CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

TINYML ANOMALY DETECTION

A thesis submitted in partial fulfillment of the requirements for the degree
of Master of Science in Computer Science

By

Mansoureh Lord

May 2021

The thesis of Mansoureh Lord is approved:

Jeff Wiegley , Ph.D.	Date
John Noga , Ph.D.	Date
Adam Kaplan , Ph.D., Chair	Date

California State University, Northridge

Dedication

I dedicate this thesis project to my husband Gary and my daughter Sara.

Acknowledgements

I would like to thank all my professors in the Computer Science Department at California State University, Northridge. Specifically, I would like to thank my committee Dr. Jeff Wiegley, Dr. John Noga, and Dr. Adam Kaplan. I would like to give a special thanks to Dr. Adam Kaplan for giving me the freedom to explore this topic, supporting me in my endeavors, and encouraging me on each step of this project.

Table of Contents

List of Figures

ABSTRACT

TINYML ANOMALY DETECTION

By

Mansoureh Lord

Master of Science in Computer Science

When manufactured parts of industrial machines fail in the field, the result can be quite costly. Rather than incur the cost of repairs and replacement parts, an anomaly detection AI can be used to constantly monitor machine health and notify operators whenever any disfunction is detected. This can help identify anomalies before they become severe enough to alert machine operators, or before a complete system failure occurs.

In this thesis, we explore machine learning on an embedded device to detect anomalies with sophisticated low-power neural networks. We employ the state-of-the-art TinyML framework, which enables the development and deployment of Edge AI. With TinyML, deep learning can execute on the same low-power computing platforms which collect sensor data in the field. We leverage this deep learning approach to detect physical anomalies as they occur, using a battery-powered embedded device with no network connection.

Our embedded platform is the Arduino Nano 33BLE, mounted to a Kenmore top-load washing machine. We read the Arduino accelerometer sensor to collect normal data from a balanced laundry load and abnormal data from an unbalanced laundry load, as they are washed by the machine. Then, we use normal data to train two different neural network models: autoencoder and variational autoencoder. These neural networks are used to detect accelerometer anomalies as unbalanced washing machine loads. After generating the model, we use TensorFlow Lite to load the model on the Arduino Nano 33BLE board. The Arduino device detects and indicates unbalanced washing machine loads with 92% accuracy, 90% precision and 99% recall using the autoencoder model.

When we use variational autoencoder model, we experience 66% accuracy, 74% precision and 80% recall. We also measured the battery life for the TinyML anomaly detector with autoencoder model as 20 hours with 5 V lithium batteries.

Chapter 1

Introduction

1.1 Anomaly Detection

Anomaly detection can be used to identify unexpected patterns in a stream of data. These unusual patterns are called *outliers* [21]. Anomaly detection has many use cases in business, system health monitoring, network intrusion analytics and fraud detection. Companies can save customers and businesses time and money by detecting anomalous data early on a given machine. Anomalies can be:

1) Point anomalies: An instance of data that is far off from the rest of the data like unusually high value credit card transactions. This thesis project seeks to identify point anomalies.

2) Contextual anomalies: The abnormality is context specific. These anomalies are common in time-series data, for example 68 F day in Alaska in January.

3) Collective anomalies: A set of data instances helps in detecting anomalies. For example a series of log in pages requests and typing in a password is probably an anomaly. The individual instances within a collective anomaly are not anomalous by themselves.

1.2 TinyML

Machine learning algorithms can be used to detect anomalies in data such as accelerometer samples. Moreover, recent advances in hardware have allowed embedded Microcontroller Units (MCUs) to execute complex machine learning algorithms on low-power embedded devices. TinyML is one such framework, enabling the execution of complex neural network models while consuming mere milliwatts of power.

As Figure 1.1 shows, TinyML sits at the intersection of embedded system design and machine learning. TinyML exports a machine learning model to a low-power embedded device, characterized by home-consumer products such as Alexa, or to even smaller sensor-driven devices such as autonomous vehicle controllers. The smallest and lowest-power end of this spectrum includes ap-

plications that operate on less than 1 mW of power, which can be powered for months by a small coin-battery. TinyML is defined as machine learning architectures, techniques, tools and approaches capable of performing on-device analytics for a variety of sensing modalities (vision, audio, motion, ...) at mW (low power) battery-operated device [3].

It is not always feasible to send every second of microcontroller data to a data center for analyzing the data and getting feedback. Communicating data to a data center requires network bandwidth, and also consumes battery power with each packet sent or received over the network interface. TinyML alleviates this concern by bringing AI to the source of the data, instead of data to the AI. This vastly reduces the overall communication, and addresses the aforementioned bandwidth and energy concerns. The TinyML paradigm proposes to integrate Machine Learning (ML)-based mechanisms within small objects powered by MCUs [16].

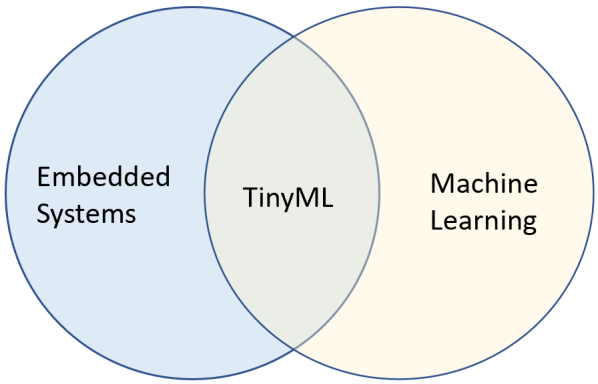


Figure 1.1: TinyML

The benefits of TinyML include the following:

1) Energy efficiency: MCUs use batteries therefore they are unconstrained by the availability of power outlets.

2) Low cost: This is the main reason that TinyML can be used in industries such as agriculture and e-health.

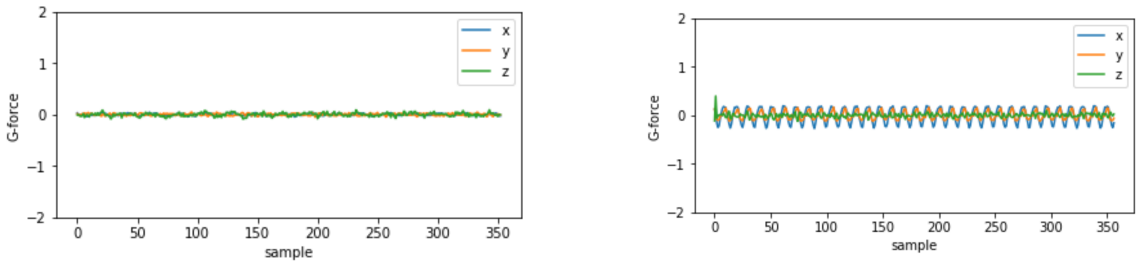
3) System reliability and security: Big data applications require that raw data be streamed over unpredictable and lossy wireless channels from end-devices to the cloud. This strategy is bandwidth consuming and prone to transmission errors and cyber-attacks [16]. TinyML performs data process-

ing and learning on an embedded device, avoiding the security concerns of data transmission, which include attacks on confidentiality and data integrity.

4) Latency: Performing local processing instead of uploading data on the cloud and waiting for a response will reduce delays to the operation.

1.3 Objective

We utilize a generic top-load washing machine Kenmore model 110.28974891 to detect point anomalies in unbalanced spin dry cycle. Figure 1.2a shows normal 3-axis accelerometer data for low speed washing and Figure 1.2b shows data for fast speed spin dry with a balanced load. Figure 1.3 shows abnormal 3-axis accelerometer data for unbalanced spin dry cycle.



(a) Low Speed (b) Fast Speed

Figure 1.2: Normal Samples

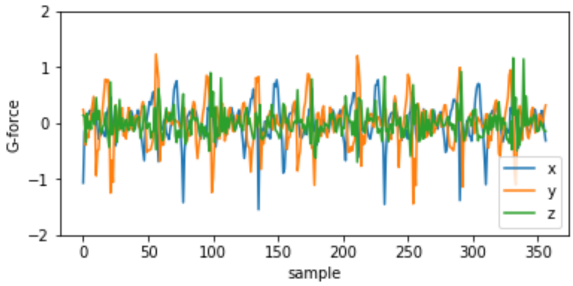


Figure 1.3: Abnormal samples

In this context, a simple amplitude test is not enough to determine an imbalanced load. In such a situation, we don't necessarily know what an anomaly looks like prior to analyzing data. Moreover there is no way to determine if a given accelerometer sample is anomalous with a simple value-check, as surrounding samples must also be analyzed to determine if this data is part of a larger contextual anomaly.

