STARS

University of Central Florida
STARS

Electronic Theses and Dissertations, 2020-

2022

Deep Learning Anomaly Detection Using Edge AI

William Holdren
University of Central Florida

 Part of the Computer Sciences Commons
Find similar works at: https://stars.library.ucf.edu/etd2020
University of Central Florida Libraries http://library.ucf.edu

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2020- by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Holdren, William, "Deep Learning Anomaly Detection Using Edge AI" (2022). *Electronic Theses and Dissertations, 2020-*. 1026.
https://stars.library.ucf.edu/etd2020/1026

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

DEEP LEARNING ANOMALY DETECTION USING EDGE AI

by

WILLIAM HOLDREN
B.S. University of Central Florida, 2019

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2022

ABSTRACT

Deep learning anomaly detection is an evolving field with many real-world applications. As more and more devices continue to be added to the Internet of Things (IoT), there is an increasing desire to make use of the additional computational capacity to run demanding tasks. The increase in devices and amounts of data flooding in have led to a greater need for security and outlier detection. Motivated by those facts, this thesis studies the potential of creating a distributed anomaly detection framework. While there have been vast amounts of research into deep anomaly detection, there has been no research into building such a model in a distributed context. In this work, we propose an implementation of a distributed anomaly detection system using the TensorFlow library in Python and three Nvidia Jetson AGX Xavier deep learning modules. The key objective of this study is to determine if it is practical to create a distributed anomaly detection model without a significant loss of accuracy on classification. We then present an analysis of the performance of the distributed system in terms of accuracy and runtime and compare it to a similar system designed to run on a single device. The results of this study show that it is possible to build a distributed anomaly detection system without a significant loss of accuracy using TensorFlow, but the overall runtime increases for these trials. This proves that it is possible to distribute anomaly detection to edge devices without sacrificing accuracy, and the runtime can be improved with further research.

ACKNOWLEDGMENTS

I would like to thank Dr. Yanjie Fu for giving me this opportunity to conduct this research by taking me on as a thesis student and providing the hardware on which the anomaly detection framework was built. I would also like to thank Dongjie Wang for providing guidance on the implementation of the model and datasets.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

In the age of modern computing, there are increasingly many devices that are capable of connecting to the internet and rely on that infrastructure to perform their tasks. Any device that has embedded systems that can collect and send data using a connection to the internet is an Internet of Things (IoT) device. IoT devices typically send their data to an IoT gateway or another edge device for analyzing locally or transmitting it to the cloud for analysis. However, this explosive growth of the IoT has led to massive increases in the amount of data that needs to be managed, and the more information and data that is shared, the more potential for hackers to steal that information [1]. Predictions on the number of IoT devices vary, but a 2015 to 2019 survey by [2] recorded 10 billion IoT devices in 2019 and predicts a total of 26.5 billion devices connected to the internet in 2022 with 16.4 billion being IoT. It is apparent that the sheer amount of data generated by that many billions of devices is not able to be reasonably processed.

One method of data analysis that has gained much popularity is the use of a deep neural network (DNN). DNNs have advanced speech recognition, natural language processing, and computer vision to name a few, and they are continuing to be applied to many new areas. Several advantages of DNNs include no need for feature engineering, good performance on unstructured data, no requirement for data labels, and good results with correct training [3]. This means that DNNs are effective at analyzing the data generated by IoT devices and are a potential tool for solving the problem of anomaly detection.

Anomaly detection, or outlier detection, is an application of data science involving the identification of unusual points that deviate too far from a baseline, achievable using many

different methods including classification, clustering, and local factoring [4]. The field of anomaly detection has a long history of research into improving the accuracy of the methods to better determine abnormal data objects. This research all started with the proposal of an intrusion detection system by D.E. Denning in 1987 [5]. Since then, there have been many advancements in machine learning techniques to improve the classification accuracy of outliers, and anomaly detection models have become popular in cybersecurity. More recently, considerable headway has been made by using DNNs in solving some challenges faced when using other machine learning methods. Since there can be a lot of outliers in IoT data, DNNs can be applied to help detect and filter the anomalous data. They can also help strengthen the security of IoT devices against attackers. There were 1.5 billion IoT cyberattacks recorded by Kaspersky in the first half of 2021, so this is a prevalent security threat to businesses and individuals' data [6].

1.1 Challenges in Anomaly Detection

In the context of an intrusion detection system, the most significant challenges machine learning methods tend to suffer from are high amounts of false positives and low throughput from high evaluation costs on more complex models [7]. The latter is especially prevalent when large amounts of data or rapid data streams are involved. Neural networks in particular have high computational requirements.

More generally, there are several challenges in anomaly detection that remain unsolved: achieving a high anomaly recall rate, detecting anomalies in high-dimensional or dependent data, data-efficient learning of normal and abnormal points, robustness to noise, and detecting complex anomalies. In recent years, much headway has been made in creating DNN methods for anomaly

detection that have considerably better performance than traditional methods on challenging problems [8]. However, the computational demands of running these detection networks are still high.

1.2 Thesis Statement

Many real-world anomaly detection tasks involve large amounts of data. In order to tackle the computation demands in the realm of IoT, this thesis aims to determine if it is possible to leverage edge computing resources to train and evaluate a DNN anomaly detection model without a substantial loss of accuracy. This thesis presents a distributed deep anomaly detection model built using TensorFlow, verifies the impacts on accuracy, and reports on the results to support the following statement: by utilizing edge devices, it is possible to reduce the computational load on the chief server and handle big data anomaly detection tasks using a DNN without reducing the accuracy of the predictive model.

