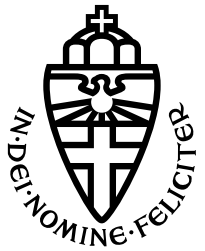
RADBOUD UNIVERSITY NIJMEGEN



FACULTY OF SOCIAL SCIENCES

Leveraging the Transformer Architecture for Anomaly Detection on Multivariate Time Series Data

MASTER'S THESIS IN ARTIFICIAL INTELLIGENCE

Author:
Max Jacobus Johannes de Grauw

External Supervisor:
dr. Paolo Pileggi
TNO, Monitoring & Control Services

Internal Supervisor:
dr. Umut Güçlü
Department of Artificial Intelligence

Second Assessor:
dr. Yagmur Güçlütürk
Department of Artificial Intelligence

August 2020

CONTENTS

List of Abbreviations

AE Autoencoder

CNN Convolutional Neural Network

DAD Deep Anomaly Detection
DM Digital Model
DS Digital Shadow
DT Digital Twin

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

ML Machine Learning
MTS Multivariate Time Series

NRMSE Normalized Root Mean Squared Error

PLM Product Lifecycle Management

RNN Recurrent Neural Network

TCN Temporal Convolutional Network

XAI Explainable Artificial Intelligence

Leveraging the Transformer Architecture for Anomaly Detection on Multivariate Time Series Data

Max de Grauw
Dept. of Artificial Intelligence
Radboud University
Nijmegen, The Netherlands
m.degrauw@student.ru.nl

Paolo Pileggi*
Dept. Monitoring & Control Services
TNO
The Hague, The Netherlands
paolo.pileggi@tno.nl

Umut Güçlü*
Dept. of Artificial Intelligence
Radboud University
Nijmegen, The Netherlands
u.guclu@donders.ru.nl

*Thesis Supervisors

***Abstract*—Due to widespread digitalization, predictive maintenance and Digital Twinning are increasingly viable and popular options for industry aiming for smarter operation and maintenance of industrial equipment. Machine Learning will play a crucial role in this transition, augmenting existing expert knowledge. There is a growing need for robust and transparent methods for analyzing and acting upon the wealth of sensor data available in this field. In this thesis we study the feasibility of using a Transformer model for anomaly detection in an industrial setting. Leveraging the Transformer architecture for anomaly detection allows for studying the effects of neural attention distributions on predictions, a step towards more interpretable deep anomaly detection models. We compare the signal reconstruction and anomaly detection performance of an LSTM encoder-decoder model with that of a Transformer. We also apply our models to a real world use case, detecting anomalies in an industrial gas turbine. The lack of easy access to anomalous training data in industrial settings remains a challenge. In the future our approach can be extended to perform anomaly prediction, functioning as an early warning system or scheduling predictive maintenance as part of a Digital Twin.**

***Index Terms*—Machine Learning, Transformer, LSTM, Gas Turbine, Anomaly Detection, Digital Twin**

I. Introduction

Increasing availability of data for complex physical systems, such as combustion turbines, has led to a growing interest in the Digital Twin (DT) paradigm [1–3], a coupling of virtual models made of an artefact to the artefact itself that may lead to novel insights into system functioning and increasing of predictive power. In this thesis we present an anomaly detection model based on the Transformer architecture [4] and discuss its suitability for integration into the system health monitoring component of a DT. To the best of our knowledge we provide the first implementation of a Transformer network for industrial anomaly detection based on Multivariate Time Series (MTS) data.

A. Use Case

OPRA Turbines[1] manufactures and services state of-the-art gas turbine systems. The gas turbine that was studied for this project is the OP16 model (see figure 1): a single-shaft, all radial, gas turbine for industrial and commercial use. This gas turbine is integrated into a larger portable generator package consisting of the turbine, gearbox, generator, and necessary auxiliary equipment and systems.

The generator is subject to a variety of external conditions such as the air pressure, temperature and humidity of the environment whilst also being influenced by operational conditions such as the fuel type and power settings. Adding to that, the complexity of its internal components means that there are many possible explanations for abnormal performance in this system.

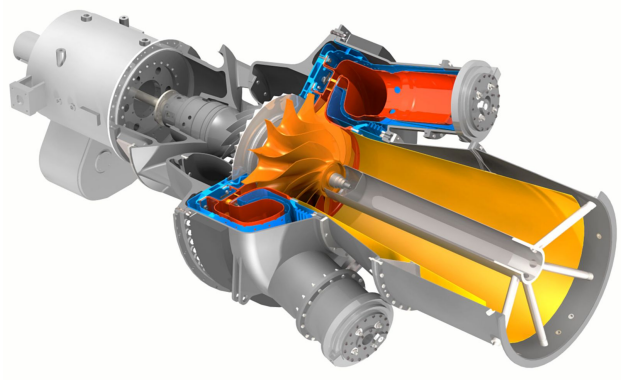


Figure 1. The OP16 gas turbine (courtesy of OPRA Turbines)

OPRA manages their fleet of installed turbines using a variety of sensor measurements of the turbine and other parts of the generator (approximately 256 different signals).

OPRA currently uses a rule-based model to link sensor values to potential root causes of anomalies [5]. Using this

[1]https://www.opraturbines.com/, last accessed 14 August 2020

model they aim to minimize overall life cycle costs through predictive maintenance. The rule-based model is used both for diagnostics and for predicting likely problems in future operation on a component level. The existing model compares an offline thermodynamic model of the generator against online performance of a specific gas turbine to detect anomalies.

B. Problem Statement

OPRA's current anomaly detection model detects anomalies which fall outside of pre-defined ranges of normal behaviour. If the use case owner wishes to change these ranges to incorporate new insights it has to be done manually by introducing new rules or changing existing ones. Their distinctions between normal and abnormal behaviour are based on experiences from service engineers and domain experts, which can be subject to personal biases and the limits of human perceptual abilities.

By incorporating a self-learning model, that can automatically infer relations between sensor data and anomalies using historical data, personal biases can be avoided and subtle errors can be identified and remedied. Such a model could also continuously update and refine its definitions of normal and abnormal behaviour based on new operational data, resulting in more accurate long-term predictions.

By improving the performance of anomaly detection models we can provide early warnings of artefact failures, reduce unnecessary maintenance and contribute to an improved product lifetime and uptime.

Machine Learning (ML) components intended to be integrated into the maintenance model would need to be robust to noise, and the diverse working conditions of the artefact. Furthermore, interpretability of the model would help those who use it on a daily basis learn to trust the decisions of the model and manage false positives.

In recent years there has been a shift towards black-box ML methods, such as neural networks, over more traditional and transparent statistical models. This has harmed our ability to explain *how* and *why* a model makes decisions [6]. Recovering this information is crucial when verifying predictions or when attempting to gain novel insights from a model [7]. Nevertheless, the greatly improved performance of modern ML techniques is too significant to ignore. Therefore, it is crucial to use or develop techniques that allow us to interpret black-box ML models.

To be considered a true improvement over the existing solution, an anomaly detection and classification model would need to outperform the current baseline and meet the interpretability requirements set by the operator.

C. Contributions

We evaluated two types of neural network architectures, a Long Short-Term Memory (LSTM) network [8] and a Transformer, on anomaly detection tasks. We also applied these models on experimental data supplied by OPRA for detection of gas turbine ignition anomalies.

We investigated the effects of variable input/output window lengths and increasing the number of studied sensors on detection performance. We provide arguments supporting the viability of replacing traditional Recurrent Neural Networks (RNN) with a Transformer architecture for anomaly detection.

For an ML model to be applied in a real world setting it needs to be robust and explainable to be trusted by experts. We provide insights, based on recent studies, on how a Transformer model can be made interpretable. In the future, explainable anomaly detection models may yield new insights into the origins of failures and optimal sensor configuration. Our contributions are:

1) **A novel application of the Transformer architecture to industrial anomaly detection using MTS data.**
2) **A comparative study of the performance of a Transformer and LSTM architecture on this task.**
3) **An evaluation of our model and approach on real world operational data gathered from a gas turbine.**

D. Outline

This introductory section provided an overview of the use case, problem space and motivation behind our research. Section II will provide the required background knowledge for anomaly detection on MTS data and interpretability of ML models. Section III discusses related research efforts in this domain. Section IV provides a detailed descriptions of our chosen approach and describes the experiments we ran. We present the outcomes of our experiments in section V. In Section VI we discuss our observations, the limitations of our approach and the applicability of our tested models outside of a research setting. Finally, Section VII concludes this thesis with our key findings and suggestions for future work.

II. Background

In this section we introduce the required background knowledge for this thesis. We start with a brief discussion of the DT, in which we foresee an application of interpretable anomaly detection models. We then discuss the field of anomaly detection itself and common ML methods used for solving problems in this domain in section II-B. We also introduce a vocabulary for positioning an interpretable model in the larger context of Explainable Artificial Intelligence (XAI) in section II-C. Finally, we discuss neural attention and the Transformer architecture itself in section II-D.

A. The Digital Twin & Product Lifecycle Management

There are many competing definitions and interpretations for the DT. We have adapted ours from that of Glaessgen & Stargel's [9]:

> "A DT is an integrated multiphysics, multiscale, probabilistic simulation of a product or system that uses physical models, and real-time sensor and field data *to mirror, as closely as possible, the life cycle of its physical counterpart*."

A fully-implemented DT consists of three components: the physical artefact, virtual model of said artefact and a *digital thread* of data connecting them.

Kritzinger et al. [10] distinguish between Digital Models (DM), Digital Shadows (DS) and the DT.

A DM is a virtual representation of the artefact in which there is no automatic flow of information between the physical object and the model. Such models are often constructed using historical data, simulations and expert knowledge. They may also incorporate insights from "fleet leaders", the artefacts in the fleet that experienced the most use or the worst degradation. Since there is no automatic information transfer in a DM it cannot be tailored to each specific instance of an artefact that it models.

We can contrast this with a DS in which there is a one-way automatic information transfer from the artefact to the virtual model. With this information the virtual model can mirror the state and conditions of each physical artefact allowing for more fine-grained predictions.

Finally, a DT differs from a DS in that there is also a transfer of information from the virtual model to the physical artefact. In a DT the virtual component can also affect the state of the physical component, for example, lowering fuel intake when it predicts, through simulation, that there is a risk of the engine overheating in the near future.

Building a DT from scratch is a major undertaking requiring specialized information at multiple levels of abstraction and expertise which may not be available for each use case. A study of the literature shows that only a few research groups have managed to complete a fully developed DT [10].

Taking an iterative approach to DT development may be wise. In this thesis we focused on contributing to system and sensor health modeling by improving anomaly detection capabilities. In later stages of DT development our solution could be leveraged for system health predictions and simulations.

Product Lifecycle Management (PLM) [11, 12] is a product-centric approach to managing the diverse business processes required for successful development and deployment of a service or product. The product lifecycle is commonly defined as consisting of six stages: development, manufacture, distribution, use, maintenance and retirement [13]. The digital revolution has made it possible to generate, store and process vast amounts of data for each of these stages.

Using knowledge of the physics underpinning an artifact or system, combined with environmental, operational and sensor data a DT should be able to simulate the complete life cycle of a product. Although the DT may benefit many different aspects of product lifecycle management [14], our research is situated in the stages of product use and maintenance. By analyzing the data generated in these two stages predictions can be made about the modelled artefact's health, with the intent of prolonging the useful life of the product and minimizing maintenance costs. End users expect stable products, and the demand for more sustainability in recent years has led to a growing market for services like predictive maintenance and PLM [15]. There is a strong value proposition for companies able to leverage the DT to deliver these services.

B. Deep Anomaly Detection

Anomaly detection is the problem of finding patterns in data that differ significantly from the norm according to a certain metric. It is a research topic with a long history in diverse research areas [16]. Anomaly detection techniques have been successfully applied in industrial settings (e.g., detection of component failures in spacecraft [17], ensuring the security of cyber-physical systems [18,19], intrusion detection in wireless sensor networks [20], and damage prevention in the petroleum industry [21]). Nevertheless, anomaly detection can be a challenging problem: it is often hard to define normal behaviour, which is not guaranteed to be static, gathering labeled data for training models is often expensive and noise present in real world data tends to follow similar trends as actual anomalous data. These and other problems makes discerning true anomalous data far from trivial in complex use cases.

A further challenge in our gas turbine use case is that we are working with *temporal*, or *time series* data. When detecting anomalies in time series we look for *contextual* or *conditional anomalies*, in which the temporal structure of the data is relevant to determining whether behaviour is anomalous.

Another attribute of our data is that anomalies are defined with respect to the entire data sequence. A single point in time may not represent an anomaly but a group of inconsistent points may constitute a *collective anomaly*.

Our use case belongs to the category of *industrial damage detection* or more specifically *system health management*, in which anomalies correspond to defects or behaviour that may lead to defects. Data collected for this task is often referred to as sensor data as it is collected using sensors measuring different aspects of the modeled system. Accurate and representative anomalous data is usually expensive to acquire, due to the infrequent occurrence of defects compared to normal operation.

Although techniques exist to simulate anomalous behaviour according to known properties, more frequently semi-supervised classification approaches are used in which we use only data from normal behaviour to train a detection model. If we have an accurate understanding of normal behaviour we can contrast this with new inputs to detect anomalies as outliers to our model.

Previously, anomaly detection models used to be mostly statistical or based on simple neural architectures [22]. However, we can see a clear trend towards more complex neural architectures starting in the late 2000s [23], culminating in Deep Anomaly Detection (DAD) in the 2010s [24]. Although it remains common practice to develop models specifically for a certain use case, there are also some architectures which see broader use. For industrial damage detection and time series data DAD is often based on Convolutional Neural Networks (CNNs) [25, 26], Recurrent Neural Networks

(RNNs) (especially Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRU)) [27,28] or Autoencoders (AEs) [29]. For a review of older anomaly detection methods and applications we refer to Prasad et al. [23], whereas Wang et al. [24] and Chalapathy et al. [30] provide reviews of more recent ML techniques for DAD.

A challenge in DAD is keeping computational complexity low as the length of time over which anomaly detection is performed increases. Most DAD applications also require the possibility of real time detection which further limits the algorithms which can be used. We will now discuss some of the common neural architectures used for sequence modeling and DAD, starting with Autoencoders.

Autoencoders are a form of *unsupervised* neural network that uses backpropagation to learn an identity function that maps the input representation, through a bottleneck, to the output representation [31]. Due to this bottleneck, AE are forced to learn to reconstruct the output using generalized features from its inputs [32].

Since AEs are an unsupervised learning method they can be trained using just an input representation, they do not require data to be labelled as anomalous or normal [33]. This major advantage has led to AEs being widely used for anomaly detection tasks where labelled data is hard to acquire [24, 30].

Another benefit of AEs is that we can analyze the representations in the bottleneck to discover what general features the model has found in the input data for reconstruction of the output [34].

An often employed manner of utilizing AEs for anomaly detection is to train them on data which is known to be representative of normal behaviour, which is often easy to collect, and then evaluate them based on the reconstruction error between the AEs input and output representations for novel data. A model trained on normal behaviour should have relatively low reconstruction errors on similar normal behaviour whilst performing poorly on abnormal inputs. This allows it to differentiate normal from abnormal behaviour. We will use a similar approach for detecting anomalies using more complex models in our experiments. AEs are often outperformed by other architectures, such as RNNs, when labelled data is available but they can also be incorporated in more complex neural architectures for feature discovery [33].

Recurrent Neural Networks [35, 36] have their origins in the 1980's[2] when they were developed with the goal of modeling sequential data using neural networks. In its broadest sense an RNN is any neural network architecture where sequential dependencies are dealt with by a one-way directed connection between previous instantiations of the network to the current network. This is often achieved by using a recurrent connection in which the output of the network at step t_{x-1} is used as input at step t_x. RNNs are notoriously hard to train due to the *vanishing* and *exploding gradient problem* [37] in which, due to the many backpropagation steps in a recurrent network, the error gradient quickly dissipates or blows up.

[2]Although they were preceded by Little-Hopfield networks, a subclass of the more general RNN.

LSTM networks [38] are a popular form of RNN used in a wide variety of sequence modeling tasks. It follows the same basic principle of recurrence as in RNNs. LSTMs however, add a couple of additional components. The *cell state* can be thought of as the memory of the network. This cell state is modulated by three *gate units*. First, the *forget gate* allows the model to prevent certain information being carried over from previous time steps. Then the *input gate* decides what information from the current time step to add to the cell state. This may mean updating old values or adding completely new information to the memory. Finally an *output gate* is used to filter out the relevant information from the cell state and to create the prediction for that time step. For the next time step the unfiltered cell state is used, in combination with new observations, as input for the network. Figure 2 gives an overview of the inner workings of a single LSTM unit.

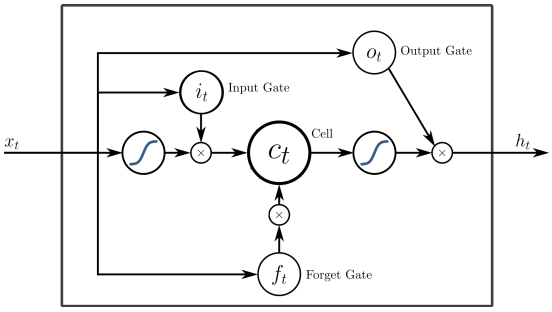


Figure 2. An overview of the LSTM unit architecture

LSTMs significantly reduced the vanishing gradient problem of training RNNs with back-propagation. By encoding temporal dependencies in the cell state and modifying it through the use of gates it would take the subgradients of all three gates to vanish before we would reach the situation where a network will stop learning. GRU models [39] are a popular alternative for LSTMs with a similar architecture and performance [40]. Long-term dependencies remain challenging to learn using canonical LSTMs [41], with recent research even indicating that the long-held notion that recurrence is a necessity for sequence modeling may be false [42].

Previously, researchers often combined a convolutional network and a recurrent network for temporal classification as a stacked model to offset this limitation. The convolutional architecture being used to encode spatiotemporal features locally, which are then used as input for a recurrent network which was tasked to find the long term, high-level relationships between features.

Lai et al. [43] implemented such a model, LSTNet, for MTS forecasting while Zhao et al. [44] introduced their own variant, convolutional bi-directional LSTM for machine health monitoring. Whilst these kinds of hybrid models performed

quite well for some time they also have a significant drawback: the added complexity of having to train and fine-tune two models, each with their own intricacies proved to be a limiting factor. This has lead to increasing interest in non-recurrent network architectures. We discuss attention based models as an alternative to recurrent architectures for sequence modeling in section II-D.

