M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Data-Driven Failure Diagnostics of Industrial Assets

An Empirical Study of State-of-the-Art Multivariate Time Series Classification Models with Advanced Data Augmentation Using Generative Adversarial Networks

Sebastian Hedegaard Hansen (S163870)
Mikkel Schmidt (S153990)

Supervisors:
Tommy Sonne Alstrøm (DTU)
Anders Hvashøj (ZEVIT Aps)

Kongens Lyngby 2021

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

When industrial assets malfunction, the financial consequences can be immense as operations potentially need to temporarily shut down or slow down. With assets becoming more complex the need for digital troubleshooting systems become more apparent as humans are limited by their weak ability to comprehend and analyse a large amount of complex data. Such a system needs to detect when a failure has occurred, diagnose the most likely failure, guide a technician or operator in resolving the failure, and lastly test if the asset is operating as normal after the repair. This thesis focuses on the diagnosis of the assets based on the multivariate time series data coming from the asset's sensors. An extensive set of experiments were carried out on both open-source datasets and a real-world dataset from five wind turbines. An automatic prepossessing and hyperparameter tuning process of multiple state-of-the-art time series classification algorithms, including InceptionTime and MiniROCKET, are presented. The method is proven to be effective on the open-source dataset where the achieved results match those from published reviews of the same algorithms using the same datasets. The method was used on the real-world data where it was shown that it was possible for the classification algorithms to identify, to a wide extent, in which component of the wind turbine the failure had occurred.

Failures on these industrial assets can be rare. The classification models will therefore have too few examples to learn the general characteristics of a failure. Therefore, the use of generative adversarial networks (GANs) to increase the performance of the classification models was investigated. For this, a version of the newly published generative model, uTSGAN, was developed and compared to the original. The experiment showed that it was possible to train a GAN that was able to generate realistic samples that could improve the performance of the classification model when combining it with the true data. More work however needs to be done in the academic community to make it feasible to use GANs in an operational setting.

Preface

This M.Sc. thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark under the supervision of Associate Professor Tommy Sonne Alstrøm. The thesis was conducted in collaboration with the company ZEVIT Aps with the supervision of Anders Hvashøj. The work was carried out in the spring of 2021 and the project covers a workload of 65 ECTS points.

Kongens Lyngby, July 16, 2021

Sebastian Hedegaard Hansen (S163870)
Mikkel Schmidt (S153990)

Acknowledgements

The authors would like to thank Tommy Sonne Alstrøm for his competent supervision throughout the project. He always asked all the questions we did not think about our-self which has elevated the project. We would also like to thank Anders Hvashøj and ZEVIT in general for the collaboration. Your guidance has helped a lot with the understanding of the root of the problem and all the many different nuances there are to it. Finally, we would like to thank Kaleb Earl Smith, the author of the uTSGAN papers, for answering our questions and discussing our suggested improvements to his original architecture.

Contents

CHAPTER 1

Introduction

When industrial assets are malfunctioning, the financial consequences can be immense as operations potentially need to temporarily shut down or slow down [Ise97]. The existence of robust and reliable maintenance- and troubleshooting systems are therefore essential for companies that rely on a steady and consistent operation.

Humans are highly limited by their weak ability to analyse multiple features at the same time along with remembering a large amount of information for longer periods. Troubleshooting complex industrial assets can therefore be problematic as interconnected dependencies between an asset's components can be difficult to fully comprehend. Previous fixes for a diagnosis can be forgotten and symptoms associated with a given diagnosis can be ignored if the technician has not experienced it before.

When a symptom is detected it is not guaranteed that an efficient plan of action is immediately obvious for a technician or operator. A series of questions by the technician or operator must be answered to classify the diagnosis for the experienced symptoms. This process can be very cumbersome, especially with offshore assets. Imagine if an offshore wind turbine is showing some symptoms (by for example producing less energy than expected), then it would be valuable for the operators to have an idea of what the potential diagnosis could be before sending a technician to the asset. This would reduce the number of times sending a technician to the platform and thereby reducing the amount of money needed for transportation and general downtime. Troubleshooting is today depending on skilled technicians who through education and working experience with an asset can understand the potential diagnoses associated with the displayed symptoms [Gho+19]. For novel technicians, the process is often associated with a "trial-and-error" approach, where no initial knowledge or structure about the system is present except for his education. The transfer of knowledge from the experienced technician to the novel technician can be a slow process that may take years before the novel technician can achieve independence. Knowledge, therefore, needs to be moved from the brains of the technicians to a system. This can reduce the duration of the onboarding and training of new technicians. By logging the symptoms of an asset, the actual diagnosis associated with the symptoms, and the final solution to the diagnosis, it would be possible to create such a digital system that could become an efficient assistant to both new and experienced technicians and operators. Advances in machine learning and an ever-increasing amount of data collected by companies and researchers are setting the scene for new possibilities for creating more intelligent systems.

1.1 Thesis Scope

To understand the scope of this project some background context is required. Guided troubleshooting can roughly be divided up into four parts: A detection part, a diagnostic part, a guided repair part, and a test part. These elements will be explained in further detail in section 2.1. It is not possible to touch upon every element of the problem during the duration of a thesis. The most challenging part is the diagnostic as many failure types can be closely related with subtle differences in displayed symptoms. This element will therefore be the main focus of this thesis. These subtle symptoms display themselves in the asset's sensor data along with unstructured heuristic data like its service reports and general repair history. Both of these types of data are valuable to identify which fault the asset is experiencing. A model is therefore needed that can handle and analyze both the unstructured heuristic data and the structured sensor data.

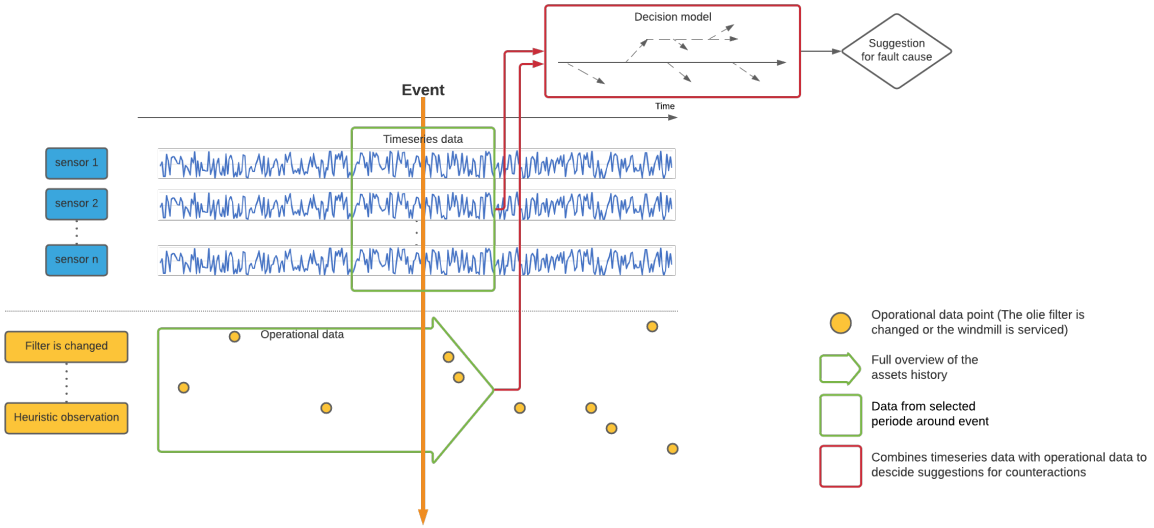

Figure 1.1: Model of the different elements of diagnostic: The data relevant for an asset are divided into two parts: Sensor data and operational data. A model will classify the failure of an asset from a given period surrounding an event of abnormal behaviour. The probabilities from this classification algorithm along with the relevant operational data are used as an input for a strategy decision model. This will then, based on the input, suggest the most optimal steps to help the operator or technician to identify the actual failure of the asset.

Models for both of these types of data should be unified in a *strategy decision model* that outputs a suggestion for steps that allows an operator or technician to identify the root cause of the problem. The models for the two types of data should be triggered when some event is registered. This could be a *fault detection model* indicating abnormal behaviour or an operator/technician observing something unusual. The unification of the sensor data and operational data will be the subject of future work and will be discussed in section 7. This unification of the two types of data and the strategy decision model is illustrated in figure 1.1. This thesis will focus on the time series sensor data and to what extent it is possible to classify multivariate time series data into known faults. This will be done by setting up a series of experiments where different state-of-the-art time series classification models will be used to predict a fault. The datasets that are going to be used are both open-source, where benchmark results have been published by researchers around the world, but also relevant data from operational wind turbines where failure types have been registered through a two year period. The reason to put so much effort into creating a good classification algorithm is to, in the end, make the *strategy decision model* simpler and more efficient. If the output from the time series classification model is more reliable the *strategy decision model* can weigh these inputs higher and thereby put less emphasis on the less structured and harder to interpret operational data. Therefore, the classification model is a valuable step in the process of creating the *strategy decision model*.

This thesis is carried out with the fact in mind that the models will be deployed in an operational setting. A considerable effort has therefore been put into implementing an infrastructure that allows for dynamic data prepossessing and model initialization independent of the dataset in question. This is done by implementing a pipeline with a grid search incorporated that allows for automatic hyperparameter tuning and easy continued training when more data becomes available. The thesis can therefore be categorized as both an implementation and a research project. The proposed methods are new in an operational setting and only a few similar methods have been proposed in the literature. The presented experiments are therefore conducted to test if the proposed methods have any validity to solve the problems associated with creating a model that can classify a failure on an asset based on its sensor

data.

Even though massive amounts of data are collected about industrial assets the logging of specific failures is limited. This means that the data-hungry models that are used to classify the failure types possibly do not have the necessary data foundation to make correct predictions on unseen data. Research into companies working in the guided troubleshooting space revealed a bottleneck in implementation time due to slow mapping of symptoms to failures, traditionally done manually by domain experts. All these issues will be further explained in section 2.5. Methods for handling these issues and reduce their influence on a successful model will be suggested. Here state-of-the-art generative adversarial models (GANs) are proposed to enrich the original dataset with synthetically generated data that draws samples from the underlying data distribution. The thesis will contribute to the general development of GANs for time series generation by building upon the newest generative models presented in the literature.

1.2 Research Question

Based on the brief introduction to the subject and the associated problems in the previous section, the following research question has been formulated. The research question aims to contain the following elements for a final system:

1. Diagnose most likely fault on an asset based on its sensor data.
2. Can be implemented with a smaller data foundation.
3. Reduces the implementation time for the companies.

- To what extent is it possible to create a model for fault diagnostics of industrial assets by classifying time series data from its sensors?
- Could a data set be enriched with synthetic data generated from the original dataset to make it more balanced and larger?
- How should a system be implemented to allow for dynamic prepossessing, training, and continuous improvement?

1.3 Thesis Structure

The thesis is structured in the following way:

- The first chapter gives an overall introduction to the general topic along with the problem and scoping relevant for the work conducted during the thesis.
- The second chapter consists of a deeper dive into the background of the topic with a description of related work in the industry and literature.
- The third chapter presents the theory used throughout the thesis.
- The fourth chapter presents the high-level design idea behind the implemented system, how it is used in the context of this thesis, but also in the context of an operational setting. Next, the implementation of the infrastructure is presented along with the different proposed algorithms and methods.
- The fifth chapter presents the experimental method and setup used to test the stated research questions.
- The sixth chapter goes through all the results from the experiments. The results will be analysed and discussed along with the limitations of the proposed methods.

- The final seventh chapter will conclude on the stated research question presented in chapter one.

The ideas behind developing intelligent troubleshooting systems have now been explained. These mainly focus on the human limitations that prevent a large amount of data to be analysed. Therefore systems need to be developed that can collect, store, and analyse data across time and assets to be used when a fault is detected. The knowledge about the assets should be transferred from the skilled technician to a digital system so for instance AI models can assist technicians to achieve independence faster. The scope of this thesis will be on the classification of the time series data generated by the sensors on industrial assets. This will in future work be used in collaboration with heuristic operational data to create a unified *strategy decision model* meant to guide operators or technicians in diagnosing the asset when a fault is detected. In the following section, a more detailed presentation of the background and project will be presented. Furthermore will related work from the literature and industry be presented.

CHAPTER 2

Background

The following section will focus on the theoretical background for a guided troubleshooting system. Methods that are used in the industry today will be presented along with newer and more experimental methods presented in academic studies. Furthermore, experiments from other industries, like medical and software development, will be presented for inspiration for future work.

