilde{N}$, and m are the numbers of input, hidden, and output nodes, respectively. Suppose an n-dimensional input $\boldsymbol{x} \in \mathbb{R}^{k \times n}$, where k is the batch size, is given; an m-dimensional output chunk $\boldsymbol{y} \in \mathbb{R}^{k \times m}$ is computed as follows.

$$\boldsymbol{y} = \mathrm{G}(\boldsymbol{x} \cdot \boldsymbol{\alpha} + \boldsymbol{B})\boldsymbol{\beta}, \tag{2.1}$$

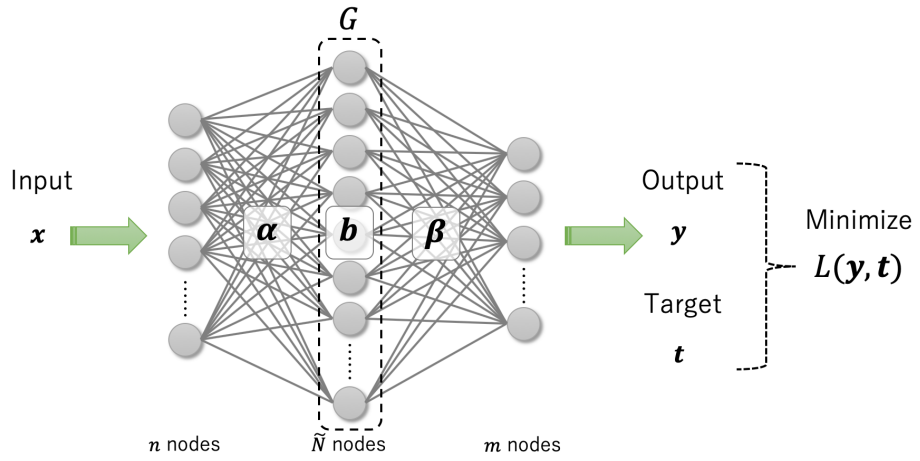

Figure 2.1: Extreme Learning Machine

where $\boldsymbol{\alpha} \in \mathbb{R}^{n \times \tilde{N}}$ is the input weight matrix connecting input layer and hidden layer, and $\boldsymbol{\beta} \in \mathbb{R}^{\tilde{N} \times m}$ is the output weight matrix connecting hidden layer and output layer. $\boldsymbol{B} \in \mathbb{R}^{k \times \tilde{N}}$ is the set of k vertical duplications of the bias vector $\boldsymbol{b} \in \mathbb{R}^{1 \times \tilde{N}}$[1]. G represents the activation function applied to the hidden layer output.

If an SLFN can approximate an m-dimensional target chunk $\boldsymbol{t} \in \mathbb{R}^{k \times m}$ with zero error, it implies that there exists $\boldsymbol{\beta}$ that satisfies the following equation.

$$\mathrm{G}(\boldsymbol{x} \cdot \boldsymbol{\alpha} + \boldsymbol{B})\boldsymbol{\beta} = \boldsymbol{t} \tag{2.2}$$

Let $\boldsymbol{H} \in \mathbb{R}^{k \times \tilde{N}}$ be the hidden layer output $\mathrm{G}(\boldsymbol{x} \cdot \boldsymbol{\alpha} + \boldsymbol{B})$; then the optimal output weight $\hat{\boldsymbol{\beta}}$ is given by

$$\hat{\boldsymbol{\beta}} = \boldsymbol{H}^{\dagger}\boldsymbol{t}, \tag{2.3}$$

where $\boldsymbol{H}^{\dagger}$ is the pseudo inverse of $\boldsymbol{H}$. $\boldsymbol{H}^{\dagger}$ can be calculated with several matrix decomposition methods such as SVD (Singular Value Decomposition) [20]. In particular, as long as $\mathrm{rank}(\boldsymbol{H}) = \tilde{N}$, $\boldsymbol{H}^{\dagger}$ can be calculated in an efficient way with $\boldsymbol{H}^{\dagger} = (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T$ or $\boldsymbol{H}^{\dagger} = \boldsymbol{H}^T(\boldsymbol{H}\boldsymbol{H}^T)^{-1}$.

The whole training procedure finishes by replacing $\boldsymbol{\beta}$ with $\hat{\boldsymbol{\beta}}$. $\boldsymbol{\beta}$ is the only training parameter of ELM. $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ are constant parameters that can be initialized with any random values, meaning the conversion $\boldsymbol{x} \to \boldsymbol{H}$ is a sort of random projection.

[1] $\boldsymbol{B} \in \mathbb{R}^{k \times \tilde{N}} \equiv \begin{bmatrix} \boldsymbol{b} \\ \vdots \\ \boldsymbol{b} \end{bmatrix}$

2.1.1.1 Advantages over BP-NN

Advantages of ELM over BP-NN are summarized in the following three remarks.

1. **Global Solution**
   BP-NN's optimization method is based on stochastic gradient descent which suffers from the local minima problem [21] and sub-optimal solutions can be produced. ELM, in contrast to BP-NN, always gives the global solution $\hat{\boldsymbol{\beta}}$ meaning that ELM minimizes training error analytically.

2. **Fast Optimization**
   Basically, BP-NN requires a number of training epochs on a dataset. ELM can finish the whole training procedure with one-shot optimization, which makes training time much shorter than BP-NN [19].

3. **Few Hyper-parameters**
   BP-NN has a number of hyper-parameters that affect training results: the number of intermediate layers, configuration of each layer, activation function, loss function, optimization algorithm, batch size, the number of training epochs, and so on. On the other hand, ELM's hyper-parameters are only an activation function and the number of hidden nodes, which saves a lot of time for hyperparameter tuning [19].

2.1.1.2 Drawback

One of the biggest drawbacks of ELM is being a batch learning algorithm. ELM cannot learn training data one by one or chunk by chunk; the entire training dataset is assumed to be available in advance. To train an ELM model with an additional set of training data, it must be re-trained with the entire set of training data including past training data. Since the computational complexity of the training algorithm of ELM (Equation 2.3) is $\mathrm{O}(N^3)$, where N equals the total number of training samples, training time rapidly increases if the size of dataset is large. Hence, ELM is not an appropriate choice when the dataset is large or the entire dataset is not available in advance.

2.1.2 OS-ELM

OS-ELM (Online Sequential Extreme Learning Machine) [22] is an ELM variant that can perform sequential learning instead of batch learning.

Given the initial training chunk $\{\boldsymbol{x}_0 \in \mathbb{R}^{k_0 \times n}, \boldsymbol{t}_0 \in \mathbb{R}^{k_0 \times m}\}$, ELM finds the solution by $\boldsymbol{\beta}_0 = \boldsymbol{K}_0^{-1}\boldsymbol{H}_0^T\boldsymbol{t}_0$, where $\boldsymbol{H}_0 \equiv \mathrm{G}(\boldsymbol{x}_0 \cdot \boldsymbol{\alpha} + \boldsymbol{B})$ and $\boldsymbol{K}_0 \equiv \boldsymbol{H}_0^T\boldsymbol{H}_0$. Then, if the next training chunk

$\{\boldsymbol{x}_1 \in \mathbb{R}^{k_1 \times n}, \boldsymbol{t}_1 \in \mathbb{R}^{k_1 \times m}\}$ is given, ELM finds the solution $\boldsymbol{\beta}_1$ that satisfies $\begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix} \boldsymbol{\beta}_1 = \begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \end{bmatrix}$ by the following equation.

$$\boldsymbol{\beta}_1 = \boldsymbol{K}_1^{-1} \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \end{bmatrix}, \tag{2.4}$$

where $\boldsymbol{H}_1 \equiv \mathrm{G}(\boldsymbol{x}_1 \cdot \boldsymbol{\alpha} + \boldsymbol{b})$ and $\boldsymbol{K}_1 \equiv \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}$. Equation 2.4 is the ELM training algorithm with $\boldsymbol{H}$ and $\boldsymbol{t}$ replaced with $\begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}$ and $\begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \end{bmatrix}$, respectively. ELM needs re-training with the past data $\{\boldsymbol{x}_0, \boldsymbol{t}_0\}$ to train with the new data $\{\boldsymbol{x}_1, \boldsymbol{t}_1\}$, as pointed out in the previous section. To find $\boldsymbol{\beta}_1$ without re-training with past data, $\boldsymbol{\beta}_1$ and $\boldsymbol{K}_1$ must be computed without $\boldsymbol{x}_0$ and $\boldsymbol{t}_0$.

$\boldsymbol{K}_1$ can be written in a recurrence form as follows;

$$\begin{aligned} \boldsymbol{K}_1 &= \begin{bmatrix} \boldsymbol{H}_0^T & \boldsymbol{H}_1^T \end{bmatrix} \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix} \\ &= \boldsymbol{K}_0 + \boldsymbol{H}_1^T \boldsymbol{H}_1. \end{aligned} \tag{2.5}$$

Also, $\begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \end{bmatrix}$ can be decomposed as follows by using Equation 2.5;

$$\begin{aligned} \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \end{bmatrix} &= \boldsymbol{H}_0^T \boldsymbol{t}_0 + \boldsymbol{H}_1^T \boldsymbol{t}_1 \\ &= \boldsymbol{K}_0 \boldsymbol{K}_0^{-1} \boldsymbol{H}_0^T \boldsymbol{t}_0 + \boldsymbol{H}_1^T \boldsymbol{t}_1 \\ &= \boldsymbol{K}_0 \boldsymbol{\beta}_0 + \boldsymbol{H}_1^T \boldsymbol{t}_1 \\ &= (\boldsymbol{K}_1 - \boldsymbol{H}_1^T \boldsymbol{H}_1) \boldsymbol{\beta}_0 + \boldsymbol{H}_1^T \boldsymbol{t}_1 \\ &= \boldsymbol{K}_1 \boldsymbol{\beta}_0 - \boldsymbol{H}_1^T \boldsymbol{H}_1 \boldsymbol{\beta}_0 + \boldsymbol{H}_1^T \boldsymbol{t}_1. \end{aligned} \tag{2.6}$$

$\boldsymbol{\beta}_1$ can be written in a recurrence form by combining Equation 2.4 and Equation 2.6.

$$\begin{aligned} \boldsymbol{\beta}_1 &= \boldsymbol{K}_1^{-1} \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \end{bmatrix}^T \begin{bmatrix} \boldsymbol{t}_0 \\ \boldsymbol{t}_1 \end{bmatrix} \\ &= \boldsymbol{K}_1^{-1} (\boldsymbol{K}_1 \boldsymbol{\beta}_0 - \boldsymbol{H}_1^T \boldsymbol{H}_1 \boldsymbol{\beta}_0 + \boldsymbol{H}_1^T \boldsymbol{t}_1) \\ &= \boldsymbol{\beta}_0 + \boldsymbol{K}_1^{-1} \boldsymbol{H}_1^T (\boldsymbol{t}_1 - \boldsymbol{H}_1 \boldsymbol{\beta}_0), \end{aligned} \tag{2.7}$$

where $\boldsymbol{K}_1 = \boldsymbol{K}_0 + \boldsymbol{H}_1^T \boldsymbol{H}_1$. Note that Equation 2.7 does not require the past data $\boldsymbol{x}_0$ and $\boldsymbol{t}_0$.

Let's consider the recurrence form of general $\boldsymbol{\beta}_i$. Given the ith training chunk $\{\boldsymbol{x}_i \in \mathbb{R}^{k_i \times n}, \boldsymbol{t}_i \in \mathbb{R}^{k_i \times m}\}$, we have the following equations by recursively calculating subsequent $\boldsymbol{\beta}_2, \boldsymbol{\beta}_3, \ldots, \boldsymbol{\beta}_i$ in the same way as Equation 2.5 ~ Equation 2.7.

$$\begin{aligned} \boldsymbol{K}_i &= \boldsymbol{K}_{i-1} + \boldsymbol{H}_i^T \boldsymbol{H}_i \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{K}_i^{-1} \boldsymbol{H}_i^T (\boldsymbol{t}_i - \boldsymbol{H}_i \boldsymbol{\beta}_{i-1}), \end{aligned} \tag{2.8}$$

where $\boldsymbol{K}_i \equiv \begin{bmatrix} \boldsymbol{H}_0 \\ \vdots \\ \boldsymbol{H}_i \end{bmatrix}^T \begin{bmatrix} \boldsymbol{H}_0 \\ \vdots \\ \boldsymbol{H}_i \end{bmatrix}$. Note that $\boldsymbol{K}_i^{-1}$ rather than $\boldsymbol{K}_i$ is used to compute $\boldsymbol{\beta}_i$. The recurrence form of $\boldsymbol{K}_i^{-1}$ is derived using the Woodbury formula [23] [2].

$$\begin{aligned} \boldsymbol{K}_i^{-1} &= (\boldsymbol{K}_{i-1} + \boldsymbol{H}_i^T \boldsymbol{H}_i)^{-1} \\ &= \boldsymbol{K}_{i-1}^{-1} - \boldsymbol{K}_{i-1}^{-1} \boldsymbol{H}_i^T (\boldsymbol{I} + \boldsymbol{H}_i \boldsymbol{K}_{i-1}^{-1} \boldsymbol{H}_i^T)^{-1} \boldsymbol{H}_i \boldsymbol{K}_{i-1}^{-1} \end{aligned} \tag{2.9}$$

Let $\boldsymbol{P}_i \equiv \boldsymbol{K}_i^{-1}$, the final form of the OS-ELM training algorithm is derived.

$$\begin{aligned} \boldsymbol{P}_i &= \boldsymbol{P}_{i-1} - \boldsymbol{P}_{i-1} \boldsymbol{H}_i^T (\boldsymbol{I} + \boldsymbol{H}_i \boldsymbol{P}_{i-1} \boldsymbol{H}_i^T)^{-1} \boldsymbol{H}_i \boldsymbol{P}_{i-1} \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i \boldsymbol{H}_i^T (\boldsymbol{t}_i - \boldsymbol{H}_i \boldsymbol{\beta}_{i-1}) \end{aligned} \tag{2.10}$$

$\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ are computed as follows.

$$\begin{aligned} \boldsymbol{P}_0 &= (\boldsymbol{H}_0^T \boldsymbol{H}_0)^{-1} \\ \boldsymbol{\beta}_0 &= \boldsymbol{P}_0 \boldsymbol{H}_0^T \boldsymbol{t}_0 \end{aligned} \tag{2.11}$$

Make sure that $\boldsymbol{H}_0^T \boldsymbol{H}_0 \in \mathbb{R}^{\tilde{N} \times \tilde{N}}$ is not a singular matrix. Mathematically speaking the rank of $\boldsymbol{H}_0 \in \mathbb{R}^{k_0 \times \tilde{N}}$ must be $\tilde{N}$, meaning that the number of initial training samples k_0 should be much greater than that of hidden nodes $\tilde{N}$.

Remarks of OS-ELM are summarized as follows.

1. **Superset of ELM**

   If the initial batch size k_0 is equal to the number of all the training samples (in this case only the initial training is performed) OS-ELM is mathematically equal to ELM. OS-ELM can be seen as a superset of ELM.

2. **Fast Sequential Learning**

   OS-ELM sequentially finds the optimal output weight for the new training chunk without memory or re-training with past training data, unlike ELM. Also, OS-ELM is known to find the optimal solution faster than BP-NNs [22].

[2] $(\boldsymbol{A} + \boldsymbol{UCV})^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}(\boldsymbol{C}^{-1} + \boldsymbol{V}\boldsymbol{A}^{-1}\boldsymbol{U})^{-1}\boldsymbol{V}\boldsymbol{A}^{-1}$

3. **Batch Size Does Not Affect Training Results**
   OS-ELM finds the same solution $\boldsymbol{\beta}_i$ if the training dataset is the same regardless of batch size k_i.

4. **Same Learning Results with ELM**
   When rank($\boldsymbol{H}_0$) = $\tilde{N}$, OS-ELM and ELM obtain the same solution as long as the same training dataset is used.

2.1.3 Autoencoder

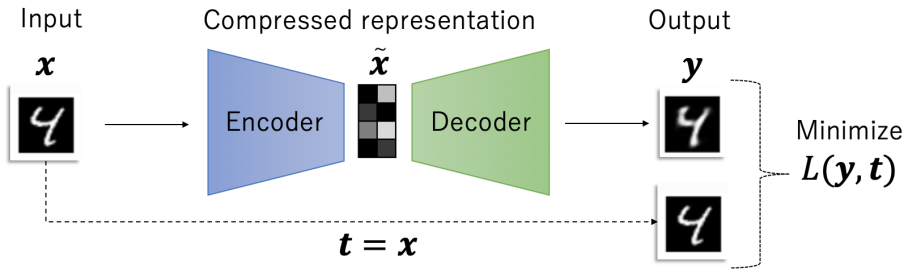

Figure 2.2: Autoencoder

An autoencoder [24] illustrated in Figure 2.2 is a neural network-based unsupervised learning model for finding a well-characterized dimensionality-reduced form $\tilde{\boldsymbol{x}} \in \mathbb{R}^{k \times \tilde{n}}$ of an input chunk $\boldsymbol{x} \in \mathbb{R}^{k \times n}$, where $\tilde{n} < n$. An autoencoder takes $\boldsymbol{x}$ as an input and predicts $\tilde{\boldsymbol{x}}$ in the middle layer of the model. The bottom half of the model is called "Encoder" because it encodes a raw input $\boldsymbol{x}$ into the dimensionality-reduced form $\tilde{\boldsymbol{x}}$, while the top half is called "Decoder" as it decodes $\tilde{\boldsymbol{x}}$ back into $\boldsymbol{x}$. The decoder part is unnecessary for dimensionality reduction, but is used during training.

In the training process, an input is used as its target (i.e., $\boldsymbol{t} = \boldsymbol{x}$), meaning an autoencoder is trained so that it can correctly reconstruct input data as output data. It is empirically known that $\tilde{\boldsymbol{x}}$ tends to become well-characterized when the training error between input data and reconstructed output data converges [24]. Labeled data are not required during the whole training process; this is why an autoencoder is categorized as an unsupervised learning model. Autoencoders are not only used for dimensionality-reduction but also pre-training models and generating complicated data.

Several autoencoder variants have been proposed over the years. Sparse autoencoders produce sparse dimensionality-reduced forms by adding an L0 regularization term limiting the number of non-zero element of $\tilde{\boldsymbol{x}}$, into the loss function [25]. A De-noising

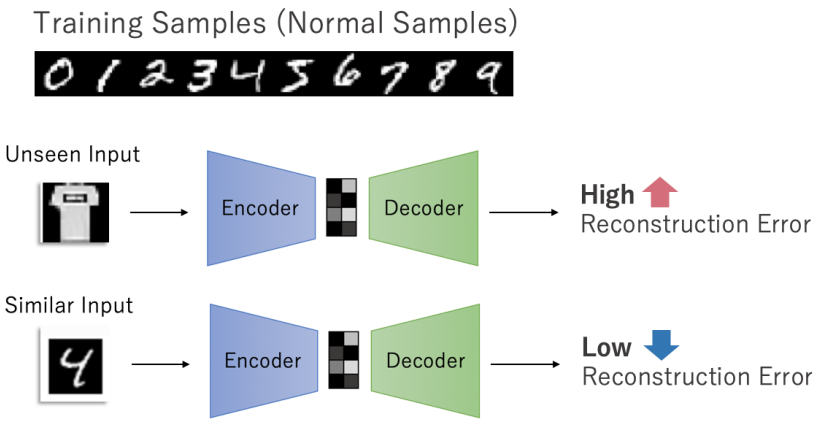

Figure 2.3: Autoencoder-Based Anomaly Detection

autoencoder puts a random noise on $\boldsymbol{x}$ and is trained so that it can remove the noise of $\boldsymbol{x}$. De-noising-autoencoder is known to be more robust than the original autoencoder [26].

2.1.4 Semi-Supervised Anomaly Detection Using Autoencoder

Autoencoders have been attracting attention in the field of semi-supervised anomaly detection [14,27]. To perform semi-supervised anomaly detection, an autoencoder is trained only with normal data, which makes reconstruction error relatively high when anomaly data (not similar to the normal data) are fed to the model. Thus anomaly data can be detected by setting a threshold for reconstruction error. This approach is categorized as a semi-supervised anomaly detection method since only normal data is used for training.

PCA (Principal Component Analysis) [28] and kernel PCA [29] are often compared with autoencoders. Sakurada *et al.* showed that autoencoder-based anomaly detection models can detect subtle anomalies that PCA and kernel PCA fail to pick up [14]. Also, the autoencoder-based models can perform nonlinear transformation without costly computations that kernel PCA requires, which saves much execution time.

2.2 Method

ONLAD leverages OS-ELM as its core component. Section 2.2.1 offers a theoretical analysis on the computational cost and the space cost of the OS-ELM training algorithm then Section 2.2.2 demonstrates that these costs are minimized when batch size = 1 without any deterioration of training stability or results. Also, Section 2.2.3 reviews one of the latest OS-ELM variants with a forgetting mechanism, called FP-ELM (Forgetting Parameter ELM) and proposes a new light-weight forgetting mechanism to tackle with concept drift at low computational cost. Finally, the algorithm of ONLAD is formulated in Section 2.2.4.

2.2.1 Cost Analysis of OS-ELM

In the following sections n, $\tilde{N}$ and m are the numbers of input, hidden, output nodes, respectively. k is batch size. S_k represents the space cost of the OS-ELM training algorithm while I_k is the computational cost.

2.2.1.1 Space Cost

The bottleneck of space cost is the memory space for matrices and vectors existing in the algorithm. Suppose the space cost of a matrix $\boldsymbol{A} \in \mathbb{R}^{p \times q}$ is pq and that of a vector $\boldsymbol{a} \in \mathbb{R}^r$ is r, the total space cost when batch size = k is calculated as follows.

$$S_k = k^2 + (3\tilde{N} + m)k + \tilde{N}^2 \tag{2.12}$$

Equation 2.12 shows that the space cost is proportional to the squares of batch size and the number of hidden nodes (i.e., $\mathrm{O}(k^2 + \tilde{N}^2)$).

2.2.1.2 Computational Cost

The bottleneck of computational cost is those of (1) matrix products and (2) matrix inversions. Suppose the computational cost of a matrix product $\boldsymbol{A} \in \mathbb{R}^{p \times q} \cdot \boldsymbol{b} \in \mathbb{R}^{q \times r}$ is pqr and that of a matrix inversion $\boldsymbol{C}^{-1} \in \mathbb{R}^{r \times r}$ is r^3, the total computational cost when batch size = k is calculated as follows.

$$I_k = k^3 + 2\tilde{N}k^2 + \tilde{N}(4\tilde{N} + 2m + 2n)k \tag{2.13}$$

Equation 2.13 shows that the computational cost is proportional to the cube of batch size (i.e., $\mathrm{O}(k^3)$).

2.2.2 Insight of Cost Analysis

The following equations are derived from Equation 2.13.

$$\begin{aligned} I_k &= k^3 + 2\tilde{N}k^2 + \tilde{N}(4\tilde{N} + 2m + 2n)k \\ &= k(k^2 + 2\tilde{N}k + \tilde{N}(4\tilde{N} + 2m + 2n)) \\ &\geq k(1 + 2\tilde{N} + \tilde{N}(4\tilde{N} + 2m + 2n)) \\ &= kI_1 \end{aligned} \tag{2.14}$$

Finally, $I_k \geq kI_1$ is derived. This inequality shows that the total computational cost of the OS-ELM training algorithm becomes smaller by training k times with batch size = 1, rather than one-shot training with batch size = k, for any $k, n, \tilde{N}, m \geq 1$. Simply speaking, the computational cost of the training algorithm is minimized when batch size = 1, regardless of the model size. There are also a few favorable by-products from setting batch size = 1 summarized as follows;

1. **No More Matrix Inversion**
   The training algorithm has one matrix inversion (i.e., $(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T)^{-1}$). Since the matrix size of $(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T)$ is $k \times k$, the matrix inversion is replaced with a reciprocal operation $\frac{1}{\boldsymbol{I}+\boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T}$ when batch size = 1, which reduces much computational cost and even contributes to simplifying hardware implementation of ONLAD Core because implementing a matrix decomposition algorithm in hardware is much harder than simple operations like matrix products. Besides, hardware resources utilized for a matrix inversion module are saved.

2. **Low Cost Singularity Check**
   The OS-ELM training algorithm should be skipped if $(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T)$ is singular. One typical way to check if $(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T)$ is singular or not is calculating the determinant $\det(\boldsymbol{I}+\boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T)$ of which computational cost is approximately $O(k^3)$. When batch size = 1 the cost is minimized and just checking if $(\boldsymbol{I} + \boldsymbol{H}_i\boldsymbol{P}_{i-1}\boldsymbol{H}_i^T) > 0$ is enough.

3. **Minimizing Space Cost**
   As shown in Section 2.2.1.1 the space cost of the training algorithm is proportional to k^2. Obviously the cost is minimized when $k = 1$, which contributes to save a lot of hardware resources of ONLAD Core.

Based on the above analysis, batch size is always fixed to 1 in ONLAD and ONLAD Core. When $k = 1$, the OS-ELM training algorithm can be re-written as follows.

$$\begin{aligned} \boldsymbol{P}_i &= \boldsymbol{P}_{i-1} - \frac{\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T\boldsymbol{h}_i\boldsymbol{P}_{i-1}}{1 + \boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T} \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{h}_i^T(\boldsymbol{t}_i - \boldsymbol{h}_i\boldsymbol{\beta}_{i-1}), \end{aligned} \tag{2.15}$$

where $\boldsymbol{h} \in \mathbb{R}^{1\times\tilde{N}}$ is the special case of $\boldsymbol{H} \in \mathbb{R}^{k\times\tilde{N}}$ when $k = 1$.

2.2.2.1 On Side Effects of Fixing Batch Size to 1

In BP-NNs, the convergence of training error becomes unstable when batch size is too small, because training parameters are adapted to a set of few samples [30]. However, the training result of OS-ELM is not affected even when batch size = 1, because OS-ELM gives the same output weight when training is performed once with batch size = N or N times with batch size = 1. OS-ELM can fully enjoy the insight without any deterioration of training results, which is a notable difference from BP-NNs.

2.2.3 Light-Weight Forgetting Mechanism For OS-ELM

In a certain real environment, the distribution of normal data may change as time passes. In this case, ONLAD should have a functionality to adaptively forget past learned normal data with a tiny additional computational cost. To deal with this challenge, this section proposes a computationally light-weight forgetting mechanism based on FP-ELM (Forgetting Parameters Extreme Learning Machine) [31], one of the latest OS-ELM variants with a forgetting mechanism.

2.2.3.1 Review of FP-ELM

This section provides a brief review of FP-ELM. Given an initial training chunk $\{\boldsymbol{x}_0 \in \mathbb{R}^{k_0\times n}, \boldsymbol{t}_0 \in \mathbb{R}^{k_0\times m}\}$, FP-ELM finds $\boldsymbol{\beta}_0$ that minimizes

$$\mathrm{L}(\boldsymbol{\beta}_0) = \frac{\lambda}{2}\|\boldsymbol{\beta}_0\|_F^2 + \frac{1}{2}\|\boldsymbol{H}_0\boldsymbol{\beta}_0 - \boldsymbol{t}_0\|_F^2, \tag{2.16}$$

where λ is the L2 regularization parameter which contributes to avoid over-fitting. Note that $\|\cdot\|_F$ represents the frobenius norm given by $\|\boldsymbol{A}\|_F = \sqrt{\sum_i \sum_j |a_{ij}|^2}$. $\boldsymbol{\beta}_0$ can be solved by differentiating the above equation.

$$\begin{aligned} \frac{\mathrm{dL}(\boldsymbol{\beta}_0)}{\mathrm{d}\boldsymbol{\beta}_0} &= (\lambda\boldsymbol{I} + \boldsymbol{H}_0^T\boldsymbol{H}_0)\boldsymbol{\beta}_0 - \boldsymbol{H}_0^T\boldsymbol{t}_0 = \boldsymbol{O} \\ \boldsymbol{\beta}_0 &= (\lambda\boldsymbol{I} + \boldsymbol{H}_0^T\boldsymbol{H}_0)^{-1}\boldsymbol{H}_0^T\boldsymbol{t}_0 \end{aligned} \tag{2.17}$$

Then, given the next training chunk $\{\boldsymbol{x}_1 \in \mathbb{R}^{k_1 \times n}, \boldsymbol{t}_1 \in \mathbb{R}^{k_1 \times m}\}$, FP-ELM finds $\boldsymbol{\beta}_1$ minimizing

$$\mathrm{L}(\boldsymbol{\beta}_1) = \frac{\lambda}{2}\|\boldsymbol{\beta}_1\|_F^2 + \frac{1}{2}(\|\alpha_1(\boldsymbol{H}_0\boldsymbol{\beta}_1 - \boldsymbol{t}_0)\|_F^2 + \|\boldsymbol{H}_1\boldsymbol{\beta}_1 - \boldsymbol{t}_1\|_F^2), \tag{2.18}$$

where $0 < \alpha_1 \leq 0$ is a forgetting parameter that controls the weight (i.e., impact) of the initial training chunk $\{\boldsymbol{x}_0, \boldsymbol{t}_0\}$. $\boldsymbol{\beta}_1$ can be solved by differentiating the above equation in the same way as Equation 2.17.

$$\begin{aligned} \frac{\mathrm{dL}(\boldsymbol{\beta}_1)}{\mathrm{d}\boldsymbol{\beta}_1} &= (\lambda\boldsymbol{I} + \alpha_1^2\boldsymbol{H}_0^T\boldsymbol{H}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1)\boldsymbol{\beta}_1 - (\alpha_1^2\boldsymbol{H}_0^T\boldsymbol{t}_0 + \boldsymbol{H}_1^T\boldsymbol{t}_1) = \boldsymbol{O} \\ \boldsymbol{\beta}_1 &= (\lambda\boldsymbol{I} + \alpha_1^2\boldsymbol{H}_0^T\boldsymbol{H}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1)^{-1}(\alpha_1^2\boldsymbol{H}_0^T\boldsymbol{t}_0 + \boldsymbol{H}_1^T\boldsymbol{t}_1) \end{aligned} \tag{2.19}$$

Let $\boldsymbol{K}_0 \equiv \boldsymbol{H}_0^T\boldsymbol{H}_0$, the recurrence form of $\boldsymbol{\beta}_1$ is derived by applying $(\lambda\boldsymbol{I} + \boldsymbol{K}_0)\boldsymbol{\beta}_0 = \boldsymbol{H}_0\boldsymbol{t}_0$ to Equation 2.19.

$$\begin{aligned} \boldsymbol{\beta}_1 &= (\lambda\boldsymbol{I} + \alpha_1^2\boldsymbol{K}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1)^{-1}(\alpha_1^2(\lambda\boldsymbol{I} + \boldsymbol{K}_0)\boldsymbol{\beta}_0 + \boldsymbol{H}_1^T\boldsymbol{t}_1) \\ &= (\lambda\boldsymbol{I} + \alpha_1^2\boldsymbol{K}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1)^{-1}((\lambda\boldsymbol{I} + \alpha_1^2\boldsymbol{K}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1)\boldsymbol{\beta}_0 \\ &\quad - \lambda(1 - \alpha_1^2)\boldsymbol{\beta}_0 + \boldsymbol{H}_1^T\boldsymbol{t}_1 - \boldsymbol{H}_1^T\boldsymbol{H}_1\boldsymbol{\beta}_0) \\ &= \boldsymbol{\beta}_0 + (\lambda\boldsymbol{I} + \alpha_1^2\boldsymbol{K}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1)^{-1}(\boldsymbol{H}_1^T(\boldsymbol{t}_1 - \boldsymbol{H}_1\boldsymbol{\beta}_0) - \lambda(1 - \alpha_1^2)\boldsymbol{\beta}_0) \\ &= \boldsymbol{\beta}_0 + (\lambda\boldsymbol{I} + \boldsymbol{K}_1)^{-1}(\boldsymbol{H}_1^T(\boldsymbol{t}_1 - \boldsymbol{H}_1\boldsymbol{\beta}_0) - \lambda(1 - \alpha_1^2)\boldsymbol{\beta}_0), \end{aligned} \tag{2.20}$$

where $\boldsymbol{K}_1 = \alpha_1^2\boldsymbol{K}_0 + \boldsymbol{H}_1^T\boldsymbol{H}_1$.