Temporal Convolutional Networks (TCNs) [45] are one class of neural network architectures that do not use recurrence when processing temporal data. Bai et al. [42] show how a general TCN architecture can outperform a variety of recurrent models (the canonical RNN, LSTM and GRU). Their sequence modeling network, which was deliberately kept simple, is based on a 1D fully-convolutional network using causal convolutions. In these convolutions an output at time t_n is computed using only output from time $t_{n-x} \dots t_n$ thus respecting the autoregressive property. Bai further uses *dilated convolutions* to increase the receptive field of the model, as can be seen in figure 3. With dilated convolutions a dilation factor d introduces a fixed gap between the filter measurements, improving how far back the model can look when computing the output for the current time step.

Benefits to using TCNs compared to RNN are increased parallelization, low memory-requirements and truly stable gradients. This allows for the training of large models with increased performance over RNNs.

One drawback of TCNs however is that the amount of history considered by the model is fixed in place by the amount of layers and the dilation factor of the model. Different tasks may require longer or shorter lookback distances, and it remains an open research question how to determine the optimal distance for a complex task or one requiring multiple prediction horizons.

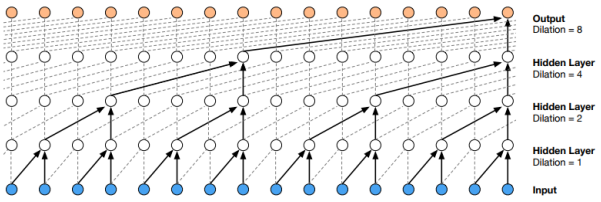


Figure 3. Visualization of stacked dilated convolution (Van den Oord et al. [46], figure 3)

C. Explainable Machine Learning

We already mentioned the dilemma with using most state-of-the-art ML techniques. On the one hand they outperform classical methods on a variety of complex problems and by a significant margin. On the other hand they are hard to interpret.

To put it differently these models are *black boxes*. We know the inputs and outputs of the model but know very little about what operations the model performs to transform the one into the other.

This raises various challenges. Whereas previously with statistical approaches we would have to explicitly model the underlying distribution of the data, thus formalizing our assumptions on its generative mechanism, with neural networks we are often allowed to skip this step, thus weakening our ability to improve our understanding of a problem by looking at the fitted model. Samek et al. [47] provide further motivations for lifting the lid on these black box models.

In many settings, be it medical, financial or industrial, models need to be able to generate explanations for their decisions so that these can be verified by those who act on them. External measures for confidence into the models decisions are also necessary for making risk assessments. Furthermore, when improving an existing model it is helpful to be able to discern why a model failed under certain conditions.

Finally, we are also seeing a trend towards legal requirements on explainability for self-learning models. For example, under the European Union's General Data Protection Regulation (GDPR) [48], data subjects have a "right to explanation" on how decisions by automated systems, without human intervention, are reached [49].

The DARPA XAI program [50] is one of many recent attempts at developing ML techniques that maintain the high performance of current state-of-the-art self-learning models while also making these models more explainable. We recommend their exhaustive literature review [51] for an overview of XAI research.

There are also ongoing efforts for developing tools that allow us to extract new insights from existing black box models. These developments have led to the creation of a large vocabulary used to differentiate the different methods developed in recent years, see Table I in Appendix A for an overview of some common distinctions between XAI techniques.

We believe our focus should be on model-specific, global explanations. These explanations should be interpretable by non-experts of the model with a background in the functioning of the artefact itself.

Care must be taken when attempting to explain a neural network by its learned representations as there may be a near infinite number of instantiations leading to the same performance range on a specific problem. RNNs, for example, are Turing complete [52] and can, given infinite resources, learn an arbitrary transformation function. Feed-forward neural networks can also learn a wide range of continuous functions [53].

Lipton et al. [54] warn for blindly accepting post-hoc model explanations which do not explain the action mechanisms of a black box model but attempt, through various other methods, to explain the predictions and outputs. Plausible yet misleading explanations can be constructed by such methods if not rigorously applied.

D. Transformers

Neural attention mechanisms in ML were inspired in part by the human visual system [55], in which salient details are

extracted from an image not through distilling increasingly complex features from abstract low-level descriptors (as in the encodings of a CNN), but by dynamically allocating importance through an attention mechanism. While already seeing limited use for machine translation [41], Xu et al. [56] improved the then state-of-the-art on multiple image captioning datasets in 2015 using attention mechanism. This opened the door for widespread adoption of attention mechanisms in neural networks.

Xu also made the distinction between "hard", stochastic and "soft", deterministic attention. In stochastic attention only one region of interest is attended to at a time, which results in a non-differentiable model that requires a reinforcement learning approach to be trained. This approach of only using a select region of the input correlates the most with our understanding of attention in the human visual system. Whilst for deterministic attention we can use the expectation of the *context vector*, consisting of representations of the input and surrounding data, as proposed by Bahdanau et al [41].

Deterministic attention can be understood as approximately optimizing the marginal likelihood over all possible regions of interest. This results in a differentiable model, but computation may be expensive when the input is large. Deterministic attention alleviates the problem of distance, whether temporal or spatial, between input components relevant to the output confounding predictions. It realizes this by learning a weighting with which it combines the inputs of all previous states for constructing the output of the model. The output at state t_x is thus no longer solely dependant on the previous state t_{x-1} and its representation of past states (i.e. the memory). Instead, with attention we construct the output at state t_x using a function that combines the inputs of states t_{x-n} up to state t_{x-1}. By using such a function the model is no longer restricted by how much information it can store in its memory, effectively giving the model a global receptive field. However, as mentioned before this does come at the cost of additional computational and space requirements.

The Transformer Architecture

Attention was already well established as a component of recurrent neural architectures when Vaswani et al. took this one step further when they introduced the Transformer network [4] in their aptly named paper "Attention Is All You Need". The Transformer does not use recurrence *or* convolution and is a solely attention-based network.

At its introduction the model reached a new state-of-the-art in a variety of language sequence modeling tasks while allowing for more parallelization than competing RNN models, resulting in faster training of larger models.

Transformers are based on the concept of *self-attention*[3] in which the model learns to compute relations for different positions of an input sequence in order to construct a representation of that sequence. This is different from regular attention mechanisms in which the general features to attend to are usually computed on other examples from the datasets.

Like its competitor neural sequence modelers Transformers use an *encoder-decoder* structure, where the first part of the network encodes an input sequence into an internal representation, which the second part of the network decodes into the required output.

Care must be taken to preserve the auto-regressive property in the decoding part of transformers: we cannot allow the global receptive field of the attention mechanism to incorporate data from time points ahead of the current time when generating output predictions. By masking the input for subsequent positions we can prevent this from happening.

Whereas regular attention is often applied solely to allow the decoder in an encoder-decoder architecture to choose which parts of the input of the encoder it should attend to, self-attention can be used internally by both the encoder and decoder to hierarchically refine their respective outputs.

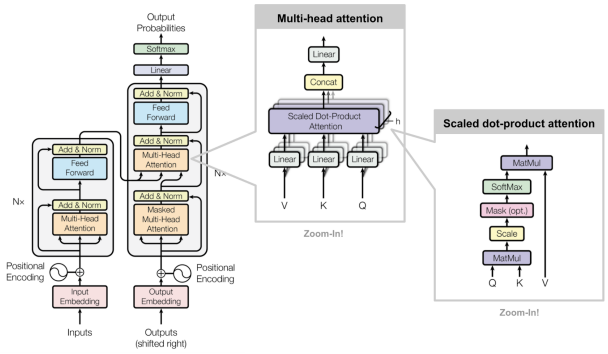


Figure 4. An overview of the canonical Transformer architecture (Vaswani et al. [4] figure 1 & 2)

To implement neural attention mechanisms an alignment score function needs to be specified. This function should quantify to what degree the input at position x_i matches the output at position y_t. Vaswani et al. use a "Scaled Dot-Product" alignment function. First three matrices Q, K and V are constructed for respectively the *queries*, *keys* and *values*. We can think of these matrices as in the context of an information retrieval system where we want to map the queries against known attributes of our data (i.e. the keys), to get the best matching outputs, the values.

In the Transformer the queries are the output of the encoder part of the network for the target sequence (i.e. the sequence of predictions the network has already made), whilst the keys and values are the output of the encodings of the source sequence (i.e. the sequence of input data). In Scaled Dot-Product attention the attention weight is calculated by a dot-product of the queries with the keys. This corresponds to finding the aligning embeddings in Q and K. The resulting matrix is then scaled through a division by the square root of the dimensionality of the source sequence hidden state. This score is then put into a softmax function followed by a multiplication with the values, V, returning the most probable values according to the network.

[3] Sometimes alternatively called intra-attention.

The scaling factor mentioned previously ensures that the softmax function does not return extremely small gradients when the source dimensionality is large.

In the Transformer this attention function is calculated multiple times in parallel in each attention block. According to Vaswani et al. "Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions". The entire procedure can be summarized using the following equations:

$$\begin{aligned}
\text{alignmentScore}(Q,K) &= \frac{Q \cdot K^T}{\sqrt{d_k}} \\
\text{attention}(Q,K,V) &= \text{softmax}(\text{alignmentScore}(Q,K)) * V \\
\text{multiHeadAttention}(Q,K,V) &= \text{concat}(\text{attention}_{H1}(Q,K,V), \ldots, \text{attention}_{Hi}(Q,K,V))
\end{aligned}$$

Figure 4 provides an overview of the Transformer architecture at multiple levels of detail. The leftmost block of the model corresponds to the encoder, where each layer consists of a multi-head attention block and a fully connected feed forward network. In between these two there is a residual connection, which gets added to the output of the sublayers followed by *layer normalization* [57]. In the block to the right, the decoder, we use the previous predictions of the model as input to the first multi-head attention block, taking care to mask future positions. The output of the encoder, Q and K, is then used in concert with the V from the previous block for a second attention block to combine the source and output representations.

Since the Transformer does not use convolutions or recurrence Vaswani et al. needed a different way to encode information about the relative and absolute position of the points in the sequence. They use *positional encodings*, of the same dimension as the model and the embeddings so that the two can later be summed. After some experimentation Vaswani et al. decided on using a sinusoidal function so that the model extrapolates to sequence lengths longer than the ones encountered during training.

Transformers and Explainability

The widespread adoption of attention mechanism for sequence modeling was not only motivated by increased performance but also by widely held beliefs that attention could be directly interpreted as an explanation of a models behaviour. This misconception can be found in the early work of Xu et al. [56] on using attention for computer vision up to more recent work by Humpreys et al. [58] who used the self attention matrix of a Transformer encoder trained for software defect prediction as an explanation for token importance.

In their paper "Attention is not Explanation" Jain et al. [59] show that direct interpretation of a standard additive attention module[4] in a bidirectional LSTM does not yield meaningful explanations.

When using counterfactual examples, where a different attention distribution is used keeping other parameters of the model constant, one would expect the output of the network to change accordingly[5]. Jain et al. show that there are many attention distributions yielding widely different interpretations that nevertheless result in similar model predictions. Furthermore, they show that attention weights, in general, do not consistently correlate with gradient based measures of feature importance or differences in model output when using a leave-one-out training strategy[6].

Brunner et al. [60] conducted a thorough analysis of attention-based explainability in Transformer networks. They looked at the *identifiability* of attention weights. Investigating to what degree the model learns stable attention representations that can be found across different instantiations of the model. They also studied the identifiability of input/output relations, the token identifiability. Brunner et al. have shown that when the input sequence is longer than the attention head dimension, attention weights are unidentifiable. Furthermore, the subsequent attention layers in Transformers strongly mix the representation of the token with its context leading to decreasing token identifiability with layer depth. In general however, the original input token continues to provide the highest individual contribution across layers. These constraints on identifiability in Transformers provide further warnings that (self-)attention cannot be directly interpreted for explainability as originally shown in visualizations of attention by, amongst others, Vaswani et al. [4].

There have been recent attempts however at providing alternative tools for interpreting Transformers and their attention representations. Brunner et al. introduced the concept of *effective attention* which is the part of the attention weight that actually modifies the output. It can be computed by projecting out the non-trivial null space from the attention weights which results from the input sequence length being larger than the number of attention heads. Brunner et al. further introduce *Hidden Token Attribution* a general, gradient-based method which can be used to investigate how the model builds context vectors by mixing inputs.

Furthermore, Van Aken et al. [61] have shown that we can interpret, qualitatively, the hidden states of the network at different layers to get an idea of how the model processes inputs. Their results seem to indicate that different layers of the Transformer are fine-tuned to perform specific tasks.

III. Related Work

To the best of our knowledge our model is the first application of the Transformer architecture to multivariate industrial anomaly detection. However, Transformers have already been applied for MTS forecasting by li et al. [62] and for multi-horizon forecasting by lim et al. [63]. By trending forecasts against expected behaviour one could perform anomaly detection and as such our work is not without precedent.

Surveying existing literature for recent anomaly detection implementations using Transformers we found Mäkinen's The-

[4]The authors report comparable results when using a scaled-dot attention module

[5]Unless both distributions would represent equally plausible explanations

[6]Using a Kendall τ metric, see the original paper for a discussion on the limitations of this approach.

sis [64] wherein he performs anomaly detection in Linux system logs. He compared the performance of a LSTM, GRU, TCN and a Transformer model, with the Transformer outperforming its competitors. His work was based on interpreting human-readable system logs however, which differs significantly from the sensor data we use to predict anomalies.

He et al. [65] also performed semi-supervised anomaly detection on MTS data using a TCN with a very similar approach to ours.

Transformers are not the only architecture using attention in current use for sequence modelling. Shih et al. [66] forecast MTS data using an LSTM with an attention mechanism for selecting relevant time-invariant features, building on previous work by Lai et al. [43] who used a hybrid RNN/CNN model that allowed forecasting on MTS data with hundreds of time series. By transforming the data to be time-invariant using learned filters Shih et al. allow for their attention mechanism to select the most interesting variable, instead of the most interesting time step.

When using MTS we need to keep a clear distinction between the signals of the different variables across time. Selecting a time step and averaging the values of different variables would worsen the signal to noise ratio as unnecessary signals that do not contribute to the prediction are included as well. We hypothesize that their methods also translates well to multivariate sensor data.

Wang et al. employ a different approach to MTS forecasting, using a hybrid AE-TCN model [33] which compares favorably against AE-LSTM hybrids.

Child et al. [67] introduced multiple tweaks to the canonical Transformer to reduce the memory footprint and allow for the training of deeper models. Using their Sparse Transformer they can model sequences with a length in excess of ten thousand time steps.

LogSparse Transformers introduced by Li. et al. [62] further reduce memory requirements and introduce *causal convolutions*, allowing local context aware matching of shapes when generating keys and queries, improving the models performance on MTS forecasting tasks.

The "Reformer" introduced by Kitaev et al. [68] is another performance improvement over the original Transformer, using hashing techniques and other optimalizations to change its complexity from $O(L^2)$ to $O(L \log L)$ where L is the input sequence length.

Totaro et al. [69] introduce a learnable softmax which can be plugged into attention modules and improves performance on time series forecasting benchmarks.

Finally Lim et al. [63] introduced the Temporal Fusion Transformer, an attention based model for multi-horizon forecasting that is able, through its attention mechanisms, to visualize patterns both persistent and abrupt.

IV. Methodology

A. Approach

Leveraging the Transformer for industrial anomaly detection requires modifying an architecture originally designed for sequence to sequence translation and natural language inputs, to use temporal sensor data on a binary or multilabel classification task. We will be attempting to detect, based on a sequence of sensor readings, whether or not an anomaly occurred during a specific time window, and judging which sensors were recording anomalous behaviour. We have opted for a semi-supervised approach where we first learn to model the signals under their "normal" conditions so that we may then predict future timesteps, to which we can compare the actual measured signal at that time. By limiting how large the difference between the reference signal and actual signal may be, we can mark a subset of the data points to be anomalous.

In this approach care must be taken to ensure that the reconstructed signal represents the overall patterns of the normal data correctly and without overfitting, which would otherwise lead to a high recall yet low precision. The threshold itself can be chosen to minimize misclassifications or alternatively, learned when a suitable amount of labeled data is available.

The benefit of this approach lies in that it allows for application to use cases where labeled failure data is not readily available, and collection not feasible.

For time series data, RNNs and especially LSTMs are popular architectures for learning to reconstruct signals. As a baseline by which to compare our model we have chosen to implement an LSTM encoder-decoder similar to the ones used by Malhotra et al. for anomaly detection [70] and remaining useful life prediction [71] in industrial settings.

When forecasting sequences there are two general approaches: one can generate output for multiple timesteps at once, also known as multi-step prediction, or one can predict a single time step at a time and incorporating the predicted value when making subsequent predictions. This is also known as a recursive prediction strategy.

Malhotra et al. compared these two prediction strategies for LSTMs on a variety of anomaly detection tasks [70]. He concluded that, for predictable time series without manual interventions and/or unmonitored environmental conditions, a traditional layered LSTM architecture [72] predicting one step ahead proved to be most successful. In a more unpredictable setting however, where users can intervene with the functioning of the artefact, a encoder-decoder model reconstructing the entire sequence outperformed the recursive approach [70]. Since for our use case user interventions are a possibility, and since forecasting multiple outputs at once eases access to possible explanatory attention weights, we chose to implement a multistep prediction strategy.

Our approach differs from that by Malhotra et al. in that our models does not just reconstruct the current and past data. Instead, it is trained to forecast a sequence of future timesteps. When enough labeled anomalous and normal data is available to train this model in a supervised setting, after it has learned to forecast these sequences, it can then not only perform anomaly detection but also predict whether or not anomalies will occur in the future. This would allow for predictive maintenance where the modelled artefact can be serviced before an anomaly occurs. When labeled data is unavailable, or as a means to

evaluate the model's prediction, one can still compare the reconstruction error when the actual measurements for the predicted future timesteps are taken.

We performed two experiments to compare the performance of a Transformer and LSTM on this task. In the first experiment we trained variations of these models on synthetic data generated using randomly sampled parameters to determine the correct configuration of these networks for different input sizes and numbers of signals.

We then compared anomaly detection performance of the best performing initializations of these two models. We compared performance of two different anomaly detection approaches using a Transformer in order to judge the feasibility of explaining model decisions using self-attention. We performed anomaly detection as a binary multilabel classification task, where we decide for each sensor, whether anomalous data was recorded over the prediction window. In a second step we also evaluate the models performance in detecting at which specific timesteps, for each sensor, whether an anomaly occurred.