2.1 What is Troubleshooting

The basic aspects of troubleshooting will now be explored. There are in general many different ways to approach this subject, but the troubleshooting process can roughly be divided up into the following four parts:

1. A symptom detecting part.
2. A diagnostic part.
3. A guiding solution part.
4. A testing part.

Literature by R. lsermann [Ise97; Ise05] explain these basic components in a troubleshooting system as follows:

- A system that processes incoming data from either sensors, operators, technicians etc. and determining if the observed system is showing symptoms of a fault. This is in the literature mainly described as a symptom detection system.
- When some symptoms are registered, the origin of these symptoms needs to be determined. This step is in the literature referred to as diagnostics.
- When the diagnosis is known, a set of actions must be determined to stop the symptoms and get the system running in its normal state.
- Finally, tests need to be devised to make sure the performed repair resolved the underlying fault of the diagnosis.

The following sections will explain the above four aspects of troubleshooting in further detail.

2.1.1 Symptom Detection

Numerous symptom detection methods have been suggested in the literature. These can however roughly be divided into two sections, namely an analytic symptom detection and heuristic symptom detection. The analytical method looks at quantifiable measurable symptoms, like sensor readings. Heuristic symptoms refer to the qualitative observations that operators or technicians encounter in daily operations (smells, noises, oil on the floor etc.) [Ise97]. Within the analytical method different ways to determine the state of the system exist:

- Limit value checking: If some values exceed some predefined thresholds.
- Signal processing: The use of signal processing algorithms to determine if the system's *characteristic values* are not behaving correctly. This could be the use of *anomaly detection*.

The heuristic symptoms are more loosely defined. Here most observations are made by an operator or technician and can be defined as a data point. An asset's history can however also be included in heuristic data and will altogether be referred to as operational data.

2.1.2 Fault Diagnostics

To be able to find the diagnosis that is causing the showed symptoms, both the analytical symptoms and operational data must be taken into consideration. Because, sensor data may indicate a heating system must be replaced, but if it can be seen in the operational data that the temperature sensor has been replaced lately, it is perhaps more likely that the new sensor has some production fault. Here operational data can be expressed as statements such as that different parts have been inspected, repaired or replaced. But it can also be expressed as fuzzy logic, so for instance, how loud is a noise is on a scale.

One diagnosing approach is to unify the analytical symptoms and operational data and then apply different methods for diagnosing, for instance, classification- or reasoning methods. The classification method could for instance be a deep neural network, as it can accommodate for the non-linearity that is present in the data. But it is also possible to explain the causal relationship as a Bayesian network, which is something that Dezide is doing and will be explained in more detail in section 2.2.1 [Ott12].

Another approach is to analyse the analytical symptoms separately to predict the different diagnosis probabilities. Then the diagnosis probabilities and the operational data can be unified and the same approach as before can be used. This would result in a simpler classification- or reasoning method, as it should work on less complex data.

2.1.3 Guided Solution

When a diagnosis has been determined, a series of actions should be performed by an operator or technician to resolve the underlying fault of the diagnosis. It becomes increasingly harder for humans to keep up with the complexity of modern machines and technical components [Gup+18].

The information needed to repair an underlying fault can either come from previous knowledge about the diagnosis, technical documents, or a "trial-and-error" approach [DSH19].

Several different methods and tools have been suggested to guide or assist technicians in the repair phase of troubleshooting. The technical manuals can be utilized to correct known faults by guiding the technician based on "if this then that" principles [Gup+18]. But troubleshooting mainly relies on the technician's ability to identify the issues and come up with steps to solve it themselves. Here the technician must be well educated in the components they are working with and can visualize the inner functions of the machines [DSH19].

2.1.4 Testing

When a repair of a fault has been conducted it is important to make sure that the repair solved the problem. For this, a test must be devised to test the state of the system or sub-system. Different test strategies can be applied, where some are listed below [TRY21]:

- Analytical strategy: Testing based on predefined requirements or risks.
- Model-based strategy: Takes an expected situation and creates a model for it. Here the test accounts for input, output and expected general behaviour (E.g. the testing of a web application - max/min traffic, input signals etc.).

- Methodical strategy: Follow predefined standards like ISO standards or internally defined.
- Standards compliant or Process compliant strategy: Process and guidelines for the exact behaviour of a system based on industry experts (E.g. food and drug operations have minimum requirements for the quality of their products).
- Reactive strategy: Testing is done as the system is deployed (E.g. software is released in beta versions so users test if there are any bugs).
- Consultative strategy: Key stakeholders define the conditions and requirements/scope of the test.
- Regression averse strategy: Testing to avoid the regression risk (E.g. when a system becomes increasingly more complex, the probability that new changes will cause problems for existent functionalities increases).

Depending on the type of asset, different test strategies can be utilized. Manufacturing equipment is for instance quality controlled based on a standards-compliant strategy where parts are inspected if they fall within some predefined tolerance [Gho+19]. A test point (being either logical or physical) is used to identify symptoms of that system or subsystem. Here either automated measurements can be made or a manual inspection is needed. The combination of these tests will be fundamental for evaluating fault diagnosis. The basic aspects of troubleshooting were explained as; symptom detection, fault diagnosis, testing, and counteractions. The flow of troubleshooting is described in fig 2.1.

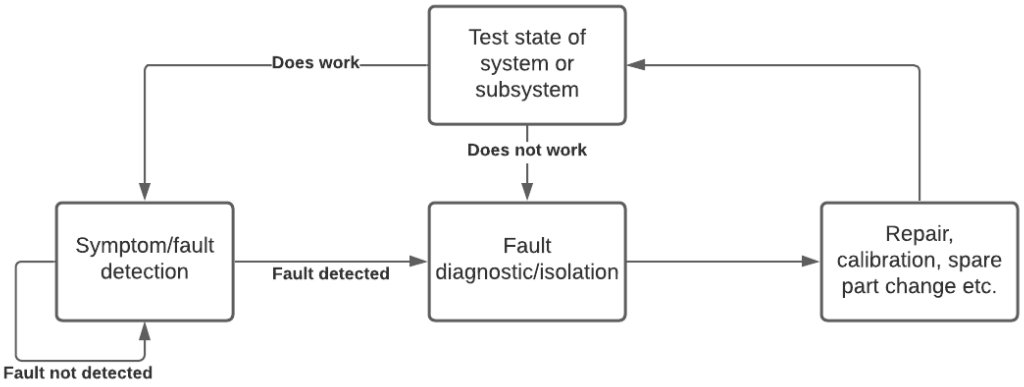

Figure 2.1: Flow of the troubleshooting process. If a symptom is *detected* a *diagnosis* is performed. A technician tries to *repair* the system. If the *test* tells that the problem is resolved the system should look for new symptoms, if not, the diagnosis should continue.

2.2 Guided Troubleshooting in Industry

2.2.1 Dezide

Dezide markets themselves as a system that provides AI-powered interactive troubleshooting. Their system is focusing on interactively guiding a technician to locate the problem via statistical questioning. This means that the system states a question, and based on the technician's answer the system states a new question until the problem is identified. This could e.g. be "did loosen the bolt decrease the pressure of the tank? - Yes or no."

2.2.1.1 Technology

Dezide uses a combination of different technologies. The backbone of their system is based on a Bayesian network which combines a directed acyclic graph (DAG) with local probability distributions. The local probability distributions are represented as nodes in the network and the edges between the nodes are then the causalities. This means that if node A can be caused by node B, then there is an edge from node B to node A. An example of a Bayesian network can be seen in figure 2.2

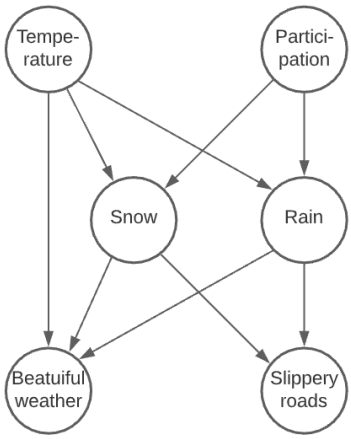

Figure 2.2: Example of a Bayesian network.

From figure 2.2 it can be seen that *beautiful weather* depends on if it *snows* if it *rains*, and the *temperature*. It could for instance be beautiful weather if it snows and does not rain, regardless of the temperature. It could also be that the weather is not beautiful due to low temperature, no snow, and no rain. Rain and snow both depends on the temperature and the participation, as it can only snow or rain if there is participation and it will either rain or snow depending on the temperature. Then the roads could be either not slippery, a little slippery, or very slippery depending on if it snows or rains. The specification of the whole network can be stated as the following probability tables:

P(Temperature)	*P(Participation)*
P(Snow|Temperature,Participation)	*P(Rain|Temperature,Participation)*
P(Beautiful weather|Temperature,Snow,Rain)	*P(Slippery roads|Snow,Rain)*

As the table shows, the probability distributions of the different nodes are straightforward to define. The particular distributions themselves are however not straightforward, as they either require experts to estimate the distributions, infer numbers from historical statistics, or a combination of both. [Ott12]

The backbone Bayesian network is combined with utility theory. Utility theory is shortly described as a way to make better decisions. There exists two types of decisions, which are test decisions and action decisions. Test decisions are also referred to as observations, which is exactly what the nodes in a Bayesian network are. Action decisions, referred to as actions, are something that can be performed to affect the state of the given environment.

So when utility theory and a Bayesian network are combined it forms a decision tree with different types of nodes, namely observation-, action-, and working nodes. Working nodes are just a node that asks whether the previous action solved the problem. So an example of such a tree could look as in figure 2.3.

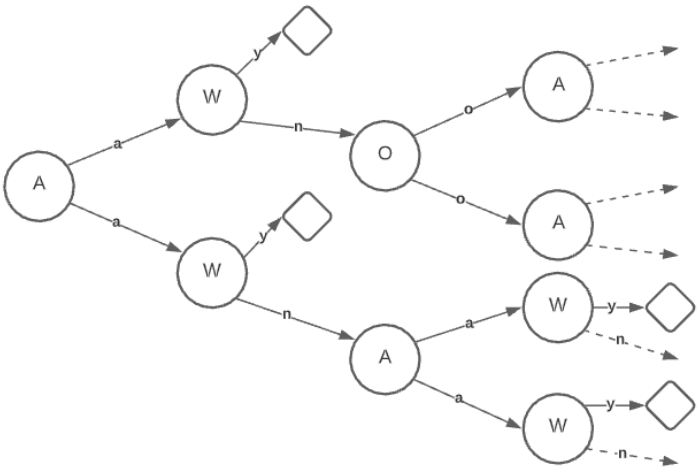

Figure 2.3: Example of decision tree. *A*, action nodes. *W*, did it solve the issue nodes. *O*, observation nodes. Diamonds, the issue is resolved.

For each of the action nodes, *A*, an action can be performed. Depending on the action, it will move on following the edge for the particular action. Each action node is followed by the *W* node, which asks if the issue is resolved. If it is resolved the job is done, but if not it moves on following the *n* edge. If it comes to an observation node, *O*, then it makes an observation and moves on according to the observation. It will continue until the issue is resolved. One route from the root of the tree to a diamond node is called a strategy, so a decision tree contains multiple strategies. The goal is now to find the optimal strategy. That is the strategy with the maximum expected utility.

2.2.1.2 Associated Problems

So as seen in the previous section, Dezide uses a technology that is capable of mapping out all the different strategies to resolve every single issue that can occur. This makes the system perform really well once it has been configured for the given use case.

While the combination of Bayesian networks, decision trees, and utility theory creates the foundation for a very well-performing system, it has its downsides. To actually define all the different strategies for all of the different issues, requires a lot of manual work. It requires several experts to estimate prior probabilities for the different observations and define the different actions that can be performed at which nodes. This means that it requires a lot of time to implement this solution to the different use cases. Even if the use cases are the same the different strategies and actions might be very different. Expanding the system with new failures is also a manual process where new probabilities must be defined and new guides for the technicians must be created. The big bottleneck is therefore the implementation time and the ability to scale and adjust to new failure instances.

2.2.2 CMMS Systems

Computerised maintenance management systems (CMMSs) are the most widely used software solutions to manage larger systems with many complex subsystems. They allow operators to schedule maintenance and repairs and log the history of an asset over time. Wienke et. al. [WHV16] argues that the use of computerised solutions for repair and maintenance are essential as a system becomes more and more complex. This system has the following features according to Lopes et. al. [Lop+16]:

- Analysis of failures to reduce its occurrences, and to plan maintenance activities and condition monitoring.

- Access to the information in real-time at different places to facilitate technicians actions.
- Support scheduling functions attending both maintenance, technician availability, and production plan.
- Support performance assessment and improvement initiatives.
- Track the movement of spare parts.

