ADDAI: Anomaly Detection using Distributed AI

Maede Zolanvari
Computer Science and Engineering
Washington University
St. Louis, MO USA
maede.zolanvari@wustl.edu

Ali Ghubaish
Computer Science and Engineering
Washington University
St. Louis, MO USA
aghubaish@wustl.edu

Raj Jain
Computer Science and Engineering
Washington University
St. Louis, MO USA
jain@wustl.edu

***Abstract*—When dealing with the Internet of Things (IoT), especially industrial IoT (IIoT), two manifest challenges leap to mind. First is the massive amount of data streaming to and from IoT devices, and second is the fast pace at which these systems must operate. Distributed computing in the form of edge/cloud structure is a popular technique to overcome these two challenges. In this paper, we propose ADDAI (Anomaly Detection using Distributed AI) that can easily span out geographically to cover a large number of IoT sources. Due to its distributed nature, it guarantees critical IIoT requirements such as high speed, robustness against a single point of failure, low communication overhead, privacy, and scalability. Through empirical proof, we show the communication cost is minimized, and the performance improves significantly while maintaining the privacy of raw data at the local layer. ADDAI provides predictions for new random samples with an average success rate of 98.4% while reducing the communication overhead by half compared with the traditional technique of offloading all the raw sensor data to the cloud.**

***Index Terms*—Distributed AI (DAI), Artificial Intelligence (AI), Machine Learning, Industrial IoT (IIoT), Industry 4.0.**

I. Introduction

Internet of Things (IoT) integrated into the Industrial Control Systems (ICSs) is called Industrial Internet of Things or IIoT. IIoT systems are the foundations of the critical infrastructures of a nation. Industrial processes such as smart manufacturing, oil and gas exploration, water distribution and treatment, etc., rely heavily on these systems. Utilizing IoT technology in ICSs enhances network intelligence in the optimization and automation of industrial operations.

Moreover, the impact of Artificial Intelligence (AI) analysis on the rapid growth of the IoT is undeniable. IoT has become a fundamental part of both personal lives and the industrial environments. The integration of AI and IoT automates connections, interactions, and data exchange among machines and devices for comprehensive functionality and higher efficiency without requiring any human in the loop. This automation needs high-speed computations; however, the heavy computational load of AI-based algorithms leaves a burden on the system's data processing speed. This concern is even more troublesome in systems such as IIoT, where real-time operations are essential.

According to the annual report of Stanford University's AI Index, the speed of AI is outpacing Moore's Law. Since 2012, AI's required computing power has increased 300000 times compared to the expected seven times increase under Moore's law [1]. The computation power is expected to double every two years based on Moore's law, while the amount of computing power required for AI doubles every 3.4 months . This proves the unsustainable computing power of AI and the fact that in several years, reaching a general intelligence will not be through a highly powerful complex AI algorithm, but rather through collective intelligence. Collective intelligence means distributing the computation and making decisions through several learning components concurrently and in parallel. This fact emphasizes why distributing the computations in an AI-based IoT system is essential.

On the other hand, traditional centralized computing is not capable of carrying out, keeping up, and adapting to the required security operations. At the same time, all the sensor data is offloaded into a large and complex AI. It also suffers from an important security vulnerability, i.e., a single point of failure, which negatively impacts the main goal of IIoT systems, high availability and reliability. Also, it comes with high latency, communication costs, and privacy issues. Therefore, our goal in this project is to reduce the need to send all the field data to a central remote computing processor. This will lead to a promising architectural solution to a fundamental network problem, latency.

Despite the increased interest in using rapid computations in AI analysis, most research works focus on distributing the training process. Our proposed model fragments the decision making-process of AI models in a granular fashion. It can also make decisions locally and later take advantage of those decisions when it comes to classification tasks in the upper layer hierarchy.

The research contributions of this work are as follows:

1) We propose a universal, accurate, and well-performing DAI model called ADDAI that does not compromise the learning performance. At the same time, it provides a fair distribution in the computational loads.
2) Our case study introduces a unique perspective on the assessment of AI-based distributed systems for IIoT and Industry 4.0 [2].
3) The proposed framework can be utilized as a low overhead intrusion detection system (IDS) for IIoT systems.
4) Our IIoT security dataset is released to support the research community for a more extensive and diverse data collection in the emerging field of DAI for IIoT. [1]

[1] https://www.cse.wustl.edu/ jain/iiot2/index.html

2021 IEEE International Conference on Networking, Sensing and Control (ICNSC), Xiamen, China, 3-5 Dec. 2021

II. Related Work

DAI is an immense area that includes the distribution of different parts of AI models, such as the datasets, training, tasks, etc. Distribution among different computing entities can be decided based on their differences in energy consumption, memory restrictions, computing power, and many more factors. Additionally, a centralized solution is not even an option when data is inherently distributed or too big to store on a single device. There exists several comprehensive surveys related to this area such as [3], [4], and [5].