Let's consider the recurrence form of general $\boldsymbol{\beta}_i$. Given the ith training chunk $\{\boldsymbol{x}_i \in \mathbb{R}^{k_i \times n}, \boldsymbol{t}_i \in \mathbb{R}^{k_i \times m}\}$, FP-ELM finds $\boldsymbol{\beta}_i$ that minimizes

$$\mathrm{L}(\boldsymbol{\beta}_i) = \frac{\lambda}{2}\|\boldsymbol{\beta}_i\|_F^2 + \frac{1}{2}\sum_{j=0}^{i}\|\prod_{k=j+1}^{i}\alpha_k(\boldsymbol{H}_j\boldsymbol{\beta}_i - \boldsymbol{t}_j)\|_F^2 \tag{2.21}$$

In the above equation the weight of the past $0 \leq j < i$-th training chunk ($\equiv w_j$) is given by

$$w_j = \begin{cases} \prod_{k=j+1}^{i}\alpha_k, & (0 \leq j < i) \\ 1. & (j = i) \end{cases} \tag{2.22}$$

We can derive the final training algorithm of FP-ELM by recursively calculating subsequent $\boldsymbol{\beta}_2, \boldsymbol{\beta}_3, \ldots, \boldsymbol{\beta}_i$ in the same way as Equation 2.19 ~ Equation 2.20.

$$\begin{aligned} \boldsymbol{K}_i &= \alpha_i^2\boldsymbol{K}_{i-1} + \boldsymbol{H}_i^T\boldsymbol{H}_i \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + (\lambda\boldsymbol{I} + \boldsymbol{K}_i)^{-1} \cdot (\boldsymbol{H}_i^T(\boldsymbol{t}_i - \boldsymbol{H}_i\boldsymbol{\beta}_{i-1}) - \lambda(1 - \alpha_i^2)\boldsymbol{\beta}_{i-1}) \end{aligned} \tag{2.23}$$

Please note that α_i is a dynamic parameter that can be adaptively changed. $\boldsymbol{K}_0$ and $\boldsymbol{\beta}_0$ are computed as follows.

$$\begin{aligned} \boldsymbol{K}_0 &= \boldsymbol{H}_0^T\boldsymbol{H}_0 \\ \boldsymbol{\beta}_0 &= (\lambda\boldsymbol{I} + \boldsymbol{H}_0^T\boldsymbol{H}_0)^{-1}\boldsymbol{H}_0^T\boldsymbol{t}_0, \end{aligned} \tag{2.24}$$

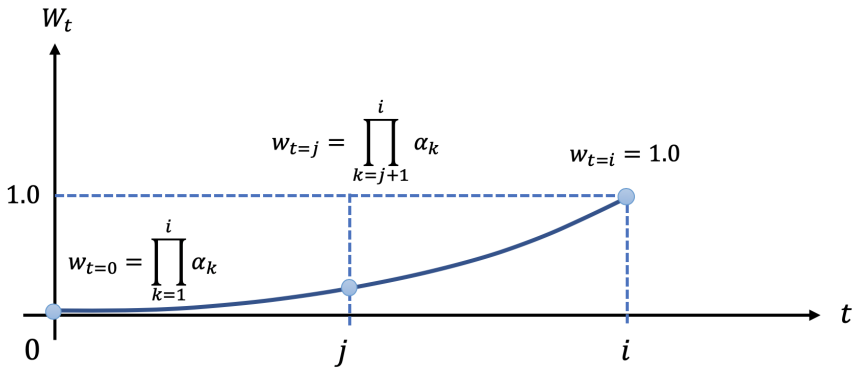

Figure 2.4: Forgetting Curve of FP-ELM

2.2.3.2 Light-Weight Forgetting Mechanism

FP-ELM is able to control the weight of past learned data. However, it cannot remove the matrix inversion $(\lambda \boldsymbol{I} + \boldsymbol{K}_i)^{-1}$ existing in Equation 2.23 even when batch size is 1, because the size of $(\lambda \boldsymbol{I} + \boldsymbol{K}_i)$ is $\tilde{N} \times \tilde{N}$, where $\tilde{N}$ is the number of hidden nodes. To address this issue, FP-ELM is modified so that it can remove the matrix inversion when batch size is 1.

The following equations are derived by disabling the L2 regularization (i.e., let $\lambda = 0$) in Equation 2.23.

$$\begin{aligned} \boldsymbol{K}_i &= \alpha_i^2 \boldsymbol{K}_{i-1} + \boldsymbol{H}_i^T \boldsymbol{H}_i \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{K}_i^{-1} \boldsymbol{H}_i^T (\boldsymbol{t}_i - \boldsymbol{H}_i \boldsymbol{\beta}_{i-1}) \end{aligned} \tag{2.25}$$

Then the recurrence formula of $\boldsymbol{K}_i^{-1}$ is derived by applying the Woodbury formula [23] [3].

$$\begin{aligned} \boldsymbol{K}_i^{-1} &= (\alpha_i^2 \boldsymbol{K}_{i-1} + \boldsymbol{H}_i^T \boldsymbol{H}_i)^{-1} \\ &= (\frac{1}{\alpha_i^2} \boldsymbol{K}_{i-1}^{-1}) - (\frac{1}{\alpha_i^2} \boldsymbol{K}_{i-1}^{-1}) \boldsymbol{H}_i^T \\ &\cdot (\boldsymbol{I} + \boldsymbol{H}_i (\frac{1}{\alpha_i^2} \boldsymbol{K}_{i-1}^{-1}) \boldsymbol{H}_i^T)^{-1} \boldsymbol{H}_i (\frac{1}{\alpha_i^2} \boldsymbol{K}_{i-1}^{-1}) \end{aligned} \tag{2.26}$$

The training algorithm is derived by replacing $\boldsymbol{K}_i^{-1}$ with $\boldsymbol{P}_i$.

$$\begin{aligned} \boldsymbol{P}_i &= (\frac{1}{\alpha_i^2} \boldsymbol{P}_{i-1}) - (\frac{1}{\alpha_i^2} \boldsymbol{P}_{i-1}) \boldsymbol{H}_i^T \\ &\cdot (\boldsymbol{I} + \boldsymbol{H}_i (\frac{1}{\alpha_i^2} \boldsymbol{P}_{i-1}) \boldsymbol{H}_i^T)^{-1} \boldsymbol{H}_i (\frac{1}{\alpha_i^2} \boldsymbol{P}_{i-1}) \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i \boldsymbol{H}_i^T (\boldsymbol{t}_i - \boldsymbol{H}_i \boldsymbol{\beta}_{i-1}) \end{aligned} \tag{2.27}$$

[3] $(\boldsymbol{A} + \boldsymbol{UCV})^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}(\boldsymbol{C}^{-1} + \boldsymbol{V}\boldsymbol{A}^{-1}\boldsymbol{U})^{-1}\boldsymbol{V}\boldsymbol{A}^{-1}$

$\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ are computed in the same algorithm as Equation 2.11. The proposed forgetting mechanism eliminates the matrix inversion when batch size = 1 because the size of $(\boldsymbol{I} + \boldsymbol{H}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{H}_i^T)$ is $k \times k$, where k is batch size. Note that Equation 2.27 is equal to the original training algorithm of OS-ELM when $\frac{1}{\alpha_i^2}\boldsymbol{P}_i$ is replaced with $\boldsymbol{P}_i$; the proposed algorithm provides a forgetting functionality with a tiny additional computational cost to the original training algorithm of OS-ELM. However, it may suffer from overfitting, since the L2 regularization is disabled. The trade-off is quantitatively evaluated in Section 2.3.

2.2.4 Algorithm of ONLAD

ONLAD leverages OS-ELM of batch size = 1 in conjunction with the proposed lightweight forgetting mechanism. The following equations are derived by combining Equations 2.15 and 2.27;

$$\begin{aligned}\boldsymbol{P}_i &= (\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1}) - \frac{(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T\boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})}{1 + \boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T} \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{h}_i^T(\boldsymbol{t}_i - \boldsymbol{h}_i\boldsymbol{\beta}_{i-1}),\end{aligned} \tag{2.28}$$

where $\boldsymbol{h}_i \in \mathbb{R}^{1\times\tilde{N}}$ is the special case of $\boldsymbol{H}_i \in \mathbb{R}^{k\times\tilde{N}}$ when $k = 1$. ONLAD is built on an OS-ELM-based autoencoder to construct a semi-supervised anomaly detector; $\boldsymbol{t}_i = \boldsymbol{x}_i$ holds in Equation 2.28. The training algorithm of ONLAD is as follows;

$$\begin{aligned}\boldsymbol{P}_i &= (\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1}) - \frac{(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T\boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})}{1 + \boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T} \\ \boldsymbol{\beta}_i &= \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{h}_i^T(\boldsymbol{x}_i - \boldsymbol{h}_i\boldsymbol{\beta}_{i-1}).\end{aligned} \tag{2.29}$$

$\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ are computed as follows;

$$\begin{aligned}\boldsymbol{P}_0 &= (\boldsymbol{H}_0^T\boldsymbol{H}_0)^{-1} \\ \boldsymbol{\beta}_0 &= \boldsymbol{P}_0\boldsymbol{H}_0^T\boldsymbol{x}_0.\end{aligned} \tag{2.30}$$

As indicated in Equation 2.29, ONLAD performs training and forgetting operations at the same time.

The prediction algorithm is given by

$$score = \mathrm{L}(\boldsymbol{x}, \mathrm{G}(\boldsymbol{x}\cdot\boldsymbol{\alpha} + \boldsymbol{b})\boldsymbol{\beta}), \tag{2.31}$$

where L represents the loss function and *score* is the anomaly score of $\boldsymbol{x} \in \mathbb{R}^{1\times n}$. *score* would take a high value if $\boldsymbol{x}$ differs from training data, while it will take a low value if $\boldsymbol{x}$ is similar to training data.

2.2.5 Example of Using ONLAD

Algorithm 1 shows an example of ONLAD intended for practical use. First, $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ are initialized with random values generated by any probability density functions. Then $\boldsymbol{\beta}_0$ and $\boldsymbol{P}_0$ are computed with Equation 2.30. Please note that the number of initial training samples k_0 should be larger than that of hidden nodes $\tilde{N}$ to make $\boldsymbol{H}_0^T\boldsymbol{H}_0$ non-singular. At the ith training step in the subsequent loop, the inequality $\epsilon > 1 + \boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T$, where ϵ is a small scalar, is evaluated. If the inequality is true the rest of processes are skipped because it means that $1 + \boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T$ is singular and can be numerically unstable, which should be avoided. If the inequality is false, the anomaly score of $\boldsymbol{x}_i$, is computed with Equation 2.31. $\boldsymbol{x}_i$ is judged to be an anomaly sample if the score is greater than a user-defined threshold θ; otherwise it is judged to be a normal sample. Finally, sequential learning is executed with Equation 2.29.

Algorithm 1 Example of Using ONLAD

$\boldsymbol{\alpha} \in \mathbb{R}^{n \times \tilde{N}} \leftarrow$ random()

$\boldsymbol{b} \in \mathbb{R}^{1 \times \tilde{N}} \leftarrow$ random()

$\boldsymbol{B} \in \mathbb{R}^{k_0 \times \tilde{N}} \leftarrow \begin{bmatrix} \boldsymbol{b} \\ \vdots \\ \boldsymbol{b} \end{bmatrix}$

$\boldsymbol{H}_0 \leftarrow \mathrm{G}(\boldsymbol{x}_0 \in \mathbb{R}^{k_0 \times n} \cdot \boldsymbol{\alpha} + \boldsymbol{B})$ ▷ k_0 should be much greater than $\tilde{N}$

$\boldsymbol{P}_0 \leftarrow (\boldsymbol{H}_0^T \boldsymbol{H}_0)^{-1}$

$\boldsymbol{\beta}_0 \leftarrow \boldsymbol{P}_0 \boldsymbol{H}_0^T \boldsymbol{x}_0$

$i \leftarrow 1$

while $\{\boldsymbol{x}_i \in \mathbb{R}^{1 \times n}, 0 < \alpha_i \leq 1\}$ exists **do**

    $\boldsymbol{h}_i \leftarrow \mathrm{G}(\boldsymbol{x}_i \cdot \boldsymbol{\alpha} + \boldsymbol{b})$

    if $\epsilon > 1 + \boldsymbol{h}_i(\frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1})\boldsymbol{h}_i^T$ **then**

        print("Singular matrix encountered")

    else

        $score \leftarrow \mathrm{L}(\boldsymbol{x}_i, \boldsymbol{h}_i \boldsymbol{\beta}_{i-1})$

        if $score > \theta$ **then**

            print("Anomaly Detected")

        end if

        $\boldsymbol{P}_{i-1} \leftarrow \frac{1}{\alpha_i^2}\boldsymbol{P}_{i-1}$

        $\boldsymbol{P}_i \leftarrow \boldsymbol{P}_{i-1} - \frac{\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T \boldsymbol{h}_i \boldsymbol{P}_{i-1}}{1 + \boldsymbol{h}_i \boldsymbol{P}_{i-1} \boldsymbol{h}_i^T}$

        $\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i \boldsymbol{h}_i^T (\boldsymbol{x}_i - \boldsymbol{h}_i \boldsymbol{\beta}_{i-1})$

    end if

    $i \leftarrow i + 1$

end while

Table 2.1: Datasets

Name	Samples	Features	Classes
Fashion MNIST [32]	70,000	784	10
MNIST [33]	70,000	784	10
Smartphone HAR [34]	5,744	561	6
Drive Diagnosis [35]	58,509	48	11
Letter Recognition [36]	20,000	16	26

2.3 Evaluations

In this section, anomaly detection performance of ONLAD is evaluated in comparison with other models. A common server machine (OS: Ubuntu 18.04, CPU: Intel Core i7 6700 3.4GHz, GPU: Nvidia GTX 1070 8GB, DRAM: DDR4 16GB, Storage: SSD 512GB) is used as the experimental environment in this section.

2.3.1 Experimental Setup

ONLAD is compared with the following three models: (1) **FPELM-AE**, (2) **NN-AE**, and (3) **DNN-AE**. FPELM-AE is an FP-ELM-based autoencoder, which is mathematically equal to ONLAD with L2 regularization enabled. The purpose of introducing this model is to quantitatively evaluate the side effect of disabling the L2 regularization in ONLAD. NN-AE is a 3-layer BP-NN-based autoencoder, while DNN-AE is a BP-NN-based deep autoencoder consisting of five layers. The purpose of introducing these models is to compare OS-ELM-based autoencoders (i.e., FPELM-AE and ONLAD) with BP-NN-based ones. All the models, including ONLAD, were implemented with TensorFlow v1.13.1 [37] in common for a fair comparison.

For a comprehensive evaluation, two testbeds: (1) **Offline Testbed** and (2) **Online Testbed** are conducted. Offline Testbed simulates a static environment where all training and test data are available in advance and no concept drift occurs. This is a standard experimental setup to evaluate a semi-supervised anomaly detection model [2]. The purpose of Offline Testbed is to measure the precision of ONLAD in the context of anomaly detection. In Offline Testbed, the forgetting mechanism of ONLAD and FPELM-AE is disabled by setting $\alpha_i = 1$, since no concept drift occurs in this testbed and the forgetting mechanism is unnecessary. On the other hand, Online Testbed simulates an environment where at first only a small part of the dataset is given and the rest arrives as time passes. Online Testbed assumes that concept drift occurs; in other words the feature of normal

Table 2.2: Search Ranges of Hyper-Parameters

	ONLAD	FPELM-AE
G_{hidden}	{Sigmoid [38], Identity}	{Sigmoid, Identity}
$p(x)$	Uniform [0,1]	Uniform [0,1]
L	MSE	MSE
α_i	{0.95, 0.96, ..., 1.00}	{0.95, 0.96, ..., 1.00}
$\tilde{N}_1$	{8, 16, 32, ..., 256}	{8, 16, 32, ..., 256}
λ		0.02
	NN-AE	DNN-AE
G_{hidden}	{Sigmoid, Relu [39]}	{Sigmoid, Relu}
G_{out}	Sigmoid	Sigmoid
L	MSE	MSE
O	Adam [40]	Adam
B	{8, 16, 32}	{8, 16, 32}
E	{5, 10, 15, 20}	{5, 10, 15, 20}
$\tilde{N}_1$	{8, 16, 32, ..., 256}	{8, 16, 32, ..., 256}
$\tilde{N}_2, \tilde{N}_3$		{8, 16, 32, ..., 256}

data changes in time series. The purpose of Online Testbed is to evaluate the robustness of the proposed forgetting mechanism against concept drift in comparison with the other models.

Several public datasets listed in Table 2.1 are used to construct Offline Testbed and Online Testbed. All data values are normalized within [0, 1] using min-max normalization. Hyper-parameters are explored within the ranges detailed in Table 2.2 [4].

[4]The definitions of the hyper-parameters are as follows.

- G_{hidden}: activation function applied to hidden layers.
- G_{out}: activation function applied to output layer. Only available for NN-AE and DNN-AE since ONLAD and FPELM-AE cannot put an activation function on output layer.
- $p(x)$: probability density function used for random initialization of ONLAD and FPELM-AE.
- $\tilde{N}_i$: number of the ith hidden nodes counting from input layer. For instance DNN-AE has $\tilde{N}_1$, $\tilde{N}_2$, $\tilde{N}_3$ since it consists of five layers (i.e., three hidden layers).
- L: loss function.
- α_i: forgetting factor of ONLAD and FPELM-AE.
- λ: L2 regularization parameter of FPELM-AE.
- O: optimization algorithm of NN-AE and DNN-AE.
- B: batch size of NN-AE and DNN-AE. This parameter is fixed to 1 in ONLAD and FPELM-AE.
- E: number of training epochs of NN-AE and DNN-AE.

Algorithm 2 Algorithm of Offline Testbed

$auc \leftarrow 0$
for $i \leftarrow 0, c-1$ **do**
  $\boldsymbol{X}_{normal\_train} \leftarrow \boldsymbol{X}^{(i)}_{train}$
  $\boldsymbol{X}_{normal\_test} \leftarrow \boldsymbol{X}^{(i)}_{test}$
  $\boldsymbol{X}_{anomaly\_test} \leftarrow \boldsymbol{X}^{(j\neq i)}_{test}$
  $model$.train($\boldsymbol{X}_{normal\_train}$)
  $num \leftarrow \frac{\text{length}(\boldsymbol{X}_{normal\_test})}{9.0}$ ▷ anomaly samples : normal samples = 1 : 9
  $\boldsymbol{X}'_{anomaly\_test} \leftarrow$ sample($\boldsymbol{X}_{anomaly\_test}, num$)
  $scores \leftarrow model$.predict(shuffle($\{\boldsymbol{X}_{normal\_test}, \boldsymbol{X}'_{anomaly\_test}\}$))
  $auc \leftarrow auc +$ calc_auc($scores$)
end for
$auc \leftarrow \frac{auc}{c}$

2.3.2 Experimental Procedure of Offline Testbed

Algorithm 2 shows the procedure of Offline Testbed. In Offline Testbed, a dataset is divided into training samples $\boldsymbol{X}_{train}$ (80%) and test samples $\boldsymbol{X}_{test}$ (20%), respectively. Suppose we have a dataset that consists of c classes in total; training samples of class i (i.e., $\boldsymbol{X}^{(i)}_{train}$) are used as normal data for training (i.e., $\boldsymbol{X}_{normal\_train}$) and test samples of class i (i.e., $\boldsymbol{X}^{(i)}_{test}$) are used as normal data for testing (i.e., $\boldsymbol{X}_{normal\_test}$). Test samples of class $j \neq i$ (i.e., $\boldsymbol{X}^{(j\neq i)}_{test}$) are used as anomaly data for testing (i.e., $\boldsymbol{X}_{anomaly\_test}$). Each model is trained on $\boldsymbol{X}_{normal\_train}$. NN-AE and DNN-AE are trained with batch size = B for E epochs, while ONLAD and FPELM-AE are trained with batch size = 1 for only one epoch. Once the training procedure ends, each model's AUC is calculated using a test dataset that mixes $\boldsymbol{X}_{normal\_test}$ and $\boldsymbol{X}_{anomaly\_test}$. A small set of $\boldsymbol{X}_{anomaly\_test}$ is randomly sampled so that the ratio of anomaly samples : normal samples = 1 : 9 to simulate a practical situation; anomalies are much rarer than normal ones in most cases.

The above procedure is repeated for $i \leftarrow 0 \ldots c-1$ then all the c AUC scores are averaged, and the averaged AUC is recorded as a result of a single trial. The final AUC scores reported in Table 2.3 are averages over 50 trials. 10-fold cross-validation is conducted for hyperparameter tuning.

Algorithm 3 Algorithm of Online Testbed

$indices \leftarrow [0, \ldots, c-1]$
shuffle($indices$)
$\boldsymbol{X}_{concept} \leftarrow []$
for $i \leftarrow 0, c-1$ **do**
    $num \leftarrow \frac{\text{length}(\boldsymbol{X}_{normal}^{(indices[i])})}{9.0}$ ▷ anomaly samples : normal samples = 1 : 9
    $\boldsymbol{X}'_{anomaly} \leftarrow \text{sample}(\boldsymbol{X}_{anomaly}^{(j \neq indices[i])}, num)$
    $\boldsymbol{X}_{concept}$.append(shuffle($\{\boldsymbol{X}_{normal}^{(indices[i])}, \boldsymbol{X}'_{anomaly}\}$))
end for
$model$.train($\boldsymbol{X}_{init}^{(indices[0])}$) ▷ Do initial training
$scores \leftarrow []$
for $i \leftarrow 0, c-1$ **do**
    for all $\boldsymbol{x} \in \boldsymbol{X}_{concept}[i]$ **do**
        $score \leftarrow model$.predict($\boldsymbol{x}$)
        $scores$.append($score$)
        $model$.train($\boldsymbol{x}$) ▷ Do sequential training
    end for
end for
$auc \leftarrow$ calc_auc($scores$)

2.3.3 Experimental Procedure of Online Testbed

Algorithm 3 shows the procedure of Online Testbed. In Online Testbed, a dataset is divided into initial data $\boldsymbol{X}_{init}$ (10%), test data $\boldsymbol{X}_{test}$ (45%), and validation data $\boldsymbol{X}_{valid}$ (45%). $\boldsymbol{X}_{init}$ represents data samples that exist in the beginning. $\boldsymbol{X}_{test}$ and $\boldsymbol{X}_{valid}$ represent data samples that sequentially arrive as time goes by. $\boldsymbol{X}_{test}$ is used to measure the final AUC scores, while $\boldsymbol{X}_{valid}$ is only for hyperparameter tuning. Both are further divided into normal samples $\boldsymbol{X}_{normal}$ (90%) and anomaly samples $\boldsymbol{X}_{anomaly}$ (10%). In the first step, a sequence (denoted as $indices$) consisting of integers $0 \ldots c-1$ is constructed and randomly shuffled. The shuffled sequence indicates the normal class of each concept; supposing that $indices = [2, 0, 1]$, the normal class transitions in the order $2 \rightarrow 0 \rightarrow 1$. The ith concept $\boldsymbol{X}_{concept}[i]$ mixes normal samples of class $indices[i]$ and anomaly samples of class $j \neq indices[i]$. The number of anomaly samples is limited so that anomaly samples : normal samples = 9 : 1, as with Offline Testbed.

For initial training, each model is trained with initial data of the first normal class (i.e., $\boldsymbol{X}_{init}^{(indices[0])}$). NN-AE and DNN-AE are trained with batch size = B for E training

epochs, while ONLAD and FPELM-AE are trained with batch size = 1 for only one epoch. Then the model computes anomaly scores for each data sample $\boldsymbol{x}$ continuously given from $\boldsymbol{X}_{concept}[0] \ldots \boldsymbol{X}_{concept}[c-1]$. Every time an anomaly score is computed, the model is sequentially trained with $\boldsymbol{x}$. After all the data samples are fed to the model, an AUC score is calculated. This AUC score is recorded as a result of a single trial; the final AUC scores reported in Table 2.4 are averages over 50 trials. Hyperparameter tuning is conducted with the same algorithm for 10 trials by replacing $\boldsymbol{X}_{test}$ with $\boldsymbol{X}_{valid}$ in Algorithm 3.

2.3.4 Experimental Results of Offline Testbed

Experimental results of Offline Testbed are shown in Table 2.3. Hyperparameter settings are listed in Table 2.5. Here, NN-AE and DNN-AE achieve slightly higher AUC scores than those of ONLAD by approximately 0.01∼0.03 point on almost all the datasets. This outcome implies that BP-NN-based autoencoders have slightly higher anomaly detection capability than OS-ELM-based ones in a static environment. However, NN-AE and DNN-AE have to be iteratively trained for a number of training epochs. As shown in Table 2.5, they need $5 \sim 20$ epochs to achieve their best performance, while ONLAD always finds the optimal solution in one epoch. Also, ONLAD achieves its best AUC scores with an equal or smaller model size compared to NN-AE and DNN-AE for all the datasets, which contributes to reducing computational cost and saving hardware resources required to implement ONLAD Core.

Note that there is only a slight difference between the scores of ONLAD and FPELM-AE ONLAD even surpasses or performs equal to FPELM-AE on three out of five datasets, which implies that ONLAD keeps anomaly detection accuracy even if the L2 regularization is disabled. Here follows a consideration of this counter-intuitive outcome; The L2 regularization prevents over-fitting of the solution $\boldsymbol{\beta}$. In the context of autoencoder, the

Table 2.3: AUC Scores of Offline Testbed

Dataset	ONLAD	FPELM-AE	NN-AE	DNN-AE
Fashion MNIST	0.905	0.905	**0.925**	0.913
MNIST	0.944	0.945	0.958	**0.961**
Smartphone HAR	**0.929**	0.928	0.922	0.910
Drive Diagnosis	0.939	0.943	0.952	**0.961**
Letter Recognition	0.952	0.950	0.978	**0.985**

L2 regularization helps the model reconstruct even unseen input data correctly, which is clearly a benefit for an autoencoder. However, the benefit can be harmful when using an autoencoder as a semi-supervised anomaly detector because even unseen input data may be reconstructed correctly if the L2 regularization is enabled. That may introduce false negatives. Taking the above consideration into account, the observation, there is only a slight difference between ONLAD and FPELM-AE, seems reasonable.

In summary, ONLAD has comparable anomaly detection accuracy to that of BP-NN-based models in much smaller training epochs with an equal or smaller model size. Also, anomaly detection accuracy of ONLAD is almost the same with FPELM-AE even though ONLAD disables the L2 regularization and its computational cost is lower than FPELM-AE.

2.3.5 Experimental Results of Online Testbed

Experimental results of Online Testbed are shown in Table 2.4. Hyperparameter settings are also listed in Table 2.6. Here, another model, named **ONLAD-NF** (ONLAD-No-Forgetting-mechanism) is introduced in order to examine the impact of the proposed forgetting mechanism. ONLAD-NF is a special case of ONLAD, where the forgetting mechanism is disabled by fixing α_i to 1. Hyperparameter settings of ONLAD-NF are the same as those of ONLAD except for α_i. As shown in the results, ONLAD-NF suffers from significantly lower AUC scores compared to ONLAD. The reason of this outcome is quite obvious; ONLAD-NF does not have any functionalities to forget past training data, therefore it gradually becomes more difficult to detect anomalies every time concept drift occurs. NN-AE and DNN-AE, on the other hand, achieved much higher AUC scores than ONLAD-NF because BP-NNs have the catastrophic forgetting nature [41] which works as a kind of forgetting mechanism. However, the BP-NN-based models do not have any numerical parameters to analytically control the progress of forgetting unlike ONLAD. For this reason, ONLAD stably achieves high AUC scores. ONLAD and

Table 2.4: AUC Scores of Online Testbed

Dataset	ONLAD-NF	ONLAD	FPELM-AE	NN-AE	DNN-AE
Fashion MNIST	0.575	**0.869**	0.866	0.685	0.697
MNIST	0.591	**0.899**	0.898	0.787	0.755
Smartphone HAR	0.558	0.781	0.788	0.785	**0.799**
Drive Diagnosis	0.552	0.786	0.849	0.744	**0.853**
Letter Recognition	0.548	**0.882**	0.879	0.737	0.788

FPELM-AE achieved similar AUC scores on most of the datasets as with the results of Offline Testbed. This outcome shows that the L2 regularization has only a slight impact on anomaly detection accuracy in a concept-drifting environment too.

In summary, ONLAD achieved much higher AUC scores than those of NN-AE and DNN-AE by approximately 0.10 $\sim$ 0.18 point on three of five public datasets. ONLAD also achieved comparable AUC scores to those of the BP-NN-based models on the other two datasets. The L2 regularization has only a slight impact on anomaly detection accuracy in both static and concept-drifting environments.