These systems lack proper utilization of the incoming data streams for the assets they are monitoring and controlling. The key deficiencies with these systems are according to Labib et. al. [Lab04] as follows:

- The use of the data collected from the individual machine is very sparse.
- Expensive ”off the shelf” software solution is bought without fully integrating it in the correct business.
- Lack of decision-related software - If a failure is detected, no suggestions on how it should be fixed are made.
- The role of the CMMS system is often deduced to just being a database of spare parts and as a scheduling software.
- Not very user-friendly systems.
- Users’ lack of trust in complex mathematical models.

The main takeaway is the need for better mathematical- or AI models that can utilize the huge amount of data companies are collecting from their assets.

2.3 Guided troubleshooting in the literature

Now that some of the relevant actors in the industry have been presented it makes sense to present some of the methods from the literature within guided troubleshooting with a special focus on diagnostics and the use of synthetically generated data.

Zhang et al. in [Zha+17] reviews the most common methods within diagnostics of industrial assets and equipment. Many of the methods are based on classical signal processing. The signals coming from the sensors on the asset are denoised and features are extracted using various methods such as kernel density estimation (KDE) and frequency domain features using the Fourier transform. A classifier can then be fitted on the extracted features using different classification algorithms such as the support vector machine or random forest classifiers. Zhang et al. argue that the problem with this method within the fault diagnosis context is the need for domain experts to validate the quality of the extracted features. Furthermore, the feature extraction transforms the signal data so the temporal coherence of the data is lost which can have a big effect as the context of a data point is lost.

Several articles have looked into the use of time series classification algorithms that takes the raw sensor data as input instead of a prepossessed dataset. The aim of this method is to eliminate the need for prior knowledge about a system. One such method is proposed by Pan et al. [Pan+17]. They have developed the deep neural network *liftingNet*. The network is especially designed to handle noisy mechanical data much like the one expected from big industrial assets like wind turbines that are investigated in this thesis. To be even more detailed the network is optimized for rotating mechanical actions that are sampled at a high frequency such as a bearing of transmission shaft. The network is however designed for univariate time series which highly limits its use as a generic diagnosis system. In [Zha+17] a similar model was developed to diagnose faults on a bearing. These models for specific components are very effective with accuracy scores in the high 90’s but suffers from the same limitations as mentioned above.

Liu et al. [Liu+19] tried to deal with the issue of unbalanced fault versus normal states. They do this by generating synthetic data from the sensor data coming from a set of wind turbines when they are experiencing some failure. They use a Generative Adversarial Network (GAN), which learns the distribution of the provided data and generates new data that follows the structure of the original dataset. The problem they are investigating is however not a diagnostic problem but rather a detection problem which in general are simpler. The GAN is also helped a long way by domain experts defining the low-frequency patterns of the individual sensor before the GAN is applied. The GAN therefore only needs to add the final details that are too complex for a human to identify. The results obtained in this paper is promising as they manage to increase the correctly detected faults by adding generated data.

2.4 Perspective to Other Industries

This project will focus on troubleshooting industrial assets. But other industries are relevant to explore for inspiration and techniques. One of these industries is the medical sector. Here it is fairly easy to view a patient as an asset with some kind of failure showed through a series of symptoms and the doctor or nurse as a technician. Patients are being monitored by both analytical and heuristic methods when at the hospital, but when going about their life only a few analytical measurements are used to monitor their health. The use of wearable health sensors is used more and more in recent years to explore the benefit of catching critical conditions for patients.

In [Cli+13] such an experiment was carried out on 200 post-operation cancer patients. Here the patients were equipped with a sensor that measures everything from heartbeat to blood oxygen saturation levels. One of the main issues with monitoring patients is to determine a normal state, as different individuals have very different lifestyles. Here a large set of *normal* measurements from other patients can help determine the accepted values expected by a given patient. This problem is easier with industrial assets as their normal operation is more clearly defined and operations are more predictable across the same asset type [Hay+00]. Features were extracted from the incoming sensor data and combined with the clinical observations by the doctors in a tabular manner. For predicting if the patient would experience severe complications a support vector machine was used to classify the abnormalities in the 200 cancer patients with promising results. For diagnostic a series of models have been explored. Here successfully diagnosed patients' symptom data have been used to train a classifier (Naivë Bayes, Neural network, decision tree and KNN) to investigate if it was possible to determine a patient's illness solely based on their data from monitoring sensors [Kon01]. Several real-world cases were presented with interesting results. These use cases are relevant for the detection and diagnosis of industrial assets.

Another industry that is relevant to gain inspiration from is the software development industry. Here a lot of resources is used on fixing problems (bugs) and testing code. The process of *debugging* a piece of software can be a cumbersome process as it is often hard to isolate the real source of the problem. In [ESK18] an AI method was used to detect and diagnose potential bugs in software components. Training data was collected using commits from a version control software on open source projects. Here the bugs referred to the file where the changes were made. This made it possible to automatically create a labelled dataset where the developers continuously generated training data for the classifier. For diagnosing the faults based on the detection the use of the *Barinel software diagnosis algorithm* is proposed [AZV09].

2.5 Observed Issues

Troubleshooting is known to be a complex problem and other studies have tried to solve it in different ways as described in the previous sections. The common issues from the different studies are summarised in the following sections.

2.5.1 Data Foundation

No matter which approach is used to solve the troubleshooting problem it requires some form of data. It can both be data stored in computers, but also data stored in humans who are experts in the given area. As in many other cases, good quality data can be hard and expensive to obtain and that also applies to the troubleshooting problem. The reasons for this are mainly:

- Fault data must be asset type-specific.
- Fault data can occur very rarely.

In many cases, the fault data must be specific to the type of asset, as assets can be built very differently and therefore do not share the same structure and thereby faults. As some faults only occur once every e.g. fourth year then those faults are not as likely to be present in the data or are very sparsely represented. It is also worth mentioning that companies owning an asset are most likely not interested in sharing their assets' data openly as other companies potentially can reverse engineer the inner workings of the system.

2.5.2 Implementation Time

Due to the sparse amount of failure data an efficient system traditional needs to be configured by humans where the internal causality between a failure and symptoms needs to be mapped out. The specification of an asset, therefore, needs to be understood by individual technicians. Furthermore does the need to create guides for the technician take extensive time.

2.5.3 Single-fault vs. Multi-fault

Many existing solutions are designed only to predict single-faults. This assumption of only one fault occurring at the same time does not translate to an operational setting. This problem has been the subject of much research. One article by Shakeri et al. [Sha+94] tries to solve this issue with a strategy called *SURE*. This strategy is a *test-and-repair* strategy that will continue to repair parts of the system until the whole system is functioning again. This solution avoids the super-exponential computation complexity of $O(2^m)$. Ottosen [Ott12] also tried to expand their Bayesian network approach to allow multi-fault detection but failed to deliver a theoretically and practically satisfying solution. This could indicate that a new approach is needed to create a satisfying solution. It is however most likely not necessary to consider multi-fault if a system or asset is frequently assessed according to [Sha+94], as the probability of faults occurring at the same time is reduced.

In this section, companies and systems within the guided troubleshooting industry have been presented. The technologies forming the backbone of their systems are based on Bayesian networks. This allows them to model the relationship between the different components of the system. To implement an operational system does however include the mapping of components functional dependencies and the extensive indexing of actions associated with a fault. Furthermore, the most common software solution, CMMS systems, has been presented. A review of the most popular version of these systems suggests the need for more intelligent systems that utilize the data streams from a organisations assets. Finally, inspiration from other industries such as software development and the medical industry have been explored. Knowledge and methods generated here could be a relevant inspiration for troubleshooting industrial assets.

CHAPTER 3

Theory

In the following section, the theory used throughout the experiments will be presented. The section is structured so the most basic elements of time series are presented first. Next, the principle of deep learning is presented followed by the state of the art classification algorithms used in the experiments. The different elements of GANs are then explained and finally, the evaluation metric used to determine the performance of the classification models and the quality of the generated data from the GANs are covered.

3.1 Time Series

A time series is a set of data points that occur in chronological order, where each data point has a certain value for a certain point in time. Time series are most commonly appearing with data points that are equally spaced in time, but they can also appear with data points randomly spaced in time. Time series data can occur as a univariate time series, which consists of a single time-dependent variable, but also as multivariate which consists of multiple time-dependent variables. In a multivariate time series, each variable not only depends on the time but also depend on the other variables. Examples of univariate time series data could be ECG data or stock data, as it only consists of signals from one sensor or source. To analyse such a time series it necessary to look at the temporal dependencies. In the multivariate time series case, it is not only necessary to look at the temporal dependencies but also the dependencies of the different sensors or sources of data. An example of this could be EEG data, where multiple sensors record signals over time.

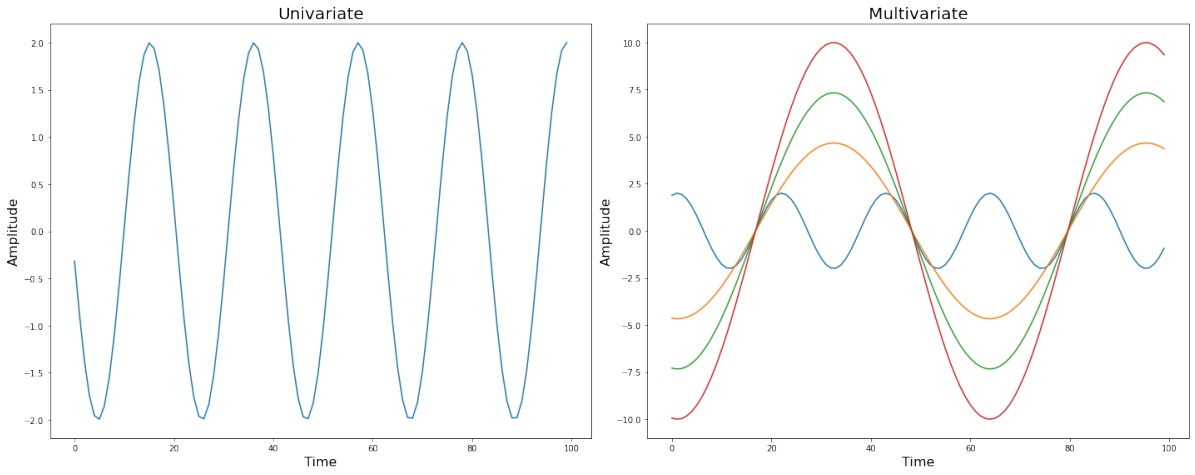

Figure 3.1: Examples of a univariate and multivariate time series.

Figure 3.1 shows two simple examples of the univariate and multivariate time series. As seen, the multivariate consists of multiple sources of data but can still be viewed as a single plot with a line for

each of the sources.

3.1.1 Spectrogram

Another way to look at time series is to look at their spectrograms. This will be used in the generative models, which will be explained in section 3.4.3. A spectrogram is a visual representation of the different frequencies in a time series over time. So for instance the time series in figure 3.2 could be represented as the spectrogram in the same figure. It can be seen from the spectrogram that the frequency changes from a low frequency to a higher frequency around the middle of the time series. The intensity change in the colour reveals that the amplitude also changes. In the multivariate case, a spectrogram would be created for each of the sensors.

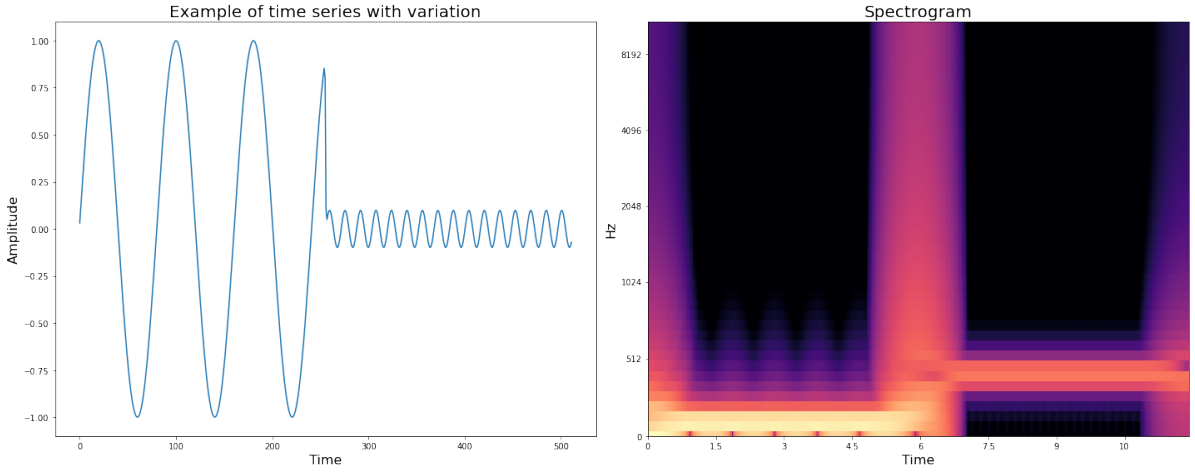

Figure 3.2: Examples of a univariate time series with frequency variation over time. On the left the time series can be seen and on the right its spectrogram can be found.