In the second experiment we apply an LSTM and Transformer to perform binary anomaly detection, judging whether one or more anomalies occurred in a data sequence, on real world gas turbine data.

B. Data

As is often the case with anomaly detection tasks [23], data for normal behaviour is easy to obtain whereas data for anomalous behaviour, which is often even difficult to define, is hard to collect. This also holds for our use case.

Due to the high reliability requirements on gas turbines, machine failures are very infrequent events compared to normal operation. OPRA Turbines has data available from experiments they ran using a test engine where artificial anomalies with a known root cause were introduced (see figure 12 in Appendix A). However, due to the expensive and time-consuming nature of these experiments it is not feasible to create a suitably large experimental dataset, with a wide variety of anomalies and root causes, that allows for a standard supervised deep learning solution.

To complement the experimental data there are a number of turbines in current operation from which sensor data can be collected. However, this data may be affected by confounding variables outside of our control, such as usage patterns varying by the different operators, maintenance decisions or operation climate.

The experimental data which was used for this project was collected by OPRA engineers in order to test how quickly the on-board system of the turbine would recognize an ignition failure. To guarantee the ignition would fail the gas flow was cut, ensuring no fuel was present during ignition.

The motivation for this experiment was a single field unit experiencing issues where it took multiple ignition attempts for the generator to start successfully, which can be a costly setback for the gas turbine operator.

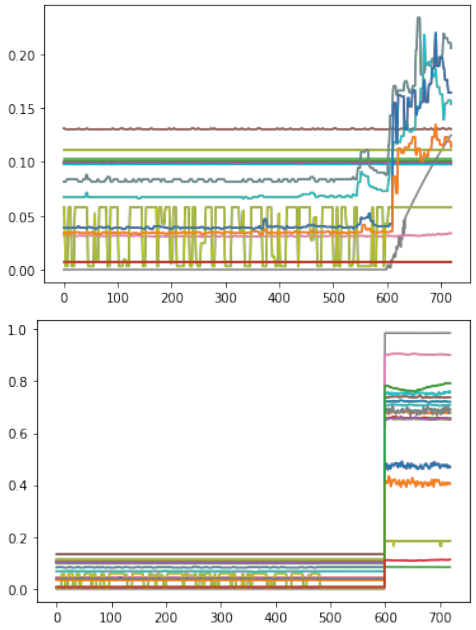


Figure 5. Normalized and dimensionless gas turbine data. Top: failed ignition attempt. Bottom: successful ignition attempt.

Two experiments were conducted: for the first experiment we have measurements for eighty seven signals and for the second experiment ninety six signals. We refer to figure 5 for a sample of relevant normalized sensor measurements for a successful and failed ignition attempt. In general, if the rotor speed, increases to the standard operational number of rotations per minute the ignition was deemed successful.

By adding the data of both experiments together we have 10 successful and 17 unsuccessful ignitions under experimental conditions. Next to this small pool of experimental data, we augmented our use case dataset with additional data from an operational generator in the field. This resulted in a hybrid controlled/uncontrolled pool of data to which we applied our models. In total we were able to extract 114 ignition attempts of which 52 were successful.

In order to properly train and compare the performance of deep learning models we may need more than the above mentioned number of cases, which is why decided to use additional synthetic data. By creating our own data we could also control the complexity and ensure the correctness of labeling.

We based our synthetic data on abstractions for the different sensors from the experimental dataset. Sixteen signals were picked as inspiration for our synthetic data based on experiences from OPRA engineers, with an additional four signals (generator power & gearbox, bearing house and generator vibration) being added by the authors due to perceived relevance. For an overview of these signals see table II.

We respected the varying complexities, scales and known/unknown dependencies of signals when generating data. However, we did not extract direct features from the

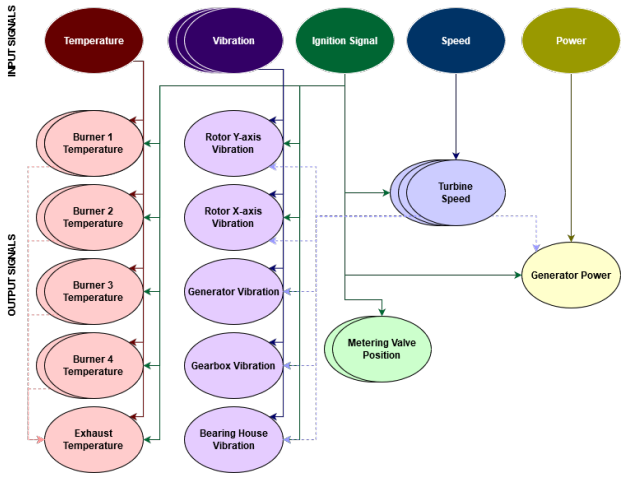


Figure 6. Overview of the internal and external dependencies between the generated synthetic signals

real world data to construct our synthetic data. Evaluating our models on more general MTS data allows us to infer how likely they are to generalize well outside of our specific ignition use case.

Similarly to the experimental data we have four blocks of two sensors representing the temperature measurements of the gas burners, one for the exhaust temperature, five signals for vibration measurements, one signals each for power output, rotor speed and two signals for the valve positions. Each of these signals was modeled using linear combinations and multiplications of sinusoids of varying complexity, with randomly sampled frequency and amplitude values being drawn from predefined ranges.

A varying amount of Gaussian noise was added to each signal based on their type. The input to these signals was provided by eight additional sinusoid signals representing the changing ignition signal, speed and power settings, environmental temperature and (three independent) vibration components.

The external dependencies between the input and output signals and the internal dependencies between the input signals themselves can be seen in Figure 6. External dependencies were resolved by a weighted multiplication of the input and ouput signals, whereas internal dependencies consist of a weighted additive or subtractive effect of an output signal modulating a different output signal.

When generating synthetic data we allow for anomalies of varying length, ranging from quick spikes of a few timesteps to anomalies lasting for the entire time window studied. An anomaly can result in both a positive or negative change to the original signal, with the possible range of change being a predetermined parameter. Multiple signals can face anomalies, and these anomalies can overlap. In Appendix B we discuss in detail how our data was generated and how it can be reproduced.

C. Models

We tested two types of neural sequence forecasting models: an LSTM encoder-decoder and a Transformer. Our non-stateful LSTM forecasts the signal by first constructing an internal representation of the input through a bottleneck. After the bottleneck this representation is turned into a prediction for the next n timesteps using linear dense layers. Inputs are normalized on a per sensor basis to a $[-1, 1]$ range.

In our experiments we first searched for the best number of layers and neurons per layer for different input sequence and prediction lengths. We tested whether reversing the prediction order improved performance, attempting to replicate results by Sutskever et al. [73]. However, we found this decreased model performance for our use case and returned to predicting sequences chronologically.

Due to the limited time and computational power at our disposal we had to fix certain hyperparameters in place during our experiments. After some initial experimentation we decided to use the Rectified Adam (RAdam) optimizer introduced by Liu et al. [74], as it resulted in faster and more stable training of our models. By using this optimizer we also did not need to manually reduce the learning rate at plateaus. We used a learning rate setting of $1\mathrm{e}{-4}$ with a warmup proportion of 0.1 and a minimum learning rate of $1\mathrm{e}{-7}$.

For the LSTM layers we used a dropout rate of 0.1 and tanh activation. We also used early stopping with best weight restoration with a minimal delta of 0.001 and a patience of 4 epochs to speed up the training of multiple models, by stopping training once the model has converged to a (local) optima. Each model was trained for a maximum of 300 epochs, however all models in our experiments converged before this point. An overview of our LSTM encoder-decoder can be found in figure 7.

Our Transformer consists of two modules or steps. The encoder is used to forecast a number of timesteps, which can already be used to detect anomalies in a semi-supervised manner. We then use a second Transformer module, for which we use as input the attention weights from the encoder, and to which we concatenate the difference between the forecasted signal and the measured signal. This Transformer then learns to discern anomalous from normal sequences using labeled data. The benefit to this setup is that it could allow for analysis of the attention weights to gain insight into both the sequence forecasting and anomaly detection components of the model. Inputs for our Transformer are also normalized.

For fine-tuning of our Transformer model we tested the number of blocks (consisting of a multi-head attention layer followed by a feedforward layer), the hidden state size and the number of attention heads. Due to the Transformer model being more complex, with a larger number of parameters, we use a larger dropout rate of 0.2. We again used RAdam for the optimizer and early stopping with similar parameters as for the LSTM models.

We trained both our models using batch sizes of 32 samples. For more details regarding the models' implementations we refer to appendix C.

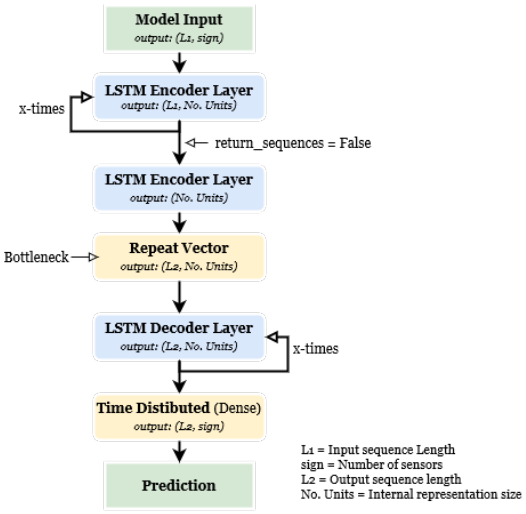


Figure 7. Model architecture for the LSTM encoder-decoder

D. Application to Synthetic Data

We tested our models on the synthetic data with input lengths of either 300 or 600 samples and with either 20 or 40 signals. In the 40 signal setting we sampled a second, independent, version for each sensor from the original synthetic dataset. These additional sensors represent adding another copy of an already existing sensor, measuring the same aspect of the artefact.

Adding duplicate sensors reduces the chance of us having no reference to normal sensor behaviour when an anomaly is added to a specific sensor. This provides additional contextual information that should make it easier for a model to detect an anomalous sensor. However, the trade-off being a larger imbalance between positive and negative cases and requiring larger models to process the data.

For both the 20 and 40 signal conditions we also have the model forecast the ignition signal, which determines overall behaviour. This signal represents operator behaviour influencing system performance. Modeling this behaviour should in theory give the model more information to learn how the different sensors interact and respond to the changes induced by operators of the artefact. This input signal does not contain anomalies in our experiments. It is also excluded from the performance metric calculations, which are performed on the output signals.

In the first sensor anomaly experiment we compared the normalized root mean squared reconstruction error (NRMSE) of LSTMs and Transformers with different initializations. For this experiment we used three different random seeds to generate the synthetic datasets on which we trained our models. We generated 2^{13} training sequences and 2^{12} sequences both for the validation and test set. In the second experiment we further perform anomaly detection on differently sized forecasting windows, of either 10-, 30- or 50% of the input sequence length. Since we trained fewer models in this round we could use a larger dataset. We used 2^{14} training sequences, 2^{12} validation and test sequences and 2^{13} anomaly detection sequences.

The models were trained on the test set, with early stopping monitoring the validation loss. After training, the model were evaluated on the test set to determine the maximum reconstruction error which was then used to classify the anomaly detection sequences. Of the anomaly detection sequences 90% contained at least one anomalous signal. For further details regarding the exact parameterization of the synthetic data, we refer to Appendix B.

We tested two different anomaly detection methods, per signal and then per timestep. First we performed semi-supervised anomaly detection, using the LSTM and Transformer. We found the maximum reconstruction error per sensor for the forecasting phase on the test set, in which only normal behaviour sequences are used, and then compare these to the reconstruction error during the anomaly detection phase, in which a mixture of normal and abnormal sequences are forecasted by the model. We labeled a sequence as anomalous when one of the sensors reconstruction error exceeds the threshold. We then calculated the weighted F1 score, accuracy score and Hamming loss based on these predictions. The reported mean F1 score is the average score across the sensors. For the 2-step Transformer, we further performed supervised anomaly detection using the data labels, training this second module with k-fold cross validation to ensure more stable results.

E. Application to Real World Use Case

For the real world use case we trained and evaluated our models using leave-one-out cross validation, due to the limited amount of available data. We trained on input sequences of 600 timesteps, corresponding to 60 seconds worth of sensor measurements. We then forecasted the next 120 timesteps, or 12 seconds worth of data. Thirty-four sensors were identified by OPRA engineers as being possible indicators for a failed ignition attempt.

Anomaly detection was performed by taking the maximum reconstruction NRMSE per sensor from the training phase and comparing that to the reconstruction error on the test sequence. If one or more of the sensors on the the test sequence exceeded this threshold the sequence was labeled as anomalous (i.e. a failed ignition attempt).

V. Results

A. Synthetic data

We first trained a series of LSTM models to discern the best network depth (4 or 6 layers) and number of hidden units (256, 512 or 1024) for different input lengths and numbers of sensors. We then trained multiple Transformer networks to determine the best number of attention-feedforward blocks (1 or 2), embedding size (512 or 1024 neurons) and number of attention heads (16 or 32). We evaluated these models on input sequences of either 300 or 600 steps with either 21 or 41 sensors. We compared the models based on their average test NRMSE reconstruction error across sensors. The

final scores are an aggregation of the results on the three different synthetic data sets. As can be seen in Figure 8, the best Transformer model configurations outperformed the best LSTM configurations for each of these four conditions, with a 19.4% lower NRMSE on average. For the full results we refer to tables III, IV, V and VI in Appendix A.

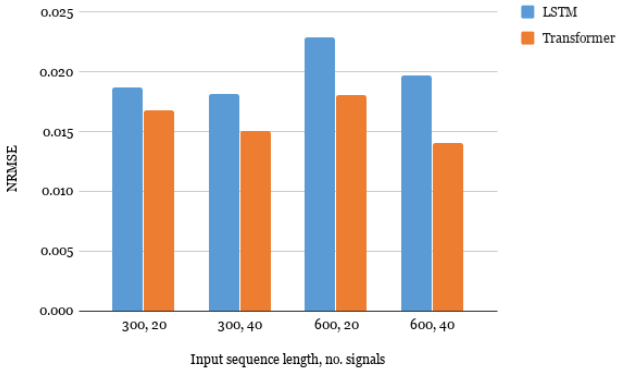


Figure 8. Mean NRMSE reconstruction error of best performing model initializations across the three synthetic datasets

We then used the best model configurations for each condition to perform anomaly detection. For both input sequence lengths and number of sensors we tested differently sized forecasting windows of 10-, 30- or 50% of the input size, training one LSTM and Transformer for each condition. At this stage we also trained the 2-step Transformer to perform supervised anomaly detection.

The second part of the Transformer has the same configuration in terms of blocks, embedding size and attention heads as the encoder. To increase the stability of the results obtained from training this second model we used 4-fold cross validation on the test set. We trained the second Transformer submodule using weighted binary cross-entropy, weighing the less frequent anomalous cases twice during calculation of the loss to counteract class imbalance. We calculated the overall anomaly detection F1 scores, both on a per sensor and per timestep basis. These results can be found in Figure 9. We can see that the Transformer clearly outperforms both the LSTM model and the 2-step Transformer, when detecting anomalies on a per sensor basis. However, for the per timestep anomaly detection the LSTM and Transformer perform similarly. For the full results we refer to tables VIII & IX in Appendix A.

B. Real World Use Case

For the real world use case we again tested a variety of LSTM models (using either 2 or 4 layers with 16, 32, 64, 128 or 256 units) and Transformers (consisting of 1 or 2 blocks with an embedding size of 32, 64, 128 or 256 neurons and 16 or 32 attention heads). We chose to use smaller model sizes for the application to the real world data, as we had far fewer sequences on which to train our models.

Due to the poor performance of our 2-step Transformer on the synthetic data, where it achieved a far lower average

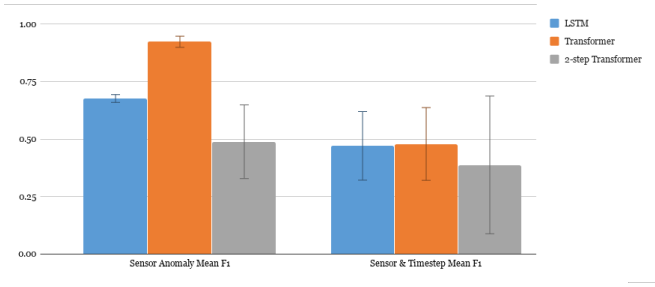


Figure 9. Overall anomaly detection F1 scores on synthetic data, the error bars show the standard deviation

F1 score than its competitors, we did not evaluate it on this dataset. Additionally, the low amount of available data would not allow for the training of another model component in addition to our regular Transformer, as this would require splitting the already small amount of data into two separate datasets for semi-supervised and supervised training.

On the real world data our best performing LSTM model achieved a reconstruction NRMSE of 0.654 and F1 score of 0.286 whereas our best performing Transformer model achieved an NRMSE of 0.210 and F1 score of 0.840. The full results can be found in tables X & XI. Note that for the LSTM models the changing number of layers and units has little to no effect on the reconstruction error and anomaly detection score.

VI. Discussion

Our results, on both the real and synthetic data, indicate that a Transformer is at least as suited for sequence forecasting and anomaly detection as an LSTM encoder-decoder model.

On the synthetic datasets the Transformer outperformed the LSTM for each condition, both in reconstruction error and F1 score on a per sensor basis. For annotating timesteps however, both the LSTM and Transformer models tested in this thesis performed poorly and with high variance, indicating the need for further model refinements.

This performance gap can partly be explained by the extreme class imbalances when classifying on a per timestep basis. Furthermore, the design of our synthetic anomalies, which increasingly deviate from, and then (in some cases) return to normal behaviour, makes it hard for our models to determine the exact starting and stopping point of an anomaly. Due to the class imbalance, misclassifying these border timesteps has a large effect on the F1 score.

This problem may be partly resolved by letting the model output the probability of a timestep being anomalous, instead of performing binary classification. This could be achieved by using a softmax function for the output layers. By then using a performance metric that does not overly penalize misclassifying anomaly shapes, we may more fairly evaluate these models. In real world industrial use cases, it is often more helpful to have an anomaly detection model determine the approximate start time and duration of an anomaly with

high recall, than be extremely precise with the temporal shape of the anomaly.

Overall, the second Transformer module performed worse than the simple anomaly detection schemes. This could be due to errors being propagated across the modules. Letting the 2-step Transformer access the attention weights of the Transformer encoder and concatenating the difference between the predicted and actual sensor measurements does not seem to provide the model with the required information to detect anomalies. More complex architectural designs and training procedures may need to be developed if interpretation of the attention weights for anomaly detection is desired.