Table 2.5: Hyperparameter Settings on Offline Testbed

Dataset	ONLAD $\{G_{hidden}, p(x), \tilde{N}_1, L, \alpha_i\}$	FPELM-AE $\{G_{hidden}, p(x), \tilde{N}_1, L, \alpha_i, \lambda\}$
Fashion MNIST	{Identity, Uniform, 64, MSE, 1.00}	{Identity, Uniform, 64, MSE, 1,00, 0.02}
MNIST	{Identity, Uniform, 64, MSE, 1.00}	{Identity, Uniform, 64, MSE, 1.00, 0.02}
Smartphone HAR	{Identity, Uniform, 128, MSE, 1.00}	{Identity, Uniform, 128, MSE, 1.00, 0.02}
Drive Diagnosis	{Sigmoid, Uniform, 16, MSE, 1.00}	{Sigmoid, Uniform, 16, MSE, 1.00, 0.02}
Letter Recognition	{Sigmoid, Uniform, 8, MSE, 1.00}	{Sigmoid, Uniform, 8, MSE, 1.00, 0.02}
Dataset	NN-AE $\{G_{hidden}, G_{out}, \tilde{N}_1, L, O, B, E\}$	DNN-AE $\{G_{hidden}, G_{out}, \tilde{N}_1, \tilde{N}_2, \tilde{N}_3, L, O, B, E\}$
Fashion MNIST	{Relu, Sigmoid, 64, MSE, Adam, 32, 5}	{Relu, Sigmoid, 64, 32, 64, MSE, Adam, 8, 10}
MNIST	{Relu, Sigmoid, 64, MSE, Adam, 32, 5}	{Relu, Sigmoid, 64, 32, 64, MSE, Adam, 8, 10}
Smartphone HAR	{Relu, Sigmoid, 256, MSE, Adam, 8, 20}	{Relu, Sigmoid, 128, 256, 128, MSE, Adam, 8, 20}
Drive Diagnosis	{Relu, Sigmoid, 256, MSE, Adam, 8, 10}	{Relu, Sigmoid, 128, 256, 128, MSE, Adam, 8, 20}
Letter Recognition	{Relu, Sigmoid, 256, MSE, Adam, 8, 20}	{Relu, Sigmoid, 128, 256, 128, MSE, Adam, 8, 20}

Table 2.6: Hyperparameter Settings on Online Testbed

Dataset	ONLAD $\{G_{hidden}, p(x), \tilde{N}_1, L, \alpha_i\}$	FPELM-AE $\{G_{hidden}, p(x), \tilde{N}_1, L, \alpha_i, \lambda\}$
Fashion MNIST	{Sigmoid, Uniform, 64, MSE, 0.99}	{Sigmoid, Uniform, 64, MSE, 0.99, 0.02}
MNIST	{Sigmoid, Uniform, 64, MSE, 0.99}	{Sigmoid, Uniform, 64, MSE, 0.99, 0.02}
Smartphone HAR	{Identity, Uniform, 16, MSE, 0.97}	{Sigmoid, Uniform, 16, MSE, 0.97, 0.02}
Drive Diagnosis	{Sigmoid, Uniform, 16, MSE, 0.99}	{Sigmoid, Uniform, 16, MSE, 0.97, 0.02}
Letter Recognition	{Identity, Uniform, 8, MSE, 0.95}	{Identity, Uniform, 8, MSE, 0.95, 0.02}
Dataset	NN-AE $\{G_{hidden}, G_{out}, \tilde{N}_1, L, O, B, E\}$	DNN-AE $\{G_{hidden}, G_{out}, \tilde{N}_1, \tilde{N}_2, \tilde{N}_3, L, O, B, E\}$
Fashion MNIST	{Relu, Sigmoid, 64, MSE, Adam, 32, 5}	{Relu, Sigmoid, 64, 32, 64, MSE, Adam, 8, 10}
MNIST	{Relu, Sigmoid, 64, MSE, Adam, 32, 5}	{Relu, Sigmoid, 64, 32, 64, MSE, Adam, 8, 10}
Smartphone HAR	{Sigmoid, Sigmoid, 32, MSE, Adam, 8, 20}	{Sigmoid, Sigmoid, 32, 2, 32, MSE, Adam, 8, 20}
Drive Diagnosis	{Sigmoid, Sigmoid, 16, MSE, Adam, 8, 10}	{Sigmoid, Sigmoid, 16, 8, 16, MSE, Adam, 8, 20}
Letter Recognition	{Relu, Sigmoid, 16, MSE, Adam, 8, 20}	{Relu, Sigmoid, 16, 8, 16, MSE, Adam, 8, 20}

2.4 Summary

This chapter introduced ONLAD as the basis of the thesis. ONLAD realizes a light-weight semi-supervised anomaly detection with a fast sequential training functionality by constructing an autoencoder with OS-ELM. A cost analysis conducted in the chapter revealed that both of space and computational complexities of the training algorithm of OS-ELM are significantly reduced by just setting batch size = 1, which realizes a fast sequential learning functionality of ONLAD with a small memory size. Also this chapter proposed a computationally light-weight forgetting mechanism of OS-ELM based FP-ELM, on one of the latest OS-ELM variants with forgetting mechanism. It allows ONLAD to follow concept drift at a low computational cost.

Experimental results using public datasets showed that ONLAD has comparable anomaly detection accuracy to that of BP-NN-based models in much smaller training epochs with an equal or smaller model size. The experiments also showed that ONLAD keeps high anomaly detection accuracy even in a concept-drifting environment, outperforming BP-NN-based models by $0.10 \sim 0.18$ point in AUC on three out of five public datasets. ONLAD achieved comparable AUC scores to the BP-NN-based models on the rest of two datasets.

2.5 Future Work

BP-NNs are known to gain more representation capability by stacking multiple layers. Although the original OS-ELM algorithm is limited to have only one hidden layer, ML-OSELM (Multi-Layer Online Sequential Extreme Learning Machine) [42] proposed by Mirza *et al.* provides a multi-layer framework for OS-ELM. According to this work ML-OSELM outperforms OS-ELM on well-known open classification datasets by 0.15 $\sim$ 2.58 point in terms of test accuracy. Thus, anomaly detection capability of ONLAD may be further improved by leveraging the ML-OSELM framework. It would be worth attempting to work with the multi-layer version of ONLAD.

Chapter 3

Leveraging Multiple ONLAD Instances

OS-ELM, a core component of ONLAD, is a shallow three-layer neural network. It suffers from low anomaly detection performance when the distribution of normal data is complex or mixed of some sub-distributions. In the real world, there exist several systems that consist of multiple actions such as air conditioners, robot arms, gas turbines, and so on [2, 43]. These systems structure mixture distributions of normal data and ONLAD will suffer from a low performance due to its limited representation capability. To address the problem, this chapter proposes an ensemble approach leveraging multiple instances of ONLAD. The method shares a common idea with [44] where a set of training data is classified into multiple clusters and an instance is trained with one cluster to reduce complexity of the distribution of training data that each instance learns.

The rest of this chapter is organized as follows; Section 3.1 proposes the multi-instance method. The proposed method is evaluated in Section 3.2. Section 3.3 makes a brief summary of this chapter.

3.1 Method

The method consists of two phases: (1) **initial phase** and (2) **online phase**. Section 3.1.1 and Section 3.1.2 describes the initial and online phases, respectively.

Algorithm 4 Initial Phase

Require: $\boldsymbol{X} \in \mathbb{R}^{N \times n}$: A set of normal data. $c \in \mathbb{N}$: Number of clusters.

1: $\{\boldsymbol{X}^{(0)}, \ldots, \boldsymbol{X}^{(c-1)}\} \leftarrow$ clustering$(\boldsymbol{X}, c)$ ▷ Classify $\boldsymbol{X}$ into c sub-clusters
2: $\{onlad_0, \ldots, onlad_{c-1}\} \leftarrow$ create_instances(c) ▷ Create c ONLAD instances
3: **for** $i \leftarrow 0, c-1$ **do**
4: $\quad onlad_i$.initial_train$(\boldsymbol{X}^{(i)})$ ▷ Execute initial training
5: **end for**

Algorithm 5 Online Phase

Require: $\boldsymbol{x} \in \mathbb{R}^n$: An input vector.

1: **for** $i \leftarrow 0, c-1$ **do**
2: $\quad score_i \leftarrow onlad_i$.predict$(\boldsymbol{x})$ ▷ Compute anomaly scores
3: **end for**
4: $score_j \leftarrow \min(score_0, \ldots, score_{c-1})$ ▷ Find minimum anomaly score
5: **if** $score_j > \theta$ **then**
6: $\quad$ print("Anomaly Detected")
7: **else**
8: $\quad onlad_j$.online_train$(\boldsymbol{x})$ ▷ Execute online training
9: **end if**

3.1.1 Initial Phase

Initial phase does some pre-processes for the subsequent online phase. Algorithm 4 details the algorithm. Suppose a set of normal data $\boldsymbol{X} \in \mathbb{R}^{N \times n}$ with N samples of n-dimensional inputs is given. First $\boldsymbol{X}$ is classified into c sub-clusters using a clustering algorithm like K-Means [45]. Then create c ONLAD instances and train each instance using one cluster $\boldsymbol{X}^{(i)}$ for $0 \leq i \leq c-1$. This process groups together similar normal data into a cluster and reduces the complexity of normal data each instance is to learn, which makes detecting anomaly samples much easier for each instance.

3.1.2 Online Phase

Suppose an input $\boldsymbol{x} \in \mathbb{R}^n$ is given, calculate $score_i$, the anomaly score of the ith ONLAD instance, for $0 \leq i \leq c-1$. Then the following inequality is evaluated;

$$\min(score_0, score_1, \ldots, score_{c-1}) > \theta, \tag{3.1}$$

where θ is a user-defined threshold. If Equation 3.1 is true, $\boldsymbol{x}$ is judged to be an anomaly because it means that $\boldsymbol{x}$ is anomalous for all the ONLAD instances; in other words $\boldsymbol{x}$ is different from the data that any ONLAD instance has been trained on. If the inequality is false, $\boldsymbol{x}$ is judged to be a normal sample and is used as a training data. Let's say $score_j = \min(score_0, \ldots, score_{c-1})$, the jth ONLAD instance is trained on $\boldsymbol{x}$, since it means that $\boldsymbol{x}$ is most close to the data that the jth ONLAD instance has been trained on so far.

3.2 Evaluation

This section evaluates the proposed multi-instance approach. A common server machine (OS: Ubuntu 16.04, CPU: Intel Core i5 3470S 2.90 GHz, GPU: NVIDIA GTX 1080Ti 12 GB, DRAM: DDR4 16 GB) is used throughout the experiments.

The original single-instance version, denoted as **ONLAD**, is the baseline. It is compared to the proposed multi-instance version denoted as **ONLAD-Multi** in order to quantitatively evaluate the effectiveness of leveraging multiple instances. These two models are implemented in Numpy v1.16.0, and the numbers of input, hidden, and output nodes are configured to 784, 32, 784 in common. Identity function $G(\boldsymbol{x}) = \boldsymbol{x}$ is used as the activation function, and the mean absolute error $L(\boldsymbol{t}, \boldsymbol{y}) = \frac{1}{m}\sum_{i=0}^{m-1}|t_i - y_i|$ is used as the loss function.

Table 3.1: Datasets

Name	Training Samples	Test Samples	Features	Classes
Fashion MNIST [32]	60,000	10,000	784	10
MNIST [33]	60,000	10,000	784	10

3.2.1 Experimental Procedure

In this section Fashion MNIST [32] is used as normal data, and MNIST [33] is used as anomaly data. Both datasets consist of 60,000 training samples and 10,000 test samples, with the same number of input dimensions (= 784) and output classes (= 10). See Table 3.1 for details of the datasets. F-measure[1] is used as the evaluation metric.

In initial phase, (1) 5,000 training samples are randomly chosen from MNIST and classified into c clusters using K-Means [45]. Here this set of 5,000 samples is denoted as $\boldsymbol{X}_{train}$ and the ith output cluster is $\boldsymbol{X}^{(i)}_{train}$. (2) ONLAD is trained with $\boldsymbol{X}_{train}$. (3) The ith instance of ONLAD-Multi is trained with $\boldsymbol{X}^{(i)}_{train}$. In online phase, (3) 5,000 and 500 test

[1]F-measure f is a harmonic mean of "precision" p and "recall" r given by;

$$p = \frac{TP}{TP + FP}, r = \frac{TP}{TP + FN}, f = \frac{2pr}{p + r}. \tag{3.2}$$

TP (True Positives) is the count of anomaly samples correctly judged to be anomaly samples, while FP (False Positives) is that of normal samples wrongly judged to be anomaly samples. TN (True Negatives) is the count of normal samples correctly judged to be normal samples, while FN (False Negatives) is that of anomaly samples wrongly judged to be normal samples. F-measure is a meta score considering precision and recall at the same time similarly to AUC and is often used to determine the value of threshold. In most cases, a threshold that outputs the best f-measure score is chosen [2].

samples are randomly chosen from MNIST and Fashion MNIST. The former is denoted as $\boldsymbol{X}_{normal}$ and it represents normal samples. The latter is $\boldsymbol{X}_{anomaly}$, representing anomaly samples. (5) Let $\boldsymbol{X}_{test} \equiv \{\boldsymbol{X}_{normal}, \boldsymbol{X}_{anomaly}\}$, then compute anomaly scores of $\boldsymbol{X}_{test}$ and calculate an f-measure.

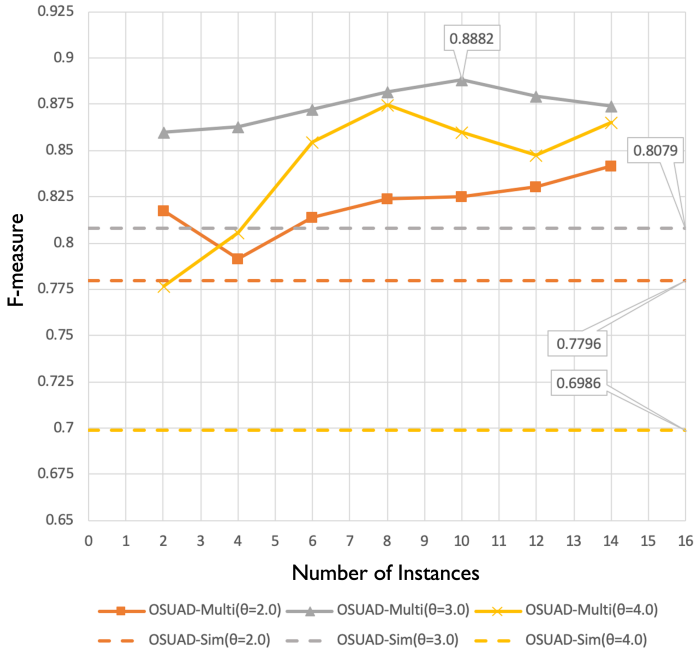

Figure 3.1: F-Measures with Varying Numbers of ONLAD Instances and Thresholds θ

3.2.2 Experimental Results

Figure 3.1 shows the f-measures of ONLAD and ONLAD-Multi with varying the number of instances of ONLAD-Multi and thresholds θ. Each score in the figure is the average of 50 repetitions of the experimental procedure described in the previous section.

The proposed ONLAD-Multi produced the best f-measure score at $\{\theta, c\} = \{3.0, 10\}$, surpassing the best score of ONLAD (at $\theta = 3.0$) by 8.03%. ONLAD-Multi is expected to make the best performance when the number of instances is close to the true number of sub-clusters of the dataset. Here the true number of sub-clusters (i.e., classes) of MNIST is 10, which is consistent with the result where ONLAD-Multi made the best score at $\{\theta, \underline{c}\} = \{3.0, \underline{10}\}$. Note that the smaller the number of instances c is from 10, the f-measure of ONLAD-Multi gradually declines because the complexity of data distribution that each instance is to learn increases. However, all the scores of ONLAD-Multi outperform ONLAD at the same threshold $\theta \in \{2.0, 3.0, 4.0\}$, meaning that the multi-instance approach improves anomaly detection performance of ONLAD in many cases.

In summary, it is strongly recommended that the number of instance c is set to the number equal to or close to the true number of sub-clusters c_{true} of the dataset if c_{true} is known in advance; otherwise estimating the optimal number of clusters utilizing X-Means [46], or choosing an enough large number rather than a small one is recommended.

3.3 Summary

OS-ELM, a core component of the ONLAD algorithm is a shallow three-layer neural network. It suffers from low anomaly detection performance if the distribution of normal data is complex or mixed of some sub-distributions due to limited representation capability of OS-ELM. To overcome this challenge, an ensemble approach leveraging multiple instances of ONLAD was proposed in this chapter. The proposed ensemble method reduces complexity of the distribution of data that each instance learns by clustering data with similar features. The proposed method outperformed the original single-instance ONLAD by 8.03% in f-measure under an anomaly detection task built on public datasets.

Chapter 4

ONLAD Core

This chapter introduces ONLAD Core, a hardware IP core implementing the ONLAD algorithm. Section 4.1 describes the module-level design and implementation of ONLAD Core and the FPGA-CPU co-architecture assuming a small FPGA evaluation board PYNQ-Z1. Section 4.2 evaluates ONLAD Core in terms of latency, energy, and FPGA resource utilization with an actual PYNQ-Z1 board. Experimental results show ONLAD Core can execute training and prediction computations faster and more energy-efficient compared to a CPU-only software implementation of ONLAD. Also the results show ONLAD Core can be implemented into even a smaller FPGA chip with proper tuning.

4.1 Design and Implementation

This section describes the design and implementation of ONLAD Core. As the evaluation platform, the PYNQ-Z1 board (Figure 4.1) is used throughout this chapter. PYNQ-

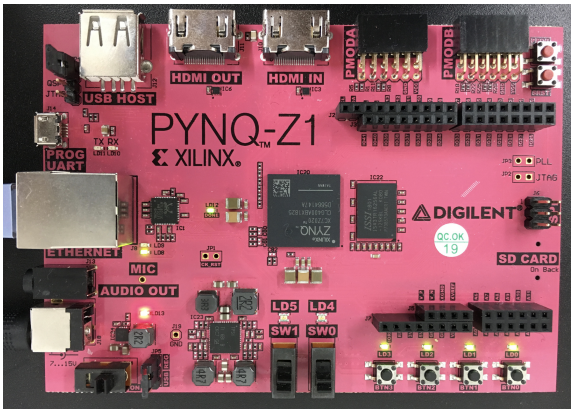

Figure 4.1: PYNQ-Z1 Evaluation Board

Table 4.1: Specifications of PYNQ-Z1 Evaluation Board.

Board Specifications	
Linux Image	Ubuntu v22.04
CPU	ARM Cortex-A9 dual-core 650 MHz
FPGA	See below
DRAM	DDR3 512 MB
FPGA Specifications	
BRAM (2.25 KB)	280 instances (630 KB)
DSP	220 slices
Flip-Flop	106,400 instances
LUT (6-inputs)	53,200 instances

Z1 is one of the most resource-limited FPGA-CPU SoC (System on Chip) boards currently available on the market, which will best fit to the target devices of ONLAD Core, resource-limited edge devices. Hardware specs are shown in Table 4.1. PYNQ-Z1 integrates a dual-core CPU running at 650 MHz and a low-end FPGA chip from Xilinx.

For development tools, Vitis HLS 2022.2, a high-level synthesis tool from Xilinx, is used to design ONLAD Core. Vivado 2022.2 is used to design and implement the FPGA-CPU co-architecture for ONLAD Core based on PYNQ-Z1. In the rest of this section, Section 4.1.1 clarifies the design policy of ONLAD Core then Section 4.1.2 gives module-to-module descriptions on ONLAD Core in detail. Finally, Section 4.1.3 describes the design of FPGA-CPU co-architecture based on PYNQ-Z1.

4.1.1 Design Policy

Here clarifies the design policy of ONLAD Core.

4.1.1.1 Latency vs. Throughput

In many cases, latency is more important than throughput for small edge devices because edge devices are rarely required to process thousands of queries per second. It is more reasonable to execute such compute-intensive tasks on cloud servers. In anomaly detection applications especially, response time can be one of the most important factors. Besides, the training algorithm of ONLAD, is not well suited for pipelining because there exist many data dependencies of variables so it requires a lot of data copies to realize pipelining, which may result in more resource utilization. This is a problem for a resource-limited edge device, the target platform of ONLAD Core. Considering the above

discussion, the design of ONLAD Core is more optimized towards a low latency and a small resource utilization rather than throughput.

4.1.1.2 Data Format

In ONLAD Core, fixed-point data formats are used for arithmetic units and all the arrays existing in the ONLAD algorithm. A fixed-point value is virtually an integer with a scale factor, so it can save a lot of cycles in arithmetic operations compared to float and double, resulting in a fast latency of ONLAD Core. Also the length of a fixed-point value is flexibly configurable so you can reduce FPGA resource utilization by reducing integer or fractional bits of a fixed-point value.

4.1.1.3 Resource Assignment Policy for Arrays

The ONLAD algorithm is built on a lot of matrices and vectors which consume a large portion of memory resources, so performance of ONLAD Core can be improved by assigning proper FPGA resources to implement them. The size of matrices and vectors of the ONLAD algorithm is one of the following four cases: (1) $\tilde{N} \times n$, (2) $\tilde{N} \times \tilde{N}$, (3) n, and (4) $\tilde{N}$. n and $\tilde{N}$ are the numbers of input and hidden nodes. Although there exist two more cases, $\tilde{N} \times m$ and m, these are equal to $\tilde{N} \times n$ and n because $n = m$ holds in ONLAD. The order of these sizes is always

$$(\tilde{N} \times n) \gg (\tilde{N} \times \tilde{N}) \gg n \gg \tilde{N}, \tag{4.1}$$

because $n \gg \tilde{N}$ in ONLAD (remember ONLAD is based on an autoencoder). n can be more than 10x larger than $\tilde{N}$ according to Section 2.3, the evaluations of the ONLAD algorithm (see Table 2.5 and Table 2.6 for details).

Taking the above property into account, the top-two largest $\tilde{N} \times n$ and $\tilde{N} \times \tilde{N}$ matrices (e.g, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{P}$) are implemented in BRAMs (Block RAMs) which are optimal for large arrays, while the bottom-two smallest $\tilde{N}$ and n-dimensional vectors (e.g, $\boldsymbol{b}$ and $\boldsymbol{x}$) are with LUT-based distributed RAMs which are suitable for small arrays in terms of latency and resource utilization.

4.1.2 Details of ONLAD Core

This section describes the module-level implementation of ONLAD Core in detail. Figure 4.2 illustrates the block diagram of ONLAD Core with four important sub-modules: (1) parameter buffer, (2) input buffer, (3) train module, and (4) predict module. Each sub-modules is explained in the rest of this section.

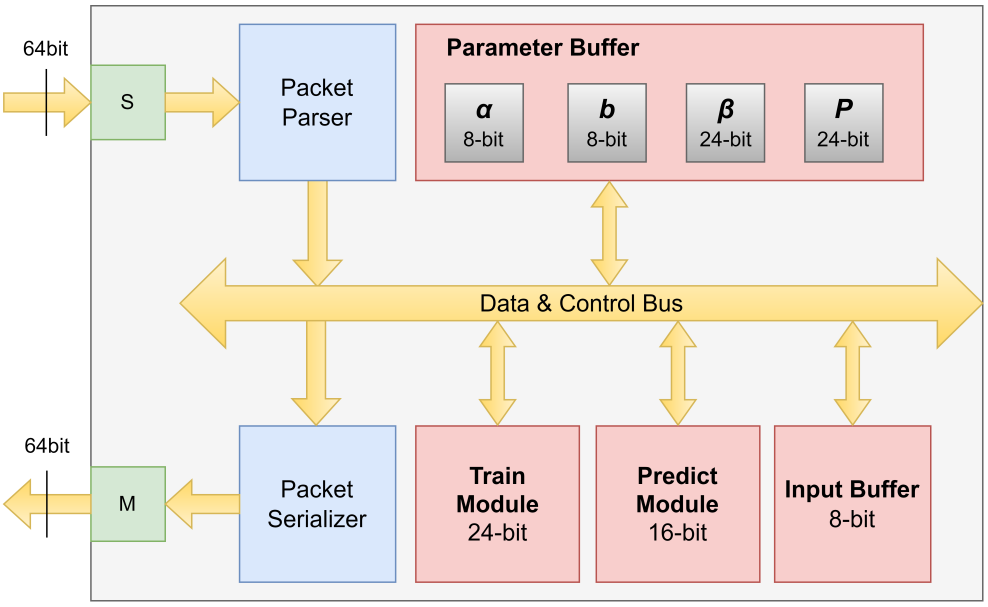

Figure 4.2: Block Diagram of ONLAD Core

4.1.2.1 Parameter Buffer and Input Buffer

The parameter buffer stores random parameters and training parameters of ONLAD. According to the resource assignment policy $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\boldsymbol{P}$ are implemented with BRAM blocks while $\boldsymbol{b}$ is with LUTs. The length of fixed-point values of $\boldsymbol{\alpha}$, $\boldsymbol{b}$, $\boldsymbol{\beta}$, and $\boldsymbol{P}$ are configured to 8 bits, 8 bits, 24 bits, and 24 bits, respectively. Random parameters $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ do not need so many bits, because their values are constant once randomly initialized. Since a constant can take only one value, there is no need to allocate extra integer bits to avoid potential overflows and underflows. It is also allowed to manually select parameters of random generation (i.e., value range and distribution) so that the generated values fit into the data format, since $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ accept any random values with arbitrary distribution [22]. On the other hand, $\boldsymbol{\beta}$ and $\boldsymbol{P}$ need much more number of bits because their value ranges will dynamically change as training proceeds.

The most resource-consuming buffer is one for $\boldsymbol{\beta}$. For example, as large as 71.6% of resources of parameter buffer is consumed when $\{n, \tilde{N}\} = \{1024, 64\}$. $\boldsymbol{\alpha}$, $\boldsymbol{P}$, and $\boldsymbol{b}$ consume 23.9%, 4.48%, and 0.02%, respectively.

Input buffer stores a single n-dimensional input vector $\boldsymbol{x} \in \mathbb{R}^n$ and is implemented with LUTs. Input buffer is shared with train module and predict module to read input data.

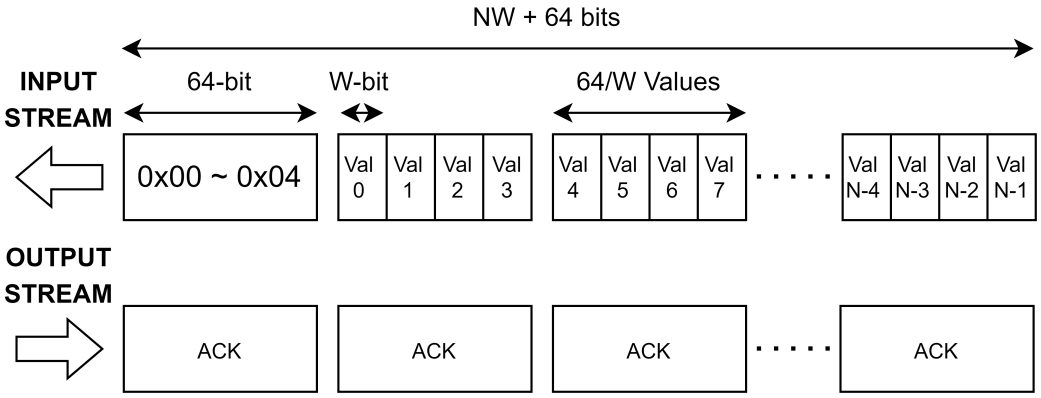

Figure 4.3: Input and Output Stream Packets for Writing Values of Parameter Buffer and Input Buffer

4.1.2.2 Writing Values of Parameter Buffer and Input Buffer

Figure 4.3 shows how to write values of parameter buffer and input buffer. The heading input packet determines which variable to write. 0x00, 0x01, 0x02, 0x03, and 0x04 correspond to $\boldsymbol{x}$, $\boldsymbol{b}$, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{P}$, respectively. Subsequent input packets store fixed-point write values of the target variable. Each write value expects a W-bit fixed-point data format. For example $W = 8$ when the variable is $\boldsymbol{x}$ (the value length of $\boldsymbol{x}$ is 8). Each input packet contains $\frac{64}{W}$ write values, thus $(NW + 64)$ bits in total are transferred to ONLAD Core where N is the number of total elements of the variable.

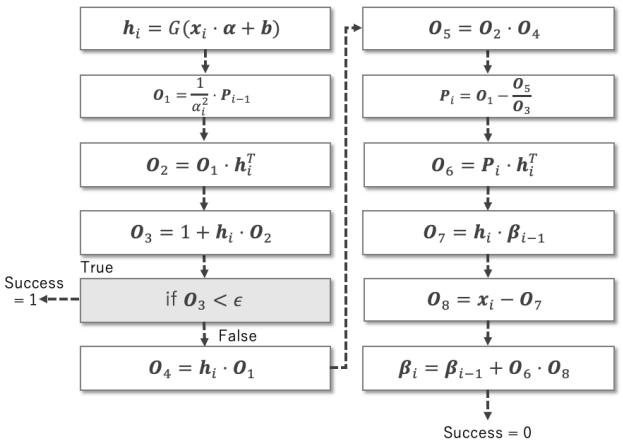

Figure 4.4: Computation Flow of Train Module

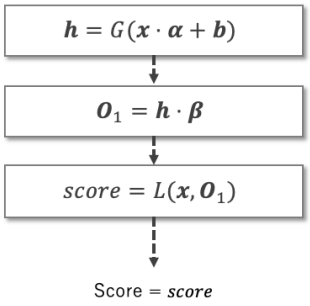

Figure 4.5: Processing Flow of Predict Module

4.1.2.3 Train Module and Predict Module

Train module executes the training algorithm of ONLAD in order to update training parameters stored in parameter buffer. Figure 4.4 shows the computation flow. Note that each process block is sequentially executed without overlapping; in other words it is not pipelined hence the input interval is equal to the latency. Train module interrupts the computation when $\boldsymbol{O}_3 < \epsilon$ holds, meaning a singular matrix is detected. In this work ϵ is set to $1e^{-5}$. The output signal Success indicates whether a training is skipped (= 1) or successfully executed (= 0). For resource assignment, only $\boldsymbol{O}_5$ is implemented with BRAM blocks and the others are all in LUTs in accordance with the resource assignment policy. Train module uses 24-bit fixed-point values throughout the module.

