Modeling Control Signals for Reconstruction-based Time Series Anomaly Detection

by

Grace Y. Song

S.B. Computer Science and Engineering, Massachusetts Institute of Technology (2024)

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

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Grace Y. Song Department of Electrical Engineering and Computer Science May 20, 2024
Certified by:	Kalyan Veeramachaneni Principal Research Scientist, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Modeling Control Signals for Reconstruction-based Time Series Anomaly Detection

by

Grace Y. Song

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Automated time series anomaly detection methods can provide insights while reducing the load placed on human experts in a variety of settings. Machine-generated signals, such as those produced by sensors, often contains control signals in addition to the target observation signal. These signals may provide additional insight about the normal vs. abnormal properties of the observation signal. Despite this fact, even recent anomaly detection methods using deep learning give limited consideration to the relationship between observation and control signals, often failing to handle the control signal at all. This work proposes pre-processing, modeling, and evaluation methods for multivariate, heterogeneous time series to examine how using information from the control signal can improve anomaly detection. We develop a deep learning reconstruction-based pipeline and test its performance on the NASA Soil Moisture Active Passive (SMAP) satellite and the Mars Science Laboratory (MSL) Rover, which contains heterogeneous sensing data from exploratory missions. The pipeline follows the *Sintel* machine learning framework and is accessible through the *Meissa* library, which builds on the capabilities of the open-source library *Orion* for end-to-end unsupervised time series anomaly detection pipelines.

Thesis supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

First and foremost, I would like to thank Kalyan Veeramachaneni, who welcomed me into the lab from our first conversation and provided valuable guidance on the project, not to mention insight into how well-designed data analysis tools can transform lives.

This thesis also wouldn't be anywhere with Sarah Alnegheimish, who helped me get acquainted with *Orion* and the Sintel ecosystem throughout the year and answered countless questions without hesitation. Her kindness and work ethic is one of a kind.

My MEng experience was also made special by Ola Zytek, Sara Pìdo, Linh Nguyen, and other members of the Data to AI Lab. I always feel honored to work among such talented people and will miss the discussions. Similarly, I wanted to recognize members of the 9.66 and 6.1800 teaching staff, led by Prof. Joshua Tenenbaum and Katrina LaCurts, whom I had the pleasure of working with. Jordan, Alicia, Phoebe, Tony, and others: you made the job so much fun.

Lastly, I wanted to express additional gratitude to Katrina LaCurts for supporting me as my instructor, undergraduate officer, and general mentor. MIT had its fair share of challenges, but you helped me believe in myself.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Time series anomaly detection (AD) methods can be applied to problems ranging from identifying fraudulent activity or exceptional events in financial data to detecting faults in industrial workloads. In the latter case where anomalies may indicate device malfunction or human error, AD methods also help anticipate and prevent imminent failures.

One can imagine the data from such settings growing rapidly in volume and complexity beyond what a human can feasibly analyze, especially if the signal contains multiple channels that each must be monitored. Automated AD methods – in particular deep-learning methods – have grown in popularity due to their ability to take in large amounts of training data at once while extracting the underlying features of the time series.

A particular setting that the thesis considers is when features of multivariate signals are related; this is the case with **control signals**, which we define as signals resulting from intentionally programmed intervention such as human operation of a switch. The value of a control signal signifies a state of the device that may correlate with or directly affect other signals. We provide an example from the Soil Moisture Active Passive (SMAP) dataset used for this thesis in Figure 1.1. The first plot shows the continuous observation channel with the expert-labeled anomalous region highlighted in yellow, while the second and third plots show the values of 2 control signal channels.

If we only looked at the observation channel, we might be tempted to think that the middle segment starting just before 2011 could also be an anomalous interval due to an abrupt change in the values range. However, looking at the control signal channels can give us more information: channel 3 shows the density of positive values increasing with the rising pattern of the time series until right after the start of the anomalous segment. Here, the state of the control signal is correlated with the relative magnitude of the observation signal for all of the non-anomalous region, but becomes flat at the end. Channel 4, on the other hand, is sparse but the location of its positive values is particularly telling, as they correspond to the start and end of the middle segment where the range of signal values become compressed. This might indicate that channel (4) in particular becomes equal to 1 upon a state change. As such, being able to model a signal using all of its channels can help provide additional insight into the AD process.

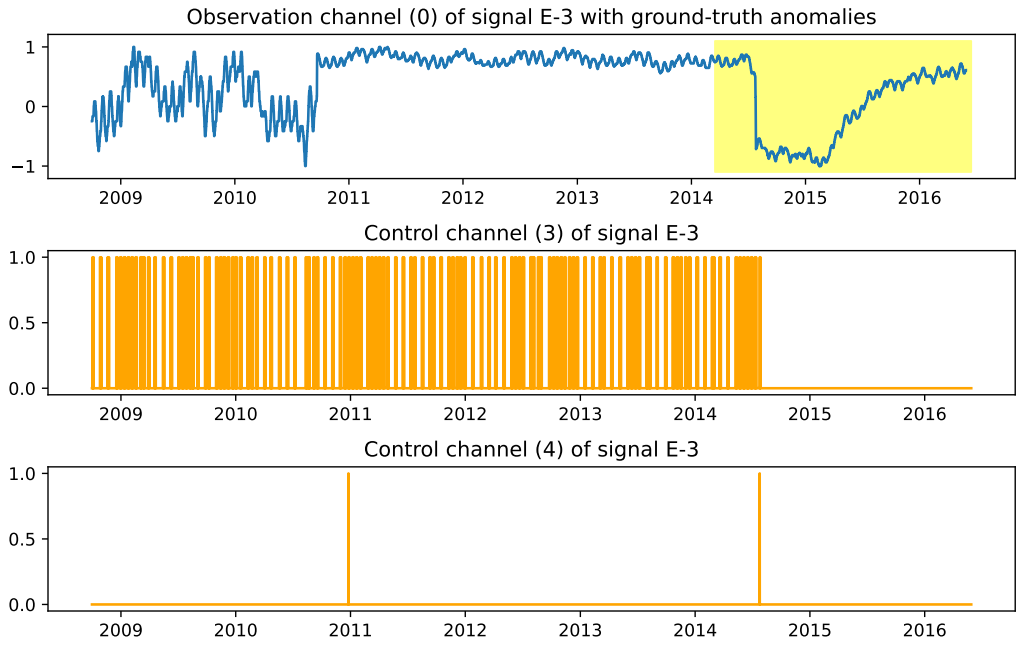

Figure 1.1: Plot of control and observation channels of signal 'E-3' from the SMAP dataset. The yellow highlighted region indicates a known anomaly.

Previous work on deep learning models for AD such as [1] train and test their models on

univariate signals even if the original input signal contains multiple channels, as is the case of the NASA datasets used in this thesis. Part of this reason is because popular benchmark datasets such as the Yahoo S5 dataset[1] and the Numenta Anomaly Benchmark [2] are univariate. Although control signals are not typically assumed to be anomalous (i.e. no anomaly detection is performed on them), they can provide additional insights into the patterns and latent state of the target signal. For instance, if the control signal mediates a particular property of the observation signal(s) (e.g. the amplitude), one can model the inter-channel correlation to more accurately detect abnormal segments. Furthermore, the state of the control signal can also indicate which segments should not be considered anomalous at all if otherwise abnormal changes in the observed signal's values are directly tied to a change in the control signal's state.

Deep anomaly detection methods also differ in how much they rely on ground-truth labels to identify anomalies and can be classified as supervised, unsupervised, or semi-supervised [3]. Supervised approaches use labeled anomalies during the training process, typically by having the model directly classify whether a point is anomalous and using the label to score. In unsupervised anomaly detection, no labeled anomalies are required; rather, the model aims to learn the "normal" features of the data during training and anomalies are identified by looking for points or intervals where the observed data deviates significantly from the model output. Semi-supervised approaches aim for the model to accurately learn the patterns of the data through providing an incomplete subset of labeled samples [4]. For instance, a semi-supervised approach could involve training on only normal instances and testing on both normal and anomalous cases. Such an approach can be desirable in applications such as fault detection, where labeled anomalies are harder to obtain.

Unsupervised and semi-supervised methods tend to be more practical since the nature of anomalies is that they are rare occurrences and thus depending on the task, a well-labeled dataset may not exist at all. There have been both successful prediction-based and

[1] https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70

reconstruction-based unsupervised pipelines for multivariate signals in recent years such as the ones in [5] [6], but few can capture multiple patterns that may be the result of changing states in other channels of the data. This could be in part due to (a) not explicitly accounting for different characteristics between control and observed signals, or (b) not capturing the relationship between the signal's channels.

1.1 Contributions

This thesis presents a framework for modeling control signals in AD using reconstruction-based methods and evaluates an example LSTM auto-encoder pipeline called `MixedLSTM` developed within the framework. The goal of improving anomaly detection is two-fold:

- First, we aim to improve the reconstruction accuracy by incorporating control signals;
- Second, we aim to more finely filter candidates for anomalies using the state of the control signal, reducing the incidence of false positives/negatives.

The pipeline and its model, pre-processing, and post-processing primitives are accessible through a user-friendly interface in the *Meissa* library, which extends the existing functionality of the *Orion*[2] AD library.

1.2 Thesis Organization

The work is organized as follows: Chapter 2 provides background on time series anomaly detection and related work and describes *Orion* and the Sintel[7] ecosystem in greater detail. Chapter 3 outlines the methodology for developing the `MixedLSTM` pipeline and the relevant primitives, while Chapter 4 describes how the objects and functionality can be accessed through *Meissa*. Chapter 5 provides background on the benchmark data used and compares the performance between naively handling control signals in *Orion* vs. explicitly handling

[2] https://github.com/sintel-dev/Orion

them in *Meissa* in terms of both model architecture and evaluation. Lastly, Chapter 6 examines these results and proposes future directions.

Chapter 2

Sintel Background

2.1 Time series

Time series data presents a unique framework for analysis, characterized by its sequential arrangement of data points indexed by time. Central to its structure is the concept of *temporal dependency*, wherein each observation's value is influenced by its preceding ones. This inherent time ordering underscores the significance of understanding the data's historical context. Furthermore, time series often display discernible *trends*, indicating long-term shifts in values, and *seasonality*, manifesting as cyclical patterns at regular intervals. These trends and seasonal fluctuations encapsulate crucial insights into underlying phenomena, necessitating specialized methodologies for their detection and interpretation. However, such observable patterns and behaviors can also evolve over time; this *non-stationary* property poses a unique challenge for modeling time series compared to other sequential data such as text.

In addition to temporal dependency and non-stationarity, time series data is also subject to inherent noise, obscuring meaningful patterns amidst random fluctuations. The challenge lies in disentangling signal from noise, a task complicated further by irregular events or anomalies, which typically represent significant departures from the data's expected behav-

ior. Converting the time series into a stationary form facilitates more robust analysis and modeling. Achieving stationarity may require preprocessing steps such as de-trending or differencing, as well as domain expertise [8].

The following sections outline how anomalies are defined and detected, and how the anomaly detection task is set up as an end-to-end pipeline in *Orion* and *Meissa*. A more precise description of how the data is represented and transformed is also provided in Section 2.3.

2.1.1 Anomalies

Generally, we can consider an anomaly as an observation or pattern in data that significantly deviates from the expected or normal behavior, indicating potential errors, outliers, or noteworthy events. Anomalies can represent beneficial or harmful occurrences depending on the context, but we typically want to be made aware of them so that we can handle them separately. For instance, major events such as holidays can lead to spikes in sales, but such data intervals should be excluded when predicting sales over a "normal" week or month.

Anomalies can be broadly categorized as *point* or *contextual* , each requiring different detection approaches and interpretations:

- *Point anomalies* refer to data points that deviate significantly from the overall pattern of the time series. These anomalies stand out as extreme values compared to the rest of the data and can often be detected using statistical measures such as standard deviation, z-scores, or percentile thresholds. *Point* anomalies are characterized by their isolated nature, occurring independently of the surrounding context or neighboring data points. For example, a sudden spike or dip in stock prices amidst relatively stable trading activity would be considered a *point* anomaly.

- *Contextual anomalies*, or collective anomalies if they occur throughout the data, refer to when a time segment is considered anomalous in a specific temporal context, but not

otherwise [2]. Unlike *point* anomalies, *contextual* anomalies may not be global extreme values. Detecting *contextual* anomalies is typically a more difficult task as it requires analyzing subtle temporal patterns, spatial correlations, or multivariate interactions.

2.2 Anomaly detection

What makes detecting anomalies a difficult task? As mentioned in Section 2.1, the mixture of anomalies arising from noise and anomalies arising from phenomena can make it hard to distinguish random outliers from meaningful ones. Furthermore, anomalies occur infrequently by nature, and the scarcity of labeled anomalous data further compounds the difficulty of detection and generalization especially in supervised settings. Despite the inherent challenges, many successful approaches have emerged over the years.

2.2.1 Early anomaly detection methods

Early anomaly detection methods encompass a variety of techniques developed prior to the widespread adoption of deep learning approaches. These methods often rely on statistical or nearest-neighbor/clustering algorithms [9][10] to identify deviations from expected patterns in time series data. One such method is statistical thresholding, where anomalies are detected based on predefined thresholds applied to statistical measures such as mean, standard deviation, or percentiles. Another common approach is using regression models to decompose the time series into trend, seasonality, and residual components and identifying anomalies in the latter; this was done in the Autoregressive Moving Average (ARMA) approach outlined by [11]. Autoregressive Integrated Moving Average (ARIMA) [12] attempts to estimate such residuals while accounting for non-stationarity. Such methods reflect the intuition that we often think of anomalies as outliers in a dataset. Indeed, rule-based methods try to define what an outlier is in the time series in terms of various features (rather than just the standard deviation of the point). These methods have the benefit of being simpler and less

computationally expensive, but their detection ability is often limited to *point* anomalies.

