Machine Learning for Time Series Anomaly Detection

by

Ihssan Tinawi

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 10, 2019

Certified by .....
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Department Committee on Graduate Theses

Machine Learning for Time Series Anomaly Detection

by

Ihssan Tinawi

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

In this thesis, I explored machine learning and other statistical techniques for anomaly detection on time series data obtained from Internet-of-Things sensors. The data, obtained from satellite telemetry signals, were used to train models to forecast a signal based on its historic patterns. When the prediction passed a dynamic error threshold, then that point was flagged as anomalous. I used multiple models such as Long Short-Term Memory (LSTM), autoregression, Multi-Layer Perceptron, and Encoder-Decoder LSTM.

I used the "Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding" paper as a basis for my analysis, and was able to beat their performance on anomaly detection by obtaining an $F_{0.5}$ score of 76%, an improvement over their 69% score.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

Bismillah. In the name of God, the Most Gracious, the Most Merciful.

I would like to thank SES for their support in our project.

I would also like to thank my supervisor Kalyan for all I have learned from him.

Thanks to my friends and especially my roommate Rami for making this year a memorable one.

Finally, special thanks to my parents without whom none of this would be possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Machine Learning for Anomaly Detection

Machine learning is a field within statistics in which programs use historic data to make predictions about future data points or unknown labels. A model is said to learn when it is able to produce estimates of a data point based on whatever structure it has inferred from previous data. Machine learning can be used in the context of time series forecasting to predict the value at the next time step; in classification problems where given an input, the model's goal is to predict the label; and in clustering to determine which points resemble each other in high-dimensional space. These are just three examples of common uses of machine learning.

The goal of machine learning is to map high-dimensional data and to try to separate data points based on their labels. Models are trying to minimize distances between similar points and correctly classify them. At the core of machine learning is data and loss functions. Models are trying to learn features from the data with the help of loss functions that generalize our models to previously unseen data.

Time series anomaly detection is a field that has historically utilized several statistical methods in order to try to guess anomalies in sequential data. time series is used to refer to data that has values associated with timesteps. Examples of time series data are stock prices or temperature readings from a thermometer. In both cases data is sampled at a certain frequency, and the values at those instances are

recorded. Often, time series data exhibits patterns like periodicity, cyclic growth, exponential decay, to name a few characteristics. The data can also sometimes undergo shocks or changes that go against its predictable nature. For example, a company's stock price can tank if news leaks that it has sustained a big loss. Similarly, a sudden increase in the temperature of a room in a factory can signal that there is a fire. In any of these situations, it would be beneficial to have a system that can detect and flag anomalies, giving administrators the ability to minimize losses. We will utilize a publicly available spacecraft data set to analyze the validity of using machine learning systems for anomaly detection tasks in spacecraft.

There are several ways to carry out anomaly detection, using statistical methods such as clustering, wavelet transforms, autoregressive models, and other machine learning and digital signal processing techniques. Within machine learning, the time series anomaly detection problem can be classified into two categories: supervised and unsupervised. The first category occurs when historic anomalies are known and models are fitted to identify similar anomalies, and this is known as "supervised learning" because data points are labeled with their true nature. In other settings, the labels are not known and the model tries to predict whether a point is anomalous based on how close it is to previously seen data, and these instances are referred to as "unsupervised learning" tasks.

Unsupervised learning settings are much more common, because machines don't know a priori whether they are experiencing glitches, so the data doesn't contain explicit information about when and where an anomaly is taking place. In these settings, it is expensive to hire experts to label whether data points are anomalous, and it is even possible for the experts to misidentify or entirely miss abnormalities. A good model is tuned so that is not strict leading to too few anomalies being noticed and not over-sensitive such that a lot of normal points are flagged as anomalies. This enables companies to adjust their services to situations of high demand or to replace failing components of vehicles before they endanger passengers. In spacecraft, where the machine is already launched and cannot be maintained, knowing ahead of time about failure of components enables the operators to migrate their services to other

crafts or to rely on other components of the same machine.

1.2 Motivations for Anomaly Detection

If we were to plot the amount of data being generated within organizations, it would be an exponential curve as in Figure 1-1 below. To make sense of all this data, companies need to hire analysts to match the large rate of growth of the data. However, organizations can at best only afford to hire in a linear manner due to financial constraints. Therefore, there is a growing gap between the amount of data being generated and the number of personnel available to go through it. This causes an intelligence gap, and it represents a large problem that companies in the digital age are facing. Namely, it is too expensive to hire humans to review data at the rate it is being generated. This is where computers can contribute. Machine intelligence programs can be built in order to monitor information from critical systems and flag anomalies as they occur. Human experts can then view these warnings, and decide to deal with them on a case-by-case basis. This would decrease the amount of work that the human experts have to do, thereby decreasing the burden on companies to hire more and more experts.

Building machine learning systems that can process information and identify anomalies is more cost effective than creating human-based teams. It can also detect changes in the signal that are too subtle for humans to identify. Statistical methods can be used to determine context anomalies, which are changes that shift across seasonal trends. For instance, the average temperature is much lower in the winter than in the summer. So, if you encounter a temperature of 80 degrees Fahrenheit in the winter, it's an anomaly, whereas in the summer it would be a normal temperature. In our work, we focus on creating an anomaly detection system for satellites and spacecraft. The input to the system is telemetry data coming from the satellite, and the output is anomalies. The system flags anomalies and alerts technicians, reducing the overall amount of data they need to review. The best system has a small number of false positives and negatives. That is, the system is not sensitively flagging any

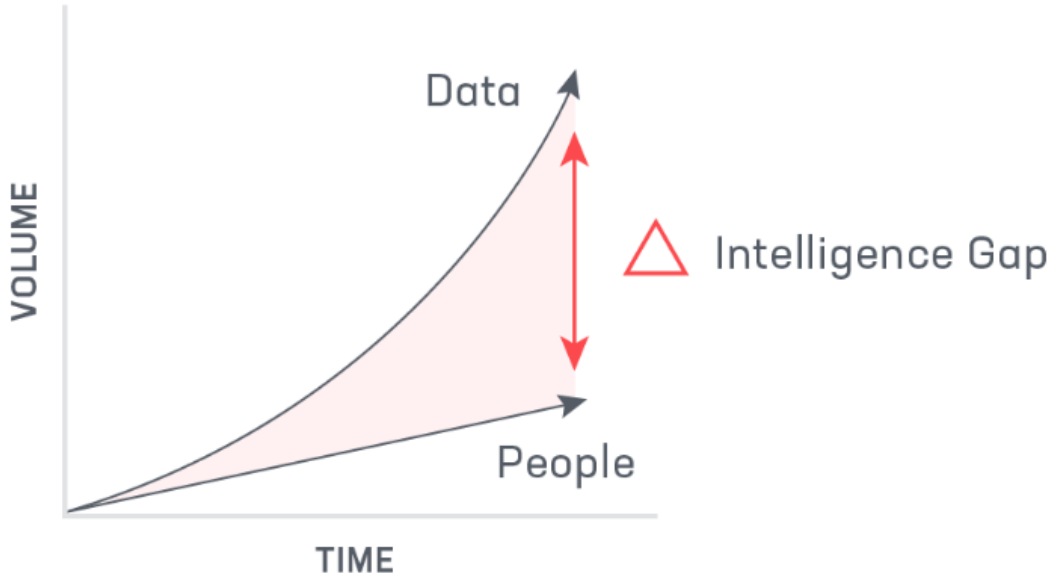

Figure 1-1: Divergence of growths between data and human resources within organizations.

and all variations in the data (increasing the amount of work technicians have to do) and at the same time the system's threshold is not too strict that it cannot identify anomalies.

1.3 Contributions & Key Findings

In this work, we explore several time series forecasting methods and apply them in the context of spacecraft data to build an anomaly detection system. We study the performance of multilayer perceptrons (MLP), Long Short-Term Memory (LSTM), LSTM Encoder-Decoder, and linear autoregression models to contrast which architectures perform the best for time series data forecasting.

We work with a public dataset provided by researchers at NASA. The data comes from two of their spacecrafts, and is available in a format that is standard to time series data. As such, our system can be adapted to work with any dataset that shares the format of the NASA data. More information about the data and its format can be found in chapter 3.

Anomaly detection is a technique used in the field of statistics to determine outliers from signals. In our work, we apply state of the art machine learning and traditional statistical techniques to spacecraft telemetry data to infer anomalies from data in an unsupervised manner. We adapt the dynamic error thresholding technique found in NASA's LSTM and Nonparametric Thresholding Paper [2] in order to detect anomalies. Below, we summarize our main contributions and findings.

- Built an end-to-end machine learning pipeline that can detect anomalies and is modular for trying out different models.
- Tested out the pipeline with 4 different architectures (MLP, LSTM, LSTM AutoEncoder, and Linear Autoregression).
- We found that increasing the length of the input sequence to the models yielded improved results.
- By decreasing the number of hidden units, we were able to outperform NASA's model even without using their pruning technique, obtaining an $F_{0.5}$ score of 0.76 over the reference paper's 0.69 score.
- The LSTM Autoencoder got a high accuracy prediction score of 0.83, close to the NASA model's precision score of 0.88.