Predict module computes an anomaly score of the data stored in input buffer. Figure 4.5 is the processing flow of predict module. As with train module, predict module is not pipelined and each computation block is sequentially executed without overlaps. For resource assignment, all the arrays are implemented in LUTs. The length of fixed-point values of predict module is configured to 16 bits; predict module needs less number of bits than train module since predict module is a feed-forward computation graph and computation error will not be accumulated as training proceeds unlike train module.

4.1.2.4 Execution of Training and Prediction

Figure 4.6 shows input and output stream packets to execute training and prediction. Train module is triggered with a 64-bit heading packet with value 0x0a. The corresponding

output packet stores a success signal meaning whether the training is skipped (= 1) or successfully executed (= 0). Predict module is triggered with a 64-bit heading packet with value 0x0b then computes an anomaly score which will be stored in the output packet as a 64-bit fixed-point value.

Both train and predict modules expect for an input vector already stored in input buffer. Thus, to execute a single training or prediction, a series of $(128 + 8N)$ must be transferred into ONLAD Core, including number of bits to write an N-dimensional input vector.

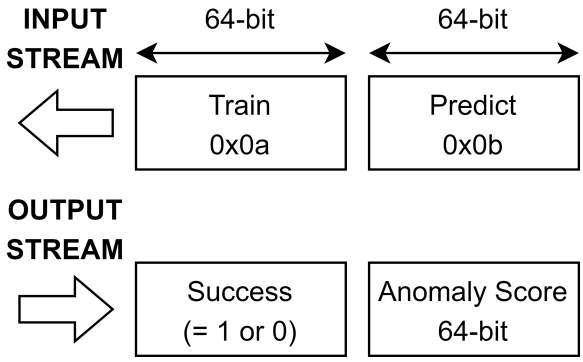

Figure 4.6: Input and Output Stream Packets for Triggering Train Module and Predict Module

4.1.2.5 Implementation of Matrix Operations

```
float sum = 0;
for (int i = 0; i < P; i++) {
    for (int j = 0; j < R; j++) {
        for (int k = 0; k < Q; k++) {
#pragma HLS pipeline
#pragma HLS unroll factor=N
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
        sum = 0;
    }
}
```

Figure 4.7: Example HLS Code of Matrix Product

Matrix operations of ONLAD Core are implemented as dedicated circuits using high-level synthesis. Figure 4.7 shows an example code of a matrix product $\boldsymbol{A} \in \mathbb{R}^{P \times Q} \cdot \boldsymbol{B} \in \mathbb{R}^{Q \times R} = \boldsymbol{C} \in \mathbb{R}^{P \times R}$. The unroll directive (*#pragma unroll factor=N*) unrolls the inner-most loop and executes N product-sum computations in parallel, resulting in up to $\frac{1}{N}$ latency. Note that the input arrays $\boldsymbol{A}$ and $\boldsymbol{B}$ must be separated into N RAM blocks with at least

one read/write port to realize parallel execution. Also parallel execution requires Nx more arithmetic units.

The pipeline directive (*#pragma pipeline*) improves throughput of the product-sum computations by overlapping the execution of operations from different loops. Each product sum is computed in one cycle thanks to the directive. Other matrix operations such as matrix add/sub are implemented in the same design methodology with matrix product too.

4.1.3 FPGA-CPU Co-Architecture Based on PYNQ-Z1

Figure 4.8 illustrates the design of FPGA-CPU co-architecture for ONLAD Core based on PYNQ-Z1. The processing part mainly consists of CPU and DRAM. Generation of random parameters $\{\boldsymbol{\alpha}, \boldsymbol{b}\}$ and initial training are executed in software. The FPGA part implements a DMA engine and ONLAD Core, running at 142.8 MHz. The DMA engine is responsible for data transfer between DRAM and ONLAD Core. It is controlled by software in the processing part via a 32-bit master GP port. A series of input data stored in DRAM is transferred to the DMA engine and there it is converted into axi4-stream packets with TWIDTH = 64. The input stream packets are fed to ONLAD Core then output packets of TWIDTH = 64 are sent back into DRAM via the DMA engine so that the developer can check output data from software.

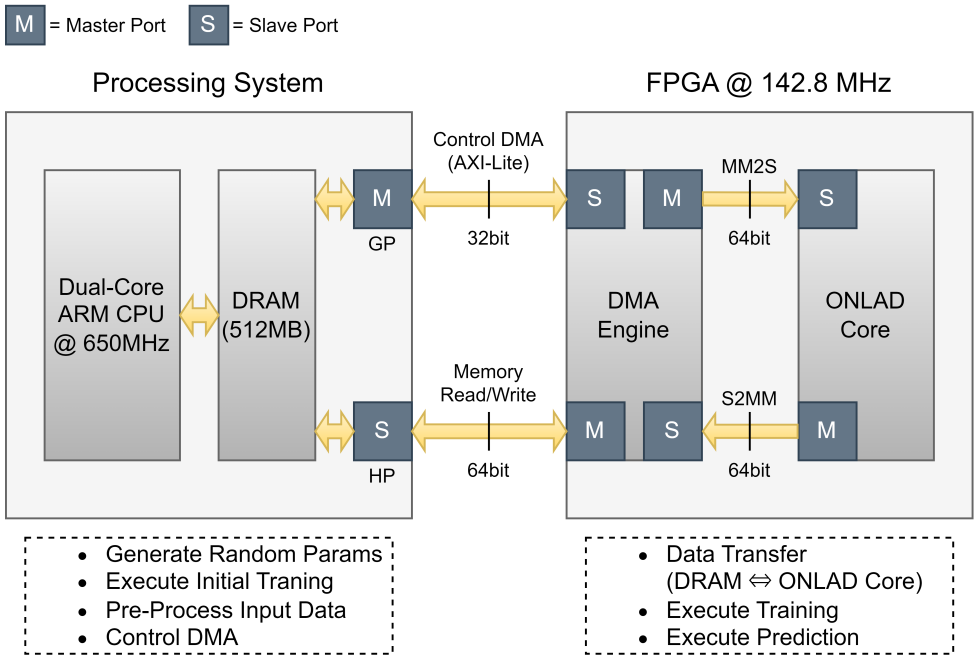

Figure 4.8: FPGA-CPU Co-Architecture for ONLAD Core Based on PYNQ-Z1

4.2 Evaluations

In this section, ONLAD Core is evaluated in terms of latency, energy and FPGA resource utilization in comparison with a cpu-implemented counterpart, called **ONLAD-CPU**. ONLAD Core offloads training and prediction computations to FPGA, but ONLAD-CPU execute all within the processing system. Both ONLAD Core and ONLAD-CPU are implemented in the same PYNQ-Z1 board for an apple-to-apple comparison. ONLAD-CPU is implemented with Numpy 1.24.1 source-compiled with NEON and OpenBLAS enabled to exploit multi-threading execution of the dual-core CPU. The maximum number of threads are fully utilized by setting OPENBLAS_NUM_THREADS = 2 for further acceleration. The training and prediction algorithms of ONLAD-CPU are implemented with 32-bit float. Unroll factor of ONLAD Core is configured to 8 for parallel execution of matrix operations.

As described in Section 4.1.2 ONLAD Core is implemented in fixed-point format instead of 32-bit float. Table 4.2 gives a quantitative evaluation of the impact of quantization by comparing ONLAD Core to ONLAD-CPU on the offline testbed introduced in Section 2.3. The model configurations (i.e., model size, activation function, loss function, etc) of ONLAD Core are the same with those of ONLAD-CPU. The AUC scores of ONLAD Core are 1.6 ~ 3.7 % (= 0.016 ~ 0.037 point) lower than ONLAD-CPU due to quantization effect, but ONLAD Core can cut 42.3 ~ 48.1 % of the total number of bits required for implementing the matrices and vectors of ONLAD-CPU.

Table 4.2: AUC Scores of Offline Testbed (ONLAD-CPU and ONLAD Core)

Dataset	ONLAD-CPU (float32)	ONLAD Core (fixed-point)	AUC Gap	Size Cut [%]
Fashion MNIST	0.905	0.868	-0.037	-48.1
MNIST	0.944	0.928	-0.016	-48.1
Smartphone HAR	0.929	0.901	-0.028	-45.4
Drive Diagnosis	0.939	0.917	-0.022	-44.0
Letter Recognition	0.952	0.920	-0.032	-42.3

4.2.1 Latency

Here ONLAD Core and ONLAD-CPU are compared in terms of “training latency” and “prediction latency”. A training latency refers to an elapsed time from receiving an input vector to the end of training. A prediction latency is an elapsed time from receiving an

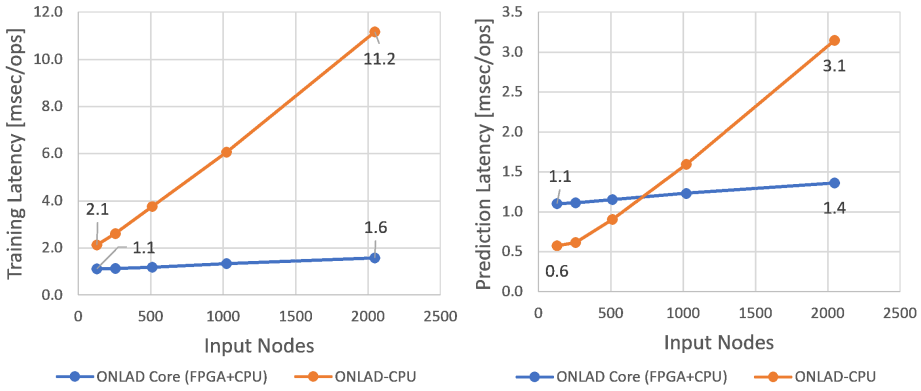

Figure 4.9: Comparison of Training Latency (Left) and Prediction Latency (Right)

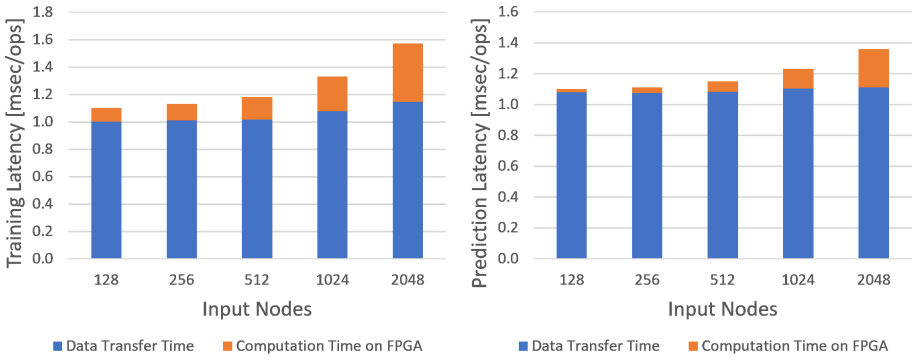

Figure 4.10: Breakdown of Training Latency (Left) and Prediction Latency (Right) of ONLAD Core

input vector to the end of calculation of an anomaly score. ONLAD Core includes time for input and output data transfers between DRAM and FPGA while ONLAD-CPU does not.

Figure 4.9 shows experimental results. The x-axis is the number of input nodes ranging {128, 256, 512, 1024, 2048}, while the y-axis represents training latency in msec unit. Each plot is an average time over 5,000 operations with the number of hidden nodes = 64. ONLAD Core is faster than ONLAD-CPU in terms of training latency by x1.9 ~ x7.1. Also ONLAD Core can make predictions faster by x1.3 ~ x2.3 at input nodes = {1024, 2048}, but is slower than ONLAD-CPU at small numbers of input nodes = {128, 256, 512}. Figure 4.10 reveals the cause. This figure shows time breakdowns of training and prediction latencies of ONLAD Core. For both training and prediction, the majority of execution time is taken up by data transfer time between DRAM and FPGA. The static overhead time is about 1 msec regardless of the number of input nodes, thus ONLAD

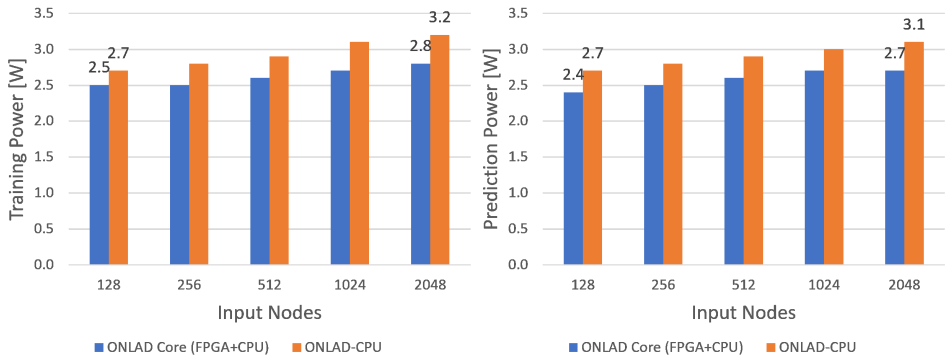

Figure 4.11: Power Consumption for Training (Left) and Prediction (Right)

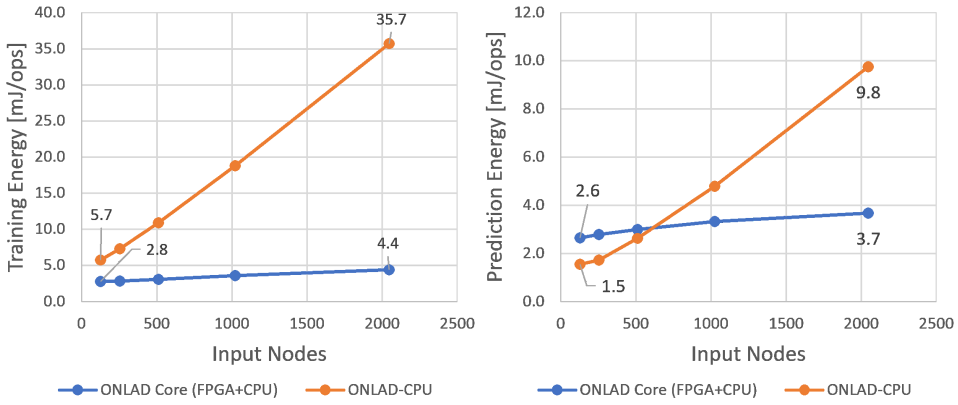

Figure 4.12: Energy Consumption for Training (Left) and Prediction (Right)

Core cannot be faster than ONLAD-CPU when ONLAD-CPU is faster than 1 msec. In summary, ONLAD Core can outperform ONLAD-CPU when the latency of ONLAD-CPU is slower than 1 msec in both training and prediction. To invest the cause of the large overhead time is one of the future works.

4.2.2 Energy and Power Consumption

Figure 4.11 shows runtime power consumptions of ONLAD Core and ONLAD-CPU when training or prediction is executed. Each plot is measured by a normal watt-hour meter of 0.1 W precision. It represents an overall power consumptions of PYNQ-Z1 board running ONLAD Core or ONLAD-CPU, so it includes power consumptions of peripherals other than CPU and FPGA. Even though ONLAD Core utilizes both CPU and FPGA, the overall power consumptions are lower than ONLAD-CPU by 0.2 W $\sim$ 0.4 W. ONLAD Core utilizes CPU just for data transfers while ONLAD-CPU executes intensive matrix computations with CPU with multi-threading enabled, resulting in a high CPU

power consumption overwhelming the sum of power consumptions of CPU and FPGA of ONLAD Core. Figure 4.12 shows energy consumptions of ONLAD Core and ONLAD-CPU. ONLAD Core is x2.1 ~ x8.1 more energy-efficient than ONLAD-CPU for training. Also ONLAD Core can execute predictions at x1.4 ~ x2.7 lower energy consumptions at input nodes = {1024, 2048}. However ONLAD Core suffers from sub-optimal efficiency at small numbers of input nodes because of the overhead time mentioned in the previous section.

In summary, power consumption of ONLAD Core is slightly lower than ONLAD-CPU by offloaing compute-intensive workloads to FPGA. ONLAD Core can execute training at x2.1 ~ x8.1 lower energy consumptions. ONLAD Core can also execute prediction at x1.4 ~ x2.7 lower energy consumption at large numbers of input nodes, but it suffers from higher energy consumptions when small numbers of input nodes due to a large overhead time of data transfers.

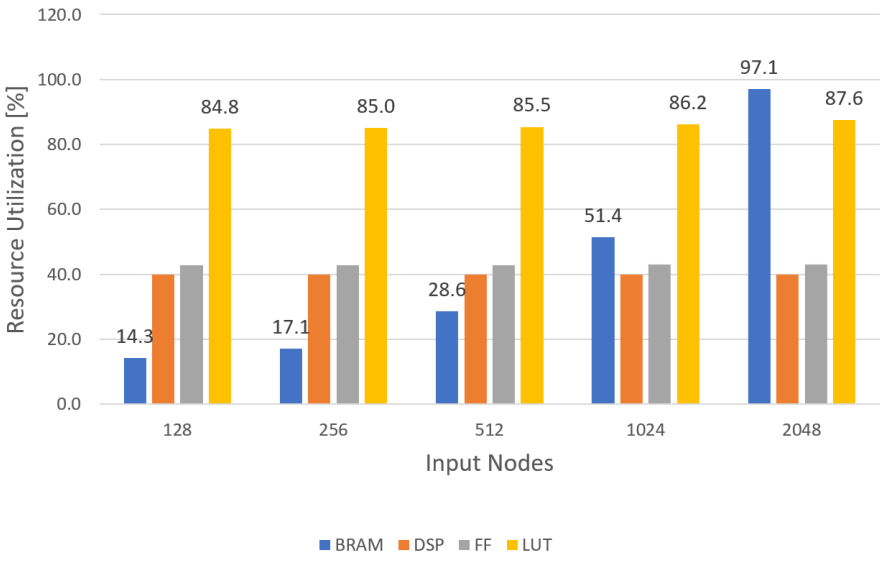

Figure 4.13: FPGA Resource Utilization of ONLAD Core

4.2.3 FPGA Resource Utilization

Figure 4.13 shows FPGA resource utilizations of ONLAD Core. Thanks to the algorithm-level space optimization of ONLAD and highly-tuned fixed-point data formats, ONLAD Core fits into a low-end small FPGA chip of PYNQ-Z1 up to number of input nodes = 2048, meaning ONLAD Core can handle up to 2048-dimensional data as input. Although ONLAD Core cannot handle color images with a few mega number of input dimensions, it

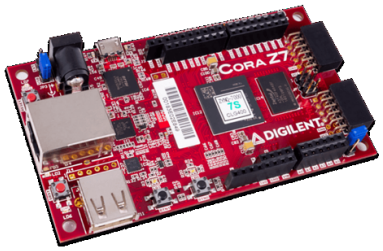

Figure 4.14: Cora Z7 Board

Table 4.3: Exploration of FPGA Resource Utilization.
n represents the number of input nodes and $\tilde{N}$ is the number of hidden nodes.

n \ $\tilde{N}$	8	16	32	64
128				
256				
512				
1024				
2048				

can be used sensing solutions with hundreds or thousands of input dimensions, including one featured in Chapter 6.

ONLAD Core can be implemented into even smaller FPGA chips by reducing unroll factor and the numbers of input and hidden nodes. Table 4.3 explores FPGA resource utilizations of ONLAD Core with unroll factor = 1, where a blue cell means that ONLAD Core can be implemented into **Cora Z7** board [47][1] (see Figure 4.14) at the combination of the input and hidden nodes, while a red cell means ONLAD Core does not fit into the chip. Core Z7 is a toy evaluation board with a tiny FPGA of which resource size is only 1/3 of PYNQ-Z1's FPGA. The table shows that ONLAD Core can be implemented into Cora Z7 except for when $\{n, \tilde{N}\}$ = {2048, 32}, {2048, 64}, {1024, 32}, {1024, 64}, and {512, 64}.

In summary, ONLAD Core with unroll factor = 8 can handle up to 2048-dimensional input data on PYNQ-Z1. ONLAD Core can be implemented into even smaller FPGA chips such as Cora Z1 by setting unroll factor = 1 and tuning the number of input and hidden nodes properly.

[1]Core Z7 has 100 BRAM blocks, 66 DSP slices, 28,800 flip-flops, and 14,400 6-input LUT instances. To the best of our knowledge, Core Z7 is one of the smallest FPGA boards available on the market.

4.3 Summary

This chapter proposed ONLAD Core, a hardware IP core implementing the ONLAD algorithm and described its design and implementation in detail. This chapter also introduced an FPGA-CPU co-architecture to utilize ONLAD Core for low-end small FPGA evaluation boards.

ONLAD Core was evaluated in terms of latency, energy, and FPGA resource utilization using an FPGA evaluation board PYNQ-Z1 which consists of a dual-core ARM CPU and a very small FPGA chip. Experimental results showed that ONLAD Core can execute the training algorithm x1.9 $\sim$ x7.1 faster and x2.1 $\sim$ x8.1 more energy-efficient compared to a CPU-only software implementation of ONLAD Core with full multi-threading enabled. It was also shown that ONLAD Core can be implemented into even a smaller FPGA board (Cora Z7) by tuning the configuration.

Chapter 5

Fixed-Point Data Format Optimization for OS-ELM Digital Circuits

OS-ELM has been one of the promising neural-network-based online algorithms for on-device learning because it can perform online training at a low computational cost and is easy to implement as a digital circuit [48, 49]. Several papers have proposed design methodologies and implementations of OS-ELM digital circuits and shown that OS-ELM can be implemented in a small FPGA chip and still be able to perform fast online training [48–51]. Existing OS-ELM digital circuits often employ the fixed-point data format and is manually tuned to meet resource and timing constraints. However, manual tuning may cause overflow or underflow which can lead to unexpected behaviors of the circuit. A lot of works have proposed data format optimization methods that analytically derive the lower and upper bounds of variables and automatically optimize the data format, ensuring that overflow and underflow never happen [52–54]. For on-device learning devices, an overflow/underflow-free design has a significant impact because online training is continuously executed and the intervals of variables will dynamically change in time-series.

This chapter proposes an overflow/underflow-free fixed-point data format optimization method for OS-ELM digital circuits. The proposed optimization method can be used for OS-ELM-based digital circuits including ONLAD Core. Contributions of this work are summarized as follows;

- This work proposes an interval analysis method for OS-ELM using affine arithmetic [55], one of the most widely-used interval arithmetic models. Affine arithmetic has been used in a lot of existing works for determining optimal integer bit-widths that never cause overflow and underflow.
- In affine arithmetic, division can be defined only if the denominator does not include

zero; otherwise the algorithm cannot be represented in affine arithmetic. OS-ELM's training algorithm contains one division; Section 5.2.2.2 analytically shows that the denominator does not include zero and proposes a simple mathematical trick to safely represent OS-ELM in affine arithmetic.

- Affine arithmetic can represent only fixed-length computation graphs and unbounded loops are not supported in affine arithmetic. However, OS-ELM's training algorithm is an iterative algorithm where current outputs are used as the next inputs endlessly. Section 5.2.2 propose an empirical solution for this problem based on simulation results, and verify its effectiveness in Section 5.4.3.

- The proposed method is evaluated using **OS-ELM Core**, an IP core that implements OS-ELM with fixed-point data format, in terms of occurrence rate of overflow/underflows and additional area cost to guarantee being overflow/underflow-free. OS-ELM Core is a slightly modified version of ONLAD Core introduced in Chapter 4.

This chapter is organized as follows; Section 5.1 provides preliminaries of this work. Section 5.2 proposes the overflow/underflow-free fixed-point format optimization method. Section 5.3 gives a brief introduction of OS-ELM Core. The proposed method is evaluated using OS-ELM Core in Section 5.4. Section 5.5 summarizes this chapter. See Table 5.1 for the notation rule of this chapter. Table 5.2 also shows the definitions of special variables that appear in the text.

Table 5.1: Notation Rules on Chapter 5

Notation	Description
x (*italic*)	Scaler.
$\hat{x}$	Affine form of x.
$\boldsymbol{x}$ (***bold italic***)	Vector or matrix.
$\hat{\boldsymbol{x}}$	Affine form of $\boldsymbol{x}$
$x_{[u,v]}$	uv element of $\boldsymbol{x}$.
$\hat{x}_{[u,v]}$	Affine form of the uv element of $\boldsymbol{x}$.

Table 5.2: Definitions of Special Variables Appearing in Chapter 5

Variable	Description
$n, \tilde{N}, m \in \mathbb{N}$	Number of input, hidden, or output nodes of OS-ELM.
$\boldsymbol{\alpha} \in \mathbb{R}^{n \times \tilde{N}}$	Constant random weight matrix connecting input and hidden layers of OS-ELM.
$\boldsymbol{\beta} \in \mathbb{R}^{\tilde{N} \times m}$	Trainable weight matrix connecting hidden and output layers of OS-ELM.
$\boldsymbol{P} \in \mathbb{R}^{\tilde{N} \times \tilde{N}}$	Trainable intermediate weight matrix of OS-ELM.
$\boldsymbol{b} \in \mathbb{R}^{1 \times \tilde{N}}$	Constant random bias vector of hidden layer.
G	Activation function applied to hidden layer output.
$\boldsymbol{x} \in \mathbb{R}^{1 \times n}$	Input vector.
$\boldsymbol{t} \in \mathbb{R}^{1 \times m}$	Target vector.
$\boldsymbol{y} \in \mathbb{R}^{1 \times m}$	Output vector.
$\boldsymbol{h} \in \mathbb{R}^{1 \times \tilde{N}}$	Output vector of hidden layer (after activation).
$\boldsymbol{e} \in \mathbb{R}^{1 \times \tilde{N}}$	Output vector of hidden layer (before activation).
$\boldsymbol{X} \in \mathbb{R}^{k \times n}$	Chunk of input vectors with batch size = k.
$\boldsymbol{T} \in \mathbb{R}^{k \times m}$	Chunk of target vectors with batch size = k.
$\boldsymbol{Y} \in \mathbb{R}^{k \times m}$	Chunk of output vectors with batch size = k.
$\boldsymbol{H} \in \mathbb{R}^{k \times \tilde{N}}$	Chunk of hidden layer output vectors with batch size = k (after activation).
$\boldsymbol{\gamma}^{(1)}, \ldots, \boldsymbol{\gamma}^{(10)}$	Intermediate variables existing in the training algorithm of OS-ELM.

5.1 Preliminaries

5.1.1 Interval Analysis

To realize an overflow/underflow-free fixed-point design, it is needed to know the interval of each variable and allocate a sufficient but minimum number of integer bits that never cause overflow and underflow. Existing interval analysis methods for fixed-point digital circuits are categorized into (1) dynamic methods or (2) static methods [56]. Dynamic methods [57–60] often take a simulation-based approach using tons of test inputs. It is known that dynamic methods often produce a better result close to the true interval compared to static methods, although they tend to take a long execution time due to exhaustive search and may encounter overflow or underflow if unseen inputs are found in runtime. Static methods [52–54, 61, 62], on the other hand, take a more analytical approach; they often involve solving equations and deriving the upper and lower bounds of each variable without simulation. Static methods produce a more conservative result (i.e., a wider interval estimation) compared to dynamic methods, although the result is analytically guaranteed. In this work a static method is employed for interval analysis since the goal is to realize an overflow/underflow-free fixed-point OS-ELM digital circuit with an analytical guarantee.

5.1.2 Interval Arithmetic

Interval arithmetic (IA) [63] is one of the oldest static interval analysis methods. In IA, a variable is represented in an interval $[x_1, x_2]$ where x_1 and x_2 are the lower and upper bounds of the variable. Basic operations $\{+, -, \times\}$ in IA are defined as follows;

$$\begin{aligned}
[x_1, x_2] + [y_1, y_2] &= [x_1 + y_1, x_2 + y_2], \\
[x_1, x_2] - [y_1, y_2] &= [x_1 - y_2, x_2 - y_1], \\
[x_1, x_2] \times [y_1, y_2] &= [\min(x_1y_1, x_1y_2, x_2y_1, x_2y_2), \max(x_1y_1, x_1y_2, x_2y_1, x_2y_2)].
\end{aligned} \tag{5.1}$$

IA bounds the intervals of intermediate variables as long as the intervals of input variables are given. However, IA suffers from the *dependency problem*; for example, the answer of a subtraction $x - x$ where $x \in [x_1, x_2]$ is 0 in ordinary algebra, although the result in IA is $[x_1 - x_2, x_2 - x_1]$, a much wider interval than the true range $[0, 0]$, which makes the intervals of subsequent variables get wider and wider. The cause of this problem is that IA ignores the correlation of variables; $x - x$ is treated as a self-subtraction in ordinary algebra, while it is regarded as a subtraction between independent intervals in IA.

5.1.3 Affine Arithmetic

Affine arithmetic (AA) [55] is a refinement of IA proposed by Stolfi *et. al*. AA keeps track of correlation of variables and is known to produce tighter bounds close to the true range compared to IA. AA has been applied into several fixed-point/floating-point bit-width optimization systems [52, 61, 64, 65] and still widely used in recent works [62, 66, 67]. This work uses AA as a core part of the proposed interval analysis method for OS-ELM.

In AA, the interval of a variable x is represented in an *affine form* $\hat{x}$ given by;

$$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \cdots + x_n\epsilon_n, \tag{5.2}$$

where $\epsilon_i \in [-1, 1]$. x_i is a coefficient and ϵ_i represents an *uncertainty variable* that takes $[-1, 1]$; an affine form is a linear combination of uncertainty variables.