In this data, an anomaly is not necessarily captured by an instantaneous sample point, and even unusual samples may represent outliers without indicating an imbalanced laundry load. For such a situation, a machine learning approach is suitable to detect imbalanced laundry loads.

Chapter 2

Related Work

2.1 Manufacturing Defect Detection

There have been many recent studies attempting to monitor and control the manufacturing process using artificial intelligence. The goal is to identify measurements that are different from normal samples. Such samples may indicate a system anomaly, and can help in detecting defects in order to alert operators.

Feng Han et al used Convolution Neural Network (CNN) to extract a defect image feature and trained the network to learn the defect class label and its position [10]. Rajhans Singh et al used CNN for defect detection and Generative Adversarial Networks (GANs) to create defect modes that are difficult to create manually [17]. Nakazawa et al explained how to detect abnormal wafer map defect patterns by using deep convolutional encoder-decoder neural network [14].

2.2 Medical Diagnosis

Data points like X-ray, ECG and MRI indicate health status. Usually these data points are collected by a medical device and anomaly detection can determine abnormal events in this data, which could be harmful or even life-threatening to a patient if untreated.

Ines Krissaane et al used classification approaches to predict sepsis before clinical diagnosis. They trained a model on patients free of sepsis with an autoencoder neural network and then used reconstruction error to extract features. They used random forest classifiers to learn the most important features with which to classify the patients. Then, they combined the features from both approaches with a variety of standard classification models [11].

2.3 Fraud Detection

Fraud detection is essential for the financial services industry. Fraud can be detected by comparing a transaction with historical transactions, with abnormal transactions being flagged as potentially fraudulent.

Olena Vynokurova et al proposed a hybrid system of machine learning for fraud detection that uses both an anomaly detection subsystem (based on unsupervised learning) and an interpretation subsystem (based on supervised learning) that can determine the type of anomaly. This is a high speed real time data processing system [19]. Chien-Hung Chang identified fraud event as the outliers of the reconstruction error of a trained autoencoder. The loss of false negative frauds can be evaluated by the thresholds from the percentiles of reconstruction error after training data on the normal data [6].

2.4 Network Intrusion Detection

Network Intrusion Detection and Prevention Systems (IDPSs) use machine learning to flag suspected network attacks and, if possible, stop these attacks by employing countermeasures. Such countermeasures include adjustments to network firewall configuration or re-calibration of networking thresholds.

Qiang Duan et al explained how security threats are a severe concern for the WiFi protocol. They present a CNN intrusion detection algorithm for WiFi networks. As in this thesis, Duan et al solved their overfitting problem with the dropout technique [8], which will be elaborated in a subsequent chapter. Dashun Liao et al designed a network intrusion detection by using a GAN model. The model increases the training set by continuously generating samples. They worked with a supervised learning multi-classification model [13].

2.5 Deployment of TinyML

Miguel de Prado et al leveraged tinyCNNs to control a mini-vehicle, which learns by imitating a computer vision algorithm. They reduced latency by over 13 times and energy consumption by 92% with using GAP8, a parallel ultra-low-power RISC-V SoC [7].

Xiaying Wang et al presented an open-source toolkit built upon the fast artificial neural network (FANN) library to run lightweight and energy-efficient neural networks on MCUs based on both the ARM Cortex-M series and the novel RISC-V-based parallel ultralow-power platform [20]. It also represents that the parallel implementation on the octa-core RISC-V platform reaches a speedup of 22 times and a 69% reduction in energy consumption and with respect to a single-core implemen-

tation on Cortex-M4 for continuous real-time classification.

In previous papers, deep learning approaches, especially neural networks are popular algorithms for anomaly detection. These approaches are designed to work with high dimensional data. Also deep learning methods are suitable for nonlinear relationships within data and in the most cases achieve high accuracy. In some papers, they used a hybrid system that is based on both supervised and unsupervised learning.

Our study uses only unsupervised learning on two neural networks, autoencoder and variational autoencoder with one hidden layer. These networks detect anomaly events as the outliers of the reconstruction error of the model. The algorithms are not deep learning since they have only one hidden layer.

Chapter 3

Methodology

3.1 Hardware

We selected the Arduino Nano 33 BLE Sense as the target MCU because this board is supported by TensorFlow Lite [4] for TinyML usage. The Nano 33 BLE comes with a 9-axis inertial measurement unit (IMU) which means that it includes an accelerometer, a gyroscope, and a magnetometer with 3-axis resolution apiece [2]. The accelerometer sample rate is 119.00 Hz. Another way of saying this is that a sample is collected from the accelerometer every 8.4 ms. Figure 3.1 shows the Nano 33 BLE board, pictured with its X, Y, and Z accelerometer axes labeled.

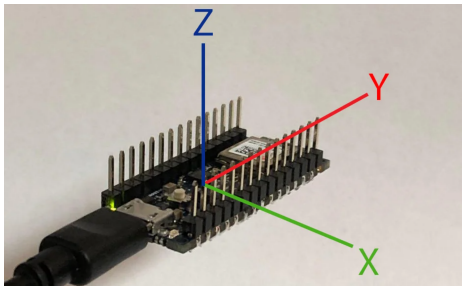


Figure 3.1: Arduino Nano 33 BLE Sense

3.2 Software

The code for this project was written in Python using the TensorFlow [4] platform on Google Colab as the machine learning engine. Colab is a free Jupyter notebook environment that runs entirely in the cloud [1]. TensorFlow is Google's open source machine learning library. It was developed by Google and released to the public in 2015.

The Arduino Nano was programmed with C and C++. The TensorFlow Lite library was used for exporting the model to the Arduino board. Also, Netron was used as a neural network visualizer.

3.3 TensorFlow Lite

TensorFlow Lite is an open source deep learning framework for on-device inference [4]. TensorFlow Lite for MCUs is written in C++ and the framework is available as an Arduino Library.

Google built TensorFlow Lite in 2017 for running neural network models efficiently and easily on mobile devices. In 2018, they built a specialized version of TensorFlow Lite for embedded platforms because they needed to create models that would fit within 20 KB or less on low-power MCUs. Although the TensorFlow Lite framework cannot be used to generate a model nor train a model, it can reduce the size and complexity of a pre-trained model by dropping the features that are less common. TensorFlow Lite supports running inference operations on a model that was trained on Colab. Some data types, such as double, are not supported by the framework.

TensorFlow Lite has two main components, as follows:

1) TensorFlow Lite Converter: This converts a TensorFlow model to a space-efficient format for use in the restricted memory footprint of an embedded device. It uses the TensorFlow Lite converter's Python API to write the Keras model to the disk in the form of a *flatbuffer*, a special file format for space-efficiency. In this project, the size of the autoencoder model was reduced from 4096 bytes to 1424 bytes after converting to TensorFlow Lite.

Tensorflow Lite does not supports all Tensorflow operators, therefore not every model is convertible. To convert those models, we can enable certain TensorFlow operators (also known as *ops*) in the tensorFlow Lite model. However, this increases the TensorFlow Lite interpreter binary size because it requires loading of the core TensorFlow runtime. Our variational autoencoder model was not supported for direct conversion by TensorFlow Lite. We enabled TensorFlow Lite ops to convert the model, but this increased the model size from 4096 bytes to 4652 bytes.

Tensorflow Lite converter can also apply optimization to the model, which can reduce the size of a model. A smaller model will correspondingly use smaller storage size, smaller download size, and smaller memory usage. Quantization is one such optimization method. The weights and biases in a neural network model are stored as 32-bit floating-point numbers. Quantization reduces the precision of these numbers so they can fit into 8-bit integers. Although this reduces the size of the model, it may also reduce the accuracy of the model.

Convert to a C array: Most MCUs don't have a file system. Thus, the model must be provided as a C source file that can be included in the cross-compiled binary and loaded into MCU memory.

The UNIX command *xxd* can convert a TensorFlow Lite model to a C char array. The command *cat* can be used to print the entire resulting array, as shown in part by figure 3.2.

```
unsigned char g_model[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
  0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00, 0x00, 0x00,
  ...
  0xff, 0xff, 0xff, 0xff, 0x03, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff, 0xff,
  0x04, 0x00, 0x04, 0x00, 0x04, 0x00, 0x00, 0x00
};
unsigned int g_model_len = 1424;
```

Figure 3.2: C Source File of TensorFlow Lite Autoencoder Model

We copied this file from Colab and pasted it in our Arduino Sketch in the file model.cc. We changed the deceleration to *const* for better memory efficiency on Arduino platform.