In terms of our experimental design, it could have been beneficial to performance to use beam search [75] to determine optimal hyperparameters, fixing fewer options in place. Training more models would also have allowed us to say, with statistical confidence, whether a Transformer outperforms LSTMs on this anomaly detection task and whether there is a significant effect of varying the amounts of sensors and prediction window lengths. With more available labeled data on which to train models, we could also have evaluated the anomaly prediction aspect of our approach. However, these changes to our experimental design would require additional computational power and time, as even the relatively small models tested in this thesis were time-consuming to train (see Appendix C for the hardware used for training our models).

Industry is often hard-pressed to keep up with requirements in terms of data quantity and quality of new models developed by academia. However, by application of our approach to the OPRA use case we have shown how Transformer based anomaly detection is viable, even when the available amount of data is relatively small. This can be contrasted with the LSTM model, which failed to converge sufficiently on the real world data in our experiments, being unable to accurately forecast the modeled signals.

VII. Conclusions

In this thesis, we have shown how the Transformer architecture can be leveraged for industrial anomaly detection on MTS data. We have shown that a Transformer is a viable substitute for an LSTM model on this task. Using our model we achieved a respectable weighted F1 score of 0.84 on the gas turbine ignition use case.

Future research may look into how we can generate explanations of model behaviour by analyzing self-attention, study whether our method of forecasting instead of reconstructing signals can be leveraged for anomaly prediction, and how our anomaly detection model could be integrated into the machine health monitoring component of a DT. The current iteration of our anomaly detection model is not yet suited for use by industry, according to the interpretability and robustness standards we laid out in this thesis.

For application of our approach to industry, it would be helpful to determine if it is possible to use transfer learning [76] or simulate data similar to the use case to pretrain models. This could help circumvent the data bottleneck while investments from industry bring data quantity and quality up to speed.

In conclusion, further research into operationalizing ML for industry is required, if we wish to bring about the advantages in operational efficiency these techniques can offer us.

VIII. Acknowledgements

I would like to express my gratitude to my primary supervisor, Paolo Pileggi, for his guidance during all phases of this research project. I would also like to thank my internal supervisor, Umut Güçlü, for his help with the organizational details. Finally, I am grateful for the helpful feedback provided by Elena Lazovik and Ron Snijders from TNO and Sietse Drost and Lars-Uno Axelsson from OPRA who reviewed this thesis.

References

[1] T. H.-J. Uhlemann, C. Lehmann, and R. Steinhilper, "The digital twin: Realizing the cyber-physical production system for industry 4.0," *Procedia Cirp*, vol. 61, pp. 335–340, 2017.

[2] F. Tao, H. Zhang, A. Liu, and A. Y. Nee, "Digital twin in industry: State-of-the-art," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2018.

[3] Q. Qi and F. Tao, "Digital twin and big data towards smart manufacturing and industry 4.0: 360 degree comparison," *Ieee Access*, vol. 6, pp. 3585–3593, 2018.

[4] A. Vaswani, G. Brain, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in neural information processing systems*, no. Nips, pp. 5998–6008, 2017.

[5] L. U. Axelsson, V. Singh, W. Visser, R. Braun, B. A. Gmbh, J. Strasse, and D. Aachen, "an Innovative Fleet Condition Monitoring Concept for a 2Mw Gas Turbine," no. October, pp. 1–10, 2018.

[6] A. Holzinger, "From machine learning to explainable ai," in *2018 World Symposium on Digital Intelligence for Systems and Machines (DISA)*, pp. 55–66, IEEE, 2018.

[7] K. Gade, S. C. Geyik, K. Kenthapadi, V. Mithal, and A. Taly, "Explainable ai in industry," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3203–3204, 2019.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] E. H. Glaessgen and D. S. Stargel, "The digital twin paradigm for future NASA and U.S. Air force vehicles," *Collection of Technical Papers - AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, pp. 1–14, 2012.

[10] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital Twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[11] D. Kiritsis, A. Bufardi, and P. Xirouchakis, "Research issues on product lifecycle management and information tracking using smart embedded systems," *Advanced Engineering Informatics*, vol. 17, no. 3-4, pp. 189–202, 2003.

[12] M. W. Grieves, "Product lifecycle management: the new paradigm for enterprises," *International Journal of Product Development*, vol. 2, no. 1-2, pp. 71–84, 2005.

[13] W. Liu, Y. Zeng, M. Maletz, and D. Brisson, "Product lifecycle management: a survey," in *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. 1213–1225, American Society of Mechanical Engineers, 2009.

[14] F. Tao, J. Cheng, Q. Qi, M. Zhang, H. Zhang, and F. Sui, "Digital twin-driven product design, manufacturing and service with big data," *International Journal of Advanced Manufacturing Technology*, vol. 94, no. 9-12, pp. 3563–3576, 2018.

[15] S. Takata, F. Kirnura, F. J. van Houten, E. Westkamper, M. Shpitalni, D. Ceglarek, and J. Lee, "Maintenance: changing role in life cycle management," *CIRP annals*, vol. 53, no. 2, pp. 643–655, 2004.

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

[17] R. Fujimaki, T. Yairi, and K. Machida, "An approach to spacecraft anomaly detection problem using kernel feature space," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pp. 401–410, 2005.

[18] C. Feng, T. Li, and D. Chana, "Multi-level anomaly detection in industrial control systems via package signatures and lstm networks," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 261–272, IEEE, 2017.

[19] J. Yang, C. Zhou, S. Yang, H. Xu, and B. Hu, "Anomaly detection based on zone partition for security protection of industrial cyber-physical systems," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 5, pp. 4257–4267, 2017.

[20] D. Ramotsoela, A. Abu-Mahfouz, and G. Hancke, "A survey of anomaly detection in industrial wireless sensor networks with critical water system infrastructure as a case study," *Sensors*, vol. 18, no. 8, p. 2491, 2018.

[21] L. Martí, N. Sanchez-Pi, J. M. Molina, and A. C. B. Garcia, "Anomaly detection based on sensor data in petroleum industry applications," *Sensors*, vol. 15, no. 2, pp. 2774–2797, 2015.

[22] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial intelligence review*, vol. 22, no. 2, pp. 85–126, 2004.

[23] N. R. Prasad, S. Almanza-Garcia, and T. T. Lu, "Anomaly detection," *Computers, Materials and Continua*, vol. 14, no. 1, pp. 1–22, 2009.

[24] R. Wang, K. Nie, T. Wang, Y. Yang, and B. Long, "Deep Learning for Anomaly Detection," no. January, pp. 894–896, 2020.

[25] S. Faghih-Roohi, S. Hajizadeh, A. Núñez, R. Babuska, and B. De Schutter, "Deep convolutional neural networks for detection of rail surface defects," in *2016 International joint conference on neural networks (IJCNN)*, pp. 2584–2589, IEEE, 2016.

[26] D. Lee, V. Siu, R. Cruz, and C. Yetman, "Convolutional neural net and bearing fault analysis," in *Proceedings of the International Conference on Data Mining (DMIN)*, p. 194, The Steering Committee of The World Congress in Computer Science, Computer . . . , 2016.

[27] N. N. Thi, N.-A. Le-Khac, *et al.*, "One-class collective anomaly detection based on lstm-rnns," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXVI*, pp. 73–85, Springer, 2017.

[28] L. Banjanovic-Mehmedovic, A. Hajdarevic, M. Kantardzic, F. Mehmedovic, and I. Dzananovic, "Neural network-based data-driven modelling of anomaly detection in thermal power plant," *Automatika: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije*, vol. 58, no. 1, pp. 69–79, 2017.

[29] Y. Yuan and K. Jia, "A distributed anomaly detection method of operation energy consumption using smart meter data," in *2015 International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, pp. 310–313, IEEE, 2015.

[30] R. Chalapathy and S. Chawla, "Deep Learning for Anomaly Detection: A Survey," no. January, 2019.

[31] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014.

[32] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[33] Y. Wang and D. Hu, "Multivariate Time Series Prediction Based on Optimized Temporal Convolutional Networks with Stacked Autoencoders," pp. 157–172, 2019.

[34] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, pp. 4–11, 2014.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[36] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Phys. Rev. Lett.*, vol. 59, pp. 2229–2232, Nov 1987.

[37] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowlege-Based Systems*, 1998.

[38] S. Hochreiter and J. Urgen Schmidhuber, "Long Shortterm Memory," *Neural Computation*, vol. 9, no. 8, p. 17351780, 1997.

[39] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[40] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[41] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[42] S. Bai, J. Z. Kolter, and V. Koltun, "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling," 2018.

[43] G. Lai, W. C. Chang, Y. Yang, and H. Liu, "Modeling long- and short-term temporal patterns with deep neural networks," *41st International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2018*, no. July, pp. 95–104, 2018.

[44] R. Zhao, J. Wang, R. Yan, and K. Mao, "Machine health monitoring with LSTM networks," *Proceedings of the International Conference on Sensing Technology, ICST*, pp. 1–6, 2016.

[45] C. Lea, R. Vidal, A. Reiter, and G. D. Hager, "Temporal convolutional networks: A unified approach to action segmentation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9915 LNCS, pp. 47–54, 2016.

[46] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[47] W. Samek, T. Wiegand, and K.-R. Müller, "Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models," 2017.

[48] "Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation)."

[49] M. Goddard, "The eu general data protection regulation (gdpr): European regulation that has a global impact," *International Journal of Market Research*, vol. 59, no. 6, pp. 703–705, 2017.

[50] D. Gunning, "Explainable artificial intelligence (xai)," *Defense Advanced Research Projects Agency (DARPA), nd Web*, vol. 2, 2017.

[51] S. T. Mueller, R. R. Hoffman, W. J. Clancey, A. Emrey, and G. Klein, "Explanation in human-ai systems: A literature meta-review, synopsis of key ideas and publications, and bibliography for explainable AI," *CoRR*, vol. abs/1902.01876, 2019.

[52] H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," in *Proceedings of the fifth annual workshop on Computational learning theory*, pp. 440–449, 1992.

[53] B. C. Csáji *et al.*, "Approximation with artificial neural networks," *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 24, no. 48, p. 7, 2001.

[54] Z. C. Lipton, "The mythos of model interpretability," *Queue*, vol. 16, no. 3, pp. 31–57, 2018.

[55] H. Larochelle and G. Hinton, "Learning to combine foveal glimpses with a third-order Boltzmann machine," in *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010, NIPS 2010*, 2010.

[56] K. Xu, J. Ba, R. Kiros, K. Cho, R. Courville, Aaron Salakhudinov, R. Zemel, and Y. Bengio, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," in *International conference on machine learning*, vol. 37, pp. 2048–2057, 2015.

[57] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[58] J. Humphreys and H. K. Dam, "An explainable deep model for defect prediction," *Proceedings - 2019 IEEE/ACM 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2019*, pp. 49–55, 2019.

[59] S. Jain and B. C. Wallace, "Attention is not explanation," *CoRR*, vol. abs/1902.10186, 2019.

[60] G. Brunner, Y. Liu, D. Pascual, O. Richter, M. Ciaramita, and R. Wattenhofer, "On Identifiability in Transformers," pp. 1–35, 2019.

[61] B. Van Aken, A. Löser, B. Winter, and F. A. Gers, "How does BERT answer questions? A layer-wise analysis of transformer representations," *International Conference on Information and Knowledge Management, Proceedings*, pp. 1823–1832, 2019.

[62] S. Li, X. Jin, Y. Xuan, X. Zhou, W. Chen, Y.-X. Wang, and X. Yan, "Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting," no. NeurIPS, 2019.

[63] B. Lim, S. O. Arik, N. Loeff, and T. Pfister, "Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting," 2019.
[64] M. S. Mäkinen, "Deep Learning for Anomaly Detection in Linux System Log," Master's thesis, 2019.
[65] Y. He and J. Zhao, "Temporal Convolutional Networks for Anomaly Detection in Time Series," *Journal of Physics: Conference Series*, vol. 1213, no. 4, 2019.
[66] S. Y. Shih, F. K. Sun, and H. yi Lee, "Temporal pattern attention for multivariate time series forecasting," *Machine Learning*, vol. 108, no. 8-9, pp. 1421–1441, 2019.
[67] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating Long Sequences with Sparse Transformers," 2019.
[68] N. Kitaev, Ł. Kaiser, and A. Levskaya, "Reformer: The Efficient Transformer," pp. 1–12, 2020.
[69] S. Totaro, A. Hussain, and S. Scardapane, "A non-parametric softmax for improving neural attention in time-series forecasting," *Neurocomputing*, vol. 381, pp. 177–185, 2020.
[70] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "LSTM-based Encoder-Decoder for Multi-sensor Anomaly Detection," 2016.
[71] P. Malhotra, V. TV, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Multi-Sensor Prognostics using an Unsupervised Health Index based on LSTM Encoder-Decoder," 2016.
[72] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, "Long Short Term Memory networks for anomaly detection in time series," *23rd European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2015 - Proceedings*, no. April, pp. 89–94, 2015.
[73] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 4, no. January, pp. 3104–3112, 2014.
[74] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," *arXiv preprint arXiv:1908.03265*, 2019.
[75] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.
[76] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

APPENDICES

Appendix A: Additional Tables & Figures

Terms	Definition
Interpretability vs. Explainability	In XAI literature interpretability is a term originating from early work with expert systems. These systems could output the specific rules that led to their decisions as an explanations of their behaviour, but these were often incomprehensible to those without understanding of the systems design. We will use interpretability as a condition for a subclass of explanations that can be used by non-experts of the system.
Justifications vs. Explanations	A justification is an argument for *why* a specific decision was made in contrast with explanations which can also be a visualization of the systems knowledge or representations at a specific state.
Model-agnostic or Model-specific	A model-agnostic method for explanation will work for any type of model whereas model-specific approaches leverage certain aspects of a model for generating explanations.
Local vs. Global Explanations	A local explanation explains why a decision was made for a particular case whereas global explanations give us insight into how the system works generally
Explanation Goodness & Satisfaction	Some criteria for a good explanation are: striking the right balance between detail and volume, being relevant to the goals of the explanation request, having only a single (correct) interpretation. For a more complete discussion see Mueller et al. [51] table 6.2.
Contrastive Reasoning	In contrastive reasoning explanations are generated by highlighting the difference between two cases.

Table I
A VOCABULARY FOR DISCUSSING XAI METHODS

Sensor Configuration	**Description**	**Unit**
4x2 sensors	Burner Temperatures	°C
1 sensor	Exhaust Temperature	°C
3 sensors	Rotor Speed	rpm
2 sensors	Ignition ON Commands	-
1 sensor	Rotor X-axis Vibration	μm
1 sensor	Rotor Y-axis Vibration	μm
2 sensor	Metering Valve Position	%-open
1 sensor	Generator Power	kW
1 sensor	Gearbox Vibration	mm/s
1 sensor	Bearing House Vibration	mm/s
1 sensor	Generator Vibration	mm/s

Table II
SENSORS USED AS INSPIRATION FOR THE CREATION OF THE SYNTHETIC DATA

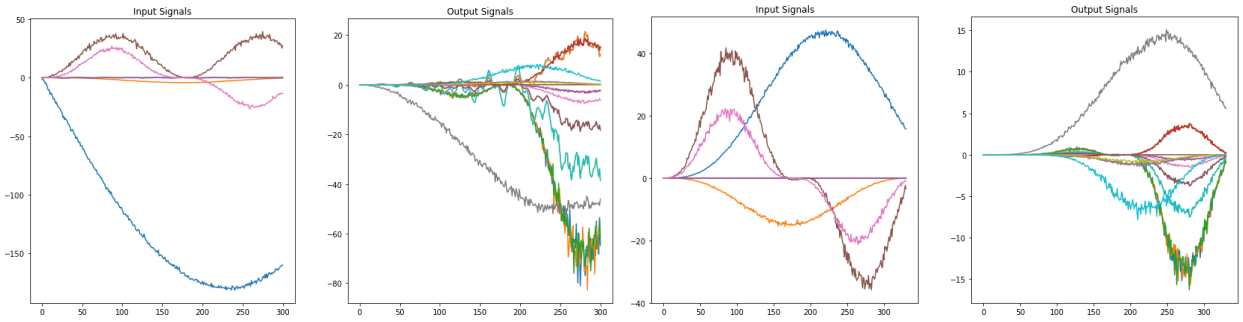


Figure 10. Two samples of synthetic data. Note how the input signals modify the output signals.