The interval of $\hat{x}$ can be computed as below;

$$\begin{aligned} \text{interval}(\hat{x}) &= [\inf(\hat{x}), \sup(\hat{x})], \\ \inf(\hat{x}) &= x_0 - \sum_i |x_i|, \\ \sup(\hat{x}) &= x_0 + \sum_i |x_i|. \end{aligned} \tag{5.3}$$

$\inf(\hat{x})$ returns the lower bound of $\hat{x}$ and $\sup(\hat{x})$ is the upper bound. Conversely, a variable that ranges $[a, b]$ can be converted into an affine form $\hat{x} = x_0 + x_1\epsilon_1$ with

$$\begin{aligned} x_0 &= \frac{b+a}{2}, \\ x_1 &= \frac{b-a}{2}. \end{aligned} \tag{5.4}$$

5.1.3.1 Basic Operations of Affine Arithmetic

Addition/subtraction between affine forms $\hat{x}$ and $\hat{y}$ is simply defined as $\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_i (x_i \pm y_i)\epsilon_i$. However, multiplication $\hat{x} * \hat{y}$ is a little bit complicated;

$$\begin{aligned} \hat{x} * \hat{y} &= x_0 y_0 + \sum_i (x_0 y_i + y_0 x_i)\epsilon_i + Q, \\ Q &= \sum_i (x_i\epsilon_i) \sum_i (y_i\epsilon_i). \end{aligned} \tag{5.5}$$

Note that the quadratic term Q is not an affine form (i.e., Q is not a linear combination of ϵ_i) hence it needs approximation to convert it into an affine form. A conservative

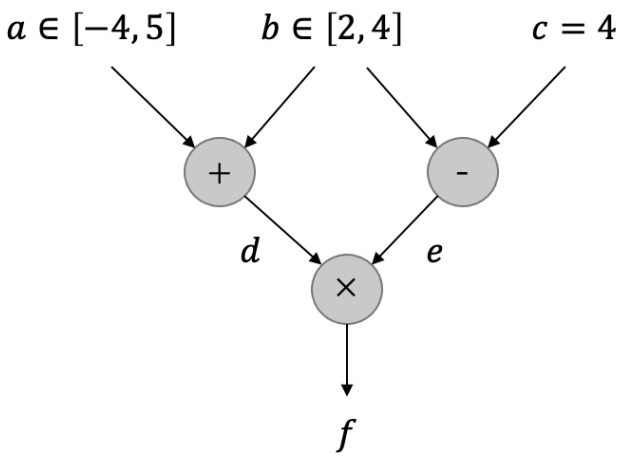

Figure 5.1: An Example of Affine Arithmetic

approximation shown below is often applied [52, 61, 62].

$$\begin{aligned} Q &\approx uv\epsilon_*, \\ u &= \sum_i |x_i|, \\ v &= \sum_i |y_i|, \end{aligned} \tag{5.6}$$

where $\epsilon_* \in [-1, 1]$ is an additional uncertainty variable. Note that $uv\epsilon_*$ is an upper bound of the quadratic term; in mathematically $uv\epsilon_* \geq \sum_i (x_i\epsilon_i) \sum_i (y_i\epsilon_i)$.

See Figure 5.1 for a simple tutorial of AA. In AA, all the intervals of input variables ($\{a, b, c\}$ in Figure 5.1) must be known. The affine forms of $a, \ldots, f$ are calculated as follows;

$$\begin{aligned} \hat{a} &= 0.5 + 4.5\epsilon_a, \\ \hat{b} &= 3.0 + \epsilon_b, \\ \hat{c} &= 4.0, \\ \hat{d} &= 3.5 + 4.5\epsilon_a + \epsilon_b, \\ \hat{e} &= -1.0 + \epsilon_b, \\ \hat{f} &= -3.5 - 4.5\epsilon_a + 2.5\epsilon_b + 5.5\epsilon_f. \end{aligned} \tag{5.7}$$

The intervals of intermediate variables d, e, and f are given by;

$$\begin{aligned} \text{interval}(\hat{d}) &= [-2.0, 9.0], \\ \text{interval}(\hat{e}) &= [-2.0, 0.0], \\ \text{interval}(\hat{f}) &= [-16.0, 9.0]. \end{aligned} \tag{5.8}$$

5.1.3.2 Division of Affine Arithmetic

Division $\hat{z} = \frac{\hat{x}}{\hat{y}}$ is often separated into $\hat{x} * \frac{1}{\hat{y}}$. There are mainly two approximation methods used to compute $\frac{1}{\hat{y}}$: (1) the min-max approximation and (2) the Chebyshev approximation [55]. Here we show the definition of $\frac{1}{\hat{y}}$ with the min-max approximation.

$$
\begin{aligned}
p &= \begin{cases} -\frac{1}{b^2} & (\text{if } b > a > 0) \\ -\frac{1}{a^2} & (\text{if } 0 > b > a), \end{cases} \\
q &= \frac{(a+b)^2}{2ab^2}, \\
d &= \frac{(a-b)^2}{2ab^2}, \\
\frac{1}{\hat{y}} &= (p \cdot y_0 + q) + \sum_i p \cdot (y_i \epsilon_i) + d\epsilon_*,
\end{aligned}
\tag{5.9}
$$

where $a = \inf(\hat{y})$ and $b = \sup(\hat{y})$. Note that $\frac{1}{\hat{y}}$ is defined only if $b > a > 0$ or $0 > b > a$; in other words $\frac{1}{\hat{y}}$ is undefined when the interval of the denominator y includes zero.

5.1.4 Determination of Integer Bit-Width

Suppose we have an affine form $\hat{x}$, the minimum number of integer bits that never cause overflow and underflow is given by;

$$
\begin{aligned}
IB &= \lceil \log_2(\max(|\inf(\hat{x})|, |\sup(\hat{x})|) + 1) + \alpha, \\
\alpha &= \begin{cases} 1 & (\text{if signed}) \\ 0 & \text{else.} \end{cases}
\end{aligned}
\tag{5.10}
$$

IB represents the optimal integer bit-width.

Algorithm 6 $\mathrm{T}(\boldsymbol{x}_i, \boldsymbol{t}_i, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{P}_{i-1}, \boldsymbol{\beta}_{i-1}) \mapsto \{\boldsymbol{P}_i, \boldsymbol{\beta}_i\}$ $(1 \leq i \leq N)$.

Require: $\boldsymbol{x}_i, \boldsymbol{t}_i, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{P}_{i-1}, \boldsymbol{\beta}_{i-1}$

Ensure: $\boldsymbol{h}_i = \mathrm{G}(\boldsymbol{x}_i \cdot \boldsymbol{\alpha} + \boldsymbol{b})$, $\boldsymbol{P}_i = \boldsymbol{P}_{i-1} - \frac{\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T\boldsymbol{h}_i\boldsymbol{P}_{i-1}}{1+\boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T}$, $\boldsymbol{\beta}_i = \boldsymbol{\beta}_{i-1} + \boldsymbol{P}_i\boldsymbol{h}_i^T(\boldsymbol{t}_i - \boldsymbol{h}_i\boldsymbol{\beta}_{i-1})$

1: $\boldsymbol{e}_i \leftarrow \boldsymbol{x}_i \cdot \boldsymbol{\alpha}$
2: $\boldsymbol{h}_i \leftarrow \mathrm{G}(\boldsymbol{e}_i + \boldsymbol{b})$
3: $\boldsymbol{\gamma}_i^{(1)} \leftarrow \boldsymbol{P}_{i-1} \cdot \boldsymbol{h}_i^T$
4: $\boldsymbol{\gamma}_i^{(2)} \leftarrow \boldsymbol{h}_i \cdot \boldsymbol{P}_{i-1}$
5: $\boldsymbol{\gamma}_i^{(3)} \leftarrow \boldsymbol{\gamma}_i^{(1)} \cdot \boldsymbol{\gamma}_i^{(3)}$
6: $\gamma_i^{(4)} \leftarrow \boldsymbol{\gamma}_i^{(2)} \cdot \boldsymbol{h}_i^T$
7: $\gamma_i^{(5)} \leftarrow \gamma_i^{(4)} + 1$
8: $\boldsymbol{\gamma}_i^{(6)} \leftarrow \frac{\boldsymbol{\gamma}_i^{(3)}}{\gamma_i^{(5)}}$
9: $\boldsymbol{P}_i \leftarrow \boldsymbol{P}_i - \boldsymbol{\gamma}_i^{(6)}$
10: $\boldsymbol{\gamma}_i^{(7)} \leftarrow \boldsymbol{P}_i \cdot \boldsymbol{h}_i^T$
11: $\boldsymbol{\gamma}_i^{(8)} \leftarrow \boldsymbol{h}_i \cdot \boldsymbol{\beta}_{i-1}$
12: $\boldsymbol{\gamma}_i^{(9)} \leftarrow \boldsymbol{t}_i - \boldsymbol{\gamma}_i^{(8)}$
13: $\boldsymbol{\gamma}_i^{(10)} \leftarrow \boldsymbol{\gamma}_i^{(7)} \cdot \boldsymbol{\gamma}_i^{(9)}$
14: $\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}_{i-1} + \boldsymbol{\gamma}_i^{(10)}$ **return** $\{\boldsymbol{P}_i, \boldsymbol{\beta}_i\}$

Algorithm 7 $\mathrm{P}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{\beta}) \mapsto \boldsymbol{y}$

Require: $\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{\beta}$

Ensure: $\boldsymbol{y} = \mathrm{G}(\boldsymbol{x} \cdot \boldsymbol{\alpha} + \boldsymbol{b})\boldsymbol{\beta}$

1: $\boldsymbol{e} \leftarrow \boldsymbol{x} \cdot \boldsymbol{\alpha}$
2: $\boldsymbol{h} \leftarrow \boldsymbol{e} + \boldsymbol{b}$
3: $\boldsymbol{y} \leftarrow \boldsymbol{h} \cdot \boldsymbol{\beta}$ **return** $\boldsymbol{y}$

5.2 Method

In this section an AA-based interval analysis method for OS-ELM is proposed. The process is two-fold: (1) Build the computation graph equivalent to OS-ELM. (2) Compute the affine form and get interval for every variable existing in OS-ELM, using Equation 5.3. Figure 5.2 shows computation graphs of OS-ELM. The "training graph" corresponds to the online training algorithm (Equation 2.15), while the "prediction graph" corresponds to the prediction algorithm (Equation 2.1).

$\mathrm{T}(\boldsymbol{x}_i, \boldsymbol{t}_i, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{P}_{i-1}, \boldsymbol{\beta}_{i-1}) \mapsto \{\boldsymbol{P}_i, \boldsymbol{\beta}_i\}$ defined in Algorithm 6 represents a sub-graph that computes a single iteration of the online training algorithm. Training graph refers to the whole graph concatenating N sub-graphs, where N is the total count of train-

ing steps. Training graph takes $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_N, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{P}_0, \boldsymbol{\beta}_0\}$ as input and outputs $\{\boldsymbol{P}_N, \boldsymbol{\beta}_N\}$. $\mathrm{P}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{\beta}) \mapsto \boldsymbol{y}$ defined in Algorithm 7 represents prediction graph. Prediction graph takes $\{\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{\beta}\}$ as input and maps it to $\boldsymbol{y}$. The goal is to obtain the intervals of $\{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ $(1 \leq i \leq N)$ for training graph and $\{\boldsymbol{e}, \boldsymbol{h}, \boldsymbol{y}\}$ for prediction graph, through interval analysis.

In this work, the interval of a matrix $\boldsymbol{A} \in \mathbb{R}^{u \times v}$ is given by;

$$\begin{aligned} \mathrm{interval}(\hat{\boldsymbol{A}}) &= [\inf(\hat{\boldsymbol{A}}), \sup(\hat{\boldsymbol{A}})], \\ \inf(\hat{\boldsymbol{A}}) &= \min(\inf(\hat{A}_{[0,0]}), \ldots, \inf(\hat{A}_{[u-1,v-1]})), \\ \sup(\hat{\boldsymbol{A}}) &= \max(\sup(\hat{A}_{[0,0]}), \ldots, \sup(\hat{A}_{[u-1,v-1]})), \end{aligned} \tag{5.11}$$

where $\hat{\boldsymbol{A}}$ is the affine form of $\boldsymbol{A}$, while $\hat{A}_{[i,j]}$ is the ij element of $\hat{\boldsymbol{A}}$.

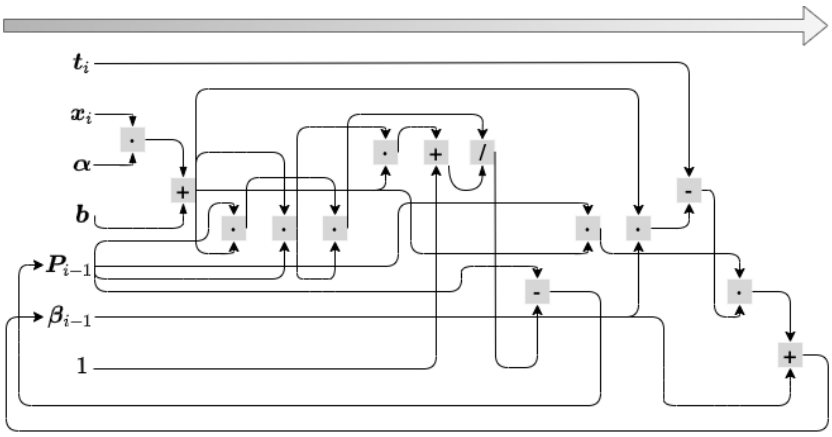

Figure 5.2: Computation graphs for OS-ELM

5.2.1 Constraints

Remember that all input intervals must be known in AA; in other words the intervals of $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_N, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{P}_0, \boldsymbol{\beta}_0\}$ of training graph and $\{\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{\beta}\}$ of prediction graph, must be known in advance. In this work the intervals of input and target data (i.e., $\{\boldsymbol{x}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_N\}$) are assumed to be all $[0, 1]$, and those of random parameters $\{\boldsymbol{\alpha}, \boldsymbol{b}\}$ are $[-1, 1]$. The initial training parameters $\{\boldsymbol{P}_0, \boldsymbol{\beta}_0\}$ are given as constants by Equation 2.11. The interval of $\boldsymbol{\beta}$ (an input of prediction graph) is given in the way described later in Section 5.2.3.

5.2.2 Interval Analysis for Training Graph

The aim of training graph is to find the intervals of $\{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ for $1 \leq i \leq N$. However, we have to deal with a critical problem, determination of the total number of training steps N. If the entire set of training dataset is given in advance and assuming any further training computations do not happen, just simply setting $N = \{\text{number of total training samples}\}$ is optimal. On the other hand, the scope of this work is on-chip learning assuming that further online trainings can happen in runtime; in other words N is not bounded and can increase in runtime. When N is unknown, training graph grows endlessly and interval analysis becomes infeasible. We need to determine a "reasonable" value of N for training graph.

Table 5.3: Classification Datasets

Name	Initial training samples	Online training samples	Test samples	Features	Classes	Model size
Digits [68]	358	1,079	360	64	10	{64, 48, 10}
Iris [69]	30	90	30	4	3	{4, 5, 3}
Letter [36]	4,000	12,000	4,000	16	26	{16, 32, 26}
Credit [70]	6,000	18,000	6,000	23	2	{23, 16, 2}
Drive [35]	11,701	35,106	11,702	48	11	{48, 64, 11}

5.2.2.1 Determination of N

To determine N, this section conducts an experiment to analyze the intervals of $\{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ for $1 \leq i \leq N$. The experimental procedure is as follows;

1. Implement OS-ELM's initial and online training algorithms in double-precision format.

2. Compute the initial training algorithm using initial training samples of Digits [68] dataset (see Table 5.3 [1] for details) then initial training parameters $\{\boldsymbol{P}_0, \boldsymbol{\beta}_0\}$ are obtained.

3. Compute the online training algorithm *by one step* using an online training sample. $\{\boldsymbol{P}_j, \boldsymbol{\beta}_j\}$ is obtained at the jth training step.

[1] "Initial training samples" are used to get $\{\boldsymbol{\beta}_0, \boldsymbol{P}_0\}$ with the initial training algorithm. "Online training samples" are used to get $\{\boldsymbol{\beta}_i, \boldsymbol{P}_i\}$ at the ith training step using the online training algorithm. "Test samples" are used to evaluate test accuracy. "Model size" is the model size $\{n, \tilde{N}, m\}$ for each dataset, where n, $\tilde{N}$, and m are the numbers of input, hidden, and output nodes.

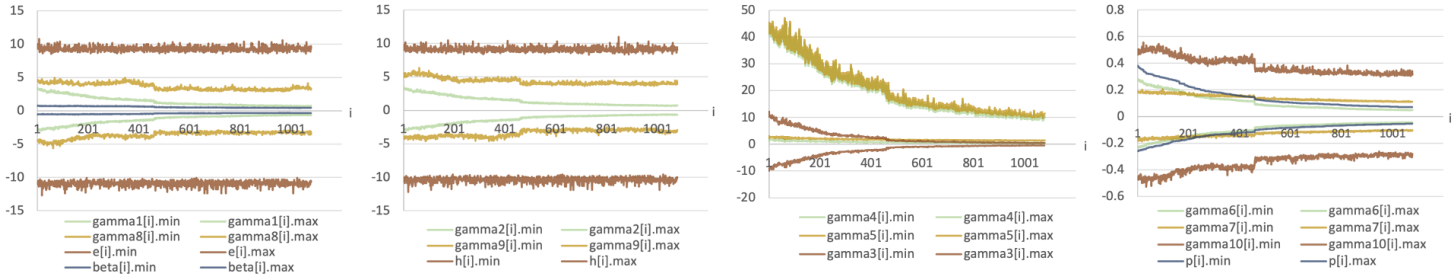

Figure 5.3: Observed Intervals of $\{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ $(1 \leq i \leq N = 1{,}079)$ on Digits. The x-axis represents the training step i, and the y-axis plots the observed intervals (the maximum and minimum values) of each variable at training step i.

4. Generate 1,000 random training samples $\{\boldsymbol{x}, \boldsymbol{t}\}$ with uniform distribution within [0, 1]. Feed all the random samples into the online training algorithm of step = j and measure the maximum and minimum values for each of $\{\boldsymbol{\gamma}_j^{(1)}, \ldots, \boldsymbol{\gamma}_j^{(10)}, \boldsymbol{P}_j, \boldsymbol{\beta}_j, \boldsymbol{e}_j, \boldsymbol{h}_j\}$.

5. Iterate 3-4 until all the online training samples are exhausted.

The experimental results are shown in Figure 5.3. It was observed that all the intervals gradually converges or keeps constant as the training step i proceeds. Similar outcomes were observed on the other datasets too (see Section 5.4.3 for complete results). Taking these outcomes into account, here makes a hypothesis that $\forall \boldsymbol{A}_i \in \{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ roughly satisfies interval$(\boldsymbol{A}_1) \supseteq$ interval$(\boldsymbol{A}_i)$ for $2 \leq i$, meaning the interval of $\boldsymbol{A}_1$ can also be used as upper and lower bounds of $\boldsymbol{A}_2, \ldots, \boldsymbol{A}_N$ too. The hypothesis is verified in Section 5.4.3 using multiple datasets.

Based on the hypothesis, N is set to 1 in training graph. The proposed interval analysis method for training graph is summarized as follows;

1. Build training graph $\mathrm{T}(\boldsymbol{x}_0, \boldsymbol{t}_0, \boldsymbol{\alpha}, \boldsymbol{b}, \boldsymbol{P}_0, \boldsymbol{\beta}_0) \mapsto \{\boldsymbol{P}_1, \boldsymbol{\beta}_1\}$.

2. Compute $\{\hat{\boldsymbol{\gamma}}_1^{(1)}, \ldots, \hat{\boldsymbol{\gamma}}_1^{(10)}, \hat{\boldsymbol{P}}_1, \hat{\boldsymbol{\beta}}_1, \hat{\boldsymbol{e}}_1, \hat{\boldsymbol{h}}_1\}$ using AA.
   The intervals are used as those of $\{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ for any $i \geq 1$.

5.2.2.2 On Division of OS-ELM Training Algorithm

The online training algorithm of OS-ELM has a division $\frac{\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T\boldsymbol{h}_i\boldsymbol{P}_{i-1}}{1+\boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T}$. As mentioned in Section 5.1.3, in AA the interval of the denominator $\gamma_i^{(5)} = 1 + \boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T$ must not include zero. Fortunately, $\gamma_i^{(5)}$ never take zero for $i \geq 1$. The rest of this section gives a brief derivation of the property.

Theorem 1. $\boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T \neq 0$ *for any* $i \geq 1$.

Proof. $\boldsymbol{P}_i = (\boldsymbol{P}_{i-1}^{-1} + \boldsymbol{h}_i^T\boldsymbol{h}_i)^{-1}$ holds by applying the Sherman-Morrison formula[2] to Equation 2.15; in other words, $\boldsymbol{P}_{i-1}$ becomes symmetric positive-definite for $i \geq 1$ as long as $\boldsymbol{P}_0$ is symmetric positive-definite.

$\boldsymbol{P}_0 = \boldsymbol{H}_0^T\boldsymbol{H}_0$ is symmetric positive-definite because $\boldsymbol{P}_0$ is assumed to be a regular matrix in OS-ELM, and $\boldsymbol{u}\boldsymbol{P}_0^{-1}\boldsymbol{u}^T = \boldsymbol{u}\boldsymbol{H}_0^T\boldsymbol{H}_0\boldsymbol{u}^T = (\boldsymbol{u}\boldsymbol{H}_0^T)\cdot(\boldsymbol{u}\boldsymbol{H}_0)^T \geq 0$ holds for any real vectors $\boldsymbol{u} \in \mathbb{R}^{1\times\tilde{N}}$. Hence, $\boldsymbol{P}_{i-1}$ is symmetric positive-definite for $i \geq 1$.

An $n \times n$ real symmetric positive-definite matrix $\boldsymbol{V} \in \mathbb{R}^{n\times n}$ satisfies $\boldsymbol{u}\boldsymbol{V}\boldsymbol{u}^T > 0$ for any n-dimensional real vectors $\boldsymbol{u} \in \mathbb{R}^{1\times n}$. Thus, $\boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T > 0$ holds for $i \geq 1$, which guarantees $0 \neq 1 + \boldsymbol{h}_i\boldsymbol{P}_{i-1}\boldsymbol{h}_i^T \Leftrightarrow 0 \neq \gamma_i^{(5)}$ for $i \geq 1$. □

Note that interval($\hat{\gamma}_i^{(5)}$) can include zero because interval($\hat{\gamma}_i^{(5)}$) can be wider than the true interval of $\gamma_i^{(5)}$. To address this problem this work proposes to compute $\min(1, \inf(\hat{\gamma}_i^{(5)}))$ for the lower bound of $\hat{\gamma}_i^{(5)}$ instead of just $\inf(\hat{\gamma}_i^{(5)})$. This trick prevents $\hat{\gamma}^{(5)}$ from including zero and at the same time makes the interval close to the true interval. Thanks to this trick, now the online training algorithm of OS-ELM can be safely represented in AA.

5.2.3 Interval Analysis for Prediction Graph

Prediction graph takes $\{\boldsymbol{x}, \boldsymbol{\beta}\}$ as input. As described in Section 5.2.1, the interval of $\boldsymbol{x}$ is assumed to be [0, 1]. Applying the hypothesis made in Section 5.2.2, the interval of $\boldsymbol{\beta}$ should be a superset of the intervals of $\boldsymbol{\beta}_0$ and $\hat{\boldsymbol{\beta}}_1$; the interval of $\boldsymbol{\beta}$ is given by;

$$\begin{aligned} \text{interval}(\hat{\boldsymbol{\beta}}) &= [\inf(\hat{\boldsymbol{\beta}}), \sup(\hat{\boldsymbol{\beta}})], \\ \inf(\hat{\boldsymbol{\beta}}) &= \min(\min(\boldsymbol{\beta}_0), \inf(\hat{\boldsymbol{\beta}}_1)), \\ \sup(\hat{\boldsymbol{\beta}}) &= \max(\max(\boldsymbol{\beta}_0), \sup(\hat{\boldsymbol{\beta}}_1)). \end{aligned} \tag{5.12}$$

$\boldsymbol{\beta}_0$ is given by the initial training algorithm as a constant. The affine form $\hat{\boldsymbol{\beta}}_1$ is given as an output of training graph.

[2] $(\boldsymbol{V} + \boldsymbol{u}^T\boldsymbol{w})^{-1} = \boldsymbol{V}^{-1} - \frac{\boldsymbol{V}^{-1}\boldsymbol{u}^T\boldsymbol{w}\boldsymbol{V}^{-1}}{1+\boldsymbol{w}\boldsymbol{V}^{-1}\boldsymbol{u}^T}$ $(\boldsymbol{V} \in \mathbb{R}^{n\times n}, \boldsymbol{u} \in \mathbb{R}^{1\times n}, \boldsymbol{w} \in \mathbb{R}^{1\times n})$.

5.3 OS-ELM Core

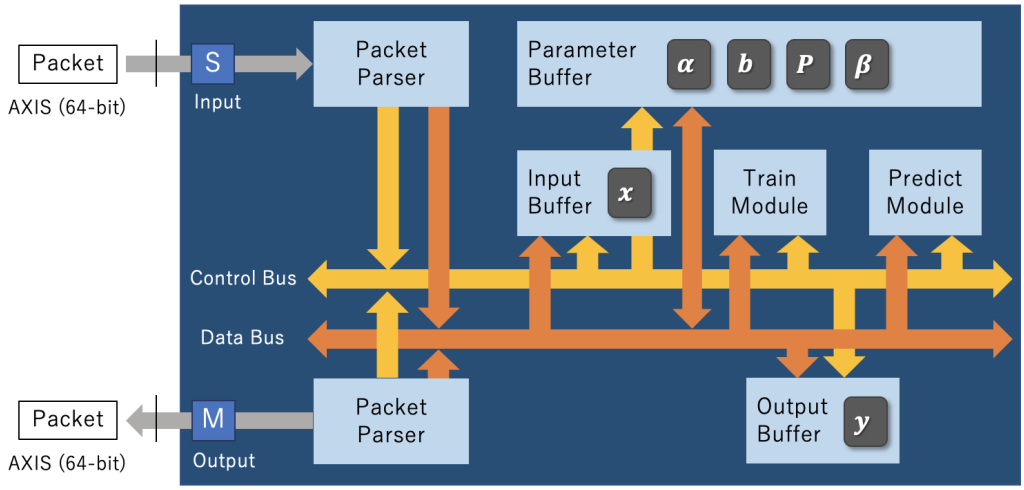

Figure 5.4: Block Diagram of OS-ELM Core

OS-ELM Core, a fixed-point IP core implementing OS-ELM, is a slightly modified version of ONLAD Core. OS-ELM Core has been modified to computes an output vector instead of an anomaly score. All integer bit-widths of the data format are parametrized and the output of the proposed interval analysis method is used as the arguments. PYNQ-Z1 [71] (280 BRAM blocks, 220 DSP slices, 106,400 flip-flops, and 53,200 6-inputs LUTs) is used as the evaluation platform.

Figure 5.4 shows the block diagram of OS-ELM Core. Predict module has been modified to output an output vector instead of an anomaly score. The output vector will be stored in an additional module, named “output buffer”, newly introduced to OS-ELM Core. Output buffer is implemented with LUTs and its value length is configured to 8 bits.