Chapter two discusses related work in the literature, and describes the difference between our approach and other papers in the field.

Chapter three details the data that we are using.

Chapter four provides an overview of the system that was implemented. It describes the architecture of the anomaly detection system, and breaks it down into pre-processing, modeling, and post-processing.

Chapter five is a comprehensive summary of our results and contains an analysis of the different models we tested out. Possible future expansions to the project are also discussed.

Chapter 2

Relevant Work

In recent years, a lot of deep learning models are being employed by researchers for time series forecasting. State-of-the-art models now apply methods such as Long Short Term Memory (LSTM), stacked LSTMs, and autoencoders, among other architectures, to forecast time series data and then detect anomalies from those predictions, once the real measurements are observed. For example in a paper titled "Detecting Spacecraft Anomalies Using LSTMs and Nonparametric Dynamic Thresholding" [2], researchers from NASA apply a stacked LSTM model to telemetry data obtained from two of their satellites. The signal is passed through the model and it generates a forecast for the next timestamp. The predicted value is then compared to the realized value to compute an error. After applying smoothing techniques, the error is passed through a dynamic threshold that determines whether the observed value is an anomaly, based on the previous errors of the model. We will be using several such papers and drawing upon the literature to implement different techniques to generate several pipelines that we can train our data on and use to generate anomalies.

The NASA paper introduced a novel technique called dynamic error thresholding. This method is different from previously used error cutoffs such as fixed error thresholds like x-sigma or distance based approaches such as clustering [4]. Non-stationary data is characterized by having a mean and variance that shifts with time. For instance, the data can have growth trends or exhibit cyclic behavior. This non-stationarity makes it difficult for fixed error thresholding techniques to work in prac-

tice. Thus, dynamic error techniques, which are not constrained by non-stationary data, have the potential of performing better. The error thresholding technique proposed by the NASA team is nonparametric, dynamic, and unsupervised. We use this error thresholding technique and compare its performance to x-sigma thresholding to see which provides a prediction rate for anomalies. More information about the theory underpinning dynamic error thresholding can be found in chapter 4 (System Architecture).

2.1 Anomaly Detection for Spacecraft

In the field of aerospace engineering, there has been extensive use of anomaly detection. As previously mentioned, these systems are difficult to create, because on the one hand you do not want to have a threshold that is too sensitive, resulting in too many false flags. Such a sensitive threshold would be expensive to maintain, because technicians and observers will have to verify a large number of anomalies. And on the other hand, you do not want a threshold that is too strict, meaning that a lot of anomalies will go by undetected.

Regardless, a lot of anomaly detection techniques exist for spacecraft. For example, on the International Space System, there is the Inductive Monitoring System (IMS), built by NASA to deal with anomalous behavior. The IMS uses a clustering-based approach to determine the health of signals, as compared to nominal performance. Clustering techniques are distance-based, meaning that clusters are formed based on how far a data point is from others. Clustering methods do not require the data to be labeled, making it a good fit for our unsupervised task. However, the problem with clustering algorithms is that you need to know a priori the number of clusters, and they are very sensitive to outlier data points. For instance, one outlier data point will warp the shape of a cluster so that it spans incorrect regions.

To outline how a clustering algorithm would carry out anomaly detection, we describe a setup that utilizes k-means clustering. K-means clustering is not to be confused with k-nearest neighbors, which is used in classification and regression. K-

means clustering serves to partition n observations into k clusters based, where each new data point is assigned to the cluster with the closest mean. Clustering can be solved with an approximation algorithm that has two steps: assignment and update. To conduct time series anomaly detection, the signal is passed in as individual data points, and the number of clusters can be set to 2 (one anomalous and one normal). Alternatively, we can experiment with 3 clusters (anomalous, normal, and outliers) to see how that improves the performance of the system.

The problem with a clustering approach is that typically algorithms assume that the data is equally distributed across the clusters, which is not a valid assumption to make in the case of anomalies. For instance, a signal might have little to no anomalies in a given sample. Another limitation of clustering-based approaches is that the data might go through multiple phases. For instance, if we are looking at a thermometer's signal, and the thermometer is on Earth, then data in the summer will be higher on average than data in the winter. This will lead the clustering algorithm to assign the temperature readings into one cluster in the summer and another in the winter. Thus, contextual anomalies are lost, where an unexpectedly high temperature in the winter is just assigned to the summer cluster.

Beyond clustering-based approaches for anomaly detection, regression error techniques are the most common in time series forecasting and anomaly detection, and this is the approach that was taken by several papers on which we base our work. The NASA paper, mentioned above, uses LSTMs as the regressor in the model. In another paper, researchers from TCS Research in India, use LSTM-based encoder-decoder model for multi-sensor anomaly detection in spacecraft [3]. The system we build carries out modeling of the signal using different neural networks, adapted from such papers, and then applies NASA's dynamic error thresholding technique to determine whether the residual error is too large, indicating that the input wasn't in line with what was expected. The process behind this approach is outlined further in chapter 4 (System Architecture).

2.2 Using Neural Networks for Anomaly Detection

Recently, there has been large interest from research groups to apply neural architectures in the field of time series forecasting. With the advent of faster computing chips and the abundance of data, a new Golden Era of machine learning has started, making researchers interested in applying deep learning techniques to any decades-old statistical field. Among these contributions are several papers that we use as basis for the time series forecasting in our system. These methods are in contrast to the "traditional" machine learning tools such as nearest neighbor or k-means clustering that we explored earlier. They are also different from the statistical techniques such as low-pass filters and other digital signal processing methods.

In deep learning, architectures such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks were found to perform better compared to feed forward neural network architectures. Due to their ability to encode temporal properties, RNNs and LSTMs perform well in time series forecasting, speech recognition, and natural language processing tasks, since they both rely on modeling sequences. LSTMs have the ability to learn the relationship between past data and current values. The capacity of LSTM to model this dependency has led to a large academic interest in using these units in machine learning for time series forecasting. The NASA and TCS Research papers that we implement are some examples, and we compare and contrast these with different architectures in our work.

Chapter 3

Data

The data we are using is in time series format. This means that every value is a reading from the sensors of the spacecraft that occurred at a specific moment in time. The data was obtained from a public dataset used in a study published by research scientists at NASA. The telemetry data comes from two spacecrafts: the Soil Moisture Active Passive satellite (SMAP) and the Curiosity Rover on Mars (MSL). The data was anonymized with respect to time, but it retained its sequential nature. The anonymization does not affect any time series forecasting of the data, as the timestamp itself doesn't matter, and we only care about the values in the sequences.

Furthermore, the telemetry signals are divided into several channels, where the channel category and the signal number describe the signals and give hints as to which of them are related. For example, signals "P-1" and "P-2" are related, as they come from components representing "Power" signals of the craft. There are 82 signals available in the NASA dataset, with 55 coming from the SMAP satellite and 27 from the MSL Mars Rover. The signals were individually plotted and evaluated for qualitative features, such as being discrete or continuous and exhibiting any periodic trends. 54 of the 82 signals were found to be continuous by inspection, and the remaining signals were discrete. In order to determine characteristics for each signal, the Python Plotly toolkit was used to plot each signal. After that, we visually inspected each graph, zooming in on parts of the signal to determine whether the data was indeed continuous or discrete at the lowest level.

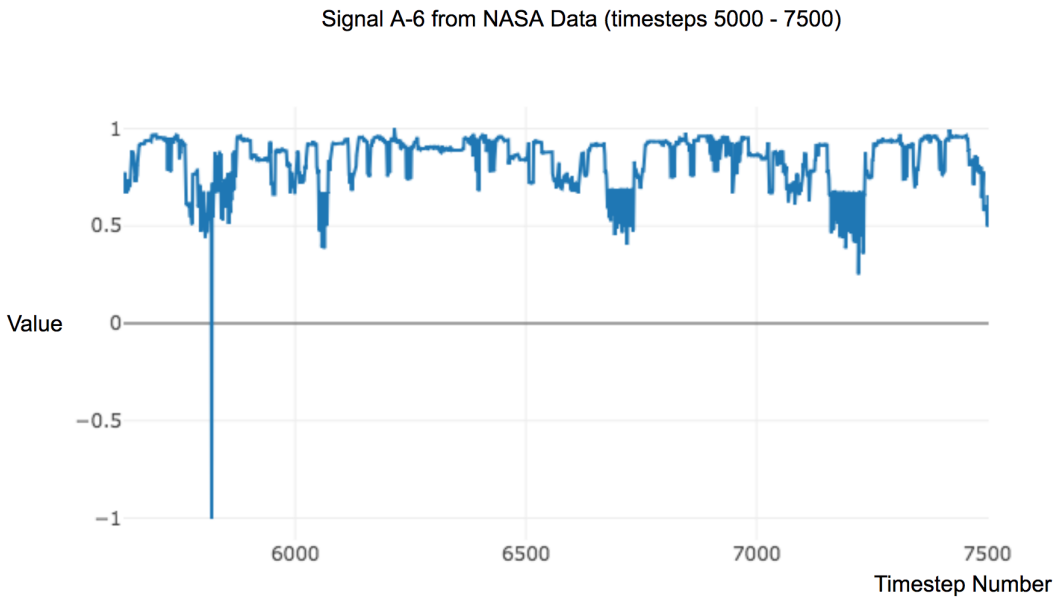

Figure 3-1: NASA Signal A-6 plotted to show its continuos nature.