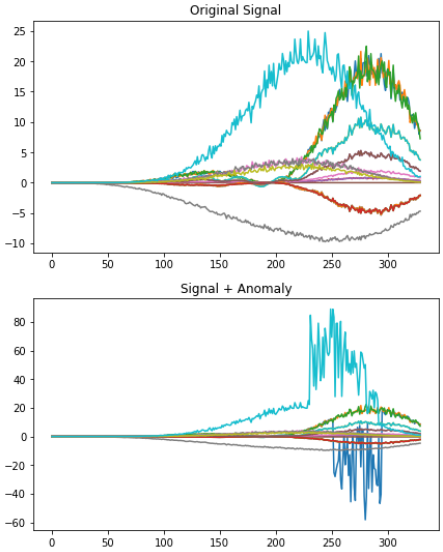


Figure 11. Example of an anomaly added to the synthetic data

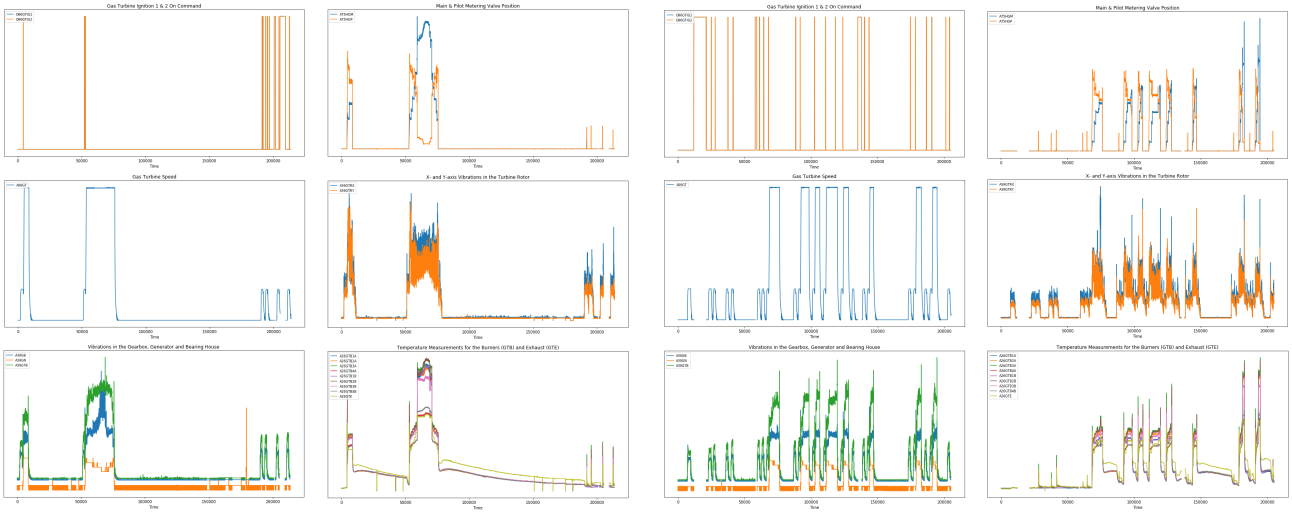


Figure 12. An overview of the sensor data collected under experimental conditions from the controlled ignition attempt experiments

Parameters:		Input Length 300; 20 Sensors					Input Length 300; 40 Sensors				
No. LSTM layers	**No. Units**	**Seed 1**	**Seed 2**	**Seed 3**	**Mean**	**Variance**	**Seed 1**	**Seed 2**	**Seed 3**	**Mean**	**Variance**
4	256	0.02780	0.02012	0.01956	0.02249	0.0000212	0.02020	0.01925	0.02250	0.02065	0.0000028
4	512	**0.02116**	**0.01722**	**0.01757**	**0.01865**	**0.0000048**	0.02383	0.02005	**0.01736**	0.02041	0.0000106
4	1024	0.02198	0.02065	0.02008	0.02090	0.0000010	0.01760	0.02324	0.01866	0.01983	0.0000090
6	256	0.02354	0.02056	0.02786	0.02399	0.0000135	0.03238	0.02847	0.02366	0.02817	0.0000191
6	512	0.02334	0.02338	0.02120	0.02264	0.0000016	**0.01759**	**0.01752**	0.01923	**0.01811**	**0.0000009**
6	1024	0.02255	0.01950	0.02202	0.02136	0.0000027	0.01674	0.01848	0.01937	0.01820	0.0000018

Table III
NORMALIZED RMSE RESULTS FOR HYPERPARAMETER TUNING OF LSTM FOR INPUT SEQUENCE LENGTH OF 300

Parameters:		Input Length 600; 20 Sensors					Input Length 600; 40 Sensors				
No. LSTM layers	No. Units	Seed 1	Seed 2	Seed 3	Mean	Variance	Seed 1	Seed 2	Seed 3	Mean	Variance
4	256	0.02942	0.02213	0.02498	0.02551	0.0000135	0.02237	0.01889	0.03345	0.02490	0.0000578
4	512	0.02862	0.01683	0.03058	0.02534	0.0000553	0.03105	0.0183	0.02371	0.02435	0.0000410
4	1024	0.02922	**0.01634**	0.02318	**0.02291**	**0.0000415**	0.02169	0.01823	**0.02277**	0.02090	0.0000056
6	256	0.02653	0.01903	0.02755	0.02437	0.0000216	0.02363	0.01905	0.03131	0.02466	0.0000384
6	512	**0.02550**	0.01759	0.02807	0.02372	0.0000298	**0.01818**	0.01682	0.02398	**0.01966**	**0.0000145**
6	1024	0.03089	0.01917	**0.02186**	0.02397	0.0000377	0.02215	**0.01494**	0.02381	0.02030	0.0000222

Table IV
NORMALIZED RMSE RESULTS FOR HYPERPARAMETER TUNING OF LSTM FOR INPUT SEQUENCE LENGTH OF 600

Parameters:			Input Length 300; 20 Sensors					Input Length 300; 40 Sensors				
No. blocks	Hidden Size	No. Heads	Seed 1	Seed 2	Seed 3	Mean	Variance	Seed 1	Seed 2	Seed 3	Mean	Variance
1	512	16	0.01973	0.01783	0.02429	0.02062	0.0000110	0.02907	0.01901	0.01424	0.02077	0.0000573
1	512	32	0.01980	**0.01490**	0.02376	0.01949	0.0000197	0.02104	0.01380	0.01544	0.01676	0.0000144
1	1024	16	**0.01745**	0.01781	0.01699	0.01742	0.0000002	**0.01761**	**0.01261**	0.01493	**0.01505**	**0.0000063**
1	1024	32	0.01762	0.01552	0.01727	**0.01680**	**0.0000013**	0.02659	0.01436	0.01422	0.01839	0.0000504
2	512	16	0.01981	0.01623	0.01847	0.01817	0.0000033	0.02972	0.01846	0.01555	0.02124	0.0000560
2	512	32	0.02367	0.01971	0.01645	0.01994	0.0000131	0.02176	0.01953	**0.01201**	0.01777	0.0000261
2	1024	16	0.02153	0.02222	0.01798	0.02058	0.0000052	0.02320	0.01546	0.01430	0.01765	0.0000234
2	1024	32	0.01872	0.02185	**0.01548**	0.01868	0.0000101	0.01970	0.01336	0.01230	0.01512	0.0000160

Table V
NORMALIZED RMSE RESULTS FOR HYPERPARAMETER TUNING OF TRANSFORMER FOR INPUT SEQUENCE LENGTH OF 300

Parameters:			Input Length 600; 20 Sensors					Input Length 600; 40 Sensors				
No. blocks	Hidden Size	No. Heads	Seed 1	Seed 2	Seed 3	Mean	Variance	Seed 1	Seed 2	Seed 3	Mean	Variance
1	512	16	0.02636	0.01512	0.02258	0.02135	0.0000327	0.01571	0.01398	0.01757	0.01575	0.0000032
1	512	32	0.02345	0.01578	0.02454	0.02126	0.0000228	0.01627	0.01378	0.01641	0.01549	0.0000022
1	1024	16	0.02461	0.01641	**0.01925**	0.02009	0.0000173	0.01557	**0.01359**	**0.01299**	**0.01405**	**0.0000018**
1	1024	32	0.02119	0.01535	0.02094	0.01916	0.0000109	**0.01431**	0.01611	0.01712	0.01585	0.0000020
2	512	16	0.01954	0.02272	0.01951	0.02059	0.0000034	0.02445	0.01441	0.01653	0.01846	0.0000280
2	512	32	0.02077	0.01570	0.01990	0.01879	0.0000074	0.02586	0.01388	0.01664	0.01879	0.0000394
2	1024	16	**0.01922**	**0.01466**	0.02027	**0.01805**	**0.0000089**	0.01751	0.01390	0.02100	0.01747	0.0000126
2	1024	32	0.02280	0.01501	0.01934	0.01905	0.0000152	0.02331	0.01418	0.02026	0.01925	0.0000216

Table VI
NORMALIZED RMSE RESULTS FOR HYPERPARAMETER TUNING OF TRANSFORMER FOR INPUT SEQUENCE LENGTH OF 600

Parameters			NRMSE		Weighted Cross Entropy
No. Signals	No. Input Samples	Prediction length	LSTM	Transformer	2-step Transformer
20	300	10%	0.01681	**0.01268**	0.09497
20	300	30%	0.03103	**0.02540**	0.08033
20	300	50%	0.04963	**0.03398**	0.08234
20	600	10%	0.01572	**0.01472**	0.15657
20	600	30%	**0.02444**	0.03613	0.13161
20	600	50%	**0.03202**	0.04442	0.12767
40	300	10%	0.01447	**0.01061**	0.05934
40	300	30%	0.02610	**0.01698**	0.06179
40	300	50%	0.03565	**0.02201**	0.06243
40	600	10%	0.01336	**0.01072**	0.15505
40	600	30%	0.04372	**0.02176**	0.12623
40	600	50%	0.05558	**0.02407**	0.11488

Table VII
NORMALIZED RMSE AND WEIGHTED CROSS ENTROPY RESULTS FOR MODEL EVALUATION ON THE SECOND EXPERIMENT

Parameters			F1 signal			Accuracy score signal			Hamming loss signal		
No. Signals	No. Input Samples	Prediction length	LSTM	Transformer	2-step Transformer	LSTM	Transformer	2-step Transformer	LSTM	Transformer	2-step Transformer
20	300	10%	0.68603	**0.94645**	0.45923	0.62154	**0.95193**	0.38284	0.37846	**0.04807**	0.61716
20	300	30%	0.70537	**0.91183**	0.37755	0.63975	**0.92629**	0.32131	0.36025	**0.07371**	0.67869
20	300	50%	0.67245	**0.91586**	0.42749	0.60453	**0.92876**	0.35876	0.39547	**0.07124**	0.64124
20	600	10%	0.67205	**0.90836**	0.71745	0.62748	**0.92417**	0.65452	0.37252	**0.07583**	0.34548
20	600	30%	0.68773	**0.89530**	0.68568	0.64702	**0.91694**	0.61667	0.35298	**0.08306**	0.38333
20	600	50%	0.70095	**0.88276**	0.66427	0.65718	**0.90796**	0.59551	0.34282	**0.09204**	0.40449
40	300	10%	0.68367	**0.96376**	0.35741	0.61477	**0.96649**	0.30209	0.38523	**0.03351**	0.69791
40	300	30%	0.66754	**0.95570**	0.27686	0.59450	**0.96067**	0.24897	0.40550	**0.03933**	0.75103
40	300	50%	0.66522	**0.93994**	0.28582	0.59331	**0.94801**	0.25316	0.40669	**0.05199**	0.74684
40	600	10%	0.64713	**0.92088**	0.68079	0.58764	**0.93545**	0.60840	0.41236	**0.06455**	0.39160
40	600	30%	0.65365	**0.91885**	0.49390	0.61692	0.93442	**0.95810**	0.38308	**0.06558**	0.04190
40	600	50%	0.67275	**0.91232**	0.43369	0.62999	**0.93020**	0.35662	0.37001	**0.06980**	0.64338

Table VIII
PER SIGNAL ANOMALY DETECTION RESULTS ON SYNTHETIC DATA

Parameters			F1 signal x timestep			Accuracy score signal x timestep			Hamming loss signal x timestep		
No. Signals	No. Input Samples	Prediction length	LSTM	Transformer	2-step Transformer	LSTM	Transformer	2-step Transformer	LSTM	Transformer	2-step Transformer
20	300	10%	0.64239	**0.66329**	0.51762	0.93883	**0.96948**	0.96528	0.06117	**0.03052**	0.03472
20	300	30%	0.57035	0.48312	**0.60568**	0.96000	0.95826	**0.96947**	0.04000	0.04174	**0.03053**
20	300	50%	0.39066	0.38961	**0.46468**	0.94308	0.95312	**0.96352**	0.05692	0.04688	**0.03648**
20	600	10%	**0.47739**	0.40647	0.13103	0.94556	**0.95381**	0.94831	0.05444	**0.04619**	0.05169
20	600	30%	**0.33895**	0.28533	0.13835	0.94920	**0.94961**	0.94880	0.05080	**0.05039**	0.05120
20	600	50%	**0.30599**	0.23370	0.12580	0.94538	0.94527	**0.94543**	0.05462	0.05473	**0.05457**
40	300	10%	0.64377	**0.75106**	0.71368	0.94338	0.97712	**0.97756**	0.05662	0.02288	**0.02244**
40	300	30%	**0.69781**	0.66811	0.65459	0.96746	0.97138	**0.97425**	0.03254	0.02862	**0.02575**
40	300	50%	0.50252	0.56812	**0.60244**	0.95997	0.96539	**0.97244**	0.04003	0.03461	**0.02756**
40	600	10%	**0.50232**	0.48418	0.13717	0.94327	**0.96079**	0.95209	0.05673	**0.03921**	0.04791
40	600	30%	0.31531	**0.42849**	0.26747	0.94788	**0.95810**	0.95723	0.05212	**0.04190**	0.04277
40	600	50%	0.25578	**0.38003**	0.29238	0.94493	**0.95650**	0.95817	0.05507	0.04350	**0.04183**

Table IX
PER TIMESTEP ANOMALY DETECTION RESULTS ON SYNTHETIC DATA

LSTM					
No. LSTM layers	No. Units	NRMSE	F1 signal	Accuracy score signal	Hamming loss signal
2	16	0.65784	0.28577	0.45614	0.54386
2	32	0.65821	0.28577	0.45614	0.54386
2	64	0.65757	0.28577	0.45614	0.54386
2	128	0.65902	0.28577	0.45614	0.54386
2	256	0.65765	0.28577	0.45614	0.54386
4	16	0.65724	0.28577	0.45614	0.54386
4	32	0.65740	0.28577	0.45614	0.54386
4	64	0.65819	0.28577	0.45614	0.54386
4	128	0.65745	0.28577	0.45614	0.54386
4	256	**0.65427**	**0.28577**	**0.45614**	**0.54386**

Table X
LSTM ANOMALY DETECTION RESULTS ON REAL WORLD IGNITION DATA

Transformer						
No. Encoder/Decoder blocks	No. Attention Heads	Hidden Size	NRMSE	F1 signal	Accuracy score signal	Hamming loss signal
1	16	32	0.23418	0.67127	0.69298	0.30702
1	16	64	0.21943	0.76448	0.77193	0.22807
1	16	128	0.21687	0.76448	0.77193	0.22807
1	16	256	0.21221	0.84035	0.84211	0.15789
1	32	32	0.22689	0.68216	0.70175	0.29825
1	32	64	0.21884	0.76448	0.77193	0.22807
1	32	128	0.21488	0.76448	0.77193	0.22807
1	32	256	0.21568	0.84035	0.84211	0.15789
2	16	32	0.22514	0.67127	0.69298	0.30702
2	16	64	0.21395	0.69290	0.71053	0.28947
2	16	128	0.21418	0.76448	0.77193	0.22807
2	16	256	**0.20975**	**0.84035**	**0.84211**	**0.15789**

Table XI
TRANSFORMER ANOMALY DETECTION RESULTS ON REAL WORLD IGNITION DATA

Appendix B: Synthetic Data Generation

When generating synthetic data with our script the following parameters need to be specified: the training sequence length (L1), forecasting window length (L2), number of output signals (sign) and the size of the training, validation, test and anomaly detection datasets. A specific generation run can be reproduced by using the same random seed.

We must also decide on the number of input signals to be used. For our synthetic data we used 5 signals representing the ignition program, temperature, speed and power curves and overall vibrations. Each input signal can consist of multiple independent copies, we used 3 copies of the vibration signal (for the x-, y- and z-axis). Then, the complexity of the signal in terms of the number of linear combinations of sine or cosine waves is set. In our case we used more complex signals for hard to model aspects such as vibration, whereas temperature is be represented by a simpler signal with fewer components. We must also determine the ranges from which to sample how much Gaussian noise to add to our signals, and determine from which frequency and amplitude ranges we will sample our signal components. Finally we can set whether we would like to have our signal consists of sine waves, cosine waves or a combination of the two. This process is then replicated for the output signals.

We then move on to defining the input/output dependency matrix, in which we determine by what input signals our output signals are modulated. For example, with our synthetic data all output signals depend on the ignition program, however, the temperature input signal only modulates the output signals representing the gas burner and exhaust temperatures. Next we define the internal dependency matrix, which formalizes which output signals modify other output signals. Since these modification happen sequentially not all combinations are possible. Instead, the changes propagate over time similarly as to how changes to a component early in an industrial pipeline might affect components down the line. For example, with our synthetic data, a change to the signal representing turbine speed affects the amount of vibrations in the rotor.

Having provided all these settings the script can now generate normal data. For the anomalous data further choices need to be made. We must first set the overall anomaly rate, the probability of at least one output signal being anomalous. We can then set the individual anomaly rates per output signal, which represent the probability of something anomalous occurring with that signal. We then set the time ranges in between which anomalies may be added by our script (we use [L, L+L2]). We then set the maximum change in value of the signal an anomaly may enact, for our data we use changes ranging from 50 to 150% percent of the normal value.

The user must now set the probabilities of occurrence for the different modelled anomaly types: ‘constant-lasting’, ‘constant-fleeting’, ‘random-lasting’, ‘random-fleeting’. For the constant anomalies the signal is modified with either a constant increase or decrease sampled from the maximum change ranges. The buildup to this constant value is smoothed across a user defined number of timesteps. If it is a constant-lasting anomaly it will last for the entire remainder of the sequence, whereas if it is constant-fleeting it will smoothly decrease again after a random number of timesteps (the minimum of which must be defined by the user). The random-lasting and random-fleeting anomalies on the other hand are erratic, with no smoothing. For these anomalies we sample new modifiers, which can be positive or negative, at each timestep. Using these settings we can now also generate the anomalous data.

Figure 10 shows two examples of input and output signals from our synthetic dataset. Figure 11 shows an example of anomalies being added to the model outputs.

Our data generation script was built intending for it to be used to study the viability of using self-attention in Transformers to find correlates for the internal and external dependencies. It could also be used to test the effects of different anomaly types, forecasting window lengths and numbers of sensors on prediction performance. However, due to the necessity of scoping the research project this was left to future work.

Appendix C: Notes on Implementation

Our models were implemented in Python (v3.8) using the Tensorflow (v2.2.0) and Keras (v2.3.1) ML libraries. For the Transformer implementation we built on previous work[7] by github user ’huseinzol05’. For our RAdam implementation we used the ’keras-rectified-adam’ package[8] (v0.17.0). We ran the majority of our experiments on the TNO research cloud which has access to a NVIDIA V100 GPU with 32GB of memory. We used NVIDIA CUDA® (v10.1) and cuDNN (v7) for our computations. It took an average of 8 and 12 seconds respectively, to train our best performing LSTM and Transformer models for one epoch on our synthetic data with this setup. A Dockerfile with which to replicate the software used by us for this project is available at our GitHub repository: https://github.com/MJJdG/MTS-Anomaly-Detection-Transformer.

[7]https://github.com/huseinzol05/Stock-Prediction-Models/blob/master/deep-learning/16.attention-is-all-you-need.ipynb, last retrieved on 21/08/2020

[8]https://pypi.org/project/keras-rectified-adam/, last retrieved on 21/08/2020

Appendix D: Code

The complete code for all our experiments is available on GitHub: https://github.com/MJJdG/MTS-Anomaly-Detection-Transformer. Here we reproduce the main code for creating the synthetic data and creating the different LSTM & Transformer models.

