Journal of Signal Processing Systems
https://doi.org/10.1007/s11265-023-01835-1

Design and Implementation of an FPGA-Based DNN Architecture for Real-time Outlier Detection

Nadya Mohamed[1] · **Joseph Cavallaro**[1]

Received: 9 June 2022 / Revised: 25 October 2022 / Accepted: 27 December 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
Deep neural networks (DNNs) have recently become the standard tool for solving practical problems in various applications, including timely data analysis and near real-time accurate decision-making. DNNs have proven effective in outlier detection, one of sensor networks' primary motivating data analysis applications. Despite the great potential of deep neural networks, their computational resource requirements create a vast gap when it comes to the fast processing time required in real-time applications using low-power, low-cost edge devices. Special care must be taken into account when designing DNNs computational units. This work proposes an FPGA-based Deep Neural Network (DNN) architecture for real-time outlier detection in time series data. The proposed architecture integrates a fine-tuned Autoencoder network and a Long short-term memory (LSTM) network to predict and detect outliers in real-time. The hardware accelerator of the integrated networks combines serial-parallel computation with matrix algebra concepts to reduce computational complexity and enhance the throughput. Experimental results on the resource-constrained Xilinx PYNQ-Z1 board using an open-source sensor network dataset show that the proposed architecture can efficiently analyze and detect outliers in real-time. The implemented design achieves 0.22 ms average latency and 1GOPS throughput. The proposed design's low latency and 94mW power consumption make it suitable for resource-constrained edge platforms.

Keywords Outlier detection · Sensor Networks · FPGA · Accelerator · DNN · Edge computing

1 Introduction

The world we live in and the one we are designing for tomorrow is based on monitoring physical systems, specifically the capability of continuously gathering fine-grained information from the physical world using sensor networks. The ultimate goal of sensor networks goes beyond monitoring and data collection; they are concerned with timely data analysis and accurate real-time decision-making [1, 2]. Outlier detection is a primary motivating data analysis application in sensor networks. Outliers are data points that deviate significantly from the remaining data to arouse suspicions that a different mechanism generated them [3]. The recognition of outliers provides valuable insights into the characteristics of the underlying generating system. A central network entity, such as the cloud, is usually used for the task of outlier detection. However, this requires moving the data from the data source, sensors, to a centralized location in the cloud. Such a scheme introduces several challenges. Sending data to the cloud for inference incurs additional propagation delays from the network, leading to failure to satisfy the end-to-end low-latency requirements for real-time interactive applications. In addition, uploading data from the sources to the cloud introduces scalability issues in network resource utilization, especially when not all the data from all the resources are needed. Besides, uploading sensitive information to the cloud and how the cloud or applications will use these data risks privacy.

Hierarchical computing models, Fig. 1, have been introduced to reduce the amount of data uploaded to the cloud and enable timely data analysis at different network ends. Fog computing is an extension of the cloud that leverages computing capabilities within the local cloud network. On the other hand, edge computing enables analytics to be physically close to the data source at sensing and aggregation nodes. Enabling data analytics, including outlier detection close to

✉ Nadya Mohamed
nam7@rice.edu

Joseph Cavallaro
cavallar@rice.edu

[1] Department of Electrical and Computer Engineering, Rice University, 6100 Main St, Houston 77005, TX, USA

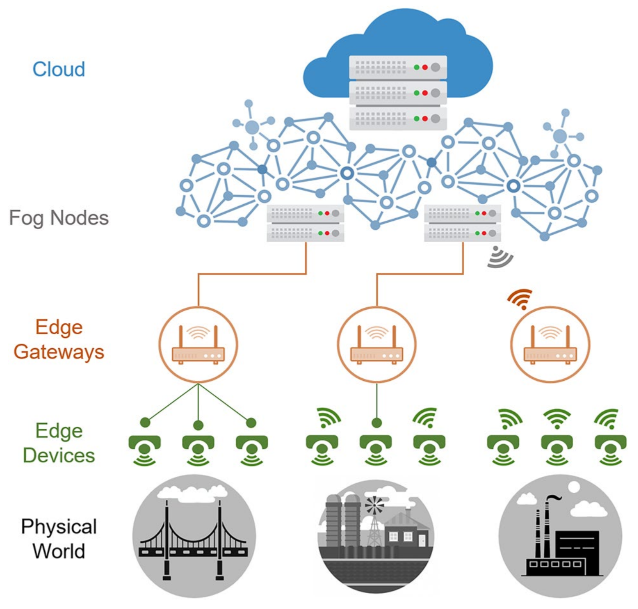

Figure 1 The general hierarchical computing model proposed to reduce the amount of data uploaded to the cloud and enable data analytics at different network ends.

the end devices through edge computing, is a viable solution to meet the latency, privacy, and scalability challenges described earlier. The limited computational resources on the edge nodes and devices motivate the development of lightweight, energy-efficient outlier detection algorithms.

1.1 Related Work

There is an increasing research interest in the area of outlier detection in wireless sensor networks (WSNs). Several solutions based on various approaches have been proposed in the literature [4, 5]. The general idea is to model the distribution of what is considered "normal" and then check if the new target data deviate from the distribution to a significant degree. Various statistical techniques surveyed in [4, 5] model the data in the form of a closed-form probability distribution, and the parameters of the models are learned. However, fitting data to a particular distribution may sometimes be inappropriate. Other techniques based on thresholding approaches are also surveyed by [4, 5], but identifying a suitable threshold is challenging. Even if found, it is hard to generalize to all settings. Therefore, non-parametric methods are proposed to overcome the limitations of parametric methods. In [6], the k-nearest neighbors' algorithm is proposed to create a hyper-grid around a given data point and consider the data point abnormal if less than k other data points lie inside that hyper-grid. The computational complexity of the proposed algorithm is $O(2^{M-1})$, where M is the input data dimension. For high-dimensional data, the complexity of the algorithm will prohibit its usage. One possible solution to overcome the limitation of the k-nearest approach proposed in [6] is to use a distributed algorithm. A distributed algorithm that relies on message exchange among sensor nodes to detect outliers is proposed in [7]. It is a classical spatial correlation-based outlier detection method. Although the approach is computationally less intensive, it requires reliable communications among sensor nodes, which imposes serious power consumption. To reduce the communication overheads, authors in [8] proposed a distributed online anomaly detection based on a hyper-ellipsoidal one-class support vector machine (SVM). The presented approach in [8] takes advantage of the spatiotemporal correlation between sensor data and updates the SVM ellipsoidal boundary to reflect the change in the normal sensor data. However, this technique includes matrix inversions, which are computationally unfavorable to resource-constrained sensors.

1.2 Contribution

Inspired by the success of Deep neural networks in solving various practical problems, our earlier work published in [13] explored the integration of Deep Autoencoder (AE) and Long Short-Term Memory (LSTM) networks for outlier detection. AEs are feedforward multi-layer neural networks for feature learning. They have significantly impacted areas such as speech recognition, object recognition, natural language processing, and dynamical system control [14, 15, 22]. AE's primary objective is to learn a map from the input to itself through a pair of encoding and decoding phases. The learned mapping could then be used as input to another machine-learning model. LSTM networks are a special kind of Recurrent Neural Network (RNN) capable of learning long-term dependencies, making them suitable for sequential data processing [16]. LSTM networks are fully connected single or multi-layer networks with complex neurons and internal states at each time step that enable them to build a memory of time-series events. The proposed solution in [13] consisted of a fine-tuned deep Autoencoder to extract the latent features in sensor data, followed by an LSTM network to predict the next step and detect outliers in real-time. The effectiveness of the proposed architecture is validated on a resource-constrained Xilinx PYNQ-Z1 board using an open-source WSN meteorological dataset.

This work is an extension of [13] and focuses on optimizing the FPGA hardware implementation of the proposed AE-LSTM architecture. Deploying DNNs on resource-constrained devices requires hardware implementations that are energy efficient. Hardware architectures on resource-constrained devices and embedded systems face the constraint of computing resources, reduced memory, and memory bandwidth. Therefore, reducing the computation complexity is critical for the intended networks on these platforms. Proposed methods to reduce the computation complexity of DNNs include quantization, pruning, and special hardware modules. Quantizing weights or activations helps reduce the area of arithmetic units and memory requirement [17–19]. Additionally, pruning connections with small weight values using special training methods could help create networks with a small number of parameters [21], thus reducing the number of computations. Special hardware modules such as multipliers based on Look-Up Tables (LUTs) can also be used for area reduction in networks with low bit precision parameters [20].

This work presents a unified parallel solution for accelerating the proposed AE-LSTM network. Our goal in this work is to reduce the hardware complexity, data dependencies, and computation patterns and increase hardware utilization and throughput while considering network performance and power consumption. The proposed solution combines serial-parallel computation with matrix algebra concepts to reduce computational complexity and enhance the throughput. A systolic ring of outer product-based processing elements (PEs) and a single reusable activation function block (AFB) is adopted in the architecture. The outer product generates and accumulates partial sums in parallel, eliminating data dependencies and increasing hardware utilization and system throughput. Besides that, the proposed unified computing kernel can perform recurrent and non-recurrent fully connected layers (FC) computations, improve hardware utilization and support various applications, including the AE-LSTM network presented in this work. Furthermore, this work extensively studies the network performance and effectiveness in detecting different types of outliers potentially contained in the dataset. This paper makes the following contributions:

- Present a unified parallel solution that combines serial-parallel computation with matrix algebra concepts to accelerate recurrent and non-recurrent fully connected layers computations in the AE-LSTM network. The unified parallel solution is generic and could be utilized in other LSTM and feedforward neural network applications.
- Validate the presented architecture on the Xilinx PYNQ-Z1 development board using an open-source meteorological dataset collected using a multi-hop WSN.
- Present an extensive study of the network performance and effectiveness in detecting different types of outliers potentially contained in the meteorological dataset.

The rest of this article is organized as follows. Section 2 presents the background of Autoencoder and RNN LSTM networks. Section 3 gives an overview of the proposed Autoencoder-LSTM outlier detection system. Section 4 describes the accelerator architecture and implementation on the Xilinx PYNQ-Z1 development board. Section 5 discusses experimental results, and Section 6 concludes the article.