3.1.1.1 Short-Time Fourier Transform - STFT

To calculate the spectrogram for a given time series it is desired to know which frequencies the time series consists of and when. The frequencies can be calculated with the discrete Fourier transform which has the formula seen in equation 3.1. Here $X(k)$ is energy provided by frequency k for the time series x and N the length of the time series.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn} \tag{3.1}$$

The idea behind the discrete Fourier transform, DFT, is that any signal can be represented by adding multiple waves with different frequencies and amplitudes. The issue with the DFT is that it does not catch the variation of frequencies over time. Short-time Fourier transform solves this by applying the DFT in the manner of a sliding window over the time series. The formula for STFT can be seen in equation 3.2. Here $X(m, k)$ is the energy provided by frequency k for window number m, w the window function, N the size of the window, and H the hop size.

$$X(m, k) = \sum_{n=0}^{N-1} x(n + mH) w(n) e^{-j\frac{2\pi}{N}kn} \tag{3.2}$$

As equation 3.2 shows, the signal $x(n)$ from equation 3.1 is now exchanged with a new signal $x(n + mH)w(n)$. This new signal is the signal for time n for the window number m. This basically means that the STFT calculates the DFT for a specific point in time by only looking at the values within the window. The type and size of the window are changeable parameters that influence the time-frequency precision trade-off.

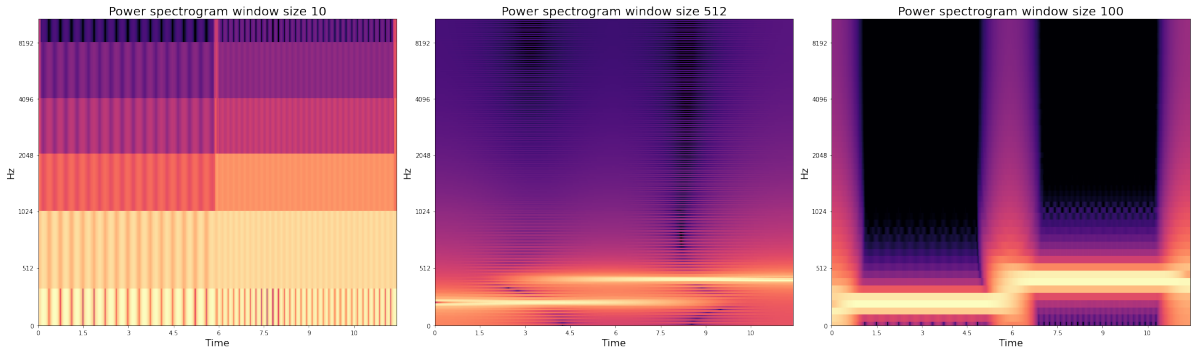

Figure 3.3: Examples of how the window size in the STFT affects the spectrogram of the signal from figure 3.2. The left figure uses a small window size, the middle a large window size, and the right a balanced window size.

Figure 3.3 shows how the size of the window influences the spectrogram. From the left spectrogram in the figure, it can be seen that is really easy to determine when in time the signal changes in frequency. But it is, however, hard to see which frequencies it changes from and to. This is due to a small window size as it would not have many data points to determine the frequency from, which also is the reason why it quickly adapts to variation in frequency. In the middle image of the figure, a spectrogram can be seen where a large window size is used (as large as the length of the signal). Here it is easier to see the exact frequencies, but it is hard to see where the frequency changes, as it looks like the whole middle part of the signal consists of two different frequencies. This is caused by the large window size as it has many data points to determine the frequencies from, which also means that the frequencies will leak into the adjacent points in time. The right figure has a balanced window size, where it is both possible to determine where in time the frequency changes, but also which frequency it changes from and to. It however still shows leakage in both dimension

The window type also affects frequency leakage and frequency resolution. In figure 3.4 two different window shape types and their Fourier transformations are shown.

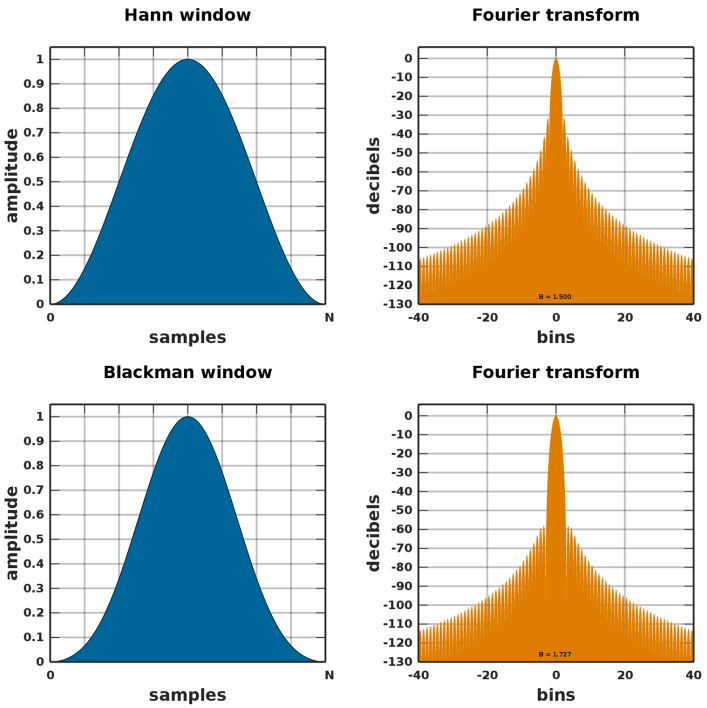

Figure 3.4: The Hann and Blackman windows and their Fourier transformations[Nie13b; Nie13a].

From figure 3.4 it can be seen that while the windows look quite similar, but their spectral shape are quite different. In the Fourier transform of the Hann window, it can be seen that the main lobe is more shallow compared to the main lobe of the Fourier transformation Blackman window. The more shallow this lobe is, the higher frequency resolution is obtained, which results in the window being better to distinguish close frequencies. But on the other hand, the distance from the top of the main lobe to the top of the adjacent lobes is small. This means it will have more spectral leakage and thereby provide amplitude to the surrounding frequencies. From the figure, it can be seen that is opposite in the Blackman window. Here the width of the main lobe is large and the distance from the top of the main lobe to the top of the adjacent lobes is large. So this results in lower frequency resolution but less spectral leakage, hence there is this trade-off.

3.2 Deep Learning

Deep learning models will be used throughout the project to both to classify a given fault based on time series data and generate synthetic data. It is therefore necessary to understand the theory behind the different elements of deep learning, which will be described in the following sections.

3.2.1 Perceptron

The simplest artificial neural network architecture is the so-called perceptron, which was invented in 1958 by Frank Rosenblatt [Ros58]. Its goal is to mimic a biological neuron, where it takes multiple input and convert it to a single output. But instead of only working with binary values, it uses numbers as inputs and output. A single perceptron looks as figure 3.5 below:

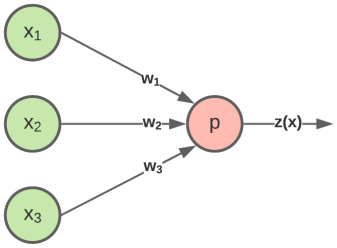

Figure 3.5: A perceptron with three inputs and hence three weights.

Here it can be seen that for each of the inputs, there is a weight associated with it. The perceptron works by computing the weighted sum of its inputs and applying an activation function, which can be seen in equation 3.3.

$$z(x) = h(x^T w) \tag{3.3}$$

Here $z(x)$ is the output of the perceptron for the input x, x^T is the column vector containing the inputs, w the row vector containing all the weights associated with the inputs, and h the activation function.

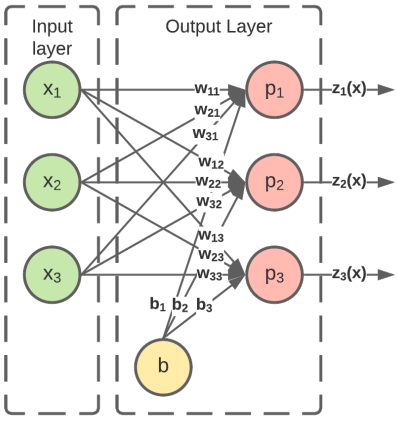

Figure 3.6: A perceptron layer with three hidden nodes, where each input is connected to every hidden node. The hidden nodes are also affected by a bias (b).

The perceptrons can also be stacked to form a perceptron layer, which is illustrated in figure 3.6. This layer has multiple perceptrons that produce an output each but all have the same inputs. However, now there exists a weight for each of the connections from the input layer to the perceptron layer. This means, that having different weights will make the outputs of the perceptrons different. Having different outputs from the different perceptrons is equal to extracting or computing different information for each of the perceptrons. The output vector z can be computed as equation 3.4.

$$Z(X) = h(XW + b) \tag{3.4}$$

Here $Z(X)$ is the output for the input X. It can be seen that X is now a matrix as it computes the output for multiple input instances. So each column is a new input instance and each row represents the inputs. This also means that the output $Z(X)$ is a matrix with a row for each input instance and a column for each of the output perceptrons. W is also a matrix for all the weights, where each row is the weight vector for the given perceptron. b is introduced as the bias term, which can be seen as an

additional input of the value 1 and its weights; *b1, b2, b3*. These biases affect how easy it is to get the neuron to trigger [Gér19].

3.2.2 Multilayer Perceptron - MLP

When combining multiple of the perceptron layers described in the previous section, they form a multilayer perceptron, also referred to as an MLP. Each of the layers is called hidden layers, except for the first and last, which respectively are referred to as the input layer and output layer. Here the outputs for each layer gets transferred as input to the next layer, to allow for further feature extraction. When all the outputs from one layer get transferred to all the neurons of the next layer it is called a fully connected layer, and is illustrated in figure 3.7 below:

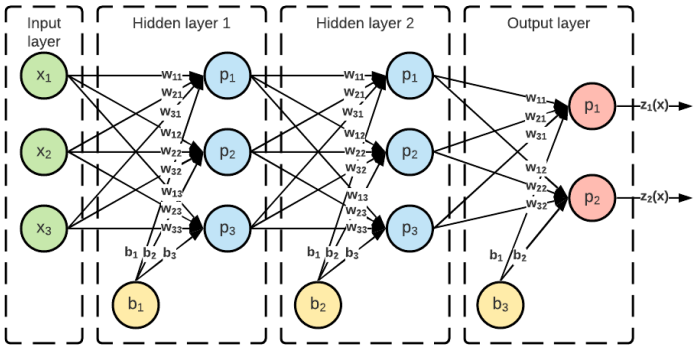

Figure 3.7: A multilayer perceptron with two hidden layers, each with three nodes, and a output layer with two nodes.

As seen from figure 3.7, it now looks more complicated with connections everywhere. But with the power of linear algebra and a little help of recursion, it can mathematically be expressed as equation 3.5.

$$\begin{aligned} Y(X) &= Z_L \\ Z_l &= h_l(A_l) \qquad\qquad l = 1, \ldots, L \\ A_l &= Z_{(l-1)}W_l + b_l \quad l = 2, \ldots, L \\ A_1 &= XW_1 + b_1 \end{aligned} \tag{3.5}$$

So as seen from equation 3.5, the goal is to calculate the output matrix $Y(X)$, which is equal to the output matrix, Z, of the output layer, L. The output matrix, Z_L, is then calculated by the activation function for that layer, h_L, on the output matrix from the previous layer, Z_{L-1}, times the weight matrix for the output layer, W_L, added with the bias vector of the output layer, b_L. Now the output matrix from the previous layer, Z_{L-1}, get calculated recursively till the first layer is reached, which will just use the input matrix, X, instead of the output from the previous layer, as there is no previous layer. This procedure of calculating the output matrix $Y(X)$, is called a forward pass [Gér19].

