Design and Implementation of Autoencoder-LSTM Accelerator for Edge Outlier Detection

Nadya A. Mohamed and Joseph R. Cavallaro
Department of Electrical and Computer Engineering
Rice University
Houston, TX, USA

***Abstract*—Sensors are used to monitor various parameters in many real-world applications. Sudden changes in the underlying patterns of the sensors readings may represent events of interest. Therefore, event detection, an important temporal version of outlier detection, is one of the primary motivating applications in sensor networks. This work describes the implementation of a real-time outlier detection that uses an Autoencoder-LSTM neural-network accelerator implemented on the Xilinx PYNQ-Z1 development board. The implemented accelerator consists of a fine-tuned Autoencoder to extract the latent features in sensor data followed by a Long short-term memory (LSTM) network to predict the next step and detect outliers in real-time. The implemented design achieves $2.06\ ms$ minimum latency and $85.9\ GOp/s$ maximum throughput. The low latency and $0.25\ W$ power consumption of the Autoencoder-LSTM outlier detector makes it suitable for resource-constrained computing platforms.**

***Index Terms*—Accelerator, FPGA, embedded systems, Autoencoder, LSTM, outliers, CORDIC**

I. Introduction

With the increasing advances of science and technology in digital electronics and wireless communication, new breeds of tiny embedded systems known as wireless sensor nodes have emerged. A large collection of these devices forms a wireless sensor network (WSN). The ultimate goal of the WSNs goes beyond monitoring and data collection. Instead, it concerns timely data analysis and accurate real-time decision-making [1], [2]. Outlier detection is one of the primary motivating data analysis applications in sensor networks. Outlier is a data point that deviates significantly from the remaining data to arouse suspicions that a different mechanism generated them. The recognition of outliers provides valuable insights into the characteristics of the underlying generating system [3]. Although a central network entity could do the task of outlier detection, such a scheme tends to cause inefficient resource utilization and undesirable delay. Therefore, pushing outlier detection to the network's edge would be suitable for improving resource utilization and system responsiveness. Typically, WSN systems are resource-constrained. A significant proportion of network resources are consumed by data transmission. Therefore, a well-designed outlier detection algorithm could reduce the number of data transmissions while ensuring data accuracy.

Deep neural networks (DNNs) have recently become the standard tool for solving various practical problems with state-of-art performance [4]. For example, Deep Autoencoder (DAE), a feed-forward multi-layer neural network for feature learning, has greatly impacted areas such as speech recognition, object recognition, and natural language processing [5], [6]. DAE'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 an input to another machine learning model. Recurrent Neural Networks (RNNs) are another subset of DNNs capable of handling long-term dependencies, making them useful for sequential data processing [6], [7]. RNNs 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 prediction accuracy of RNNs is further improved by introducing gating units to let information through the network optionally. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are the most popular gated variants of RNNs nowadays.

Deploying DNNs on resource-constrained devices require 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 running on these platforms. Proposed methods to reduce the computation complexity of DNNs include quantization, pruning, and special hardware modules. Quantization of weights or activations helps to reduce the area of arithmetic units and memory requirement [8]–[10]. Special hardware modules such as multipliers that are based on Look-Up Tables (LUTs) can also be used for area reduction in networks with low bit precision parameters [11]. Additionally, pruning connections with small weight values using special training methods could help in creating networks with a small number of parameters [12].

Inspired by the success of Deep neural networks in solving various practical problems, this work explores the integration of DAE and LSTM networks for outlier detection on resource-constrained devices. The proposed solution consists 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. Our contributions in this work are:

- A novel two-stage hardware architecture for real-time outlier detection in time series data based on Autoencoder and LSTM networks.

- A low complexity FPGA-based implementation of the proposed architecture utilizing quantization, activation pipelining, and special hardware modules based on the Coordinate rotation digital computer algorithm (CORDIC).
- Validation of the implemented architecture on the Xilinx PYNQ-Z1 development board using meteorological dataset collected using a multi-hop WSN.

The rest of the paper is organized as follows. Section II gives an overview of the proposed Autoencoder-LSTM outlier detection system. Section III describes the accelerator architecture and its implementation on the Xilinx PYNQ-Z1 development board. Section IV discuss experimental results in addition to performance/complexity tradeoffs. Finally, section V concludes the paper.