2 Background

2.1 Autoencoder Neural Network

Autoencoder neural networks are a common solution for dimensionality reduction and outlier detection. AEs are unsupervised learning techniques that leverage artificial neural networks for feature extraction and representation learning. Their primary objective is to reconstruct the inputs instead of predicting target variables. The AE network forces a compressed knowledge representation of the original information by imposing a bottleneck in the network architecture. If any structure exists in the input data, it could be learned and compressed into a compact, latent representation. A general structure of the AE network with three hidden layers is shown in Fig. 2. The neural network structure on both sides of the middle hidden layer is often symmetric. The input and output layers of the network have the same size. Each input x_i is reconstructed to $\hat{x}_i$ for the *ith* dimension. The projection of inputs to reduced representation (code) is termed *encode*, while *decode* reconstructs the outputs from the reduced representation. The network is trained by minimizing the aggregated reconstruction error in all d-dimensions. The reconstruction error $\mathcal{L}(x, \hat{x})$ measures the difference between the original input x and the network reconstructed version $\hat{x}$. Using the reconstruction error to penalize the network forces the network to learn the most important attributes in the input data and how to best reconstruct the original input from the reduced representation (code). With the use of nonlinear activation functions at each layer, the Autoencoders can learn nonlinear relationships, making them a more powerful (nonlinear) generalization to PCA.

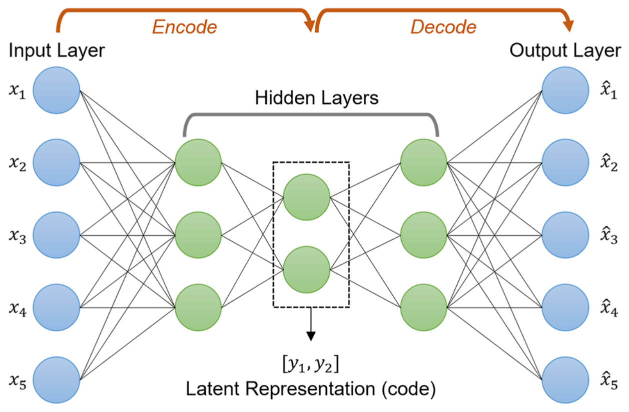

Figure 2 A general structure of autoencoder network with three hidden layers. The projection of inputs to latent representation (code) is termed encode, while decode reconstructs the outputs from the latent representation.

To further improve the Autoencoder model and balance the network sensitivity to input variations, a regularization term is introduced into the loss function $\mathcal{L}(x,\hat{x})$. The newly constructed loss function consists of two terms, the reconstruction loss and a regularizer.

$$\hat{\mathcal{L}}(x,\hat{x}) = \mathcal{L}(x,\hat{x}) + \alpha \cdot regularizer \tag{1}$$

The reconstruction loss $\mathcal{L}(x,\hat{x})$ will ensure that the network model is sensitive enough to the input and can build an accurate reconstruction. In contrast, the regularizer will discourage memorizing and overfitting the training data. The scaling parameter α added in the front of the regularizer term is called the regularization rate and is used for trading off the two terms' objectives. This work utilizes the Mean Absolute Error (MAE) as a loss function and weight decay $L2$ regularizer. Additional details are provided in the network training section.

2.2 LSTM Recurrent Neural Network

Long short-term memory networks are a special kind of recurrent neural network capable of learning long-term dependencies, making them suitable for time series analysis. They have the form of a chain of repeating modules called LSTM cells, shown in Fig. 3. Each LSTM cell consists of layers of neural networks, internal states, and point-wise operations.

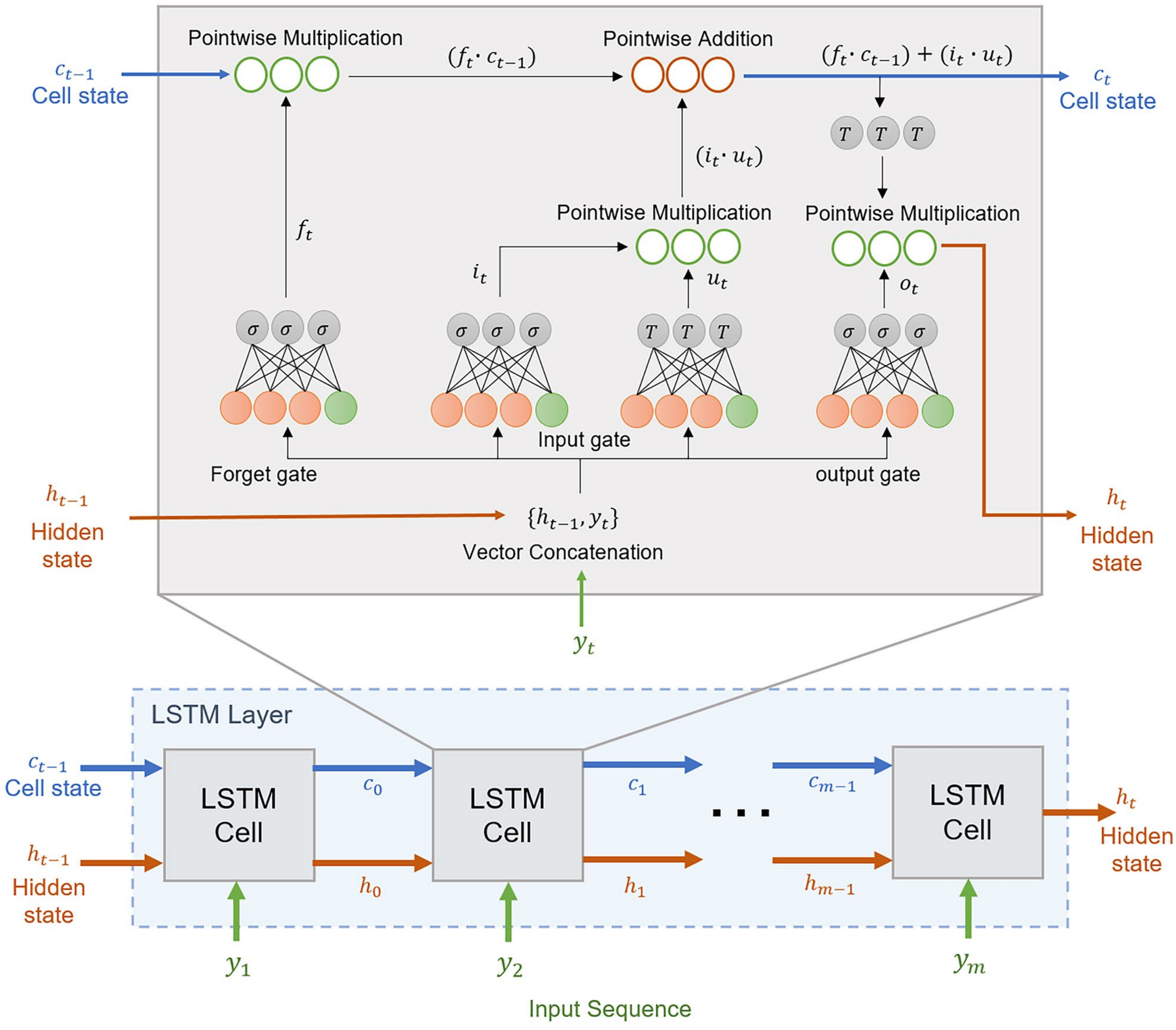

Figure 3 LSTM layer architecture and a single LSTM cell details. LSTM Layer has the form of a chain of repeating modules called LSTM cells. Each LSTM cell consists of layers of neural networks, internal states, and point-wise operations. The σ represents the sigmoid activation function while T the tanh activation function.

The key to LSTM networks is the cell state, c_t, which could be viewed as the extracted information from the input sequence at each time step. The LSTM cell can add or remove information to the cell state using Gate structures. Gates are ways to let information through optionally. They are composed of sigmoid and tanh neural net layers and point-wise operations. The LSTM cell has three main gates; forget, input, and output. Forget gate determines the fraction of history information to forget by multiplying the value of the cell state, c_t, by a number between 0 (delete) and 1 (keep everything). The multiplication value is determined by the current input, x_t, and the LSTM cell hidden state from the previous time step, h_{t-1}. The input gate has two parts; the tanh layer, which creates a vector of new candidate values, u_t, and the sigmoid layer, which decides the amount of new candidates to be added to c_t. The LSTM cell hidden state and also the LSTM cell's output, h_t, is a manipulated version of c_t. The cell state, c_t, is first passed through a tanh layer to push the values between -1 and 1, then multiplied by a number between 0 (no outputs) and 1 (preserve output) generated using the output gate structure. The size of the LSTM cell is defined by the number of elements in the hidden state, h_t, and the number of input features per time step (number of features in x_t). The computations in a single LSTM cell with n hidden state units and m-dimensional input features are described using the following set of equations:

$$\begin{aligned}
f_t &= sigmoid(U_f h_{t-1} + W_f x_t + b_f) \\
i_t &= sigmoid(U_i h_{t-1} + W_i x_t + b_i) \\
u_t &= tanh(U_u h_{t-1} + W_u x_t + b_u) \\
o_t &= sigmoid(U_o h_{t-1} + W_o x_t + b_o) \\
c_t &= (f_t \cdot c_{t-1}) + (i_t \cdot u_t) \\
h_t &= o_t \cdot tanh(c_t)
\end{aligned} \tag{2}$$

where $f_t, i_t, u_t, o_t \in \mathbb{R}^n$ are the outputs of the forget gate, the input gate, and the output gate, respectively. As described earlier, the c_t and h_t are the cell state and the hidden state/output of the LSTM cell. They are initialized to zero and updated at each time step, demonstrating the "recurrent" nature of LSTM. $W_j \in \mathbb{R}^{nxm}$, $U_j \in \mathbb{R}^{nxn}$ and $b_j \in \mathbb{R}^n$ $(j = f, i, u, o)$ are weight and bias parameters learned during the training process.

The LSTM network could be a single or multi-layer network. The network could be layered or stacked by connecting the LSTM layer cells' hidden state to the input of the following LSTM layer cells. For final processing, the hidden state, h_t, of the last layer is often connected to a non-recurrent fully connected layer described using the following equation:

$$y_t = AF(W_y h_t + b_y) \tag{3}$$

where W_y is the weight matrix, b_y is a bias vector, h_t is the hidden state of the last LSTM layer, and AF is the activation function used in the layer.