3.2.3 Backpropagation

Now that the input has been fed forward through the network, the network should measure how far it was from the true values. This measure is called a loss and can be calculated with different loss functions. When the loss has been calculated, it is time to update the weights of the network to minimize the loss and in that way make the network learn. To change the weights some variant of stochastic gradient descent usually is used. This means that the weights should change according to equation 3.6.

$$w^{new} = w - \eta \nabla C(w) \tag{3.6}$$

Here the new weight vector of a layer can be calculated by subtracting the gradient of the loss function, $\nabla C(w)$, for the given weight vector times a learning rate, η. To do this for all the layers, the error has to be backpropagated down through the layers of the network. This follows the general backpropagation formulas BP1, BP2, BP3, and BP4.

$$\delta^L = \nabla_a C \odot h'_L(Z^L) \tag{BP1}$$

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot h'_l(Z^l) \tag{BP2}$$

$$\frac{\delta C}{\delta b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\delta C}{\delta w^l_{j,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Here it can be seen that BP1 is used to calculate the gradient matrix, δ^L, of the output layer, where each element in the matrix corresponds to the gradient of the weight for an input instance for the output layer. $\nabla_a C$ denotes the matrix containing the partial derivatives $\frac{\delta C}{\delta a^L_{i,j}}$, which are the derivatives of each of the activations in the output layer with respect to the loss function. This matrix will element-wise be multiplied with the matrix containing the derivatives of the activation function for the output layer, denoted as $h'_L(Z^L)$.

BP2 then makes the same approach as BP1 for calculating the gradient matrix of the l^{th} layer, but on the gradients and weights from the $(l+1)^{th}$ layer. Here $(W^{l+1})^T$ denotes the transposed weight matrix of the $(l+1)^{th}$. When that matrix is multiplied with the gradient from the $(l+1)^{th}$ layer, δ^{l+1}, it moves the gradients backwards down a layer in the network, which gives a measure of the gradient, error, of the l^{th} layer. This gradient matrix of the l^{th} layer is then element-wise multiplied with the matrix containing the derivatives of the activation function for the l^{th} layer, denoted $h'_L(Z^l)$, just like in BP1.

So with BP1 and BP2 it is possible to calculate the gradients for all the layers in the network. From BP3 it can be seen that the change in the bias vector of a layer is just equal to the gradient vector of a layer. From BP4 it can be seen that the weights of a layer are changed with the activations of the $(l-1)^{th}$ layer multiplied by the gradients calculated for the l^{th} layer.

However, the δ^L and δ^l in BP1 and BP2 are matrices, as they hold the gradients for each of the input instances. So to use them for changing the biases and weights in BP3 and BP4, the average gradient for each of the biases and weights have to be calculated. This is done to calculate the true gradients of the weights. But as neural networks are most commonly trained on mini-batches, the gradients calculated are the stochastic gradients, which is the reason behind the name, stochastic gradient descent.[Nie15]

3.2.4 Activation Functions

In the previous section, it was seen that an activation function was used with the perceptrons. But what is an activation function? An activation function is basically just a function that takes in a number and translates it into some other number. So it could for instance just be a function that adds 1 to the number or a function that just outputs 1 all the time. But none of those is actually good choices of an activation function.

First of all, the activation function should represent a function where its derivative is well defined everywhere and is not 0 everywhere. This is necessary to be able to calculate the gradient no matter what the value is, and make sure that the gradient is not 0 all the time. If the derivative would be 0 everywhere, the gradient would not allow the network to learn.

Secondly, an activation function should be non-linear. Because two chained linear functions will just output another linear function. For instance if the two functions $f(x)$ and $g(x)$ looks like equation 3.7.

$$f(x) = 3x + 7 \tag{3.7}$$
$$g(x) = 4x + 2 \tag{3.8}$$

Then $f(g(x))$ would look like equation 3.9

$$(g(x)) = 3(4x + 2) + 7 = 12x + 7 + 7 = 12x + 14 \tag{3.9}$$

So without this non-linearity between the different layers in the network, then a deep network could just have been expressed as a single layer network.

Four of the most common activation functions can be seen in figure 3.8 along with their derivative.

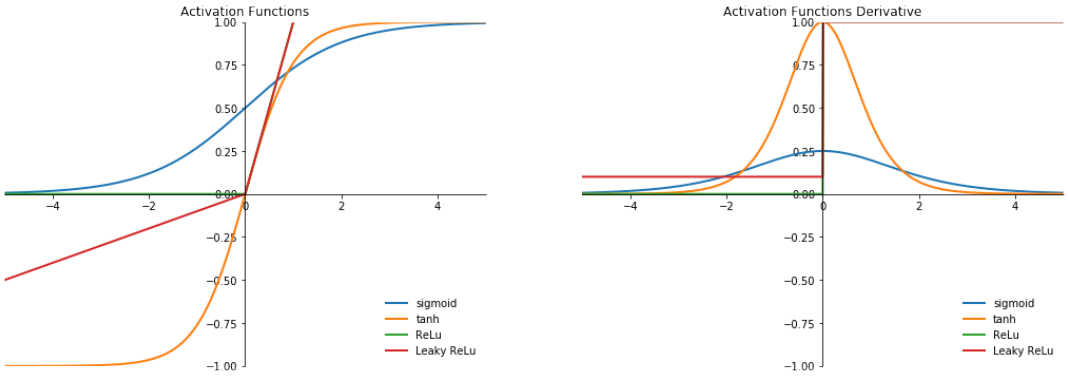

Figure 3.8: The sigmoid, tanh and ReLu activation functions and their derivative.

As it can be seen from figure 3.8, the derivative of the *ReLu-* and *Leaky ReLU* activation function is actually not well defined for 0. So the derivative for this is implementation-specific. It however works very well in practice due to its fast computation time. And the fact that it does not have a maximum output value helps to avoid the vanishing gradient problem, as the derivative will not converge to 0 for large values.

3.2.5 Convolutional Layers

While multilayer perceptrons, MLPs, show great results, they have their limitations and flaws when used for data that is not tabular. For instance when looking at image data or time series data where data points are stored in a spatial structure. An MLP would have connections between all data points meaning not taking the spatial structure into consideration. Therefore convolutional layers are introduced, as they try to extract local features by using local receptive fields.

In the example in figure 3.9 it can be seen that the feature, p_1, in the first convolutional layer gets calculated based on the input values x_1, x_2, x_3. This is because in this example a kernel of size 3 is used. This kernel is just a weight vector (matrix if in 2D) that can be slid over the input values to calculate all the local features. This slide is referred to as *stride*, which also can be set to different values than 1. As seen in the first convolutional layer of figure 3.9, p_1 gets calculated by the first three input values, while p_2 gets calculated by the input values x_2 to x_4, p_3 by x_3 to x_5 and so on. It is also illustrated in the figure how the local features get calculated by the same kernel, as each feature gets calculated with the same weights, represented by the coloured connections. This means that the weights of the kernel will be learned based on the whole time series and hence learn a general feature of the input such as an increase, decrease or bump in the values. Therefore it is necessary to use multiple kernels, such that multiple features can be learned in each layer. The reason why multiple features will be learned is due to the random initialization of the weight. The amount of different kernels to use in each layer

is referred to as *out channels* [Nie15]. A concrete example of a kernel striding over a time series can be seen in figure 3.10.

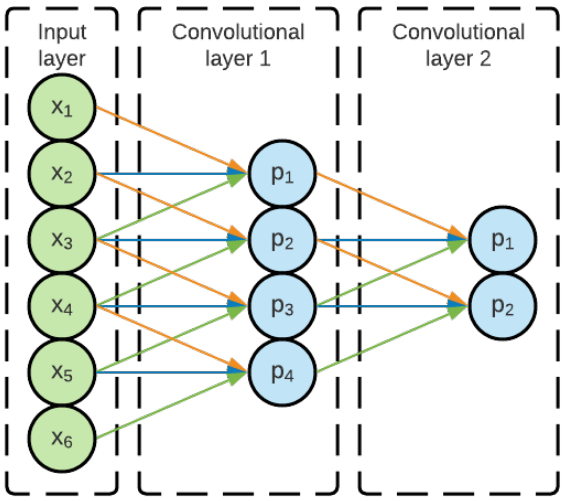

Figure 3.9: Part of a convolutional network with convolutional layers.

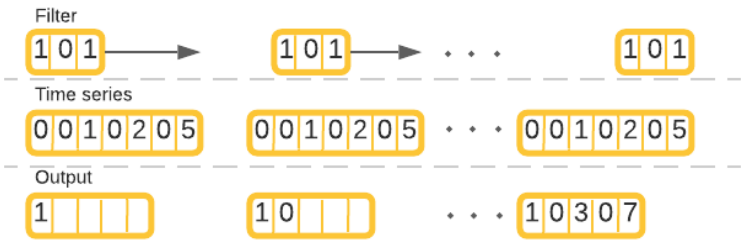

Figure 3.10: Concrete example of a filter striding over a time series.

In the convolutional layers of figure 3.9 and 3.10 it can be seen that the number of dimensions decreases. This decrease in dimensions can be omitted by introducing padding to the input. So for instance in the example from the figure, padding of one 0 can be inserted on both sides of the input layer, which would mean that the number of dimensions in the first convolutional layer would be the same as the input layer.

The exact formula for calculating the j^{th} output feature for i^{th} kernel can be written as:

$$p_{j,i} = h(b_i + \sum_{l=0}^{k-1} w_{i,l} a_{j+l}) \tag{3.10}$$

In equation 3.10 b_i denotes the bias for out channel i, $w_{i,l}$ weight l for out channel i, k the kernel size, and a_{j+l} the activation from the $j + l'th$ neuron from the previous layer. h denotes the activation function for the layer as usual. So from this it can be seen that each kernel shares the weights and bias for all receptive fields. This not only makes the kernels learn more general features across the entire time series, but it also greatly reduces the number of parameters to be learned. It should be seen that

the amount of learnable parameters in the convolutional layer does not depend on the length of the time series, which makes them computationally faster.

3.2.6 Regularization Methods

When training deep learning models on some training data the models will tweak their parameters to minimize the loss. Eventually, if the model is complex enough, the loss will approach 0 and thereby the accuracy approach 100%. However, this does not necessarily mean that the model has generalized well. This only means that the model has adapted to the training data. Therefore it is necessary to validate the model against a validation dataset that is not a subset of the training dataset. If the model generalizes well, then the loss and accuracy for the training data and the validation data would follow each other. If the model starts to perform better on the training data than on the validation data, then it is a sign of overfitting. An example of this can be found in figure 3.11, where the training loss continues to decrease for each epoch, but from epoch 10 the loss for the validation data begins to increase. This indicates that from epoch 10 the models start to overfit and adapt too much to the training data instead of generalizing.

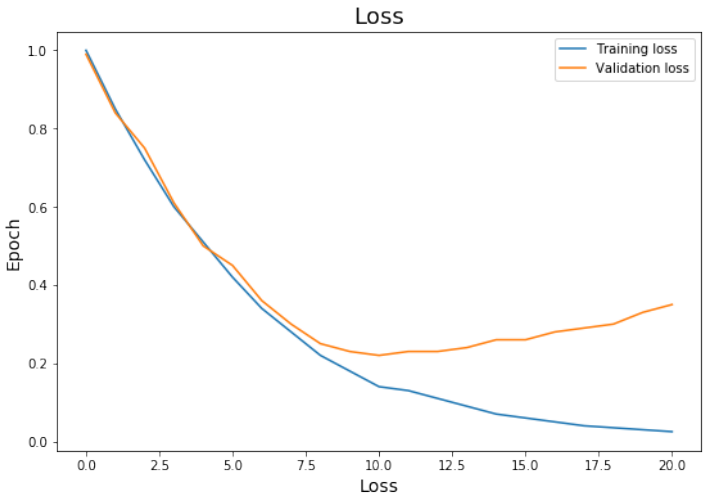

Figure 3.11: Example loss for overfitting model.

3.2.6.1 Early stopping

One method to avoid overfitting is by stopping training once the model starts to overfit. This method is referred to as *early stopping*. In the example from figure 3.11, the model would stop training at epoch 10. Of course, the model should not stop training as soon as it sees an increase in the validation loss as the loss curve can be noisy and therefore lead to too early stopping. Therefore early stopping works with something called patience. The patience of 10 would for instance mean that the model would stop training if it has not improved for more than 10 epochs on the validation dataset. Then it would roll back 10 epochs and use those parameters in the final model.