Table 5.4: Intervals Estimated by Simulation (sim) and Proposed Interval Analysis Method (ours) on Each Dataset

	$\boldsymbol{\gamma}_i^{(1)}$	$\boldsymbol{\gamma}_i^{(2)}$	$\boldsymbol{\gamma}_i^{(3)}$	$\gamma_i^{(4)}$	$\gamma_i^{(5)}$
Digits (sim)	$[-0.642, 0.694]$	$[-0.642, 0.694]$	$[-0.446, 0.482]$	$[0.371, 9.75]$	$[1.37, 10.7]$
Digits (ours)	$[-9.92e^3, 9.91e^3]$	$[-9.26, 9.69]$	$[-24.5, 27.8]$	$[0.0, 1.46e^3]$	$[1.0, 1.46e^3]$
Iris (sim)	$[-5.94, 5.85]$	$[-5.94, 5.85]$	$[-4.89, 35.3]$	$[9.27e^{-3}, 3.24]$	$[1.01, 4.24]$
Iris (ours)	$[-1.55e^3, 1.55e^3]$	$[-63.5, 19.1]$	$[-388, 388]$	$[0.0, 48.0]$	$[1.0, 41.7]$
Letter (sim)	$[-6.72e^{-3}, 7.54e^{-3}]$	$[-6.72e^{-3}, 7.54e^{-3}]$	$[-5.06e^{-5}, 5.68e^{-5}]$	$[2.79e^{-3}, 0.0397]$	$[1.0, 1.04]$
Letter (ours)	$[-0.301, 0.307]$	$[-0.0593, 0.0785]$	$[-2.42e^{-3}, 2.44e^{-3}]$	$[0.0, 3.49]$	$[1.0, 4.49]$
Credit (sim)	$[-0.115, 0.116]$	$[-0.115, 0.116]$	$[-8.36e^{-3}, 0.0135]$	$[5.89e^{-3}, 0.253]$	$[1.01, 1.25]$
Credit (ours)	$[-32.9, 32.9]$	$[-2.22, 3.25]$	$[-0.589, 0.589]$	$[0.0, 32.4]$	$[1.0, 33.4]$
Drive (sim)	$[-6.97e^5, 6.92e^5]$	$[-6.98e^5, 6.92e^5]$	$[-3.71e^{11}, 4.87e^{11}]$	$[5.26e^4, 4.72e^6]$	$[5.26e^4, 4.72e^6]$
Drive (ours)	$[-6.56e^{15}, 6.56e^{15}]$	$[-1.33e^7, 1.56e^7]$	$[-1.4e^{13}, 1.4e^{13}]$	$[0.0, 1.55e^9]$	$[1.0, 1.55e^9]$
	$\boldsymbol{\gamma}_i^{(6)}$	$\boldsymbol{\gamma}_i^{(7)}$	$\boldsymbol{\gamma}_i^{(8)}$	$\boldsymbol{\gamma}_i^{(9)}$	$\boldsymbol{\gamma}_i^{(10)}$
Digits (sim)	$[-0.0447, 0.0472]$	$[-0.102, 0.109]$	$[-3.25, 3.29]$	$[-3.0, 3.94]$	$[-0.291, 0.306]$
Digits (ours)	$[-25.8, 27.8]$	$[-9.92e^3, 9.91e^3]$	$[-12.1, 15.4]$	$[-8.38, 9.0]$	$[-8.93e^4, 8.93e^4]$
Iris (sim)	$[-1.32, 8.32]$	$[-1.68, 1.67]$	$[-1.24, 1.69]$	$[-1.5, 2.12]$	$[-2.1, 2.77]$
Iris (ours)	$[-397, 397]$	$[-1.55e^3, 1.55e^3]$	$[-2.61, 2.3]$	$[-2.3, 2.84]$	$[-4.4e^3, 4.4e^3]$
Letter (sim)	$[-4.87e^{-5}, 5.46e^{-5}]$	$[-6.47e^{-3}, 7.25e^{-3}]$	$[-1.29, 1.03]$	$[-0.869, 2.21]$	$[-0.0104, 0.0129]$
Letter (ours)	$[-2.84e^{-3}, 2.86e^{-3}]$	$[-0.301, 0.307]$	$[-3.11, 2.02]$	$[-1.87, 3.31]$	$[-1.01, 1.01]$
Credit (sim)	$[-7.11e^{-3}, 0.0115]$	$[-0.0994, 0.0989]$	$[-2.19, 3.9]$	$[-3.89, 3.03]$	$[-0.314, 0.245]$
Credit (ours)	$[-0.606, 0.606]$	$[-32.9, 32.9]$	$[-11.5, 10.7]$	$[-6.25, 5.62]$	$[-206, 206]$
Drive (sim)	$[-1.36e^5, 1.65e^5]$	$[-1.55, 1.39]$	$[-962, 1.01e^3]$	$[-1.01e^3, 970]$	$[-345, 308]$
Drive (ours)	$[-1.4e^{13}, 1.4e^{13}]$	$[-6.56e^{15}, 6.56e^{15}]$	$[-1e^4, 8.36e^3]$	$[-3.42e^3, 3.44e^3]$	$[-2.26e^{19}, 2.26e^{19}]$
	$\boldsymbol{P}_i$	$\boldsymbol{\beta}_i$	$\boldsymbol{e}_i$	$\boldsymbol{h}_i$	$\boldsymbol{y}$
Digits (sim)	$[-0.0544, 0.0705]$	$[-0.351, 0.451]$	$[-10.6, 9.15]$	$[-10.0, 9.19]$	$[-3.16, 3.25]$
Digits (ours)	$[-27.4, 26.2]$	$[-8.93e^4, 8.93e^4]$	$[-23.1, 20.1]$	$[-22.5, 20.8]$	$[-3.39e^7, 3.39e^7]$
Iris (sim)	$[-1.72, 11.4]$	$[-3.44, 5.32]$	$[-2.44, 1.41]$	$[-3.0, 2.21]$	$[-1.23, 1.79]$
Iris (ours)	$[-358, 435]$	$[-4.4e^3, 4.4e^3]$	$[-2.53, 1.58]$	$[-3.1, 2.38]$	$[-1.71e^4, 1.71e^4]$
Letter (sim)	$[-1.66e^{-3}, 2.45e^{-3}]$	$[-0.34, 0.294]$	$[-4.6, 5.33]$	$[-4.86, 6.01]$	$[-1.25, 1.18]$
Letter (ours)	$[-9.2e^{-3}, 0.0126]$	$[-1.35, 0.99]$	$[-6.6, 7.8]$	$[-6.87, 8.48]$	$[-95.7, 95.3]$
Credit (sim)	$[-0.0649, 0.115]$	$[-1.83, 1.38]$	$[-4.66, 5.5]$	$[-5.55, 6.22]$	$[-2.18, 3.77]$
Credit (ours)	$[-0.625, 1.05]$	$[-204, 208]$	$[-8.29, 9.66]$	$[-9.19, 10.4]$	$[-1.09e^4, 1.09e^4]$
Drive (sim)	$[-1.4e^5, 1.7e^5]$	$[-317, 318]$	$[-9.9, 7.42]$	$[-9.35, 8.29]$	$[-1.21e^3, 318]$
Drive (ours)	$[-1.4e^{13}, 1.4e^{13}]$	$[-2.26e^{19}, 2.26e^{19}]$	$[-18.3, 16.8]$	$[-17.7, 16.0]$	$[-1.06e^{22}, 1.06e^{22}]$

5.4 Evaluations

In this section the proposed interval analysis method is evaluated. Here the proposed method is implemented in software with double-precision. All the experiments here are executed on a common server machine (Ubuntu 20.04, Intel Xeon E5-1650 3.60 GHz, DRAM 64 GB, SSD 500 GB). Table 5.3 lists the classification datasets used for experiments. For all the datasets, the intervals of input $\boldsymbol{x}$ and target $\boldsymbol{t}$ are normalized into $[0, 1]$. Random parameters $\boldsymbol{b}$ and $\boldsymbol{\alpha}$ are generated with the uniform distribution of $[-1, 1]$. The model size on each dataset is shown in "Model Size" column. The number of hidden nodes $\tilde{N}$ is set to the number that performed the best test accuracy in a given search space;

the search spaces for Digits, Iris, Letter, Credit, and Drive are {32, 48, 64, 96, 128}, {3, 4, 5, 6, 7}, {8, 16, 32, 64, 128}, {4, 8, 16, 32, 64}, and {32, 64, 96, 128}, respectively.

5.4.1 Optimization Results

First, this section shows the results of the proposed interval analysis method for each dataset, comparing with an ordinary simulation-based interval analysis. In the simulation method, all the initial training, online training, and prediction algorithms of OS-ELM are implemented in software with double-precision format. Here is a brief description of the baseline simulation method;

1. Execute the initial training algorithm using initial training samples. Then $\boldsymbol{P}_0$ and $\boldsymbol{\beta}_0$ are obtained.

2. Execute the online training algorithm by one step using a single online training sample. Then $\boldsymbol{P}_j$ and $\boldsymbol{\beta}_j$ are obtained at the jth training step.

3. Generate 1,000 random samples $\{\boldsymbol{x}, \boldsymbol{t}\}$ with uniform distribution of [0, 1].

4. Feed all the random samples into the online training algorithm of step = j and measure the values of $\{\boldsymbol{\gamma}_j^{(1)}, \ldots, \boldsymbol{\gamma}_j^{(10)}, \boldsymbol{P}_j, \boldsymbol{\beta}_j, \boldsymbol{e}_j, \boldsymbol{h}_j\}$.

5. Feed all the random samples into the prediction algorithm and measure the values of $\boldsymbol{y}$.

6. Repeat 2-5 until all the online training samples are exhausted.

Table 5.4 shows the intervals obtained from the baseline simulation method (sim) and those from the proposed method (ours). All the intervals obtained from the proposed method cover corresponding simulated intervals. Note that the interval of $\gamma_i^{(5)} = 1 + \boldsymbol{h}_i \boldsymbol{P}_{i-1} \boldsymbol{h}_i^T$ given from simulation satisfies $\gamma_i^{(5)} > 1$, which is consistent with the property described in Section 5.2.2.2.

5.4.2 Occurrence Rate of Overflow/Underflows

In this section the baseline simulation method and the proposed method are compared in terms of occurrence rate of overflow/underflows, using OS-ELM Core. The experimental procedure is as follows;

Table 5.5: Occurrence Rate of Overflow/Underflows

	Arithmetic Operations	Overflow/Underflows
Digits (sim)	5,512,688,688	0
Digits (ours)		0
Iris (sim)	4,714,041	197,342 (4.19%)
Iris (ours)		0
Letter (sim)	17,793,216,000	0
Letter (ours)		0
Credit (sim)	11,039,328,000	0
Credit (ours)		0
Drive (sim)	187,259,827,356	5,467,945,469 (2.92%)
Drive (ours)		0

1. Execute both simulation and proposed interval analysis methods and convert the results into integer bit-widths using Equation 5.10. An extra bit is added to the bit-widths given by the simulation method to reduce overflow/underflows. Note that no extra bt is added to the bit-widths given by the proposed method.

2. Synthesize a pair of OS-ELM Cores using the bit-widths obtained from the simulation and the proposed methods, respectively.

3. Execute online training by one step on both OS-ELM Cores using a single online training sample.

4. Generate 250 random samples $\{\boldsymbol{x}, \boldsymbol{t}\}$ with uniform distribution of $[0, 1]$. Then feed all the random samples into the OS-ELM Cores and execute prediction and online training.

5. Check the count of overflow/underflows that happened in the above process.

6. Repeat 4-7 until all the online training samples are exhausted.

Experimental results are shown in Table 5.5. The "Arithmetic Operations" column shows the total count of arithmetic operations, while the "Overflow/Underflows" column is the count of overflow and underflows that happened during the experiment. The occurrence rate of overflow/underflows over the total arithmetic operations is written in parentheses. The simulation method causes no overflow and underflows on Digits, Letter, and Credit datasets, but it suffers from as many overflow/underflows as 2.92% and 4.19% of the arithmetic operations on Iris and Drive datasets, respectively. OS-ELM is an

online learning algorithm; a few overflow/underflows at early training steps are all propagated to subsequent computations and eventually these can result in a drastic increase of overflow/underflows. This cannot be perfectly prevented as long as a simulation-based random exploration is involved in interval analysis. The proposed method, on the other hand, encounters totally no overflow and underflows since it analytically derives upper and lower bounds of variables and computes sufficient integer bit-widths that never cause overflow/underflows. Although the proposed method produces some redundant bits and it results in a larger area size (see Section 5.4.4), it can safely realize an overflow/underflow-free fixed-point OS-ELM circuit.

5.4.3 Verification of Hypothesis

Figure 5.5 shows the entire set of the simulation results conducted in Section 5.2.2.1. Similar outcomes to Figure 5.3 are observed on all the datasets, which supports the hypothesis that $\forall \boldsymbol{A}_i \in \{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ roughly satisfies interval$(\boldsymbol{A}_1) \supseteq$ interval$(\boldsymbol{A}_i)$ for $2 \leq i$.

In iterative learning algorithms it is known that learning parameters, $\boldsymbol{\beta}_i$ and $\boldsymbol{P}_i$ for OS-ELM, gradually converge to some values as training proceeds. It is considered that the numerical property results in a convergence of the dynamic ranges of $\boldsymbol{\beta}_i$ and $\boldsymbol{P}_i$ as observed in Figure 5.5, then it tightens the dynamic ranges of other variables too as a side-effect via an enormous number of multiplications existing in the online training algorithm of OS-ELM. In the future work, it is planed to investigate the hypothesis either by deriving an analytical proof or using a larger dataset.

5.4.4 Area Cost

This section evaluates the proposed interval analysis method in terms of area cost. The BRAM utilization of OS-ELM Core is referred to as "area cost" in this section, considering that most arrays of OS-ELM Core are implemented in BRAM blocks and the bottleneck of area cost is BRAM utilization. The proposed interval analysis method is compared with the simulation method to clarify how much additional area cost arises to guarantee OS-ELM Core being overflow/underflow-free. The experimental procedure is as follows;

1. Execute the simulation and the proposed interval analysis methods and convert the results to integer bit-widths. Then synthesize a pair of OS-ELM Cores with the output bit-widths.

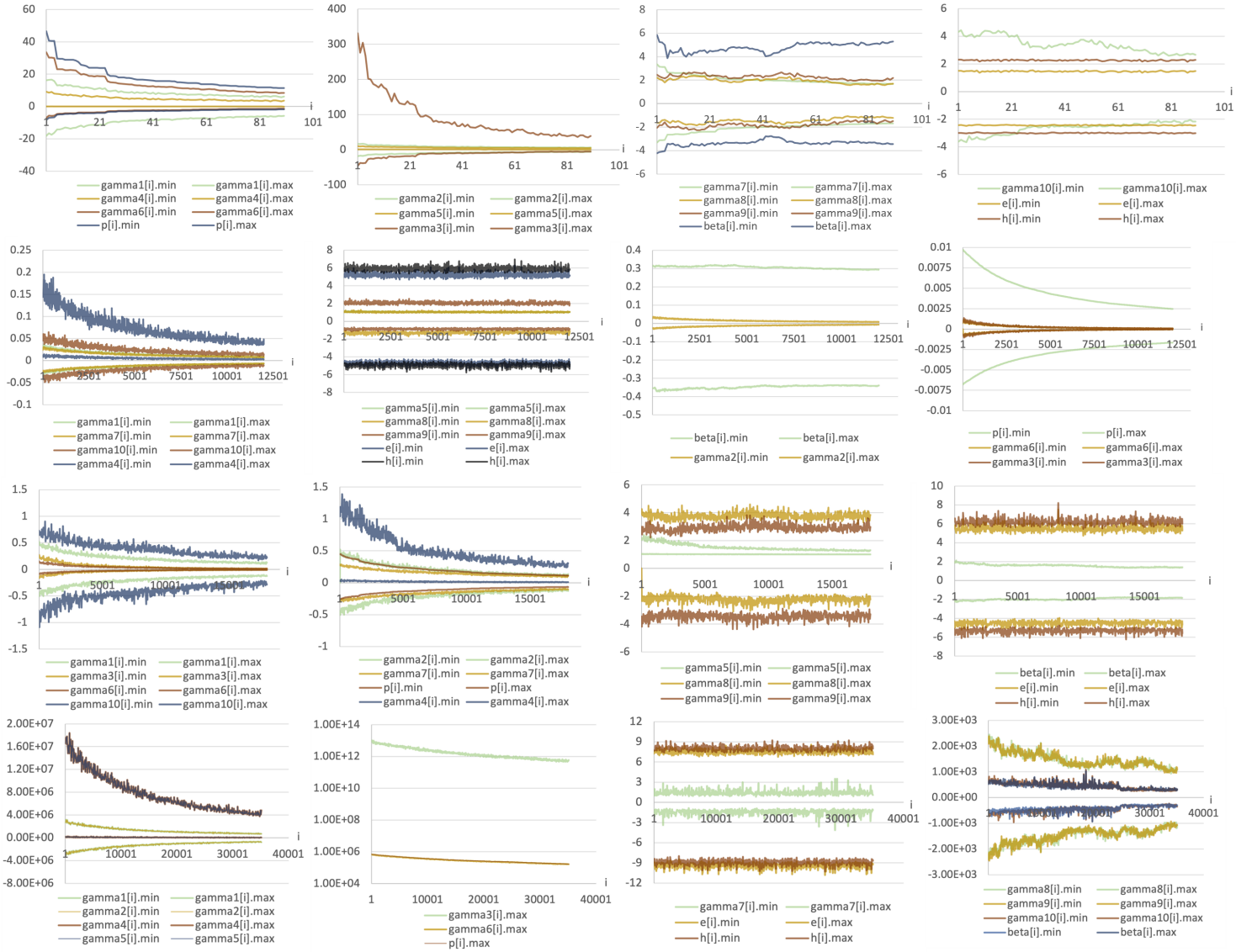

Figure 5.5: Observed Intervals of $\{\boldsymbol{\gamma}_i^{(1)}, \ldots, \boldsymbol{\gamma}_i^{(10)}, \boldsymbol{P}_i, \boldsymbol{\beta}_i, \boldsymbol{e}_i, \boldsymbol{h}_i\}$ on Iris (Top Row), Letter (2nd Row), Credit (3rd Row), and Drive (Bottom row)

2. Check the BRAM utilizations of the proposed method and the simulation method.

3. Repeat 1-2 for all the datasets.

Experimental results are shown in Figure 5.6. The proposed method requires 1.0x ~ 1.5x more BRAM blocks to guarantee that OS-ELM Core never encounters overflow/underflows, compared to the simulation method.

Remember that a multiplication in AA causes overestimation of interval; there should be a strong correlation between the additional area cost and the count of multiplications

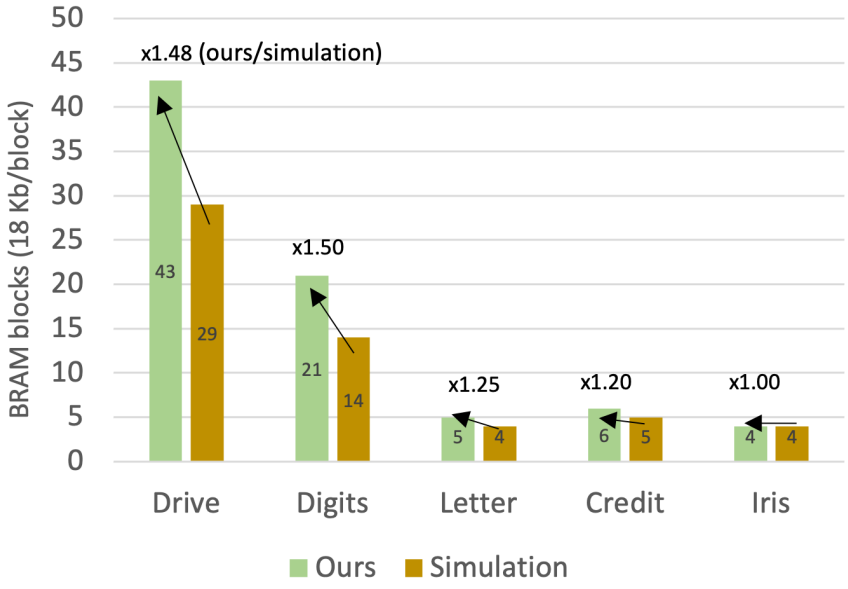

Figure 5.6: Comparison of BRAM Utilization. Green bars represent BRAM utilizations of the proposed method, while brown bars are of the simulation method.

existing in the algorithms of OS-ELM.

$$\mathrm{M}(n, \tilde{N}, m) = 4\tilde{N}^2 + (3m + n + 1)\tilde{N} \tag{5.13}$$

$\mathrm{M}(n, \tilde{N}, m)$ returns the total count of multiplications in the online training and prediction algorithms of OS-ELM, Equation 5.13 implies that $\tilde{N}$ has the largest impact on additional area cost (i.e., Ours - Simulation), which is consistent with the result that 2.0x more additional area cost is observed in Drive compared to Digits, with fewer inputs nodes (Drive: 48, Digits: 64), more hidden nodes (Drive: 64, Digits: 48), and almost the same number of output nodes (Drive: 11, Digits: 10). In summary, the proposed interval analysis method is highly effective especially when the model size is small and the number of hidden nodes has the strongest impact on additional area cost.

5.5 Summary

This chapter proposed an overflow/underflow-free data format optimization method for fixed-point OS-ELM digital circuits. In the proposed method, affine arithmetic is used to estimate the intervals of intermediate variables and compute the optimal number of integer bits that never cause overflow and underflow. In this chapter, two critical problems in realizing the method were clarified: (1) OS-ELM's training algorithm is an iterative algorithm and its computation graph grows endlessly, which makes interval analysis infeasible in affine arithmetic. (2) OS-ELM's training algorithm has a division operation and if the denominator can take zero OS-ELM can not be represented in affine arithmetic. To address those challenges, an empirical solution to prevent the computation graph from growing endlessly was proposed. Also this work provided a mathematical proof that the denominator does not take zero at any training step and proposed a mathematical trick to safely represent OS-ELM in affine arithmetic, based on the proof.

Experimental results confirmed that no underflow/overflow occurred in the proposed method on multiple public datasets. The method realized an overflow/underflow-free OS-ELM digital circuit with 1.0x - 1.5x more area cost compared to the baseline simulation method where overflow or underflow can happen.

Chapter 6

ONLAD-Based Wireless Sensor

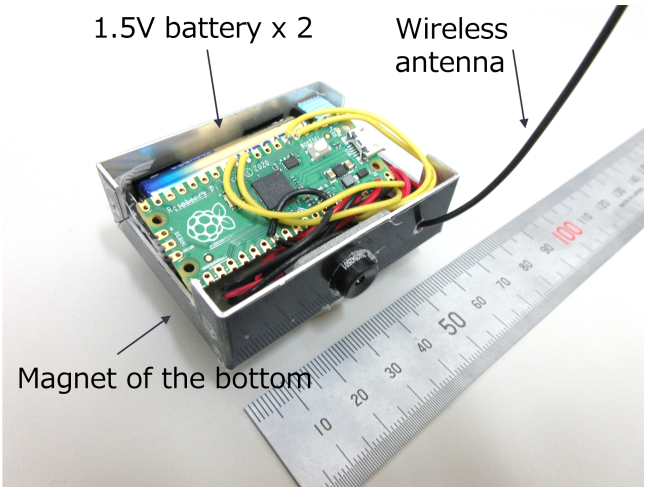

Figure 6.1: ONLAD Sensor

In existing edge-cloud cooperative anomaly detection systems such as AWS Monitron [72], edge sensors are assumed to collect sensor data and execute prediction computations, or just only to collect data. These data are aggregated into cloud servers. In cloud servers, anomaly detection models are trained with collected data, and prediction is also executed in cloud if the edge devices are dedicated to sensing. After training process finishes, updated parameters are delivered to edge sensors if prediction is offloaded to edge.

This chapter proposes an ONLAD-based wireless sensor node, called **ONLAD Sensor**, which executes all the sensing, prediction, and training processes on device. Only prediction results are transferred to cloud servers. The advantages of ONLAD Sensor are summarized as follows;

- **Minimizing Data Traffic**: ONLAD Sensor sends only prediction results to cloud. Since the data size of a prediction result is, in most cases, much smaller than that

of input data[1], the proposed sensor minimizes data traffic, which reduces execution time and energy consumption for communication.

- **Minimizing Cloud WorkLoads**: When using ONLAD Sensor the main task of a cloud server will be just to aggregate prediction results sent from ONLAD Sensor. The cloud server does not need to train models or execute predictions of a high computational cost, which results in reducing workloads on the server.

- **Flexible On-site Adaptation**: In some cases preparing model parameters in advance may be unrealistic. Let us think of anomaly detection of rotary machines like induction motors or cooler fans using their vibration spectrums. The "normal" vibration spectrum is not unique and hard to know in advance because it drastically changes, for example, depending on the mounting position of the sensor, noise from surrounding environment, condition of bearings, and so on. Also it can even change as time passes. In this case it is very hard to prepare model parameters before deployment and the on-site adaptation functionality of ONLAD becomes an essential part to realize anomaly detection in such difficult but not rare cases.

The rest of this chapter is organized as follows; Section 6.1 describes the design and implementation of ONLAD Sensor. The section also describes an experimental edge-cloud cooperative anomaly detection system for evaluating ONLAD Sensor. The proposed sensor is evaluated in terms of energy consumption, latency, and anomaly detection accuracy in Section 6.2. Section 6.3 summarizes this work.

[1]Suppose a 100-class classifier that takes 256x256x3 color images as input. The size of input data is as large as 256 x 256 x 3 x 1 = 0.2 M while that of output vector is only 100.

6.1 Design and Implementation

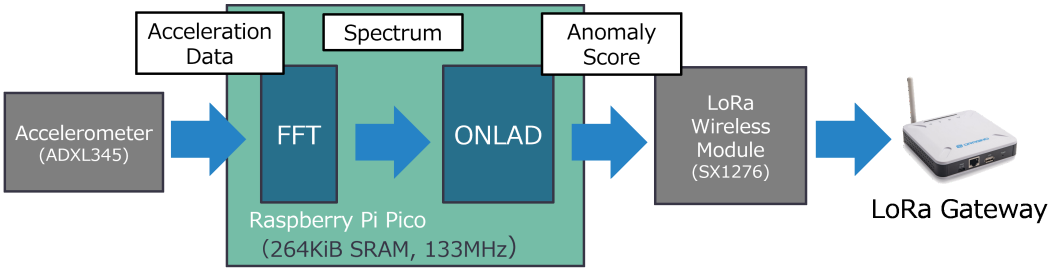

Figure 6.2: Breakdown of ONLAD Sensor

Here the design and implementation of ONLAD Sensor and those of an experimental edge-cloud anomaly detection system to evaluate ONLAD Sensor are described. Figure 6.1 shows ONLAD Sensor. The aim of ONLAD Sensor is to capture anomalous vibrations of rotary machines like induction motors, cooler fans, etc. ONLAD Sensor equips a strong magnet on the bottom of the package so that the user can easily mount the sensor on a machine of interest to detect anomalous vibrations. Figure 6.2 illustrates the module-wise breakdown of ONLAD Sensor. ONLAD Sensor is implemented on a very tiny microcontroller board, Raspberry Pi Pico (MCU: ARM Cortex-M0+ 133 MHz, SRAM: 264 KB). ONLAD Sensor has an accelerometer, ADXL345 from Analog Devices (Voltage: 2.5 V, Resolution: 10 ~ 13 bits, Range: ±16 g), to capture vibrations from a rotary machine as acceleration data. A series of acceleration data is transformed into a spectrum utilizing a software FFT (Fast Fourier Transformation), then the vibration spectrum is fed to ONLAD implemented as a software on the board. The ONLAD part predicts an anomaly score of the spectrum and executes online training.

Figure 6.3 shows the architecture of an experimental system built for evaluation of ONLAD Sensor. Anomaly scores predicted in the ONLAD part are transmitted to the gateway through a LoRa wireless module (Semtech SX1276). LoRa is one of the LPWA (Low Power Wide Area) wireless communication protocols, of which communication range is around ~15km with strictly limited throughput around 0.3 ~ 27 kbps depending on the spreading factor. The LoRa communication protocol is suitable for resource-limited edge devices like ONLAD Sensor due to its ability to communicate over a long distance at very low power consumption. The cloud side manages Grafana, an open-source visualization and monitoring interface, for users to see anomaly scores sent from ONLAD Sensor.

ONLAD Sensor employs the multi-instance ensemble method introduced in Chapter 3

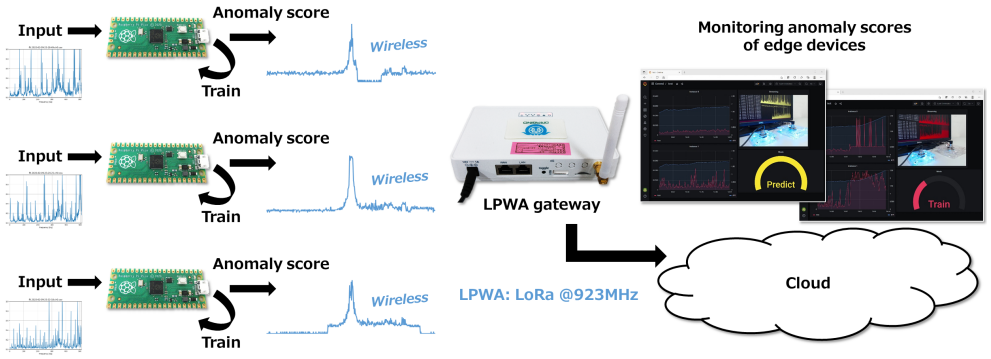

Figure 6.3: Experimental System for Evaluation of ONLAD Sensor

with four ONLAD instances. The model size of each instance is $\{n, \tilde{N}, m\} = \{256, 32, 256\}$, where $n/\tilde{N}/m$ represents the size of input/hidden/output nodes. Random parameters $\boldsymbol{\alpha}$ and $\boldsymbol{b}$ are shared among the instances to save memory consumption. The parameter size of ONLAD Sensor is 176 KB, which fits the limit of Rasperry Pi Pico, in this configuration. 1,024 acceleration data are captured to make one vibration spectrum through FFT.

Figure 6.4 shows the breakdown of execution time of ONLAD Sensor. ONLAD Sensor iterates following five processes for each attempt; (1) **Sensing**: Captures 1,024 acceleration data samples using an accelerometer (ADXL345) implemented in ONLAD Sensor. (2) **FFT**: Converts the series of acceleration data into a vibration spectrum (256 points, 2-Hz resolution). (3) **Predict**: Computes an anomaly score, taking the 256-dimensional spectrum data as an input. (4) **Train**: Executes online training using the spectrum data. (5) **Communication**: Sends a 16-byte packet data containing the anomaly score to cloud, then goes to sleep mode. When the next trigger is issued, it goes back to active mode. The bottleneck of execution time is clearly the sensing part. The higher the rpm (revolutions-per-minute) of the rotary machine is, the more acceleration data are required to capture high frequency components, resulting in a linear increase in the execution time of sensing. Predict part takes more time than Train because the prediction algorithm is executed for the number of ONLAD instances (= 4) while the online training

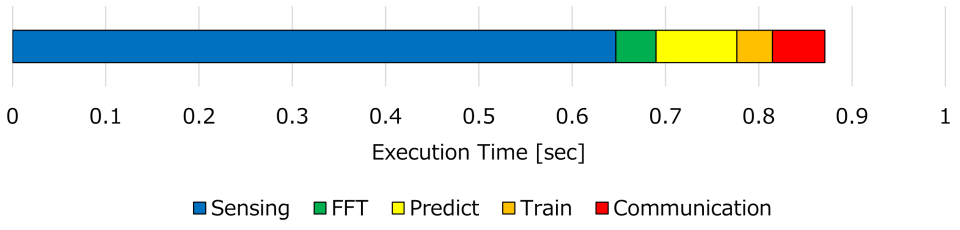

Figure 6.4: Breakdown of Execution Time

is only once.

6.2 Evaluations

6.2.1 Comparison of Execution Time and Power Consumption

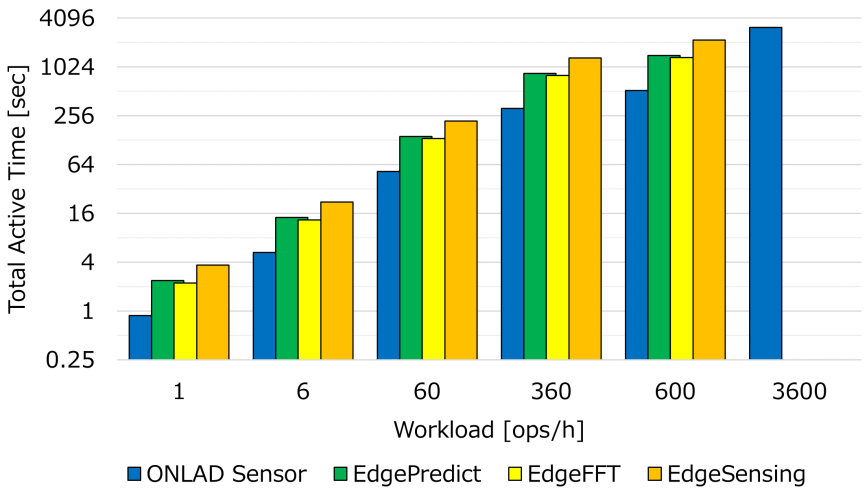

Figure 6.5: Comparison of Total Active Times with Varying Size of Workload

In this section ONLAD Sensor is compared with following three implementations in terms of execution time and power consumption; (1) **EdgePredict**: equivalent to ONLAD Sensor of which training feature is offloaded to cloud, which executes sensing, fft, prediction, and sends 1,024 bytes of spectrum data to cloud. (2) **EdgeFFT**: equivalent to ONLAD Sensor of which training and prediction features are offloaded to cloud, which executes sensing, fft, and sends 1,024 bytes of spectrum data to cloud. (3) **EdgeSensing**: equivalent to ONLAD Sensor of which training, prediction, and fft features are offloaded to cloud, which executes sensing and sends 2,048 bytes of raw acceleration data to cloud. These counterparts represent traditional architectures where training is not executed on device. The aim of the experiment is to clarify the merit of on-device learning approach by comparing the three traditional architectures and ONLAD Sensor.