1.3 Thesis Contribution

In this thesis, we tackle the challenge of handling anomaly detection on the edge to better leverage available resources. The goal of this is to have the model be scalable and able to handle the large amounts of data generate by IoT devices, something which a DNN anomaly detector running on a single server would not be able to handle. By proving the validity of the thesis statement, it will enable further research and development of distributed deep anomaly detection models capable of handling the continually growing influx of data from the IoT without concern over losing accuracy.

In Chapter 2, we review related works and present key concepts and terms useful in understanding the material presented in this paper. The structure of the distributed autoencoder anomaly detector used in this study is described in detail in Chapter 3. The results and analysis from testing the model are given in Chapter 4. Finally, we conclude by summarizing the results and propose future avenues for this research.

CHAPTER 2: PRELIMINARIES AND TERMINOLOGY

This section covers related work, introduces definitions of common terms, and provides some background on key concepts utilized in this thesis.

2.1 Related Work

In classic network anomaly detection, a survey in [9] identifies the main categories of the methods to be classification-based, statistical based, knowledge-based, and clustering-based and evaluates the strengths and weaknesses of each. An earlier survey performed in [7] identifies similar methods and evaluates many of the same model categories. Neither of them go into depth with DNNs nor do they cover any distributed context considerations.

A recent, extensive survey on anomaly detection using deep learning is presented by [8] in which 180 works are referenced. It shows that there has been copious amounts of research into improved deep anomaly detection models with three main categories: deep learning for feature extraction, learning normal feature representations, and end-to-end anomaly scoring. The authors of the survey also present their own end-to-end anomaly scoring network in [10] where they make use of deviation networks, a few labeled anomalies, and prior probability to add a statistical component. While each of these methods tackles certain subsets of the challenges with anomaly detection, all aimed at improving accuracy, none of them tackle the problem of computational costs and big data.

In the category of distributed anomaly detection using DNNs, very few works exist, and none of them perform a comparison study on the accuracy impact due to the distributed training. Some

works focus on optimizing the runtime of distributed DNNs in general [11, 12], but they do not perform any analysis on potential differences in accuracies for models trained on a single machine compared to their distributed training methods. Additionally, [12] focuses on a branchy network that runs separate branches on different IoT devices and aggregates the results to improve accuracy, but they do not mention any comparison to the accuracy of a non-branchy model that simply trains on a set of fully aggregated data. For anomaly detection using distributed DNNs, the work in [13] focuses on optimizing the detection delay in IoT datasets. In [14], the distributed model does not seem to run distributed training on a single predictor but instead runs two separate preprocessing and predictor pipelines on separate sets of edge devices and aggregates the predictions later. Since there is no research into this specific area and very little on distributed deep anomaly detection in general, this study aims to address the questions of impact on accuracy from transitioning anomaly detection to a distributed context.

2.2 Metrics

The metrics that are used to evaluate the models in this thesis are accuracy, precision, and recall. Accuracy is defined as the number of correct labels. It is calculated by:

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + False\ Positive + True\ Negative + False\ Negative}.$$

Precision is a measure of how many of the predicted positive labels are actually positive, but it does not factor in false negatives. Precision is given by:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}.$$

Recall is a measure of how many of the positive labels are captured by the model, but it does not penalize false positives. Recall is given by:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}.$$

The precision tends to be more important when the cost of a false positive is high while the recall is more important when the cost of misclassifying positive points is high. In the context of anomaly detection, they carry approximately equal weight since anomalies do not want to be missed but excessive false alarms are also undesirable.

The loss metric used during training and in calculating the threshold for the models is mean absolute error (MAE). MAE is a measure of the absolute value of the differences between true values and predicted values, calculated by:

$$MAE = \frac{\sum_{i=1}^{n} |y - \hat{y}|}{n},$$

where y is the true value, $\hat{y}$ is the predicted value, and n is the total number of predictions. ReLU, the rectified linear unit, is an activation function for DNN layers that is piecewise linear. ReLU outputs the input directly if it is positive and outputs zero otherwise. The main benefit of ReLU is that it solves the vanishing gradient problem, where gradients get so small the model becomes almost impossible to train, from which both sigmoid and tanh activation functions suffer [15]. Adam, or adaptive moment estimation, is the type of optimizer that is used when compiling the DNN models in this thesis. The optimizer is responsible for handling the gradients during training

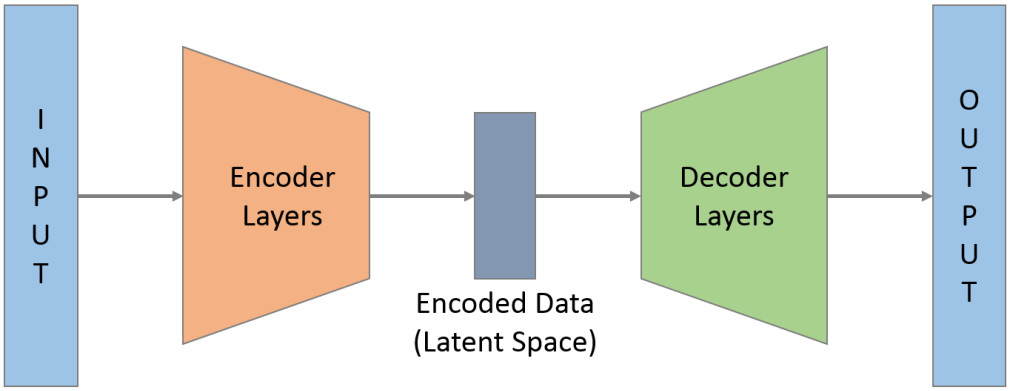

Figure 2.1: Diagram of autoencoder structure.

which allows the model to converge. Adam has good runtime performance and achieves good accuracy in many settings [16].