3 Proposed System Architecture

3.1 System Overview

The proposed system integrates a fine-tuned Autoencoder and a Long short-term memory (LSTM) neural network for real-time outlier detection in time series data. The flow in the proposed architecture is illustrated in Fig. 4. The sensor data is preprocessed, buffered, and then fed to the AE for dimensionality reduction and feature extraction. The preprocessing step involves data normalization, specifically minimax normalization. The normalization step is crucial for the subsequent AE and LSTM networks to find trends and patterns in the sensor data. The sensor data buffer is a sliding window to update the inputs fed to the AE at each time step. Generally, in the case of time-series data, a past history window is used to analyze outliers. The AE generates an m-dimensional code using d-dimensional buffered sensor data, $d \gg m$. The LSTM predictor forecasts the next time step using the AE m-dimensional code. Given that in time series data, the values in consecutive time steps do not change significantly or change smoothly, the forecasted time step p_i out of the LSTM predictor is compared against the new sensed value x_{i+1}. If the absolute difference between

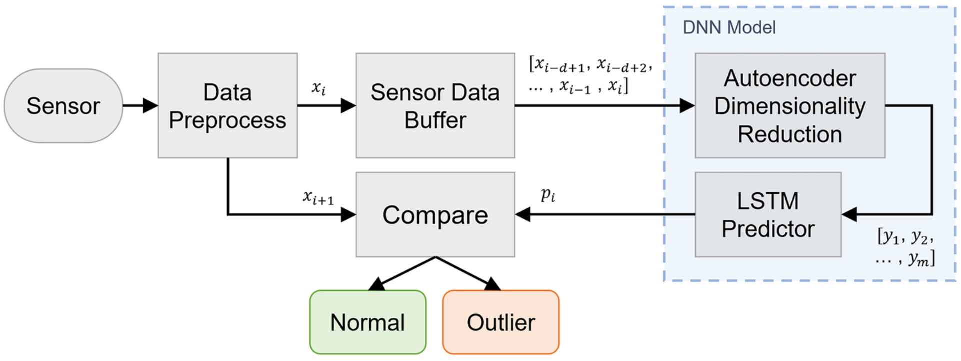

Figure 4 The general architecture of the proposed deep neural network outlier detection system.

the predicted value and the new sensed value is sufficiently high, the new sensed value is flagged as an outlier.

3.2 Network Training

Training the proposed AE and LSTM networks are done using the Keras library running on top of the TensorFlow framework [12]. The Grand St. Bernard open-source dataset [9] is used to demonstrate the performance of the proposed approach. The dataset consists of temperature measurements and other metrological characteristics of the environment collected for two months from multiple wireless sensor nodes deployed at the Grand St. Bernard pass, located between Switzerland and Italy. The network setup consisted of 23 sensor nodes deployed in two clusters, as shown in Fig. 5. The larger cluster consisted of 18 nodes, while the small one had five. Since this work focuses on outlier detection in univariate series, the temperature measurements from the same dataset are only considered.

The Autoencoder and the LSTM networks are trained, validated, and tested using the overlapping windows extracted from the temperature measurements in both clusters' sensor nodes. A 3 *hours* window (90 *samples*) is selected to capture the increase and decrease trends in ambient temperature measurement while maintaining acceptable computational complexity of both the AE and the LSTM networks. The training set size is [169580, 90], while the validation and test sets are [36600, 90] each. The 70 : 30 ratio is used for dataset separation. Each window of length 90 is treated as a 90-dimensional data point. The Autoencoder is first trained using the normalized extracted overlapping windows. The trained AE model is then used to generate the low-dimensional code to train the LSTM predictor. Therefore, the LSTM predictor training process did not include the AE decoding part. The autoencoder encoding part consists of two dense layers (AE1, AE2), while the LSTM predictor comprises an LSTM layer followed by two fully connected layers (FC1, FC2). The autoencoder network is trained for 500 epochs using the MAE loss function and L_2 regularizer with regularization rate $\alpha = 5 \times 10^{-5}$. On the other hand, the LSTM network is trained for 200 epochs using the MAE loss function. Adam optimizer is used to update both network parameters with a learning rate of 1×10^{-4}. A grid search across multiple parameters is used to find the selected regularization and learning rates. As a last step in the network training process, the AE and the LSTM networks are stacked, as shown in Fig. 6. The dense layers AE2, FC1, and FC2, are fine-tuned to improve the prediction performance. Table 1 shows a summary of the Keras TensorFlow model that gave the desired results in terms of performance and computational complexity.

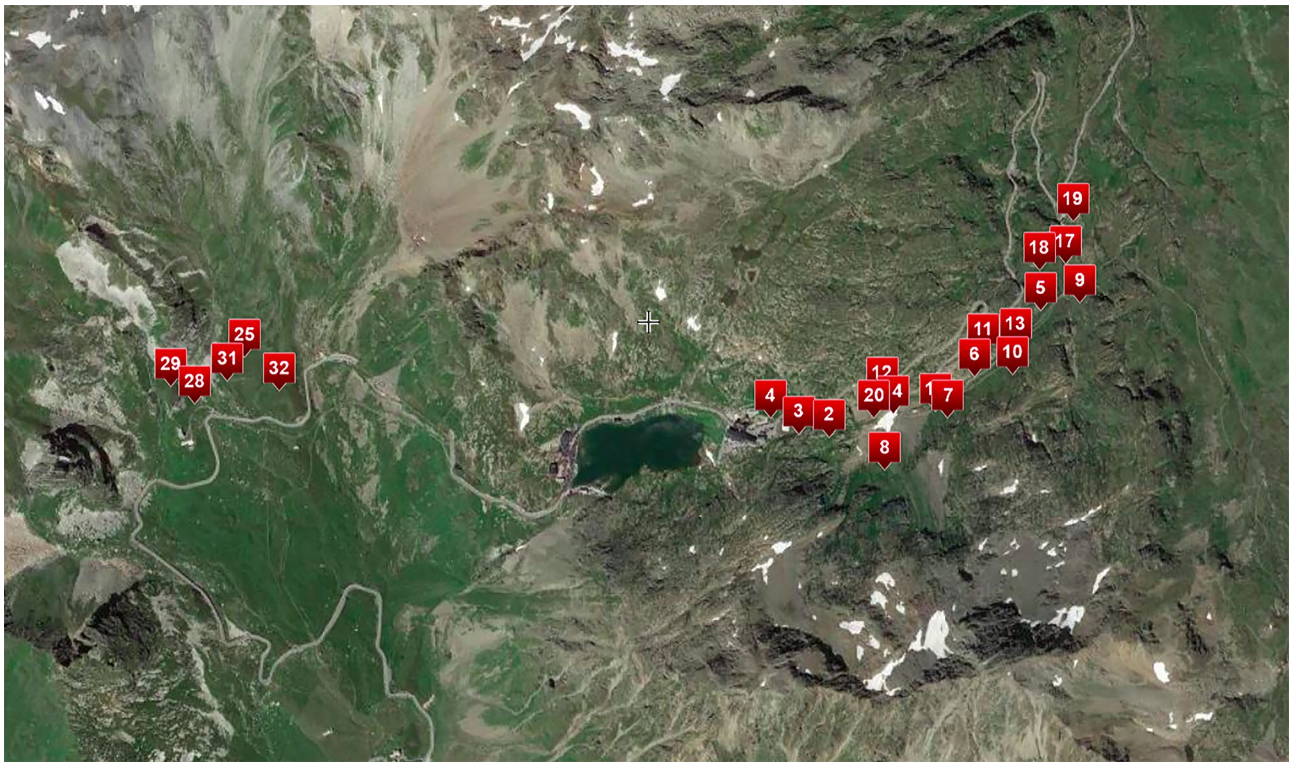

Figure 5 The setup of the wireless sensor network deployed at the Grand St. Bernard pass, Switzerland.

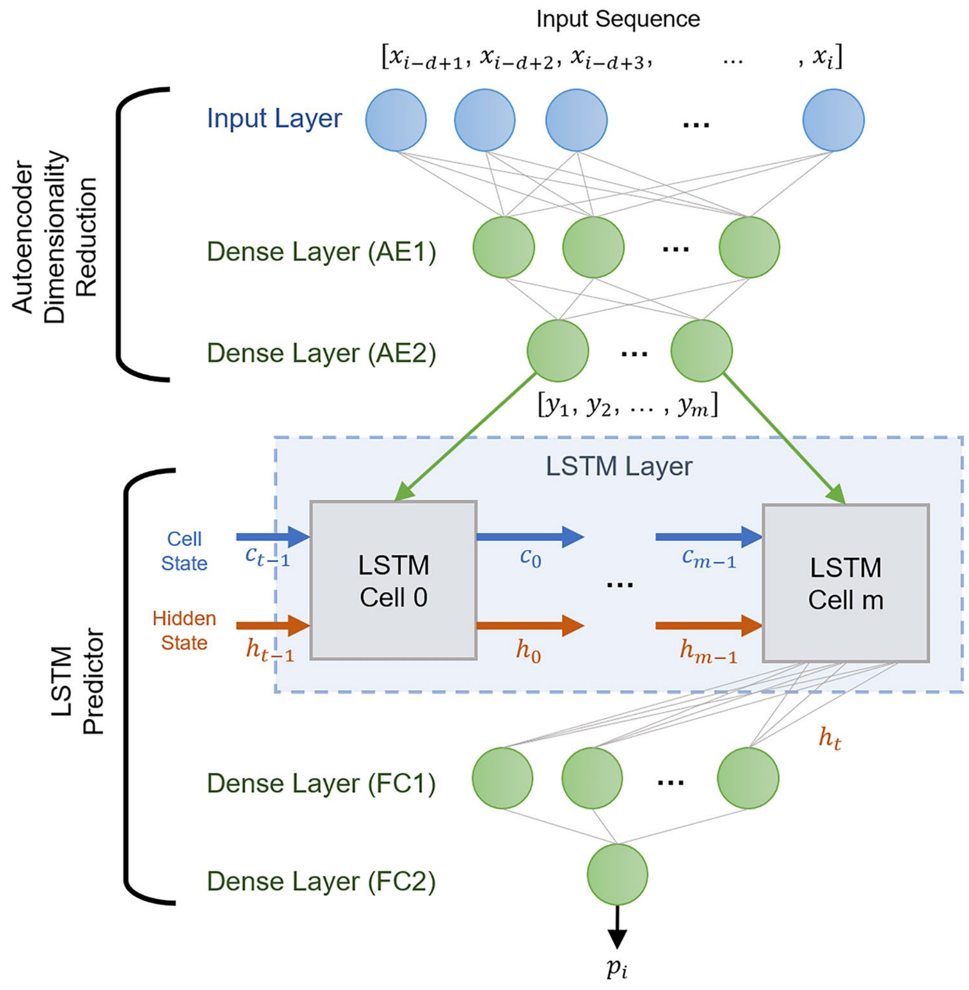

Figure 6 The general structure of the fine-tuned AE-LSTM network.