In Figure 3-1, we see the example of Signal A-6 from the NASA data. As can be seen in the figure, the graph shows the data from timesteps 5000 to 7500. As mentioned above, the data has been anonymized with respect to the actual timestamp, so we deal with timestamps as the timestep sequence number. Signal A-6 is continuous, and oscillates between values of $(-0.356, 0.356)$. Additionally, it seems there is an an outlier value at around timestep 5500. This qualitative analysis is useful as we try to justify later why some signals might have been better modeled using certain architectures. It also provides an additional layer of insight when we are trying to see why a certain signal was difficult to predict the errors for. The qualitative inspection was carried out for all signals.

For a given signal, let's say "M-1", the data has m rows and n columns. The m rows represent the commands sent to different modules, in binary format, and the last row is the telemetry value itself. The layout of the data can be found in Figure 3-1, adapted from the NASA paper. There are n rows corresponding to the number of timestamps or readings taken for that signal. Therefore, the signal at each timestep is an $m \times 1$ matrix. For our purposes, we do not use the commands as input features for our models, so we only create sequences from the telemetry value itself. So, if we

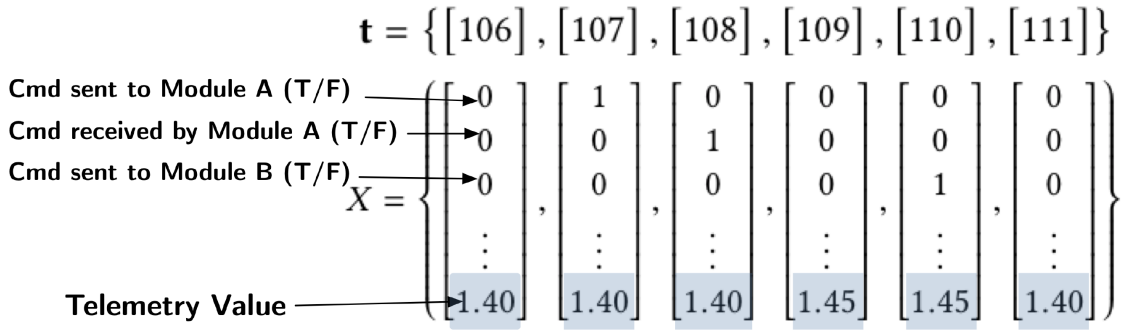

Figure 3-2: NASA Data Format.

decide to use a sequence of length 250, then the input to the model would be a of size 250×1, instead of $250 \times m$. The structure of input and output data to and from the models is explained in more detail in chapter 4 (System Architecture).

Upon looking at the data, one notices that all the telemetry values are between -1 and 1. This doesn't represent the real values that were captured from the devices. The researchers pre-processed the data to make its values scale to that range. This technique is an important step for time series forecasting analysis, as it confines the range of predictions the model has to learn. Therefore, our system assumes that data has already been scaled to be in the aforementioned range, and an algorithm to apply this to new and real-time data can be found in section 5.2 entitled "Further work." The original researchers of the paper scaled the public data, but we also did two changes to the data before carrying out the analysis.

First, we combined the "train" and "test" signals into one data file for each signal. This enables us to try out different train/test splits. The paper had limited train/test splits of the data, and in some cases there were more testing points compared to training data points. In time series anomaly detection, you are implicitly assuming that all training data does not have any anomalies. Therefore, the train/test split will have a large effect on the outcome of the experiment. We found that to be true in our experiments, where we encountered poorer performance when we used a train/test split different from the one used in the NASA paper. We discuss this further in our analysis (chapter 5). The second important change that was done in the data is that

Table 3.1: Data Summary

Number of Continuous Signals	53
Number of Discrete Signals	29
Total Number of Anomalies	106
Average Anomalies Per Signal	1.3

we are using only the telemetry value and not the series of commands that was input to the different modules. While these input commands can act as features (such as the mode of the satellite), the NASA paper ignores this part of the signal, and instead the focus is on re-creating telemetry values from previous ones and not by contextualizing it with other features of the craft. That is, at each point we want to predict the value of the telemetry signal, and we do not necessarily want to guess the type of command that was sent to different modules. As mentioned in the previous paragraph, this means that the input is now of the shape $sequence\_length \times 1$ rather than $sequence\_length \times m$, where m is the number of rows in the input matrix. In both our work and the NASA paper, we limit the dimension of the input vector $m = 1$.

Despite the data being unsupervised, NASA researchers hired experts to sift through the collected spacecraft logs and determine which of the signals were anomalous. This is part of a normal analysis that typically occurs after mission data has been collected. In a separate file called *labeled_anomalies* NASA provided a summary of the anomalous regions of each of the 82 signals, as identified by its engineers. The file is publicly available on the GitHub repository of the paper.[1] It's important to note, however, that there may have been anomalies that were overlooked during the annotation process due to human misjudgment. Nevertheless, the *labeled_anomalies* file is taken as the ground truth in our analysis. The labeled anomalies file has 82 rows and 5 columns for the anomaly sequences, the name of spacecraft, the Channel ID, the type of anomaly (contextual or point), and the number of telemetry values in the stream. In the field of anomaly detection, a point anomaly is defined as data that is

[1] https://github.com/khundman/telemanom

far too large or too small compared to the entire dataset. A contextual anomaly, however, is anomalous only in the context of the data around it. For example, on vacation spending $100+ on food might be normal whereas on normal days it is considered anomalous. In this work, we apply NASA's dynamic error thresholding, which takes into account the context of a data point, mitigating the incorrect classifications that may occur if errors are compared to a static threshold. A more thorough description of the technique can be found in the following chapter 4 (System Architecture).

Chapter 4

System Architecture

4.1 Overview

We are building an anomaly detection system, which is capable of swapping similar primitives to rapidly prototype different architectures and converge on the model that performs best for a given task. With that design consideration, we split our anomaly detection task into three modular components. In the first one, we carry out all pre-processing tasks, readying the data for a time series forecasting model. In the second module, we can use any linear, non-linear, or machine-learning regressor to generate a prediction for the next time step. In the last module, we compute the error residuals between the predicted values and the realized values. Based on the error thresholding technique used, anomalies are flagged and stored into a database.

In each of these sections, we talk about the algorithms that were implemented, detailing the inputs and outputs, as well as which parameters could be changed and tuned to achieve different results. In the results section, we delve deeper into the performance results on the anomaly detection, and highlight which models and error thresholding techniques achieved better performance in practice.

Chapter 4 is organized as follows. In section 4.2, we talk about the pre-processing phase and the two algorithms we use to shape the input to the models. In section 4.3, we talk about different architectures that we have used in our work. This section discusses the theory behind the models. Section 4.4 talks about the post-processing

and dynamic error thresholding technique that we use to detect anomalies. And finally, section 4.5 walks us through a sample of the pipeline using one of the NASA signals.

4.2 Data Pre-processing

The raw telemetry data could not be directly ingested into our pipeline, and it needs to first undergo some processing. We implement two methods that changes the data into a format that enables us to carry out time series forecasting. The first method is called *time series aggregation*, and it combines data across time intervals, thereby changing the frequency of the data. The other is called *rolling window sequences*, and it creates overlapping sequences that are used to predict in the time series forecasting models. We will now go over each of these methods, and describe their functionalities.