II. Autoencoder-LSTM outlier detection system

A. System Overview

The flow of the proposed two-stage real-time outlier detection in time series data based on Autoencoder and LSTM networks is shown in Fig. 1. The input data from the sensor is preprocessed, buffered, and then fed to the Autoencoder (AE) for dimensionality reduction and feature extraction. The preprocessing step will normalize the input data using minimax normalization. The normalization step is essential to enabling the following system to find trends and patterns in the input data. The sensor data buffer will serve as a sliding window to update the inputs fed to the AE. Typically, in the case of time-series data, a past window of history is used to determine outliers. The implementation of the AE includes the encoding part only, as the LSTM prediction does not require the AE reconstruction part. 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 timestamps do not change significantly or change in a smooth way, the forecasted time step p_i produced by the LSTM predictor is compared against the new sensed value x_{i+1}. If the absolute difference is sufficiently high, the new sensed value is flagged as an outlier.

B. AutoEncoder Neural Network Model

Autoencoders (AE) are unsupervised learning techniques that leverage artificial neural networks for representation learning. Therefore, autoencoder networks' primary objective is to reconstruct the inputs instead of predicting some target variables. The AE learns by encoding the inputs; the network forces a compressed knowledge representation of the original input by imposing a bottleneck in the network architecture. Therefore, if some structure exists in the input data, this structure could be learned, compressed into a compact, latent representation. A general structure of the AE network with three hidden layers is shown in Fig. 2. The number of outputs is the same as the number of inputs, and 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 $decode$ is the reconstruction of the outputs from the reduced representation. The network is trained by minimizing the aggregated reconstruction error $\sum_{i=1}^{d} |x_i - \hat{x}_i|$ over all d-dimensions.

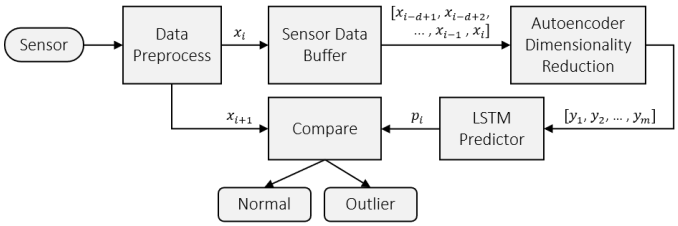

Fig. 1. The proposed Autoencoder-LSTM outlier detection system architecture.

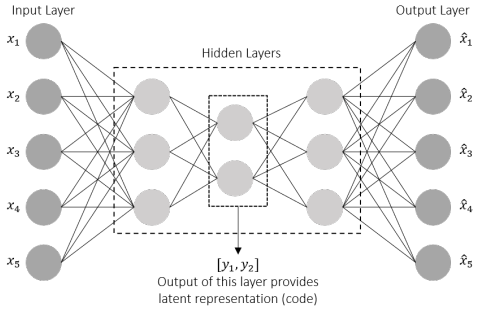

Fig. 2. A general structure of autoencoder network. The projection of inputs to latent representation (code) is termed encode, while decode is the reconstruction of the outputs from the reduced representation. The implementation of our proposed system includes the encode part only. Given that the latent representation vector is used as the input sequence to the LSTM predictor, the decoding part is not included in the implementation.

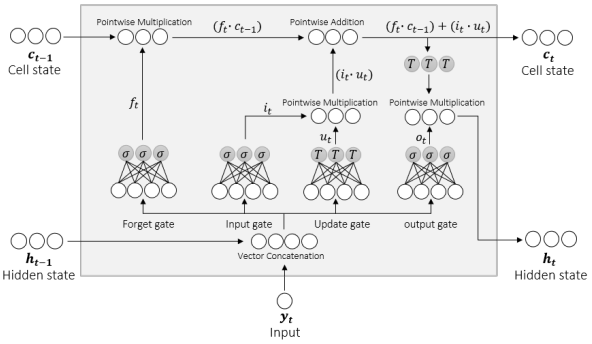

Fig. 3. A single LSTM cell architecture. Cell inputs are, the prior LSTM cell hidden-state h_{t-1} and cell-state, c_{t-1} and a single element y_t from the input sequence, which is in our proposed system the latent representation vector from the autoencoder network. The cell outputs are the updated hidden-state h_t and cell-state c_t. The σ represents the sigmoid function while T the tanh function.