4 Accelerator Design and Implementation

The proposed real-time DNN-based outlier detection system aims to achieve low complexity, latency, and power consumption making it a viable data analysis solution for resource-constrained edge devices. The primary focus of this work is to optimize the FPGA hardware implementation of the proposed system. Figure 7 shows a simplified block diagram of the designed hardware accelerator. The main modules consist of a central computing module, an input buffer, an internal results memory unit, and a control module. The central computing module consists of a unified parallel architecture that accelerates the inference of the LSTM cells and the dense fully-connected layers. The unified parallel computing module consists of an array of computationally independent processing elements, a parallel-in serial-out shift register (PISO), and a single activation function block (AFB). The outer product generates and accumulates partial sums in parallel, eliminating data dependencies and making it possible to use a single activation function block to perform nonlinear network computations. The PISO shift register interfaces the PEs and the AFB. The AFB consists of sigmoid and tanh activation functions implemented using piecewise-linear approximation (PLA). Distributed BRAMs are used to store internal network results and implement the input buffer that will serve as a sliding window to update the input to the unified parallel computing module. The control module encodes instructions and controls data movements and storage. The implemented architecture is generic and could be extended in terms of network layers and hidden units per layer as long as hardware resources are available.

Table 1 Summary of fine-tuned AE-LSTM network model.

Layer	*Type*	*Output Shape*	*Activation*	*Param #*
Input	Input	(90,1)	tanh	0
AE1	Dense	(60,1)	tanh	5460
AE2	Dense	(30,1)	tanh	1830
LSTM	LSTM	(40,1)	Sigmoid, tanh	6720
FC1	Dense	(20,1)	tanh	820
FC2 (Output)	Dense	(1,1)	tanh	21
Total:				14,851

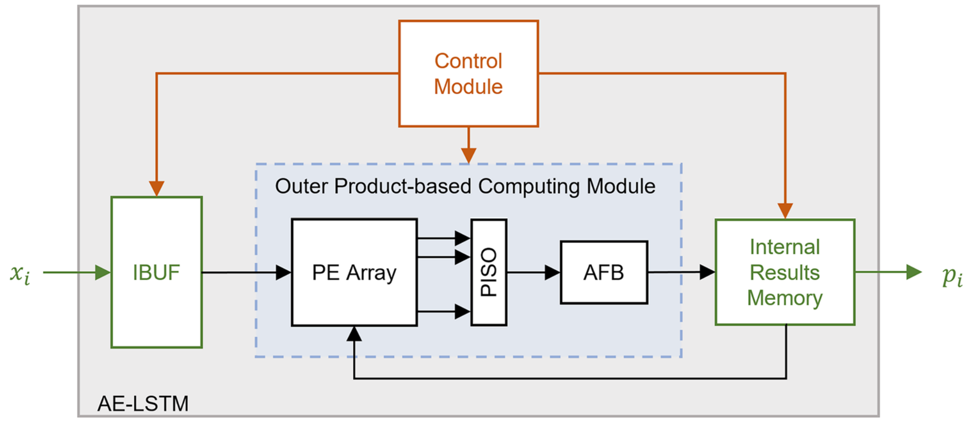

Figure 7 A simplified block diagram of the hardware accelerator.

4.1 Systolic Outer Product-based Computing Module

The proposed unified parallel architecture accelerates the inference of the LSTM layer in addition to dense fully-connected layers. Given that the LSTM layer latency dominates other layers' latency, the optimization efforts are directed toward accelerating the LSTM layer computations. The main computations effort in the LSTM layers, as described earlier in Section 2, comes from the computations of the LSTM cell gates, cell state, and hidden state. Each LSTM gate requires two MACs and element-wise vector additions. However, given that the weight matrices $W_j \in \mathbb{R}^{nxm}$, $U_j \in \mathbb{R}^{nxn}$ and the bias vector $b_j \in \mathbb{R}^n$ in each gate share the same first dimension n, it is possible to combine them into one matrix of dimension $(n \times (n + m + 1))$, where n is the number of hidden state units and m represent input features dimension. Likewise, since all the gates share the same dimensionality and input vector, it is possible to merge the gates' combined matrices into one big matrix of size $((4n) \times (n + m + 1))$. Thus, rather than optimizing four matrix-vector multiplications, each time step would focus on optimizing a single large matrix-vector multiplication (MxV), as in dense fully-connected layer computations. In general, the matrix-vector multiplication of a matrix $W \in \mathbb{R}^{nxm}$ by vector $x \in \mathbb{R}^m$ could be optimized in two forms:

- Inner product-based: Multiply in parallel all elements of the input vector x by the matrix row vectors W_i. Such a structure requires m multipliers and an adder tree. In addition, the process should be repeated for all n rows of the matrix W. The elements of the output vector in this configuration are computed sequentially.
- Outer product-based: Multiply in parallel a single element of the input vector x by the matrix column vectors W_j. Such configuration requires n MAC units. Besides, the process should be repeated for all m columns of the matrix W. The output vector elements in this structure are computed in parallel.

The conventional designs of the MxV used in most of the existing DNN network architectures are of the inner product-based option. The main drawback of such a structure is the hardware pipeline stall time imposed by the recurrent nature of the LSTM networks and the data dependencies between the output vector of the current time step and the input vector of the next time step. In such a case, the system must wait for the newly computed hidden state, h_t, before starting the subsequent step computations. This indicates that the whole system pipeline must be drained before starting the subsequent time step matrix-vector multiplication. Pipeline latency is critical to achieving a high throughput system. Therefore, our proposed scheme adopted the outer product-based approach to reduce the hardware stall time and the data dependencies between different time-step computations. To illustrate the proposed scheme's computations, the matrix-vector multiplication in (4) shows a simplified example of an m-dimensional input vector with four hidden-state units in a single time step. The matrix W represents the combined parameters matrix described earlier. Each row in W represents the weight of one hidden unit, and the first column includes all units' biases. After $(m + 1)$ computations, the resulting vector y contains the sum of the products of each hidden unit. Using as many multipliers as hidden units working in parallel, the partial sum of all the units will be simultaneously computed for each element in the input vector; an example of a single partial sum is boldly marked in (4). Similarly, each unit's final sum of products could simultaneously be obtained using an arithmetic accumulator per unit.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_0 & \mathbf{w_{00}} & w_{01} & \dots & w_{0m} \\ b_1 & \mathbf{w_{10}} & w_{11} & \dots & w_{1m} \\ b_2 & \mathbf{w_{20}} & w_{21} & \dots & w_{2m} \\ b_3 & \mathbf{w_{30}} & w_{31} & \dots & w_{3m} \end{bmatrix} \begin{bmatrix} 1 \\ \mathbf{x_0} \\ x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_0 + \mathbf{w_{00}x_0} + w_{01}x_1 & \dots & w_{0m}x_m \\ b_1 + \mathbf{w_{10}x_0} + w_{11}x_1 & \dots & w_{1m}tx_m \\ b_2 + \mathbf{w_{20}x_0} + w_{21}x_1 & \dots & w_{2m}x_m \\ b_3 + \mathbf{w_{30}x_0} + w_{31}x_1 & \dots & w_{3m}x_m \end{bmatrix} \tag{4}$$

After obtaining the result of the hidden units, the output values are stored in memory to be evaluated by the activation function before scheduling the computations of the subsequent time step. Given the serial processing nature of the input vector, a single activation functions block can serve all the hidden units forming a systolic ring topology. Accordingly, the subsequent time-step computations could start without waiting for the system pipeline to be drained. Additionally, the hardware could iteratively be reused to compute all the time steps, reducing the hardware complexity and power consumption.

A simplified block diagram of the proposed systolic outer product-based architecture is shown in Fig. 8. The main modules consist of an array of computationally independent processing elements for MxV, a parallel-in serial-out shift register, and a single activation function block.

Each processing element consists of two main parts: a single port Block RAM (BRAM) and a MAC unit. The BRAMs hold the stationary network parameters (weights, biases), and the MAC units perform the MxV computations. Since the available external memory bandwidth on low-power IoT and edge devices is limited, the distributed on-chip BRAMs are used to enable the concurrent operation of the PEs. In this case, the external I/O communication is substantially reduced to only loading new features, sharing results, and updating the network parameters when needed. In addition, to maintain a regular structure and reduce memory addressing complexity, equal-sized distributed BRAMs are used. Each PE BRAM contains the weight values of one neuron unit per layer. The network input data are sequentially introduced and multiplied concurrently by their corresponding weights in each PE. The accumulator register in each PE holds the sum of products of the input data by the corresponding weights. The final sums of products from all PEs are passed to the PISO shift register to be shifted through the activation function block. The activation function block is a configurable activation block. It handles the dense fully-connected layer activations, LSTM gates activation, the cell state update, and the hidden state computations.

4.2 Activation Function Block

4.2.1 Nonlinear Activation Functions

The nonlinear activation function is one of the main components of the artificial neural network computational units. Each neuron in the hidden and output layers needs an activation function. The output of the MAC units passes through the nonlinear activation function to compute the final output of each neuron. Logistic sigmoid and hyperbolic tangent (tanh) functions are the most widely used activation functions. In addition, they are the primary activation functions in the LSTM cell. The logistic sigmoid output is defined in the [0, 1] range, while the tanh is in the $[-1, 1]$ range. Mathematically, the sigmoid and tanh functions are defined as,

$$\begin{aligned} tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ sigmoid(z) &= \frac{1}{1+e^{-z}} \end{aligned} \tag{5}$$

The straightforward implementation of these functions is costly, given that both of them require computing the exponential and division. Various methods have been proposed for implementing activation functions on hardware. These methods generally fall