On the other hand, there are situations in which it is beneficial or even required to isolate some subsets of the data. These concerns usually happen when privacy issues are involved. In [6], a framework of differential privacy is considered, and a deep neural network with a modest total privacy loss is developed. Another technique to training models in a privacy-sensitive context is utilizing distributed ensemble models. This guarantees the complete separation of the training data subsets where privacy needs to be preserved. [7] proposes a federated learning approach that uses ensemble distillation in a medical relation dataset. The suggested technique also overcomes the communication bottleneck caused by the need to upload a large number of parameters in regular federated learning models. Another challenge in these kind of models is that a method needs to be found that properly balances each model's input for an unbiased training result.

Another popular application of DAI is in unreliable networks, where we have a lossy network and cannot really get a guarantee that sent information will successfully get to the supposed receiver. In [8], such scenarios are studied. To overcome this challenge, parameter servers are introduced to store the parameters of the learning model (e.g., the weights of a neural network). These servers serve the parameters to local units (Workers) in charge of processing the data and computing updates to the parameters. Each server connects to all the local units and serves a partition of the model, and each local unit holds a replica of the whole model. However, every communication link between each server and each local unit has a non-zero probability of being dropped.

DAI is a well-known approach in edge-cloud computing. The concept is to move all or some of the training workloads from the central computing unit (e.g., the cloud) to the edges. This way, we reduce the enormous network communication overhead and provide low-latency solutions. To name some challenges in this approach, we can mention parallel training, model synchronization, and workload balancing to address the imbalance of workload and computational power of different edge nodes. For instance, [9] studies a case study of utilizing DAI in video surveillance systems. Popular services are also cached on the edge servers, so that the latency can be reduced even more, and the computation can be offloaded easily. However, only the cost of bringing services to the edge is considered, and the cost of transmitting data between the edge and cloud server is assumed negligible.

Not all the elements in the IIoT systems have the same computing capabilities. The distribution of the tasks among the elements of the hierarchy must be fair. Therefore, their different computation and communication capabilities should be assessed. The trade-off between cost and capacity of the resources is another important consideration for optimizing resource sharing. There is an extensive study and survey on resource management of different levels of hierarchy and their constraints in [10] and [11].

Another approach in the distributed AI is to divide the tasks into micro-services and spread them out in a distributed fashion. These tasks can be data filtering and cleansing, training, testing, labeling, security computation, etc. Different stages of learning development can also be divided into layers among a few high-capacity processing entities. Afterward, the developed model can be accessed simultaneously by several smaller processors to predict the labels of new instances, which is a low computation task. The same mechanism can be applied to other computation tasks. As another example, breaking the training set or the training process into several parallel local tasks can also provide an early result at the local layer and help deal with massive datasets and develop scalable learning. Some examples of research works in this concept include [12], [13], and [14].

III. Background

ADDAI is composed of two learning models, autoencoders and AdaBoost. For the sake of completeness, we first provide the basics of how these two models function before diving into the details of our proposed model.

A. Autoencoder

Autoencoder is a type of neural network with a bottleneck layer in its network, forcing it to produce a compressed knowledge representation of the original input (also known as the code). The network consists of two parts, an encoder part that produces the code and a decoder that reconstructs the input from the code. By learning a smaller size code, autoencoders can work with voluminous unlabeled data and extract useful features with good accuracies [15].

Suppose we have a training set $X \neq \emptyset \in \mathbb{R}^{N \times K}$, where each sample x_i in $X = \{x_1, x_2, ..., x_N\}$ is a $1 \times K$ vector (i.e., K features). An example is shown in Fig. 1. In this example, we have a dataset in which each sample has $K = 40$ features, and the size of the produced code is 20.