2) TensorFlow Lite Interpreter: This executes a converted TensorFlow Lite model on the MCU by using space-efficient operations. As a preliminary, the TensorFlow Lite library must be installed within the Arduino IDE, and the TensorFlow Lite components needed by a given project must be included by its sketch file(s). Before setting up the interpreter, two steps must be taken:

Creating an AllOpsResolver: We must create an instance of AllOpsResolver as *static tflite::AllOpsResolver resolver;* This class is defined in *all_ops_resolver.h.* It enables all of operations that are available to TensorFlow Lite for MCU interpretation.

Define A Tensor Arena: We need to allocate an area of memory for storing the model's input, output, and intermediate tensors. In this project, we allocated an array size of 2048 bytes. It's difficult to know how much memory we need. We can scale n in n*1024 by trial and error to find the correct size. We initialize the size by choosing a large n, and then we reduce n until our model stops working. The last number that worked is the right n. In our case, n is 2.

After creating an AllOpsResolver and defining a tensor arena, we create an interpreter to run the model with as:

static tflite::MicroInterpreter static_interpreter(model, resolver, tensor_arena, kTensorArenaSize, error_reporter);

This class is the most important part of TensorFlow Lite for MCU as it will execute our model on

live input data collected from sensors. We next allocate memory from *tensor_arena* for all of the tensors defined by the model with: *interpreter.AllocateTensors();*

3.3.1 TensorFlow Lite Inference

Inference is the process of executing a TensorFlow Lite model on a device to make predictions based on input data and it consists of these steps:

1)Loading The Model: We convert and load *.tflite* model as C array into memory. This contains the model execution graph.

2)Transforming Data: Usually the raw input data is not match the model input data. In this project, we needed to normalize the raw data before presenting it to the model.

3)Running Inference: We use TensorFlow Lite API to execute the model. We performed the steps of building the interpreter and allocating tensors as explained earlier in this chapter.

4)Interpreting Output: After obtaining results from the model inference, we need to interpret the tensors in a way that is useful for our application. In this project, we calculate mean absolute error between output and input tensors and compare it with a threshold value to detect anomaly input.

3.4 Battery Life

Battery life is a major consideration in development of a TinyML product that can operate for a reasonable amount of time[15]. Ability to classify industrial anomalies in a low power embedded application depends on how much energy it can use or store. In other words, the battery life should be sufficiently long-lasting to achieve success, perhaps even over multiple laundry loads.

The Arduino Nano power options are 5 VDC (volts of direct current) via USB connector or 4.5 to 21 VDC on pin 15. The test shown in figure 3.3 was performed to determine at what voltage the Arduino Nano stops working. First we loaded the provided demo program *blink* onto Arduino Nano as a minimal program. This program indicates (via blinking LED) that the Nano is running. Next we connected a ground (GND) wire to pin 14 of the Arduino Nano board, and another wire to pin

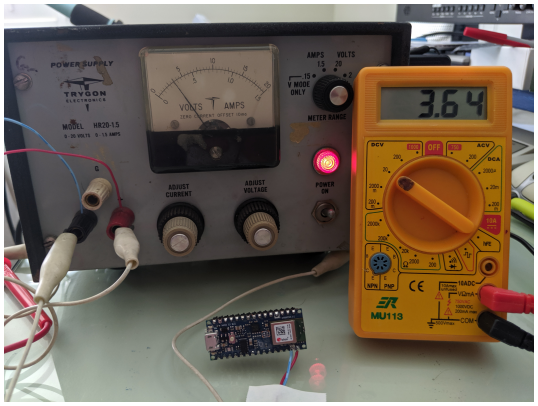


Figure 3.3: Voltage Test

15 to behave as the input voltage (VIN). Then we used a variable-voltage DC power-supply and a digital voltmeter to reduce the voltage until the Nano stops running. The MPM3610 synchronous step down converter on the Arduino Nano board shuts down at 3.64 VDC. This was observed as the power LED went out and the blink program stopped. Then we slowly raised the input voltage to 3.87 VDC, which allowed the regulator to start and the blink program to resume. We hence use the 3.64 VDC value to aid in determining battery lifetime of our models on this platform.

Chapter 4

Preliminaries

4.1 Data Science

4.1.1 Gathering Data

Our Arduino sketch was programmed to collect washing machine vibration data (x, y, z) from an Arduino Nano 33 BLE Sense accelerometer sensor through its USB communication port. Normal data was gathered from balanced loads at low, medium and fast speeds. Abnormal data was gathered from an unbalanced load at fast spin dry cycle. Normal and abnormal data was saved in two different comma-separated values (CSV) files.

4.1.2 Cleaning Data

Good data quality in the context of machine learning refers to attributes such as completeness and absence of duplicates. After loading up data in Pandas, null and duplicated data was removed from the normal and abnormal data frames. Figure 4.1a shows a sample of 10292 normal data and Figure 4.1b shows a sample of 1070 abnormal data.

	X	Y	Z
0	0.003	0.072	0.805
1	0.138	0.024	1.384
2	0.014	0.046	0.854
3	0.036	0.044	0.995
4	0.035	0.082	0.948

(a) Normal Data frame

	X	Y	Z
0	-0.097	-0.482	0.949
1	-0.179	-0.418	1.485
2	-0.340	0.165	0.874
3	-0.234	0.260	0.590
4	-0.099	0.233	0.863

(b) Abnormal Data Frame

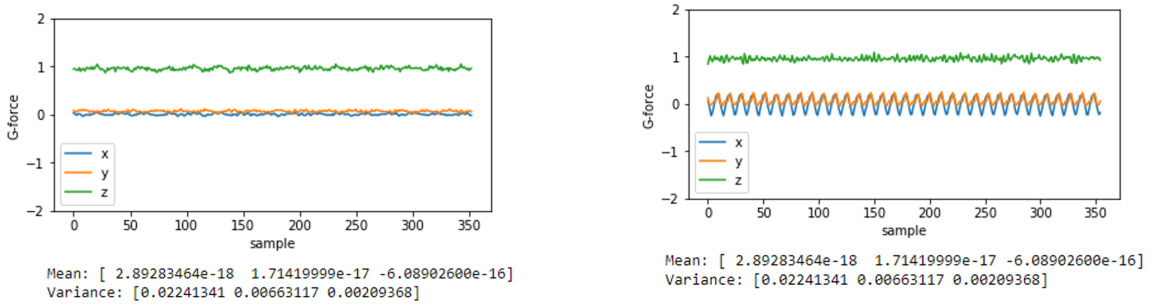
Figure 4.1: Normal and Abnormal Data Frames

To work with TensorFlow platform, we changed Panda's data frame for both normal and abnormal data points to a NumPy array. For better results, we shuffled each set of data separately with *np.random.shuffle(data)*. Then we divided the normal data set into two subsets, eighty percent for

the training set and twenty percent for the testing set. We use only the normal data set for training. The subsequent testing data set should be a mix of normal and abnormal data points. Therefore, to create a testing data set we added abnormal data to normal test data and shuffled them.

4.1.3 Visualizing Data

Data visualization is the graphical representation of data. Charts and graphs provide a way to see and understand patterns and outliers in the data. Data visualization can give us important information about anomalies. In Python, visualization can be performed by a variety of open-source packages such as the Matplotlib package. Matplotlib, a Python alternative to Matlab, is a 2-D plotting library that helps in visualizing figures.



(a) Low Speed (b) Fast Speed

Figure 4.2: Normal Samples

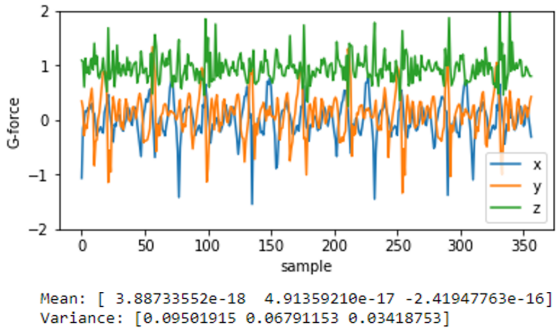


Figure 4.3: Abnormal samples

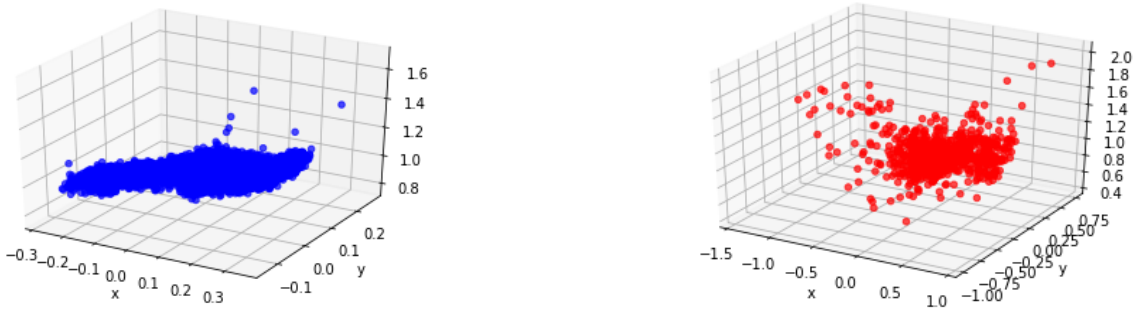
4.1.3.1 Plot

Figures 4.2 and 4.3 show accelerometer Gforcet (x, y, z) for 350 sample data points with mean and variance from Python Notebook. We can subtract the mean of each sample set from the mea-

surement in the set to remove DC from each graph. The result will be figures 1.2 and 1.3. These revised figures will make it easier to visualize accelerometer vibrations.