C. LSTM Neural Network Model

Long short-term memory networks (LSTM) are a special kind of recurrent neural network (RNN), capable of learning long-term dependencies. LSTM networks have the form of a chain of repeating modules of neural networks. Each repeating

module known as LSTM cell consists of four neural network layers interacting uniquely, as shown in Fig. 3. The key to LSTMs is the cell-state, c_t. The LSTM cell can add or remove information to the cell state using structures called gates. Gates are ways to let information through optionally. They are composed out of sigmoid and tanh neural net layers, in addition to point-wise multiplication operations. The equations for a single LSTM cell with n neurons and m-dimensional inputs are given as:

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

where $f_t, i_t, u_t, o_t \in \mathbb{R}^{\mathrm{n}}$ are the outputs of forget gate, input gate, update gate, and output gate respectively. The c_t and h_t are the cell-state and the hidden-state. $W \in \mathbb{R}^{\mathrm{mxn}}$, $U \in \mathbb{R}^{\mathrm{nxn}}$ are weight matrices and $b \in \mathbb{R}^{\mathrm{n}}$ are bias vectors.

III. Design and Implementation

A. AE-LSTM Network Training

Keras, Tensorflow [15] is used to train both the AE network and the LSTM predictor. The Grand St. Bernard dataset [16] is used to demonstrate the performance of the proposed approach in section II. The dataset consists of temperature measurements along with other metrological characteristics of the environment collected for two months with a sampling frequency of two minutes from multiple sensor nodes deployed at the Grand St. Bernard pass, located between Switzerland and Italy. Since our focus in this work is outlier detection in univariate series, the temperature measurements were used to extract the overlapping windows to train the networks. 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. Each such window of length 90 is treated as a 90-dimensional data point. The Autoencoder is first trained using the preprocessed extracted overlapping windows; then, the encoding part is used to generate the low dimensional code to train the LSTM predictor. Both networks were trained to find the right balance between bias, variance, and computational complexity. Grid search, a neural network hyperparameters optimization technique, is used to find the number of layers, the hidden units per layer, activation function, and learning rate. Table I shows a summary of the Keras TensorFlow model that gave the desired results in terms of performance and computational complexity. The autoencoder encoding part consists of two fully connected 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 Mean Absolute Error (MAE) loss function and L_2 regularizer with factor $\beta = 5 \times 10^{-5}$. On the other hand, the LSTM is trained for 200 epochs, also using the MAE loss function. Adam optimizer was used to update both network parameters with a learning rate of 1×10^{-4}. Finally, the two networks are stacked, and the dense layers AE2, FC1, and FC2 are fine-tuned to improve prediction performance further.

TABLE I
AE-LSTM model summary.

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

B. Accelerator Design

The design of Autoencoder-LSTM aims to achieve low complexity, low latency real-time outlier detection on resource-constrained sensor nodes. Fig. 4 shows the architecture of the Autoencoder-LSTM accelerator. The main modules consist of the input buffer (INBUF), the Autoencoder (AE) unit, the LSTM unit, and the fully connected (FC) unit. The IBUF implemented using BRAM will serve as a sliding window to update the inputs fed to the Autoencoder unit (AE). The AE unit will perform the matrix-vector multiplications required to compute the latent representation of the buffered sensor readings. The LSTM unit performs the computations presented in the LSTM cell in Fig. 3 using the output of the AE unit. The FC unit comes last to calculate the predictor output using the hidden state output from the LSTM unit. All three computation units, AE, LSTM, and FC, have access to the sigmoid and tanh activation units implemented using the CORDIC algorithm. In addition to the input buffer and computational network units, BRAMs units are used to hold constant network parameters and internal results to reduce external memory access and improve latency.

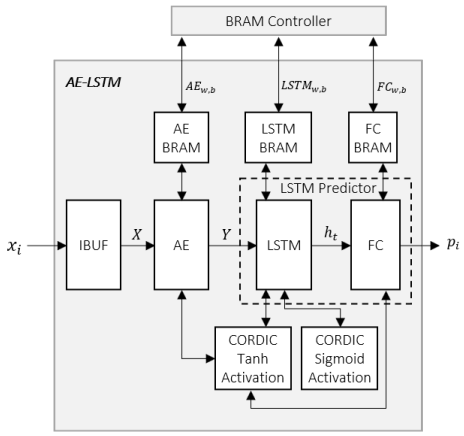

Fig. 4. Autoencoder-LSTM Accelerator architecture.