2.3 Autoencoder Structure

The anomaly detector network design in this thesis uses an autoencoder. An autoencoder consists of two parts, the encoder and the decoder. The encoder portion maps the input feature space to a condensed lower dimensional latent space, and the decoder maps that latent space to another space that has the same dimensionality as the input space to attempt to match the original input. Figure 2.1 gives a visual representation of this structure. Reconstruction error is the difference between the input to the encoder and the output of the decoder. The idea behind this is that an autoencoder trained to accurately reconstruct the normal data with a low error will have difficulty reconstructing data that is abnormal. The higher reconstruction errors can then be used to determine anomaly data points [17].

There are two types of layers used in the autoencoders in this study. Dense layers are fully connected layers that are useful for modeling latent features and can be applied to many prediction problems. Long short-term memory (LSTM) layers are a type of layer commonly used in recurrent neural networks. LSTM nodes store a history of the data that has passed through in order to help predict the next value. This makes them useful in predicting temporal data.

CHAPTER 3: DISTRIBUTED ANOMALY DETECTION

This section describes the configuration and hardware of the host server and edge server modules used to conduct this research experiment. It also presents the implementation of the anomaly detection system on the host and edge servers and provides details on the execution of the study. The host server is run on an older Asus model G751JY laptop. The edge servers are three Nvidia Jetson AGX Xavier modules. Additional hardware and software setup is required for each of the edge server modules to be able to perform the required deep learning tasks in coordination with the rest of the cluster.

3.1 Learning Cluster Configuration

3.1.1 Jetson AGX Xavier Modules

Each Jetson AGX Xavier module is distributed as an independent machine learning system on module (SoM) from Nvidia. They come with 32 GB of LPDDR4x RAM, 32 GB internal storage that can be upgraded, 8-core ARM CPU, 512-core Volta GPU with tensor cores, 2 NVDLA engines for deep learning acceleration, and a 7-way VLIW vision processor. The modules do not come with built-in Wi-Fi, so a separate card and antennae were installed in each module to make them wireless. The Wi-Fi card used is an Intel dual band wireless-ac 8265 with Bluetooth. The antennae used are IPEX MHF4 internal antennae for Wi-Fi WLAN cards.

These modules are flashed with a modified Ubuntu 18.04 OS provided by the Nvidia SDKManager which contains added packages to enable GPU accelerated deep learning and image processing in addition to running machine learning container environments. The development environments

used for testing are Docker containers that run isolated virtual environments with specific packages installed. The pre-built Linux4Tegra (L4T) ML Docker container provided by Nvidia runs TensorFlow 1.15 while the pre-built L4T TensorFlow 2.5 container does not contain JupyterLab or other machine learning libraries, so a customized Docker container is required. The custom TensorFlow 2.5 environment used on each module is built on the L4T base r32.6.1 Docker container and has the L4T r32.6.1 version of TensorFlow 2.5 and other required machine learning libraries installed along with JupyterLab to enable remote development. The code used to build the Docker container is modified from the public GitHub code available in [18]. The modules come with multiple power modes that operate at specified watts and downclock CPU and GPU speeds to meet the power constraints of that mode. For all experiments, the modules are set to the "MAXN" power mode which does not impose power constraints and allows full utilization of the CPU and GPU without downclocking.

3.1.2 Host Server

The laptop used for the host server is running Windows 8.1 and has 32 GB of DDR3 RAM, a 4-core Intel i7-4870HQ 2.5 GHz processor (8-core virtual CPU), and a Nvidia GTX 980m GPU with 1536 CUDA cores. The host server, in the role of the chief, needs to run a similar Python environment to the modules in order to communicate properly with them. For this, the Anaconda development environment is used. Communication to the modules is established using SSH to control them and initialize the Docker container environments. Once environments are instantiated, a TCP connection to a JupyterLab server run on the module can be opened in a browser on the host for programming and running of TensorFlow servers on the module. This

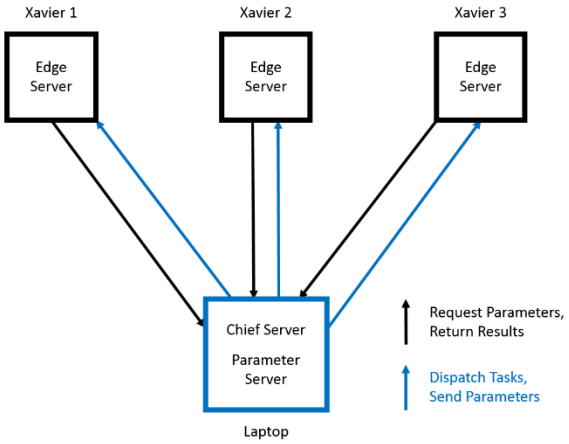

Figure 3.1: Diagram of the learning cluster configuration.

allows a user to perform all programming and monitoring from the host server machine. The host machine in this implementation also has the dual role of running a parameter server that stores the information and weights of the anomaly detection network and disseminates that information to the host and edge servers when they need to run any training or prediction tasks. Figure 3.1 shows a diagram of the cluster interactions during training and evaluation.