4.1.3.2 Scatter Plot

A scatter plot uses dots to represent values for two variables. They are used to observe relationships between variables. We can create a 3D scatter plot for three variables (x, y, z) in Python. Figure 4.5 divides data points into two groups based on how closely sets of points are clustered. Blue is for normal and red is for abnormal data points. Also, maintaining two separate clusters will be helpful to generate a machine learning model with normal data from the first set and then to recognize anomalous data from the second set.



(a) Scatter Plot for Normal Data

(b) Scatter Plot for Anomaly Data

Figure 4.4: Scatter Plots

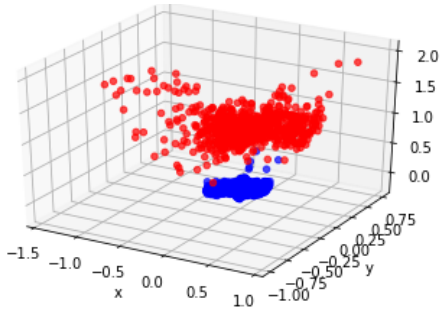


Figure 4.5: Scatter Plot for Normal vs Anomaly Data

4.1.3.3 Correlation

The correlation matrix gives us the information about how the two variables are related in direction and magnitude. It is based on the Pearson correlation coefficient. Panda's *dataframe.corr()* is used to find the pairwise correlation of all columns in the dataframe. In figure 4.7 the correlation

between X and Y for normal data is a positive correlation because X increases as Y increases.

	X	Y	Z
X	1.000000	0.774891	0.057137
Y	0.774891	1.000000	-0.188163
Z	0.057137	-0.188163	1.000000

Figure 4.6: Correlation Scatter Matrix

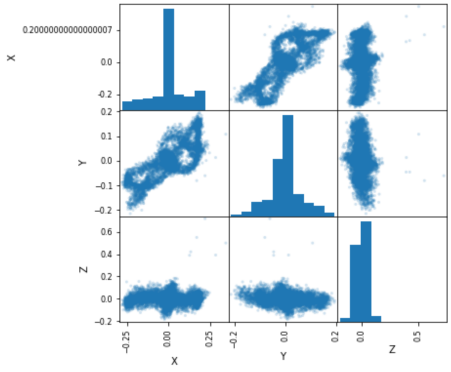


Figure 4.7: Correlation for Normal Samples

4.1.4 Data Normalization

Many machine learning methods work better after the data has been normalized. The common method for feature normalization is *simple re-scaling*. The goal is to re-scale the data along each data dimension so that the final data vectors will be in the range [0, 1]. In this project, we used *min-max scaling* by subtracting the minimum value and dividing by the range(maximum value - minimum value) of each column. Now, each new column has zero as minimum value and one as its maximum value. We performed this method on both training and testing data sets, and also on the abnormal data set. figure 4.8 shows part of the normalized normal data.

```
tf.Tensor(
[[0.13278855 0.17773238 0.64964247]
 [0.12257405 0.19152196 0.61644536]
 [0.10010214 0.17058222 0.65679264]
 ...
 [0.14044943 0.18488254 0.6113381 ]
 [0.02196118 0.14198162 0.59652704]
 [0.13840653 0.18232891 0.6159346 ]]
```

Figure 4.8: Sample of Normalized data

Chapter 5

Approaches

In this section, we summarize the various well-known approaches to using machine learning for anomaly detection.

5.1 Supervised Learning

Supervised anomaly detection requires a data set that has labeled all data as *Normal* or *Abnormal*. Here anomaly detection can be like a classifier, via which the machine can learn and then predict if an input is an anomaly or not. The problem with this approach is that in most use cases, the amount of anomalous data is much less than the amount of normal data available for training a model. Also, sometimes there are many anomalous classes which are not known a priori, which will not be detected. Therefore, supervised learning can classify anomalies when all types of abnormal data have been provided and labeled in advance, but it cannot detect previously unseen anomalies.

5.2 Unsupervised Learning

Unsupervised learning is a more popular approach for anomaly detection than supervised learning because the quantity of labeled anomalous data is much smaller than that of labeled normal data in most of its use cases. Here, data is not pre-labeled as *Normal* or *Abnormal*. Rather the algorithm learns the structure of the input features in order to determine what constitutes an anomaly.

5.3 Semi-supervised Learning

Semi-supervised learning uses a large amount of unlabeled normal data with a small amount of labeled abnormal data. This is also a suitable approach when the amount of normal data is much greater than the amount of abnormal data. This method has been heavily applied to network intrusion detection [18].

5.3.1 Learning Normal Behavior

The underlying strategy for most approaches to anomaly detection is based on their dis-similarity to a model that represents the normal behavior [5]. This is usually performed in two steps. The first step is training a model with unlabeled normal (non-anomaly) data and building a model of normal behavior. Based on this model, any labeled test data that is a measure of deviation from normal behavior will be anomaly data. The second step is to select an error threshold. Any future data that is encountered and produces an error above this threshold would tag as anomalous data. The sensitivity of anomaly tagging can be tuned by changing this threshold. In this study, we used unsupervised approach to learn normal behavior of the washing machine appliance, helping us to detect the anomaly circumstance of unbalanced laundry loads.

Chapter 6

Application Architecture

Our application obtains input, runs inference and processes output to inform the user if an anomaly has occurred. Our model takes preprocessed Arduino accelerometer data (x,y,z) as input data. Then it runs TensorFlow Lite inference as explained in Chapter 3 and determines if the input represents an anomaly. When an anomaly is detected, the application will activate an onboard LED to inform the user of unbalanced laundry. Figure 6.1 shows the structure of our TinyML anomaly detection application.

The application main parts are:

Main loop: The application runs in a continuous loop. Since the model is small and simple and it runs very fast (multiple inference per second), we delayed the inference with *millis()* function for 5000 ms.

Accelerometer handler: This capture data from the accelerometer and writes it to the model's input tensor after applying normalization.

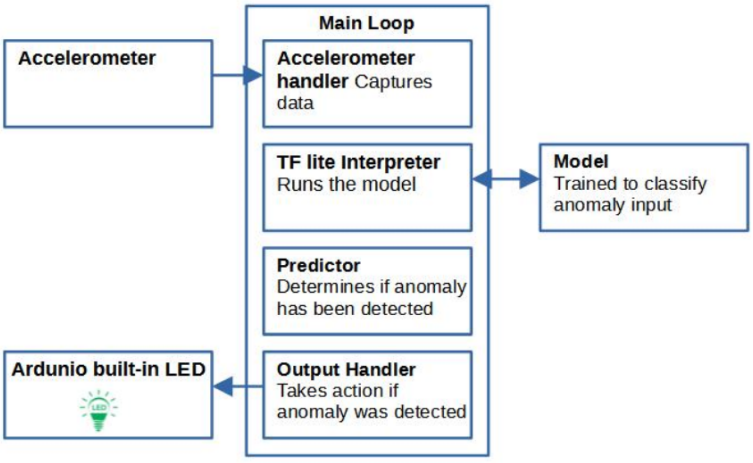


Figure 6.1: Application Structure

TF Lite interpreter: The interpreter runs TensorFlow Lite model as explained in chapter 3.

Model: The model is included as a C array and executes upon the interpreter. The model's size, once converted, consumed a mere 1.42 KB.

Predictor: This component takes the model's output and computes the mean absolute error(mae) between model input and output. It predicts input data as anomaly if mae is bigger than a given threshold.

$$mae = (\frac{1}{n}) \sum_{i=1}^{n} |y_i - x_i| \tag{6.1}$$

Output handler: This component activates (turns on) an onboard Arduino LED when an anomaly is detected.