Transformer code:

```
#!/usr/bin/env python
# coding: utf-8
"""
Implementation of a Transformer for multivariate time series forecasting and anomaly detection
@author: Max de Grauw (M.degrauw@student.ru.nl)
BUilding upon the base Transformer implementation of github user 'huseinzol05'
https://github.com/huseinzol05/Stock-Prediction-Models/blob/master/deep-learning/16.attention-is-all-you-need.ipynb
"""

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from sklearn.metrics import f1_score, hamming_loss, accuracy_score
from sklearn.model_selection import KFold
import gc

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

from tensorflow.keras.utils import HDF5Matrix
from keras_radam.training import RAdamOptimizer

from MTS_utils import Scaler3DMM

def layer_norm(inputs, epsilon=1e-8):
    mean, variance = tf.nn.moments(inputs, [-1], keep_dims=True)
    normalized = (inputs - mean) / (tf.sqrt(variance + epsilon))

    params_shape = inputs.get_shape()[-1:]
    gamma = tf.get_variable('gamma', params_shape, tf.float32, tf.ones_initializer())
    beta = tf.get_variable('beta', params_shape, tf.float32, tf.zeros_initializer())

    outputs = gamma * normalized + beta
    return outputs

def multihead_attn(queries, keys, q_masks, k_masks, future_binding, num_units, num_heads):

    T_q = tf.shape(queries)[1]
    T_k = tf.shape(keys)[1]

    Q = tf.layers.dense(queries, num_units, name='Q')
    K_V = tf.layers.dense(keys, 2*num_units, name='K_V')
    K, V = tf.split(K_V, 2, -1)

    Q_ = tf.concat(tf.split(Q, num_heads, axis=2), axis=0)
    K_ = tf.concat(tf.split(K, num_heads, axis=2), axis=0)
    V_ = tf.concat(tf.split(V, num_heads, axis=2), axis=0)

    align = tf.matmul(Q_, tf.transpose(K_, [0,2,1]))
    align = align / np.sqrt(K_.get_shape().as_list()[-1])

    paddings = tf.fill(tf.shape(align), float('-inf'))

    key_masks = k_masks
    key_masks = tf.tile(key_masks, [num_heads, 1])
    key_masks = tf.tile(tf.expand_dims(key_masks, 1), [1, T_q, 1])
    align = tf.where(tf.equal(key_masks, 0), paddings, align)

    if future_binding:
        lower_tri = tf.ones([T_q, T_k])
        lower_tri = tf.linalg.LinearOperatorLowerTriangular(lower_tri).to_dense()
        masks = tf.tile(tf.expand_dims(lower_tri,0), [tf.shape(align)[0], 1, 1])
        align = tf.where(tf.equal(masks, 0), paddings, align)

    align = tf.nn.softmax(align)
    query_masks = tf.to_float(q_masks)
    query_masks = tf.tile(query_masks, [num_heads, 1])
    query_masks = tf.tile(tf.expand_dims(query_masks, -1), [1, 1, T_k])
    align *= query_masks

    outputs = tf.matmul(align, V_)
    outputs = tf.concat(tf.split(outputs, num_heads, axis=0), axis=2)
```

```
    outputs += queries
    outputs = layer_norm(outputs)
    return outputs

def pointwise_feedforward(inputs, hidden_units, activation=None):
    outputs = tf.layers.dense(inputs, 4*hidden_units, activation=activation)
    outputs = tf.layers.dense(outputs, hidden_units, activation=None)
    outputs += inputs
    outputs = layer_norm(outputs)
    return outputs

def sinusoidal_position_encoding(inputs, mask, repr_dim):
    T = tf.shape(inputs)[1]
    pos = tf.reshape(tf.range(0.0, tf.to_float(T), dtype=tf.float32), [-1, 1])
    i = np.arange(0, repr_dim, 2, np.float32)
    denom = np.reshape(np.power(10000.0, i / repr_dim), [1, -1])
    enc = tf.expand_dims(tf.concat([tf.sin(pos / denom), tf.cos(pos / denom)], 1), 0)
    return tf.tile(enc, [tf.shape(inputs)[0], 1, 1]) * tf.expand_dims(tf.to_float(mask), -1)

def label_smoothing(inputs, epsilon=0.1):
    C = inputs.get_shape().as_list()[-1]
    return ((1 - epsilon) * inputs) + (epsilon / C)

def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer

def mtsTransformerForecasting(modelID, fn, L1, L2, sign=21, num_blocks=1, size_embedding=512, num_heads=16,
                              batch_size=32, epochs=300, learning_rate =1e-4):

    ################################################################################
    # Read data
    ################################################################################
    print('Loading Data...')
    folder = ''

    print("Signals:",np.array(HDF5Matrix(folder+fn+'.hdf5', 'column_names')))

    #Scale the features using min-max scaler in range 0 -> 1
    sc = Scaler3DMM(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_train')))

    x_train = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_train')))
    y_train = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_train')))
    x_val = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_val')))
    y_val = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_val')))
    x_test = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_test')))
    y_test = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_test')))
    x_anomalous = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_anomalous')))
    y_anomalous = np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_anomalous'))

    ################################################################################
    # Initialize model
    ################################################################################
    tf.reset_default_graph()

    forecasting_model = Forecasting_Attention(size_embedding, learning_rate, x_train.shape[1],\
                                              y_train.shape[2], num_blocks, num_heads, L2)

    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())

    ################################################################################
    # Training loop forecasting
    ################################################################################
    pbar = tqdm(range(epochs), desc = 'train loop')
    prev_rmse, counter = 1e3, 0
    for i in pbar:
        total_rmse = []
        max_rse = np.zeros((L2, sign))
        for k in range(batch_size, x_train.shape[0], batch_size):
            logits, _, rmse, indiv_max_rse = sess.run(
                [forecasting_model.logits, forecasting_model.optimizer,\
                 forecasting_model.rmse, forecasting_model.indiv_max_rse],
                feed_dict = {
                    forecasting_model.X: x_train[k-batch_size:k, :, :],
                    forecasting_model.Y: y_train[k-batch_size:k]
                },
            )
```

```
        total_rmse.append(rmse)
        max_rse += indiv_max_rse
    pbar.set_postfix(rmse = np.mean(total_rmse), max_rse = (max_rse/(x_train.shape[0]/batch_size))[0])

    ################################################################################
    # Evaluate the model on the validation data
    ################################################################################
    val_rmse = 0
    val_indiv_max_rse = np.zeros((x_val.shape[0], L2, sign))
    for l in range(batch_size, x_val.shape[0], batch_size):
        out_logits = sess.run(
            forecasting_model.logits,
            feed_dict = {
                forecasting_model.X: x_val[l-batch_size:l, :, :],
            },
        )
        val_rmse += np.sqrt(np.mean(np.square(y_val[l-batch_size:l] - out_logits)))
        val_indiv_max_rse[l-batch_size:l] = np.sqrt(np.max(np.square(y_val[l-batch_size:l] - out_logits),axis=0))
    print("val_rmse:",val_rmse/(x_val.shape[0]/batch_size))
    print("val_max_rse:", np.max(np.max(val_indiv_max_rse, axis=0), axis=0))
    if prev_rmse-val_rmse/(L2*(x_val.shape[0]/batch_size)) <= .00001: # Early stopping
        counter += 1
    if counter == 3:
        print("Early stopping at epoch",i)
        break
    prev_rmse = val_rmse/(L2*(x_val.shape[0]/batch_size))

################################################################################
# Evaluate the model on the test data
################################################################################
pbar = tqdm(range(1), desc = 'test loop')
for i in pbar:
    test_rmse = 0
    test_indiv_max_rse = np.zeros((x_test.shape[0], L2, sign))
    for k in range(batch_size, x_test.shape[0], batch_size):
        out_logits = sess.run(
            forecasting_model.logits,
            feed_dict = {
                forecasting_model.X: x_test[k-batch_size:k, :, :],
            },
        )
        test_rmse += np.sqrt(np.mean(np.square(y_test[k-batch_size:k] - out_logits)))
        test_indiv_max_rse[k-batch_size:k] = np.sqrt(np.max(np.square(y_test[k-batch_size:k] - out_logits),axis=0))
    pbar.set_postfix(test_rmse = test_rmse/(x_test.shape[0]/batch_size),\
                     test_max_rse = np.max(np.max(test_indiv_max_rse, axis=0), axis=0))
    thresholds = np.max(test_indiv_max_rse, axis=0)

################################################################################
# Predict future ts using last n days as a start for anomaly detection
################################################################################
x_anompred = np.zeros(x_anomalous.shape)
x_anompred_ffw = np.zeros((x_anomalous.shape[0], L1, size_embedding))
pbar = tqdm(range(batch_size, x_anomalous.shape[0], batch_size), desc = "anom pred")
anom_rmse = 0
anom_indiv_max_rse = np.zeros(x_anomalous.shape)
for k in pbar:
    batch_pred = np.zeros((batch_size, x_anomalous.shape[1], x_anomalous.shape[2]))
    batch_pred[:,:L1,:] = x_anomalous[k-batch_size:k,:L1,:]
    out_logits, out_ffw = sess.run(
        [forecasting_model.logits, forecasting_model.feedforward_outputs],
        feed_dict = {
            forecasting_model.X:batch_pred[:,:L1,:]
        },
    )
    anom_rmse += np.sqrt(np.mean(np.square(x_anomalous[k-batch_size:k,L1:] - out_logits)))
    anom_indiv_max_rse[k-batch_size:k,L1:] = np.sqrt(np.max(np.square(x_anomalous[k-batch_size:k,L1:] - out_logits),
         axis=0))
    batch_pred[:, L1:, :] = out_logits
    x_anompred[k-batch_size:k, :, :] = batch_pred
    x_anompred_ffw[k-batch_size:k, :, :] = out_ffw

print("Mean RMSE anomaly forecasting:", anom_rmse/(x_anomalous.shape[0]/batch_size))
print("Individual Max RSE anomaly forecasting:", np.max(np.max(anom_indiv_max_rse, axis=0), axis=0))

print("Signal anomaly prediction:")
labels = np.amax(y_anomalous[:,L1:,:sign-1], axis=1)
max_rse_prediction = np.sqrt(np.max(np.square(x_anomalous[:,L1:,:sign-1] - x_anompred[:,L1:,:sign-1]),axis=1))
predictions = np.zeros(max_rse_prediction.shape)
for idx, sample in enumerate(max_rse_prediction):
    predictions[idx] = (sample >= [x for x in np.max(thresholds, axis=0)[:sign-1]])

meanF1, mean_accSa, mean_hamLa = 0, 0, 0
for i in range(sign-1):
```

```
            meanF1 += f1_score(labels[:,i], predictions[:,i], average='weighted')
            mean_accSa += accuracy_score(labels[:,i], predictions[:,i])
            mean_hamLa += hamming_loss(labels[:,i], predictions[:,i])
            #print("F1 signal", i, f1_score(labels[:,i], predictions[:,i], average='weighted'))

        meanF1a = meanF1/(sign-1)
        mean_accSa = mean_accSa/(sign-1)
        mean_hamLa = mean_hamLa/(sign-1)

        print("Mean F1 signal",meanF1a)
        print("accuracy_score:", mean_accSa)
        print("Hamming_loss:", mean_hamLa)

        print("Signal x Ts anomaly prediction:")
        labels = y_anomalous[:,L1:,:sign-1]
        max_rse_prediction = np.sqrt(np.square(x_anomalous[:,L1:,:sign-1] - x_anompred[:,L1:,:sign-1]))
        predictions = np.zeros(max_rse_prediction.shape)
        for idx, sample in enumerate(max_rse_prediction):
            for jdx, step in enumerate(sample):
                predictions[idx,jdx,:] = (step >= [x for x in thresholds[jdx,:sign-1]])

        meanF1 = 0
        mean_accSb, mean_hamLb = np.zeros(sign-1), np.zeros(sign-1)
        for i in range(sign-1):
            meanF1 += f1_score(labels[:,:,i], predictions[:,:,i], average='weighted')
            #print("F1 signal", i, f1_score(labels[:,:,i], predictions[:,:,i], average='weighted'))
            for j in range(labels.shape[0]):
                mean_accSb[i] += accuracy_score(labels[j,:,i], predictions[j,:,i])
                mean_hamLb[i] += hamming_loss(labels[j,:,i], predictions[j,:,i])

        meanF1b = meanF1/(sign-1)
        mean_accSb = np.mean(mean_accSb/labels.shape[0])
        mean_hamLb = np.mean(mean_hamLb/labels.shape[0])

        print("Mean F1 signal",meanF1b)
        print("accuracy_score:", mean_accSb)
        print("Hamming_loss:", mean_hamLb)

        ###############################################################################
        # Write results to file
        ###############################################################################
        with open("Transformer_results.txt", "a") as file:
            file.write('test metrics model %s:\n' % modelID)
            file.write("test RMSE: %s\n" % (test_rmse/(x_test.shape[0]/batch_size)))
            for item in np.max(np.max(test_indiv_max_rse, axis=0), axis=0):
                file.write("%s\n" % item)
            file.write('meanF1a: %s\n' % meanF1a)
            file.write("accuracy_score_a: %s\n" % mean_accSa)
            file.write("Hamming_loss_b: %s\n" % mean_hamLa)
            file.write('meanF1b: %s\n' % meanF1b)
            file.write("accuracy_score_b: %s\n" % mean_accSb)
            file.write("Hamming_loss_b: %s\n" % mean_hamLb)

        ###############################################################################
        #Plot first anomalous samples
        ###############################################################################
        for i in range(4):
            f, ([ax1, ax2], [ax3, ax4]) = plt.subplots(2, 2, figsize=(16,10));
            ax1.plot(x_anomalous[i,L1:]);
            ax1.set_title('True Signal');
            ax2.plot(x_anompred[i,L1:]);
            ax2.set_title('Predicted Signal');

            difference = np.sqrt(np.square(x_anomalous[i,L1:]-x_anompred[i,L1:]))
            ax3.plot(difference);
            ax3.set_title('Absolute Difference');
            ax4.plot(y_anomalous[i][L1:,:]);
            ax4.set_title('Anomalies');

        sess.close()

        del forecasting_model, sc, x_train, y_train, x_val, y_val, x_test, y_test, x_anomalous, y_anomalous
        gc.collect()

        return x_anompred, x_anompred_ffw

    def mtsTransformerPrediction(modelID, prediction, encoder_output, fn, L1, L2, sign=21, num_blocks=1,
                                 size_embedding=512, num_heads=16, batch_size=32, epochs=300,
                                 learning_rate =1e-4, loss_weight=False, folds=4):

        ###############################################################################
        # Read data
        ###############################################################################
```

```
print('Loading Data...')
folder = ''

#Scale the features using min-max scaler in range 0 -> 1
sc = Scaler3DMM(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_train')))

x_anomalous = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_anomalous')))
y_anomalous = np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_anomalous'))

################################################################################
# K-fold cross validation
################################################################################

kf = KFold(n_splits=folds, shuffle=False, random_state=None)
fold = 1
fold_meanLoss, fold_meanF1a, fold_meanF1b = 0, 0, 0
fold_mean_accSa, fold_mean_hamLa, fold_mean_accSb, fold_mean_hamLb = 0, 0, 0, 0
for train_index, test_index in kf.split(x_anomalous, y_anomalous):
    print("FOLD: ",fold)
    encoder_train, encoder_test = encoder_output[train_index], encoder_output[test_index]
    prediction_train, prediction_test = prediction[train_index], prediction[test_index]
    X_train, X_test = x_anomalous[train_index], x_anomalous[test_index]
    y_train, y_test = y_anomalous[train_index], y_anomalous[test_index]

    ############################################################################
    # Intialize model
    ############################################################################
    tf.reset_default_graph()

    if loss_weight:
        _, counts = np.unique(y_anomalous, return_counts=True)
        if not isinstance(loss_weight, int): loss_weight = (counts[0]/counts[1])
    else:
        loss_weight = 1

    prediction_model = Prediction_Attention(L1, sign, size_embedding, learning_rate,\
                                            num_blocks, num_heads, batch_size, L2, loss_weight)

    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())

    ############################################################################
    # Training loop
    ############################################################################
    pbar = tqdm(range(epochs), desc = 'anomaly train loop')
    for i in pbar:
        total_loss = []
        for k in range(batch_size, X_train.shape[0], batch_size):
            _, loss = sess.run(
                [prediction_model.optimizer, prediction_model.loss],
                feed_dict = {
                    prediction_model.EncoderOutput: encoder_train[k-batch_size:k],
                    prediction_model.Xtrue: prediction_train[k-batch_size:k, -L2:, :],
                    prediction_model.Xanom: X_train[k-batch_size:k, -L2:, :],
                    prediction_model.Yanom: y_train[k-batch_size:k, -L2:, :]
                },
            )
            total_loss.append(loss)
        pbar.set_postfix(loss = np.mean(total_loss))

    ############################################################################
    # Testing loop
    ############################################################################
    pred = np.zeros((X_test.shape[0], L2, sign-1))
    for k in range(batch_size, X_test.shape[0], batch_size):
        logits, los = sess.run(
            [prediction_model.logits, prediction_model.loss],
            feed_dict = {
                prediction_model.EncoderOutput: encoder_test[k-batch_size:k],
                prediction_model.Xtrue: prediction_test[k-batch_size:k, -L2:, :],
                prediction_model.Xanom: X_test[k-batch_size:k, -L2:, :],
                prediction_model.Yanom: y_test[k-batch_size:k, -L2:, :]
            },
        )
        total_loss.append(loss)
        pred[k-batch_size:k,:,:] = logits[:,:,:sign-1]
    print("Test loss:", np.mean(total_loss))
    fold_meanLoss += np.mean(total_loss)

    ############################################################################
    # Predict and calculate F1-score
    ############################################################################
    print("Signal anomaly prediction:")
```

```
    labels = np.amax(y_test[:,-L2:,:sign-1], axis=1)
    predictions = np.zeros((pred.shape[0], pred.shape[2]))
    for idx, sample in enumerate(np.amax(pred, axis=1)):
        predictions[idx] = np.max(sample) >= 0.5

    meanF1, mean_accSa, mean_hamLa = 0, 0, 0
    for i in range(sign-1):
        meanF1 += f1_score(labels[:,i], predictions[:,i], average='weighted')
        mean_accSa += accuracy_score(labels[:,i], predictions[:,i])
        mean_hamLa += hamming_loss(labels[:,i], predictions[:,i])
        #print("F1 signal", i, f1_score(labels[:,i], predictions[:,i], average='weighted'))

    meanF1a = meanF1/(sign-1)
    mean_accSa = mean_accSa/(sign-1)
    mean_hamLa = mean_hamLa/(sign-1)

    print("Mean F1 signal",meanF1a)
    print("accuracy_score:", mean_accSa)
    print("Hamming_loss:", mean_hamLa)

    fold_meanF1a += meanF1a
    fold_mean_accSa += mean_accSa
    fold_mean_hamLa += mean_hamLa

    print("Signal x Ts anomaly prediction:")
    labels = y_test[:,-L2:,:sign-1]
    predictions = pred >= 0.5

    meanF1 = 0
    mean_accSb, mean_hamLb = np.zeros(sign-1), np.zeros(sign-1)
    for i in range(sign-1):
        meanF1 += f1_score(labels[:,:,i], predictions[:,:,i], average='weighted')
        #print("F1 signal", i, f1_score(labels[:,:,i], predictions[:,:,i], average='weighted'))
        for j in range(labels.shape[0]):
            mean_accSb[i] += accuracy_score(labels[j,:,i], predictions[j,:,i])
            mean_hamLb[i] += hamming_loss(labels[j,:,i], predictions[j,:,i])

    meanF1b = meanF1/(sign-1)
    mean_accSb = np.mean(mean_accSb/labels.shape[0])
    mean_hamLb = np.mean(mean_hamLb/labels.shape[0])

    print("Mean F1 signal",meanF1b)
    print("accuracy_score:", mean_accSb)
    print("Hamming_loss:", mean_hamLb)

    fold_meanF1b += meanF1b
    fold_mean_accSb += mean_accSb
    fold_mean_hamLb += mean_hamLb

    ##############################################################################################
    # Write results to file
    ##############################################################################################
    with open("Transformer_results.txt", "a") as file:
        file.write("Supervised, fold %s:\n" % fold)
        file.write("Test loss: %s\n" % np.mean(total_loss))
        file.write('meanF1a: %s\n' % meanF1a)
        file.write('meanF1b: %s\n' % meanF1b)
        file.write('accSa: %s\n' % mean_accSa)
        file.write('hamLa: %s\n' % mean_hamLa)
        file.write('accSb: %s\n' % mean_accSb)
        file.write('hamLb: %s\n' % mean_hamLb)

    fold += 1
    sess.close()

##############################################################################################
# Write final results to file
##############################################################################################
with open("Transformer_results.txt", "a") as file:
    file.write("Supervised, total\n")
    file.write('Test loss over folds: %s\n' % (fold_meanLoss/folds))
    file.write('meanF1a over folds: %s\n' % (fold_meanF1a/folds))
    file.write('accSa: %s\n' % (fold_mean_accSa/folds))
    file.write('hamLa: %s\n' % (fold_mean_hamLa/folds))
    file.write('meanF1b over folds: %s\n' % (fold_meanF1b/folds))
    file.write('accSb: %s\n' % (fold_mean_accSb/folds))
    file.write('hamLb: %s\n' % (fold_mean_hamLb/folds))

print("\nFinal results supervised:")
print('Test loss over folds: %s' % (fold_meanLoss/folds))
print('meanF1a over folds: %s' % (fold_meanF1a/folds))
print('accSa over folds: %s' % (fold_mean_accSa/folds))
print('hamLa over folds: %s' % (fold_mean_hamLa/folds))
```

```
    print('meanF1b over folds: %s' % (fold_meanF1b/folds))
    print('accSb over folds: %s' % (fold_mean_accSb/folds))
    print('hamLb over folds: %s\n' % (fold_mean_hamLb/folds))

    del prediction_model, sc, x_anomalous, y_anomalous
    gc.collect()

    return pred

class Forecasting_Attention:
    def __init__(self, embedded_size, learning_rate, input_size, output_size,
                 num_blocks, num_heads, future_day=1, min_freq = 50, dropout_rate = 0.8):
        self.X = tf.placeholder(tf.float32, (None, input_size, output_size))
        self.Y = tf.placeholder(tf.float32, (None, future_day, output_size))

        encoder_embedded = tf.layers.dense(self.X, embedded_size)
        encoder_embedded = tf.nn.dropout(encoder_embedded, keep_prob = dropout_rate)
        x_mean = tf.reduce_mean(self.X, axis = 2)
        en_masks = tf.sign(x_mean)
        encoder_embedded += sinusoidal_position_encoding(self.X, en_masks, embedded_size)

        for i in range(num_blocks):
            with tf.variable_scope('encoder_self_attn_%d'%i,reuse=tf.AUTO_REUSE):
                encoder_embedded = multihead_attn(queries = encoder_embedded,
                                                  keys = encoder_embedded,
                                                  q_masks = en_masks,
                                                  k_masks = en_masks,
                                                  future_binding = False,
                                                  num_units = embedded_size,
                                                  num_heads = num_heads)

            if i == num_blocks-1: self.attention_outputs = tf.identity(encoder_embedded)

            with tf.variable_scope('encoder_feedforward_%d'%i,reuse=tf.AUTO_REUSE):
                encoder_embedded = pointwise_feedforward(encoder_embedded,
                                                         embedded_size,
                                                         activation = tf.nn.relu)

            if i == num_blocks-1: self.feedforward_outputs = tf.identity(encoder_embedded)

        encoder_embedded = tf.transpose(tf.layers.dense(encoder_embedded, output_size), perm=[0,2,1])
        self.logits = tf.transpose(tf.layers.dense(encoder_embedded, future_day), perm=[0,2,1])
        self.rmse = tf.sqrt(tf.reduce_mean(tf.square(self.Y - self.logits)))
        self.indiv_max_rse = tf.sqrt(tf.reduce_max(tf.square(self.Y - self.logits), axis=0))
        self.optimizer = RAdamOptimizer(learning_rate=learning_rate,
                                        total_steps=10000,
                                        warmup_proportion=0.1,
                                        min_lr=1e-7).minimize(self.rmse)

class Prediction_Attention:
    def __init__(self, input_size, output_size, embedded_size, learning_rate, num_blocks, num_heads,\
                 batch_size, future_day=1, loss_weight=1, min_freq = 50, dropout_rate = 0.8):
        self.EncoderOutput = tf.placeholder(tf.float32, (batch_size, input_size, embedded_size))
        self.Xtrue = tf.placeholder(tf.float32, (batch_size, future_day, output_size))
        self.Xanom = tf.placeholder(tf.float32, (batch_size, future_day, output_size))
        self.Yanom = tf.placeholder(tf.float32, (batch_size, future_day, output_size))

        #Prepare encoder output
        encoder_embedded = tf.layers.dense(self.EncoderOutput, embedded_size)
        encoder_embedded = tf.nn.dropout(encoder_embedded, keep_prob = dropout_rate)

        # Prepare difference between forecast and actual measurement
        Xdiff = tf.math.sqrt(tf.math.square(tf.math.subtract(self.Xtrue, self.Xanom)))
        anom_embedded = tf.layers.dense(Xdiff, embedded_size)
        anom_embedded = tf.nn.dropout(anom_embedded, keep_prob = dropout_rate)

        # Concatenate
        anom_encoder_embedded = tf.concat([encoder_embedded, anom_embedded], 1)
        x_anom_mean = tf.reduce_mean(anom_encoder_embedded, axis = 2)
        en_anom_masks = tf.sign(x_anom_mean)
        anom_encoder_embedded += sinusoidal_position_encoding(anom_encoder_embedded, en_anom_masks, embedded_size)

        self.attention_outputs = [tf.Variable(np.empty((batch_size, input_size+future_day, embedded_size), dtype=np.
            float32))] * num_blocks
        self.feedforward_outputs = [tf.Variable(np.empty((batch_size, input_size+future_day, embedded_size), dtype=np.
            float32))] * num_blocks
        for i in range(num_blocks):
            with tf.variable_scope('encoder_self_attn_%d'%i,reuse=tf.AUTO_REUSE):
                anom_encoder_embedded = multihead_attn(queries = anom_encoder_embedded,
                                                  keys = anom_encoder_embedded,
                                                  q_masks = en_anom_masks,
                                                  k_masks = en_anom_masks,
                                                  future_binding = False,
```

```
                                                    num_units = embedded_size,
                                                    num_heads = num_heads)

            self.attention_outputs[i] = self.attention_outputs[i].assign(tf.identity(anom_encoder_embedded))

            with tf.variable_scope('encoder_feedforward_%d'%i,reuse=tf.AUTO_REUSE):
                anom_encoder_embedded = pointwise_feedforward(anom_encoder_embedded,
                                                              embedded_size,
                                                              activation = tf.nn.relu)

            self.feedforward_outputs[i] = self.feedforward_outputs[i].assign(tf.identity(anom_encoder_embedded))

        anom_encoder_embedded = tf.transpose(tf.layers.dense(anom_encoder_embedded, output_size), perm=[0,2,1])
        self.logits = tf.transpose(tf.layers.dense(anom_encoder_embedded, future_day), perm=[0,2,1])
        self.loss = tf.nn.weighted_cross_entropy_with_logits(labels=self.Yanom, logits=self.logits, pos_weight=loss_weight
            )
        self.optimizer = RAdamOptimizer(learning_rate=learning_rate,
                                        total_steps=10000,
                                        warmup_proportion=0.1,
                                        min_lr=1e-7).minimize(self.loss)
```