3.2 Dataset Preparation

3.2.1 KDD99 Dataset

The datasets used to evaluate the autoencoder are taken from the KDD Cup 1999 (KDD99) datasets available from the UCI machine learning repository [19]. More specifically, the 10 percent train data and 10 percent labeled test data (file named "corrected.gz" in the repository) are used. The datasets model a variety of simulated intrusions in a military network environment, and while they are not representative of modern network attacks, they are still usable for evaluating potential drops in accuracy between a centralized and a distributed version of a model. There are 494,021 entries

in the 10 percent labeled data and 311,029 entries in the 10 percent labeled test data. The original data is multiclass, so both the train and test datasets have their labels converted to a binary classification problem using 0 as a label for normal connections and 1 as a label for attack instances to be compatible with the anomaly detection problem.

There are 41 features, of which only six are retained: protocol_type, service, flag, logged_in, is_host_login, and is_guest_login. The categorical features protocol_type, service, and flag are converted to numeric labels. For the training dataset, the data objects are split into an 80% train set and a 20% validation set. The train set is further filtered to include only the normal connections to allow the model to determine a baseline score against which to compare the anomalous connections.

3.2.2 SWaT Dataset

The Secure Water Treatment dataset is a real-world dataset of recorded attacks on a water treatment testbed available from [20]. The specific version of the datasets used is SWaT.A2_Dec 2015. The dataset normal v1 is used to train the model, and the test dataset is attack v0. The data consists of the values from 51 sensors and actuators with the normal set containing 495,000 points and attack set containing 449,919 points. Feature selection is performed to reduce the dimensionality, eliminate features that do not model the attacks, and remove features that generate too much noise to provide a usable baseline. Continuous features are normalized, and the data is converted into a time series dataset using a lookback of 501 points and only taking every 20th point. The resulting time series has 26 timesteps in each sequence for training and prediction.

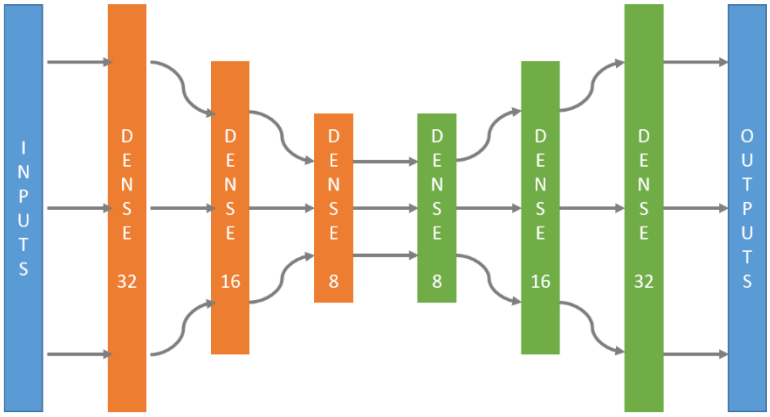

Figure 3.2: Diagram of the autoencoder model layers.

This is done to reduce the memory requirements and better gauge the future values since there are a considerable number of data points close together without much change.

3.3 Autoencoder Design and Evaluation

3.3.1 Centralized

The autoencoder is first constructed to run on a single machine to give a baseline for runtimes and accuracy comparisons. Figure 3.2 shows a diagram of the structure of the autoencoder that is used. The encoder portion uses three dense layers with ReLU activations with 32, 16, and 8 nodes for layers 1, 2, and 3, respectively. The decoder uses three dense layers with ReLU activations to decode and one dense layer for outputs. The number of nodes are 8, 16, and 32 for layers 1, 2, and 3 of the decoder, respectively.

The autoencoder is compiled using the Adam optimizer and MAE loss. It is then fit on the normal train data for 10 epochs with a batch size of 2048. The autoencoder is used to perform a reconstruction pass on the normal train data where it encodes and then decodes the data, and the

MAE loss is evaluated between the reconstructions and the original data for all entries. The mean and standard deviation of the loss are taken and added together to determine the threshold between normal and anomaly data. Any loss greater than or equal to the threshold is considered an anomaly. The model is evaluated on both the validation and test sets where accuracy, precision, and recall are recorded for each. Evaluations are performed on the sets three times each, one without using batching, one using the predict function with a batch size of 65,536, and one using a TensorFlow Dataset with a batch size of 65,536.

3.3.2 Distributed

The distributed autoencoder has the same structure as the centralized version, but the setup and the way that the training and evaluation steps are executed are different. A parameter server is initialized on the host machine to store and distribute the variables to the chief and workers. Each parameter server and worker runs a TensorFlow server with a specific environment configuration that includes its role and the IP addresses and port numbers of the other members of the cluster. For the purposes of this study, the parameter server is run in a separate JupyterLab notebook on the host machine and uses a different port from the chief. None of the code for the autoencoder model is stored on the workers; the workers receive their tasks from the chief and query the parameter server for the most recent variables for the model.

The dataset requires a specific format to be compatible with TensorFlow distributed training. The dataset must be packaged inside a function that produces a TensorFlow Dataset and appropriately shards, or horizontally partitions, the dataset for the number of input pipelines, one for each edge server worker. Inside this dataset function, the batching for training needs to be performed and is

set at a batch size of 2048 to keep it consistent with the training for the centralized autoencoder. A DatasetCreator object must then be declared that takes the dataset function as its argument. This DatasetCreator handles the division of the dataset during training.