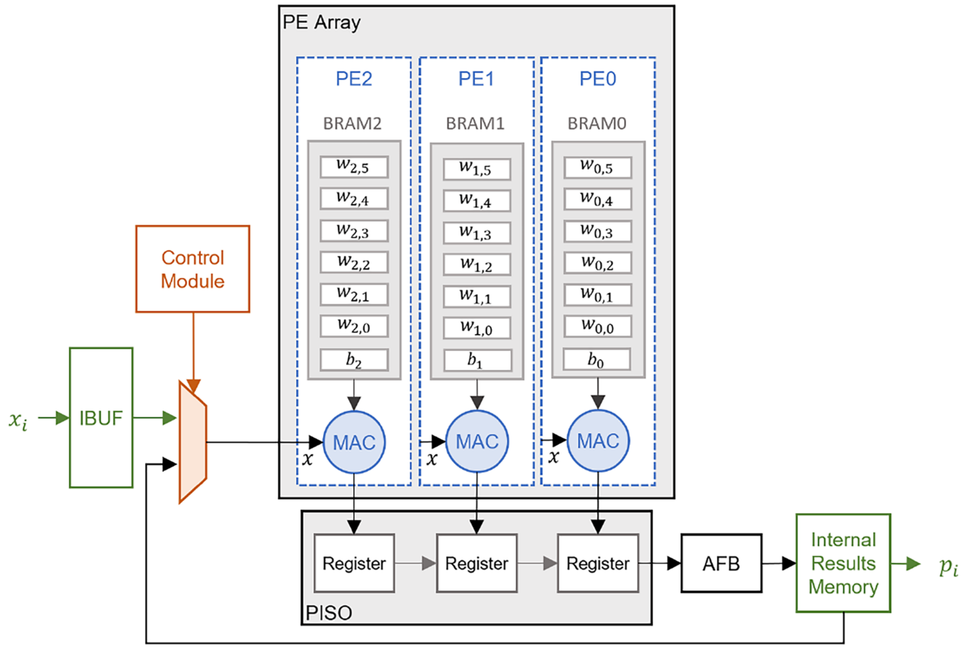
PE Array
PE2
PE1
PE0
BRAM2
BRAM1
BRAM0
$w_{2,5}$ $w_{2,4}$ $w_{2,3}$ $w_{2,2}$ $w_{2,1}$ $w_{2,0}$ b_2
$w_{1,5}$ $w_{1,4}$ $w_{1,3}$ $w_{1,2}$ $w_{1,1}$ $w_{1,0}$ b_1
$w_{0,5}$ $w_{0,4}$ $w_{0,3}$ $w_{0,2}$ $w_{0,1}$ $w_{0,0}$ b_0
Control Module
x_i
IBUF
MAC
x
Register
PISO
AFB
Internal Results Memory
p_i

Figure 8 A simplified block diagram of the systolic outer product-based computing unit. The main modules consist of an array of processing elements (PEs) for MxV, a parallel-in serial-out shift register (PISO), and a single activation function block (AFB).

into two main categories, piecewise-linear approximation and look-up table-based (LUT) approaches. This work adopts the PLA approach to achieve high-fidelity approximations without sacrificing the network performance. Given the symmetrical nature of both activation functions, the approximation in this work focused on the positive side and interpreted the results for the opposing side by flipping and rotating. In addition, given that the sigmoid function converges to 1 for inputs greater than 5, the input range [0, 5] is only considered in the approximation process. Similarly, the input range [0, 3] is considered for the tanh activation function approximation. An 11-line approximation is used for the logistic sigmoid, while the hyperbolic tangent requires a 16-line approximation to give a comparable MAE. Each line is described using $y = Sx + B$, where S controls the slope of the line and B is the function intercept. The approximation performance is plotted in Fig. 9. The top part of Fig. 9 shows the similarity in shape between the real-valued functions using (5) and the PLA approximation for both sigmoid and tanh activations. The bottom part of the same figure shows the absolute error of the approximations. The mean absolute error of both approximations is in the order of $\sim 2.5 \times 10^{-4}$.

Implementing the adopted PLA approach achieves low latency and requires low hardware resources. The computation process requires a 3 clock cycle pipeline delay, and the resource utilization, i.e., the amount of LUTs, registers, and DSP blocks, is negligibly small in both activation functions. Therefore, the adopted PLA approach is a viable solution for low-latency applications on resource-constrained devices.

4.2.2 LSTM Cell Activation Pipeline

As mentioned earlier, the activation function block is a configurable activation block that handles the dense fully-connected layer activation, the LSTM gates activation, the cell state update, and the hidden state computations. To reduce the implementation complexity and enable activation function and arithmetic units reuse, the LSTM gates activation and point-wise operations are pipelined. Figure 10 shows the data flow in the LSTM cell activation six stage pipeline, $S0 - S5$. The LSTM cell formulation shown in (2) is divided into six steps; each corresponds to one pipeline stage. The inputs to the activation pipeline are the matrix-vector multiplication results of the forget gate, input gate, and output gate shown in Fig. 3 and expressed in the LSTM cell formulation shown in (2).

4.3 Data Representation

The most used software frameworks for Deep learning perform inference by adopting the floating-point representation to ensure the best accuracy. However, considering hardware implementation on resource-constrained devices, floating-point arithmetic is not optimal for resource usage. Therefore, this work uses fixed-point representations to ensure a suitable precision for the computations and low resource usage. A software version of the AE-LSTM network exploiting the fixed-point toolbox included in the python environment is used to test different fixed-point configurations and evaluate

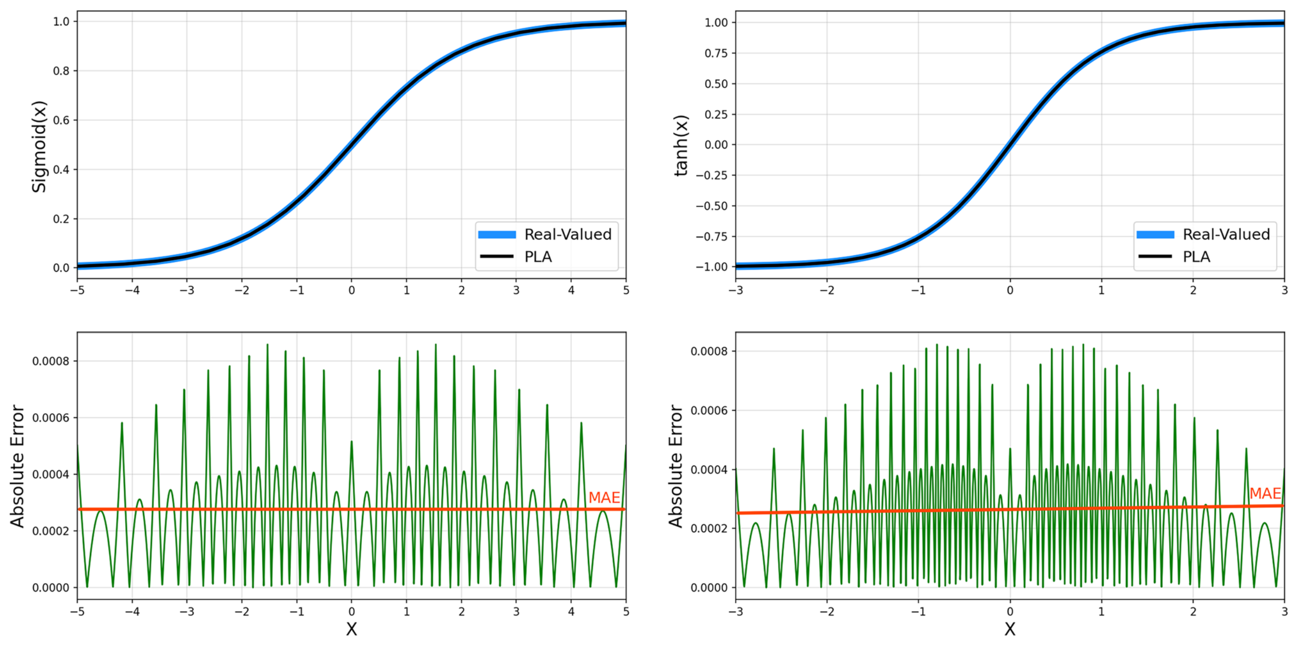

Figure 9 Activation functions approximation performance. The left plots show the similarity in shape between the real-valued sigmoid using (5) and the PLA and the absolute error of the approximation. The right plots show the same but for the tanh activation function.

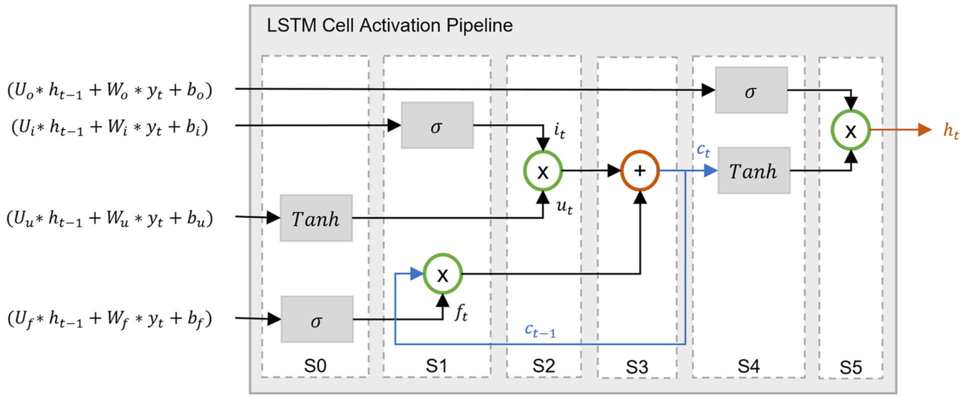
LSTM Cell Activation Pipeline
$(U_o * h_{t-1} + W_o * y_t + b_o)$
$(U_i * h_{t-1} + W_i * y_t + b_i)$
$(U_u * h_{t-1} + W_u * y_t + b_u)$
$(U_f * h_{t-1} + W_f * y_t + b_f)$
σ
$Tanh$
i_t
u_t
f_t
c_t
c_{t-1}
h_t
S0
S1
S2
S3
S4
S5

Figure 10 Data flow in the LSTM cell activation pipeline. The inputs to the activation pipeline are the matrix-vector multiplication results of the forget gate, input gate, and output gate. The outputs are the cell state c_t and the hidden state h_t.

the error between the floating-point-based implementation and the fixed-point one. The performed experiments showed that the optimal solution is obtained using an 18 bits representation, 4 bits for the signed integer part, and the remaining 14 bits for the fractional part. All network parameters and normalized input vectors are quantized into the same 18 bits Q4.14 fixed-point representation. The mean squared error between the floating-point and the fixed-point implementation is in the order of $\sim 10^{-4}$.

5 Experimental Results

5.1 Experimental Setup

Xilinx PYNQ-Z1 FPGA development board is used to analyze the performance of the proposed architecture on hardware [10]. The development board consists of an XC7Z020 ZYNQ series FPGA containing a Dual ARM Cortex-A9 core processor. In addition, it is a hardware platform for the PYNQ open-source framework, which comprises software running on the ARM CPUs and a base hardware library. Xilinx's Vivado tools are used to design the proposed system hardware accelerator [11]. After quantizing the trained network and activation function parameters into Q4.14 fixed-point representation, the parameters are extracted into a C++ header file and used in Xilinx's Vivado HLS tool to design the proposed custom accelerator hardware. After verifying the functionality of our custom-designed accelerator modules, the designed hardware accelerator is exported to Xilinx Vivado Design Suite to build the hardware overlay. The resulting hardware overlay is loaded on the PYNQ-Z1 board. A python code is developed to interface with the hardware overlay. The network parameters are loaded into the PL-distributed BRAMs using the AXI high-performance (HP) port. Then, a single sensor reading is transferred from the ARM core processing system (PS) to the accelerator using the AXI4-Lite interface at each time step. The accelerator outputs are extracted and evaluated offline to measure the performance.