Chapter 7

Algorithms

7.1 General Structure of an Artificial Neural Network(ANN)

An Artificial Neural Network (ANN) is a networked computing model based on the structure and observed function of the animal brain. An ANN is comprised of interconnected nodes called *neurons*, as these emulate the function of the biological neurons in the brain. Much as biological neurons collaborate to examine incoming signals, and filter them to decide which to pass onward, the neurons of an ANN are connected in a manner that allows them to weigh the relative importance of input data. Each neuron is characterized by its weight, bias and activation function.

Figure 7.1 shows a neural network structure. It represents interconnected input and output units with some hidden layers between them. Each connection between nodes has an associated weight. If a neural network has more than one hidden layer, it is referred as deep neural network(DNN). Bias is an extra input to each neuron and it is always has a value of one(1). Thus, even when all the inputs to a neuron are zero, there is still an activation in the neuron.

ANN computes a weighted sum of inputs, then applies an activation function to the sum to make outputs. The activation function outputs move to the next hidden layer and the same process repeats. This process is known as *forward propagation*.

An ANN takes a set of input data points, trains the neurons to recognize patterns in the input data, and predicts the output. Then it measures the network's output error. The network output error is the difference between predicted output and actual output of the network. Then, it goes through each layer in reverse to measure the error contribution from each connection, and update the connection weights to reduce the error by using an algorithm like gradient descent. This process is known as *back propagation*.

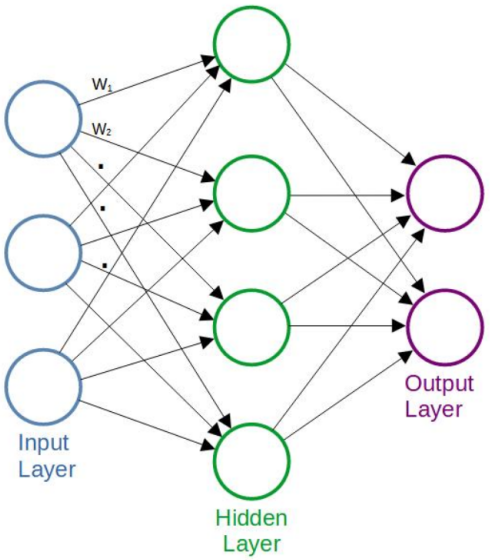


Figure 7.1: Neural Network

7.1.1 Activation Function

Without activation function, neurons will perform a linear transformation on the inputs by using weights and biases. In this circumstance, the ANN is not able to learn complex patterns from the input data. To learn non-linear patterns, ANNs are often given an activation function. If the input to the activation function is greater than a threshold, then the neuron will be activated, otherwise it is not.

There are many possible activation functions adopted in practice. We elaborate on those that we utilized in this study.

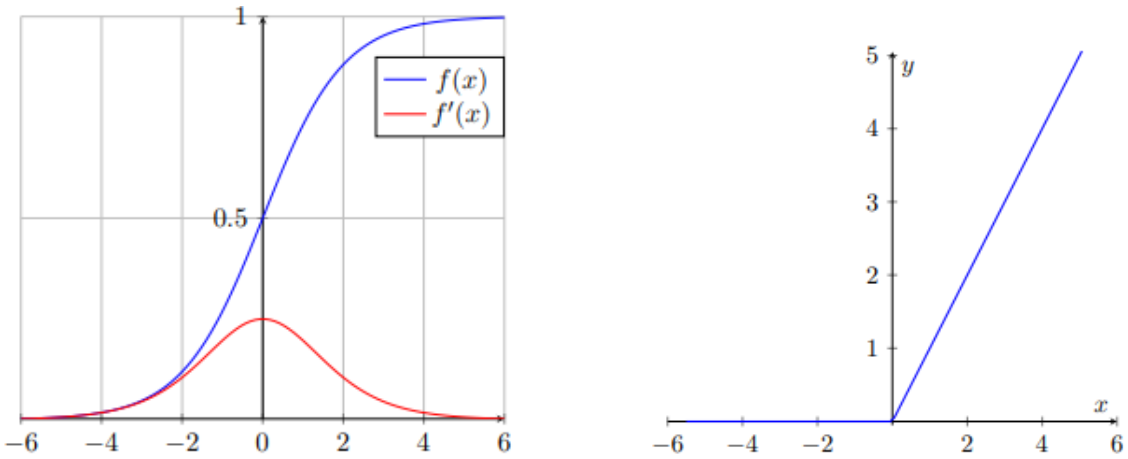
7.1.1.1 Sigmoid

Sigmoid is a non-linear activation function that transforms the values between the range 0 and 1. When neurons have a sigmoid function as their activation function, their output is non-linear. Equation 7.1 is the mathematical expression for sigmoid and 7.2 is the derivative of sigmoid function.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{7.1}$$

$$f'(x) = sigmoid(x) * (1 - sigmoid(x)) \tag{7.2}$$

Figure 7.2a represents both functions. The sigmoid function is not symmetric around zero, therefore output of all the neurons will be of the same sign. The gradient values for the derivative of the sigmoid function are very small for values less than -3 and greater than 3. Gradients are calculated to update the weights and biases during the back propagation process. The ANN does not learn when the gradient value is too small or zero.



(a) Sigmoid Function and its Derivative

(b) ReLU

Figure 7.2: Activation Functions

7.1.1.2 Rectified Linear Unit(ReLU)

ReLU is a non-linear activation function that ensures that not all neurons are activated at the same time. As Figure 7.2b shows the neurons will only be activated if the output of the linear transformation is greater than zero. So, the ReLU function is more computationally efficient compared to the sigmoid function.

$$g(x) = \max(0, x) \tag{7.3}$$

$$g'(x) = \begin{cases} 1 & x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{7.4}$$

Equation 7.3 is the mathematical expression for ReLU and equation 7.4 is the derivative of ReLU function. ReLU function is one of the commonly used activation functions in the hidden layers of an ANN [12].

7.2 Autoencoder

Autoencoder is a neural network that is designed to learn from representations of the input that has a lower dimension than input data. Autoencoder is a strong feature detector since the input data is unlabeled (unsupervised). It learns to copy its input to its output. This may seem easy work but we can limit the size of internal representation or add noise to the input data and train it to make output like input data by learning identity function. The autoencoder cannot reconstruct data different from the training data and this makes the autoencoder a very useful algorithm to detect anomalous data [9].

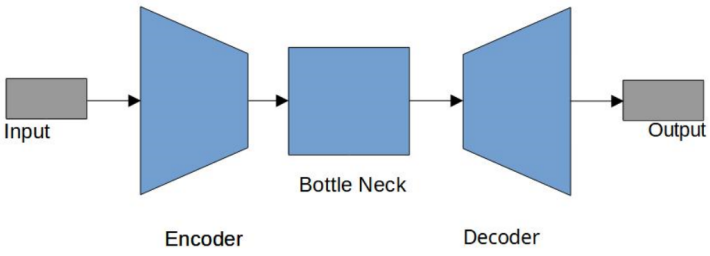


Figure 7.3: Autoencoder

As figure 7.3 shows, an autoencoder consists of two parts: *encoder* and *decoder*. The encoder converts the input data to an low-dimensional internal representation (*bottleneck*) then the decoder converts the low-dimensional internal representation to the output data. The number of neurons in the output layer must be the same as the input layer. Usually, the output is called *reconstruction* because the autoencoder reconstructs the input as output. *reconstruction loss* measures the differences between our original input and the consequent reconstruction. The *loss function* (L(x,y)) represents the mean square error between output(y) and input(x).

$$L(x, y) = \|x - y\|^2 \tag{7.5}$$

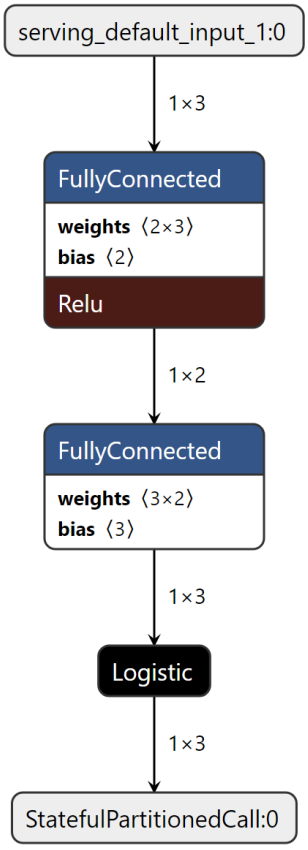
Autoencoder can have several hidden layers. In this case it is called a *stacked autoencoder* or *deep autoencoder*. Having more hidden layers helps to learn better. However, there is a drawback to making an autoencoder too powerful. A very strong autoencoder will reconstruct the exact training data very well but will fail to reconstruct previously unseen data. In this case the model is memorizing, rather than learning. The simplest architecture for constructing an autoencoder is to limit the size of internal representation layers, making the network *undercomplete*. Then the model can be forced to learn important features from input data and consequently be able to reconstruct unseen data.