The chief sets up a TensorFlow cluster resolver and a parameter server variable partitioner that are used to instantiate a TensorFlow ParameterServerStrategy object. Initialization of the strategy causes it to connect to and verify the online status of all members of the cluster. After this, the model can be created and trained.

The model must be declared and compiled within the context of strategy.scope in order for it to utilize the cluster; the Adam optimizer and MAE loss are kept the same from the centralized autoencoder. After that, the model is fit on the previously created dataset for 10 epochs and using steps per epoch equal to the floor of the number of data objects divided by the batch size. This results in a number of steps that is at most 1 less than the number of steps in the centralized version in order to approximate it as closely as possible for accurate comparisons. Due to being unable to retrieve the reconstructions directly because it is not yet implemented in the TensorFlow libraries, the threshold is calculated in the same manner as in the centralized version.

To be able to evaluate the relative performance of the distributed system, four evaluations are performed on both the validation set and the test set. The edge servers are used to perform the first evaluation with a batch size of 4096 for both the validation and test sets. Predictions are handled with the use of a ClusterCoordinator and require a distributed dataset to be created under the strategy and coordinator. A special dataset iterator and evaluation function are needed to run the predictions on the cluster. The TensorFlow library for performing these evaluations is still

experimental and does not guarantee visitation, meaning that not all data points will necessarily be used and some might be repeated, in the current version. As such, the most important information gained from this test is the runtime of the evaluation. The second evaluation is done solely on the host machine with the datasets in batches of size 4096 to compare the runtime and get the true accuracy of the distributed model. The third evaluation is performed using only the host machine and is done with a batch size of 65,536. The last evaluation is similar to the third but does not use any batching.

3.4 LSTM Autoencoder Design

The autoencoder in Section 3.3 is relatively simple, so to verify the effects that distributed training has on accuracy, a more complex autoencoder is created to predict on the SWaT dataset. This autoencoder uses LSTM layers, similar to [17] but without the dropout, to model the data because of its temporal nature. Four time-distributed dense layers are also added to increase the prediction accuracy by attempting to model correlations in the data that LSTM layers alone may miss. Figure 3.3 shows the structure of this autoencoder, including the number of nodes in each layer. The

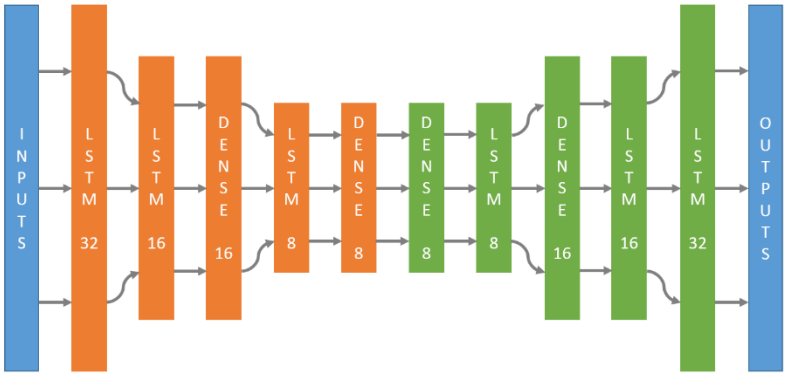

Figure 3.3: Diagram of the LSTM autoencoder model layers.

number of epochs, optimizer, and loss used to compile and train the model are the same as the previous autoencoder, but the training batch size is increased to 4096. Since the goal of this analysis is mainly focused on evaluating the accuracy, it does not do a detailed comparison of the training and prediction runtimes. Additionally, due to incompatible package versions in Anaconda causing LSTM layers to crash Python, one of the Xavier modules is used as the chief in the testing of this model with the other two modules as the edge server workers.

CHAPTER 4: ANALYSIS OF RESULTS

4.1 Results and Observations

4.1.1 Accuracy

The centralized autoencoder yields a best accuracy of 0.9560 on the validation set and 0.9234 on the test set. The distributed autoencoder yields similar accuracies with the best also being 0.9560 and 0.9234 for validation and test, respectively. The accuracies vary due to the randomness involved during training the models, but both can generate the same accuracies which all fall within a range of 0.2%. The full accuracy, precision, and recall results are listed in Table 4.1. During repeated testing, the distributed version achieves the average accuracy values more often than the centralized version. The centralized version seems marginally more stable during training based on observations of the graphs of the training loss over epochs, presented in Figure 4.1, where the loss of the distributed model has a higher tendency to fluctuate slightly as it converges.