3.2.6.2 Weight Decay - L2-Regularization

While early stopping is a good method to avoid overfitting and to avoid doing unnecessary training it would be more preferred if it was possible to bring the two loss curves closer to each other, which would result in a better validation score and no overfitting. This is exactly what *weight decay* does (also referred to as *L2-Regularization*). Weight decay works by adding an additional term to the loss function, which can be seen in equation 3.11:

$$L = L_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{3.11}$$

In equation 3.11 L_0 is the original loss function and the last part is the added regularization term. The term basically adds the sum of all the squared weights in the model which are scaled by $\frac{\lambda}{2n}$. Here λ is the regularization parameter. This effectively downscales the weights in the weight updating rule from equation 3.6 by a factor of $1 - \frac{\eta\lambda}{n}$, as $\frac{\lambda}{n}w$ is added to the partial derivative of the loss function [Nie15].

3.2.6.3 Dropout

Another method to avoid overfitting but without modifying the loss function is *dropout*. Dropout essentially drops a few of its neurons in the models training process. It does that by randomly setting some elements of the input vector to 0. Each element in the input vector has the probability p of being dropped. As the weights and biases in the model are learned when only using a certain amount of the neurons, the output of the neurons must be scaled according to the dropping probability, p, such that the neurons will receive values in the same scale as they were being trained on.

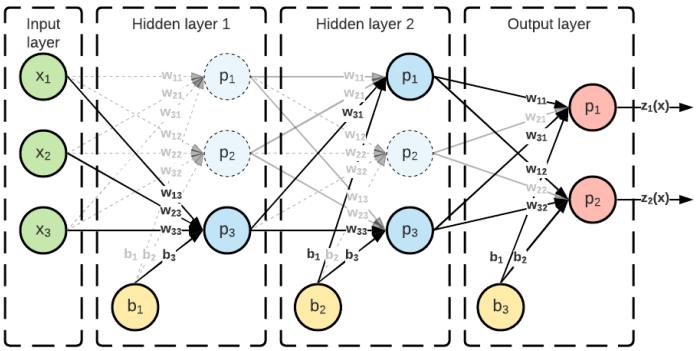

Figure 3.12: Dropout in an MLP.

In figure 3.12 it can be seen how an MLP has adapted to a random dropout. So here only one neuron in the first layer is active and two in the second layer. So this means that only the connections between those neurons are active. The next time data will be fed forward in the network, it will choose new random neurons to deactivate. The reason why dropout is regularizing is that it essentially trains multiple different models and combine them when evaluating. So it actually makes up an ensemble of different models and averaging the effects of a very large number of different models.[Nie15]

3.2.6.4 Batch Normalization

Batch normalization is also a widely applied method for training deep learning models. While it is actually not intended as a regularization method, it is affecting the network in a regularized way. Essentially batch normalization is ensuring that the outputs of a certain layer in the network is distributed around zero with a standard deviation of one. It does that by the following four formulas:

$$\mu = \frac{1}{N}\sum_{i=0}^{N-1} Z_i \tag{3.12}$$

$$\sigma = \frac{1}{N}\sum_{i=0}^{N-1} (Z_i - \mu) \tag{3.13}$$

$$Z_{i_{norm}} = \frac{Z_i - \mu}{\sqrt{\sigma^2 - \epsilon}} \tag{3.14}$$

$$\hat{Z} = \gamma * Z_{i_{norm}} + \beta \tag{3.15}$$

Here equation 3.12 and 3.13 calculated the mean and standard deviation across activations of the outputs, Z. Those statistics are then used to normalize the batch in equation 3.14. In equation 3.15 the output of the layer is calculated by applying a linear transformation with the variables γ and β. γ and β are learnable parameters that adjust the standard deviation and bias respectively.

Originally batch normalization was developed to reduce internal covariate shift [IS15] and thereby reduced the convergence time. It has later been proved that batch normalization in fact did not reduce internal covariate shift [San+18], but still evidently improved the convergence time. While the reason batch normalization improves the convergence time is still to be concluded, it is now believed that it is due to the smoothed loss landscape[San+18].

The reason why batch normalization is affecting the network in a regularized why is due to the statistics (mean and standard deviation) calculated for each batch and transformed to make those statistics fit a normal distribution with a mean of zero and a unit standard deviation. This transformation essentially infers some noise to the data, as the non-deterministic way of sampling batches leads to an instance being transformed differently in each epoch. This essentially makes it regularizing. This also means that larger batch sizes result in less regularization, as the noise inferred by the batch normalization is reduced as the true mean and standard deviation more accurately are estimated.

3.3 Classification Models

3.3.1 InceptionTime

The InceptionTime deep learning architecture was proposed by Fawaz et. al. in [Faw+20]. The authors aim to improve the training time of time series classification models without compromising the accuracy. Here the authors refer to the HIVE-COTE algorithm with high accuracy but with a computational time of $O(N^2T^4)$. InceptionTime is inspired by the traditional *ResNet* and Google's *GoogleNet* [Sze+15] where the use of very deep convolution modules first was introduced. InceptionTime's backbone is a so-called inception module that consists of the following elements:

- A bottleneck convolutional layer with kernel size and stride of 1. This significantly reduces the dimensionality of the input and reduces the number of parameters of the network.
- The output of the bottleneck is then sent through three convolutional layers in parallel. The kernel size for each of the convolutional layers varies in size. The author presents the kernel sizes of 10, 20 and 40. The input is padded beforehand to keep the output dimensionality.
- The input to the module is also passed through a max pool layer followed by a bottleneck layer. This is to make the model more invariant toward small amounts of noise and disturbances in the incoming signals.
- The output from the three convolutional layers and the maxpool-bottleneck layer is concatenated to one collected output of the module.

A figure of the inception module can be found in figure 3.13.

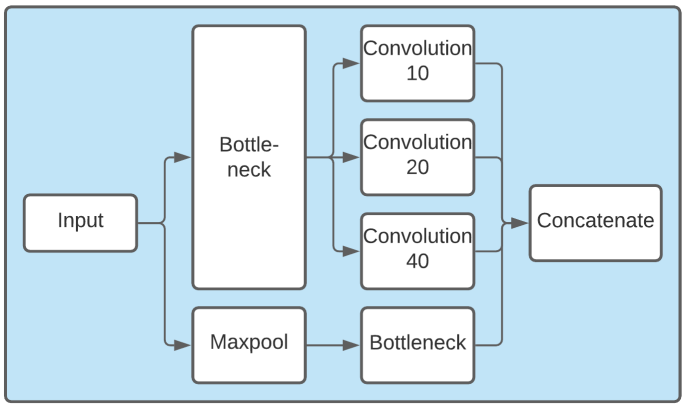

Figure 3.13: Inception time module: The bottleneck consist of a convolutional layer with a kernel size of 1 and stride of 1. The convolution layers have their representative kernel size (10, 20, 40) and a stride of 1. The Maxpool has a kernel size of 3 and a stride of 1.

These inception modules are put together in sets of 3 with a residual connection between every third module. This residual connection is to avoid the vanishing gradient problem and allow to train even deeper models where more complex features can be extracted.

Finally, a global average pooling is performed across the multivariate time dimension followed by a fully connected layer with the corresponding neurons to the number of classes. A softmax activation function is applied to get the probability of each of the classes. The full network can be found in figure 3.14.

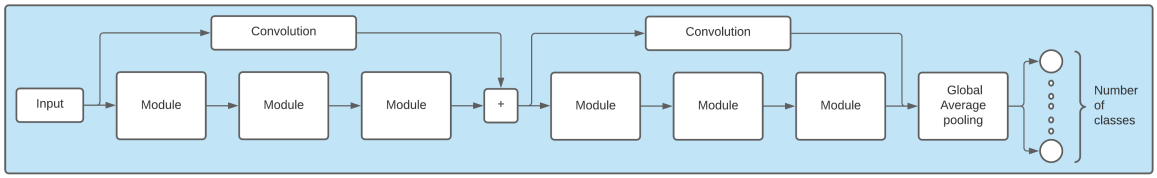

Figure 3.14: Inception time: Three modules are put in series with a residual connection to form a block. two blocks are displayed here. The network ends in a global average polling across the time dimension followed by a linear layer and a softmax activation function.

InceptionTime learns the features of the time series by applying different sized convolutional kernels which are also called different sized receptive fields (from cognitive size). The authors argue that the knowledge from image analysis can be applied to time series problems following the same arguments. In image analysis, convolutional layers are used to extract features based on areas that are close to each other. Larger receptive fields are necessary for the model to capture the context of the image. In InceptionTime this principle is used by using receptive fields of size 10, 20 and 40. This also allows the model to draw context from different sized perspectives and thereby extracting more complex features from the time series. The concept of the receptive fields can be seen in figure 3.15, where it can be seen how larger receptive fields look at more general features.

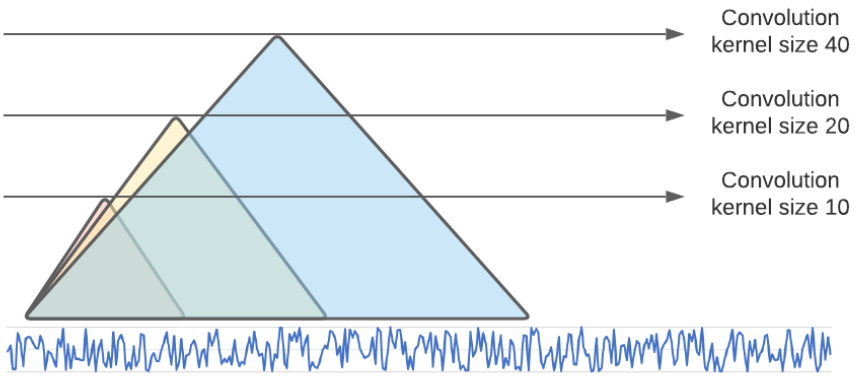

Figure 3.15: Inception time receptive field: The convolution layers have their representative kernel size (10, 20, 40) looking at different intervals of the time series thereby getting different level of context to the neighbouring values.

3.3.2 ROCKET and MiniROCKET

The classification method called Random convolutional kernel transform or ROCKET was proposed by Dempster et. al. in [DPW20]. The main purpose of the algorithm is to create a classification algorithm that beats or rivals the state of the art time series classification algorithms but at the fraction of the training time. The authors draw inspiration from deep learning where the use of convolutional networks has had increasing success in recent years, both to identify features in images but also time series as described in section 3.3.1. The idea is basically to create a large 1 layer network that transforms the incoming time series. The single-layer consists of a number of randomly initialized kernels, meaning that the sizes, weights, padding, dilation, and biases are randomly initialized. The parameters are not learned, meaning there is no training involved in the transformation of the incoming data, unlike with deep learning. Therefore a very large number of kernels can be applied without a big increase in computational time. The authors suggest a default of 10.000 kernels. The variation of these different kernels is highly spread out unlike with convolutional deep learning, where the kernels typically look very similar in each of the layers. This allows ROCKET to capture a wider variety of features in the time series. The authors highlight dilation as the most important factor for capturing these different features. The kernels then stride over the time series, each extracting features from the time series.

The kernels' parameter initialization are sampled randomly from the following options:

- **Size:** By experiment the authors reduced the number of different lengths to $7, 9, 11$.
- **Weights:** Are sampled from a normal distribution $w \sim N(0, 1)$.
- **Bias:** Are sampled from an uniform distribution. $b \sim U(-1, 1)$
- **Dialation:** Are sampled from an exponential scale where the following equation ensures that the size of the kernel output matches the input time series. $d = 2^x$, $x \sim U(0, A)$, where $A = \log_2 \frac{l_{input}-1}{s_{kernel}-1}$.
- **Padding:** For each kernel an equal chance of applying zero-padding to the incoming time series is randomly selected. The padding is depended on the size of the kernel so the middle of the kernel starts in the first element of the time series and ends in the last element: $((s_{kernel} - 1) * D)/2$.

The kernel strides over the input time series following equation 3.16, where X_i is the value at time i in the time series X, w the kernel, and d the dilation:

$$X_i * w = \sum_{j=0}^{s_{kernel}-1} X_{i+(jd)} w_j \tag{3.16}$$

For each kernel the output values are aggregated into two values, the maximum value and the proposition of positive (PPV) values, which simply is the number of values above 0 divided with the length of the time series as seen in equation 3.17.

$$ppv = \frac{1}{n} \sum_{i=0}^{n-1} [z_i > 0] \tag{3.17}$$

Here n is the length of the time series and z_i is the output values of the kernel. This results in 20.000 features if the authors' recommended number of kernels is used.

A linear classifier is then applied to the transformed data. The authors suggest a logistic regression classifier with ridge regularization for smaller datasets and a logistic regression classifier optimized with stochastic gradient descent for larger datasets. The ROCKET transformer can be see illustrated in figure 3.16.