4.2.1 Time series Aggregation

Inputs

- X: time series data, with values and time stamps
- $interval\_length$: length of the interval, in seconds, that we want to aggregate based on

Outputs

- $values$: data aggregated by taking the average across intervals
- $index$: the corresponding time stamp for each new aggregated data point

The time series aggregation method takes in time series data and clusters data spanning a certain interval, using mathematical aggregation methods such as mean and median, to name a few. The method is useful because it reduces the size of data sets, by changing the frequency at which the data is sampled. Often times, in IoT sensor data, information is collected at a very high rate, but a lot of the information is

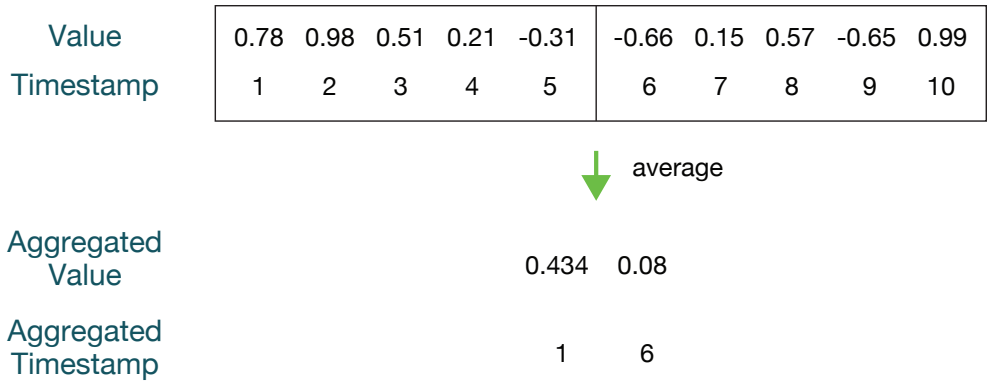

Figure 4-1: Mechanism of time series aggregation.

redundant, and can be summarized by taking averages across an interval of one hour for example. This also cleans up the data that is being used, so outlier readings are smoothed out across the interval. Though this is a limitation if we were looking for individual anomalous events, it does help to focus on long term trends. If users of the system believe that their data is sampled at the right frequency, then the interval can be set to the current sampling rate. In the case of the NASA data, each timestamp was equivalent to one anonymized time step, and thus the interval was set to 1.

4.2.2 Rolling Window Sequences

Inputs

- X: time series data, with values and time stamps
- $index$: the corresponding time stamps for values
- $window\_size$: the size of the sequence window that you want to generate
- $target\_size$: the number of steps ahead to predict

Outputs

- out_X: the sequence of $window\_size$ starting at each time stamp
- out_y: the corresponding next step(s) ahead to predict, based on $target\_size$
- X_index: time stamps associated with out_X values

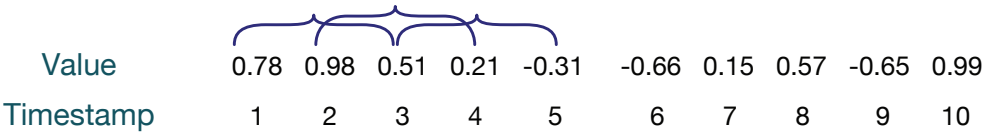

Figure 4-2: Mechanism of rolling window sequences. The sliding window corresponds to different overlapping sequences.

- Y_index: time stamps associated with out_y values

Rolling window sequences is a method that is commonly used to prepare time series data for building models for forecasting. Out_X acts as the input to the forecasting problem, giving the model context for the previous values. The model then uses the values in the sequence to predict the next $target\_size$ values. In this project, we use the sequence $window\_size$ to be 250 steps, as used in the NASA LSTM paper. In the results section, we discuss the effects of varying the length of this sequence on the performance of the models.

4.3 Models

We use a host of machine learning and statistical models in order to carry out the time series forecasting. The models use the data that was pre-processed using the methods described in section 4.2 above. The input to forecasting models is sequences of aggregated data that is scaled to be in the range [-1, 1]. There's a wide literature of work that discusses different methods of normalization. In many machine learning setups, the models perform better when they use normalization techniques [1].

As our problem is an unsupervised learning setup, then the training data used to fit the models is assumed to be *correct*, ie. without anomalies. This is because the model is primed so that all the data that it has seen in the training phase is regarded as normal. The intuition is that the model is accustomed to seeing the data in the training set. But then when the model goes live or is being subjected to testing data, then any data that exhibits characteristics different from the training data will not be successfully predicted by the model. As such, there will be a larger than usual prediction error for this any anomalous points. That's why we would like to train

models that are capable of first capturing periodic trends in the data and second performing poorly when bad data is ingested into the pipeline.

In our setup, we implemented around 10 models, and we have selected 4 models to carry out our analysis. The models are: multilayer perceptron, stacked LSTM model, LSTM encoder-decoder, and linear autoregressive model. They all rely on the same principle of using past sequential data to try and predict future data, with the exception of the LSTM enc-dec, which seeks to recreate the input itself after compressing the data with the encoder and then unzipping it with the decoder. Before going into the results of the models, we discuss the theory and fundamentals underpinning each of these models, and any implementation details that we added for necessity along the way.

4.3.1 Multilayer Perceptron (MLP)

Multilayer perceptron (MLP) is a type of feedforward artificial neural network architecture that comprises of three layers at the minimum: an input layer, hidden layer(s), and an output layer. The hidden layer units have a non-linear activation function, and the output layer in the context of a time series forecasting problem is a single unit that predicts the value at the next time step. MLP is considered to be the "vanilla" neural network, because it does not include any recurrent, convolutional, or gated layers that more recent architectures adopted. It acts as a linear perceptron, with the only difference being the `nonlinearity introduced` by adding activation functions such as `tanh`, `rectified linear` (RELU), and others.

The hyperbolic tangent, *tanh*, function maps all the real numbers to an output range of [-1, 1]. For the output node, this works well for our prediction task, because we normalize the range of our input values to [-1, 1]. This technique, called feature scaling, is important to be done in machine learning algorithms, because objective functions don't work properly without normalization. As classifiers compute the Euclidean distance between two data points, the distance is often skewed by features that have broad range of values. This normalization also helps with the faster convergence of training in stochastic gradient descent.

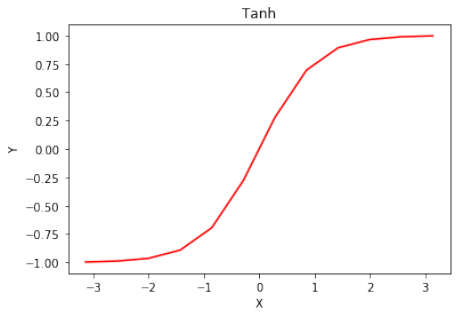

Figure 4-3: Tanh Activation Function.

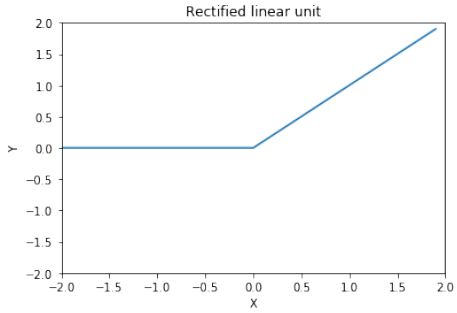

Figure 4-4: ReLU Activtion Function.

The RELU is another type of activation function that we utilize in the training of our deep neural networks, where the function definition is

$$f(x) = max(0, x)$$

RELUs apply a non-linear transformation to inputs, taking only the positive part of the input. RELUs are preferred over other activation functions like *sigmoid* and *tanh*, because they offer two benefits: sparsity and better learning. Better learning is achieved because there are lower chances of the vanishing gradient problem. This is because, when $x > 0$, the gradient has a constant value, as opposed to the gradient of tanh which becomes increasingly smaller the larger x gets. The constant gradient results in faster learning. The second advantage, sparsity, can be observed when $x \leq 0$ and the values default to 0. This transformation leads to more dense representation of matrices, speeding up the learning process for the network and decreasing its size. Tanh, on the other hand, will generate non-zero values, resulting in dense representations.

Time series forecasting problems are in essence a supervised learning task. While the data initially is not "labeled", a few easy transformations, described in Data Preprocessing (section 4.2 of System Architecture), can allow us to treat the data as if it's supervised. Primarily, the main trick is to add a lag factor to the data, and then you have the realized values of the y that you are trying to predict. Additionally, *rolling_window_sequences* creates the sequences that are used as input to the

model. In supervised learning settings, artificial neural networks are trained with a technique called gradient descent (GD). The aim of GD is to minimize an objective function, usually a heuristic cost function of how far the predictions were from the training value. In our models, we used the mean squared error as our accuracy measure.

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

where n is the size of the training data.

In gradient descent, in order to find the best value of a parameter we subtract the gradient of the cost function with respect to that parameter [5].

$$\theta_j = \theta_j - \alpha \times \frac{\partial}{\partial \theta_j} J(\theta)$$

where α is the learning rate.

However, gradient descent turns out to be slow in training, so we use a variant called Stochastic Gradient Descent (SGD), where instead of using the cost gradient of all examples, we calculate it only for one example at each iteration. In practice, SGD performs just as well but trains faster than regular gradient descent. The SGD algorithm proceeds as follows. First, we choose an initial vector of parameters w and a learning rate η. Next, we repeat the following until convergence (a minimum error is reached):

- Randomly shuffle examples in the training set.
- For $i = 1, 2, ..., n$ **do** $w := w - \eta \nabla Q_i(w)$

The architecture that we used for the multilayer perceptron is comprised of an input layer, 3 hidden layers each followed by dropout layers, and then one output layer. A dropout layer is added as a regularization technique. It reduces the overfitting of the model by preventing the coadaptation of the units based on the training data [6]. All of the hidden layers have RELU activation functions, and the output layer has a *tanh* activation. The models were trained for 35 epochs, which is the number of

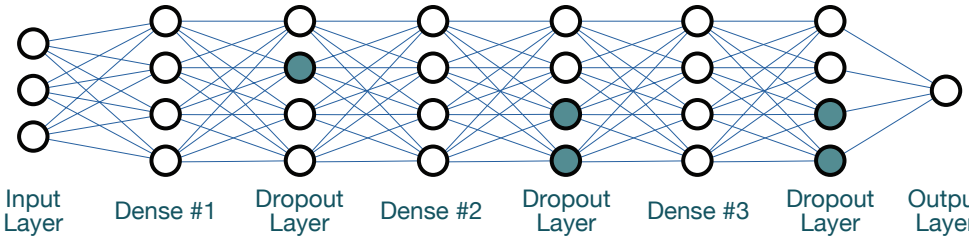

Figure 4-5: Multilayer Perceptron Architecture.

times the entire training dataset is passed through the neural network. The number of epochs was determined in practice, such that our model neither overfits nor underfits the data.