2.2.2 Deep-learning methods

Deep-learning AD methods can be broadly categorized into two kinds: **prediction-based** and **reconstruction-based**. As discussed in Chapter 1, they also vary from supervised, to semi-supervised, to unsupervised. Here we focus on highlighting unsupervised, reconstruction-based AD methods since they are most relevant to the thesis.

Prediction-based methods

Prediction-based methods typically train a model to forecast future values in the time series over sliding windows and evaluate performance based on some defined deviation from the values that were actually observed [13]. The Long Short-Term Memory Network with Dynamic Thresholding (LSTM-DT) used in Hundman et al's work [1] is a popular architecture for prediction-based AD. The paper trained and evaluated their model on the same telemetry data used in this thesis, the Mars Science Laboratory (MSL)[1] and the Soil Moisture Active Passive (SMAP)[2] dataset. Other popular methods include Hierarchical Temporal Memory and Bayesian Networks [14].

While prediction-based methods are useful for efficiently learning feature representation at specified timestamps, they have their limitations. For instance, LSTM-DT methods can be prone to false positives with high anomaly scores at early indices due to the smoothing process. On the other hand, they are prone to false negatives when it comes to *contextual* anomalies in intervals with a simple pattern and/or a small amplitude, as noted in the analysis from Wong et al [15].

[1] https://pds.nasa.gov/ds-view/pds/viewDataset.jsp?dsid=MSL-M-REMS-2-EDR-V1.0
[2] https://smap.jpl.nasa.gov/data/

Reconstruction-based methods

In reconstruction-based methods, a deep learning model such as a recurrent neural network (RNN) is trained to learn patterns in the provided signal and, when prompted, construct another signal. The discrepancy between the model-constructed signal and actual signal is used to identify anomalies. The intuition is that the latent representation of the signal created by the model during learning captures the core patterns of the time series; these core features will remain when it is recovered from the latent representation during decoding while the anomalous features would be missing.

Several reconstruction-based variants of LSTM models exist including LSTM Auto-Encoders (LSTM-AE) [16] and LSTM Variational Auto-Encoders (LSTM-VAE) [17], which learn latent space representations using an auto-encoder with LSTM layers. Such models require carefully incorporating regularization during optimization to prevent overfitting to noise and anomalous segments.

Another popular architecture for reconstruction-based AD are Generative Adversarial Networks (GANs), which consists of using *generators* to map between the input and latent data domain and *critics* to discriminate between the real and generated time-series during training. Geiger et al. [18] developed a time-series anomaly detection GAN (Tad-GAN) architecture with a framework to estimate anomaly scores that outperformed baseline methods such as LSTM, ARIMA, HTM, and LSTM-AE. Experiments such as those from [15] show that reconstruction-based methods are better at capturing a global distribution across the time series but underperform against some prediction-based models when it comes to *point* anomalies.

Hybrid methods

Some recent anomaly detection frameworks have combined prediction- and reconstruction-based methods to complement each method's strengths and weaknesses.

Wong et al.'s Auto-encoder with Regression (AER) model combines an auto-encoder with

an LSTM regressor and trains on a joint reconstruction- and prediction-based loss. With just a vanilla auto-encoder, AER outperformed ARIMA, TadGAN, and LSTM variants across several time series AD datasets, including SMAP [15].

MTAD-GAT, a graph-based model, feeds the output of parallel graph attention (GAT) layers and a Gated Recurrent Unit (GRU) into both a prediction-based model and reconstruction-based model and optimizes their combined loss [6].

The relative success of such architectures above motivate further exploration which may include experimenting with the relative contributions within the joint loss, and incorporating human feedback into the loop.

2.2.3 Related work with control signals

A number of previous studies worked with multivariate signals as sensing is a popular application of AD, and datasets such as SMAP and MSL are widely used for benchmarking. A more relevant example is the work of Hsieh et al [16], who use an auto-encoder model for reconstruction-based AD for a sequence of sensors in a production line. All the sensor signals are binary, indicating where in the chamber the sensor is moving through. To train the auto-encoder, the authors opted for optimizing the pooled mean squared error loss across sensors. While the model outperformed methods such as Vector Auto-Regression and k-Nearest Neighbors, the study did not compare its performance to other state-of-the-art models and differed in setting due to only working with binary signals rather than a heterogeneous mix. Furthermore, while the format of the signals is similar to the control signals in the SMAP/MSL dataset, their meanings differ: the signals in the former indicates state while those in the latter potentially modulate state.

2.3 Formalizing an anomaly detection pipeline

We have seen the wide variety of approaches to anomaly detection, as well as the range of settings AD is deployed. In most cases, the data cannot be modeled in its raw form and must go through several pre-processing steps before it is ready for modeling and scoring. As such, it can be more apt to describe anomaly detection as a sequential workflow composed of independent pre-processing, modeling, and post-processing modules as opposed to a single transformation [19]. This is the key idea behind *Orion*[3], *Meissa*, and other libraries within Sintel[7], which enables users to construct end-to-end pipelines for time series tasks such as AD. Below we provide a more precise definition of the AD task and an overview of available primitives in *Orion*/*Meissa*.

2.3.1 Sintel

The methods for control signals in this thesis are built and tested within the Sintel[4] ecosystem. Sintel is a machine learning framework for end-to-end time series tasks. At the core of the framework is the concept of **primitives** and **pipelines**. Such a concept originated from the Machine Learning Blocks (*MLBlocks*) framework proposed by Collazo and Xue [20] [21], who wanted to create a unified API for accessing data science software tools which have otherwise been distributed across several libraries depending on the type of data being analyzed and the step of the data science process – for example, pandas[5] supports a variety of operations for pre-processing and visualization, but not modeling. On the other hand, the `scikit-learn`[6] and *xgboost*[7] libraries provide modeling options but lack a comprehensive suite of pre-processing functions. Such specialization of libraries is not undesirable, but drawing from various libraries throughout the data science process can be confusing and

[3] https://github.com/sintel-dev/Orion
[4] https://sintel.dev/
[5] https://pandas.pydata.org/docs/index.html
[6] https://scikit-learn.org/stable/
[7] https://xgboost.readthedocs.io/en/stable/

primitive	fit method	produce method
`keras.Sequential.LSTMTimeSeriesRegressor`	`fit`	`predict`
`sklearn.preprocessing.StandardScaler`	`fit`	`transform`
`statsmodels.tsa.arima.model.ARIMA`	—	`predict`

Table 2.1: Examples of primitives and their corresponding `fit` and `produce` methods as an `MLBLocks` object.

time-consuming for users due to differences in interfaces. Collazo shows that every data problem can be mapped into the *MLBlocks* representation, from generative modeling techniques such as Hidden Markov Models to clustering. The *MLBlocks* framework later became accessible as an open-source library through the work of Smith et al. [19] and has since provided a foundation for a variety of data science tools including the time series libraries in Sintel.

At a high level, an `MLBlock` object acts as an engine for a *primitive* function or class such that users can run the primitive, tune parameters, and access metadata through the same set of methods. The functionality of each primitive is defined through its `fit` and `produce` methods; these are the methods that the user calls when running the block on a piece of data. For example, the primitive `sklearn.preprocessing.StandardScaler` which is a class from *scikit-learn* would set `fit` as the primitive's `fit` method and `transform` as the `produce` method. If a primitive does not have a relevant `fit` method, not specifying it would produce no operations (a no-op). An example list of primitives and their corresponding `fit`/`produce` methods can found in Table 2.1, and more details can be found in the *MLBlocks* documentation[8].

Many libraries in Sintel have primitives ready-to-use for a variety of time series tasks. *ml-stars*[9] is a library containing general purpose primitives for processing time series. *Orion* and *Meissa* have their own primitives that act as modular building blocks encapsulating essential functionalities for anomaly detection tasks. These operations span across all steps

[8] https://mlbazaar.github.io/MLBlocks/index.html
[9] https://pypi.org/project/ml-stars/

Pipeline	Citation
AER	[15]
ARIMA	[12]
Azure	[22]
Dense auto-encoder/LSTM auto-encoder	[23]
GANF	[24]
LNN	[25]
LSTM Dynamic Thresholding	[1]
Matrix Profile	[26]
TadGAN	[18]
VAE	[17]

Table 2.2: List of *Orion* pipelines and relevant citations

of the anomaly detection process, including data transformation during the pre-processing phase, training and prediction during the modeling phase, and error calculation during the post-processing phase.

Users can compose primitives into a **pipeline** tailored to their specific anomaly detection task, orchestrating the flow of data through a series of interconnected operations. A pipeline is simply a sequence of primitives (or blocks) strung together to provide a single interface for executing a sequence of operations. Calling methods such as `fit` and `detect` on the pipeline iteratively performs the operations through each step of the pipeline such that the output of the first block is fed in as the input of the second block, and so on.

Besides making tasks such as anomaly detection an accessible as an end-to-end pipeline, Sintel also provides a framework for benchmarking pipelines and for customizing workflows through its API and annotation function. Several anomaly detection pipelines using state-of-the-art models are readily accessible via the *Orion* library, allowing users to test the pipeline on their own data in both supervised and unsupervised settings; see Table 2.2. Further information can be found in Chapter 4 and [7].

While the methods outlined in this thesis are in the *Meissa* library, they build on the existing functionality of *Orion* and follow the same framework of primitives and pipelines outlined by Sintel. The goal of *Meissa* is to offer specialized support for working with both

Variable	Definition
$\mathbf{x}$	input time series
$\hat{\mathbf{x}}$	reconstructed time series
y_t	model prediction at time t
A	anomaly intervals
t	timestamp
n	number of timestamp observations in signal
d	number of channels in signal
n_{obs}	number of observation signal channels
k	aggregation interval
s	rolling window step size
w_r	rolling window sequence size
w_e	reconstruction error window size
T	reconstruction error threshold

Table 2.3: Notation table for variables in Chapter 2

multivariate and heterogeneous signals while allowing users to continue accessing the *Orion* modules. For the purposes of this thesis, we define a signal as *heterogeneous* if it contains a mix of observation and control signals, and we define a signal as *multivariate* if it contains more than one channel.

2.3.2 Data Representation

Here we more precisely define the inputs and outputs of the AD pipeline. Table 2.3 provides a summary.

Signal

Let us define the input to the anomaly detection pipeline as a multivariate time series $\mathbf{x} \coloneqq [x_0, x_1, \ldots, x_n]$ with $x_t \in \mathbb{R}^d$, where n is the number of observations and d is the number of channels. If the signal is heterogeneous (containing both observation and control signals), we can specify the first n_{obs} channels as *observation* signals taking on a continuous range of values, and the remaining $d - n_{obs}$ channels as binary *control* signals taking on values in $\{0, 1\}$. For the dataset tested on in this thesis, $n_{obs} = 1$ while the number of control

signals is $25 - n_{obs} = 24$ for SMAP and $55 - n_{obs} = 54$ for MSL. In reality, it is possible to have several channels to perform anomaly detection on; for example, the Server Machine Dataset contains multi-channel sensing data from 28 machines in a large Internet company measuring CPU load, network usage, and memory usage [27].

Sintel's data processing and ML pipelines such as *Orion* expect data in the form of a 2-dimensional pandas[10] DataFrame object where rows denote observations at a certain time step and columns denote signal channels. Additionally, the data should contain a `timestamp` column to indicate the order of the observations. Figure 2.1 shows an example multivariate signal from the MSL dataset loaded using *Orion's* `load_signal` method. In this case (and in the case of all signals from SMAP/MSL), the first channel after `timestamp` is observation while the remaining 54 channels are control.

	timestamp	obs_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	...	c_45	c_46	c_47	c_48	c_49	c_50	c_51	c_52	c_53	c_54
0	1222819200	-0.748738	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	1222840800	-0.748738	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	1222862400	-0.748738	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	1222884000	-0.748738	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	1222905600	-0.748738	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 2.1: Example multivariate signal 'M-2' from the MSL Dataset loaded with *Orion's* `load_signal` utility function. The first signal column `obs_0` is the observational channel, while the remaining are control.

Anomalies

We follow the definition of an anomaly detection pipeline provided by [7]. That is, a pipeline takes in a signal $\mathbf{x}$ as described above and outputs an array of intervals

$$A = [(t_s^1, t_e^1), (t_s^2, t_e^2), \ldots, (t_s^l, t_s^l)]$$

where t_s, t_e are respectively the start and end timestamps of the anomalous interval. Each anomaly can also include a *score* to indicate the severity of the anomaly, e.g. *Orion*'s

[10] https://pandas.pydata.org/

	channel	start	end	severity
0	0	1274421600	1278439200	0.767445
1	0	1280016000	1284163200	0.749520
2	0	1294228800	1319673600	1.125687
3	1	1270512000	1271678400	3.396181
4	2	1270512000	1271678400	3.396181

	start	end
0	1294488000	1319112000

Table 2.4: Example representation of detected anomalies (left) vs. ground-truth anomalies (right) from the test set of signal 'M-2' from the MSL dataset.

`find_anomalies` function computes the severity as the magnitude of the error relative to surrounding errors. Note that the input signal can be univariate as is often the case; this is just a special case of $\mathbf{x} \in \mathbb{R}^{n \times d}$ where the number of channels d is equal to 1.

In the multivariate case, anomalies may be computed for each observation channel and further filtered or aggregated. If the signal contains both observation and control signals, the magnitude of the errors in the control signal can also be used to inform the severity of the anomaly, the confidence in whether an interval is an anomaly, or both; more on this in Chapters 4 and 5. Figure 2.4 shows an example output of detected anomalies on the example data used in Figure 2.1 from the *Meissa* `MixedLSTM` pipeline.