7.2.1 Autoencoder Implementation

Figure 7.4a is a visualization of our autoencoder model architecture. It uses Keras API and has only one hidden layer with two neurons. The raw normal data of the accelerometer (x, y, z) was presented to the autoencoder as input. Then the encoder learned to compress the data set from three dimensions to the latent space, and the decoder learned to reconstruct the original input (x, y, z) from two-dimensional latent space. The ReLU activation function was used for encoder because it is faster to compute than other activation functions. The sigmoid activation function was used for the decoder in order to provide output values between zero and one. 7.4b represents a Keras table that summarizes our model architecture. This figure shows all the layers with their shapes and the number of their parameters, witch is another term for weights and biases. The table does not show the input shape as it actually exists in our implementation (None, 3).

After creating the model, it was configured with losses and metrics with *model.compile()*. We used *adam* as optimizer and mae as the loss function for this model. Then we trained the model with *model.fit()*. Since we can not pass the entire data set into the neural network at once, we divide data set into number of *batches* or sets. Also we choose *epoch* that is the number of complete passes through the training data set.

Autoencoder was trained using only normal data, but was evaluated using the test set consisting



(a) Autoencoder Model Structure

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
sequential_6 (Sequential)    (None, 2)                 8
_________________________________________________________________
sequential_7 (Sequential)    (None, 3)                 9
=================================================================
Total params: 17
Trainable params: 17
Non-trainable params: 0
```

(b) Model Summary

Figure 7.4: Autoencoder

of normal and abnormal data. The model was trained by minimizing the reconstruction error on normal data. The best prediction result occurs when both validation loss and training loss are low. Validation loss should be similar to but slightly higher than training loss because of the presence of unseen testing data. If this condition is not met, then the model is experiencing either *underfitting* or *overfitting*.

Underfitting occurs when both training loss and validation loss are increasing. In this case, the model fails to learn the important features of the data and thus performs poorly on both the training and testing data sets.

Overfitting occurs when the training loss is reducing but validation loss is increasing. Figure 7.5 shows an overfitting. Here, the model learns the training data too well but does not perform well

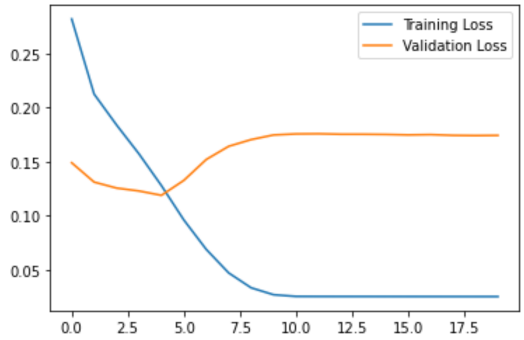


Figure 7.5: Overfitting

on the subsequent testing data. In this work, we experienced overfitting after training autoencoder model, and hence applied the following techniques to solve overfitting:

1) We increase the size of the training dataset from 8000 to 10000 samples. This helps the model learn features better, and perform better on previously unseen test data.

2)We Add a regularization term *kernel_regularizer = regularizers.l2(0.001)* to the model layer. This decreased the values of model weights, also reducing overfitting. L2 regularization is known as *weight decay*, which forces the weights to decay toward almost zero.

3)Early stopping : When we saw the performance on the validation set is getting worse, we stopped the training on the model at epoch=50 with batch_size=125.