4.3.2 Long Short-Term Memory (LSTM)

Long short-term memory (LSTM) is a type of deep neural network architecture that is recurrent, meaning that the output at one step is fed back into the system creating a loop. LSTM is capable of doing classification and prediction problems, because its architecture is conducive to learning long-term dependencies between data points. An LSTM unit does not automatically apply previous signals to the current step, but uses a set of cells to learn which previous time steps are useful to predict the current one. The unit is comprised of a cell, an input gate, an output gate, and a forget gate that work together to be able to forecast the next step. These gate units learn to open and close, controlling the constant error flow.

As it's a recurrent layer, an LSTM can be adapted to time series data. In this model, we replicate the architecture of the NASA LSTM paper, described in Figure 4-6. The architecture is comprised of an input layer, two LSTM layers with 80 hidden units, two dropout layers after each LSTM, and one output layer. The output layer is the value at the next time step that we are trying to forecast. If we wanted to predict multiple steps ahead, then we would need to adjust the number of units at the output layer. Similarly, if we wanted to change the length of the sequence, which is the pattern that the model uses to generate a prediction, then we would have to adjust the input layer to reflect that. The dropout layers are there to introduce regularization to

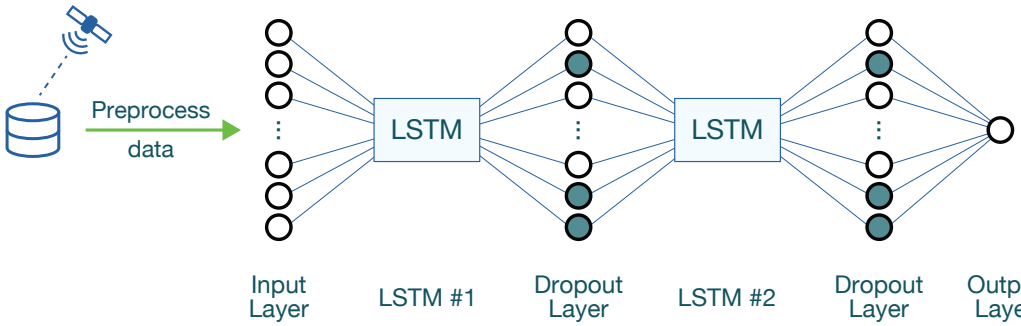

Figure 4-6: LSTM Architecture.

the network. In neural networks, models tend to overfit to the training data, leading to poor performance on the test data. That's why the introduction of regularization techniques such as dropout utilize better performance on the testing set.

During testing, we varied multiple parameters in the architecture of the stacked LSTM system. We varied the number of hidden units in each LSTM layer, the number of stacked LSTM layers, and the train/test split of the data. Each of these led to different results, which we go over in the next section.

4.3.3 LSTM Autoencoder

In machine learning, an autoencoder is a kind of artificial neural network that is used to learn efficient codings of patterns in an attempt to recreate them. Autoencoders are unsupervised, as you do not need to provide labels for the input sequences. The model aims to replicate the input sequence based on the data representation it has learned from previous sequences. Thus, the autoencoder compresses data in the input layer to an intermediate representation and then decompresses the encoding to an output that resembles the original data. As such, autoencoders are tools of dimensionality reduction. The performance of any autoencoder model is measured by the ability of the model to recreate the input sequence.

There are multiple types of autoencoders such as regular, denoising, spare, and variational autoencoders. The variational autoencoder type contain a generative model. Their latent layers are continous, allowing random sampling and interpolation. A regular autoencoder is comprised of an input layer, an output layer, and

one or more hidden layers connecting them. The input layer has to have the same number of nodes as the output layer, because we are trying to predict the input X rather than an output Y.

Regular autoencoders have a problem dealing with variable length sequences, because they are designed to work with fixed length inputs. One way to handle this problem is to use LSTM-based autoencoders that organize their architecture using what is known as the Encoder-Decoder LSTM model. This type of autoencoder is capable of supporting variable length input sequences. It also capitalizes on the recurrent LSTM layer to learn the temporal ordering of input sequences and other long-term dependencies.

Using Keras, an open-source machine learning library, the LSTM autoencoder can be implemented as follows:

1. LSTM Layer.
2. RepeatVector, which repeats the input n times, ie. if the input was of shape (num_train, seq_length) it changes it to $(num\_train, n, seq\_length)$.
3. LSTM Layer (with $return\_sequences = True$).
4. TimeDistributed, which is used to condense the input to a dense layer with $m = 1$ elements corresponding to the sequence points.

The LSTM Encoder-Decoder takes in an input which is the time series sequence. The output is the recreated sequence, as deconstructed by the LSTM decoder layer. In order to measure how accurately the model has reconstructed the input sequence, we use the mean squared error, which is outlined in chapter 5.

4.3.4 Linear Autoregression

Linear regression is a statistical technique that has been used for a long time across all sciences and humanities fields. Data scientists, physicists, economists, and so many others benefit from the power of the method. Linear regression studies the relationship between a dependent variable and several independent ones, trying to

estimate the coefficients of the latter variables. Fundamentally, we have data for both the dependent and independent variables. Often, linear regressors use a method called ordinary least squares to determine the coefficients of the variables and the intercept, if any. Linear regression written in matrix notation is

$$y = X\beta + \epsilon$$

$$\text{where } y = \begin{bmatrix} \mathrm{y}_1 \\ \mathrm{y}_2 \\ \vdots \\ \mathrm{y}_n \end{bmatrix}, X = \begin{bmatrix} 1 & \mathrm{x}_{11} & \dots & \mathrm{x}_{1p} \\ 1 & \mathrm{x}_{21} & \dots & \mathrm{x}_{2p} \\ 1 & \vdots & \ddots & \vdots \\ 1 & \mathrm{x}_{n1} & \dots & \mathrm{x}_{np} \end{bmatrix}, \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}.$$

The aim of linear regression is to estimate values for the β coefficients, using data points of the dependent variable, y, and the independent variables, x. There are multiple methods of estimation for these coefficients, such as maximum likelihood estimation, ridge regression, and generalized least squares. However, the most commonly used estimation method is ordinary least squares (OLS). In OLS, the objective function we are trying to minimize is the sum of the squares of the differences between the predicted values of y obtained from the regressor and the observed dependent values.

$$\hat{\beta} = argmin_{\beta}(||y - X\beta||^2)$$

In other words,

$$X\hat{\beta} = y$$

$$X^T X\hat{\beta} = X^T y$$

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

The matrix $X^T X$ is invertible if and only if all its columns are independent. That occurs when the independent variables are not perfectly multicollinear with the

dependent variable. This means that the output variable cannot be linearly predicted from the independent variables with a substantial degree of accuracy. In situations where there is perfect multicollinearity, the matrix X has less than full rank, and therefore the matrix $X^T X$ cannot be inverted, meaning that for a general linear model, with $y = X\beta + \epsilon$, there doesn't exist an OLS estimator.

Autoregressive (AR) models are a type of linear regressions, where the independent variables are a time lag of the previous values of the dependent or output variable. However, AR models do not satisfy the standard assumptions for least squares regression. With some assumptions, like stationarity, independently and identically distributed (iid) errors with zero mean and constant variance, AR models can be accurately estimated by least squares. In the spacecraft time series context, we make the assumption that the aforementioned conditions hold, even though in practice we know that they may not, to test the effectiveness of linear regressors in detecting anomalies. For example, we know for sure that the stationarity assumption does not hold, due to unforeseen shocks that the spacecraft can experience or different seasons/periods of time in which the environmental factors of the vehicle changes. Nevertheless, we make the assumption that these factors will not affect our data, and rely on our dynamic error thresholding techniques to capture this non-stationarity.

As linear regression is such a widely-used tool, there were multiple toolkits that could have been used in our implementation. We relied on the `sklearn.linear_model` *LinearRegression* module when running experiments for data using a linear regressor. Sklearn's linear regression can be adapted to an autoregression setup, when the input is pre-processed to be made up of time series sequences. This was already done in the *rolling_window_sequences* method that was outlined above. The architecture of the system remained very similar to LSTM. In fact, the project aimed to use modular components and functions that could be easily re-deployed to test different models, so all the pre-processing and post-processing functions remained the same for this model, and we only changed the statistical learning method in the middle. Currently, the pipeline is set up to handle univariate time series data, but it can be adapted to multivariate signals by using vector autoregressions (VARs).