TABLE II
RANGE OF THE QUANTIZED NETWORK PARAMETERS.

	Range (Float-32)	***Range (Q4.14)***
AE1	[-0.19818062, 0.24633783]	[0x3f352, 0xfc3]
AE2	[-0.387591, 0.36001575]	[0x3e732, 0x170a]
LSTM (W)	[-2.407916, 1.2157106]	[0x365e5, 0x4dce]
LSTM (U)	[-2.2846708, 2.3490372]	[0x36dc8, 0x9656]
FC1	[-0.41140792, 0.40138096]	[0x3e5ac, 0x19b0]
FC2	[-0.27747655, 0.60573983]	[0x3ee3e, 0x26c4]

C. Data Representation

The most used software frameworks for Deep learning performs inference 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 capable of ensuring a suitable precision for the computations and keeping 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 the error between the floating-point based implementation and the fixed-point one. The performed experiments showed that the optimal solution is obtained using 18-bits representation, 4 bits for the signed integer part, while the remaining 14 bits for the decimal part. All network parameters and normalized input vectors are quantized into the same 18-bits Q4.14 fixed-point representation. The range of parameter values before and after quantization is summarized in table II. The mean squared error between the floating-point and the fixed-point implementation is in the order of 10^{-4}.

D. Activation Functions

Different methods are used for approximating nonlinear activation functions on resource-constrained and mobile platforms, including Taylor Series Expansion, LUT tables, and approximation formulas [19]. This work utilizes the Coordinate rotation digital computer algorithm (CORDIC) in hyperbolic and linear rotation modes to implement the $sigmoid$ and $hyperbolic\ tangent\ (tanh)$ activation functions. The CORDIC algorithm has very low complexity, using only shifts and adds, and better precision than the previously listed approaches. Besides, it has a low memory footprint, using only a small look-up table (LUT) with as many entries as the number of precision bits required [13], [14]. The CORDIC algorithm can be seen as a sequence of micro rotations where the input vector $[x, y]$ is rotated by an angle z as shown in (2). The rotation angles shown in (2) are chosen such that the $tanh(z)$ is a power of two; therefore, the multiplication is reduced to a bit shift. Scaling the components by $K = 1/cosh(z)$, the CORDIC gain, the equations in 2 reduce to only bit shifts and additions.

$$\begin{aligned} x_{(i+1)} &= cosh(z)[x_i - tanh(z)y_i] \\ y_{(i+1)} &= cosh(z)[tanh(z)x_i + y_i] \end{aligned} \tag{2}$$

The generalized CORDIC equations at the ith iterations are described as follows.

$$\begin{aligned} Kx_{(i+1)} &= (x_i - \mu d_i(2^{-i}y_i)) \\ Ky_{(i+1)} &= (y_i - d_i(2^{-i}x_i)) \\ z_{(i+1)} &= z_i - d_i e^i \end{aligned} \tag{3}$$

where $\mu = -1$, $e^i = tanh^{-1}2^{-i}$, $K = \Pi_n(\sqrt{1-2^{-2i}})$, and n is the number of iterations in hyperbolic rotation mode and $\mu = 0$ and $e^i = 2^{-i}$ and $K = 1$ in linear rotation mode. Given that the CORDIC algorithm in the hyperbolic coordinate system allows the computation of exp function and in a linear coordinate system can be used to perform $division$ operation, the integration of the two modes allows the computation of the $sigmoid$ and $tanh$ activation functions as shown in (4).

$$\begin{aligned} tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{sinh(z)}{cosh(z)} \\ sigmoid(z) &= \frac{1}{1+e^{-z}} = \frac{1}{1+cosh(z)+sinh(z)} \end{aligned} \tag{4}$$