```
Epoch 50/50
83/83 [==============================] - 0s 2ms/step - loss: 0.0177 - val_loss: 0.0466
```

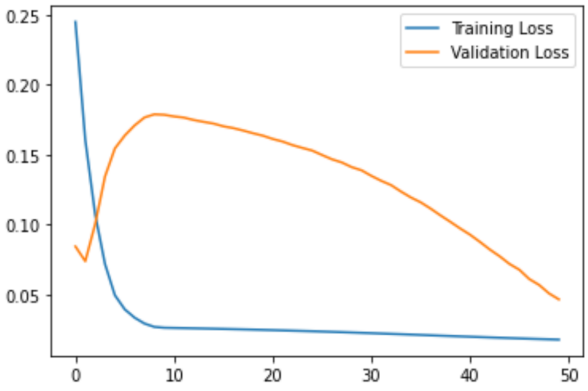
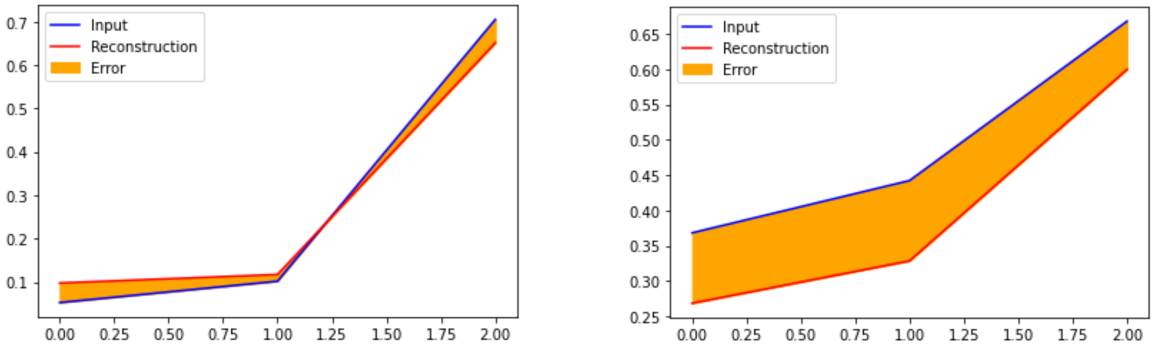


Figure 7.6: Autoencoder Loss Function

Figure7.6 shows the result of loss function after applying the aforementioned fixes for overfit-

ting. When training is completed, the final epoch shows the training loss and validation loss. Our validation accuracy was 0.99, which seems superficially excellent. However, we can not rely on validation accuracy to evaluate our model because the model's hyperparameters were tuned on the testing dataset. For better understanding of model's final performance, we evaluated it with the *model.evalaute* function.

As we expected, the Figure 7.7 shows the reconstruction error is small for a normal data and large for an unseen abnormal data. 7.7a plot represents a normal training example from accelerometer in blue and the reconstruction (after it is encoded and decoded by the autoencoder) in red. The gap between these values is filled with orange and represents the reconstruction error. The plot in Figure 7.7b shows this for abnormal accelerometer data.



(a) Reconstruction Error for Normal Input

(b) Reconstruction Error for Abnormal Input

Figure 7.7: Reconstruction Error for one sample

To detect anomalies, we chose a threshold based on reconstruction loss. Any data above that threshold would identify as anomalous data. We calculated the mean average error for normal data from training set as 0.038, then classified an accelerometer data as anomalous if its reconstruction error is grater than one standard deviation from mean of normal training set. Later, We adjusted that threshold to half standard deviation from normal training set (threshold = 0.046) for getting better accuracy, recall and precision. These terms were explained in chapter 8. Figure 7.8 represents reconstruction error on normal data with blue color and for anomalous data with red color.

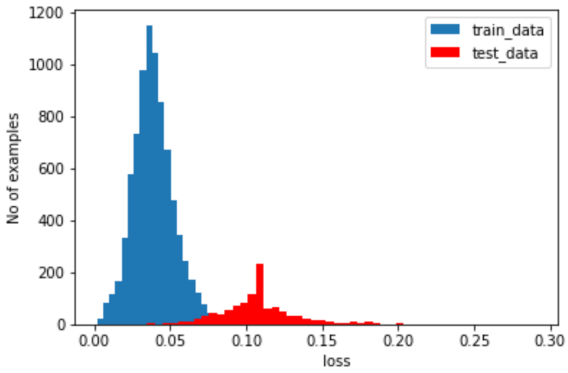


Figure 7.8: Autoencoder Reconstruction Error

7.3 Variational Autoencoder(VAE)

VAE is a modern neural network similar to the autoencoder. VAE is a *probabilistic autoencoder* whose output is partly determined by chance. VAE is a *generative autoencoder* which can generate new instances of data that is similar to the training data. Figure 7.9 shows a VAE structure that has basic structure as autoencoder with encoder and decoder. However, it is not directly producing an output from a given input.

The VAE encoder produces two vectors of size n: a mean μ and standard deviation σ. The latent samples randomly from a *Gaussian distribution* with mean μ and standard deviation σ, and then the decoder produces the output. If we were to represent the input values on a plot, the mean vector controls the ”center” of the plotted encoded values, and the standard deviation controls the ”area” of the plot, or how far apart these values spread. Thus the encoding is randomly generated from the statistical distribution of the input data, and the decoder learns that all nearby points around a single point refer to the same class. Since the decoder is exposed to a range of variations of the encoding of the same input, the decoder can decode not only a single point but also nearby points.

The Encoder defines the approximate posterior distribution p(z/x). Given a set of input data, the output will be a set of parameters for specifying the conditional distribution of the latent representation z.

The decoder defines the conditional distribution of the observation q(x/z). It takes a latent sample z as input and generates an output which is the parameters to a conditional distribution of

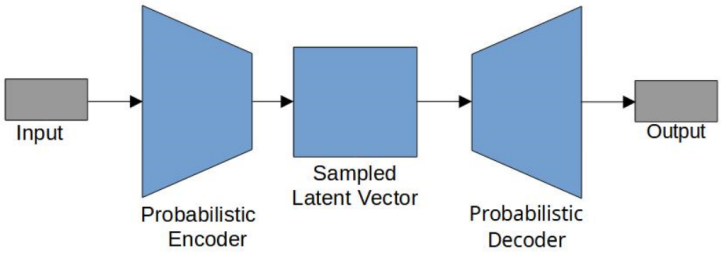


Figure 7.9: Variational Autoencoder

the input. During training, we sample from the latent distribution to generate a sample z for the decoder. One problem that occurred is that back propagation cannot flow through a random node in bottleneck [9]. To solve this problem, we approximate z using the decoder parameters and another parameter ε as expressed in equation 7.6.

$$z = \mu \ + \sigma \odot \varepsilon \tag{7.6}$$

ε can be a random noise to maintain stochasticity of z. We can generate z from a standard normal distribution. Now, The latent variable z would enable the model to back propagate gradient in the encoder through μ and σ while maintaining stochasticity through ε.

The reconstruction loss for VAE is a component of two items. The first term, like autoencoder, captures the difference between input and output, and can be used to determine how well the network generates output data similar to the given input data. The second term is a regularization term that indicates how the probability distribution is computed and how it regularizes the network. Equation 7.7 shows KL (regularization term).

$$KL = \frac{-1}{2} \sum_{j=0}^{\text{k-1}} (\sigma_j + \mu_j^2 - 1 - \log \sigma_j \ ) \tag{7.7}$$

7.3.1 VAE Implementation

The encoder has three neurons and takes normal accelerometer data (x,y,z) as input. The latent dimension is two, therefore both mean and log variance vectors have two neurons. The decoder has

three neurons as output. The model has only one hidden layer as is shown in the VAE structure as summarized in figure 7.12b. All layers are fully connected and our model has 35 parameters which includes all weights and biases. The model was trained and tuned for obtaining the minimum reconstruction error. Figure 7.10 shows the resulting loss function on the training and testing data sets.

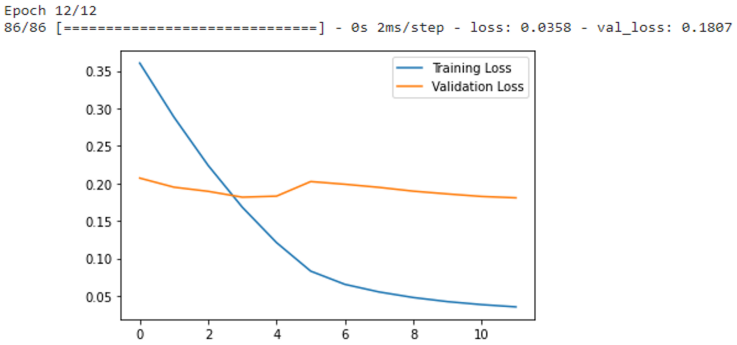


Figure 7.10: Variational Autoencoder Loss Function

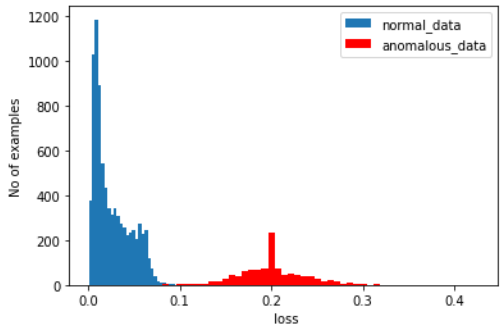
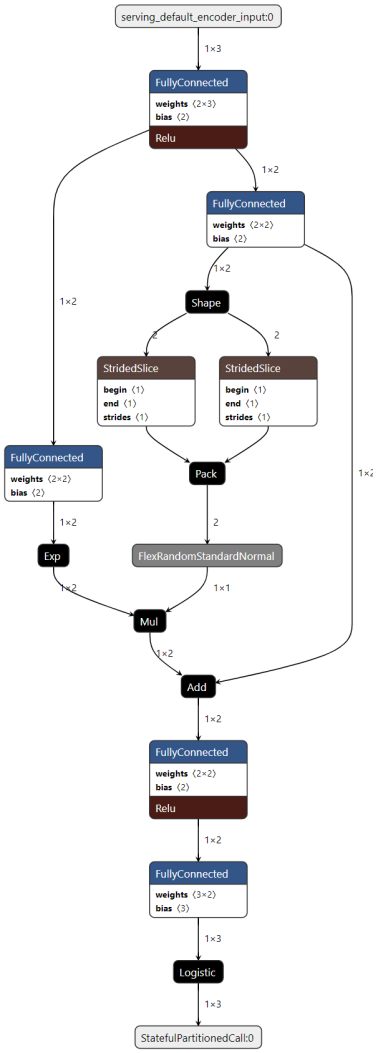


Figure 7.11: VAE Loss Function for Normal vs Anomalous data

For anomaly detection with VAE model, we selected a threshold based on the distribution of normal data from Figure 7.11 and then adjusted the threshold to obtain better accuracy, recall and precision. We calculated the mean average error for normal data from the training set as 0.025, then classified an accelerometer data as anomaly if its reconstruction error is greater than threshold 0.183.



(a) VAE Model Structure

Layer (type)	Output Shape	Param #	Connected to
encoder_input (InputLayer)	[(None, 3)]	0	
dense_5 (Dense)	(None, 2)	8	encoder_input[0][0]
z_mean (Dense)	(None, 2)	6	dense_5[0][0]
z_log_var (Dense)	(None, 2)	6	dense_5[0][0]
sampling_1 (Sampling)	(None, 2)	0	z_mean[0][0] z_log_var[0][0]
decoder (Functional)	(None, 3)	15	sampling_1[0][0]
tf.math.square_1 (TFOpLambda)	(None, 2)	0	z_mean[0][0]
tf.math.subtract_2 (TFOpLambda)	(None, 2)	0	z_log_var[0][0] tf.math.square_1[0][0]
tf.math.exp_1 (TFOpLambda)	(None, 2)	0	z_log_var[0][0]
tf.math.subtract_3 (TFOpLambda)	(None, 2)	0	tf.math.subtract_2[0][0] tf.math.exp_1[0][0]
tf.__operators__.add_1 (TFOpLam	(None, 2)	0	tf.math.subtract_3[0][0]
tf.math.reduce_mean_1 (TFOpLamb	()	0	tf.__operators__.add_1[0][0]
tf.math.multiply_1 (TFOpLambda)	()	0	tf.math.reduce_mean_1[0][0]
add_loss_1 (AddLoss)	()	0	tf.math.multiply_1[0][0]

Total params: 35
Trainable params: 35
Non-trainable params: 0

(b) Model Summary

Figure 7.12: Variational Autoencoder

Chapter 8

Experimental Result

8.1 Accuracy

Accuracy can be described as the total number of correct classifications divided by the total number of classifications. There are two possible predicted classes of data: normal or anomaly. We define some basic terms as follows:

- **True Positives(TP):** The cases in which the model correctly predicts that the input is an anomaly.
- **True Negatives(TN):** The cases in which the model correctly predicts that the input is normal (non-anomaly).
- **False Positives(FP):** The cases in which the model predicts anomaly, but the input is normal.
- **False Negatives(FN)** The cases in which the model predicts normal, but the input is an anomaly.

For evaluation of our anomaly detection model, prediction accuracy is not enough, because a model that was trained only with normal data and has high accuracy may not perform well when classifying anomalous data. Such a model can not be considered robust. It may misclassify normal data as anomalous (false positive(FP)) or anomalous data as normal (false negative(FN)). So, *precision* and *recall* provide better metrics than accuracy for anomaly detection applications.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{8.1}$$