2.3.3 Pre-processing

Before time series data can undergo analysis or be used as input in a deep learning model, several pre-processing steps need to happen:

- **Aggregation**: Often performed to adjust the temporal resolution of the data, this step involves grouping the raw time series data into larger time intervals, such as hours, days, or weeks, by computing summary statistics like mean, median, or sum within each interval. Aggregation can make unevenly sampled data evenly spaced and in general it can reduce data complexity and noise while preserving important temporal patterns. Specifically, given an aggregation interval of k timesteps and aggregation function $f : \mathbb{R}^{k \times d} \to \mathbb{R}^{1 \times d}$, we convert a time series $x_0, x_1, \ldots, x_n$ into the aggregated

version

$$f(x_{0:k-1}), f(x_{k:2k-1}), \ldots, f(x_{n-k-1:n})$$

where $x_{s:e} \coloneqq [x_s, x_{s+1}, \ldots, x_e]$.

- **Scaling**: Scaling is applied to normalize the values of the time series features since most models are sensitive to the magnitude of the input values. Common techniques include min-max scaling, where values are scaled to a range such as $[0, 1]$ or $[-1, 1]$, and standardization (Z-score normalization), where values are scaled to have a mean of 0 and a standard deviation of 1.

- **Imputation**: If the time series contains missing values, which can arise due to various reasons such as sensor failures or data collection issues, imputation can be applied. Techniques range from interpolation with a parameters like the mean to model imputation.

- **Window sequences**: Finally, we create window sequences from the data to make it suitable for training models. This involves sliding a fixed-size window over the time series data and extracting sequences of data points as inputs, along with corresponding target values. For instance, a rolling window of size w_r time steps and a step size of 1 would create $n - w_r$ sub-sequences where

$$\mathbf{x}_i = \{x_i, x_{i+1}, \ldots, x_{i+w_r-1}\}$$

for $i = 0, 1, \ldots, n - w_r$, the first observation in the given window [15]. For step sizes $s > 1$, the window would shift such that $i = 0, s, 2s, \ldots$. The rolling window sequences for the model input vs. the target differs depending on the anomaly detection setting. In prediction-based AD, the target window would be shifted forward by some number of time steps so the model to learn to predict future values, whereas the target window sequences would equal the input window sequences in reconstruction-based AD. More

detail about different AD approaches is covered in section 2.2.

2.3.4 Modeling

The exact input and output of the model depends on the signal dimensions, the model architecture, and the particular AD approach (e.g. prediction vs. reconstruction), as highlighted in Section 2.2.2. In general, a single input to the model is a window sequence of w_r observations $\mathbf{x}_i$. In *Orion* and *Meissa*, $w_r = 100$ for reconstruction-based models $w_r = 250$ for prediction-based models. For this thesis, we expect the model to output a corresponding sequence of the same number of timesteps. That is, given input $\mathbf{x}_i = \{x_i, x_{i+1}, \ldots, x_{i+w_r-1}\}$, the model outputs a one-step forecast y_{i+1} in prediction-based AD and an expected subsequence $\hat{\mathbf{x}}_i = \{\hat{x}_i, \hat{x}_{i+1}, \ldots, \hat{x}_{i+w_r-1}\}$ in reconstruction-based AD.

The number of channels output by the model depends on what channels are set as the target channel. In many cases, only one target channel is specified, regardless of the number of channels in the input. That is, each $\hat{x}/y_{i+1}$ can be anywhere between 1 and d channels. We start off Chapter 3 by running a many-to-one version of two LSTM-based pipelines on the SMAP and MSL datasets. However, the proposed pipeline in *Meissa* reconstructs the entire output to allow for additional analysis of the control signal channels.

2.3.5 Post-processing

Given the model's predicted or reconstructed sequences, we need to convert the *expected* signal – that is, what was expected by the model – into anomalous intervals. This is accomplished through post-processing methods, which can be broadly decomposed into aggregation, error computation, and anomaly scoring:

- **Aggregation**: Since the rolling window sequence inputs overlap with each other, there will be multiple outputs for a given time step of the signal in the reconstruction case. To convert these overlapping output sequences into a single time series of the original

input dimensions, we can aggregate the predictions at each time step by applying a function such as the mean or median and using that as the representative value of the time step. By default, we take the median.

- **Error computation**: Given the expected output of the model and the observed time series, we want to compute the discrepancy between the two. In prediction cases, we can simply take the *absolute error* between each forward prediction and the corresponding original point in the time series:

$$error = \sum_{i=w_r+1}^{n} |\hat{x}_i - x_i|$$

where errors are computed starting from time step $w_r + 1$ since that is the index of the first forward prediction. In the reconstruction case, a variety of methods can be employed since entire sub-sequences are reconstructed by the model. In addition to the absolute error, common approaches include *area differencing*, where for each subsequence $i = 0, \ldots, n - w$ we compute the total difference in that region

$$error = \frac{1}{2l} \left| \int_{i=1}^{i+1} \hat{x}_i - x_i dt \right|$$

where $2l$ is the length of the curve calculated using the trapezoidal rule. Yet another popular metric is the dynamic time warping (DTW) distance [28] which finds the optimal many-to-many mapping between the expected and actual output that minimizes the Euclidean distance between aligned series under all possible alignments:

$$error = \min_{\pi \in \mathcal{A}} \left(\sum_{(i,j) \in \pi} d(x_i, \hat{x}_j)^q \right)^{\frac{1}{q}}$$

where π is an alignment path consisting of index pairs (i, j) outlining a possible way to map points in x to points in $\hat{x}$, and d is a distance metric with power q (squared

error in the case of `pyts.metrics.dtw`.

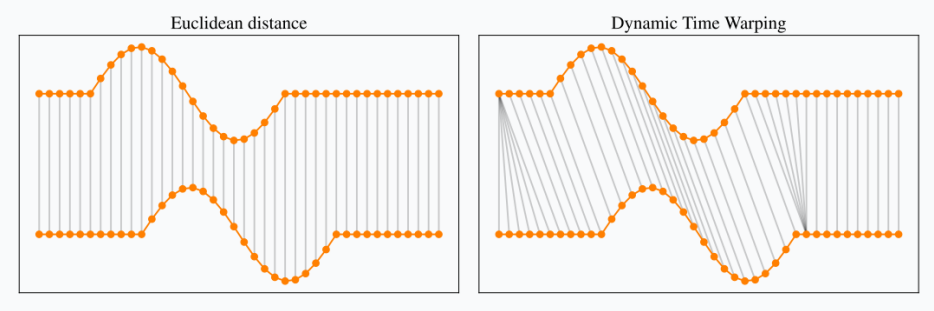

Figure 2.2: Diagram comparing Euclidean distance to DTW distance from Romain Tavenard https://rtavenar.github.io/blog/dtw.html

- **Anomaly scoring**: Given an array of errors, we wish to compute a threshold over which errors over the threshold are candidate anomalies. A threshold T is computed over each window of w_e errors, where the window size can be varied as a parameter. In our case, we opt for a fixed threshold that is computed from taking the value z standard deviations above the mean error. After finding the points where the error is above T, the anomalies are pruned to reduce the incidence of false positives. In Orion's `find_anomalies` primitive, this is done by computing the percentage increase between an error and the next highest error and removing the ones that are below a minimum threshold, which can also be set by the user.

Chapter 3

Incorporating control signals in anomaly detection

This chapter outlines the steps taken to construct an anomaly detection framework for handling control signals by looking at changes to consider at the pre-processing, modeling, and post-processing stages. It also describes the dataset and benchmarking process used, both of which are accessible through *Orion*.

3.1 Datasets

The pipeline is evaluated on real telemetry data derived from NASA's Incident Surprise Anomaly reports. Specifically, the project works with data from the Mars Science Laboratory (MSL) rover and the Soil Moisture Active Passive (SMAP) satellite. In both datasets, commands to and from various device modules at each timestep are encoded as binary control signals for a given channel. Analysis of such signals will be incorporated into the anomaly detection pipeline. Processing information in such reports is crucial for engineers to understand causes of unexpected events during the mission and better anticipate future failures.

In both the SMAP and MSL dataset, the signals consist of one observation channel

with continuous values ranging from -1 to 1, with the rest being binary control signals in $\{0, 1\}$. The number of channels is 25 and 55 for SMAP and MSL, respectively. The observations in each signal are all equally spaced into 6-hour intervals. Thus, despite the fact that `time_segments_aggregate`, `MinMaxScaler`, and `SimpleImputer` are part of the the LSTM pipelines, no aggregation, scaling, or imputation was needed for this data.

3.2 Naive approach

To start, we looked at how well the `LSTM Dynamic Threshold` (LSTM-DT) and `LSTM Auto-encoder` (LSTM-AE) pipelines in *Orion* handle multivariate signals naively. By default (and in the *Orion* benchmark), the pipelines fit and predict on the univariate versions of the SMAP/MSL dataset, which contains only a timestamp column and the continuous observation signal column. If we pass in the multivariate version of the signals as input, all columns undergo the pre-processing steps with the same parameters and are used to train the model, but the model output is only the observation column.

The `LSTM Dynamic Threshold` pipeline consists of the following primitives:

```
"primitives": [
    "mlstars.custom.timeseries_preprocessing.time_segments_aggregate",
    "sklearn.impute.SimpleImputer",
    "sklearn.preprocessing.MinMaxScaler",
    "mlstars.custom.timeseries_preprocessing.rolling_window_sequences",
    "keras.Sequential.LSTMTimeSeriesRegressor",
    "orion.primitives.timeseries_errors.regression_errors",
    "orion.primitives.timeseries_anomalies.find_anomalies"
],
```

where `LSTMTimeSeriesRegressor` consists of two hidden layers with 80 LSTM units each and dense layer that outputs a single vector of predicted values at each input timestep.

At first glance, the `LSTM Auto-encoder` pipeline seems to have mostly the same steps:

```
"primitives": [
```

```
    "mlstars.custom.timeseries_preprocessing.time_segments_aggregate",
    "sklearn.impute.SimpleImputer",
    "sklearn.preprocessing.MinMaxScaler",
    "mlstars.custom.timeseries_preprocessing.rolling_window_sequences",
    "orion.primitives.timeseries_preprocessing.slice_array_by_dims",
    "keras.Sequential.LSTMSeq2Seq",
    "orion.primitives.timeseries_errors.reconstruction_errors",
    "orion.primitives.timeseries_anomalies.find_anomalies"
],
```

but the approaches of the pipelines differ in several ways. First, as expected, the model primitive is an auto-encoder with LSTM units as opposed to the double stacked recurrent neural network architecture in LSTM-DT. This reflects the fact that `LSTM Auto-encoder` (LSTM-AE) is a reconstruction-based pipeline that first encodes the input data into a feature *vector* in the latent space through the first LSTM layer before decoding it through another LSTM layer. Like `LSTMTimeSeriesRegressor`, the final dense pooling layer condenses the hidden state into expected time series values for only the first column. The output of the auto-encoder is the what the model expects the time series to look like based on the latent features it extracted. On the other hand, `LSTMTimeSeriesRegressor` is trained to predict some number of timesteps ahead of the input window sequence.

The second major difference is in the error function used. LSTM-AE uses the dynamic time warping (DTW) distance between the reconstructed and original values over a pre-defined window of points. LSTM-DT computes the absolute errors between predicted and original points and applies Exponential Weighted Moving Average (EWMA) smoothing.

3.3 Explicitly handling control signals

In this section, we outline the methodology used to develop a new LSTM auto-encoder-based pipeline (`MixedLSTM`) in *Meissa* that handles multivariate and heterogeneous signals

explicitly. We namely describe the high level properties and differences from the LSTM-AE pipeline in *Orion*. Figure 3.1 shows the stages of the pipeline and the flow of data through pre-processing, modeling, and post-processing primitives.

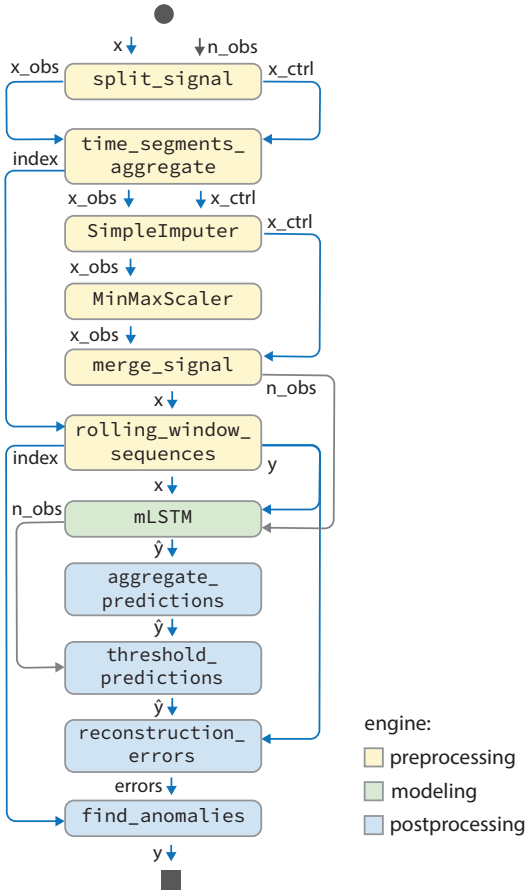

Figure 3.1: `MixedLSTM` pipeline diagram

3.3.1 Pre-processing

Given that control signals are discrete and binary in our setting, we modify a few of the pre-processing steps to account for this and describe some below. The notation used follows from Chapter 2; an overview of variable definitions can be found in Tables 2.3 and 3.1.