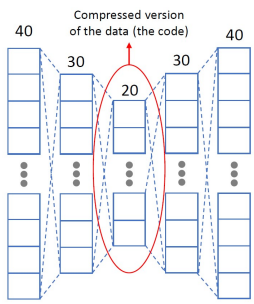

Fig. 1: Diagram of an autoencoder. The middle layer, which is called the code, is the compressed version of the input.

Any neural network can be uniquely identified and built by its hyper-parameters. There are four hyper-parameters for an autoencoder which are weights $\boldsymbol{W}$, biases $\boldsymbol{b}$, code size, and the number of hidden layers (when the number of nodes per layer and the loss function are fixed).

Choosing the size of the code and the number of hidden layers both have specific trade-offs. The size of the code should not be too small. First, we might lose more information from the data. Second, it would be tough to capture all the relationships among the input features. This causes underfitting. Meanwhile, the size of the code should not be too large. The network might simply overfit the code layer by learning the noise and small details of input data.

On the other hand, the number of hidden layers should not be too high. The gradient effects of the first layers become too small, and the input becomes almost irrelevant to the code. On the contrary, if the number of hidden layers is too small, it cannot extract useful relations. In neural networks, the first layers usually extract very simple relations like lines and curves, but deeper layers extract more complex features.

Construction error is another important concept in autoencoders. As mentioned before, the decoder part tries to mimic the input layer in the output layer. The more similar these two layers are, the more accurately the autoencoder extracts the code. The most popular way of optimizing the autoencoder's parameters to minimize the construction error is using the mean squared error (MSE), Eq. 1

$$L(x_i, x_i') = 1/K \sum_{j=1}^{j=K} (x_{i,j} - x_{i,j}')^2 \tag{1}$$

where, x_i, is a sample for the input layer (e.g., the leftmost 40 dimensional layer in Fig. 1) and x_i' is the output layer (e.g., the rightmost 40 dimensional layer in Fig. 1).

B. AdaBoost

AdaBoost, which is short for Adaptive Boosting, is a popular boosting technique. Boosting is an ensemble method that improves the performance of a number of weak learners and builds a strong learner. Decision trees are the most suited and commonly used weak learners in AdaBoost. In this technique, the weak learners are added sequentially and trained on repeatedly modified versions of the data (i.e., different weights for each instance). Training stops when a pre-defined number of weak learners are trained, or no further improvement can be made. Afterward, the predictions from all the weak learners are combined through a weighted sum or vote to be decided as the final decision, Eq. 2.

$$L_T(X) = \sum_{t=1}^{T} \alpha_t l_t(X) \tag{2}$$

where $\alpha_t = \frac{1}{2}\ln(\frac{1-\epsilon_t}{\epsilon_t})$ is the weight of t'th weak learner, in which $\epsilon_t = (\sum_{y_i \neq l_t(x_i)} w_{i,t})/(\sum_{i=1}^{N} w_{i,t})$ is its weighted error rate.

In AdaBoost, we initially give each instance in the training dataset equal weights. As the training progresses, at each step, it gives more weights to individual instances that were classified wrong and less to those already correctly classified. The updating rule for the weight of each training sample is based on Eq. 3.

$$w_{i,t+1} = w_{i,t} \times e^{-y_i l_t(x_i)\alpha_t} \tag{3}$$

If we assume $y_i \in -1, 1$, when the prediction of the weak learner is correct (i.e., both y_i and $l_t(x_i)$ are -1 or 1), the coefficient to updating the weight is $e^{-\alpha_t}$. This means we decrease the weight. When they are different, the updating term will be e^{α_t} which means more weight is assigned to a misclassified sample. [2]

IV. Proposed Model

We propose a layer-wise prediction scheme. These layers include local processor units, the edge, and the cloud. The local processor is in direct contact with a group of sensors working together. We assume near zero communication delay between the sensors and the local processor. The edge can be a middle point between the local processors and the cloud for lower latency computations than cloud offloading. For simplicity, here, we consider a two-layer hierarchy, i.e., the local-cloud scenario. In the following, we discuss the formulate the problem and details of the proposed technique.

A. Problem Formalization

In our proposed scenario, the tasks are distributed among these components (i.e., local and cloud) for higher efficiency in terms of latency and communication overhead. One important task that is done locally and distributed all over the network is anomaly detection. This way, the anomalies can be detected by the local processing units quickly without the need for sending the data to any other local unit or the cloud. On the other hand, the computation-intensive tasks are done in the cloud. These tasks include training the local models and further investigating the label of the data when we are still uncertain about it despite the local processing.