LSTM code:

```
#!/usr/bin/env python
# coding: utf-8
"""
Implementation of an LSTM for multivariate time series forecasting and anomaly detection
@author: Max de Grauw (M.degrauw@student.ru.nl)
"""

from __future__ import print_function
import numpy as np
from sklearn.metrics import f1_score, hamming_loss, accuracy_score
import os

os.environ['TF_KERAS'] = '1'

import tensorflow.keras.backend as K
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, LSTM, TimeDistributed, RepeatVector
from tensorflow.keras.utils import plot_model, HDF5Matrix
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.losses import mean_squared_error

from keras_radam import RAdam

from MTS_utils import Scaler3DMM_tanh

def root_mean_squared_error(y_true, y_pred):
    return K.sqrt(mean_squared_error(y_true, y_pred))

def metricWrapper(m):
    def max_rse(y_true, y_pred):
        return K.sqrt(K.max(K.square(y_true[m,:] - y_pred[m,:]), axis=0))
    max_rse.__name__ = 'max_rse' + str(m)
    return max_rse

def mtsLSTM(modelID, fn, L1, L2, sign=21, noLayers=4, noUnits=512, batch_size=64, epochs=300):
    K.clear_session()

    print('Loading Data')
    folder = ''

    print("Signals:",np.array(HDF5Matrix(folder+fn+'.hdf5', 'column_names')))

    #Scale the features using min-max scaler in range -1 -> 1 for tanh activation
    sc = Scaler3DMM_tanh(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_train')))
    x_train = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_train')))
    y_train = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_train')))
    x_val = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_val')))
    y_val = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_val')))

    # Define encoder
    inputEnc = Input(batch_shape=(batch_size, x_train.shape[1], x_train.shape[2]), name='enc_input')
    encoder = LSTM(noUnits, activation='tanh', return_sequences=True, stateful=False, dropout=0.1)(inputEnc)
    if noLayers > 4:
        for extra_layer in range((noLayers-4)//2):
            encoder = LSTM(noUnits, activation='tanh', return_sequences=True, stateful=False, dropout=0.1)(encoder)
    encoder = LSTM(noUnits, activation='tanh', return_sequences=False, stateful=False, dropout=0.1)(encoder)

    # Define bottleneck
    bottleneck = RepeatVector(L2)(encoder)
```

```
# Define reconstruction decoder
decoder = LSTM(noUnits, activation='tanh', return_sequences=True, stateful=False, dropout=0.1)(bottleneck)
if noLayers > 4:
    for extra_layer in range((noLayers-4)//2):
        decoder = LSTM(noUnits, activation='tanh', return_sequences=True, stateful=False, dropout=0.1)(decoder)
decoder = LSTM(noUnits, activation='tanh', return_sequences=True, stateful=False, dropout=0.1)(decoder)
decoder = TimeDistributed(Dense(sign, activation='linear', name='dec_output'))(decoder)

# Create model and compile
opt = RAdam(learning_rate=1e-4, total_steps=10000, warmup_proportion=0.1, min_lr=1e-7)
model = Model(inputs=inputEnc, outputs=decoder)
model.compile(loss=root_mean_squared_error,
              optimizer=opt,
              metrics=[metricWrapper(m) for m in range(sign)])
plot_model(model, show_shapes=True, to_file='LSTM_AE'+modelID+'.png')
model.summary()

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, min_delta=.00001, patience=3, restore_best_weights=True)

# fit model
print('Training Model...')
history = model.fit(x_train,
                    y_train,
                    epochs=epochs,
                    batch_size=batch_size,
                    validation_data=(x_val, y_val),
                    verbose=1,
                    callbacks=[es],
                    shuffle=True)

# Reduce memory load
del x_train, y_train, x_val, y_val

x_test = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_test')))
y_test = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_test')))

# Evaluate the model on the test data
print('\n# Evaluate on test data')
results = model.evaluate(x_test, y_test, batch_size=batch_size)
print('test metrics:', results)

x_anomalous = sc.transform(np.array(HDF5Matrix(folder+fn+'.hdf5', 'x_anomalous')))
y_anomalous = np.array(HDF5Matrix(folder+fn+'.hdf5', 'y_anomalous'))

x_anompred = np.zeros(x_anomalous.shape)
x_anompred[:,:L1,:] = x_anomalous[:,:L1,:]

for batch in range(x_anompred.shape[0]//batch_size):
    print("Batch:",batch)
    x_anompred[batch*batch_size:(batch+1)*batch_size,L1:,:] = model.predict(x_anompred[batch*batch_size:(batch+1)*
        batch_size,:L1,:], batch_size=batch_size)

labels = np.amax(y_anomalous[:,L1:,:sign-1], axis=1)
max_rse_prediction = np.sqrt(((x_anomalous[:,L1:,:sign-1] - x_anompred[:,L1:,:sign-1]) ** 2)).max(axis=1)
predictions = np.zeros(max_rse_prediction.shape)
for idx, sample in enumerate(max_rse_prediction):
    predictions[idx] = (sample >= [x for x in results[1:sign]])

print("Signal anomaly prediction:")
meanF1, mean_accSa, mean_hamLa = 0, 0, 0
for i in range(sign-1):
    meanF1 += f1_score(labels[:,i], predictions[:,i], average='weighted')
    mean_accSa += accuracy_score(labels[:,i], predictions[:,i])
    mean_hamLa += hamming_loss(labels[:,i], predictions[:,i])
    #print("F1 signal", i, f1_score(labels[:,i], predictions[:,i], average='weighted'))

meanF1a = meanF1/(sign-1)
mean_accSa = mean_accSa/(sign-1)
mean_hamLa = mean_hamLa/(sign-1)

print("Mean F1 signal", meanF1a)
print("accuracy_score:", mean_accSa)
print("Hamming_loss:", mean_hamLa)

labels = y_anomalous[:,L1:,:sign-1]
max_rse_prediction = np.sqrt(((x_anomalous[:,L1:,:sign-1] - x_anompred[:,L1:,:sign-1]) ** 2))
predictions = np.zeros(max_rse_prediction.shape)
for idx, sample in enumerate(max_rse_prediction):
    for jdx, step in enumerate(sample):
        predictions[idx,jdx,:] = (step >= [x for x in results[1:sign]])

print("Signal x Ts anomaly prediction:")
```

```
    meanF1 = 0
    mean_accSb, mean_hamLb = np.zeros(sign-1), np.zeros(sign-1)
    for i in range(sign-1):
        meanF1 += f1_score(labels[:,:,i], predictions[:,:,i], average='weighted')
        #print("F1 signal", i, f1_score(labels[:,:,i], predictions[:,:,i], average='weighted'))
        for j in range(labels.shape[0]):
            mean_accSb[i] += accuracy_score(labels[j,:,i], predictions[j,:,i])
            mean_hamLb[i] += hamming_loss(labels[j,:,i], predictions[j,:,i])

    meanF1b = meanF1/(sign-1)
    mean_accSb = np.mean(mean_accSb/labels.shape[0])
    mean_hamLb = np.mean(mean_hamLb/labels.shape[0])

    print("Mean F1 signal", meanF1b)
    print("accuracy_score:", mean_accSb)
    print("Hamming_loss:", mean_hamLb)

    # Write results to file
    with open("LSTM_results.txt", "a") as file:
        file.write('test metrics model %s:\n' % modelID)
        file.write("test RMSE: %s\n" % results[0])
        for item in results[1:]:
            file.write("%s\n" % item)
        file.write('meanF1a: %s\n' % meanF1a)
        file.write("accuracy_score_a: %s\n" % mean_accSa)
        file.write("Hamming_loss_a: %s\n" % mean_hamLa)
        file.write('meanF1b: %s\n' % meanF1b)
        file.write("accuracy_score_b: %s\n" % mean_accSb)
        file.write("Hamming_loss_b: %s\n" % mean_hamLb)

    return history
```