E. LSTM Cell Activation Pipeline

Given that the LSTM Cell latency dominates other system modules' latency, the LSTM cell activation is pipelined to improve the overall system latency and throughput. Fig. 5 shows the data flow in the LSTM cell activation pipeline, which has five pipeline stages, $S0 - S4$. The LSTM cell formulation shown in (1) is divided into five steps, and each step corresponds to one pipeline stage. The inputs to the activation pipeline are the matrix-vector multiplication results of the forget gate, input gate, update gate, and output gate shown in Fig. 3 and expressed in the LSTM cell formulation shown in (1). Implementing the activation functions using the low complexity CORDIC algorithm enabled the instantiation of multiple activation function blocks to operate in parallel, reducing the overall latency while consuming reasonable hardware resources. The cell state c_t and the hidden state h_t are stored in BRAMs within the LSTM cell as both are used as inputs in the following time-step calculations.

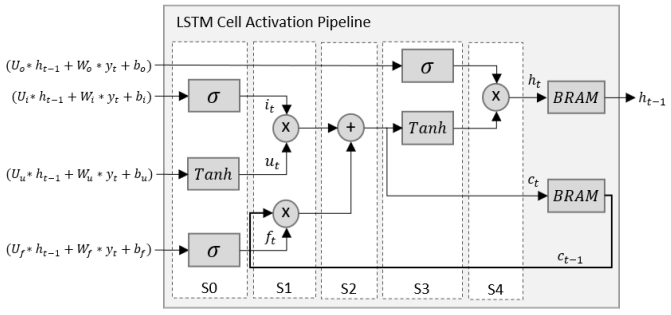

Fig. 5. Data flow in the LSTM cell activation pipeline. The inputs to the activation pipeline are the matrix-vector multiplication results of the forget gate, input gate, update gate, and output gate. The outputs are the cell state c_t and the hidden state h_t.

IV. Experimental Results

A. Experimental Setup

To analyze the performance of the proposed architecture on hardware, the Xilinx PYNQ-Z1 FPGA development board is used [17]. 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. The software running on the ARM core includes a web server hosting the Jupyter notebooks design environment, the IPython kernel and packages, Linux, and API for the FPGA. Xilinx's Vivado tools are used to design the proposed system hardware [18]. After quantizing the trained Autoencoder-LSTM network parameters into Q4.14 fixed-point representation, the parameters were extracted into C/C++ hardware header files and used in Xilinx's Vivado tools to design the proposed system hardware architecture. The resulting hardware overlay containing the Autoencoder-LSTM network IP is loaded on the PYNQ-Z1 board. A single sensor reading is transferred from the PS to the accelerator using the AXI4-Lite interface in each time step. Once the accelerator output is ready, a done register connected to the PS is flagged. The accelerator outputs are extracted and evaluated offline to measure the performance.

B. Hardware Resource Utilization

The top-level diagram of the hardware implementation using Xilinx Vivado tools targeting the PYNQ-Z1 board is shown in Fig. 6. With reference to Fig. 1, the sensor data buffer, AE, and LSTM predictors are implemented on the programmable logic (PL), while the rest are implemented on the ARM core processing system (PS). Table III shows the PL resource utilization of the floating-point implementation and the accelerated fixed-point version. A 100MHz clock globally drives the PL from the PS. The fixed-point data representation, the activation pipelining, and the CORDIC implementation of the activation functions have substantially reduced the PL resource utilization. The estimated response time of the floating-point version is $13.06\ ms$, while for the accelerated fixed-point version, it is $2.06\ ms$ giving a $6X$ reduction. In addition, the estimated power consumption of the accelerated fixed-point implementation is $0.25\ W$ compared to the floating-point version, which is $0.766\ W$.

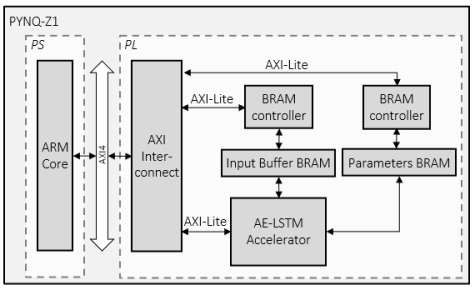

Fig. 6. Top-level diagram of the proposed system architecture on PYNQ-Z1 board.