- **Aggregation**: The default aggregation method in *Orion* is to take the mean of the

Variable	Definition
$\mathbf{x}_i$	input window sequence starting at index i
$\hat{\mathbf{x}}_\mathbf{i}$	reconstructed window sequence starting at index i
$\mathbf{x}_{o,i}$	observation subset of input window sequence
$\hat{\mathbf{x}}_{o,i}$	observation subset of reconstructed window sequence
$\mathbf{x}_{c,i}$	control subset of input window sequence
$\hat{\mathbf{x}}_{c,i}$	control subset of reconstructed window sequence
b	batch size
p	output class probability
λ	loss ratio
α	positive class weight parameter
γ	focusing parameter

Table 3.1: Notation table for variables in Chapter 3

values in the aggregation interval. This works well for continuous signals, but taking the mean or median can lead to a non-integer value between 0 and 1 for control signals. Thus, the maximum, minimum, or mode would be more appropriate depending on how the data is distributed. In our case, we choose to use `max` as our aggregation function: $f(x_{t:t+k}) = \max(x_{t:t+k}), 0 \leq t \leq n-k$, where n is the signal length and k is the aggregation window size. This is motivated by the face that values of 1 are sparse in the control signal channels and thus are more likely to encode important information. Indeed, less than 2% of control signal values in SMAP and less than 1% of control signal values in the MSL dataset are positive (see Table 3.2). In general, the aggregation method just be chosen according to the properties of the dataset, which often require interpretation by domain-knowledge experts.

- **Imputation**: Similar to aggregation, we choose to only impute values of 0 or 1 for control signals. Taking the maximum, minimum, or mode would all fulfill this property; in our case, we choose to set the mode as the default value to impute. This is due to the face that values of 1 are statistically rare. Thus, imputing 0 in an interval with only 0s (no significant events) best represents the information given by that interval.

- **Scaling**: For pipelines handling heterogeneous signals in *Meissa*, we assume that the

	Avg Length	Avg Sparsity (%)
SMAP	10485 ±1604	1.24 ±0.90
MSL	4891 ±2035	0.53 ±0.35

Table 3.2: Summary statistics of the SMAP and MSL dataset. Sparsity is the percentage of control signal values that are equal to one.

control signals have already been processed to take on only values of 0 or 1. Thus, we opt to not include any scaling of the control signal channels in the pipeline. Instead, only the observation channels specified by the `num_obs` parameter is scaled to be between $[-1, 1]$. More detail on the user interface is provided in Chapter 4.

3.3.2 Mixed LSTM model

Adjusting the model architecture to allow for specialized handling of observation and control signals is a key part of the `MixedLSTM` pipeline. The model in the `MixedLSTM` pipeline, which we will hereon refer to as `mLSTM`, is a multivariate version of the LSTM auto-encoder model that optimizes two different losses, one for the observation signals and one for the control signals. Given a multivariate signal $\mathbf{x} \in \mathbb{R}^{n \times d}$ decomposed into rolling window sequences of size $w_r = 100$, the model reconstructs the input using two LSTM layers. Specifically, the dimension of a single input is $(100, 25)$ and $(100, 55)$ for SMAP and MSL, respectively.

Layers

The first LSTM layer maps the entire input with dimension $(100, d)$ into a latent space consisting of `lstm_1_units` features, which is 60 by default. The output of this layer is an encoded feature vector of the input of dimensions $(1, 60)$ that can be itself extracted to use as a compressed form of the original data. In our case, we want to map the latent vector back into the original input space to reconstruct the original signal. This is accomplished in the decoding stage. The second LSTM layer takes an input of dimension $(100, 60)$ and outputs a vector of dimensions $(100, 60)$. This output is then matrix-multiplied with a $(60, d)$

dimension dense layer replicated to match the number of features to get the final output. A schematic of the layers and dimensions can be found in Figure 3.2 below.

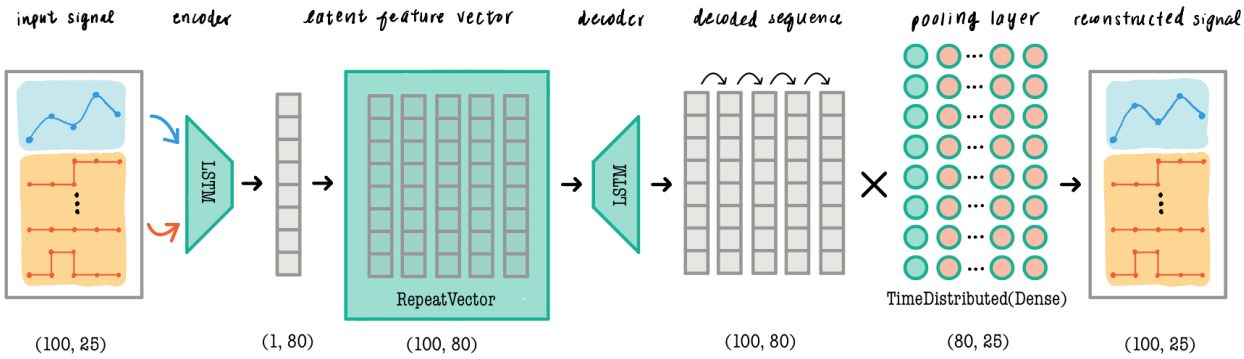

Figure 3.2: Illustration of the `mLSTM` model architecture with dimensions for a hypothetical input of $(100, 25)$ where $w = 100$ is the rolling window size, $d = 25$ is the number of features/channels, and the LSTM layers each have 80 units.

Loss function

The choice of the loss function is a crucial part of the model that distinguishes it from its original LSTM-AE counterpart. In the naive approach, we implicitly choose to treat control signals the way we treat observation signals; no distinction is given to them in modeling, as mean squared error (MSE) loss is used in both cases. That is, given a batch size of b containing w_r-sized window sequences of observations, we compute

$$MSE = \frac{1}{b}\sum_{i=1}^{b}(\mathbf{x}_i - \hat{\mathbf{x}}_i)^2$$

where $\mathbf{x}_i$ is an input sub-sequence and $\hat{\mathbf{x}}_i$ is the corresponding output reconstructed by the model. Below, we consider options for using a different loss function depending on the type of signal, as well as on the state of the signal. In the presence of both observation and control signals, the model optimizes a joint loss weighted by a loss ratio λ. We experiment with two configurations:

1. **MSE + BCE**: Compute the mean squared error for the n_{obs} observation signals and

the binary cross-entropy loss of the remaining $d - n_{obs}$ control signals. The binary cross-entropy loss (BCE) as implemented in Keras is

$$BCE = \sum_{i=1}^{b} -(\mathbf{x}_{c,i} \cdot \log \hat{\mathbf{x}}_{c,i} + (1 - \mathbf{x}_{c,i}) \cdot \log(1 - \hat{\mathbf{x}}_{c,i}))$$

where $\mathbf{x}_{c,i}$ is the input sub-sequence of w control signal values with $d - n_{obs}$ channels starting at index i, and $\hat{\mathbf{x}}_{c,i}$ is the corresponding output sequence from the model. Note that while $\mathbf{x}_{c,i} \in \{0, 1\}^{w_r \times (d - n_{obs})}$ since the input was assumed to be binary from the start, the model output takes on a real value between $[0, 1]$ as it represents the probability of being in the positive class. The two losses are then weighted by a parameter $\lambda \in [0, 1]$ that can be set by the user. Thus, the model optimizes the joint loss

$$L_{joint} = \lambda \cdot \frac{1}{b} \sum_{i=1}^{b} (\mathbf{x}_{o,i} - \hat{\mathbf{x}}_{o,i})^2 + (1 - \lambda) \sum_{i=1}^{b} -(\mathbf{x}_{c,i} \cdot \log \hat{\mathbf{x}}_{c,i} + (1 - \mathbf{x}_{c,i}) \cdot \log(1 - \hat{\mathbf{x}}_{c,i}))$$

where $\mathbf{x}_{o,i}, \hat{\mathbf{x}}_{o,i} \in \mathbb{R}^{w \times n_{obs}}$ is the observation subset and $\mathbf{x}_{c,i}, \hat{\mathbf{x}}_{c,i} \in \mathbb{R}^{w_r \times (d - n_{obs})}$ is the control subset.

2. **MSE + FL**: Similar to configuration (1), we optimize the MSE for the observation subset. For the control subset, we use the focal cross-entropy loss (FL) first proposed by Lin et al [29]. The FL equation is given by

$$FL = -\alpha(1 - p_t)^{\gamma} \log(p_t)$$

where p_t is shorthand for

$$p_t = \begin{cases} 1 & \text{if } p = 1 \\ 1 - p & \text{otherwise} \end{cases}$$

and α, γ are parameters representing the following:

(a) α is a weight-balancing factor for the positive class. It is helpful for control signals because the classes tend to be imbalanced. Without weighing positive classes more heavily, the model can end up only outputting 0.

(b) γ is the exponent of the focal parameter, which is p^γ if $y = 1$ and $(1-p)^\gamma$ otherwise. Higher values of gamma (> 1) down-weight "easy" examples (those that the model outputs with high probability). When $\gamma = 0$, the FL is just α-balanced cross entropy loss.

In this case, p is the sigmoid output of the auto-encoder for the control subset sigmoid($\hat{\mathbf{y}}_{c,i}$). In Keras, we do not need the sigmoid activation as it can directly compute the loss in terms of logits instead of probabilities. Thus $p_t \in \mathbb{R}^{w_r \times (d - n_{obs})}$ and is between $[0, 1]$. The final joint loss is

$$L_{joint} = \lambda \cdot \frac{1}{b} \sum_{i=1}^{b} (\mathbf{x}_{o,i} - \hat{\mathbf{x}}_{o,i})^2 - (1-\lambda)(\alpha)(1-p_t)^\gamma \log(p_t)$$

where λ is the loss ratio defined previously.

This approach accounts for the state of the control signal through the α parameter. That is, higher values of α allow us to value the positive class predictions more to mitigate the issues from heavy class imbalance. While γ does not depend on the control signal state, the imbalance in values of the control signal implicitly makes 0 predictions "easier"; that is, the model often predicts 0 with high confidence.

Hyperparameter tuning

Several parameters must be balanced for both configuration (1) and (2), including but not limited to α, γ and λ, as well as the number of LSTM units. The authors of [29] note that as γ increases, the optimal α decreases, reflecting the fact that as easy examples are down-weighted, less weight needs to be given to the positive class.

While the number of LSTM units is meant to be more of a fixed parameter of the

primitive/pipeline, we wanted to test a larger number of units as 60 was the default from the original studies for LSTM-DT and LSTM-AE models [1] which only took in univariate signals as input.

To get a crude idea of the parameters, we sampled a subset of signals from both SMAP and MSL and performed a grid search of combinations of α, γ, and the number of LSTM units. For λ, we found that $\lambda = 0.75$ was optimal for capturing the information in the observation signal, whereas $\lambda = 0.5$ placed too much emphasis on the control channels to be able to extract anomalies from the observation channel. More information is provided in Section 5.

3.3.3 Post-processing

As with the pre-processing stages, a few modifications were made to obtain the reconstruction of the control signals and interpret the anomalies detected in each column:

- **Thresholding**: The model computes losses using logits. Only after prediction do the control signal channels undergo a sigmoid activation to output a probability between 0 and 1. However, to reconstruct the binary output, a threshold needs to be determined to transform the probabilities. Since the ratio of 0s and 1s differ drastically from signal to signal, a fixed threshold would be too inflexible. Therefore, we opted to select the threshold by optimizing the ratio of the true positive rate (TPR) and false positive rate (FPR as given by the receiver operating characteristics (ROC) curve, which traces the TPR/FPR for every threshold between $[0, 1]$. This can be accomplished by maximizing the g-means of the ROC curve:

$$T = \text{argmax} \sqrt{\text{TPR} \cdot (1 - \text{FPR})}.$$

- **Anomaly scoring**: There are two ways of approaching evaluation of control signal reconstruction, and we talk about each approach more in Chapter 5:

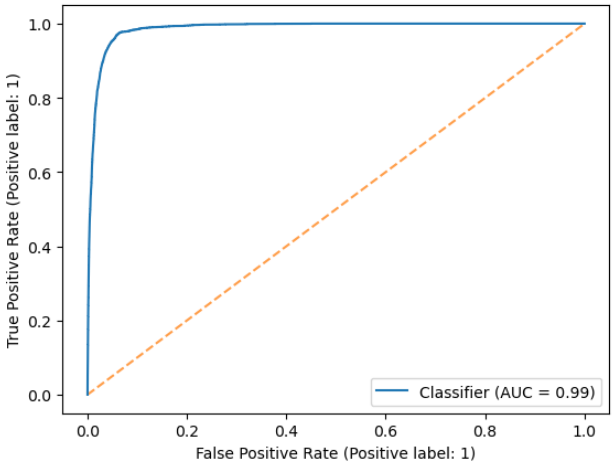

Figure 3.3: Sample ROC curve for binary threshold of signal 'E-10' in SMAP

1. We assume that the control signal contains no anomalies since its state comes from manual input such as commands. Rather, anomalies occur in the continuous observation signal. As such, the goal should be to reconstruct the non-noisy part of the control signal as well as possible so that it can give us accurate information about its relation to the observation signal. But ultimately, only anomalous intervals detected in the observation column should be considered.

2. Perform anomaly detection on all columns independently. Filter and aggregate the anomalies detected in each column by looking at the intervals detected by the most columns and/or with the highest severity levels.

Chapter 4

Meissa

One goal of the thesis is to not only uncover additional insights from explicitly modeling control signals, but to also provide a user-friendly interface for others to do the same. The work motivated the creation of a separate anomaly detection library, *Meissa*, which acts as an addition to *Orion*. *Meissa* extends the existing functionality of Orion by (a) adapting primitives and pipelines to work for multivariate signals and (b) allowing distinction between observation and control signals at each of the pre-processing, modeling, and post-processing stages.