Data Generation code:

```
# -*- coding: utf-8 -*-
"""
Synthetic Data Generator for Multivariate Time Series (MTS) Data
@author: Max de Grauw (M.degrauw@student.ru.nl)
"""

import numpy as np
from itertools import chain
import matplotlib.pyplot as plt
import h5py
from MTS_utils import Cosine, Sine

def generationPipeline(L, L2, filename, seed=42, Fs=1, sign=20, train_size=2**13,\
                       val_size=2**12, test_size=2**12, anomaly_size=2**12):
    """
    Generates normal and anomalous synthetic MTS data using DataGenerator class

    Parameters:
        L: Training Sequence Length
        L2: Forecasting Window Length
        filename: output file name
        seed: Random seed to use
        Fs: Sampling rate of used signals
        sign: Number of output signals 20 or 40
        train_size: Training set size for forecasting training
        val_size: Validation ,, ,,
        test_size: Test ,, ,,
        anomaly_size = Size of dataset for anomaly prediction

    Returns:
        h5py file containing synthetic data
    """
    np.random.seed(seed)
    if sign != 20 and sign != 40:
        print(sign,"is a nonstandard number of signals, requires manual changes to "+\
              "fixed parameters in data generation script. See code comments for help")
        return False

    ###########################################################################
    # Create datagenerator
    ###########################################################################
    dg = DataGenerator(L, L2, Fs)
    print("Generating data for",sign,"signals, with length of",L)

    ###########################################################################
    # Input & output parameter matrices:
    # 0:    number of copies,
    # 1:    number of wave combinations,
```

```
# 2,0:  [random error ranges ((a->b)/c),
# 2,1:  random frequency ranges,
# 2,2:  random amplitude ranges],
# 3:    whether to use sine and/or cos waves
#########################################################################
inputSigs =  [[1, 3, [[100, 200, 10000], [150,  200, 100000], [500, 1000, 1000]], [1, 1]], # Program
              [1, 3, [[100, 250, 10000], [200,  250, 100000], [150,  200, 1000]], [1, 1]], # Temperature
              [3, 5, [[250, 500, 10000], [500, 1000, 100000], [100,  150, 1000]], [1, 1]], # Vibration
              [1, 4, [[100, 250, 10000], [250,  300, 100000], [300,  350, 1000]], [1, 1]], # Speed
              [1, 4, [[100, 250, 10000], [250,  300, 100000], [250,  300, 1000]], [1, 1]]] # Power

sigfact = 2 if sign == 40 else 1 # Double all sensors if sign == 40
outputSigs = [[sigfact*3, 4, [[100, 250, 10000], [400, 500, 1000000], [300, 350, 1000]], [1, 1]], #GTS
              [sigfact*2, 3, [[100, 250, 10000], [100, 200, 1000000], [150, 200, 1000]], [1, 1]], #MV
              [sigfact*1, 5, [[100, 250, 10000], [500, 600, 1000000], [100, 150, 1000]], [1, 1]], #GTRVX
              [sigfact*1, 5, [[100, 250, 10000], [500, 600, 1000000], [100, 150, 1000]], [1, 1]], #GTRVY
              [sigfact*1, 5, [[100, 250, 10000], [500, 600, 1000000], [100, 150, 1000]], [1, 1]], #GBV
              [sigfact*1, 5, [[100, 250, 10000], [500, 600, 1000000], [100, 150, 1000]], [1, 1]], #BHV
              [sigfact*1, 5, [[100, 250, 10000], [500, 600, 1000000], [100, 150, 1000]], [1, 1]], #GTV
              [sigfact*2, 3, [[100, 250, 10000], [200, 300, 1000000], [250, 300, 1000]], [1, 1]], #TB1
              [sigfact*2, 3, [[100, 250, 10000], [200, 300, 1000000], [250, 300, 1000]], [1, 1]], #TB2
              [sigfact*2, 3, [[100, 250, 10000], [200, 300, 1000000], [250, 300, 1000]], [1, 1]], #TB3
              [sigfact*2, 3, [[100, 250, 10000], [200, 300, 1000000], [250, 300, 1000]], [1, 1]], #TB4
              [sigfact*1, 3, [[100, 250, 10000], [200, 300, 1000000], [200, 250, 1000]], [1, 1]], #TE
              [sigfact*1, 4, [[100, 250, 10000], [300, 400, 1000000], [250, 300, 1000]], [1, 1]]] #GTP

outputNames = np.array(['GTS', 'MV', 'GTRVX', 'GTRVY', 'GBV', 'BHV',\
                        'GTV', 'TB1', 'TB2', 'TB3', 'TB4', 'TE', 'GTP','PROGRAM'])
outputNames = [n.encode("ascii", "ignore") for n in outputNames]

#########################################################################
# Output/input dependencies matrix (size= inputSigs*outputSigs)
#########################################################################
outputDeps = [[1, 0, 0, 1, 0], #GTS
              [1, 0, 0, 0, 0], #MV
              [1, 0, 1, 0, 0], #GTRVX
              [1, 0, 1, 0, 0], #GTRVY
              [1, 0, 1, 0, 0], #GBV
              [1, 0, 1, 0, 0], #BHV
              [1, 0, 1, 0, 0], #GTV
              [1, 1, 0, 0, 0], #TB1
              [1, 1, 0, 0, 0], #TB2
              [1, 1, 0, 0, 0], #TB3
              [1, 1, 0, 0, 0], #TB4
              [1, 1, 0, 0, 0], #TE
              [1, 0, 0, 1, 1]] #GTP

#########################################################################
# Internal dependency matrix (size= outputSigs*outputSigs):
# Signal X_t can only depend on signals t-x ... t-1
#########################################################################
internalDeps =\
    #GTS    MV      GTRVX   GTRVY   GBV     BHV     GTV     TB1     TB2     TB3     TB4     TE      GTP
   [[0,     0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #GTS
    [0,     0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #MV
    [0.5,   0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #GTRVX
    [0.5,   0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #GTRVY
    [0.25,  0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #GBV
    [0.25,  0,      0,      0,   0.25,     0,      0,      0,      0,      0,      0,      0,      0], #BHV
    [0.25,  0,   0.25,   0.25,  0.125,  0.125,      0,      0,      0,      0,      0,      0,      0], #GTV
    [0,     0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #TB1
    [0,     0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #TB2
    [0,     0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #TB3
    [0,     0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0], #TB4
    [0,     0,      0,      0,      0,      0,      0,      0.25,   0.25,   0.25,   0.25,   0,      0], #TE
    [0.25,  0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0,      0]] #GTP

#########################################################################
# Initialize data generator and plot test signals
#########################################################################
dg.setParams(inputSigs, outputSigs, outputDeps, internalDeps, verbose=False)

for i in range(4):
    dg.generate(verbose=True)

#########################################################################
# Create training data
#########################################################################
x_train = np.zeros((train_size, L*Fs, sign+1))
x_val= np.zeros((val_size, L*Fs, sign+1))

y_train = np.zeros((train_size, L2*Fs, sign+1))
y_val = np.zeros((val_size, L2*Fs, sign+1))
```

```
for i in range(0, train_size+val_size):
    if i < train_size:
        x_train[i,:,:], y_train[i,:,:] = dg.generate()
    elif i < train_size+val_size:
        x_val[i-train_size,:,:], y_val[i-train_size,:,:] = dg.generate()

print("Finished creating training dataset",train_size+val_size,"of samples")

###############################################################################
# Create testing data
###############################################################################
x_test = np.zeros((test_size, L*Fs, sign+1))
y_test = np.zeros((test_size, L2*Fs, sign+1))

for i in range(0, test_size):
    x_test[i,:,:], y_test[i,:,:] = dg.generate()

print("Finished creating test dataset",test_size,"of samples")

###############################################################################
# Anomaly parameters:
# 0: Overall anomaly rate
# 1: Individual anomaly rates (size = sign)
# 2: Anomaly time ranges
# 3: Anomaly change ranges
# 4: Probabilities for different anomaly types
# 5: Minimal length of anomaly in percentage of total window (L1->L1+L2)
# 6: Minimal value to start at when smoothing edges of anomaly
###############################################################################
if sign == 20: # Base number of sensors
    # GTS, MV, GTRVX, GTRVY, GBV, BHV, GTV, TB1, TB2, TB3, TB4, TE, GTP
    anomRates = [0.2, 0.2, 0.2,\
                 0.05, 0.05,\
                 0.1,\
                 0.1,\
                 0.1,\
                 0.1,\
                 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1,\
                 0.2]
if sign == 40: # Double number of sensors
    anomRates = [0.2, 0.2, 0.2, 0.2, 0.2, 0.2,\
                 0.05, 0.05, 0.05, 0.05,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1,\
                 0.1, 0.1, 0.1, 0.1,\
                 0.1, 0.1, 0.1, 0.1,\
                 0.1, 0.1, 0.1, 0.1,\
                 0.1, 0.1, 0.1, 0.1,\
                 0.1, 0.1,\
                 0.2, 0.2]

anomaly_params = [.9, anomRates, [L, L+L2], [0.5, 1.5], [[0.5, 0.5], [0.5, 0.5]], 0.2, 0.5]

###############################################################################
# Create Anomaly Data
###############################################################################
x_anomalies = np.zeros((anomaly_size, (L+L2)*Fs, sign+1))
y_anomalies = np.zeros((anomaly_size, (L+L2)*Fs, sign+1))

# Generate anomalous data
for i in range(0, anomaly_size):
    if i < 8:
        x_anomalies[i,:,:], y_anomalies[i,:,:] = \
            dg.generateAnomalous(anomaly_params, verbose=True)
    else:
        x_anomalies[i,:,:], y_anomalies[i,:,:] = \
            dg.generateAnomalous(anomaly_params, verbose=False)
print("Finished creating anomaly dataset of",anomaly_size,"samples")

###############################################################################
# Save Data
###############################################################################
with h5py.File(filename+"_L1_"+str(L)+"_L2_"+str(L2)+"_SIG_"+str(sign)+"_SEED_"+str(seed)+".hdf5", "w") as f:
    f.create_dataset("x_train", data=x_train)
```

```
        f.create_dataset("y_train", data=y_train)
        f.create_dataset("x_val", data=x_val)
        f.create_dataset("y_val", data=y_val)
        f.create_dataset("x_test", data=x_test)
        f.create_dataset("y_test", data=y_test)
        f.create_dataset("x_anomalous", data=x_anomalies)
        f.create_dataset("y_anomalous", data=y_anomalies)
        f.create_dataset("column_names", data=outputNames)
        print("Finished saving: "+filename+"_L1_"+str(L)+"_L2_"+str(L2)+"_SIG_"+str(sign)+"_SEED_"+str(seed)+".hdf5")

    return dg

class DataGenerator(object):
    def __init__(self, L, L2, Fs):
        self.L = L+L2
        self.L2 = L2
        self.Fs = Fs

    def setParams(self, inputSigs, outputSigs, outputDeps, internalDeps, verbose=False):
        self.outputOps = []
        self.outputDeps = []
        self.internalDeps = []
        self.inputSigs = inputSigs
        self.outputSigs = outputSigs

        # Pick random operations for internal signal dependencies
        # Currently only use +, - here but could also add /, *- etc.
        self.internalDepsOps = [[np.random.choice(['+','-']) for i in internalDeps] \
                                for i in internalDeps]

        # Pick random operations for input/ouput signal dependencies
        for sigGroup in outputDeps:
            self.outputOps.append([np.random.choice(['*']) for i in sigGroup])

        # Add copies to output/input dependencies
        for idx, sig in enumerate(outputSigs):
            for copy in range(sig[0]):
                self.outputDeps.append(outputDeps[idx])
                self.internalDeps.append(internalDeps[idx])

        # Fix output function in place
        self.outputs, self.outputSigOps = self.processInputs(outputSigs)

        if verbose:
            print("outputs:")
            print(self.outputs)
            print(self.outputSigOps)
            print("internal dependency operations")
            print(self.internalDepsOps)

    def processInputs(self, sigs):
        signalList, opsList = [], []
        for sig in sigs:
            signalCopies, signalOps = [], []

            for ld in range(sig[1]): # Currently only use +, - here
                signalOps.append(np.random.choice(['+','-']))

            signal = None
            combSigs = []
            if sig[-1][0] == False: # Cosine
                signal = Cosine
            elif sig[-1][1] == False: # Sine
                signal = Sine
            else: # Combinations
                for ld in range(sig[1]): # Set combinations by group
                    combSigs.append(np.random.choice([Sine, Cosine]))

            opsList.append(signalOps)
            fs, amps, ds = [], [], []
            for ld in range(sig[1]): # Set combinations by group
                fs.append(np.random.randint(sig[2][1][0], sig[2][1][1])/sig[2][1][2])
                amps.append(np.random.randint(sig[2][2][0], sig[2][2][1])/sig[2][2][2])
                ds.append(np.random.randint(sig[2][0][0], sig[2][0][1])/sig[2][0][2])

            for copy in range(sig[0]):
                lindeps = []
                for ld in range(sig[1]):
                    if not combSigs: # Combinations
                        lindeps.append(signal(Amp=amps[ld], Freq=fs[ld],
                                              L=self.L, d=ds[ld],
                                              Fs=self.Fs))
                    else:
```

```
                    lindeps.append(combSigs[ld](Amp=amps[ld], Freq=fs[ld],
                                                L=self.L, d=ds[ld],
                                                Fs=self.Fs))

            signalCopies.append(lindeps)
        signalList.append(signalCopies)

    return signalList, opsList

def generate(self, scale=1e2, verbose=False):
    # Generate new input functions
    self.inputs, self.inputSigOps = self.processInputs(self.inputSigs)
    inps = []
    for i, _ in enumerate(self.inputs):
        inpg = []
        start = np.random.choice([-scale, scale])
        for signal in self.inputs[i]:
            j = 0
            for component in signal:
                if j == 0:
                    inp = start*component.getSignal()
                    j += 1
                else:
                    inp = eval("inp"+self.inputSigOps[i][j]+
                               "component.getSignal()")
                    j += 1
            inpg.append(inp)
        inps.append(inpg)

    outs = []
    for i, _ in enumerate(self.outputs):
        outg = []
        for signal in self.outputs[i]:
            j = -1
            # Create signal from components
            for component in signal:
                if j == -1:
                    out = component.getSignal()
                    j += 1
                else:
                    out = eval("out"+self.outputSigOps[i][j]+
                               "component.getSignal()")

            # Resolve dependencies on input
            for k, _ in enumerate(self.outputDeps[i]):
                if self.outputDeps[i][k]:
                    out = eval("out"+self.outputOps[i][k]+
                               "np.mean(inps[k], axis=0)")

            # Resolve internal dependencies
            for k, _ in enumerate(self.internalDeps[i]):
                if self.internalDeps[i][k] > 0:
                    out = eval("out"+self.internalDepsOps[i][k]+
                               "(self.internalDeps[i][k]*\
                                   np.mean(outs[k], axis=0))")
            outg.append(out)
        outs.append(outg)

    input_sequence = np.array(list(chain.from_iterable(inps))).T
    output_sequence = np.array(list(chain.from_iterable(outs))).T

    if verbose:
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(14,7));
        ax1.plot(input_sequence);
        ax1.set_title('Input Signals');
        ax2.plot(output_sequence);
        ax2.set_title('Output Signals');

    # Add program signal to dataset
    full_output_sequence = np.zeros((output_sequence.shape[0], output_sequence.shape[1]+1))
    full_output_sequence[:,:-1] = output_sequence
    full_output_sequence[:,-1:] = input_sequence[:,0].reshape(output_sequence.shape[0],1)
    return full_output_sequence[:-self.L2,:], full_output_sequence[-self.L2:,:]

def generateAnomalous(self, anomaly_params, verbose=False):
    xa, xb = self.generate()
    x = np.vstack((xa,xb))
    if np.random.random() < anomaly_params[0]: # If overall anomaly
        anomalies = [False]*len(anomaly_params[1])
        while True not in anomalies:
           for i, probability in enumerate(anomaly_params[1]):
             anomalies[i] = (np.random.random() < probability)
```

```
    ranges = []
    min_ts = round((anomaly_params[2][1]-anomaly_params[2][0])*anomaly_params[5])
    for i in anomaly_params[1]:
        start = np.random.randint(anomaly_params[2][0], anomaly_params[2][1]-min_ts)
        stop = np.random.randint(start+min_ts, anomaly_params[2][1])
        ranges.append([start, stop])

    x_anomalous = np.copy(x)
    y_anomalous = np.zeros(x.shape)
    for signal, _ in enumerate(anomaly_params[1]):
        if(anomalies[signal]):
            # Two types of anomalies:
            # Constant: constant increase/decrease with smoothing
            # Random: random additions or subtraction per timestep
            # Two lengths:
            # Lasting: anomaly stays for the entire remaining sequence
            # Fleeting: signal returns to normal values at some point
            a_length = np.random.choice(['l','f'], p=anomaly_params[4][0])
            a_type = np.random.choice(['c','r'], p=anomaly_params[4][1])
            op = np.random.choice(['+','-'])

            if a_length == 'l':
                length = anomaly_params[2][1] - ranges[signal][0]
                smoothing = np.hstack((np.linspace(anomaly_params[6], 1, round(length/3)),
                                       np.ones(length-round(length/3)*1)))
                if a_type == 'r':
                    for idx in range(ranges[signal][0],anomaly_params[2][1]):
                        x_anomalous[idx,signal] = eval("x[idx,signal]"+op+\
                                    "np.random.uniform(anomaly_params[3][0], anomaly_params[3][1])\
                                        *smoothing[idx-ranges[signal][0]]*x[idx,signal]")
                        if((x[idx,signal]-x_anomalous[idx,signal])**2 > 0):
                            y_anomalous[idx,signal] = 1
                else:
                    mean_xrange = np.mean(x[ranges[signal][0]:anomaly_params[2][1],signal])
                    constant_val = np.random.uniform(anomaly_params[3][0], anomaly_params[3][1])*mean_xrange
                    smoothing *= constant_val
                    for idx in range(ranges[signal][0],anomaly_params[2][1]):
                        x_anomalous[idx,signal] = eval("x[idx,signal]"+op+"smoothing[idx-ranges[signal][0]]")
                        if((x[idx,signal]-x_anomalous[idx,signal])**2 > 0):
                            y_anomalous[idx,signal] = 1
            else:
                length = ranges[signal][1] - ranges[signal][0]
                smoothing = np.hstack((np.linspace(anomaly_params[6], 1, round(length/3)),
                                       np.ones(length-round(length/3)*2),
                                       np.linspace(1, anomaly_params[6], round(length/3))))
                if a_type == 'r':
                    for idx in range(ranges[signal][0],ranges[signal][1]):
                        x_anomalous[idx,signal] = eval("x[idx,signal]"+op+\
                                    "np.random.uniform(anomaly_params[3][0], anomaly_params[3][1])\
                                        *smoothing[idx-ranges[signal][0]]*x[idx,signal]")
                        if((x[idx,signal]-x_anomalous[idx,signal])**2 > 0):
                            y_anomalous[idx,signal] = 1
                else:
                    mean_xrange = np.mean(x[ranges[signal][0]:ranges[signal][1],signal])
                    constant_val = np.random.uniform(anomaly_params[3][0], anomaly_params[3][1])*mean_xrange
                    smoothing *= constant_val
                    for idx in range(ranges[signal][0],ranges[signal][1]):
                        x_anomalous[idx,signal] = eval("x[idx,signal]"+op+"smoothing[idx-ranges[signal][0]]")
                        if((x[idx,signal]-x_anomalous[idx,signal])**2 > 0):
                            y_anomalous[idx,signal] = 1

    if verbose:
        f, (ax1, ax2, ax3) = plt.subplots(3, figsize=(9,18));
        ax1.plot(x[:,:]);
        ax1.set_title('Original Signal');
        ax2.plot(x_anomalous[:,:]);
        ax2.set_title('Signal + Anomaly');
        diff = x_anomalous[:,:]-x[:,:]
        ax3.plot(diff);
        ax3.set_title('Difference Due to Anomaly');

    return x_anomalous, y_anomalous
else:
    return x, np.zeros(x.shape)
```