TABLE III
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					
Utilization	26870	721	23285	43	117
Utilization %	50.51%	4.14%	21.88%	30.71%	53.18%
Accelerated Fixed-Point Implementation					
Utilization	18883	351	16364	40	18
Utilization %	35.49%	2.02%	15.38%	28.57%	8.18%

C. Outlier Detection Accuracy

Since the Grand St. Bernard dataset used in this work is unlabeled and previous examples of interesting outliers are unavailable, it is an unsupervised scenario. Therefore, the deviation of sensor reading from the proposed model prediction is used to quantify each data point's level of outlierness. The deviation vector of the training set is modeled to fit a Gaussian distribution, $X \sim \mathcal{N}(\mu, \sigma^2)$, as shown in Fig. 7. A sensor reading is flagged as an outlier if its likelihood $p \leq \tau$ and τ is set to 0.01 given a sufficient confidence interval of 99%. Suppose the absolute difference between the sensor reading and the predicted value is greater than 0.15, which is the threshold that gives a 99% confidence interval. In that case, the data point is flagged as an outlier. Fig. 8 shows an example of the system response to normal sensor readings from the Grand St. Bernard dataset. The absolute difference between the sensor reading x and the AE-LSTM prediction p is lower than the predefined threshold, denoting that the sensor readings in the sequence are normal. On the other hand, Fig. 9 shows an example of the system response to noisy sensor readings from the same dataset. The absolute difference between the sensor reading x and the AE-LSTM prediction p goes beyond the predefined threshold, indicating that the sensor readings are abnormal. In this case, a second stage of exploration is required to examine if the detected abnormality has any application-specific importance or is a weak form of outliers that could be considered as a system noise.

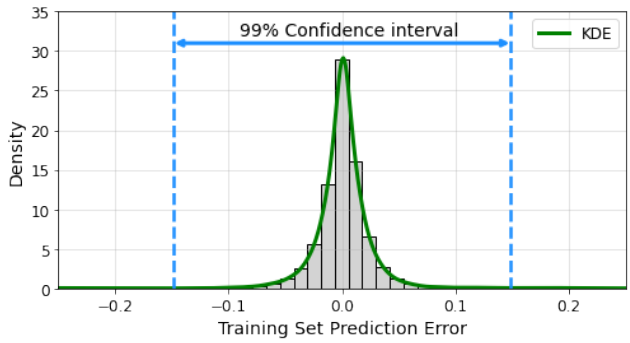

Fig. 7. Histogram of the training set prediction error with fitted Kernel Density Estimation (KDE) using Gaussian kernel with bandwidth = 0.004. The $\mu \sim 0$ and the $\sigma^2 \sim 0.06$.

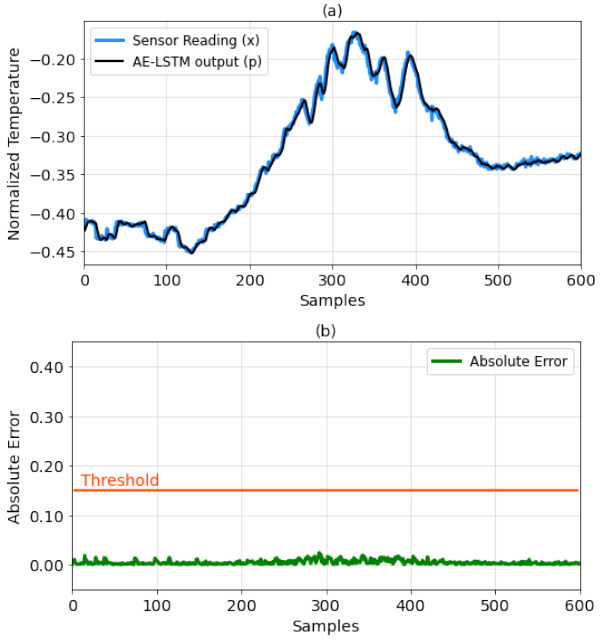

Fig. 8. System response to normal sensor readings. (a) Normal sensor readings (blue) and corresponding AE-LSTM predictions (black). (b) The absolute difference between the sensor observations and the predicted values is below the predefined threshold indicating a regular sequence.