Two scenarios can be defined whether we can trust the classification of the local model at the local processing unit or not? If yes, local decisions are made, and then only the final results are sent to the cloud. If the classifications of the local model cannot be trusted, e.g., due to low accuracy rates, further investigation is required on the sample's label. We design the local units such that at least a compressed version of the data that is much smaller compared to the original raw sensor data that would be sent to the cloud. With this scheme of layer-wise computations, the communication between the local processors and the cloud will be decreased to a significantly lower rate.

B. Architecture

We develop an autoencoder as the proper learning framework for local processing units. Fig. 2 shows the diagram of the proposed model. With their produced compressed version of the input, the communication cost significantly drops. This design also guarantees privacy restrictions because latent variables composed from the data features are sent to the cloud instead of the raw data from sensors.

[2]In our dataset, class normal is labeled as 0, but for the sake of weight updates in AdaBoost, we represent the class normal with -1 instead of 0. This is just a label and does not have any numerical value.

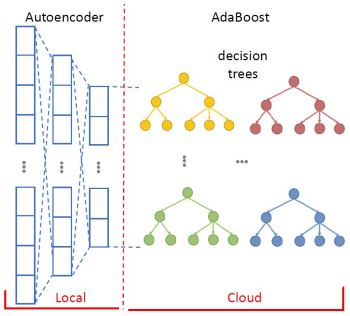

Fig. 2: Diagram of the proposed model. The raw data first is fed into a local autoencoder to compress and resize; then, they will be fed into an AdaBoost model in the cloud to decide on their label.

We train the local autoencoder by normal data only. The achieved model parameter ($\boldsymbol{W}$ and $\boldsymbol{b}$) in the trained autoencoder are used to accurately produce the compressed data (representing code) of the normal samples. Therefore, the compressed data is a model for the normal data. When an attack sample is fed to the model, the reconstruction error is very large, which is how the anomalies are detected. The larger the error is, the more confident we are that the sample is an attack.

Since the training might be computationally heavy for some of the local units, the cloud trains the autoencoder model using the data from all the sensors, $X = \{x_1, x_2, ..., x_N\}$. Here, N is the total number of samples from all the sensors across the network. After the training is done, the cloud sends the updated model parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ back to the local units to build a local version of the autoencoder. Please bear in mind that training is a one-time process.

If the predictions of the local processing units are trustworthy, in case an anomaly is detected, they can quickly take action by alarming the control center. However, sometimes the compressed data is sent to the cloud for further investigation. In the cloud, we utilize three different trained AdaBoost models. One of these models is more sensitive on detecting the normal class; another one is sensitive to the attack class, and the third one is a neutral model that pays fair attention to both classes. In other words, these three models are trained in a way that they would have different sensitivity/recall scores for different classes of the data. Since AdaBoost works with weights for individual instances, we have assigned class weights at the decision tree levels (through the weak learners). This class weights can also be very helpful when dealing with imbalanced datasets, where the minority class is usually overlooked by the model. The loss function for each of the AdaBoost model can be modeled as in Eq. 4.

$$\text{loss} = \frac{1}{N}\sum_{i=1}^{N} cw_0(y_i \times \log y_i') + cw_1((1-y_i)\times\log(1-y_i')) \quad (4)$$

where cw_0 and cw_1 are the class weights in the case of binary classification. A grid search is conducted to get the optimal value of these weights.

In our proposed model, despite the fact we might not fully trust the local decisions on the data labels, we want to take advantage of them. We utilize them in a new concept we call *local ranges*.

C. Local Range

The local range is a critical component of our proposed architecture. It determines the three ranges of the samples fed to each AdaBoost model in the cloud. It is specific for each local processing unit and depends on how certain we are in the predictions of the samples made by that local unit classifier. A local unit has an accuracy score between 0 and 1; we call it f_i for i'th local unit. Therefore, the ratio of the number of samples that are misclassified by the autoencoder in the i'th local unit is $r_i = 1 - f_i$.