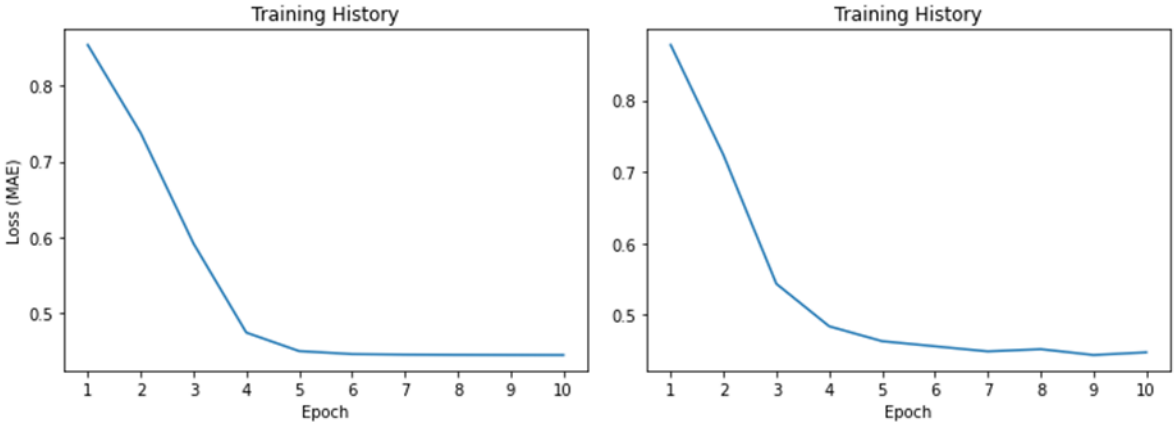

Figure 4.1: Graph of training loss over epochs for centralized (left) and distributed (right) autoencoders.

Table 4.1: KDD99 accuracy metrics for both centralized and distributed versions.

	Val Accuracy	Test Accuracy	Val Precision	Test Precision	Val Recall	Test Recall
Best Case	0.9560	0.9234	0.9576	0.9440	0.9891	0.9619
Average Case	0.9556	0.9226	0.9576	0.9440	0.9887	0.9609

The centralized LSTM autoencoder achieves an accuracy of 0.9665, precision of 0.9702, and recall of 0.7470 on the SWaT dataset. The distributed version of the LSTM autoencoder also reaches the same accuracy. As with the previous autoencoder, slight instability is observed in the distributed model's training loss as it converges, but this does not affect the final accuracy. Therefore, due to the tight range of accuracies and similar training loss curves across both autoencoder models, we have demonstrated that there is no significant accuracy change from the centralized model to the distributed model even as complexity of the model increases.

4.1.2 Runtime

The most significant difference between the two models is in the runtimes. For training, the centralized autoencoder takes approximately 4.5 seconds to complete 10 epochs while the distributed autoencoder takes approximately 32 seconds to complete the same number of epochs. While each epoch in the centralized version takes approximately equal time to train with the first only taking about twice as long, the distributed version has a considerable 20 to 25 second duration for the first epoch with subsequent epochs only taking about 1 second. This seems to indicate that there is a significant inefficiency in the overhead required to initially set up and prepare the model for training in the distributed environment. While the overall runtime is longer, the distributed

version does significantly reduce the resource load on the host machine. The CPU time for the centralized version averages 5.2 seconds, but the time for the distributed version uses only 2 seconds.

For the LSTM autoencoder, all centralized trials are performed on the Xavier module that is the chief in the modified cluster to get valid runtime comparisons. The centralized training takes about 37 seconds for the first epoch and 10 seconds for each subsequent epoch. The distributed variant spikes to around 275 seconds for the first epoch and then settles to 9 to 10 seconds for the remaining epochs. This shows that the overhead in preparing the model and cluster to perform the first epoch scales with model complexity, but there is a slight improvement over the centralized version after the first epoch even though there are only two workers in the cluster for this test. This indicates that, disregarding the initialization time, more workers do increase overall training runtime performance when compared to centralized training on a device with similar hardware. The CPU time for distributed training is about 48 seconds while the centralized training is about 108 seconds, indicating a decrease in the chief server load similar to the dense autoencoder.

The runtime for the prediction on the KDD99 validation and test datasets using the learning cluster varies greatly, but it is most typically around 14 to 16 seconds for the validation set and 14 to 25 seconds for the test set, both using a 4096 batch size. This is strange due to the test set being over three times as large as the validation set. It should be noted that, on repeated testing without restarting the kernel, the evaluation using the learning cluster will sometimes hang and result in runtimes of 1 to 4 minutes or longer. The cause of this behavior is unclear, but it is likely due to

Table 4.2: Average runtimes for prediction on validation and test sets using only the host for the distributed autoencoder.

Batch Size	Validation Prediction Time (s)	Test Prediction Time (s)
4096	3.3	11.5
65,536	0.425	1.12
None	0.175	0.29

a bug in the experimental TensorFlow libraries involved in coordinating the parameter server strategy during prediction, so those runtimes are not considered in estimates.

The evaluation of the datasets on the host only with a TensorFlow Dataset and batch size of 4096 is used to get a comparison to gauge the impact of the latency incurred due to the need to query the edge servers and the overhead from TensorFlow. The result is a consistent average of 3.3 seconds for the validation set and 11.5 seconds for the test set. This is more aligned with expected values given the relative sizes of the two datasets. Increasing the batch size to 65,536 yields an average of 0.425 seconds on the validation and 1.12 seconds on the test. Running the evaluation without any batching results in an average 0.175 seconds for validation and 0.29 seconds for test. These results are summarized in Table 4.2. When comparing test set prediction times with batch size 4096, the host takes less than half the time to perform the operation alone, but the distributed prediction results in about a 50% reduction to host resources even though it takes longer. This trend has a falloff with smaller datasets due to the coordination overhead; the validation set prediction requires more actual and CPU time using the cluster as opposed to using the host only on the distributed model.

Table 4.3: Batching average runtimes for prediction using the centralized autoencoder and batch size 65,536.