5.2 Hardware Resource Utilization

The top-level diagram of the hardware implementation using Xilinx Vivado tools on the PYNQ-Z1 board is shown in Fig. 11. With reference to Fig. 4, the sensor data buffer, AE, and LSTM predictors are implemented on the programmable logic (PL), while the rest are implemented on the PS. Table 2 shows the PL resource utilization of the proposed accelerated fixed-point architecture. With reference to the initial design reported in [13], the LUT and LUTRAM utilization are reduced by $\sim 50\%$. The flip-flop utilization is reduced by $\sim 75\%$. Increased BRAM and DSP utilization are expected, given the parallel outer product-based implementation of the matrix-vector multiplication. The estimated response time of the accelerator using the global $100MHz$ clock that drives the PL from the PS is $0.22ms$. This average latency outperforms the one reported in the initial design in [13] by a factor of 5. Besides, the implemented design achieves $1GOPS$ throughput.

5.3 Power Measurement

The Xilinx Power Analyzer is used to estimate the dynamic and static power of the proposed architecture on the PYNQ-Z1 board. Table 3 shows the analyzer breakdown. As per the breakdown, the accelerator consumes $\sim 94mW$ on average. Accessing the DDR3 to load the network parameters to PL-distributed BRAMs is the most power-hungry, but this happens only when parameters are loaded for the first time or when an update is required. A USB power meter is used to verify the total power consumption reported by the analyzer. As shown experimentally in Fig. 12, the whole PYNQ-Z1 system board burns at most $\sim 1.54W$ when the accelerator is up and running. The reported power consumption is very close to the total noted in Table 3.

5.4 Outlier Detection Accuracy

The Grand St. Bernard dataset used in this work is unlabeled, and previous examples of interesting outliers are unavailable; therefore, it is an unsupervised scenario. The deviation of sensor

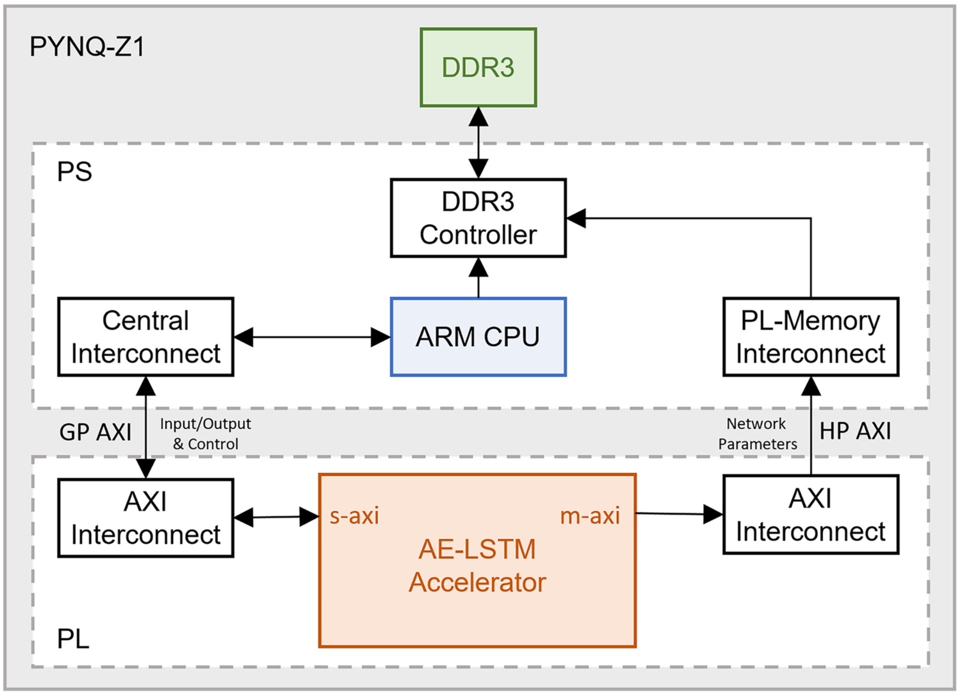

Figure 11 Top-level diagram of the proposed system architecture on PYNQ-Z1 board.

reading from the proposed model prediction is used to quantify each data point's level of outlierness. The deviation vector of the normalized training set is modeled to fit a Gaussian distribution, $X \sim \mathcal{N}(\mu, \sigma^2)$, as shown in Fig. 13. A sensor reading is flagged as an outlier if its likelihood $p \leq \tau$ and τ is set to 0.01 to give a sufficient confidence interval of 99%. The calculated deviation threshold to flag a data point as an outlier using the 99% confidence interval is 0.15. The calculated threshold is equivalent to 6 °C in the original scale. Figures 14, 15, 16, and 17 are four examples of the system response to sensor readings from the Grand St. Bernard dataset. The results are presented in the $[-1, 1]$ scale, which is the scale the AE-LSTM network uses for prediction. The results can be scaled back to the original sensor scale without affecting the detection accuracy. However, we wanted to present the results obtained directly from the hardware accelerator.

Figure 14 illustrates the system response to regular sensor readings from node three located in the large sensor network cluster shown in Fig. 5. The absolute difference between the scaled sensor reading x and the AE-LSTM system prediction p is lower than the predefined threshold, indicating that the gradual increase and decrease in the temperature measurements within 3 hours window is within the normal range. The second example, Fig. 15, presents the system response to noisy sensor readings from node 32, located in the small network cluster. The absolute difference between the sensor reading x and the AE-LSTM prediction p is slightly beyond

Table 2 Resource utilization of the proposed system implementation on PYNQ-Z1 board.

	LUT	*LUTRAM*	*FF*	*BRAM*	*DSP*
Available	53200	17400	106400	140	220
Floating-Point Implementation reported in [13]					
Utilization	24807	650	22585	30	111
Utilization %	46.63%	3.74%	21.23%	21.43%	50.45%
Fixed-Point Implementation reported in [13]					
Utilization	12544	448	11922	28.50	17
Utilization %	23.58%	2.57%	11.20%	20.36%	7.73%
Proposed Accelerated Fixed-Point Implementation					
Utilization	6172	231	2899	82.50	163
Utilization %	11.60%	1.33%	2.72%	58.93%	74.09%

Table 3 Proposed architecture power consumption breakdown on PYNQ-Z1 board using Xilinx power analyzer.

Part		*Power (W)*	*Percentage*
Dynamic Power			
PS	ARM Cortex-A9	0.27	83%
	DDR3	0.63	
	PLLs	0.344	
	Peripherals	0.012	
Accelerator		0.094	6%
Interconnect		0.009	1%
Static Power			
PL static		0.147	10%
Total:		1.506	

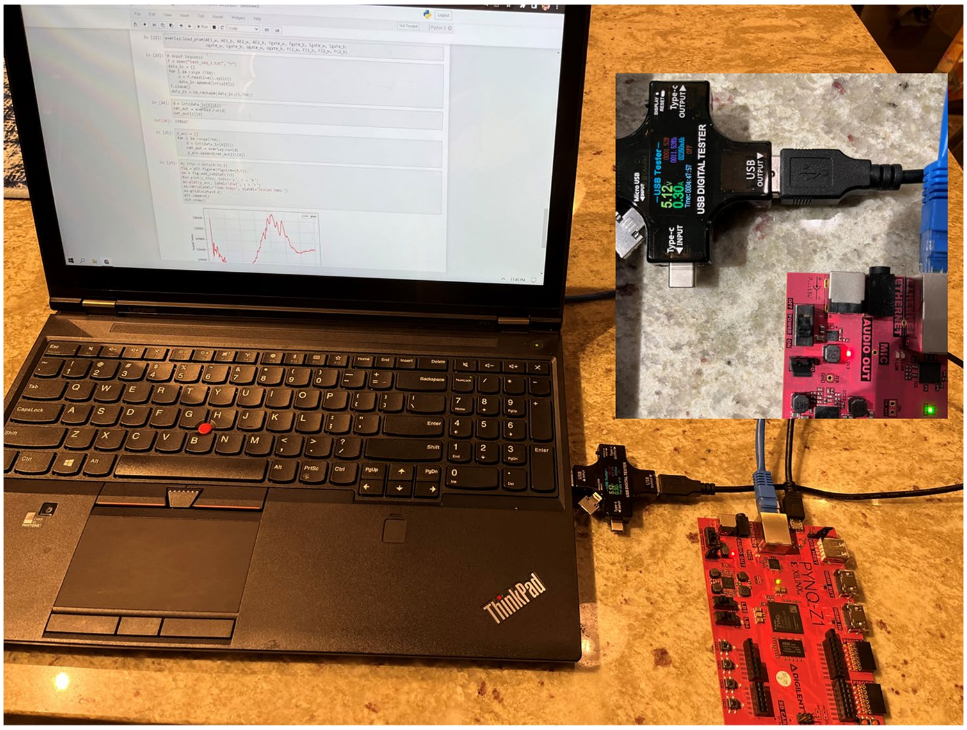

Figure 12 Total power measurement using a USB power meter. The whole PYNQ-Z1 system board burns at most $\sim 1.54W$ when the accelerator is up and running.

the predefined threshold around time steps 1137, 1172, and 1350. Given that the deviation is insignificant and the samples are not in consecutive time steps, these data points are considered weak outliers or noise.