On the other hand, when the local unit calculates the reconstruction errors, we can sort them in ascending order. Let us call the sorted array Err_i. Local autoencoder would classify any sample with a reconstruction error below a threshold, η_i, as normal and any sample above that as an attack. With good approximate, we can assume that misclassification happens around the threshold. We want to feed these samples to the regular model as it is partial to both classes. Therefore, we define the range of the reconstruction error of the samples that should be fed to the regular model is defined as

$$\begin{aligned}\{&\eta_i - Err_i[idx - \lfloor(r_i \times N_i/2)\rfloor],\\ &\eta_i + Err_i[idx + \lfloor(r_i \times N_i/2)\rfloor]\}\end{aligned} \quad (5)$$

where η_i is the threshold calculated by the autoencoder for the i'th local unit based on its pick performance on the training data specific to that local unit. idx is the sample index whose reconstruction error is the closest to the value of η_i, and N_i is the number of training samples in the local unit i.

As shown in Fig. 3, we make the decision on which sample to be fed to which model based on the local range and the value of the reconstruction error of the sample. The samples with reconstruction errors within the local range to be fed to the regular model, and below or above that to the normal and attack models, respectively. This figure shows an example of reconstruction error for a set of test samples. The more uncertain we are about the local decisions, the wider the boundary for the regular model is.

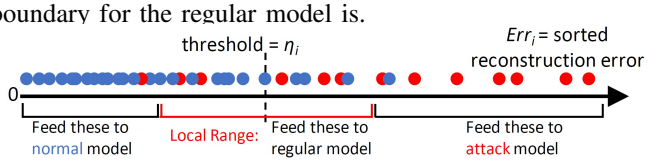

Fig. 3: Local decisions based on the reconstruction errors from the autoencoder.

D. Communication Cost

The communication cost for offloading the data to the cloud, k, can be formulated as follows:

$$k = 1 \text{ (for } \mathcal{C}) + 4 \text{ (for reconstruction error)} + 4 \times h \quad (6)$$

In this equation, $\mathcal{C}$ is the predicted class by the local unit, and one byte is assigned for it. 4 bytes are dedicated for the reconstruction error. h is the length of the code layer of the autoencoder. 4 bytes are used to represent a signed single-precision floating-point number.

V. Case Study: IIoT Security

We have built a lab-scale industrial Internet of Things (IIoT) system to collect realistic and up-to-date datasets for network-based intrusion detection systems (IDS). We have implemented a supervisory control and data acquisition (SCADA) system widely used by industries to supervise the level and turbidity of liquid storage tanks. This IIoT system is employed in industrial reservoirs and water distribution systems as a part of the water treatment and distribution. More information regarding our testbed can be found in our previous papers [16] and [17].

A. Utilized Datasets

The dataset is the one collected from our testbed, which we refer to as "WUSTL-IIoT." To collect the proper dataset, we (as a white-hat attacker) attacked our testbed with different cyber-attacks [16]. Specifics of our dataset are in Table I.

TABLE I: Specifics of the tested dataset

Dataset	WUSTL-IIoT
# of observations	1,194,464
# of features	41
# of attacks	87,016
# of normals	1,107,448

B. The Learning Models

The inputs to the learning models are the flow instances, as mentioned in the previous subsection. The output of the models can be a multi-class or binary classification. However, for simplicity, we have built ADDAI by treating the model as binary classifier with normal as 0 and attack as 1.

The autoencoder used in the local processing is programmed using the neural network module of PyTorch [18], containing the encode, decode, and forward methods. Depending on the code size, we use a different number of hidden layers. For instance, for a code size of 25, 7 layers are used in the model: input data layer, two encoding layers, code layer, two decoding layers, and the output layer, as shown in Table II. AdaBoost is used in the cloud with decision tree classifiers as the weak learner. The number of estimators in AdaBoost was set at 100, and scikit-learn library [19] was used to implement it.

TABLE II: Local autoencoder model hyperparameters

Parameter	**Typical value(s)**
# of layers	7
# of neurons per layer	40, 35, 30, 25, 30, 35, 40
# of epochs	100
Optimizer	*adam*
Dropout rate	0.05
Learning rate	0.01
Activation function	hyperbolic tanh function

C. Performance Metrics

In this paper, we evaluate the model's performance using three metrics, including Accuracy, MCC, and undetected rate (UR) shown in Eq. 8, Eq. 7, Eq. 9, respectively.

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (7)$$

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \quad (8)$$

$$\text{UR} = \frac{FN}{FN+TP} \quad (9)$$

where TN is the number of normal data labeled as normal, TP is the number of attack data classified as an attack, FP is the number of normal labeled as an attack, and FN is the number of attacks labeled as normal by the model.