Batching Type	TensorFlow Dataset	Predict Function	None
Validation Prediction Time (s)	0.175	0.115	0.07
Test Prediction Time (s)	0.48	0.24	0.17

The centralized version runtimes on the predictions are recorded to get comparisons to determine impacts from TensorFlow batching overhead and latency in querying the parameter server. For these trials, a batch size of 65,536 is used. With a TensorFlow Dataset performing the batching, the average runtimes are 0.175 seconds for validation and 0.48 seconds for test. Making use of the model's predict function batching results in average execution times of 0.115 seconds for validation and 0.24 seconds for test. Lastly, with no batching, evaluation times are 0.07 seconds for validation and 0.17 seconds for test. These results are summarized in Table 4.3. Without batching, the validation set runs 1.66 times as fast compared to the predict function and 2.5 times as fast compared to using the Dataset. The no batching test set prediction sees a lower improvement of 1.41 times compared to the predict function and a higher improvement of 2.82 times compared to using the Dataset. This indicates that Dataset batching overhead does not scale linearly with the size of the data relative to no batching, and larger datasets may suffer greater performance impacts compared to batching within the predict function.

4.2 Discussion

We have shown that accuracy can be maintained when converting an autoencoder anomaly detector from a centralized architecture to run in a distributed environment, even with more complex models. Due to the way that the distributed training is handled in the backend, this result is applicable to most DNNs, not just autoencoders. However, care must be taken to implement the distributed model properly due to the increase in number of parameters that have to be manually managed by the programmer compared to a centralized implementation. Regarding accuracy, this shows promise for the future of DNN edge computing. For applications where runtime is not a critical concern and the priority is to free up server resources, this model is a usable solution. For other applications where time is a major factor, further development of the TensorFlow libraries is required before this model will be suitable.

The slight improvement in training time during later epochs of the distributed LSTM autoencoder and smaller gap between the KDD99 test set prediction times on the host only versus the learning cluster suggest that larger datasets will perform better. Sufficiently large datasets may see an improvement in prediction times using edge computing. Unfortunately, the current technology does not allow for advantageous training times with this implementation due to TensorFlow overhead on the first epoch. It is not clear what the cause of this overhead is, but optimization to the TensorFlow libraries for initializing a distributed model would help solve this problem. The work proposed in [11] may also be able to reduce the effects and improve the runtime. For batching, the runtime analysis performed on the centralized autoencoder indicates that batching within the predict function is well optimized and should be preferred over a manually batched

Dataset object when insufficient memory is available to evaluate without batching. Since the current public release of the distributed TensorFlow libraries do not yet implement a distributed predict function and batching is required for distributed prediction, the Dataset batching is another source of overhead, albeit much less significant compared to model initialization.

One positive result can be observed when comparing the prediction times on the host only without batching between the centralized and distributed versions: the query made to the parameter server to get the variables for the model adds very little overhead on a decently strong network. This means that if the overhead problems with the initialization of training can be solved, performance improvements on large datasets with complex distributed models may be possible. It is likely that, as TensorFlow continues to be developed, it will be feasible to gain significant improvements in runtime on big data and have more stability with the cluster coordination in deep learning on edge devices. Additionally, in the frequent real-world context of distributed input, performance can be improved by having edge servers run predictions directly without waiting for a task and data dispatch from a chief coordinator. This would reduce some of the overhead related to coordinating prediction tasks, and the edge servers can forward the results to the chief or act on them as needed by the implementation. The model can be saved locally on each device that performs predictions and periodically be updated to the most recent parameters if the model is retrained or trained on additional data.

4.3 Challenges

All experiments in this study are performed in a strong, static network environment. Even with network quality not being a factor, the TensorFlow libraries struggle to coordinate a small learning

cluster efficiently. Resolving the overhead of initializing distributed training is a major challenge by itself, but accounting for a varying or weak network quality would add more uncertainty to execution times and create more challenges for efficient coordination. TensorFlow also requires specifying the IP addresses and ports for each member of the cluster. This is acceptable for a static network context, but it is not able to handle a change in the network without manually updating the configuration for each server. It would require significant implementation changes and testing to be able to adapt a cluster for a dynamic network setting.

While TensorFlow handles worker task failure, if a worker throws an error during execution of a task, it will also stop the execution of task coordination on the chief. In this study, an error occurred where one of the edge servers had a corrupted install and always returned errors when attempting to train the distributed model. The device needed to be re-flashed and the Docker container re-built before the cluster could be utilized. Problems like this can interrupt the entire learning cluster and are even more substantial when remote devices are not readily accessible to be able to resolve the issues. Since the error reporting is also not transparent to the source, determining which device is malfunctioning and why becomes tedious. This also provides another attack vector for hackers who are trying to disrupt the DNN and prevent it from performing its function. Since the number of attacks on IoT devices are only increasing, this is a significant challenge and security concern to be able to protect devices from these types of vulnerabilities.

4.4 Future Direction

Since edge computing is still a recent, developing field, the current public technologies for distributed deep learning are not yet very refined. However, the desire to make better use of edge

computing resources is not going to disappear. Therefore, it would be worth investigating an efficient method to initialize the distributed model for the first epoch of training and prepare the cluster for distributed prediction, both of which are the dominant factors in the runtime for their respective tasks. This study has also restricted its focus specifically to the use of TensorFlow for its capability with GPU accelerated deep learning, but a study of implementations using other methods may yield promising results.

CHAPTER 5: CONCLUSION

The modern world has seen a rapid growth to the number of interconnected devices. This has brought about a massive influx of new data to the point where not all of it can be reasonably analyzed due to computational limitations. However, a large portion of the computing resources of edge devices are underutilized, so the desire to offload and distribute intensive computational tasks has been growing. One common demanding application is the use of deep neural networks. Due to the versatility and high accuracies of DNNs on many prediction tasks, they have become popular tools in modeling data where high accuracies are desired. As such, it would be beneficial to be able to run distributed DNNs on edge devices without sacrificing the accuracy.