In Fig. 16, we present the system response to abnormal sensor readings from node 7 in the large cluster. The absolute difference between the sensor reading x and the AE-LSTM prediction p is significantly beyond the predefined threshold at different parts of the plot. The first deviation region, time steps 0 to 50, is due to the input buffer warmup time. The input buffer is initialized to zero, and as the sensor readings are passed to the input buffer, the prediction gets closer to the actual measurements. The warmup time is usually shorter, as in Figs. 14 and 15 when the input sensor readings are close to average readings in the input buffer. In the second region, time steps 1250 to 1500; the readings are within the normal scaled temperature range. However, they are not maintaining the gradual increase or decrease shape expected in the standard ambient temperature measurements. Therefore, the prediction system flags them as outliers. In the third region, around time step 1650, the input sensor readings returned to normal. However, given the difference between the input buffer average and the sensor readings, the system treated the readings as outliers until a gradual change was discovered. The last region

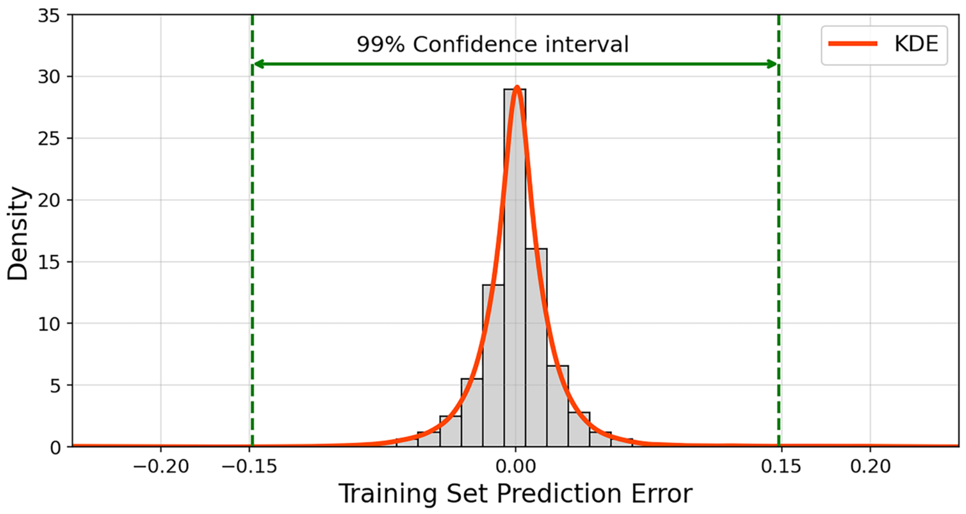

Figure 13 Histogram of the training set prediction error with fitted Kernel Density Estimation (KDE) using Gaussian kernel with bandwidth = 0.004. The $\mu \sim 0$ and the $\sigma^2 \sim 0.06$.

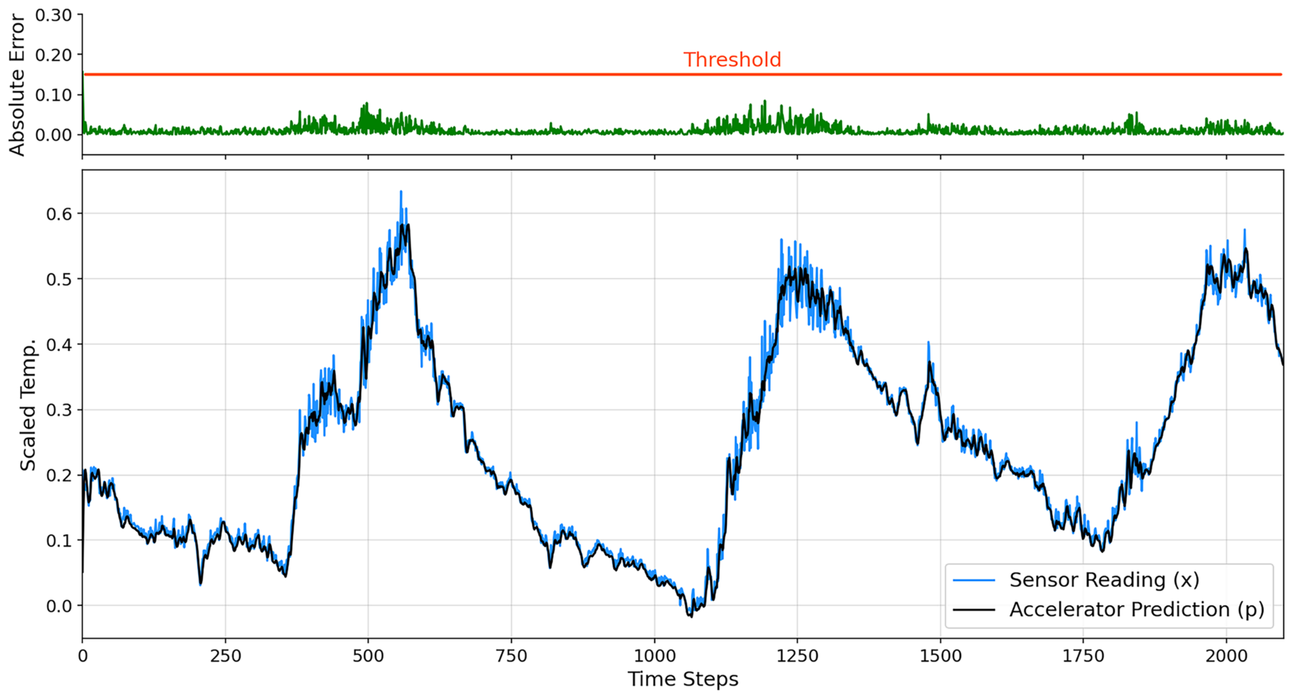

Figure 14 System response to regular sensor readings. Actual sensor readings (blue) and corresponding Accelerator predictions (black). The absolute difference between the sensor observations and the system prediction is below the predefined threshold indicating a regular sequence.

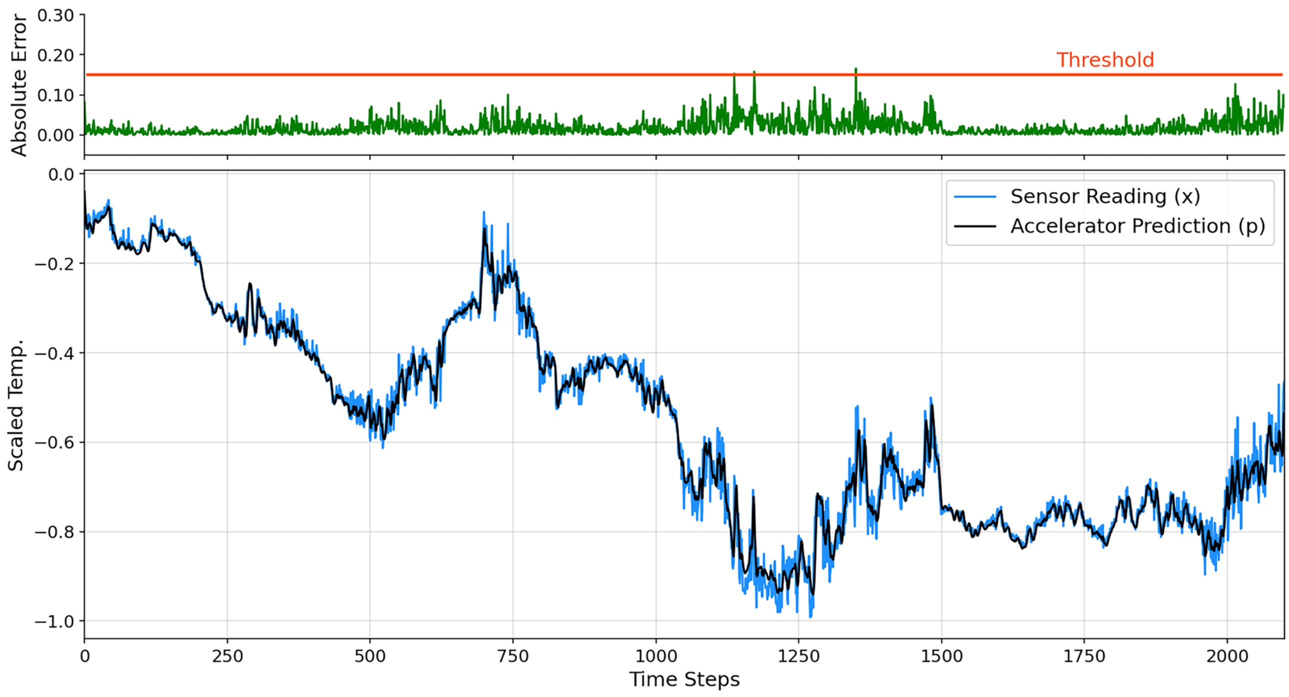

Figure 15 System response to noisy sensor readings. The absolute difference is slightly beyond the predefined threshold around time steps 1137, 1172, and 1350. These data points are considered weak outliers or noise.

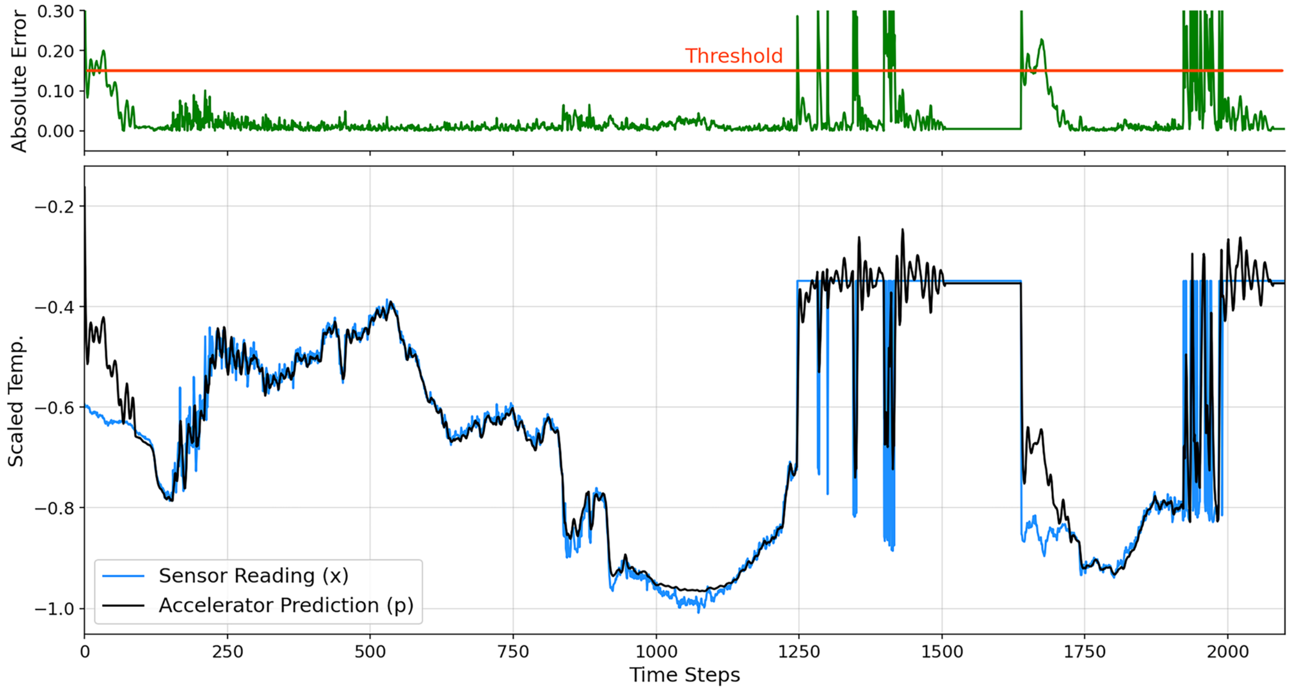

Figure 16 System response to abnormal sensor readings. The absolute difference between the sensor readings and the proposed system prediction goes beyond the predefined threshold, indicating an irregular sequence. The regions identified as outliers are due to random patterns or the out-of-context increase and decrease in the sensor readings.