In a lot of settings, linear regression performs just as well as advanced machine learning techniques. The ability of linear regression to do well makes it hard for data scientists and researchers to use advanced machine learning and deep learning techniques. In industry, where decisions are driven more by strategy and costs, a lot of experts cannot provide justification for the application of state-of-the-art models. This results in the adoption of linear regression in the technology stacks of companies. As such, we use linear regression in our analysis to answer the question of whether using more sophisticated models provides a tangible difference in the performance of the system.

4.4 Post-Processing: Anomaly Detection

In our work, we implement the anomaly detection method that was used in the NASA paper. In this section, we outline the theory behind the anomaly detection method. The input to the post-processing model is a list of forecast values and their corresponding realized values. The first step is to compute the the prediction error as given by:

$$e = |y_t - \hat{y}_t|$$

As such, a vector of errors is generated for each of the data points in the test set:

$$\mathbf{e} = [e_{t-h}, e_{t-1}, ..., e_t]$$

where h is the length of the window sequences (the number of sequences is the length of data points minus the window size). After computing the errors, they are passed through a smoothing algorithm such as the exponentially-weighted moving average (EWMA). Thus, we now have the smoothed errors vector $\mathbf{e_s}$.

To determine whether the computed errors constitute anomalies, we use an adaptation of dynamic error thresholding technique mentioned in the NASA paper. The main difference, as we outline below, is that we do not implement the pruning method.

The unsupervised method aims to compute a threshold, ϵ, that is found by taking the point that maximizes the following:

$$\epsilon = \mu(\mathbf{e_s}) + \mathbf{z}\sigma(\mathbf{e_s})$$

where ϵ is the maximum error from the set of smoothed errors such that

$$\epsilon = argmax(\epsilon) = \frac{\frac{\Delta\mu(\mathbf{e_s})}{\mu(\mathbf{e_s})} + \frac{\Delta\sigma(\mathbf{e_s})}{\sigma(\mathbf{e_s})}}{|e_a| + |E_{seq}|^2} \tag{4.1}$$

where

$$\Delta\mu(\mathbf{e_s}) = \mu(\mathbf{e_s}) - \mu(\{e_s \in \mathbf{e_s} | e_s < \epsilon\})$$

$$\Delta\sigma(\mathbf{e_s}) = \sigma(\mathbf{e_s}) - \sigma(\{e_s \in \mathbf{e_s} | e_s < \epsilon\})$$

$$e_a = \{e_s \in \mathbf{e_s} | e_s > \epsilon\}$$

$$\mathbf{E_{seq}} = \textit{contiguous sequences of anomalies in } \mathbf{e_a}$$

The value of z lies in a range between 0 and 10, and the error thresholding mechanism tries out different values to see which one minimizes the cost, as shown in equation 4.1 above. The intuition behind the dynamic error thresholding technique is that we wish to identify anomalies that, if removed, would result in the largest percent decrease in the mean and standard deviation of the vector of smoothed errors. The thresholding technique also computes a score for each anomaly, in order to be able to compare the severeness of the outliers. The formula for the severity of the anomaly is given by:

$$s^{(i)} = \frac{max(e_{seq}^{(i)}) - argmax(\epsilon)}{\mu(\mathbf{e_s}) + \sigma(\mathbf{e_s})}$$

To mitigate false positives, the NASA paper introduces a pruning measure that is aimed at reducing the sensitivity towards outliers that aren't far enough to be anomalies. To do this, a list is created that contains the maximum errors in all anomaly sequences, sorted in descending order. At the end of the list, the largest error value that hasn't been flagged as anomalous is also added to the list. The list is then stepped through and the percent decrease at each iteration is calculated. If

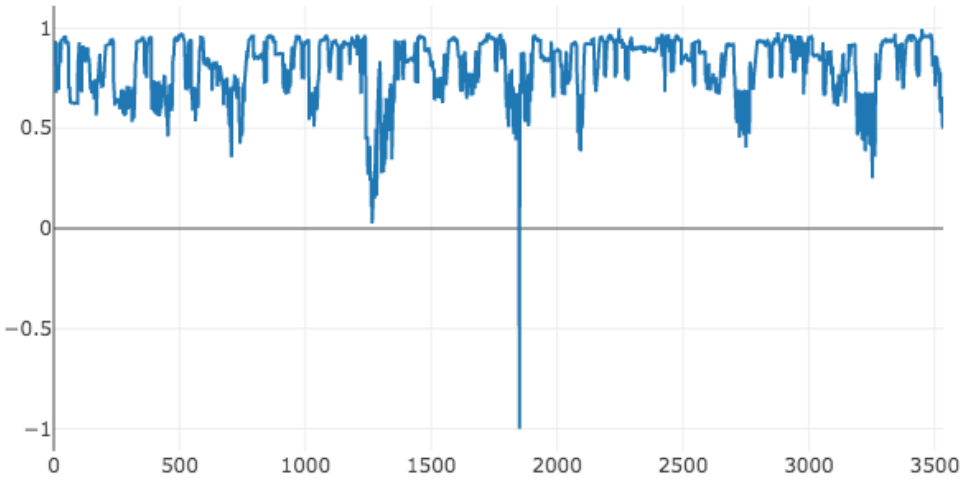

Figure 4-7: Signal P-11.

the percent decrease exceeds a certain threshold p, which they found to be good in practice at $p = 0.13$, then the error sequence is flagged as an anomaly. In our work, we do not use this pruning procedure, but still maintain good performance across different models. Using this pruning method would definitely improve the precision and therefore $F_{0.5}$ score, but we leave this as an additional step for further work.

4.5 Sample Pipeline

As shown in the previous sections of this chapter, our pipeline consists of three main components: data pre-processing, model, and post-processing. To better outline the way the different components work together, we will step through each module and show how an input signal changes in each section. We choose a sample signal, called P-11, to exhibit the way a pipeline would run on one signal.

As can be seen in Figure 4-7 signal P-11 is continuous, and follows a somewhat periodic trend. The spike in the middle of the signal is actually an anomaly, as it has

been indicated anomalous in the *labeled_anomalies* file. In this section, we try to see if our model can correctly predict that this point is an anomaly.

The first thing that the pipeline does is aggregate the data according to a specific interval. This is done using the *time_segments_average* method, outlined in section 4.2.1. Since the NASA paper did not aggregate their data, we set the interval equal to 1, meaning that we average each data point with itself, resulting in no change for the input signal. This method is nevertheless very important to have if the data is sampled at a high frequency, which is very common in time series applications.

The next step in the pre-processing section is the creation of sequences from the input signal. To this end, we used the *rolling_window_sequences* method described in section 4.2.2 which takes in the time series data and the window size and target size, corresponding to the length of the sequence and number of values ahead to predict respectively. In our experiments, we varied the window size, using lengths of 100 and 250; however, we kept the target size to one, meaning that we are only trying to predict the next time step given the input sequence.

Now that the signal has been formatted, the next step is to pass it through the model. We have outlined four different models in section 4.3 above, which are LSTM, MLP, LSTM Autoencoder, and AR. These models all have different parameters that can be tuned, but they operate on the same inputs/outputs, with the exception of the autoencoder, which uses the input as its output when it's encoding and then decoding the sequence. For more details about the mechanism of these models, we refer the reader to the corresponding subsections in 4.3 that discuss in more detail the theory behind each model. The model uses the training part of the data to learn the weights of the time series regressor that will be used to predict the testing data. When we pass in the testing data, a vector of forecast values is generated as the output. This output is carried forward to the next step of the pipeline, which is the post-processing. As we can see in Figure 4-8, the model predicts the signal in orange, but there is a discrepancy between the actual signal in blue and the forecast. Figure 4-9 contains a zoomed-in version of the forecast. We can see that the model doesn't exactly replicate the signal, but it follows it closely. These qualitative descriptions

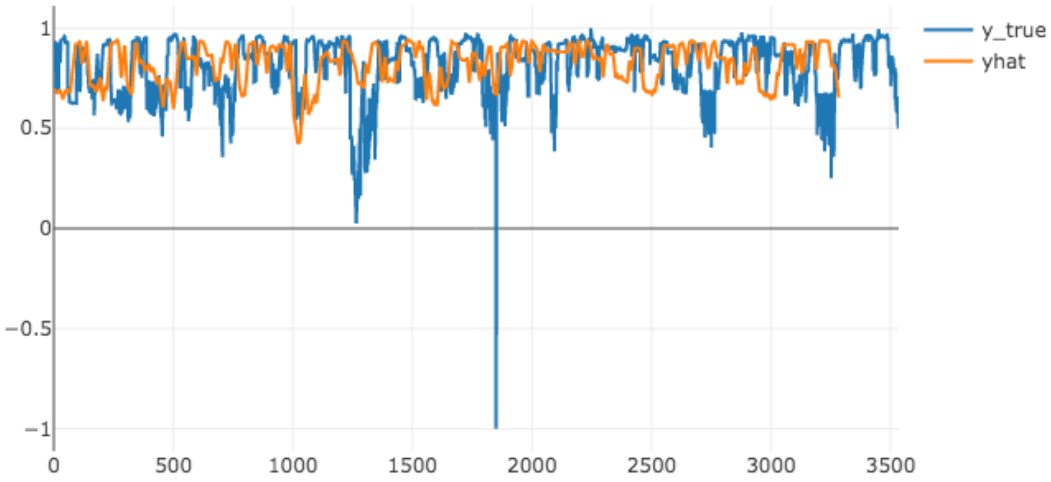

Figure 4-8: Signal P-11 and the forecast from NASA LSTM Model.

are difficult to measure, so we resort to metrics such as mean squared error that we discuss in section 5.1.