4.1 User Interface

As is the case with *Orion*, the two main classes of objects in *Meissa* are *primitives* and *pipelines*, and new pipelines can be constructed from combining primitives. Furthermore, all pipelines and datasets available on *Orion* can be accessed through *Meissa*, as such imports are made from *Orion*. This allows for users to seamlessly switch between using existing univariate pipelines when they suit the task at hand and creating a more specialized version in *Meissa* if the user wishes to reconstruct multivariate outputs, or specify different parameters for control signals.

The primary difference between *Orion*'s interface and *Meissa's* is that users must specify

the number of observation signal columns (`num_obs`) either upon initialization of a pipeline, or when calling a primitive method that distinguishes between observation and control signals. The methods assume that the data is structured such that the first `num_obs` channels of the signal are observation, and those following are all control. If no control signal is specified (i.e. `num_obs` $= d$), then it treats all columns of the multivariate signal as continuous observation signals. That is, it applies the same parameters to all channels within each primitive and uses the same MSE loss as opposed to optimizing a joint loss.

4.2 Primitives

As mentioned previously, the primary changes in *Meissa* are from adapting *Orion* primitives to handle multivariate signals, or to allow users to specify different parameters for control vs. observation signals. We start by describing the steps of creating and calling a primitive in *Meissa* before providing examples of where such changes were made, as well as the primitives that are unique to *Meissa*.

4.2.1 Creating and calling primitives

Both *Orion* and *Meissa* primitives are built on top of `MLBlock` instances, which is initialized by parsing a JSON *annotation* file describing the source library of the method to import from, the relevant arguments and hyperparameters, and the corresponding `fit` and `produce` methods as described in Section 2.3.1. For example, the JSON annotation for the `split_signal` primitive in Figure 4.1 contains a `produce` field specifying the names and types of the input arguments as well as outputs for the primitive's `produce` method. In this case, there is no `fit` method. The `hyperparameters` entry specifies the names, types, and default values of each hyperparameter. The names of the inputs and arguments are more important when building pipelines, since the object being passed through a given stage of the pipeline is identified through its name in the environment.

```
{
    "name": "meissa.primitives.timeseries_preprocessing.split_signal",
    "contributors": [
        "Grace Song <gysong@mit.edu>"
    ],
    "description": "Splits signal into observation and control subsets.",
    "classifiers": {
        "type": "preprocessor",
        "subtype": "transformer"
    },
    "modalities": [
        "timeseries"
    ],
    "primitive": "meissa.primitives.timeseries_preprocessing.split_signal",
    "produce": {
        "args": [
            {
                "name": "X",
                "type": ["ndarray", "pd.DataFrame"]
            }
        ],
        "output": [
            {
                "name": "X_obs",
                "type": "pd.DataFrame"
            },
            {
                "name": "X_ctrl",
                "type": "pd.DataFrame"
            }
        ]
    },
    "hyperparameters": {
        "fixed": {
            "time_column": {
                "type": "str or int",
                "default": "timestamp"
            },
            "num_obs": {
                "type": "int",
                "default": 1
            }
        }
    }
}
```

Figure 4.1: Example JSON annotation for the `split_signal` primitive in *Meissa*

```
from mlstars import load_primitive
from orion.data import load_signal

signal = load_signal('multivariate/M-2')
primitive = load_primitive("meissa.primitives.timeseries_preprocessing.split_signal",
                           arguments={"time_column": "timestamp", "num_obs": 1})
signal_obs, signal_ctrl = primitive.produce(signal)
```

Figure 4.2: Example usage of `split_signal` on 'M-2' signal

To call the primitive, we simply load the primitive with the *ml-stars*[1] `load_primitive` method and then call `primitive.fit(signal)` on input DataFrame `signal`, as seen in Figure 4.2.

4.2.2 Changes to Orion primitives

The `time_segments_aggregate`, `reconstruction_errors`, and `find_anomalies` primitives were widely used in *Orion*, but they were not able to directly work on multivariate signals. Thus, an initial step was to adapt such functions to work with a variable number of channels.

- Handling 3D arrays: For primitives such as `reconstruction_errors` that did not involve operations unique to control signals, the main task was to efficiently apply the intermediate operations to a 3-dimensional numpy array, whereas functions like `time_segments_aggregate` were further adapted to also perform different operations for observation vs. control signals.

- Added general functionality: The `rolling_window_sequences` primitive was also adapted to not only take in multivariate signals and output, but also to produce different input and target sequences for the model depending on whether the setting specified was reconstruction or prediction.

- Anomaly scoring for multiple columns: The `find_anomalies` primitive in *Meissa* was unique in the sense that even for a multivariate signal, the operations were applied

[1] https://pypi.org/project/ml-stars/

independently for each channel such that anomalies were scored only using the error values in that particular channel. This allows for users only examining observation signals for anomalies to see results without interference from other channels.

4.2.3 New primitives

To support the functionality of a heterogeneous and multivariate signal pipeline, a handful of new primitives were created:

- `split_signal` / `merge_signal`: As the name suggests, `split_signal` split a signal into observation and control subsets using the provided `num_obs` parameter. On the other hand, `merge_signal` performs the reverse operation by concatenating the observation and control subsets. In the pipeline, `split_signal` is called first to enable easy handling of observation-specific operations such as specifying the mean for aggregation and imputation, and scaling to between $[-1, 1]$. `merge_signal` is called before `rolling_window_sequences` since it does not apply special operations to control signals and allows for the model primitive in the next stage to work with a single input. When no control signals are specified, the observation subset returned by `split_signal` is an empty pandas DataFrame.

- `aggregate_predictions`: This primitive aggregates the reconstructed time series such that only one prediction remains at each timestep, as described in Section 2.3.5. In *Orion*, this operation was part of `reconstruction_errors`. However, with heterogeneous signals it was convenient to perform the aggregation before computing errors so that the control signal channel predictions can be thresholded using the aggregated values.

- `threshold_predictions`: As mentioned in Section 3.4.3, thresholding was needed to convert the outputted probabiltiies into binary values again. This primitive returns a threshold found from maximizing the G-means of the TPR/FPR ratio as described.

Alternatively, the user can supply a fixed threshold to apply to all control signal channels.

4.3 Pipelines

4.3.1 Creating and calling pipelines

The modularity of `MLBlock` instances allows them to be composed into an end-to-end pipeline suited for a particular data analysis task. Similar to primitives, pipelines in Meissa are special instances of the `MLPipeline` class. To create a new pipeline, the user must specify the following, typically as a JSON annotation:

- `primitives`: a list of the names of primitives composing the pipeline
- `init_params`: a dictionary containing, for each primitive, a mapping of arguments and their initialization values.
- `input_names`: a dictionary containing, for each primitive, a mapping of arguments with the actual name of the input that the primitive should expect. This allows users to pass the same variable to multiple primitives, even if the primitives name the argument differently.
- `output_names`: a dictionary containing, for each primitive, a mapping of outputs with the name that it should be renamed to.

The `Meissa` class allows users to directly initialize an `MLPipeline` instance instead of through *MLBlocks* and has attributes and methods specific to anomaly detection applications. To initialize an instance of a pipeline in *Meissa*, the user can call `pipeline = Meissa()` with the following arguments:

- `pipeline`: a string denoting the name of the `MLPipeline` instance to import, a dictionary representing the JSON annotation of the pipeline as described above, or an

`MLPipeline` instance.

- `num_obs`: the number of observation signal channels in the data. Assumes that the channels are ordered such that the first `num_obs` channels of the signal are observation, and the remaining are control.

- `hyperparameters`: a dictionary of hyperparameters for each primitive of the pipeline. Should contain, for each primitive, a mapping of hyperparameter names to the desired value. If a hyperparameter value is not specified, it uses the default value specified in the JSON annotation of the primitive.

Similar to `MLPipeline`, calling the pipeline's `fit` method will iteratively call the fit method of all the primitives. In an ML-based AD setting, this would also initiate the model training process. After fitting the pipeline, one calls `detect` – this iteratively calls the `produce` methods of all the primitives in the pipeline, which could include generating the model's prediction or reconstruction of the signal, as well as computing errors and the detected anomalies.

Additionally, a `Meissa` pipeline has a `fit_detect` method that performs the pipeline fitting and anomaly detection at once, as well as an `evaluate` method that calls the `produce` methods and scores the detected anomalies against a provided ground truth set of anomalies. Table 4.1 summarizes the `Meissa` pipeline interface, and Figure 4.3 shows an example of initializing the `MixedLSTM` pipeline and fitting it on a signal from the SMAP dataset. Note that in the code, `mixed_lstm` is the name of the pipeline in the library, while `meissa.primitives.mixed.MixedLSTM` is the path to the `mLSTM` model.

From the perspective of the user, the process of creating a pipeline in *Meissa* is the same as in *Orion* except for the `num_obs` initialization parameter. However, there are a few differences under the hood depending on the primitives used. If the user specifies different parameters for control signals vs. observation signals (e.g. using `"max"` for the former and `"mean"` for the latter), the `split_signal` primitive divides the input signal into its observation and

Method	Parameters	Returns
`fit`	`data` (pd.DataFrame)	—
`detect`	`data` (pd.DataFrame) `visualization` (bool)	pd.DataFrame of detected anomalies
`fit_detect`	`data` (pd.DataFrame) `visualization` (bool)	pd.DataFrame of detected anomalies
`evaluate`	`data` (pd.DataFrame) `ground_truth` (pd.DataFrame) `fit` (bool) `train_data` (pd.DataFrame) `metrics` (list)	pd.Series of scores of the detected anomalies against ground truth anomalies

Table 4.1: Summary of `Meissa` pipeline methods

control subsets using `num_obs` and applies the subsequent primitives separately to them. Before passing into a model, the signal needs to be re-combined into one input; this is done using the `merge_signal` primitive.

The below describes a typical sequence for working with a heterogeneous, multivariate signal in *Meissa*:

1. User provides input signal and `num_obs`, the number of observation columns
2. Split input into observation and control subsets and apply respective aggregation and imputation methods. Scale observation signal to a desired range, if needed
3. Merge observation and control signals and generate rolling window sequences
4. Fit model on data and predict or reconstruct the test set
5. Determine a binary threshold for control signal columns either from manual input or from optimizing FPR/TPR rate
6. Apply threshold and compute errors/anomalies as normal

```python
from meissa import Meissa
from orion.data import load_signal, load_anomalies

signal = load_signal("multivariate/M-2")
ground_truth = load_anomalies("M-2")

meissa = Meissa("mixed_lstm",
                num_obs=1,
                hyperparameters={
                    "meissa.primitives.timeseries_anomalies.find_anomalies#1": {'fixed_threshold': False},
                    "meissa.primitives.mixed.MixedLSTM#1": {
                        'focal_alpha': 0.25,
                        'focal_gamma': 2
                        }
                })

# fit primitives
meissa.fit(signal)

# reconstruct signal and find anomalies
anomalies = meissa.detect(signal)

# score anomalies; note that evaluate() calls detect() again
scores = meissa.evaluate(signal, ground_truth, fit=False)
```

Figure 4.3: Example of initializing the `MixedLSTM` pipeline through *Meissa* and calling methods

4.3.2 MixedLSTM Pipeline

As the name suggests, the `MixedLSTM` pipeline enables end-to-end reconstruction-based AD built around the `mLSTM` model. Below are the primitives of the pipeline; much of the workflow follows that of the `LSTM auto-encoder` pipeline. The difference is that different parameters can be specified for control vs. observation signals for primitives such as `time_segments_aggregate`, and `split_signal`/`merge_signal` is called to streamline this functionality.

```
"primitives": [
    "meissa.primitives.timeseries_preprocessing.split_signal",
    "meissa.primitives.timeseries_preprocessing.time_segments_aggregate",
    "sklearn.impute.SimpleImputer",
    "sklearn.impute.SimpleImputer",
    "sklearn.preprocessing.MinMaxScaler",
    "meissa.primitives.timeseries_preprocessing.merge_signal",
    "meissa.primitives.timeseries_preprocessing.rolling_window_sequences",
```

```
    "meissa.primitives.mixed.MixedLSTM",
    "meissa.primitives.timeseries_errors.aggregate_predictions",
    "meissa.primitives.timeseries_errors.threshold_predictions",
    "meissa.primitives.timeseries_errors.reconstruction_errors",
    "meissa.primitives.timeseries_anomalies.find_anomalies"
]
```

Figures 4.4 and 4.5 walk through how input data is changed after going through each primitive in the `MixedLSTM` pipeline. Note that in the illustrations, the examples do not necessarily follow each other; for instance, the `SimpleImputer` example provides a version of the signal that has missing values to show the functionality of the primitive. We'd also like to highlight how primitives in the pipeline can be specified to apply:

1. the same transformations to both observation and control signals;
2. different transformations to observation and control signals; and
3. transformations to only observation signals or only control signals.

For instance, `MinMaxScaler` is only applied to the observation signal channels in `MixedLSTM` since we assume that the control signals only take on values of $\{0, 1\}$ by default, and `threshold_predictions` is only applied to the control signals to convert the continuous probabilities into the same binary values. Such flexibility is achieved from calling `split_signal` to separate the original input and calling `merge_signal` when it is convenient for applying the same operation to all channels.