Consideration the large increase in the number of IoT devices and the resource demand of DNNs, we propose an implementation of a deep learning anomaly detector using edge computing. This study shows that it is possible to distribute an anomaly detection model without a loss of accuracy using TensorFlow. The distributed anomaly detector uses fewer resources on the chief server when dealing with larger datasets, proving that this model is viable for solving the challenges of big data and high computational costs of DNNs on central servers.

While accuracy is maintained and the load on the chief is reduced, the resultant runtime is currently significantly longer than if the model is trained on a single machine. The present TensorFlow libraries add considerable overhead to the initialization of the distributed model and the first epoch of training, thus increasing the overall runtime. This problem can be solved with further development and optimization of the TensorFlow implementation for coordinating a learning cluster. Prediction tasks are also dominated by the coordination overhead on small datasets, but

larger datasets observe less of an impact. However, the chief server only needs to manage the cluster and dispatch tasks which reduces its resource usage. This results in a tradeoff to spend more time in order to reduce a chief server's workload. In a setting where runtime is not a deciding factor for an implementation, the current model we propose is a viable solution for freeing up resources on a chief server. Once the overhead issues are resolved, this model will be capable of reducing a server's workload without the drawback of disadvantageous runtimes.

REFERENCES

[1] A. Gillis, "What is the internet of things (IoT)?," *IoT Agenda*, 2019. https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[2] "Global number of connected IoT devices 2015-2025," *Statista*. https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[3] V. Shchutskaya, "Deep Learning: Strengths and Challenges," *InData Labs*, Jul. 27, 2018. https://indatalabs.com/blog/deep-learning-strengths-challenges.

[4] J. Kindervag, "What Is Anomaly Detection? An Introduction," *Splunk*, 2021. https://www.splunk.com/en_us/data-insider/anomaly-detection.html.

[5] D. E. Denning, "An Intrusion-Detection Model," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 222–232, Feb. 1987, doi: 10.1109/TSE.1987.232894.

[6] C. Cyrus, "IoT Cyberattacks Escalate in 2021, According to Kaspersky," *IoT World Today*, Sep. 17, 2021. https://www.iotworldtoday.com/2021/09/17/iot-cyberattacks-escalate-in-2021-according-to-kaspersky/.

[7] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers & Security*, vol. 28, no. 1–2, pp. 18–28, Feb. 2009, doi: 10.1016/j.cose.2008.08.003.

[8] G. Pang, C. Shen, L. Cao, and A. V. D. Hengel, "Deep Learning for Anomaly Detection: A Review," *ACM Computing Surveys*, vol. 54, no. 2, pp. 1–38, Mar. 2021, doi: 10.1145/3439950.

[9] M. Ahmed, A. Naser Mahmood, and J. Hu, “A survey of network anomaly detection techniques,” *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, Jan. 2016, doi: 10.1016/j.jnca.2015.11.016.

[10] G. Pang, C. Shen, and A. van den Hengel, “Deep Anomaly Detection with Deviation Networks,” *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul. 2019, doi: 10.1145/3292500.3330871.

[11] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, “A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters,” *www.usenix.org*, Nov. 2020. https://www.usenix.org/conference/osdi20/presentation/jiang.

[12] S. Teerapittayanon, B. McDanel, and H. T. Kung, “Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices,” *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2017, doi: 10.1109/icdcs.2017.226.

[13] M. V. Ngo, T. Luo, H. Chaouchi, and T. Q. S. Quek, “Contextual-Bandit Anomaly Detection for IoT Data in Distributed Hierarchical Edge Computing,” *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, Nov. 2020, doi: 10.1109/icdcs47774.2020.00191.

[14] C. Wang, Z. Zhao, L. Gong, L. Zhu, Z. Liu, and X. Cheng, “A Distributed Anomaly Detection System for In-Vehicle Network Using HTM,” *IEEE Access*, vol. 6, pp. 9091–9098, 2018, doi: 10.1109/ACCESS.2018.2799210.

[15] J. Brownlee, “A Gentle Introduction to the Rectified Linear Unit (ReLU),” *Machine Learning Mastery*, Jan. 09, 2019. https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[16] J. Brownlee, “Gentle Introduction to the Adam Optimization Algorithm for Deep Learning,” *Machine Learning Mastery*, Jul. 03, 2017. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[17] P. Grashorn, J. Hansen, and M. Rummens, “How Airbus Detects Anomalies in ISS Telemetry Data Using TFX,” *TensorFlow Blog*, 09-Apr-2020. https://blog.tensorflow.org/2020/04/how-airbus-detects-anomalies-iss-telemetry-data-tfx.html.

[18] D. Franklin (July 2021) Dockerfile.ml (Commit 43bdf7d) [Source code]. https://github.com/dusty-nv/jetson-containers/blob/master/Dockerfile.ml

[19] D. Dua and C. Graff, 1999, “KDD Cup 1999,” UCI Machine Learning Repository. [Online]. Available: https://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/

[20] S. Adepu, K. M. Aung, D. Wan, B. S. S. B. Liyakkathali, 2015, “Secure Water Treatment (SWaT),” iTrust, Centre for Research in Cyber Security, Singapore University of Technology and Design. [Online]. Available: https://itrust.sutd.edu.sg/itrust-labs_datasets/dataset_info/