D. Results

In this experiment setup, we set the number of local units to be 3. We have randomly divided the WUSTL-IIoT dataset into three equal sized subsets. Fig. 4 shows the exact number of samples of each class per local device.

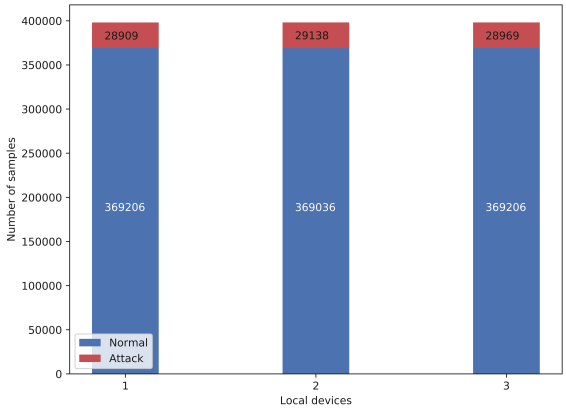

Fig. 4: Number of samples from each class of data (Attack or Normal) at each local device.

Here, we test the performance of ADDAI assuming the worst-case scenario, where we cannot trust any of the local decisions on the samples' label, and they should be sent to the cloud for further investigation. However, we take advantage of the local analysis as mentioned before.

Our analysis starts by choosing the right code size based on the performance and the communication cost. For different values of h, $10, 15, 20, 25$, and 30, the associated communication cost (Eq. 6) and the performance of the local devices are compared. Fig. 5 represents the change in the MCC results at the cloud for each of these values. The code size of 25 with an average of 105B communication cost produced the best result. Therefore, in the rest of the evaluations, we use the code size of 25 unless stated otherwise. For this setting, the stored autoencoder models on the local processing units require only 16 kB of memory.

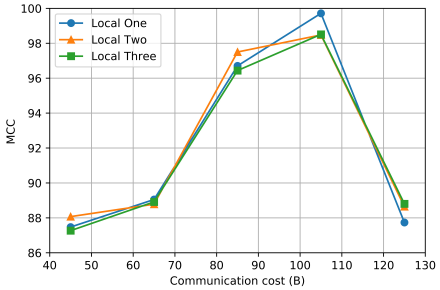

Fig. 5: Communication cost vs. MCC scores for different code lengths.

To get a concrete comparison of the performance improvement through our model, the performance metrics mentioned

in Section V-C calculated from the values in the previous tables are summarized in Fig. 6. Please note that, for accuracy and MCC metrics, the higher the value, the better, while for UR, the lower the value, the better. Each local unit splits its dataset into training and test sets with an 80:20 ratio. Since the autoencoder in the local units predicts the labels (regardless we can trust them or not), we have utilized them as the local performance on the data. The cloud results are derived from feeding different samples to different AdaBoost models (regular, normal, and attack) based on their calculated local ranges. Table III shows the utilized ranges for each local device. These numbers are calculated using Eq. 5.

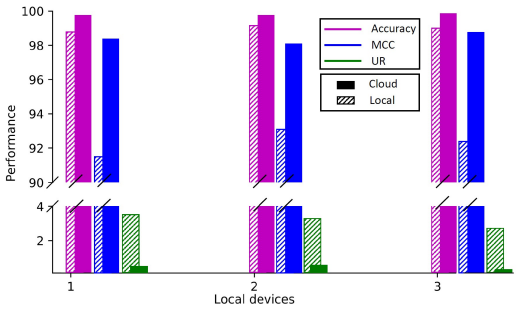

Fig. 6: Different performance metrics comparison of the three local devices vs. their corresponding cloud results.

TABLE III: Specifics of the tested dataset

Local device	η_i	Normal range
One	0.43	[0.29, 1.38]
Two	0.58	[0.17, 1.43]
Three	0.42	[0.28, 1.35]

We also tested what would happen if we feed all the samples to only one of the AdaBoost models in the cloud. For simplicity, we combined the data from all the local devices. The results are shown in Fig. 7.