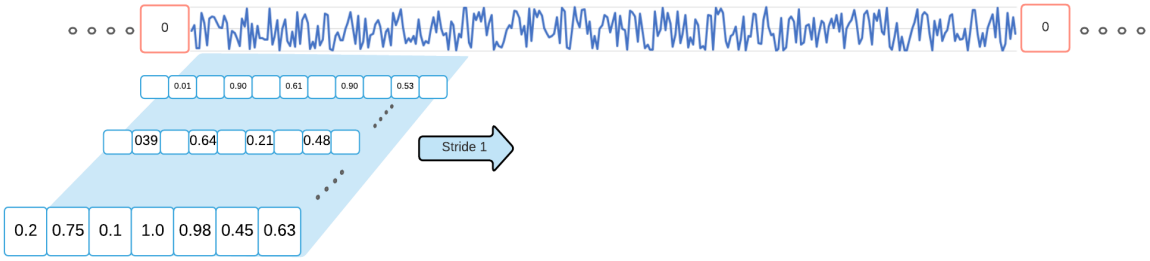

Figure 3.16: ROCKET: thousands of kernels are created with random weights (represented by the values in the cells), size, padding, dilation and bias (not illustrated in the figure). Each kernel strides over the input time series creating a feature map. The feature map is aggregated into two values the max and *ppv*.

Dempster et. al. continues their work in [DSW20] by creating the MiniROCKET transformer. The method builds upon the previously explained ROCKET transformer. The authors tried to eliminate as much of the random initialization of the different parameters and thereby making the method more deterministic and faster. They experimentally optimize the original ROCKET transformer by finding the most optimal set of hyperparameters. The hyperparameters now look as follows:

- **Size:** By experiment the kernel length is set to 9.
- **Weights:** Are not longer sampled from a normal distribution but rather set to either $-1, 2$.
- **Bias:** Are the only stochastic element in MiniROCKET. Here the bias is drawn from the output of a random training sample. The bias is calculated by convolving the kernel, W, with the random training sample, X, as follows $W * X$. A random quantile for the output is then calculated and used as the bias for the kernel, W, for the remanding training examples.
- **Dialation**: Are set to a fixed value adjusted to the input time series in accordance with the previously presented equation.
- **Padding:** Is added for every other kernel/dilation pair, so roughly half the kernels will have padding.

The use of global max pooling have also been eliminated so only PPV is used as the feature aggregated for each kernel. This results in a reduced output of half of the original ROCKET transformer. This resulting method outperforms the original ROCKET transformer in training time without compromising accuracy. The authors recommend that MiniROCKET should be used instead of the original ROCKET.

3.4 Generative Adversarial Network - GAN

The first generative adversarial network (GAN) was introduced by Ian Goodfellow in 2014 [Goo+14]. The basic idea behind this is to have a generator model learn the underlying distribution of a dataset. This is done by pitting the generator model against a discriminator network which job is to determine if the input it receives is a true data sample or a synthetic sample from the generator model. Goodfellow et. al. uses the analogy of a counterfeiter (the generator) trying to cheat a policeman (the discriminator) into believing the fake money he is producing is actually real.

The discriminator and generator are both types of neural networks as these can be very good at discovering and understanding the complex distribution present in most datasets. The generator transforms a latent noise vector z into a sample from the underlying distribution of the dataset X defined as $G(z, \theta_g)$ where G is the generator network and θ_d is the parameters of that network. The discriminator transforms the input into a single scalar representing the probability of the input being a sample taken from the true dataset, X, as $D(\hat{x}, \theta_d)$. The discriminator's job is therefore to maximize the probability of separating the true and fake samples. In the meantime the generator's job is to minimize this distance between the true and fake samples. An illustration of the relationship can be found in figure 3.17.

The discriminator gets either the true data resulting in loss function:

$$L(D(x)) = log(D(x)) \tag{3.18}$$

or the fake data from the generator resulting in loss function:

$$L(G(D(z)) = log(1 - D(G(z))) \tag{3.19}$$

The equations are derived from the cross entropy loss which calculates the difference between distributions. This results in a discriminator loss that wants to maximize expression:

$$L_D = max(log(D(x)) + log(1 - D(G(z))) \tag{3.20}$$

And the generator loss which will minimize expression:

$$L_G = min(log(D(x)) + log(1 - D(G(z))) \tag{3.21}$$

The combined loss function therefore looks as follows:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim pdata(x)}[logD(x)] + \mathbb{E}_{z \sim pz(z)}[log(1 - D(G(z)))] \tag{3.22}$$

This known as the two-player min-max game, where the value function is $V(G, D)$ and the generator, G, tries to minimize the value function while the discriminator tries to maximize it.

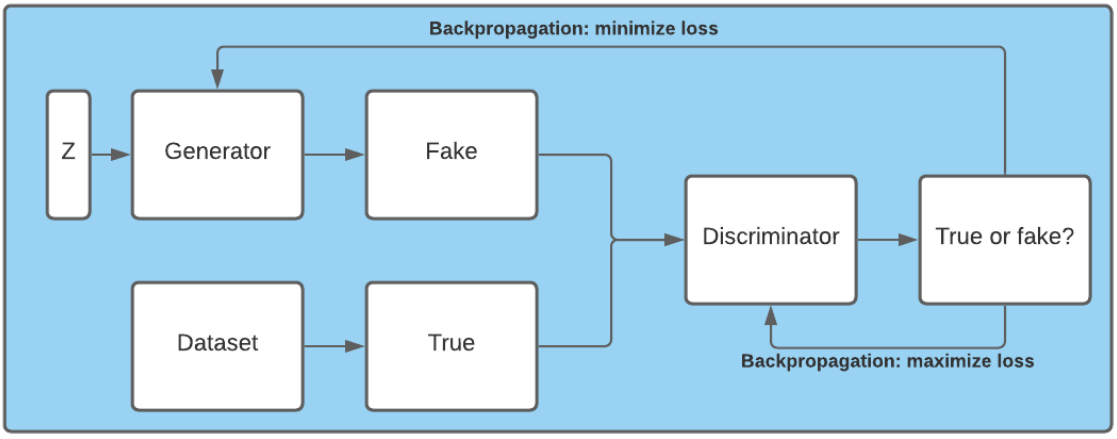

Figure 3.17: The basic idea behind a GAN. The generator network creates fake data samples trying to capture the underlying distribution of the true dataset. The discriminator is trying to maximize the distance between the two and the generator is trying to minimize it.

The difference between GANs and other popular unsupervised generative models like a *variational autoencoder*, VAE, is the way the network learn. Where the GAN is a min-max two-player game between the discriminator and the generator where the two networks try to outperform each other. VAE works by an input is fed into an encoder that encodes the input into a hidden state. A decoder then tries to decode a sample from the hidden state into a new sample. GANs have shown to outperform VAEs in most synthetic generation tasks and will therefore be used in this project.

3.4.1 Conditional GAN

Conditional generative adversarial networks are simply a way to teach the generator model from which class to generate an example from and to discriminate between the different classes [MO14]. This is done by simply providing the label for the input sample. This is in practice done using an embedding layer where the input label is embedded. The embedding layers have learnable parameters that are trained during backpropagation. This embedding layer is both implemented in the generator and discriminator so the generator learns to produce examples from a specific class and the discriminator learns to tell them apart. The embedding for the generator can be seen in figure 3.18, here the label or condition is inputted to the embedding layer and then concatenated with the input latent vector, z. An embedding layer is simply a layer that has a series of weights that based on the label inputs return a series of different values.

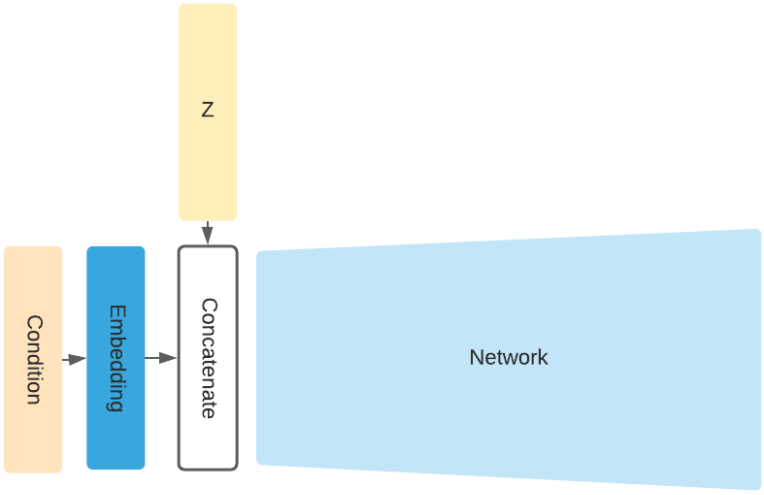

Figure 3.18: The generator does not only get the latent vector z as input but also the class for the data sample it needs to capture the distribution from. This is done through an embedding layer with learnable parameters.

3.4.2 Wasserstein GAN with Gradient Penalty - WGAN-GP

One of the problems with GANs is the phenomenon called *mode collapse*. Here the generator is able to minimize its loss by producing the same output every time and thereby not sufficiently covering the entire underlying distribution of the dataset. Another problem is unstable training as the balance between the two networks can be very sensible. The goal of the training is to find the equilibrium where the discriminator has a 50% chance of telling if the input is real or fake as it is basically just guessing. At this point, the generator has become so good at producing good samples that the discriminator no longer can identify features that tell true from fake. This is the point where both models have fully converged. These issues are occurring due to the way traditional GANs are trained by minimizing the Jensen-Shannon divergence. Arjovsky et. al. shows [ACB17] that when utilizing the Jensen-Shannon divergence to learn the probability distribution of a dataset with gradient descent some examples will occur where this will fail to converge as the loss function is not continuous. Arjovsky et. al. shows that by using the earth mover distance (Wasserstein distance) the function will in any case converge. This is the fundamental idea behind WGANs. The Wasserstein distance describes the *cost* or *energy* of converting one distribution to another. In the case of GANs, the synthetic distribution to the true distribution. This is shown in equation 3.23. Here the greatest lower bound, inf, of energy, λ, is measured to convert the generated distribution P_θ to the true distribution P_r by comparing a sample from the true distribution, x, and the synthetic, y. In the article, [ACB17] they use the term *critic* instead of *discriminator* as the discriminator no longer just returns rather or not the input is true or fake but instead a scalar describing to what degree the input is true or fake.

$$W(P_r, P_\theta) = \inf_{\lambda} \mathbb{E}_{x,y} ||x - y|| \tag{3.23}$$

The Wasserstein loss function is derived from the Kantorovich-Rubinstein duality [Gul+17] seen in equation 3.24. Here the first term is the sample, x, of some true distribution of data where the discrim-

ination of that sample is returned, minus a sample $\hat{x}$ from a generator gone through the discriminator. Note the constraint on of $||D||_L \leq 1$ this term ensures that the norm of the gradient of the discriminator stays between -1 and 1. This is to ensure that the Lipschitz continuity is preserved. This is, however, to put it in the authors' own words, "a terrible idea!" as it creates instability for the weights of the network to reach convergence.

$$\min_G \max_{||D||_L \leq 1} = \mathbb{E}_{\sim P_r}[D(x)] - \mathbb{E}_{\hat{x} \sim P_\theta}[D(\hat{x})] \tag{3.24}$$

A better solution to keeping the Lipschitz continuity was proposed by Gulrajani et. al. in [Gul+17]. They suggested penalizing the gradient by adding a regularization term to the loss function as seen in equation 3.25. Here the λ is a strength/regulating term (set to ten in the paper). $|| \bigtriangledown_{\hat{x}} D(\hat{x})||$ is the gradient of the discriminator of the sample $\hat{x}$. $\hat{x}$ is sampled from a implicit distribution $P_{\hat{x}}$ that represents the straight line between sampled points in the true and synthetic distribution P_r and P_θ. In other words, an interpolated sample where a random percentage $\epsilon \leq 1$ of the true data and $\epsilon - 1$ of the fake is combined and gone through the discriminator.

$$L = \mathbb{E}_{x \sim P_r}[D(x)] - \mathbb{E}_{\hat{x} \sim P_\theta}[D(\hat{x})] + \lambda \mathbb{E}_{\hat{x} \sim P_{\hat{x}}}[(|| \bigtriangledown_{\hat{x}} D(\hat{x})||_2 - 1)^2] \tag{3.25}$$

3.4.3 uTSGAN

The uTSGAN architecture was proposed by Smith et al. in [SS21]. It was a continuation of the authors' previously proposed network for time series generation TSGAN [SS20]. The TSGAN network utilizes two WGAN-GP networks $WGAN_x$ and $WGAN_y$ as seen in figure 3.19.