Figure 6.5 compares total active times (unit: sec) of the four implementations with varying size of workload (unit: ops/h). A total active time plotted on the figure is the sum of elapsed active times of the edge device during a workload. The x-axis unit "ops" is the count of the series of processes (i.e., 1 ops = sensing ~ communication) of each implementation. For experimental results, ONLAD Sensor completed workloads in the shortest total active times in all the cases. Although ONLAD Sensor has the highest on-device computational cost, the on-device learning approach drastically reduces the

communication size to approximately 1/100, which eventually results in a shorter active time compared to the other implementations. This outcome shows the most of active times of EdgePredict, EdgeFFT, and EdgeSensing are occupied with execution time for communication. The on-device learning approach of ONLAD Sensor has a considerable impact when communication size is large. Note that only a total active time of ONLAD Sensor is plotted at workload = 3,600 ops/h because the other implementations could not send packets in time due to saturation of bandwidth.

Similar outcomes are observed in terms of power consumption, too. Figure 6.6 compares power consumptions (unit: mWh) of the four implementations with varying size of workload (unit: ops/h). Experimental results are shown in Figure 6.6. A power consumption plotted on the figure is the power consumption per hour during a workload. Similarly to total active time, ONLAD Sensor completed workloads at the lowest power consumptions in all the cases.

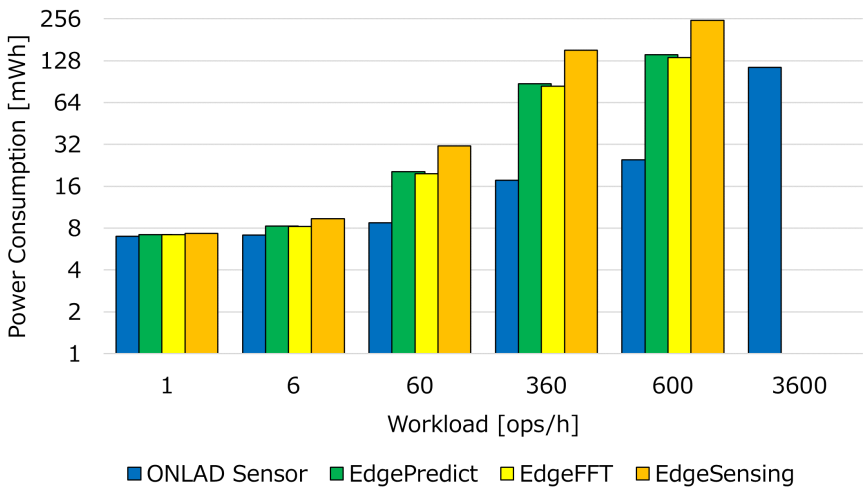

Figure 6.6: Comparison of Power Consumptions with Varying Size of Workload

6.2.2 Comparison of Anomaly Detection Performance

This section evaluates the impact of on-device learning functionality of ONLAD Sensor in a practical application, detecting anomalous vibrations of a cooling fan. **ONLAD Sensor** is compared with **EdgePredict**, the prediction-only version of ONLAD Sensor appearing in the previous section too, in terms of anomaly detection performance (AUC). The difference between the two models is that ONLAD Sensor can execute on-device

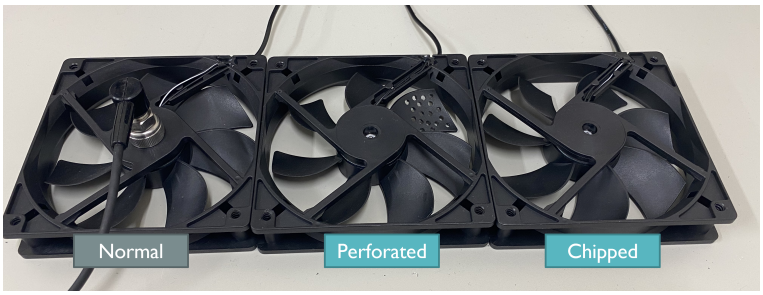

Figure 6.7: Cooling Fans

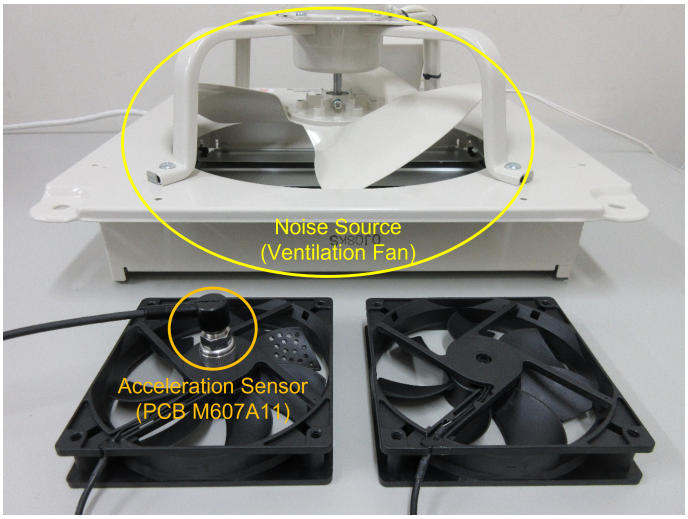

Figure 6.8: Data Acquisition Setup

online training but EdgePredict can only predict with pre-trained parameters. Through the comparison, here demonstrates advantages of the on-device training feature of ONLAD Sensor.

Figure 6.7 shows three types of cooling fans. “Normal” represents a normal cooling fan without any malfunctions or damages. “Perforated” represents a damaged cooling fan with a perforated blade. “Chipped” represents a damaged cooling fan with a chipped blade. These three types are based on the same cooling fan that works at 0 rpm, 1,500 rpm, 2,000 rpm, and 2,500 rpm. Figure 6.8 shows the setup for acquisition of vibration data from cooling fans. Vibration data are captured using PCB M607A11, a mount-type acceleration sensor. The white ventilation turbine is used for a noise source to replicate a real environment where noise is generated by surrounding equipments. Examples of spectrum data of normal cooling fans (not damaged fans) are shown in Figure 6.9. From left to right, each spectrum data is sampled at 2,500 rpm, 1,500 rpm, and 0 rpm. The upper side spectrums are obtained in an environment without any noise source. Here the

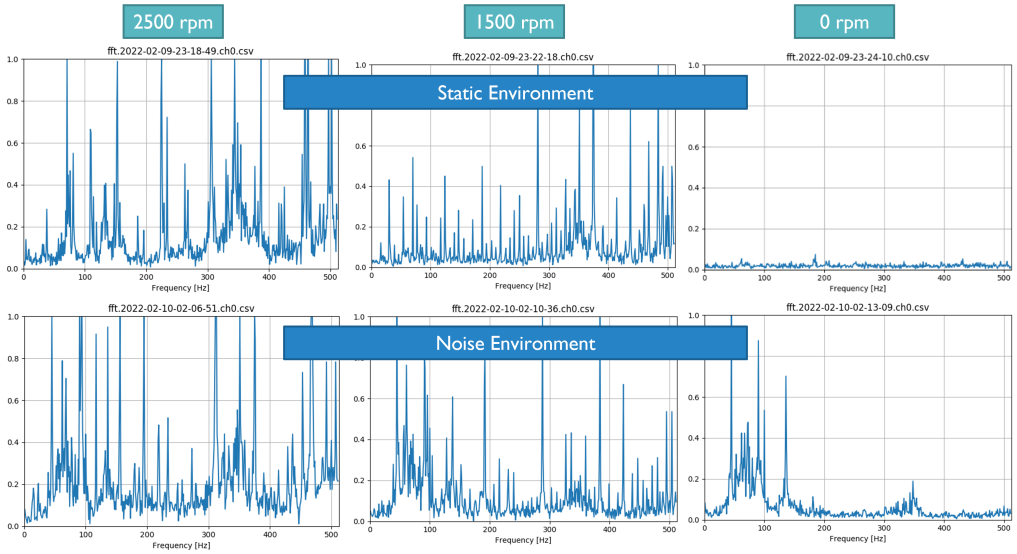

Figure 6.9: Examples of Vibration Spectrums of Cooling Fans

environment is denoted as **"static environment"**. The lower side vibration spectrums are obtained with a noise from the ventilation turbine placed near cooling fans. The environment is denoted as **"noise environment"** here. Noise from the turbine can be seen in the low frequency band of the lower side spectrum data, because the turbine's rpm is much lower than cooling fans.

The following six benchmarks are performed to compare ONLAD Sensor and EdgePredict;

1. **task-{2500,2000,1500,0}rpm**: The task is to identify normal data and anomalies from a series of 2,500-rpm → 2,000-rpm → 1,500-rpm → 0-rpm spectrum data of a normal fan. In task-{x}rpm, {x}-rpm spectrums are treated as normal data while the other types of spectrums are anomalies

2. **task-{perforated,chipped}**: The task is to identify normal data and anomaly data from a series of 2,500-rpm → 2,000-rpm → 1,500-rpm → 0-rpm spectrum data of a normal fan and subsequent spectrum data of perforated or chipped one. Spectrum data of the normal fan are treated as normal data, while those of the perforated or chipped fan are anomalies.

On each benchmark, first ONLAD Sensor and EdgePredict are trained on normal data obtained in static environment as pre-training, then tested with a mixed set of normal and anomaly data obtained in noise environment. In task-{2500,2000,1500,0}rpm, 300 samples are used for pre-training and 235 samples are for test. In task-{perforated,chipped}, 1,200 samples are used for pre-training and 470 samples are for test.

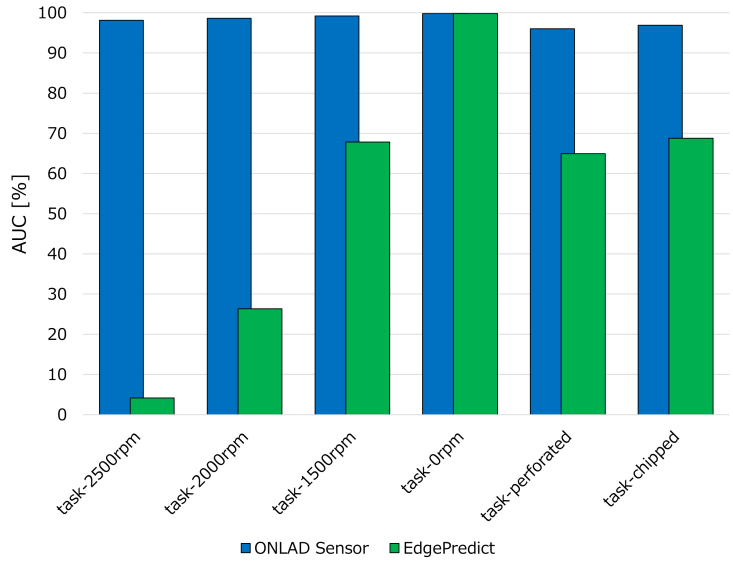

Figure 6.10: AUC Scores of Benchmarks

Benchmark results are shown in Figure 6.10. Identifying 0-rpm vibration spectrums is the easiest task even in noise environment, so both models are almost equally accurate in task-0rpm. However, in task-1500, task-2000, and task-2500, EdgePredict clearly suffers from identifying 1,500-rpm, 2,000-rpm, 2,500-rpm vibration spectrums as normal since these vibration spectrums are less monotonous compared to 0-rpm spectrums (see Figure 6.9) and are affected by resonance caused by the noise source (i.e., ventilation fan) and the running cooling fan. ONLAD Sensor keeps a high AUC score even in noise environment because it can be sequentially trained to adapt to a given environment. The experimental result of task-2500rpm is shown in the upper side of Figure 6.11, in which No.0 ~ No.60 inputs are vibrations of the 2500-rpm cooling fan then subsequent inputs are other rpms vibrations. ONLAD Sensor successfully computed low anomaly scores for 2500-rpm vibrations. In task-perforated and task-chipped, similar results were observed. The results of task-perforated are shown in the lower side of Figure 6.11, in which No.0 ~ No.234 inputs are normal vibrations and subsequent inputs are vibrations of the perforated fan. ONLAD Sensor stably computed high anomaly scores for anomaly vibrations, but EdgePredict sporadically made wrong predictions for anomaly vibrations.

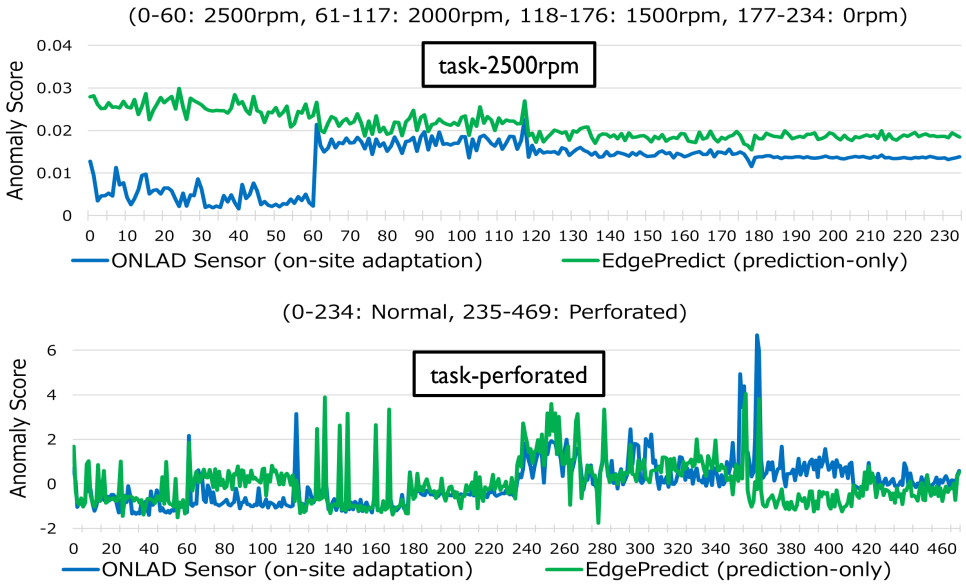

Figure 6.11: Detailed Results of Task-2500rpm (Upper Side) and Task-Perforated (Lower Side)

6.3 Summary

In real-world anomaly detection, normal and anomalous data may vary depending on a given environment. This chapter introduced ONLAD Sensor, an ONLAD-based wireless sensor node for anomaly detection, which executes sensing, prediction, and training processes all on-device. ONLAD Sensor can adapt to a given environment quickly leveraging its fast on-device sequential training functionality based on ONLAD. Also ONLAD Sensor sends only prediction outputs to cloud. Since data size of a prediction output is usually much smaller than input data in data size, ONLAD Sensor minimizes execution time and energy consumption for communication. The experimental results demonstrated that ONLAD Sensor with its sequential training functionality improves anomaly detection accuracy at a noisy environment while saving computation and communication costs for low power.

Chapter 7

Related Work

7.1 Edge Training Technologies

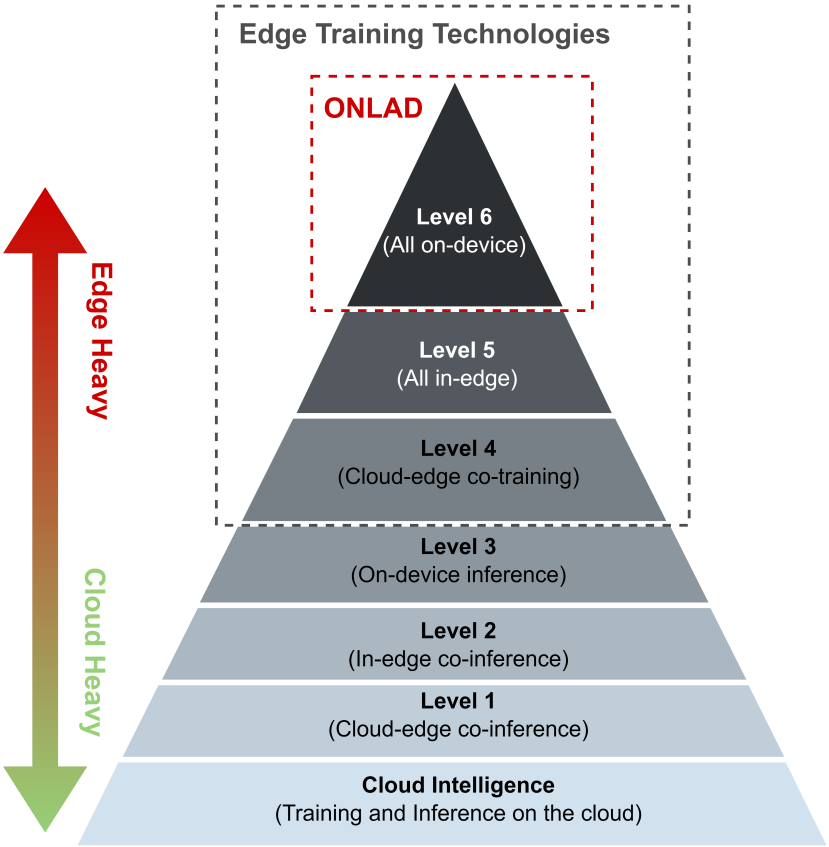

Figure 7.1: 6-Level Possible Architectures of AI Cloud-Edge Systems [1]

Figure 7.1 illustrates the 6-level possible architectures of AI cloud-edge cooperative systems proposed by Zhou *et al.* in [1]. As the level of architecture goes higher, the

computational cost on edge devices increases and the amount of data offloaing to cloud reduces. Thus the upper architectures have less communication cost and data privacy risk, but will impose more computation cost and energy consumption on edge devices. There is no best level of architecture in general and there is only a better choice depending on the application. The definition of each architecture level is given as follows;

- **Cloud Intelligence**: Training and inference are all executed in cloud. Edge devices are dedicated to sensing data.
- **Level 1 (Cloud-Edge Co-Inference)**: Training is done in cloud. Inference is executed in a cloud-edge cooperative manner.
- **Level 2 (In-Edge Co-Inference)**: Training is done in cloud. Inference is executed within a set of edge devices using a device-to-device communication.
- **Level 3 (On-Device Inference)**: Training is done in cloud. Inference is executed on each edge device without device-to-device communication.
- **Level 4 (Cloud-Edge Co-Training)**: Both training and inference are executed in a cloud-edge cooperative manner.
- **Level 5 (All In-Edge)**: Both training and inference are executed within a set of edge devices using a device-to-device communication.
- **Level 6 (All On-Device)**: Both training and inference are executed on each edge device without device-to-device communication. ONLAD belongs to this level.

Here focuses on "**edge training technologies**" which belong to one of levels $4 \sim 6$. Most of edge training technologies proposed so far derive from one of the following four core approaches: (1) Federated Learning, (2) Gossip Training, (3) Gradient Compression, and (4) Model Splitting. The rest of this section reviews related technologies of each core approach and discusses the difference and relationship between ONLAD and them.

7.1.1 Federated Learning

Federated Learning [73] realizes training neural networks without raw data transfers between edge and cloud for privacy-preservation. The key idea of federated learning is to train a global model managed in a central server without exposing raw data from each edge device by aggregating gradient updates given from the devices. Once aggregation round ends, the latest parameters of the global model are transferred back to the devices.

As an aggregation method, FedAvg proposed in [73] simply takes an averages of the gradient updates given by the devices. The challenge of federated learning is how to reduce communication cost since the size of gradient updates can be GB order easily. One simple but effective solution is to reduce aggregation rounds by accumulating local gradients for specific number of rounds [73]. [74] offers two efficient approaches called structured update and sketched update, respectively. Structured updates restricts the space of gradients updates using a matrix decomposition method or a simply random masking, while sketched update reduces the size of data communication by applying quantization and sub-sampling to gradient updates in edge devices before sending the updates to the central server.

The ONLAD approach shares a common idea that edge devices themselves execute training, but the aim is not to create a global model; ONLAD tries to create a locally personalized model for resource-limited edge device.

7.1.2 Gossip Training

Gossip training is similar to federated learning but it forms a peer-to-peer asynchronous decentralized scheme and aims for faster iterations of training. The initial idea of gossip training is proposed in [75], and GoSGD [76] offers a gossip training scheme optimized for deep neural networks. GoSGD only manages a set of end devices (no central server) and iteratively executes the following two steps: (1) gradient update and (2) mixing update. In gradient update step, each node trans the share DNN model using its local data similarly to federated learning. In the following mixing update step, this is the difference from federated learning, each nodes shares its local parameters with another node randomly chosen and updates the parameters based on an weighted average approach. The procedure is repeated on each end node asynchronously until the maximum repetition count is reached. GoSGD realizes fast iterations of training by exploiting asynchronous computations but it suffers from poorer convergence at large-scale.

Similarly to federated learning, gossip training aims to create a global model using a bunch of end nodes. In contrast, ONLAD tries to create a locally personalized model within a single device.

7.1.3 Gradient Compression

In a federated learning scheme, the amount of gradient updates aggregated into the central server is the bottleneck of data communication and has a big impact on throughput of

training. Gradient compression is an effective approach to reduce the communication size. Specifically, gradient quantization and gradient sparsification have been the main techniques. Gradient quantization is to apply lossy compression on gradient updates by encoding them in a finite number of bits to reduce the data size. Tang *et al.* proposed two decentralized SGD gradient compression algorithms, named difference compression and extrapolation compression, respectively [77]. The difference between the two algorithms is that the former quantizes difference of local gradients and it has better convergence when data variation is large, while the latter quantizes extrapolation of local gradients and it is more robust for aggressive quantization. On the other hand, the core idea of gradient sparsification is to eliminate "redundant" gradient updates. A simply way is to introduce a threshold into gradient updates and eliminate subtle updates of which values are below the threshold. Only important gradients above the threshold are sent to the central server while eliminated gradients are accumulated locally until their values surpass the threshold. According to [78] 99.9% of gradient communications in distributed SGD are redundant and their proposed DGC (Deep Gradient Compression) method greatly reduces the communication bandwidth by simply introducing a thresholding approach.

These gradient compression technique cannot be directly applied to ONLAD since it does not expect a decentralized use at least in this thesis. However, the compression methods would work on the combination of OS-ELM-based federated learning approach [79] proposed by Ito et al. and ONLAD, which will be a part of future works.

7.1.4 Model Splitting

Model splitting techniques aim to preserve privacy of user data assuming on a cloud-edge cooperative architecture. The core idea is to split a deep neural network model at somewhere of its layers and to manage the bottom half on local devices and the top half on a central server, which avoids raw data communications and exchanges only intermediate outputs and gradient updates. Mao *et al.* proposed a model splitting framework for a VGG-based face recognition model, where only the first convolutional layer locates on user devices and the rest layers are on an edge server. The framework enables in-edge co-training in a wy that both user data and model parameters are not exposed with only a small cost of local computations.

Note that model splitting will be effective especially when the model is too large to manage it on-device, as on-device manner like ONLAD is more secure in terms of privacy preservation since literally no data except for anomaly scores will be exposed.

7.2 Anomaly Detection with OS-ELM

Since sequential learning approaches are capable of learning input data online, these technologies have been utilized for anomaly detection where real-time adaptation and prediction are often required. OS-ELM is no exception; several studies have been reported on anomaly detection using OS-ELM. Nizar *et al.* proposed an OS-ELM-based irregular behavior detection system of electricity customers to prevent non-technical losses such as power theft and illegal connections [80]. They compared the proposed system with SVM based ones and showed its superiority. Singh *et al.* proposed an OS-ELM-based network traffic IDS (Intrusion Detection System). They showed that the system can perform training on a huge amount of traffic data even with limited memory space [81]. Bosman *et al.* proposed a decentralized anomaly detection system for wireless sensor networks [82]. On the other hand, the ONLAD approach utilizes OS-ELM for semi-supervised anomaly detection in conjunction with an autoencoder. As far as we know, the combination was proposed as the first work in [50] which is a part of this thesis.

7.3 OS-ELM Variants with Forgetting Mechanisms

Over the past years, several OS-ELM variants with forgetting mechanisms have been proposed. Zhao *et al.* were the first to study a forgetting mechanism for OS-ELM, called FOS-ELM [83]. FOS-ELM takes a sliding-window approach, where the latest s training chunks are taken into account (s is a fixed parameter of window size). On the other hand, λ_{DFF}OS-ELM [84] and FP-ELM [31] introduce variable forgetting factors to forget old training chunks gradually with a numerical control. These models adaptively update the forgetting factors according to the information in arriving input data or output error values. The proposed forgetting mechanism in Chapter 2 is based on FP-ELM, which is modified so that it can execute forgetting with a tiny additional computational cost to the original algorithm of OS-ELM, eliminating costly matrix inversion operations.

7.4 Hardware Implementations of OS-ELM

Several papers on hardware implementations of ELM [85–88] have been reported since 2012. However, hardware implementations of OS-ELM have just started to be reported. Tsukada *et al*. provided a theoretical analysis for hardware implementations of OS-ELM to significantly reduce the computational cost [50]. Villora *et al*. and Safaei *et al*. proposed fast and efficient FPGA-based implementations of OS-ELM for embedded systems [49, 51]. In this thesis ONLAD Core, an IP core that implements the proposed OS-ELM-based semi-supervised anomaly detection approach, is proposed. ONLAD Core can be implemented on edge devices of limited resources and works at low power consumption.

Table 7.1: Comparison of NN-based Hardware Implementations for Anomaly Detection

	Akin *et al.* [89]	Wess *et al.* [90]	Moss *et al.* [91]	Alrawashdeh [92]	ONLAD Core
Approach	supervised (classification)	supervised (classification)	semi-supervised (autoencoder)	supervised (classification)	semi-supervised (autoencoder)
NN Model	BP-NN	BP-NN	BP-NN	DBN	OS-ELM
Layers	3	3	5	4	3
Weight Parameters	12	~ 84	1,280	N/A	~ 131,072
Platform	Altera Cyclone III	Avnet Zedboard	Ettus USRP X310	Xilinx ZC706	Digilent PYNQ-Z1
Training Supported ?	No	No	No	Yes	Yes
Frequency	50 MHz	N/A	200 MHz	N/A	142.8 MHz
Latency (prediction)	~2 msec	~100 cycles	105 nsec	8μsec	~ 3.1 msec
Latency (training)	N/A	N/A	N/A	N/A	~ 11.2 msec
Power	N/A	N/A	N/A	N/A	~ 3.2 W
Efficiency	N/A	N/A	N/A	37 gops/W	~ 35.7 mJ/ops

7.5 Neural Network Based Hardware Implementations for Anomaly Detection

Here compares several NN-based anomaly detection hardware implementations with ONLAD Core in Table 7.1. Akin *et al.* proposed an FPGA based condition monitoring system of which prediction time is less than 2 msec, for induction motors [89]. The proposed system employs a supervised anomaly detection approach using a 3-layer binary-classification model; it requires both anomaly data and normal data for training. Wess

et al. proposed an electrocardiogram anomaly detection approach based on FPGA [90]. The proposed system consists of (1) feature extraction, (2) dimensional reduction, and (3) classification, in which (3) is implemented as a dedicated circuit on FPGA. They reported that the prediction latency is approximately less than 100 cycles, although their approach is also based on a classification model as well as [89]. In contrast to the above implementations, ONLAD Core takes a semi-supervised approach, where only normal data are required for training.

Moss *et al.* proposed an FPGA based anomaly detector for radio frequency signals [91]. The proposed IP core realizes semi-supervised anomaly detection using a BP-NN based autoencoder, which is a similar approach to ONLAD Core. Also its prediction latency is as fast as 105 nsec with 100x fewer weight parameters than ONLAD Core. However, its representation capability is limited and the IP core does not support training computations. Alrawashdeh *et al.* proposed a DBN (Deep Belief Network) based IP core that supports training as with ONLAD Core for anomaly detection [92]. They proposed a cost-efficient training model for the contrastive divergence algorithm of DBN and reported that performance of the IP core achieves 37 gops/W. However, the model takes a classification based approach as with [89] and [90]. On the other hand, ONLAD Core supports on-device online training with a semi-supervised anomaly detection approach, which makes it more applicable to a wide range of real-world applications.

7.6 Static Interval Analysis for Iterative Algorithms

Existing general-use static interval analysis methods, including AA, deal with iterative algorithms by expanding them into fixed-length computation graphs using loop unrolling [53–55, 63]. There must be a termination condition for the target iterative algorithm to apply loop unrolling; otherwise it cannot be represented in a fixed-length computation graph, meaning interval analysis becomes infeasible. OS-ELM's training algorithm has no termination condition and existing methods alone cannot realize interval analysis for OS-ELM.

Kinsman *et al.* proposed an SMT (satisfiability modulo theory) based static interval analysis framework designed for iterative algorithms [93] and demonstrated that the method worked for famous iterative algorithms, newton-raphson method and conjugate gradient method, but it still cannot handle algorithms without termination conditions like OS-ELM. Several papers [94–98] proposed analytical interval analysis methods for LTI (linear time invariant) circuits with feedback loops which cannot be translated into fixed-length computation graphs, but the methods are dedicated to LTI circuits and not for OS-ELM. To the best of knowledge, the AA-based method proposed in Chapter 5 is the first work to realize a static interval analysis for OS-ELM by leveraging a numerical hypothesis which is supported by experiments of the chapter.

7.7 Division on Static Interval Analysis

Most of static interval analysis methods assume that all denominators within a target algorithm do not take zero [54, 55, 63], or interval analysis becomes infeasible. The SMT-based method proposed by Kinsman *et al.* provides a mitigation solution to handle a denominator that can take zero by forcibly adding a numerical constraint that prevents it from taking zero (e.g. $y \geq 0.01$ for $z = \frac{x}{y}$), but the constraint $y \geq 0.01$ is inconsistent with the fact $0 \in y$, which can lead to overestimation or underestimation in subsequent intervals. In contrast to the above method, the AA-based method proposed in this thesis takes the safest and optimal strategy; the denominator $\gamma_i^{(5)} = 1 + \boldsymbol{h}_i \boldsymbol{P}_{i-1} \boldsymbol{h}_i^T$ of OS-ELM is analytically guaranteed that it never takes zero at any $i \geq 1$. Also a mathematical trick based on the proof to safely represent OS-ELM in AA is proposed. Note that these contributions can apply to not only AA but also other static interval analysis methods too.