In the post-processing phase, the generated forecast, $\hat{y}$, is cross-referenced with the realized signal y_{true} to see whether the true signal was in line with the model's expectations or not. If it happened that the difference between the two was very large, then it means that the signal has diverged from the assumptions that the model has learned and was therefore operating under. Thus, we flag the error as an anomaly, according to the dynamic error thresholding explained in section 4.4. Figure 4-10 shows the region that is flagged by our algorithm as anomalous. The region is the one that is just preceding the huge spike that we referenced earlier. As such, the model was able to accurately predict the anomaly by flagging the region right before it happened. What likely happened in this case is that the large anomaly was preceded by some minor abnormalities, which the model detected and flagged.

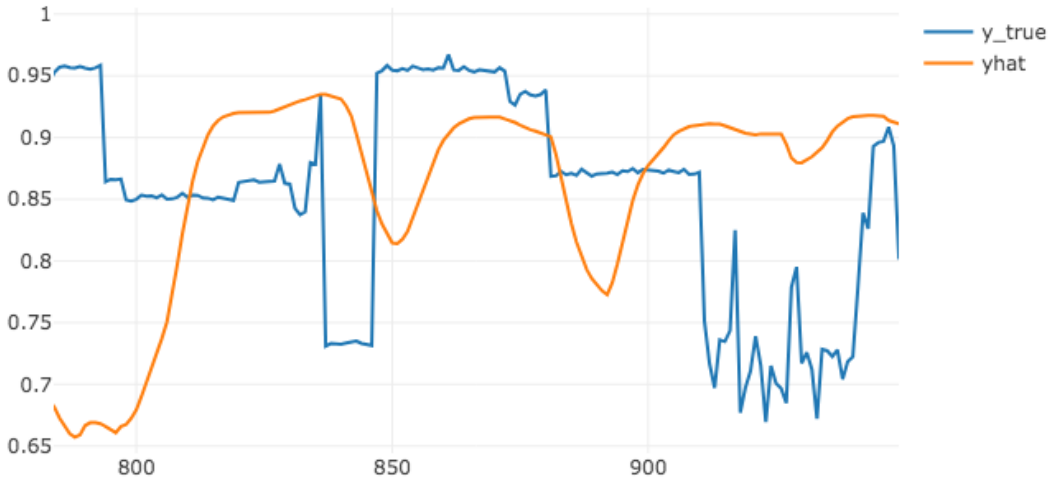

Figure 4-9: Close-up on signal P-11 and the forecast from NASA LSTM Model.

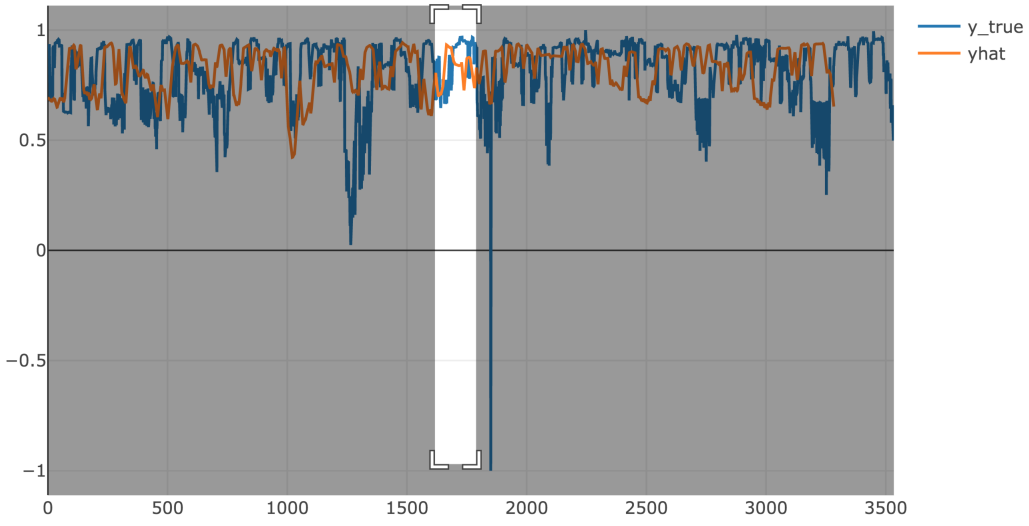

Figure 4-10: Signal P-11 and Forecast from NASA LSTM Model. Flagged Anomaly in Shaded Region.

Chapter 5

Analysis

5.1 Mean Squared Error

The NASA spacecraft telemetry dataset has 82 signals, and for each architecture a model was trained for each of the signals. We primarily dealt with univariate signals, but the models can easily be adapted to work with multivariate signals, by changing the size of the input vector and making it a matrix. Working with multivariate data can reveal additional insights and improve the accuracy of predictions, because oftentimes signals are correlated, especially when telemetry signals are coming from the same vehicle. However, using multivariate signals limits the interpretability of anomalies. For instance, if our models worked with 10s of signals at a time and an anomaly is detected, then it is hard to determine which signals are the ones that caused the anomaly to arise. This is why in our analysis we focused on evaluating univariate time signal models. A sample training script for one of the architectures can be found in Appendix B. The code shows the procedure to train 82 different models given a certain architecture.

During training of the models, the mean squared error was used as a metric for the training for each of the signals. Mean squared error (MSE) is an average of the squares of the errors between the predicted values and the real ones. The best possible is 0, and it is achieved when the predicted value is identical to the real one. The larger the MSE, the worse the predictions are, and there is no theoretical bound

on how large MSE can get. The formula for MSE is given by

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

The MSE was used to assess the models in two ways. First, while training the MSE was used as a measure of how well a certain architecture was able to learn the data and start making accurate predictions on the training sample. When the validation error exceeded the training error, then we knew that the model didn't generalize well, and that we have overfit the data. Alternatively, when the validation error was smaller than the training error, then the model was able to generalize, since it performed better on previously unseen data. We also used the training MSEs, averaged across all 82 signals, to determine which architectures performed better. A summary of the MSEs can be found in Table 5.1.

We ran our experiments on all 82 signals, training the model once on each signal. The reason that we don't train one model for all the signals is that they exhibit different features that the model seeks to learn. Therefore, a model that is trained on the entire dataset won't generalize well in our case, because the fitting signal A-1, for instance, would have no added benefit when trying to predict signal D-5. Thus, when we compute the MSE and anomalies in this chapter, we do so by generating a script that goes through each signal independently, trains a model, forecasts the testing data, and then computes anomalies as predicted by the anomaly detection method. The MSE of a model is computed by averaging the MSE across all the signals, and that's why we have a associated standard deviation in table 5.1. Moreover, the anomalies were aggregated across the signals and for each model the $F_{0.5}$ score was computed. The $F_{0.5}$ results are discussed in section 5.2.

Additionally, the MSE was evaluated by comparing it to a randomly shuffled MSE. The intuition behind this baseline is that we want to capture how well our model forecasted the signals, as opposed to a random shuffle of the input signal. We set up a script that shuffled each signal and computed the MSE between the shuffled signal and the original one. The MSE of the 82 signals were averaged to give an

Table 5.1: MSE Summary across 82 signals. Autoregression doesn't have a training MSE as the neural network models do, since the model is fitted differently.

Model Name	MSE Value (std deviation)
Shuffled MSE (baseline)	5.4×10^5 (1×10^5)
Shifted MSE (baseline)	2.1×10^6 (2×10^6)
NASA LSTM	0.03 (0.05)
NASA LSTM hidden_units=100	0.03 (0.05)
NASA LSTM 3-stacked	0.03 (0.05)
MLP x_len=250	0.095 (0.2)
MLP 6-dense layers	0.085 (0.2)
MLP x_len=100	0.053 (0.1)
LSTM EncDec x_len=100	1.6×10^5 (1×10^6)
LSTM EncDec x_len=250	0.015 (0.05)

average MSE of 5.4×10^5 which is a far larger number than the MSE that we achieve with our models, but it is nevertheless a baseline. Another baseline that we have computed, across all signals, is the MSE of the signal shifted to the right by one. That is, the MSE computes the difference between x_t , x_{t+1} at each point. This MSE is a good measure, if the signals exhibit low variance across time steps. However, we found that this MSE measure was larger than the shuffled MSE, with a value of 2.1×10^6 (also larger than the MSE of our models).