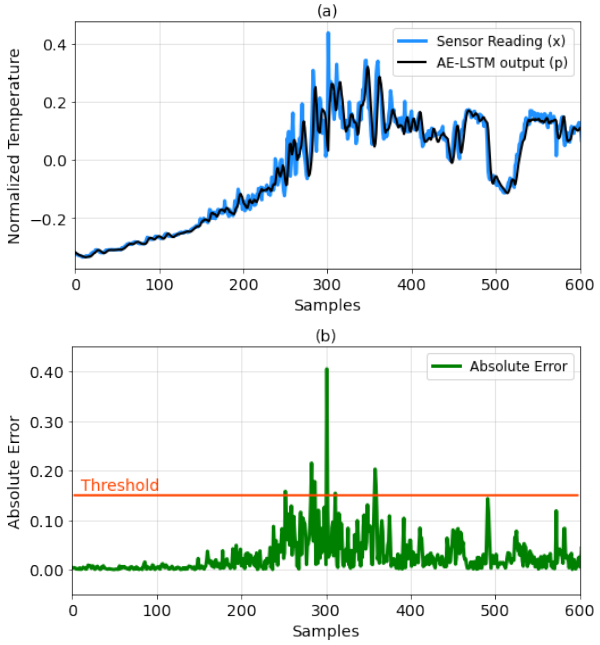

Fig. 9. System response to noisy sensor readings. (a) Noisy sensor readings (blue) and corresponding AE-LSTM predictions (black). (b) The absolute difference between the sensor observations and the predicted values exceeds the predefined threshold in the middle region, indicating abnormality that requires further study.

V. Conclusion

In this work, we introduced an FPGA-based two-stage outlier detection architecture that utilizes an autoencoder network for dimensionality reduction and feature extraction and an LSTM network for prediction and detection. The proposed architecture was implemented on the Xilinx PYNQ-Z1 board. The implementation results verify the effectiveness of the proposed architecture in detecting outliers in real-time. The $2.06\ ms$ minimum latency and $0.25\ W$ power consumption of the detector makes it suitable for resource-constrained computing platforms.

Acknowledgment

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

References

[1] L. M. Oliveira, J. J. Rodrigues, "Wireless sensor networks: a survey on environmental monitoring," Journal of Communications, vol. 6, no. 2, pp. 143-151, April 2011.
[2] G. Hua, Y. Li, X. Yan, "Research on the Wireless Sensor Networks Applied in the Battlefield Situation Awareness System," International Conference ECWAC 2011, April 16-17, 2011.
[3] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," ACM Computing Surveys, vol. 41, no. 3, pp. 15:1–15:58, 2009.
[4] I. Arel, D. C. Rose and T. P. Karnowski, "Research frontier: Deep machine learning-a new frontier in artificial intelligence research, " Comp. In tell. Mag., vol. 5, no. 4, pp. 13-18, Nov. 2010.
[5] W. Guo, J. Wang, and S. Wanga, "Deep multimodal representation learning: A survey," IEEE Access, vol. 7, pp. 63373–63394, 2019.
[6] K. Cho, et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, doi:10.3115/v1/d14-1179.
[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Comput., vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/neco.1997.9.8.1735.
[8] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2017, pp. 75–84.
[9] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in 2017 IEEE International Symposium on Circuits and Systems (ISCAS), ser. ISCAS '17, 2017.
[10] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "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), April 2017, pp. 152–159.
[11] D. Shin, J. Lee, J. Lee, and H. Yoo, "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), Feb 2017, pp. 240–241.
[12] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," CoRR, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149
[13] J. S. Walther, "A Unified algorithm for elementary functions, " Spring Joint Computer Conf., pp. 379-385, 1971.
[14] X. Hu et al., "Expanding the range of convergence of the CORDIC algorithm," IEEE Transactions on Computers, pp. 13-21, Jan. 1991.
[15] F. Chollet et al., "Keras" https://keras.io, 2015.
[16] SensorScope System [online]. Available: http://sensorscope.ep.ch/index.php/Main Page.
[17] PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC. Digilent [online]. Available: https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/
[18] Vivado Design Suite. Xilinx [online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.
[19] T. Yang, Y. Wei, Z. Tu, H. Zeng, M. A. Kinsy, N. Zheng, and P. Ren, "Design space exploration of neural network activation function circuits," IEEE Trans. CAD Integr. Circ. Syst. 38, 1974–1978, 2019.