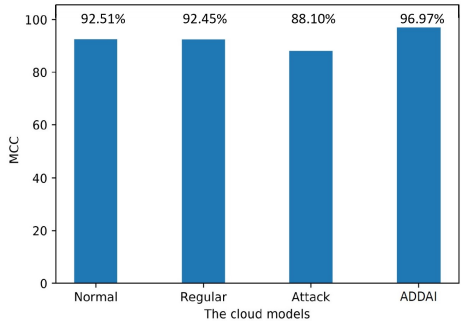

Fig. 7: The MCC results, if only one of the cloud models were utilized.

As seen from these results, our evaluations show taking advantage of our prior knowledge of the label of the data helped our proposed model achieve better performance.

VI. Conclusion

DAI has proven beneficial in applications such as IIoT. In this paper, we have proposed a low overhead DAI called ADDAI. In our proposed framework, anomaly detection is done close to the sensor level, while more investigation on the data can be done in the cloud. To conserve the communication resources, we send a compressed version of the data to the cloud. Through our proposed models, we also preserve the privacy requirements by sending a latent variant of the data to the cloud. We show empirical proof of performance improvement and decreased communication cost of our proposed technique.

Acknowledgement

This work has been supported under the NSF grant CNS-1718929. The statements made herein are solely the responsibility of the authors.

References

[1] Stanford, "Artificial intelligence index annual report," 2019.
[2] A. Gilchrist, *Industry 4.0: the industrial Internet of things*. Springer, 2016.
[3] S. Ponomarev and A. E. Voronkov, "Multi-agent systems and decentralized artificial superintelligence," *arXiv:1702.08529*, 2017.
[4] J. Queiroz, P. Leitão, J. Barbosa, and E. Oliveira, "Distributing intelligence among cloud, fog and edge in industrial cyber-physical systems," in *ICINCO*, 2019.
[5] M. Afrin, J. Jin, A. Rahman, A. Rahman, J. Wan, and E. Hossain, "Resource allocation and service provisioning in multi-agent cloud robotics: A comprehensive survey," *IEEE Communications Surveys and Tutorials*, vol. 23, no. 2, pp. 842–870, 2021.
[6] M. Abadi, A. Chu, I. Goodfellow, B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," *23rd ACM Conference on Computer and Communications Security (ACM CCS)*, p. 308–318, 2016.
[7] D. Sui, Y. Chen, J. Zhao, Y. Jia, Y. Xie, and W. Sun, "FedED: Federated learning via ensemble distillation for medical relation extraction," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2118–2128, 2020.
[8] C. Yu, H. Tang, C. Renggli, S. Kassing, A. Singla, D. Alistarh, C. Zhang, and J. Liu, "Distributed learning over unreliable networks," *Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97*, 2019.
[9] J. Chen, K. Li, Q. Deng, K. Li, and P. S. Yu, "Distributed deep learning model for intelligent video surveillance systems with edge computing," *IEEE Transactions on Industrial Informatics*, 2019.
[10] K. Tocze and S. Nadjm-Tehrani, "A taxonomy for management and optimization of multiple resources in edge computing," *Wireless Communications and Mobile Computing*, 2018.
[11] A. Mijuskovic, A. Chiumento, R. Bemthuis, A. Aldea, and P. Havinga, "Resource management techniques for cloud/fog and edge computing: An evaluation framework and classification," *Sensors*, 2021.
[12] C. Li, C. Wang, and Y. Luo, "An efficient scheduling optimization strategy for improving consistency maintenance in edge cloud environment," *The Journal of Supercomputing*, vol. 76, p. 6941–6968, 2020.
[13] H. Wu, W. Knottenbelt, K. Wolter, and Y. Sun, "An optimal offloading partitioning algorithm in mobile computing, quantitative evaluation of systems," in *13th International Conference, QEST 2016*, (Quebec City, Canada), pp. 311–328, 2016.
[14] K. Kim, J. Lynskey, S. Kang, and C. Hong, "Prediction based sub-task offloading in mobile edge computing," in *2019 International Conference on Information Networking (ICOIN)*, pp. 448–452, 2019.
[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[16] M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan, and R. Jain, "Machine learning-based network vulnerability analysis of industrial Internet of things," *IEEE Internet of Things Journal*, vol. 6, pp. 6822–6834, 2019.
[17] M. A. Teixeira, T. Salman, M. Zolanvari, R. Jain, N. Meskin, and M. Samaka, "SCADA system testbed for cybersecurity research using machine learning approach," *Future Internet*, vol. 10, pp. 1–15, 2018.
[18] A. Paszke and et al., "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.
[19] Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.