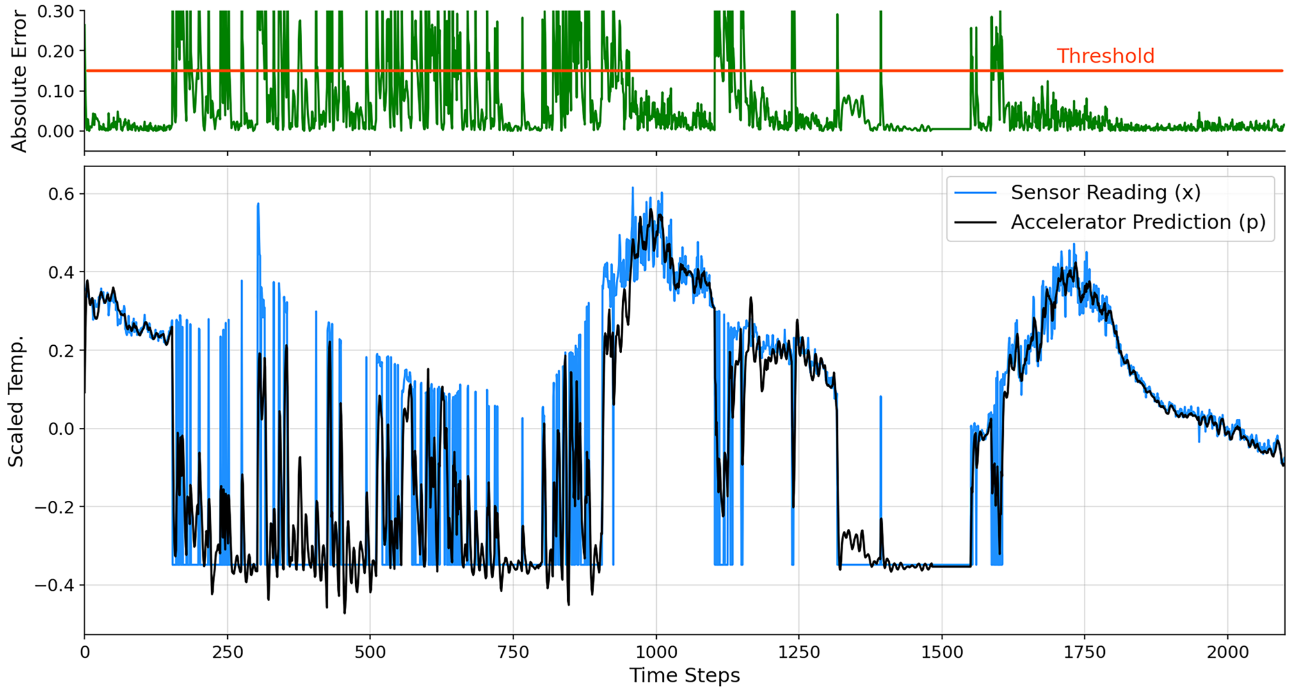

Figure 17 Another example of the system response to abnormal sensor readings. The regions identified as outliers are due to the random patterns in the sensor readings, as in the region from time step 150 to 950, or the out-of-context increase and decrease in the sensor readings, around time step 1150, as an example.

is the same as the second region in terms of interpretation. Figure 17 is another example of abnormal sensor readings obtained from sensor node 29 in the small cluster. The interpretation of the results in Fig. 17 is the same as in Fig. 16. The regions the system identified as outliers are either due to random patterns or the out-of-context increase and decrease in the sensor readings.

6 Conclusion

This work introduced an FPGA-based Deep Neural Network architecture for real-time outlier detection in time series data. The presented architecture integrates an Autoencoder and a Long short-term memory network to predict and detect outliers in real-time. The architecture computational complexity and throughput are improved using serial-parallel computation and matrix algebra concepts. A unified computing kernel is designed to perform recurrent and non-recurrent fully connected layers computations, improve hardware utilization and support various applications, including the AE-LSTM network presented in this work. In addition, the designed kernel is generic and can be extended in terms of the number of LSTM layers and cells per layer as long as hardware resources are available. An open-source meteorological dataset is used to validate the effectiveness of the design in detecting outliers in real-time. Experimental results on the Xilinx PYNQ-Z1 development board achieved low average latency and power consumption, making the proposed solution suitable for resource-constrained edge platforms.

Author Contributions Both authors contributed to the study conception and design.

Funding This work was supported in part by the US NSF under grant CNS-2016727.

Data Availability The dataset analysed during the current study are available in the Zenodo repository. https://urldefense.com/v3/__https://doi.org/10.5281/zenodo.2654726__;!!NLFGqXoFfo8MMQ!qpTJPWL5LhIFyH93DgHuI-vIX-0Yqoczd9YfMmyBAXohn9PMvJTCc4y5xpWxvQz40VaGq8guaRzcFnwHwuxDwHbIirkJ8xcVHzI$

Declarations

Conflicts of Interest The authors have no conflicts of interest to declare relevant to this article content.

References

1. Oliveira, L. M., & Rodrigues, J. J. (2011). Wireless sensor networks: a survey on environmental monitoring. *Journal of Communications, 6*(2), 143–151.
2. Hua, G., Li, Y., & Yan, X. (2011). Research on the wireless sensor networks applied in the battlefield situation awareness system. *International Conference ECWAC, 2011*(April), 16–17.
3. Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. *ACM Computing Surveys*, *41*(3), 15:1–15:58.
4. Rajasegarar, S., Leckie, C., & Palaniswami, M. (2008). Anomaly detection in wireless sensor networks. *IEEE Wireless Communications*, *15*(4).
5. O'Reilly, C., Gluhak, A., Imran, M. A., & Rajasegarar, S. (2014, Third Quarter). Anomaly detection in wireless sensor networks in a non-stationary environment. *IEEE Communications Surveys and Tutorials*, *16*(3), 1413–1432. https://doi.org/10.1109/SURV.2013.112813.00168
6. Xie, M., Hu, J., Han, S., & Chen, H.-H. (2013). Scalable hypergrid k-NN based online anomaly detection in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems, 24*(8), 1661–1670.
7. Chen, P.-Y., Yang, S., & McCann, J. A. (2015). Distributed real-time anomaly detection in networked industrial sensing systems. *IEEE Transactions on Industrial Electronics, 62*(6), 3832–3842.
8. Zhang, Y., Meratnia, N., & Havinga, P. J. (2013). Distributed online outlier detection in wireless sensor networks using ellipsoidal support vector machine. *Ad hoc networks, 11*(3), 1062–1074.
9. Barrenetxea, G. (2019). Sensorscope Data [Data set]. Zenodo. https://doi.org/10.5281/zenodo.2654726
10. PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC. Digilent [online]. Retrieved June 6, 2013, from https://urldefense.com/v3/__https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpgasoc/__;!!NLFGqXoFfo8MMQ!qpTJPWL5LhIFyH93DgHuI-vIX-0Yqoczd9YfMmyBAXohn9PMvJTCc4y5xpWxvQz40VaGq8guaRzcFnwHwuxDwHbIirkJZkg6D6s$
11. Vivado Design Suite. Xilinx [online]. Retrieved June 6, 2013, from https://urldefense.com/v3/__https://www.xilinx.com/products/design-tools/vivado.html__;!!NLFGqXoFfo8MMQ!qpTJPWL5LhIFyH93DgHuI-vIX-0Yqoczd9YfMmyBAXohn9PMvJTCc4y5xpWxvQz40VaGq8guaRzcFnwHwuxDwHbIirkJWiUTbuQ$
12. Chollet, F., et al. (2015). *Keras*. Retrieved June 6, 2013, from https://urldefense.com/v3/__https://keras.io__;!!NLFGqXoFfo8MMQ!qpTJPWL5LhIFyH93DgHuI-vIX-0Yqoczd9YfMmyBAXohn9PMvJTCc4y5xpWxvQz40VaGq8guaRzcFnwHwuxDwHbIirkJLJLb30E$
13. Mohamed, N. A., & Cavallaro, J. R. (2021). Real-time FPGA-based outlier detection using autoencoder and LSTM. *2021 55th Asilomar Conference on Signals, Systems, and Computers* (pp. 1195–1199). https://doi.org/10.1109/IEEECONF53345.2021.9723300
14. Guo, W., Wang, J., & Wanga, S. (2019). Deep multimodal representation learning: A survey. *IEEE Access, 7*, 63373–63394.
15. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H et al (2014) Learning phrase representations using RNN encoder-decoder for statistical machine translation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). https://doi.org/10.3115/v1/d14-1179
16. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
17. Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., et al. (2017). ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (pp. 75–84). ACM.
18. Chang, A. X. M., & Culurciello, E. (2017). Hardware accelerators for recurrent neural networks on FPGA. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS), ser. ISCAS '17*.
19. Guan, Y., Liang, H., Xu, N., Wang, W., Shi, S., Chen, X., Sun, G., Zhang, W., & Cong, J. (April 2017). FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 152-159).

20. Shin D., Lee J., Lee J., & Yoo, H. (Feb 2017). 14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)* (pp. 240–241).
21. Han, S., Mao, H., & Dally, W. J. (2015). Deep Compression: Compressing deep neural networks with pruning, trained quantization and Huffman Coding, CoRR, vol. abs/1510.00149. [Online]. Preprint retrieved from http://arxiv.org/abs/1510.00149
22. Que, Z., Liu, Y., Guo, C., Niu, X., Zhu, Y., & Luk, W. (2019). Real-time anomaly detection for flight testing using autoencoder and LSTM. *International Conference on Field-Programmable Technology (ICFPT), 2019*, 379–382. https://doi.org/10.1109/ICFPT47387.2019.00072

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.