5.2 F-0.5 Score

As we can see from the mean squared error tables above, MSE is not good at separating which models perform better on learning the data, because the numbers are only marginally different. Moreover, the most important metric we are looking at is the ability to detect anomalies, which is not determined by the MSE. As such, we work with another metric that is often used to assess the quality of classifications. For the system we are building, we would like to predict as many anomalies as possible, while decreasing the amount of false alarms that we get. In statistics terms, a type I error occurs when a normal point is incorrectly labeled as anomalous, and this is also known as a false positive. A metric that corresponds to type I error is precision, which

Table 5.2: $F_{0.5}$ Scores of the Models.

Model Name	$F_{0.5}$ *Score*
NASA LSTM (paper)	0.69
NASA LSTM (our implementation)	0.61
NASA LSTM hidden_units=100	0.34
NASA LSTM hidden_units=40	0.76
NASA LSTM 3-stacked	0.69
MLP x_len=250	0.42
MLP 6-dense layers	0.59
MLP x_len=100	0.23
AR x_len=100	0.25
AR x_len=250	0.57
LSTM EncDec x_len=100	0.34
LSTM EncDec x_len=250	0.66

is the percentage of selected anomalies that are true positives versus false positives. A type II error, on the other hand, occurs when a point that is anomalous in nature is not detected. Such an error is also called a false negative, and the recall metric aims to capture how many true positives were flagged as a total of all the anomalies (true positives + false negatives).

We created a script that looks at the anomalies generated by the model and checks whether they are present in the *labeled_anomalies* file. If there is an intersection between an anomaly sequence in our prediction and the *labeled_anomalies* file, then the anomaly is considered to be a true positive. If an anomaly was predicted by our model but doesn't exist in the anomalies file, then it is counted as a false positive. Finally, if an anomaly exists in the *labeled_anomalies* database but is not predicted by our model, then we count it as a false negative. With these metrics, we can now compute the $F_{0.5}$ score, which is a representation of the anomaly predictive power of our system.

In the NASA paper, the model without parametric pruning ($p = 0$ as in our system), the performance was 0.88 (precision), 0.67 (recall), and 0.69 ($F_{0.5}$). We can see from Table 5.2 that only our LSTM model with 40 hidden units outperformed the NASA baseline, while the 3-stacked LSTM and LSTM autoencoder with $x\_len = 250$

Table 5.3: Precision & Recall Scores for the Models.

Model Name	Precision	Recall
NASA LSTM (paper)	0.88	0.67
NASA LSTM	0.61	0.62
NASA LSTM hidden_units=100	0.35	0.32
NASA LSTM hidden_units=40	0.78	0.67
NASA LSTM 3-stacked	0.69	0.70
MLP x_len=250	0.42	0.43
MLP 6-dense layers	0.59	0.61
MLP x_len=100	0.23	0.22
AR x_len=100	0.25	0.28
AR x_len=250	0.58	0.53
LSTM EncDec x_len=100	0.38	0.24
LSTM EncDec x_len=250	0.83	0.35

performed close to the reference paper. We will now explore each model individually to discuss what changes worked well for each architecture.

For the NASA LSTM model, we obtained similar performance to the paper when we implemented the model, as described in their paper. Varying the number of hidden units worked very well when we decreased the number from 80 to 40, but performed poorly when we increased the number to 100. In fact, decreasing the number of hidden units to 40 was the most successful model we tried, out of all the different architectures, yielding an $F_{0.5}$ score of 0.76. Another good improvement over the NASA architecture was adding a third LSTM layer to the two-stacked model that they implemented, which yielded an $F_{0.5}$ score of 0.69, our second-best model in practice. The LSTM model, however, took the most time to train out of all the architectures that we tried. We also kept the length of the sequences constant, where $x\_len = 250$ for all the LSTM experiments that we ran.

The MLP model trained much faster than the LSTM model, and was second fastest after the linear autoregression model. Using an MLP as described in Section 4.3.1 and with $x\_len = 100$ yielded the lowest score of 0.23. This score was improved to 0.42 when we set $x\_len = 250$. Finally, when we increased the number of dense layers from 3 to 6, the $F_{0.5}$ score went up to 0.59. MLP doesn't give us the best results,

but given that it trains faster than LSTM, it provides a more efficient alternative.

In a lot of machine learning experiments, linear AR models are used to reference the performance of deep neural networks, since linear autoregressions are among the most efficient and commonly used tools in data science. In our experiments, the AR model performed decently well when $x\_len = 250$, giving an $F_{0.5}$ score of 0.57. The model didn't do that well when $x\_len = 100$, but that seemed to be the trend across the other models such as MLP and LSTM EncDec. Given the fact that AR trains way faster than all the other models, it is good for use cases where computing power is limited, but overall it is not a good architecture if we are looking for top-of-the-line results.

Finally, the LSTM Encoder-Decoder model performed generally well, achieving an $F_{0.5}$ score of 0.66 when $x\_len = 250$. The most notable thing about this model can be found in Table 5.3 that outlines the precision and recall scores of the models. We can see in the last row that the precision score of the LSTM EncDec model was the highest among all the models, with a score of 0.83. This means that the LSTM EncDec was very accurate when it came to predicting the anomalies, but interestingly the model's recall score was on the lower end of the spectrum, indicating that it couldn't successfully find all possible anomalies.

5.3 Further Work

There are several next steps that would make sense for this project. We identify three main tasks that can be done as follow-ups to the current work that we presented.

First, people interested in conducting further research can try out different parameters for our models and different anomaly detection techniques altogether. A more thorough analysis can be carried out by users who can choose a few signals they would like and run autotuning tools on the signals to figure out what the best hyperparameters are. We couldn't do this for 82 signals and 4 models, as it would be too complex to do for all the signals simultaneously. Another area that can be explored is the trying a different anomaly detection technique, such as x-sigma thresholding.

Second, for those who would like to apply this model in industry, it would be very important to be able to run this system in real-time and detect anomalies on-the-go as they happen. In order to do this, engineers need to keep in mind different data normalization techniques, in order to keep the input in the range of [-1, 1]. One effective way of doing this is to clip any data that is larger than the minimum and maximum possible values that we know ahead of time that sensors can't and shouldn't go beyond.

Third, we encourage researchers to make use of our modular system to try out different forecasting architectures. In our work, we explored linear AR, MLP, stacked LSTMs, and LSTM autoencoders, but there are so many other models that can be used. For instance, there are bidirectional LSTMs, variations of linear regression, variational autoencoders, convolutional neural networks, recurrent neural networks, memory-based models, sequence-2-sequence, and gated recurrent units. It would also be useful and interesting to explore how some of these models can be adapted to work with multivariate signals. The current architecture should work with multivariate signals, with only a few minor changes.

5.4 Conclusion

In this work, we analyzed 82 signals coming from NASA spacecraft in order to detect anomalies. We built a modular infrastructure such that we can rapidly prototype different models, then we compare and contrast different architectures such as modified versions of the NASA LSTM model, Autoregressive (AR) model, LSTM autoencoder, and Multilayer Perceptron (MLP) model. We also varied the parameters of these models to improve the ability to detect anomalies, as measured by the $F_{0.5}$ score.

We find that increasing the length of the input sequence x_len improved the performance of the model. We were able to achieve an improvement over the NASA paper's $F_{0.5}$ score of 0.69, when we decreased the number of hidden units to 40 and achieved a score of 0.76. In contrast, when we increased the number of hidden units from 80 as in the paper to 100, the results decreased to 0.34. We also got close results

when we added a third LSTM layer to the stacked LSTM model, achieving 0.69 score.

The AR model performed as well as the MLP model when the length of the sequence is 250, but both models performed lower than the NASA paper baseline. The LSTM Encoder-Decoder didn't obtain great $F_{0.5}$ score, but it yielded the highest precision score of 0.83. Overall, the best performance came from small improvements to the NASA LSTM model.

In terms of next steps, we would recommend exploring alternative error thresholding techniques such as x-sigma. We would also try out different machine learning models such as variational autoencoders, bidirectional LSTM, seq2seq, gated recurrent units, among other models. Exploring different architectures would allow us to converge to systems that best model the data and work together with humans to help detect anomalies as soon as they happen.

Bibliography

[1] Samit Bhanja and Abhishek Das. Impact of data normalization on deep neural network for time series forecasting. *CoRR*, abs/1812.05519, 2018.

[2] Kyle Hundman, Valentino Constantinou, Christopher Laporte, Ian Colwell, and Tom Soderstrom. Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 387–395, New York, NY, USA, 2018. ACM.

[3] Pankaj Malhotra, Anusha Ramakrishnan, Gaurangi Anand, Lovekesh Vig, Puneet Agarwal, and Gautam Shroff. Lstm-based encoder-decoder for multi-sensor anomaly detection. *CoRR*, abs/1607.00148, 2016.

[4] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[5] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.