Chapter 8

Conclusions

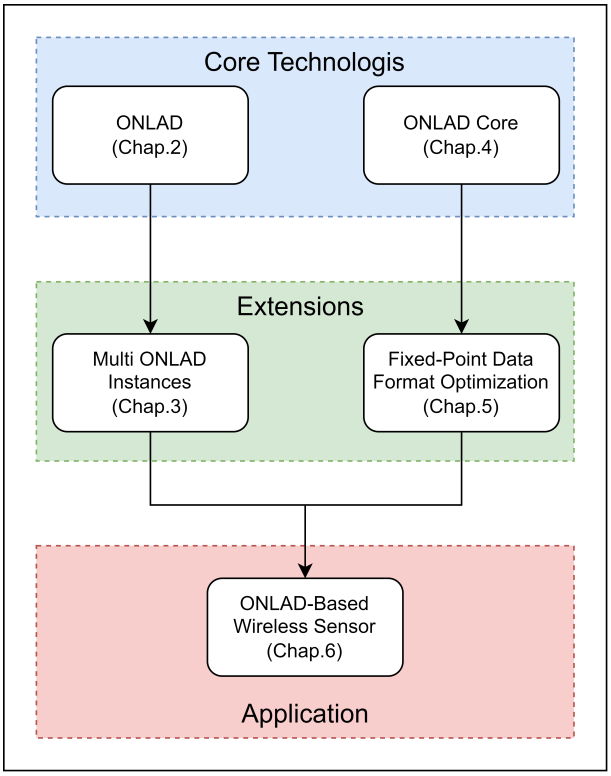

Figure 8.1: Relationships Between Proposed Technologies of Thesis

Figure 8.1 recaps the relationships between the proposed technologies. This thesis proposed the ONLAD algorithm and its IP core implementation named ONLAD Core for resource-limited edge devices. These technologies are positioned as main pillars of the thesis. On top of the basics, an ensemble learning framework leveraging multiple ONLAD instances to extend representation capability of ONLAD was proposed. Also a data format optimization method towards overflow/underflow-free OS-ELM digital circuits for more

stability was proposed. The proposed optimization method is for general OS-ELM digital circuits, thus it can be applied to ONLAD Core to since it is based on OS-ELM. Finally, as an application on top of the contributions of the thesis, an ONLAD-based wireless sensor node for anomaly detection, called ONLAD Sensor was proposed.

The rest of this chapter summarizes the contributions and the novelties of the proposed technologies chapter by chapter;

8.1 Chapter 2: ONLAD

The contributions of Chapter 2 are summarized as follows;

- ONLAD leverages OS-ELM, a light-weight neural network that can execute fast sequential learning, as a core component. Section 2.2.1 provided a theoretical cost analysis of OS-ELM in terms of space and computational complexities, then Section 2.2.2 demonstrated that both space and computational complexities of OS-ELM can be minimized by just setting batch size = 1 without any deterioration of training results unlike BP-NNs.

- Section 2.2.3 proposed a computationally light-weight forgetting mechanism for OS-ELM based on FP-ELM, one of the latest OS-ELM variants with dynamic forgetting mechanism. Since a key feature of semi-supervised anomaly detection is to learn the distribution of normal data, OS-ELM should be able to forget past learned normal data when the distribution changes due to concept drift. The proposed method provides such a function for OS-ELM with a tiny additional computational cost.

- Section 2.2.4 formulated the algorithm of ONLAD, a new sequential learning semi-supervised anomaly detector combining OS-ELM and Autoencoder. This combination, together with the other proposed techniques to reduce the computational cost, realizes fast sequential learning semi-supervised anomaly detection for resource-limited edge devices.

- Experimental results using public datasets showed that ONLAD has comparable anomaly detection accuracy to that of BP-NN-based models in much smaller training epochs with an equal or smaller model size. The experiments also showed that ONLAD keeps high anomaly detection accuracy even in a concept-drifting environment, outperforming BP-NN-based models by $0.10 \sim 0.18$ point in AUC on

three out of five public datasets. ONLAD achieved comparable AUC scores to the BP-NN-based models on the rest of two datasets.

8.2 Chapter 3: Leveraging Multiple ONLAD Instances

The contributions of Chapter 3 are summarized as follows;

- OS-ELM, a core component of ONLAD, is a shallow three-layer neural network. It suffers from low anomaly detection performance when the distribution of normal data is complex or mixed of some sub-distributions. To address this problem, Section 3.1 proposed an ensemble approach leveraging multiple instances of ONLAD. The method shares a common idea with [44] where a set of training data is classified into multiple clusters and an instance is trained with one cluster to reduce complexity of the distribution of training data that each instance learns.
- Section 3.2 evaluated the proposed method in terms of f-measure. The proposed multi-instance method outperformed the original single-instance ONLAD by 8.03% in f-measure under an anomaly detection task built on public datasets.

8.3 Chapter 4: ONLAD Core

The contribution of Chapter 4 are summarized as follows;

- Section 4.1 proposed the design and implementation of ONLAD Core, a hardware IP core implementation of the ONLAD algorithm. The section also provided an FPGA-CPU co-architecture to utilize ONLAD Core assuming a small FPGA evaluation board PYNQ-Z1 which consists of a dual-core ARM CPU and a tiny FPGA chip. ONLAD Core is based on OS-ELM-FPGA [50], an early work by the author, which is the first work to propose a hardware implementation of OS-ELM.
- Section 4.2 evaluated ONLAD Core in terms of latency, energy, and FPGA resource utilization using PYNQ-Z1. Experimental results showed that ONLAD Core can execute the training algorithm x1.9 $\sim$ x7.1 faster and x2.1 $\sim$ x8.1 more energy-efficient compared to a CPU-only software implementation of ONLAD Core with the maximum number of threads fully exploited. It was also shown that ONLAD Core can be implemented into even a smaller FPGA board (Cora Z7) of which resource size is only 1/3 of PYNQ-Z1's chip.

8.4 Chapter 5: Fixed-Point Data Format Optimization for OS-ELM Digital Circuits

The contributions of Chapter 5 are summarized as follows;

- This chapter proposes an interval analysis method for OS-ELM using affine arithmetic, one of the most widely-used interval arithmetic models. Affine arithmetic has been used in a lot of existing works for determining optimal integer bit-widths that never cause overflow and underflows.

- In affine arithmetic, division can be defined only if the denominator does not include zero, or the algorithm cannot be represented in affine arithmetic. There exists only one division operation in OS-ELM's training algorithm; Section 5.2.2.2 gives a guarantee that the denominator does not include zero. The section also proposed a simple mathematical trick to safely represent OS-ELM in affine arithmetic, based on the proof.

- Affine arithmetic can represent only fixed-length computation graphs and unbounded loops cannot be represented in affine arithmetic. However OS-ELM's training algorithm is an iterative algorithm where current outputs are used as the next inputs endlessly. Section 5.2.2 proposed an empirical solution for this problem based on simulation results, and verify its effectiveness in Section 5.4.3.

- Section 5.4 evaluated the proposed interval analysis method using OS-ELM Core, an IP core that implements OS-ELM with fixed-point data format, in terms of occurrence rate of overflow/underflows and additional area cost to guarantee being overflow/underflow-free. OS-ELM Core is a slightly modified version of ONLAD Core introduced in Chapter 4.

8.5 Chapter 6: ONLAD-Based Wireless Sensor

The contributions of Chapter 6 are summarized as follows;

- In real-world anomaly detection, normal and anomalous data may vary depending on a given environment. Section 6.1 introduced the design and implementation of ONLAD Sensor, an ONLAD-based wireless sensor node for anomaly detection, which executes sensing, prediction, and training processes all on-device. ONLAD

Sensor can adapt to a given environment quickly leveraging its fast on-device sequential training functionality based on ONLAD. Also ONLAD Sensor is designed to send only prediction outputs to cloud. Since data size of a prediction output is usually much smaller than input data in data size, ONLAD Sensor minimizes execution time and energy consumption for communication.

- In Section 6.2 experimental results using an edge-cloud cooperative system for evaluation demonstrated that ONLAD Sensor with its sequential training functionality improves anomaly detection accuracy at a noisy environment while saving computation and communication costs for low power.

Bibliography

[1] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. *Proceedings of the IEEE*, 107(8):1738–1762, Jun 2019.

[2] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Computing Surveys*, 41(3):1–58, Jul 2009.

[3] N.V. Chawla, N. Japkowicz, and A. Kolcz. Editorial: Special Issue on Learning from Imbalanced Data Sets. *SIGKDD Explorations*, 6(1):1–6, Jun 2004.

[4] D.D. Lewis and W.A. Gale. A Sequential Algorithm for Training Text Classifiers. *CoRR*, cmp-lg/9407020:1–10, Jul 1994.

[5] A. Estabrooks, T. Jo, and N. Japkowicz. A multiple Resampling Method for Learning from Imbalanced Data Sets. *Computational Intelligence*, 20(1), Feb 2004.

[6] M. Pazzani, C. Mertz, P. Murphy, K. Ali, T. Hume, and C. Brunk. Reducing misclassification costs. In *International Conference of Machine Learning*, pages 217–225, Jul 1994.

[7] M.M. Breunig, H.P. Kriegel, R.T. Ng, and J. Sander. LOF: identifying density-based local outliers. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 93–104, May 2000.

[8] Y. Liao and V.R. Vemuri. Use of K-Nearest Neighbor classifier for intrusion detection. *Computers & Security*, 21(5):439–448, Oct 2002.

[9] K. Leung and C. Leckie. Unsupervised Anomaly Detection in Network Intrusion Detection Using Clusters. In *Proceedings of the Australasian Conference on Computer Science*, pages 333–342, Jan 2005.

[10] G. Münz, S. Li, and G. Carle. Traffic Anomaly Detection Using K-Means Clustering. In *Proceedings of the GI/ITG Workshop*, pages 13–14, Sep 2007.

[11] Y. Wang, J. Wong, and A. Miner. Anomaly intrusion detection using one class SVM. In *Proceedings of the IEEE SMC Information Assurance Workshop*, pages 358–359, Jun 2004.

[12] K.L. Li, H.K. Huang, S.F. Tian, and W. Xu. Improving one-class SVM for anomaly detection. In *Proceedings of the International Conference on Machine Learning and Cybernetics*, pages 3077–3081, Nov 2003.

[13] S.M. Erfani, S. Rajasegarar, S. Karunasekera, and C. Leckie. High-dimensional and large-scale anomaly detection using a linear one-class SVM with deep learning. *Pattern Recognition*, 58:121–134, Oct 2016.

[14] M. Sakurada and T. Yairi. Anomaly Detection Using Autoencoders with Nonlinear Dimensionality Reduction. In *Proceedings of the Workshop on Machine Learning for Sensory Data Analysis*, pages 1–8, Dec 2014.

[15] T. Schlegl, P. Seeböck, S.M. Waldstein, U.S. Erfurth, and G. Langs. Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery. In *Proceedings of the International Conference on Information Processing in Medical Imaging*, pages 146–157, May 2017.

[16] I. Žliobaitė, M. Pechenizkiy, and J. Gama. An overview of concept drift adaptation. *ACM Computing Surveys*, 46(4):44:1–44:37, Apr 2014.

[17] G.I. Webb, M.J. Pazzani, and D. Billsus. Machine Learning for User Modeling. *User Modeling and User-Adapted Interaction*, 11(1-2):19–29, Mar 2001.

[18] M. Hind, S. Mehta, A. Mojsilovic, R. Nair, K.N. Ramamurthy, A. Olteanu, and K.R. Varshney. FactSheets: Increasing Trust in AI Services through Supplier's Declarations of Conformity. *CoRR*, 1808.07261:1–31, Aug 2018.

[19] G.B. Huang, Q.Y. Zhu, and C.K. Siew. Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 985–990, Jul 2004.

[20] G.H. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420, Apr 1970.

[21] G. Marco and T. Alberto. On the Problem of Local Minima in Backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, Jan 1992.

[22] N.Y. Liang, G.B. Huang, P. Saratchandran, and N. Sundararajan. A Fast and Accurate Online Sequential Learning Algorithm for Feedforward Networks. *IEEE Transactions on Neural Networks*, 17(6):1411–1423, Nov 2006.

[23] G.H. Golub and C.F.V. Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, Oct 1996.

[24] G.E. Hinton and R.R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, Jul 2006.

[25] Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[26] P. Vincent, H. Larochelle, Y. Bengio, and P.A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the International Conference on Machine Learning*, pages 1096–1103, Jul 2008.

[27] A. Jinwon and C. Sungzoon. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE*, 2:1–18, 2013.

[28] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1-3):37–52, Aug 1987.

[29] B. Schölkopf, A. Smola, and K.R. Müller. Kernel Principal Component Analysis. In *Proceedings of the International Conference on Artificial Neural Networks*, pages 583–588, Jun 2005.

[30] F. He, T. Liu, and D. Tao. Control batch size and learning rate to generalize well: theoretical and empirical evidence. In *International Conference on Neural Information Processing Systems*, pages 1143–1152, Dec 2019.

[31] D. Liu, Y. Wu, and H. Jiang. FP-ELM: An online sequential learning algorithm for dealing with concept drift. *Neurocomputing*, 207(26):322–334, Sep 2016.

[32] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. `https://github.com/zalandoresearch/fashion-mnist`, 2017.

[33] Y. Lecun and C. Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[34] K.A. Davis and E.B. Owusu. Smartphone Dataset for Human Activity Recognition. https://archive.ics.uci.edu/ml/datasets/Smartphone+Dataset+for+Human+Activity+Recognition+%28HAR%29+in+Ambient+Assisted+Living+%28AAL%29, 2016.

[35] M. Bator. Sensorless Drive Diagnosis. https://archive.ics.uci.edu/ml/datasets/dataset+for+sensorless+drive+diagnosis, 2015.

[36] D. Slate. Letter Recognition Data Set. https://archive.ics.uci.edu/ml/datasets/Letter+Recognition, 1890.

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wickle, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, March 2016.

[38] G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[39] V. Nair and G. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the International Conference on Machine Learning*, pages 807–814, Jun 2010.

[40] D.P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *CoRR*, 1412.6980:1–15, Dec 2014.

[41] M. McCloskey and N.J. Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *Psychology of Learning and Motivation*, 24:109–165, 1989.

[42] B. Mirza, S. Kok, and F. Dong. Multi-Layer Online Sequential Extreme Learning Machine for Image Classification. In *Proceedings of the International Conference on Extreme Learning Machines*, pages 39–49, Dec 2015.

[43] D.A. Reynolds. *Encyclopedia of biometrics*, pages 659–663. Springer US, 2009.

[44] B. Krawczyk, M. Wozniak, and B. Cyganek. Clustering-based ensembles for one-class classification. *Information Sciences*, 264:182–195, Apr 2014.

[45] S. Lloyd. Least squares quantization on pcm. *IEEE Transactions on Information Theory*, 28(2), Mar 1982.

[46] D. Pelleg and A. Moore. *X*-means; Extending *K*-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the International Conference of Machine Learning*, volume 1, pages 727–734, Jun 2000.

[47] Digilent Cora Z7. `https://digilent.com/reference/programmable-logic/cora-z7/start`.

[48] M. Tsukada, M. Kondo, and H. Matsutani. A Neural Network-based On-device Learning Anomaly Detector for Edge Devices. *IEEE Transactions on Computers*, 69(7):1027–1044, Jul 2020.

[49] J.V.F. Villora, A.R. Muñoz, M.B. Mompean, J.B. Aviles, and J.F.G. Martinez. Moving Learning Machine towards Fast Real-Time Applications: A High-Speed FPGA-Based Implementation of the OS-ELM Training Algorithm. *Electronics*, 7(11):1–23, Nov 2018.

[50] M. Tsukada, M. Kondo, and H. Matsutani. OS-ELM-FPGA: An FPGA-Based Online Sequential Unsupervised Anomaly Detector. In *Proceedings of the International European Conference on Parallel and Distributed Computing Workshops*, pages 518–529, Aug 2018.

[51] A. Safaei, Q.M.J. Wu, T. Akilan, and Y. Yang. System-on-a-Chip (SoC)-based Hardware Acceleration for an Online Sequential Extreme Learning Machine (OS-ELM). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Early Access)*, Oct 2018.

[52] D.U. Lee, A.A. Gaffer, R.C.C Cheung, O. Mencer, W. Luk, and G.A. Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1990–2000, Oct 2006.

[53] A. Kinsman and N. Nicolici. Bit-Width Allocation for Hardware Accelerators for Scientific Computing Using SAT-Modulo Theory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(3):405–413, Mar 2010.

[54] D. Boland and G. Constantinides. Bounding Variable Values and Round-Off Effects Using Handelman Representations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11):1691–1704, Nov 2011.

[55] J. Stolfi and L. Figueiredo. Self-Validated Numerical Methods and Applications, 1997.

[56] D. Menard, G. Caffarena, J. Antonio, A. Lopez, D. Novo, and O. Sentieys. *Fixed-point refinement of digital signal processing systems*, pages 1–37. The Institution of Engineering and Technology, May 2019.

[57] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. A methodology and design environment for DSP ASIC fixed point refinement. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 271–276, Mar 1999.

[58] A. Gaffar, O. Mencer, and W. Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *The Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–88, Apr 2004.

[59] H. Keding, M. Willems, and H. Meyr. Fridge: a fixed-point design and simulation environment. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 429–435, Feb 1998.

[60] C. Shi and R. Brodersen. Automated fixed-point data-type optimization tool for signal processing and communication systems. In *Design Automation Conference*, pages 478–483, July 2004.

[61] J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou, and Y. Zou. Evaluation of Static Analysis Techniques for Fixed-Point Precision Optimization. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, pages 231–234, Apr 2009.

[62] S. Vakili, J.M.P Langlois, and G. Bois. Enhanced Precision Analysis for Accuracy-Aware Bit-Width Optimization Using Affine Arithmetic. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(12):1853–1865, Dec 2013.

[63] R. Moore. Interval Analysis. *Science*, 158(3799):365–365, Oct 1967.

[64] C.F. Fang, R.A. Rutenbar, and T. Chen. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In *Proceedings of the International Conference on Computer Aided Design*, pages 1–8, Nov 2003.

[65] Y. Pu and Y. Ha. An automated, efficient and static bit-width optimization methodology towards maximum bit-width-to-error tradeoff with affine arithmetic model. In *Proceedings of the Asia and South Pacific Conference on Design Automation*, pages 886–891, Jan 2006.

[66] S. Wang and X. Qing. A Mixed Interval Arithmetic/Affine Arithmetic Approach for Robust Design Optimization With Interval Uncertainty. *Journal of Mechanical Design*, 138(4):041403–1–041403–10, Apr 2016.

[67] R. Bellal, E. Lamini, H. Belbachir, S. Tagzout, and A. Belouchrani. Improved Affine Arithmetic-Based Precision Analysis for Polynomial Function Evaluation. *IEEE Transactions on Computers*, 68(5):702–712, May 2019.

[68] E. Alpaydin and C. Kaynak. Optical Recognition of Handwritten Digits Data Set. `https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits`, 1998.

[69] R. Fisher. Iris Data Set. `http://archive.ics.uci.edu/ml/datasets/Iris/`, 1936.

[70] I. Yeh. Default of Credit Card. `https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients`, 2016.

[71] Digilent PYNQ-Z1. `https://japan.xilinx.com/products/boards-and-kits/1-hydd4z.html`.

[72] Amazon Monitron. `https://aws.amazon.com/monitron/`.

[73] B. McMahan, E. Moore, D. Ramage, and S. Hampson ad B.A. Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (PMLR)*, pages 1–10, Apr 2017.

[74] J. Konečný, H.B. McMahan, F.X. Yu, P. Richtárik, A.T. Suresh, and D. Bacon. Federated Learning: Strategies for Improving Communication Efficiency. *CoRR*, 1610.05492:1–10, Oct 2016.

[75] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, Jun 2006.

[76] M. Blot, D. Picard, M. Cord, and N. Thome. Gossip training for deep learning. *CoRR*, abs/1611.09726:1–5, Nov 2016.

[77] H. Tang, S. Gan, C. Zhang, T. Zhang, and J. Liu. Communication compression for decentralized training. In *Advances in Neural Information Processing Systems*, volume 31, pages 1–11, Jul 2018.

[78] Y. Lin, S. Han, H. Mao, Y. Wang, and W.J. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *CoRR*, pages 1–14, Dec 2017.

[79] R. Ito, M. Tsukada, and H. Matsutani. An On-Device Federated Learning Approach for Cooperative Model Update between Edge Devices. *IEEE Access*, 9:92986–92998, Jun 2021.

[80] A.H. Nizar and Z.Y. Dong. Identification and detection of electricity customer behavior irregularities. In *Proceedings of the IEEE Power Systems Conference and Exposition*, pages 1–10, March 2009.

[81] R. Singh, H. Kumar, and R.K. Singla. An Intrusion Detection System using Network Traffic Profiling and Online Sequential Extreme Learning Machine. *Expert Systems with Applications*, 42(22):8609–8624, Dec 2015.

[82] H.H.W.J. Bosman, G. Iacca, A. Tejada, H.J. Wörtche, and A. Liotta. Spatial Anomaly Detection in Sensor Networks using Neighborhood Information. *Information Fusion*, 33:41–56, Apr 2016.

[83] J. Zhao, Z. Wang, and D.S. Park. Online sequential extreme learning machine with forgetting mechanism. *Neurocomputing*, 87(15):79–89, Jun 2012.

[84] S.G. Soares and R. Araújo. An adaptive ensemble of on-line Extreme Learning Machines with variable forgetting factor for dynamic system prediction. *Neurocomputing*, 171(1):693–707, Jan 2016.

[85] S. Decherchi, P. Gastaldo, A. Leoncini, and R. Zunino. Efficient Digital Implementation of Extreme Learning Machines for Classification. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59(8):496–500, Aug 2012.

[86] T.C. Yeam, N. Ismail, K. Mashiko, and T. Matsuzaki. FPGA Implementation of Extreme Learning Machine System for Classification. In *Proceedings of the IEEE Region 10 Conference*, pages 1868–1873, Nov 2017.

[87] J.V.F. Villora, A.R. Muñoz, J.M.M. Villena, M.B. Mompean, J.F. Guerrero, and M. Wegrzyn. Hardware Implementation of Real-time Extreme Learning Machine in FPGA: Analysis of Precision, Resource Occupation and Performance. *Computers & Electrical Engineering*, 51:139–156, Feb 2016.

[88] A. Basu, S. Shuo, H. Zhou, M.H. Lim, and G.B. Huang. Silicon spiking neurons for hardware implementation of extreme learning machines. *Neurocomputing*, 102(15):125–134, Feb 2013.

[89] E. Akin, I. Aydin, and M. Karakose. FPGA Based Intelligent Condition Monitoring of Induction Motors: Detection, Diagnosis, and Prognosis. In *Proceedings of the IEEE International Conference on Industrial Technology*, pages 373–378, April 2011.

[90] M. Wess, P.D.S. Manoj, and A. Jantsch. Neural network based ECG anomaly detection on FPGA and trade-off analysis. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1–4, May 2017.

[91] D.J.M Moss, D. Boland, P. Pourbeik, and P.H.W. Leong. Real-time FPGA-based Anomaly Detection for Radio Frequency Signals. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1–5, May 2018.

[92] K. Alrawashdeh and C. Purdy. Fast Hardware Assisted Online Learning Using Unsupervised Deep Learning Structure for Anomaly Detection. In *Proceedings of the International Conference on Information and Computer Technologies*, pages 128–134, May 2018.

[93] A. Kinsman and N. Nicolici. Automated Range and Precision Bit-Width Allocation for Iterative Computations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1265–1278, Sep 2011.

[94] J. Ĺopez, C. Carreras, and O. Nieto-Taladriz. Improved Interval-Based Characterization of Fixed-Point LTI Systems With Feedback loops. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(11):1923–1933, Nov 2007.

[95] O. Sarbishei, Y. Pang, and K. Radecka. Analysis of Range and Precision for Fixed-Point Linear Arithmetic Circuits with Feedbacks. In *Proceedings of the IEEE international High Level Design Validation and Test Workshop*, pages 25–32, June 2010.

[96] O. Sarbishei, K. Radecka, and Z. Zilic. Analytical Optimization of Bit-Widths in Fixed-Point LTI Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(3):343–355, Mar 2012.

[97] E. Lamini, R. Bellal, H. Belbachir, and A. Belouchrani. Enhanced Bit-Width Optimization for Linear Circuits with Feedbacks. In *Proceedings of the International Design and Test Symposium*, pages 168–173, Dec 2014.

[98] S. Ohno and S. Wang. Overflow-free realizations for lti digital filters. In *Proceedings of the International Symposium on Intelligent Signal Processing and Communication Systems*, pages 1–2, Dec 2019.

Publications

Related Papers

International Journal Papers

[1] Mineto Tsukada and Hiroki Matsutani, "An Overflow/Underflow-Free Fixed-Point Bit-Width Optimization Method for OS-ELM Digital Circuit", IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Section on VLSI Design and CAD Algorithms, Vol.E105-A, No.3, pp.437-447, Mar. 2022.

[2] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "A Neural Network-Based On-device Learning Anomaly Detector for Edge Devices", IEEE Transactions on Computers (TC), Vol.69, No.7, pp.1027-1044, Jul. 2020. (Featured Paper in July 2020 Issue of IEEE TC)

Domestic Journal Papers

[3] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "OSUAD: An FPGA-Based Online Sequential Learning Unsupervised Anomaly Detector", IPSJ Transactions on Advanced Computing Systems (ACS), Vol.12, No.3, pp.34-45, Jul. 2019.

International Conference Papers

[4] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "OS-ELM-FPGA: An FPGA-Based Online Sequential Unsupervised Anomaly Detector", Proc. of the 24th International European Conference on Parallel and Distributed Computing (Euro-Par'18) Workshops, The 16th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'18), pp.518-529, Aug. 2018.

International Conference Demonstrations

[5] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "An FPGA-based On-device Sequential Learning Approach for Unsupervised Anomaly Detection", The 27th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'19), Demo Night, Apr. 2019.

Domestic Conference Papers and Technical Reports

[6] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "Anomaly Detection using On-Device Learning Algorithm on Wireless Sensor Nodes", IEICE Technical Reports CPSY2022-10 (SWoPP'22), Vol.122, No.133, pp.53-58, Jul. 2022. (IPSJ ARC Young Researcher Encouragement Award)

[7] Mineto Tsukada and Hiroki Matsutani, "Automated Fixed-Point Bit-Length Optimization for OS-ELM", IEICE Technical Reports CPSY2020-4 (SWoPP'20), Vol.120, No.121, pp.23-28, Jul. 2020.

[8] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "A Method for Improving Accuracy using Multiple Online Unsupervised Anomaly Detection Cores", IEICE Technical Reports CPSY2018-114 (ETNET'19), Vol.118, No.514, pp.247-252, Mar. 2019.

[9] Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "A Stable and Efficient Learning Method for FPGA-Based Online Sequential Unsupervised Anomaly Detector", IEICE Technical Reports CPSY2018-30 (SWoPP'18), Vol.118, No.165, pp.217-222, Aug. 2018. (IPSJ ARC Young Researcher Encouragement Award)

[10] Mineto Tsukada, Koya Mitsuzuka, Kohei Nakamura, Yuta Tokusashi and Hiroki Matsutani, "Accelerating Sequential Learning Algorithm OS-ELM Using FPGA-NIC", IEICE Technical Reports CPSY2017-127, Vol.117, No.378, pp.133-138, Jan. 2018. (IEICE CPSY Young Presentation Award)

Other Papers

International Journal Papers

[11] Rei Ito, Mineto Tsukada and Hiroki Matsutani, "An On-Device Federated Learning Approach for Cooperative Model Update between Edge Devices", IEEE Access, Vol.9, pp.92986-92998, Jun. 2021.

International Conference Papers

[12] Hirohisa Watanabe, Mineto Tsukada and Hiroki Matsutani, "An FPGA-Based On-Device Reinforcement Learning Approach using Online Sequential Learning", Proc. of the 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS'21) Workshops, The 28th Reconfigurable Architectures Workshop (RAW'21), pp.96-103, May. 2021.

[13] Tokio Kibata, Mineto Tsukada and Hiroki Matsutani, "An Edge Attribute-wise Partitioning and Distributed Processing of R-GCN using GPUs", Proc. of the 26th International European Conference on Parallel and Distributed Computing (Euro-Par'20) Workshops, The 18th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'20), pp.122-134, Aug. 2020.

[14] Rei Ito, Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, "An Adaptive Abnormal Behavior Detection using Online Sequential Learning", Proc. of the 17th International Conference on Embedded and Ubiquitous Computing (EUC'19), pp.436-440, Aug. 2019.

[15] Tomoya Itsubo, Mineto Tsukada and Hiroki Matsutani, "Performance and Cost Evaluations of Online Sequential Learning and Unsupervised Anomaly Detection Core", Proc. of the 22nd IEEE Symposium on Low-Power and High-Speed Chips and Systems (COOL Chips 22), pp.1-3, Apr. 2019.

[16] Kaho Okuyama, Yuta Tokusashi, Takuma Iwata, Mineto Tsukada, Kazumasa Kishiki and Hiroki Matsutani, "Network Optimizations on Prediction Server with Multiple Predictors", Proc. of the 16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'18), pp.1044-1045, Dec. 2018

Domestic Conference Papers and Technical Reports

[17] Rei Ito, Mineto Tsukada and Hiroki Matsutani, "An Efficient Cooperative Model Update using On-Device Learning", IEICE Technical Reports CPSY2019-65, Vol.119, No.372, pp.79-84, Jan. 2020.

[18] Hirohisa Watanabe, Mineto Tsukada and Hiroki Matsutani, "A Light-Weight Reinforcement Learning using Online Sequential Learning", IEICE Technical Reports CPSY2019-66, Vol.119, No.372, pp.85-90, Jan. 2020.

[19] Tomoya Itsubo, Mineto Tsukada and Hiroki Matsutani, “Area and Performance Evaluations of Online Sequential Learning and Unsupervised Anomaly Detection Core”, IEICE Technical Reports CPSY2018-96, Vol.118, No.431, pp.83-88, Jan. 2019. (IEICE CPSY Young Presentation Award)

[20] Rei Ito, Mineto Tsukada, Masaaki Kondo and Hiroki Matsutani, “A Case for Unsupervised Abnormal Behavior Detection Using Multiple Online Sequential Learning Cores”, IEICE Technical Reports CPSY2018-95, Vol.118, No.431, pp.77-82, Jan. 2019.

[21] Kaho Okuyama, Takuma Iwata, Mineto Tsukada, Masakazu Kishiki and Hiroki Matsutani, “FPGA and DPDK-Based Communication Acceleration Methods for Prediction Server with Multiple Predictors”, IEICE Technical Reports CPSY2018-5 (HotSPA’18), Vol.118, No.92, pp.101-106, Jun. 2018.