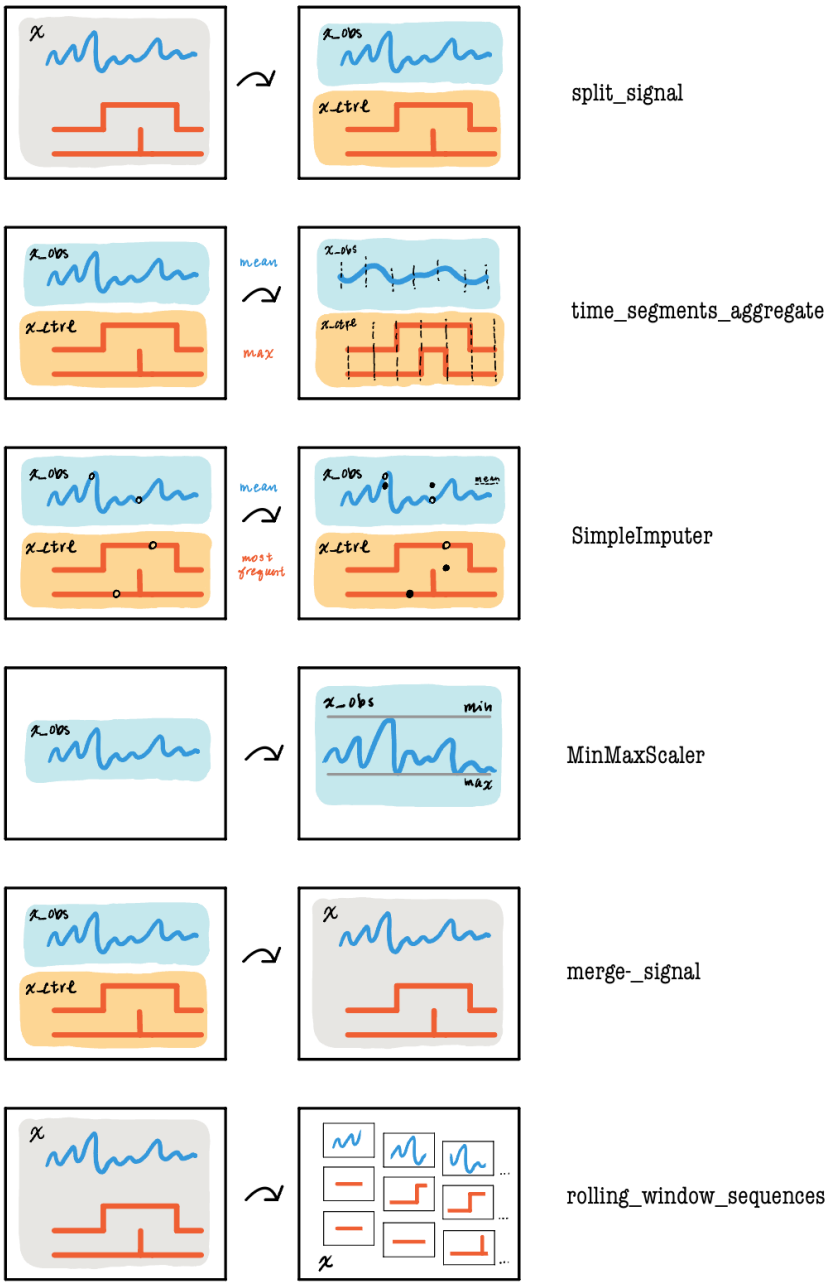

Figure 4.4: Illustration of transformations made by each primitive in the pre-processing stage.

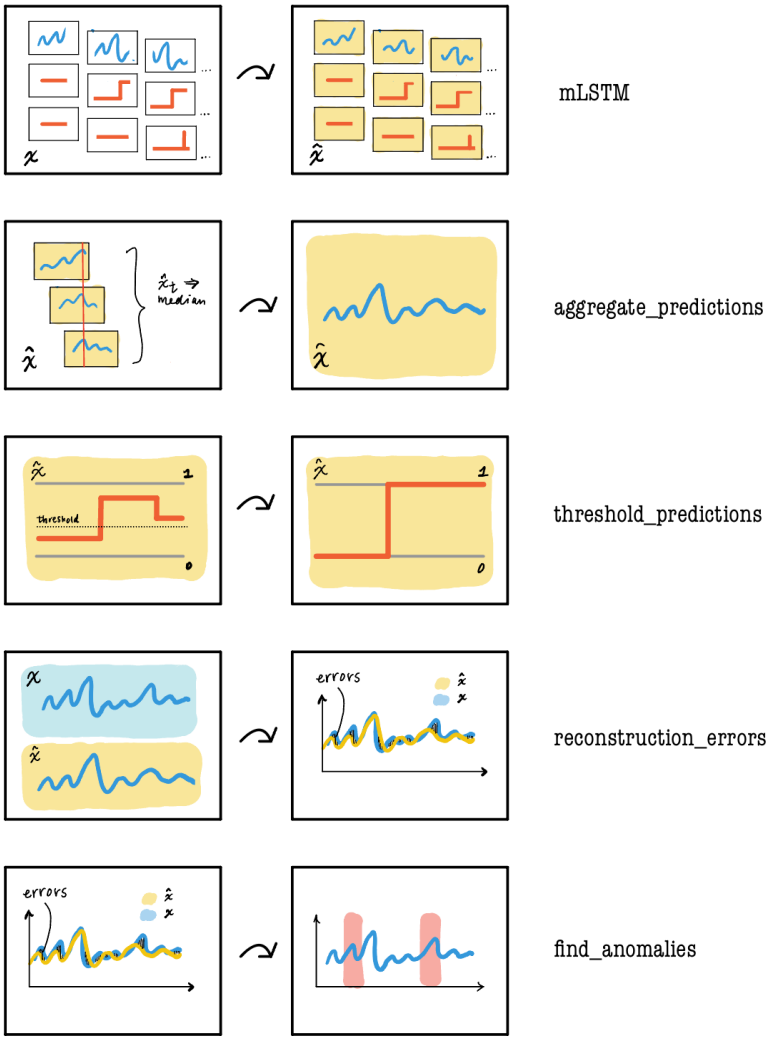

Figure 4.5: Illustration of transformations made by each primitive in the modeling and post-processing stage.

Chapter 5

Results

5.1 Benchmarking

To gauge the performance of a new pipeline, we use the benchmark framework available in *Orion*. The benchmark functionality is one of the core contributions of Sintel, as it provides an easy interface for evaluating pipelines and comparing them on even ground. The *Orion* benchmark API measures the quality of the detected anomalies using standard metrics such as precision, recall, and F1 score, but also define weighted and overlapping segment score, which are more informative for AD tasks [7]. Additionally, information about the computational cost of training a pipeline and predicting/reconstructing a given signal is provided to help assess deployability and find bottlenecks in the pipeline.

Following the dominant approach for assessing AD pipelines, the benchmark function assesses the naive and mixed pipelines by training on a non-anomalous subset of each signal, and predicting on the held out subset containing anomalies. While this is a standard approach, it is still not practical in settings such as online environments where data is expected to be analyzed as they come in, as well as where labeled data is not available within a reasonable timeframe. As such, we also test the Mixed LSTM pipeline in a completely unsupervised case where model trains and predicts on the entire signal. As for scoring, we use

an unweighted approach where an anomalous segment detected by the model is considered a true positive if it overlaps with any part of the true anomalous segment.

5.2 Naive Approach

The first step was to get an idea of the baseline performance of two stndard pipelines when taking in both observed and control signals. To start, we ran the LSTM-DT and LSTM-AE pipeline on all multivariate SMAP and MSL datasets since it the architecture used originally by Hundman et al [1]. From hereon out, we will refer to these configurations as *naive multivariate* versions of the two pipelines, while the official benchmark results from *Orion* are from the *univariate* version of the pipelines. All training and scoring was done in *Orion*.

We found that anomaly detection scores from the naive multivariate and univariate pipelines were similar for both LSTM-DT and LSTM-AE, although there were significant differences between the two datasets. Predicting on multivariate data for LSTM-DT produced higher precision for MSL data, but less for the SMAP data, suggesting that when predicting on multivariate data, the model may be more sensitive to hyperparameter changes in terms of the number of false positives detected. As we will see in the following sections, the number of false positives grows significantly when all channels are modeled and analyzed.

Table 5.1: Benchmark results of LSTM-DT on multivariate (left) vs. univariate (right) satellite data

	Precision	**Recall**	**F1**
MSL	0.407	0.667	0.503
SMAP	0.546	0.791	0.646

	Precision	**Recall**	**F1**
MSL	0.373	0.694	0.485
SMAP	0.650	0.776	0.707

The relative performance between multivariate and univariate inputs for LSTM-AE is similar to that of LSTM-DT: precision of the multivariate case was also higher for MSL but lower for SMAP data.

Table 5.2: Benchmark results of LSTM-AE on multivariate (left) vs. univariate (right) satellite data

	Precision	Recall	F1
MSL	0.514	0.528	0.521
SMAP	0.625	0.746	0.680

	Precision	Recall	F1
MSL	0.373	0.694	0.485
SMAP	0.650	0.776	0.707

5.3 Mixed approach

As described in Chapter 3, we propose a mixed LSTM auotencoder model `mLSTM` that optimizes separate loss functions for conrol vs. observation signals. Below we report the process for selecting the loss function for control signals and tuning parameters for benchmarking.

5.3.1 Choice of loss function

Binary cross-entropy

One potential source of variation in the `mLSTM` model is the choice of loss function for the control signal channels. A simple and intuitive option is to use binary cross-entropy (BCE) loss since the control signal values are in $\{0, 1\}$. The `mLSTM` model by default uses `tf.keras.losses.BinaryFocalCrossentropy` loss; to use binary cross-entropy loss for control signals, one simply needs to set `apply_class_balancing = False` and `gamma = 0` and pass it in as a hyperparameters dictionary during initialization. Figure 5.1 shows an example of how to load such a pipeline to use mixed MSE and BCE loss.

We benchmarked the pipeline with mixed MSE and BCE loss on all of the SMAP and MSL datasets for varying numbers of LSTM units. Table 5.3 show the full results. Because the number of units is a more fixed attribute of the model, we decided to not make it directly modifiable through the pipeline's interface. Rather, one must explicitly modify the number of LSTM units by passing in an edited version of the `layers` dictionary in the hyperparameters. Figure 5.1 shows an example of this as well.

Overall, using BCE loss for the control signal channels improves the average performance

```
from meissa import Meissa
from orion.data import load_signal
from mlstars import load_primitive


# Get layers of mLSTM model to modify the number of LSTM units
lstm_units = 80
mixed_lstm = load_primitive('meissa.primitives.mixed.MixedLSTM')

layers = mixed_lstm.get_hyperparameters()['layers']
for layer_idx in [0, 2]:
        layers[layer_idx]['parameters']['units'] = lstm_units


# Load pipeline and set hyperparameters
hyperparameters={
        'meissa.primitives.timeseries_anomalies.find_anomalies#1': {'fixed_threshold': False},
        'meissa.primitives.mixed.MixedLSTM#1': {
            'layers': layers,
            'gamma': 0,  # no focal parameter
            'apply_class_balancing': False
        }
}

meissa = Meissa(
        pipeline="mixed_lstm",
        hyperparameters=hyperparameters
)
```

Figure 5.1: Example code loading the `MixedLSTM` pipeline with BCE loss and a custom number of LSTM units

LSTM Units	Dataset	Precision	Recall	F1
60	**MSL**	0.531	0.472	0.500
	SMAP	0.884	0.567	0.691
80	**MSL**	0.643	0.500	0.563
	SMAP	0.886	0.582	0.703
120	**MSL**	0.667	0.500	0.571
	SMAP	0.857	0.537	0.661

Table 5.3: Full results from running the `MixedLSTM` pipeline with binary cross-entropy loss for control signals for varying number of LSTM units.

across the two datasets compared to incorporating them naively. For 80 LSTM units, the precision and F1 score for the two datasets was higher than that of running LSTM-AE and LSTM-DT naively on multivariate data (see Figures 5.1 and 5.2), although the recall was lower. With respect to datasets, the `MixedLSTM` model with 80 and 120 units outperformed both the naive multivariate and univariate versions of LSTM-DT/LSTM-AE on the MSL dataset, but not SMAP. Specifically, the F1 score of the univariate LSTM-DT and LSTM-AE pipelines was .707, while the highest F1 score on the SMAP dataset for mixed MSE and BCE pipeline was .703. The relative difference in performance between SMAP and MSL motivates further exploration into the differences in the frequency and types of anomalies between the two datasets.

As mentioned in Chapter 3, one limitation of BCE is it does not handle imbalanced classes unless a weight parameter is specified, which can be detrimental when certain channels contained sparse positive examples. When using BCE naively without weighing classes, the model ended up predicting 0 for many of the control signal channels since it minimized the overall loss. In the case of control signals in our dataset, the sparse positive examples potentially indicates useful information such that may be correlated with an event or disruption in the observation time series.

Binary focal cross-entropy

To more faithfully reconstruct the control signals, we opted for the binary focal cross-entropy loss (FL) when benchmarking the pipeline. As described in Chapter 3, FL has two parameters, α and γ, that can modulate the weight given to positive vs. negative classes as well as to high-confidence vs. low-confidence predictions. To select the parameters tested, we first performed a grid search of hyperparameters and selecting the best performing configurations on the sample set based on F1 score. More details and full results are provided below.

5.3.2 Hyperparameter tuning

To understand the impact of α and γ on reconstruction ability, we perform a coarse hyperparameter search similar to what Lin et al [29] had done for their object detection task. We also varied the number of LSTM units between 60, 80, and 120 for both the encoder and decoder. The pipeline with the given set of hyperparameters was run on a random sample of 10 signals ('E-1' 'E-2' 'M-6' 'S-1' 'P-1' 'M-1' 'E-3' 'M-2' 'P-10' 'S-2'), with 5 from each of the SMAP and MSL data. We test α values of $0.25, 0.5, 0.75$, and 0.9 and γ values of $0, 0.1, 0.2, 0.5, 1$ and 2 as Lin et al had done.

Overall, increasing α – the weight given to positive classes – increased the model's ability to detect anomalies, as the average number of true positives increases with α. The optimal γ parameter depends on the relative magnitude of α as discussed in Section 3.3.2, as increasing the positive class weight decreases the need to penalize confident errors and vice versa. We observe this effect even with our test on a subset of samples, as similar precision and recall was obtained for varying combinations of α and γ. For example, when using 80 LSTM units, we see performance increasing with γ at $\alpha = 0.25$ and $\alpha = 0.5$, while the effect of γ was more variable for higher values of α. The full results in can be found in Table 5.4.