Precision is the ratio between the true positives and all the positives. In this study, precision is the ratio of correctly-predicted anomalies to all predicted anomalies (both correct and incorrect).

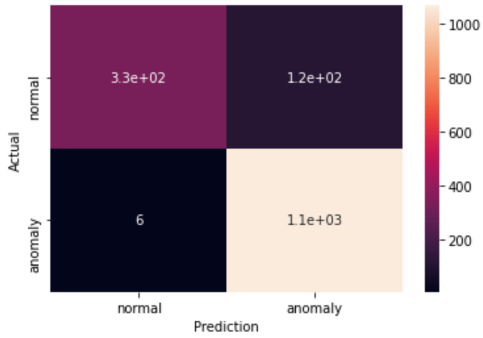
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{8.2}$$

Recall is the measure of how well the model correctly classifies anomalies. In other words, recall measures the fraction of anomalous data points which were identified correctly as anomalies.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{8.3}$$

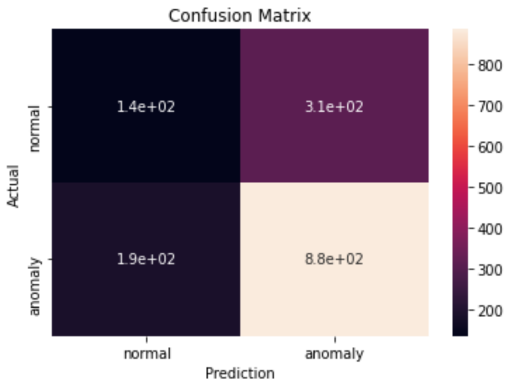
In this study, our objective was to choose a suitable threshold, then measure and improve accuracy as well as precision and recall. For the autoencoder model, as we mentioned in chapter 7.2.1, the threshold was chosen as 0.046 and the result is: Accuracy = 0.92, Precision = 0.90, Recall = 0.99. For the variational autoencoder model, as we mentioned in chapter 7.3.1, threshold was chosen as 0.183 and the result is: Accuracy = 0.66, Precision = 0.74, Recall = 0.80.

Figures 8.1 and 8.2 represent confusion matrix for both models. A confusion matrix is a table that describes the performance of a classifier on a set of test data. As figure 8.1 shows the autoencoder classifier made a total of 1519 predictions. It predicted 1182 items as anomaly and 337 items as normal. In reality, 1071 items are anomaly and 448 items are normal. Figure 8.2 shows the VAE predicted 1197 items as anomaly and 322 items as normal but actually 1071 items are anomaly and 448 items are normal.



N=1519	Predicted: Normal	Predicted: Anomaly	
Actual: Normal	TN = 331	FP = 117	448
Actual: Anomaly	FN = 6	TP = 1065	1071
	337	1182	

Figure 8.1: Autoencoder Confusion Matrix



N=1519	Predicted: Normal	Predicted: Anomaly	
Actual: Normal	TN = 136	FP = 312	448
Actual: Anomaly	FN = 186	TP = 885	1071
	322	1197	

Figure 8.2: Variational Autoencoder Confusion Matrix

8.2 Battery Life

To find out the battery life of a given model, we ran an Arduino Nano with two 3.7 VDC, 400 mAh lithium batteries in series with voltage dropping diodes to 5 VDC. We watched it until the voltage arrived at Vdead= 3.64 VDC and the Arduino shut down. The autoencoder model lived for 20 hours, which is very close to the theoretical hours computed from formula. We measured the current for autoencoder model as 16 mA and the storage capacity for our lithium battery was reported as 400 mAh. So, theoretically the lifetime of the model on this platform could be as much as (400/16)* 0.9 = 22.5 hours.

The battery life obtained for the autoencoder is within 89% of the maximum theoretical lifetime. This results from the fact that the lithium battery had been discharged and recharged previously, slightly limiting its change capacity. We did not measure battery life for the VAE model since the TensorFlow Lite library does not currently support all VAE operators.

Chapter 9

Conclusion and Future Work

Comparing autoencoder and VAE for anomaly detection on the washing machine shows that autoencoder has higher acuracy than VAE. The Arduino device detects and indicates unbalanced washing machine loads with 92% accuracy, 90% precision, and 99% recall using the autoencoder model. We experienced 66% accuracy, 74% precision, and 80% recall from VAE model. The battery life for the TinyML anomaly detector with autoencoder model is 20 hours with 5 VDC lithium batteries.

Working with a time series of data to find unbalanced load can be a future work for this project. It needs to collect time-series data for a full cycle balanced load. A combination of convolution and fully connected layers is very useful in classifying time-series sensor data like the measurements from the accelerometer.

To ensure sufficient long-life battery operation, the sensor will need to enter sleep mode for a period of time. Choosing the sleep period depends on the type of machine that you are protecting. In the case of our fast spin cycle anomaly, we need quick notification to prevent damage and cannot wait for the sensor to wake and sample. The sensor should sleep for a long period when the machine is non operational, and wake occasionally to see if the machine is running in normal modes. The sensor should stay alert (not sleep mode) and when the sensor detects that the machine is running it should be ready to detect the anomaly. The sensor can go back to the sleep mode after detecting the machine is not running. Also, the Arduino can send a message via WiFi internet of things to alert someone upon detection of the anomaly.

References

[1] https://colab.research.google.com/notebooks/intro.ipynb.

[2] https://store.arduino.cc/usa/nano-33-ble.

[3] https://tinyml.org/.

[4] Martín Abadi and Ashish Agarwal. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[5] P. Boniol, M. Linardi, F. Roncallo, and T. Palpanas. Sad: An unsupervised system for subsequence anomaly detection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1778–1781, 2020.

[6] C. H. Chang. Managing credit card fraud risk by autoencoders : (icpai2020). In *2020 International Conference on Pervasive Artificial Intelligence (ICPAI)*, pages 118–122, 2020.

[7] Miguel de Prado, Manuele Rusci, Romain Donze, Alessandro Capotondi, Serge Monnerat, Luca Benini, and Nuria Pazos. Robustifying the deployment of tinyml models for autonomous mini-vehicles. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[8] Q. Duan, X. Wei, J. Fan, L. Yu, and Y. Hu. Cnn-based intrusion classification for ieee 802.11 wireless networks. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 830–833, 2020.

[9] Y. Guo, T. Ji, Q. Wang, L. Yu, G. Min, and P. Li. Unsupervised anomaly detection in iot systems for smart cities. *IEEE Transactions on Network Science and Engineering*, 7(4):2231–2242, 2020.

[10] F. Han, S. Liu, S. Liu, J. Zou, Y. Ai, and C. Xu. Defect detection: Defect classification and localization for additive manufacturing using deep learning method. In *2020 21st International Conference on Electronic Packaging Technology (ICEPT)*, pages 1–4, 2020.

[11] I. Krissaane, K. Hampton, J. Alshenaifi, and R. Wilkinson. Anomaly detection semi-supervised framework for sepsis treatment. In *2019 Computing in Cardiology (CinC)*, pages Page 1–Page 4, 2019.

[12] Haidong Li, Jiongcheng Li, Xiaoming Guan, Binghao Liang, Yuting Lai, and Xinglong Luo. Research on overfitting of deep learning. In *2019 15th International Conference on Computational Intelligence and Security (CIS)*, pages 78–81, 2019.

[13] D. Liao, S. Huang, Y. Tan, and G. Bai. Network intrusion detection method based on gan model. In *2020 International Conference on Computer Communication and Network Security (CCNS)*, pages 153–156, 2020.

[14] T. Nakazawa and D. V. Kulkarni. Anomaly detection and segmentation for wafer defect patterns using deep convolutional encoder–decoder neural network architectures in semiconductor manufacturing. *IEEE Transactions on Semiconductor Manufacturing*, 32(2):250–256, 2019.

[15] Daniel Situnayake Pete Warden. *TinyML*. O'Reilly Media, Inc., 2019.

[16] R. Sanchez-Iborra and A. F. Skarmeta. Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 20(3):4–18, 2020.

[17] R. Singh, R. Garg, N. S. Patel, and M. W. Braun. Generative adversarial networks for synthetic defect generation in assembly and test manufacturing. In *2020 31st Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*, pages 1–5, 2020.

[18] Radoslava Švihrová and Christian Lettner. A semi-supervised approach for network intrusion detection. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[19] O. Vynokurova, D. Peleshko, O. Bondarenko, V. Ilyasov, V. Serzhantov, and M. Peleshko. Hybrid machine learning system for solving fraud detection tasks. In *2020 IEEE Third International Conference on Data Stream Mining Processing (DSMP)*, pages 1–5, 2020.

[20] Xiaying Wang, Michele Magno, Lukas Cavigelli, and Luca Benini. Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things. *IEEE Internet of Things Journal*, 7(5):4403–4417, 2020.

[21] S. B. Wankhede. Anomaly detection using machine learning techniques. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, pages 1–3, 2019.