The number of LSTM units had a less discernible difference on performance, as the average precision and recall was similar between the three settings. Setting the number of

α	γ	**fp**	**fn**	**tp**
0.25	0.0	2	7	6
	0.1	4	5	8
	0.2	2	6	7
	0.5	3	6	7
	1.0	2	5	8
	2.0	3	5	8
0.5	0.0	1	6	7
	0.1	3	6	7
	0.2	1	6	7
	0.5	1	5	8
	1.0	1	7	6
	2.0	1	6	7
0.75	0.0	1	6	7
	0.1	1	6	7
	0.2	1	5	8
	0.5	3	5	8
	1.0	3	6	7
	2.0	1	6	7
0.9	0.0	3	5	8
	0.1	2	7	6
	0.2	0	7	6
	0.5	2	7	6
	1.0	1	7	6
	2.0	2	5	8

(a) 60 LSTM units

α	γ	**fp**	**fn**	**tp**
0.25	0.0	0	7	6
	0.1	1	7	6
	0.2	1	6	7
	0.5	3	5	8
	1.0	0	6	7
	2.0	2	5	8
0.5	0.0	1	7	6
	0.1	2	6	7
	0.2	3	6	7
	0.5	2	6	7
	1.0	4	5	8
	2.0	1	6	7
0.75	0.0	3	6	7
	0.1	0	6	7
	0.2	1	5	8
	0.5	1	5	8
	1.0	3	5	8
	2.0	3	6	7
0.9	0.0	1	6	7
	0.1	2	6	7
	0.2	2	5	8
	0.5	0	6	7
	1.0	2	7	6
	2.0	3	5	8

(b) 80 LSTM units

α	γ	**fp**	**fn**	**tp**
0.25	0.0	3	6	7
	0.1	0	6	7
	0.2	4	6	7
	0.5	2	7	6
	1.0	4	6	7
	2.0	4	5	8
0.5	0.0	0	6	7
	0.1	1	6	7
	0.2	3	5	8
	0.5	1	7	6
	1.0	3	6	7
	2.0	3	5	8
0.75	0.0	3	6	7
	0.1	0	6	7
	0.2	1	6	7
	0.5	1	6	7
	1.0	3	7	6
	2.0	0	6	7
0.9	0.0	1	6	7
	0.1	0	6	7
	0.2	1	5	8
	0.5	1	6	7
	1.0	1	5	8
	2.0	1	5	8

(c) 120 LSTM units

Table 5.4: Results from running the `MixedLSTM` pipeline with varying focal loss hyperparameters on a subset of 10 signals. Tables are organized by the number of LSTM units and report the number of false positives, false negatives, and true positives.

LSTM units to 80 produced marginally better detection ability; we use 80 units for the full benchmark below but want to note that the effect on performance seems to be less than tuning the focal loss parameters. In practice, the parameters and desired precision-recall ratio should be chosen based on the context of the anomaly detection task.

5.3.3 Benchmark results

To benchmark the `MixedLSTM` pipeline with focal loss, we selected the 7 best performing α, γ combinations based on F1 score from the sample run in Section 5.3.2 and evaluated

α	γ	**Precision**	**Recall**	**F1**
0.25	1.0	0.643	0.500	0.563
	2.0	0.724	0.583	0.646
0.75	0.1	0.590	0.639	0.613
	0.2	0.724	0.583	0.646
	0.5	0.526	0.556	0.541
0.90	0.2	0.586	0.472	0.523
	0.5	0.475	0.528	0.500

(a) SMAP dataset

α	γ	**Precision**	**Recall**	**F1**
0.25	1.0	0.881	0.552	0.679
	2.0	0.814	0.522	0.636
0.75	0.1	0.837	0.537	0.655
	0.2	0.902	0.552	0.685
	0.5	0.822	0.552	0.661
0.9	0.2	0.925	0.552	0.692
	0.5	0.900	0.537	0.673

(b) MSL dataset

Table 5.5: Benchmark results of `MixedLSTM` pipeline with mixed MSE and focal loss on multivariate SMAP/MSL data

on the entire SMAP and MSL datasets. While testing so many parameter combinations is unrealistic in practice, it was important to try a wide range of values in our case since we were experimenting with a new architecture and needed greater understanding of how sensitive AD ability is to the parameters of the focal loss. We report the precision, recall and F1 scores of `MixedLSTM` below (Table 5.5) and compare it to the benchmark results of other pipelines in *Orion*, as well as the mixed MSE and BCE pipeline.

As for the effect of α and γ on anomaly detection ability, we notice a stark difference depending on the dataset. For instance, the average precision and F1 score of the pipeline increased as α increased for the MSL signals; this was almost the opposite case for SMAP, where the pipeline scored the lowest when α was 0.9. Another observation is that the recall stays mostly constant within the same values of α for the MSL dataset, but varied greatly for SMAP. This may indicate that predicted values for the control channels were much more variable for SMAP data, hence down-weighting confident observations had a more noticeable effect on overall performance. This demonstrates the importance of having experts provide feedback to the AD system and tune parameters according to what they understand about the data, as well as a deeper exploration into the differences between the two datasets.

Compared to the naive multivariate and univariate LSTM-AE pipeline, all configurations benchmarked outperformed the previous models for MSL, while the performance of half the configurations was comparable or better for SMAP. We also looked at the performance of

	Precision	Recall	F1
MSL	0.564103	0.611111	0.586667
SMAP	0.806452	0.746269	0.775194

Table 5.6: Benchmark results of AER on univariate SMAP and MSL data (Source: *Orion*)

the Auto-encoder with Regression (AER) model by Wong et al [15], which currently has the strongest performance out of all pipelines benchmarked in *Orion* (see Table 5.6). The model produces both a reconstructed sequence as well as forward and backward predictions and adds masking to its smoothing step to reduce the incidence of false positives at the beginning of sequences. We found that AER outperformed `MixedLSTM` on the SMAP dataset, but the reverse was true for MSL.

We further examine the cases in which `MixedLSTM` outperforms AER, and vice versa. We chose to use the results of the `MixedLSTM` pipeline with $\alpha = 0.75$ and $\gamma = 0.2$ since it had the highest averaged F1 score between both datasets.

In terms of number of true anomalies detected – true positives, or TPs – AER outperformed on 20/80 signals while `MixedLSTM` outperformed on 8/80. In the first case, there were 35 total anomalies among the 20 signals, of which AER detected 31 while `MixedLSTM` detected 7. Furthermore, Out of the `MixedLSTM` wins, all 10 total true anomalies were detected by the model as opposed to AER, which did not detect them.

52 signals were tied between `MixedLSTM` and AER in the number of TPs. Out of these cases, 1 signal ('M-3') had lower precision (i.e. more false positives) for `MixedLSTM` than AER. Specifically, `MixedLSTM` had 1 false positive (FP) and 1 false negative (FN), while AER only had the FN. There were 13 signals where the opposite was true: the number of TPs were equal, but AER had lower precision. AER had 19 total FPs among this subset, while `MixedLSTM` had one. The remaining 38 out of 52 were true ties – that is, the number FPs, FNs, and TPs were equal. A side-to-side comparison of the results between AER and SMAP at the signal level can be found in Table A.1 in the Appendix.

5.4 Error and anomalies aggregation

While we reconstruct all channels of the signal for the results above, only the observation signal is used for detecting anomalies and scoring. This reflects the approach taken for benchmarking other pipelines in Orion and in much of the AD literature in the past. However, we also wanted to examine ways of incorporating the reconstruction error of the control signal columns into the anomaly detection process.

A simple approach we looked into was calling `find_anomalies` on each channel independently and aggregating the detected intervals based on the relative score. The issue is that if we simply took all of the anomalies detected in the control signal columns as truth, we would have an abundance of false positives. Indeed, we observed that the number of false positives scaled linearly with the number of control signal channels. Thus, some level of filtering is needed. An example scheme would be to consider an anomalous interval detected in a channel as a "vote" for that interval, and overlapping intervals detected in other channels add additional votes. Thus, the probability of an interval being truly anomalous can be ranked according to the number of channels it was detected in. Another source of information is the severity score associated with the detected anomaly, which can be used as a weight for the likelihood of an anomaly being a true anomaly.

Unfortunately, we found several limitations with such an approach, which we will illustrate through an example signal. Table 5.7 shows the first 5 anomalies detected by the `MixedLSTM` model with $\alpha = 0.75$ and $\gamma = 0.2$ compared to the true anomalies of the 'P-11' signal. The model detects the first ground-truth anomaly in the observation channel (channel 0) but not the second.

In total, 54 anomalies were detected among the 55 channels. Out of those detected anomalies, 5 overlapped with the first ground truth anomaly, and 5 overlapped with the second ground truth anomaly. While the above sounds promising for incorporating anomalies detected in control signal channels, the channels that identified such anomalies all had relatively

	channel	start	end	severity
0	0	1346954400	1350172800	0.129006
1	11	1333908000	1336435200	0.247503
2	12	1222819200	1224007200	0.464919
3	19	1226361600	1230120000	1.439419
4	19	1262887200	1266645600	1.439419

	start	end
0	1346954400	1349546400
1	1335290400	1337580000

Table 5.7: First 5 detected anomalies (left) from `MixedLSTM` and true anomalies (right) of signal 'P-11' from the MSL dataset

low scores: the average severity score among channels that detected the first ground-truth anomaly was 0.28 and 0.42 for the second ground-truth anomaly. Even from looking at the subset of detected anomalies in Table 5.7, the interval in channel 11 overlaps with the second ground-truth anomaly, but the severity score is lower than all the other intervals below it which corresponded to false positives. Even in terms of frequency (number of channels that detected the anomaly), there were 3 other intervals detected that had the same number of "votes" from different channels.

One potential way to reduce the frequency of this issue is by using the relative ranking of the anomaly score for that channel instead of the true severity value. Additionally, one could consider recomputing the anomaly scores in terms of the error magnitudes across all channels at once, rather than independently, could create a way of more fairly ranking the anomaly severities globally. However, scaling would need to be done first such that all channels fall between the same range of values.

As evidenced above, there are many possibilities for explicitly incorporating the control signal in the post-processing stages of the AD pipeline, and we discuss some possible explorations in Chapter 6 before concluding.

Chapter 6

Discussion

In this chapter, we offer further interpretation of the results, discuss the limitations of the thesis work, and offer suggestions for future work.

6.1 Limitations

6.1.1 Control signal representation

As it stands, the *Meissa* pipelines assume that the control signal is binary and takes on [0, 1]. However, this is a simplified view as control signals can very well be categorical with more than 2 classes; for instance, one can imagine a sensor switch with $n > 2$ settings). It could also even be continuous, although those situations could be more complicated in that they share more similarities with the observation signal (and thus could do well with the same loss as in the naive method), but differ in a harder-to-observe way.

Additionally, the current approach aims to reconstruct the control signal faithfully by computing a threshold that maximizes alignment with the input signal. This does reduce the amount of information given by the reconstruction, as the output probabilities provides a more granular representation of what state the model anticipates the control signal to be in. Future iterations can look into ways of computing errors by using the output probabilities

as opposed to the thresholded output.

6.1.2 Balancing precision and recall

From the `MixedLSTM` benchmark results section in 5.3.3., we saw how introducing focal loss improved detection ability on the MSL dataset significantly, but lowered recall for the SMAP dataset. In general, introducing the focal parameter $(1 - p_t)^\gamma$ for $\gamma > 0$ reduces the incidence of false positives. This is consistent with the role of γ as a penalizing factor for over-confident false predictions that scales exponentially with the magnitude of γ. However, the model struggled with detecting anomalies when γ was too high for the given α weight, which we saw in the case of the SMAP dataset when $\alpha = 0.9$ and $\gamma = 0.5$. Furthermore, the optimal α and γ values differed significantly between datasets, which can make selecting the right values for generalization tricky.

One approach to improving recall is to incorporate control signals at the post-processing stage as well, rather than just the modeling stage. We can consider detecting anomalies by analyzing the reconstruction error for *all* channels, not just the observation channel, as discussed in Section 5.4. The drawback is that evaluating each channel independently at the anomaly detection stage (by applying `find_anomalies` to each column one at a time) means that the information in a given channel can be more easily dominated by the noise in that channel, leading to many disjoint false positive intervals. To prevent the number of false positives from scaling linearly with the number of channels, it is necessary to employ an aggregation scheme that takes into account the number of channels that had an anomaly in a given interval while also weighing severity and the relative importance of each channel, if such information is known. Consider the case where only 1 or 2 channels are correlated with true feature due to their unique role; we want to make sure those channels are still considered if they have high anomaly score, even though the anomalies form a minority among all channels.

6.1.3 Practicality

The goal of every framework in *Sintel* is to enable users to improve performance on real-life AD tasks. In such cases, fully unsupervised methods are generally better where labeled data isn't readily available. On the other hand, the benchmark functionality takes on more of a semi-supervised approach where the pipeline is purposefully trained on non-anomalous data points so that it can more clearly construct a representation of normal patterns. Furthermore, the pipeline constructed is reconstruction-based, which has its own limitations. It would be useful to consider a more *online* workflow using a prediction-based pipeline since in `mixed_lstm`, given information about a signal up to time t, it cannot offer predictions about its properties at $t + 1$; it can only look back at what's been collected so far.

6.2 Future work

6.2.1 More complex architecture

Currently, the `MixedLSTM` model is just a simple auto-encoder with one LSTM layer for each of the encoder and decoder layers. The simplicity of the architecture was helpful for providing a basis for comparing to the existing Orion pipelines since it was the same as the original LSTM-AE pipeline benchmarked on SMAP/MSL save for the input and output dimensions.

Similarly, can look into other loss functions that vary based on the state of the control signal. Focal cross-entropy kind of includes this by default but there could be more effective and generalizable methods. Again, we would ideally want to also be able to handle cases where the control signal has more than two classes of values.

6.2.2 Signal-specific error computation

It would be of interest and value to explore ways of computing regression/reconstruction errors differently for control signals, depending on whether the end task includes detecting anomalies in the control signal channels or not. This also depends on the representation of the control signal, where if we used the output probabilities as mentioned in 6.6.1., it may be more suited for certain types of error computations.

6.3 Conclusion

In summary, this thesis has demonstrated the potential of incorporating control signals into the anomaly detection process for multivariate, heterogeneous time series data. We developed a reconstruction-based pipeline, `MixedLSTM`, with pre-processing, modeling, and evaluating methods that consider both observation and control signals and provide a user-friendly interface to access and create such pipeline through the *Meissa* library. Our experiments using data from the NASA Soil Moisture Active Passive (SMAP) satellite and the Mars Science Laboratory (MSL) Rover validate the efficacy of our approach, outperforming the univariate and naive multivariate LSTM pipelines on the majority of signls. These advancements highlight the importance of leveraging all available data, including control signals, to improve the robustness and accuracy of anomaly detection systems, thereby reducing the reliance on human expertise and enhancing operational efficiency in various applications. Future work can build on these findings to further refine and expand the methodologies for even broader applicability.

Appendix A

Comparison of MixedLSTM and AER

Dataset	**Signal**	**FN**$_m$	**FP**$_m$	**TP**$_m$	**F1**$_m$	**FN**$_a$	**FP**$_a$	**TP**$_a$	**F1**$_a$
MSL	C-1	1	0	1	0.667	1	2	1	0.4
MSL	C-2	2	0	0	–	1	0	1	0.667
MSL	D-14	1	0	1	0.667	0	0	2	1.0
MSL	D-15	0	1	1	0.667	1	0	0	–
MSL	D-16	0	0	1	1.0	0	0	1	1.0
MSL	F-4	1	1	0	–	1	4	0	–
MSL	F-5	0	0	1	1.0	0	1	1	0.667
MSL	F-7	0	0	3	1.0	3	0	0	–
MSL	F-8	0	0	1	1.0	0	2	1	0.5
MSL	M-1	1	0	0	–	0	1	1	0.667
MSL	M-2	0	2	1	0.5	1	0	0	–
MSL	M-3	1	1	0	–	1	0	0	–
MSL	M-4	0	1	1	0.667	1	1	0	–
MSL	M-5	1	0	0	–	1	0	0	–
MSL	M-6	0	0	1	1.0	0	0	1	1.0

MSL	M-7	0	0	1	1.0	0	1	1	0.667
MSL	P-10	0	0	1	1.0	0	0	1	1.0
MSL	P-14	0	0	1	1.0	0	0	1	1.0
MSL	P-15	0	0	1	1.0	0	0	1	1.0
MSL	S-2	0	0	1	1.0	0	0	1	1.0
MSL	T-12	1	0	0	–	1	2	0	–
MSL	T-13	2	1	0	–	1	0	1	0.667
MSL	T-4	0	0	1	1.0	0	0	1	1.0
MSL	T-5	0	0	1	1.0	0	0	1	1.0
MSL	T-8	2	1	0	–	0	2	2	0.667
MSL	T-9	1	0	1	0.667	1	0	1	0.667
SMAP	A-1	0	0	1	1.0	0	0	1	1.0
SMAP	A-2	0	0	1	1.0	0	0	1	1.0
SMAP	A-3	0	0	1	1.0	0	1	1	0.667
SMAP	A-4	0	0	1	1.0	0	0	1	1.0
SMAP	A-5	0	0	1	1.0	0	0	1	1.0
SMAP	A-6	0	0	1	1.0	0	0	1	1.0
SMAP	A-7	1	0	0	–	0	0	1	1.0
SMAP	A-8	1	1	0	–	0	0	1	1.0
SMAP	A-9	1	0	0	–	1	1	0	–
SMAP	B-1	0	0	1	1.0	1	1	0	–
SMAP	D-1	0	0	1	1.0	0	0	1	1.0
SMAP	D-11	0	0	1	1.0	0	0	1	1.0
SMAP	D-12	0	0	1	1.0	0	0	1	1.0
SMAP	D-13	0	0	1	1.0	0	1	1	0.667
SMAP	D-2	0	0	1	1.0	0	0	1	1.0
SMAP	D-3	1	0	0	–	0	0	1	1.0

SMAP	D-4	1	0	0	–	1	0	0	–
SMAP	D-5	1	0	0	–	1	0	0	–
SMAP	D-6	0	0	1	1.0	0	1	1	0.667
SMAP	D-7	0	0	1	1.0	0	0	1	1.0
SMAP	D-8	0	0	1	1.0	0	0	1	1.0
SMAP	D-9	0	0	1	1.0	0	0	1	1.0
SMAP	E-1	1	0	1	0.667	0	0	2	1.0
SMAP	E-10	1	0	1	0.667	0	0	2	1.0
SMAP	E-11	1	0	1	0.667	0	1	2	0.8
SMAP	E-12	2	1	0	–	0	1	2	0.8
SMAP	E-13	3	0	0	–	2	0	1	0.5
SMAP	E-2	0	0	1	1.0	1	0	0	–
SMAP	E-3	1	0	0	–	0	0	1	1.0
SMAP	E-4	1	0	0	–	1	1	0	–
SMAP	E-5	1	0	0	–	0	1	1	0.667
SMAP	E-6	0	0	1	1.0	0	0	1	1.0
SMAP	E-7	0	0	1	1.0	0	0	1	1.0
SMAP	E-8	1	1	0	–	0	0	1	1.0
SMAP	E-9	0	0	1	1.0	0	0	1	1.0
SMAP	F-1	1	1	0	–	1	1	0	–
SMAP	F-2	0	0	1	1.0	0	0	1	1.0
SMAP	F-3	1	0	0	–	1	0	0	–
SMAP	G-1	1	0	0	–	1	1	0	–
SMAP	G-2	0	0	1	1.0	0	0	1	1.0
SMAP	G-3	0	0	1	1.0	0	1	1	0.667
SMAP	G-4	0	0	1	1.0	0	0	1	1.0
SMAP	G-6	0	0	1	1.0	0	0	1	1.0

SMAP	G-7	2	0	1	0.5	0	0	3	1.0
SMAP	P-1	3	0	0	–	3	0	0	–
SMAP	P-3	0	0	1	1.0	0	0	1	1.0
SMAP	P-4	2	0	1	0.5	0	0	3	1.0
SMAP	P-7	1	0	0	–	0	0	1	1.0
SMAP	R-1	0	0	1	1.0	0	0	1	1.0
SMAP	S-1	0	0	1	1.0	1	0	0	–
SMAP	T-1	1	0	1	0.667	1	0	1	0.667
SMAP	T-2	0	0	1	1.0	1	0	0	–
SMAP	T-3	0	0	2	1.0	0	0	2	1.0

Table A.1: Full benchmark results for (1) `MixedLSTM` (subscript m) with parameters $\alpha = 0.75$, $\gamma = 0.2$, and 80 LSTM units; (2) AER (subscript a).

References

[1] K. Hundman, V. Constantinou, C. Laporte, I. Colwell, and T. Soderstrom, "Detecting spacecraft anomalies using lstms and nonparametric dynamic thresholding," Jun. 2018. URL: https://arxiv.org/abs/1802.04431.

[2] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha, "Unsupervised real-time anomaly detection for streaming data," *Neurocomputing*, vol. 262, pp. 134–147, 2017, Online Real-Time Learning Strategies for Data Streams, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.04.070. URL: https://www.sciencedirect.com/science/article/pii/S0925231217309864.

[3] S. Schmidl, P. Wenig, and T. Papenbrock, "Anomaly detection in time series: A comprehensive evaluation," *Proc. VLDB Endow.*, vol. 15, no. 9, pp. 1779–1797, May 2022, ISSN: 2150-8097. DOI: 10.14778/3538598.3538602. URL: https://doi.org/10.14778/3538598.3538602.

[4] L. Ruff, R. A. Vandermeulen, N. Görnitz, A. Binder, E. Müller, K.-R. Müller, and M. Kloft, *Deep semi-supervised anomaly detection*, 2020. arXiv: 1906.02694 [cs.LG].

[5] J. Audibert, P. Michiardi, F. Guyard, S. Marti, and M. A. Zuluaga, "Usad: Unsupervised anomaly detection on multivariate time series," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 2020, pp. 3395–3404.

[6] H. Zhao, Y. Wang, J. Duan, C. Huang, D. Cao, Y. Tong, B. Xu, J. Bai, J. Tong, and Q. Zhang, "Multivariate time-series anomaly detection via graph attention network," in *2020 IEEE International Conference on Data Mining (ICDM)*, 2020, pp. 841–850. DOI: 10.1109/ICDM50108.2020.00093.

[7] S. Alnegheimish, D. Liu, C. Sala, L. Berti-Equille, and K. Veeramachaneni, "Sintel: A machine learning framework to extract insights from signals," in *Proceedings of the 2022 International Conference on Management of Data*, 2022. DOI: 10.1145/3514221.3517910.

[8] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[9] S. D. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 29–38.

[10] Y. Gao, T. Yang, M. Xu, and N. Xing, "An unsupervised anomaly detection approach for spacecraft based on normal behavior clustering," in *Proceedings of the 2012 Fifth International Conference on Intelligent Computation Technology and Automation*, ser. ICICTA '12, USA: IEEE Computer Society, 2012, pp. 478–481, ISBN: 9780769546377. DOI: 10.1109/ICICTA.2012.126. URL: https://doi.org/10.1109/ICICTA.2012.126.

[11] E. J. Hannan, *Multiple time series*. John Wiley & Sons, 2009.

[12] E. H. M. Pena, M. V. O. de Assis, and M. L. Proença, "Anomaly detection using forecasting methods arima and hwds," in *2013 32nd International Conference of the Chilean Computer Science Society (SCCC)*, 2013, pp. 63–66. DOI: 10.1109/SCCC.2013.18.

[13] N. Laptev, S. Amizadeh, and I. Flint, "Generic and scalable framework for automated time-series anomaly detection," Aug. 2015, pp. 1939–1947. DOI: 10.1145/2783258.2788611.

[14] N. Ding, H. Gao, H. Bu, H. Ma, and S. Huaiwei, "Multivariate-time-series-driven real-time anomaly detection based on bayesian network," *Sensors*, vol. 18, p. 3367, Oct. 2018. DOI: 10.3390/s18103367.

[15] L. Wong, D. Liu, L. Berti-Equille, S. Alnegheimish, and K. Veeramachaneni, "Aer: Auto-encoder with regression for time series anomaly detection," in *2022 IEEE International Conference on Big Data (Big Data)*, Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2022, pp. 1152–1161. DOI: 10.1109/BigData55660.2022.10020857. URL: https://doi.ieeecomputersociety.org/10.1109/BigData55660.2022.10020857.

[16] R.-J. Hsieh, J. Chou, and C.-H. Ho, "Unsupervised online anomaly detection on multivariate sensing time series data for smart manufacturing," in *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*, 2019, pp. 90–97. DOI: 10.1109/SOCA.2019.00021.

[17] D. Park, Y. Hoshi, and C. Kemp, "A multimodal anomaly detector for robot-assisted feeding using an lstm-based variational autoencoder," *IEEE Robotics and Automation Letters*, vol. PP, Nov. 2017. DOI: 10.1109/LRA.2018.2801475.

[18] A. Geiger, D. Liu, S. Alnegheimish, A. Cuesta-Infante, and K. Veeramachaneni, "Tadgan: Time series anomaly detection using generative adversarial networks," in *2020 IEEE International Conference on Big Data (Big Data)*, Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2020, pp. 33–43. DOI: 10.1109/BigData50022.2020.9378139. URL: https://doi.ieeecomputersociety.org/10.1109/BigData50022.2020.9378139.

[19] M. J. Smith, C. Sala, J. M. Kanter, and K. Veeramachaneni, "The machine learning bazaar: Harnessing the ml ecosystem for effective system development," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*,

ser. SIGMOD/PODS '20, ACM, May 2020. DOI: 10.1145/3318464.3386146. URL: http://dx.doi.org/10.1145/3318464.3386146.

[20] B. O. Collazo Santiago, "Machine learning blocks," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[21] W. Xue *et al.*, "A flexible framework for composing end to end machine learning pipelines," Ph.D. dissertation, Massachusetts Institute of Technology, 2018.

[22] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, "Time-series anomaly detection service at microsoft," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 3009–3017.

[23] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Lstm-based encoder-decoder for multi-sensor anomaly detection," *arXiv preprint arXiv:1607.00148*, 2016.

[24] E. Dai and J. Chen, "Graph-augmented normalizing flows for anomaly detection of multiple time series," *arXiv preprint arXiv:2202.07857*, 2022.

[25] R. Hasani, M. Lechner, A. Amini, D. Rus, and R. Grosu, "Liquid time-constant networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 7657–7666.

[26] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh, "Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *2016 IEEE 16th international conference on data mining (ICDM)*, Ieee, 2016, pp. 1317–1322.

[27] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19, Anchorage, AK, USA: Association for Computing Machinery,

2019, pp. 2828–2837, ISBN: 9781450362016. DOI: 10.1145/3292500.3330672. URL: https://doi.org/10.1145/3292500.3330672.

[28] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, 1978. DOI: 10.1109/TASSP.1978.1163055.

[29] T. Lin, P. Goyal, R. B. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *CoRR*, vol. abs/1708.02002, 2017. arXiv: 1708.02002. URL: http://arxiv.org/abs/1708.02002.