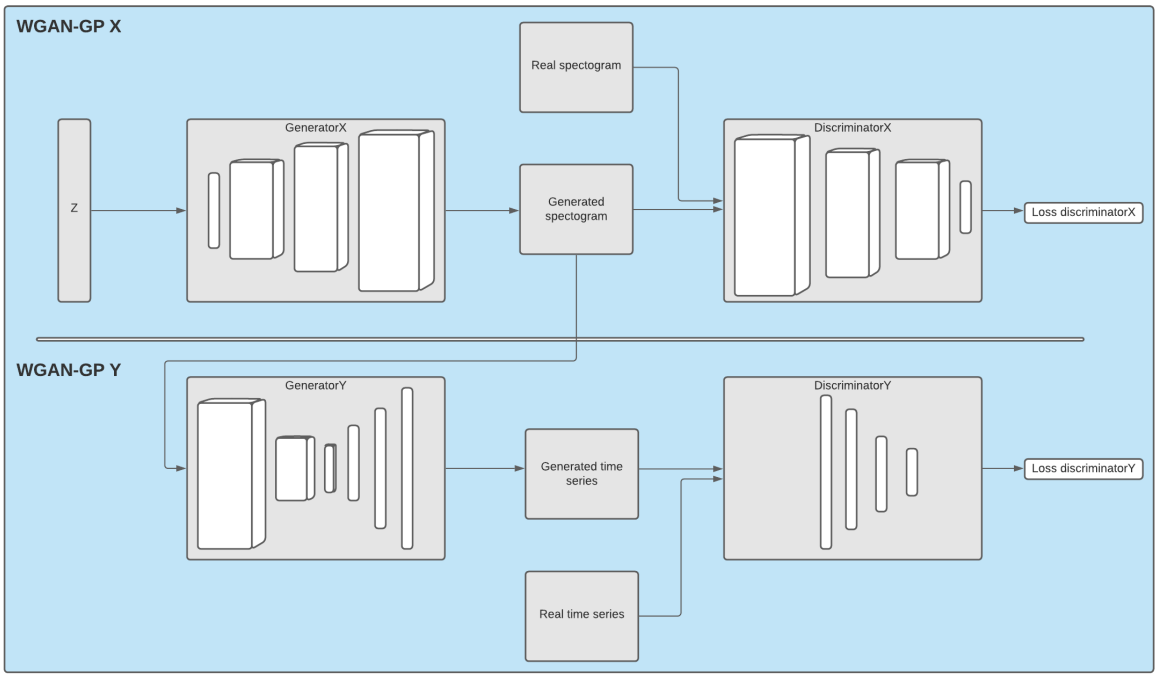

Figure 3.19: The architecture of the uTSGAN.

A random latent vector z, sampled from a Gaussian normal distribution, is created and send through generatorX. Here the one-dimensional vector is upscaled using transposed convolutional layers into

a spectrogram. The true time series is transformed into a spectrogram using a short-time Fourier transform as described in section 3.1.1. DiscriminatorX is then receiving samples of the generated spectrograms and the true spectrograms. This part of the network is not conditioned on the label. The network, therefore, does not know for what class it should produce a spectrogram. The fake spectrograms are then used as the input for generatorY. Here the input spectrograms are conditioned on the class that the spectrogram was created from. This forces the generatorY to generate time series samples that share the same characteristics as the time series from that class. GeneratorY encodes the two-dimensional spectrograms into a one-dimensional time series. The authors argue that this method utilizes the characteristics of convolutional layers first discovered in image processing:

It's a known fact that convolutional layers in a deep network become bias to texture-based features in an image which we hope to exploit with the fluctuations of the frequencies in the spectrograms creating sharp textures, key peak frequencies, and smooth textures in others [SS20].

DiscriminatorY now receives input from both the generated time series and the true time series. In the paper *Conditional GAN for timeseries generation* [SS20] the two GANs are trained in sequel with two independent loss functions based on the work done by gulrajani et al. [Gul+17] described in section 3.4.2:

$$L_x = \mathbb{E}_{x\sim P_r}[D_x(x)] - \mathbb{E}_{x\sim P_\theta}[D_x(x)] + \lambda\,\mathbb{E}_{\hat{x}\sim P_{\hat{x}}}[(||\bigtriangledown_{\hat{x}}\,D(\hat{x})||_2 - 1)^2] \tag{3.26}$$

$$L_y = \mathbb{E}_{x\sim P_r}[D_y(x)] - \mathbb{E}_{x\sim P_\theta}[D_y(G_x(G_y(x)))] + \lambda\,\mathbb{E}_{\hat{y}\sim P_{\hat{y}}}[(||\bigtriangledown_{\hat{y}}\,D(\hat{y})||_2 - 1)^2] \tag{3.27}$$

Note here that the only difference between the two loss functions is that the second loss function L_y is dependent on the output from the first and second generator. This however results in slow training times as the generatorX needs to complete a full iteration before the GeneratorY can start its training. This problem was however solved in [SS21] where the two loss functions, seen in equation 3.26 and 3.27, were combined into a unified loss function (hence the name uTSGAN). The resulting equation 3.28 is a simple average of the two previous loss functions. This allows the discriminators to train in parallel resulting in faster training.

$$L = \frac{L_x + L_y}{2} \tag{3.28}$$

3.5 Evaluation Metrics

3.5.1 Evaluation Metrics for Classification Models

To evaluate the performance of the different classification models, four primary metrics are used: *accuracy*, *precision*, *recall*, and *f1-score* [Aga21].

3.5.1.1 Accuracy

The accuracy metric is the most straightforward and easy to understand of the proposed metrics. It is simply the proportion of correctly classified samples to the total number of samples:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.29}$$

One disadvantage with the accuracy metric is however that a model trained on a very unbalanced datasets can easily be misleading. If 90% of the samples in the data set eg. belongs to one class the model will achieve a 90% accuracy by only predicting that one class. This metric therefore needs to be combined with others that takes this into consideration.

3.5.1.2 Precision

The precision metric tells the proposition of the predicted positives that actually are positive. It is possible to see from the equation below that if no false positives are registered, the precision would be 1.

$$Precision = \frac{TP}{TP + FP} \tag{3.30}$$

3.5.1.3 Recall

Recall tells something about the proportion of actual positives that are predicted correctly. In the equation below it can be seen that if there are no false negatives the recall is 1.

$$Recall = \frac{TP}{TP + FN} \tag{3.31}$$

3.5.1.4 F1-score

The f1-score combines the precision and recall metrics as follows:

$$f1\text{-}score = 2\frac{Precision \cdot Recall}{Precision + Recall} \tag{3.32}$$

The f1-score ensures the balance between the precision and recall as if either one of the metrics is low the f1-score is low. In more manageable terms the f1-score is a way to ensure that the model is precise but also good at getting a lot of true positives.

3.5.1.5 Micro- and Macro Average

When working with multi-class problems it is not possible to use the just described metrics directly. So either the micro- or macro average must be calculated. The micro average is where the true positives, true negatives, false positives, and false negatives are summed across the classes before calculating the precision, recall, and f1-score. For the macro, however, the precision, recall, and f1-score are calculated for each class, and then the average of each of the metrics are taken. The micro average weights every data instance equally, where the macro average weights every class equally. Therefore macro average is generally preferred for imbalanced datasets.

3.5.2 Evaluation Metrics for Generative Models

When evaluating a generative adversarial network, which is an unsupervised model, it is not possible to use the earlier described metrics. The goal for the GAN is to create synthetic data that is similar to the real. It is therefore in the interest to measure the similarity between the synthetic data and the real data.

3.5.2.1 Dynamic Time Warping - DTW

One method to calculate the similarity between time series is the dynamic time warping method, also referred to as DTW. DTW works by calculating the euclidean distance between two time series, but it allows to use the distance between data points that are at different timestamps, but still with the constraint that data points only can be compared sequentially. This is illustrated in figure 3.20 where it can be seen in the right plot how the DTW allows using distances that ”warps” in time compared to the simple euclidean distance.

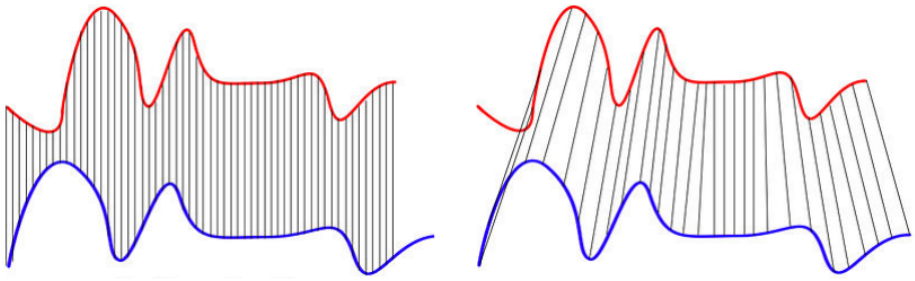

Figure 3.20: Example of euclidean distance versus DTW distance[Xan11].

For instance to calculate the DTW distance between the two time series in figure 3.21, then it is needed to calculate the cost matrix between the two time series. It can be done with the formula in equation 3.33 [Mis20]:

$$M(i,j) = |X(i) - Y(j)| + min(M(i-1,j-1), M(i,j-1), M(i-1,j)) \tag{3.33}$$

Here M is the cost matrix and X and Y are the two time series. If this equation is used for the time series from figure 3.21, then it would produce the cost matrix in table 3.1.

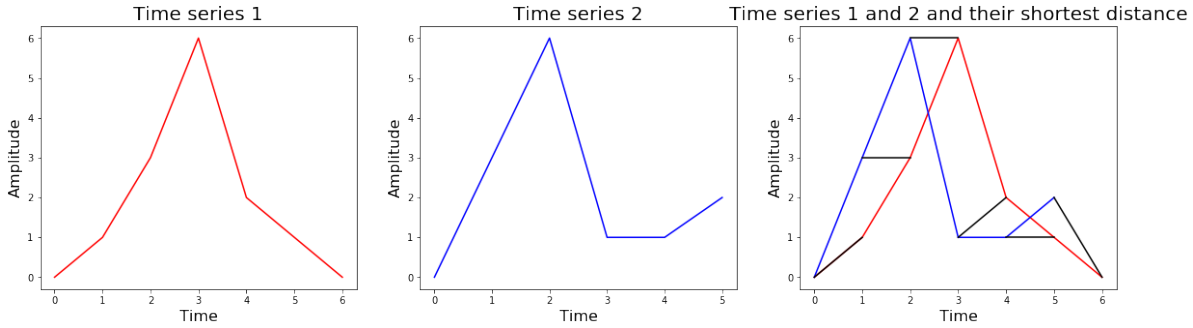

Figure 3.21: Two times series examples of different length that should be matched with DTW. In the right plot the DTW matching can be seen in black.

Table 3.1 also shows the shortest path from corner to corner, which indicates the points that form the shortest distance between the two time series. So in this case the path, π, would be equal to $[(0,0),(1,0),(2,1),(2,3),(3,4),(4,5),(5,6)]$.

	0	1	3	6	2	1	0
0	0	1	4	10	12	13	13
3	3	2	1	4	5	7	10
6	9	7	4	1	5	10	13
1	10	7	6	6	2	2	3
1	11	7	8	11	3	2	3
2	13	8	8	12	3	3	4

Table 3.1: Cost matrix for the two time series from figure 3.21. Time series 1 is in the top row and time series 2 is in the leftmost column.

So to calculate the actual distance between the two time series, the formula in equation 3.34 is used, where X and Y are the two time series and π is the path, which is the shortest path and also highlighted in table 3.1 [tsl17].

$$DTW(X,Y) = \sqrt{\sum_{(i,j)\in\pi} (X_i - Y_j)^2} \tag{3.34}$$

So for this example the actual distance is equal to:

$$DTW(X,Y) = \sum(0,1,0,0,1,0,4) = 2.45 \tag{3.35}$$

This means that the distance between the two time series is 2.45. So, a smaller distance is equal to a more similar time series, hence a small DTW distance indicates that a GAN is good at generating synthetic data.

As the synthetic data will be used for improving a model by increasing the training data and balancing the dataset, then a GAN that produces synthetic data with a DTW distance of 0 would be redundant. Therefore the real value of the synthetic data generated by a GAN can only be measured by the delta in the performance of a model trained with the original- and synthetic data.

The different theories used throughout the thesis have now been presented. The theories can roughly be divided into the *classification algorithms*, *the generative adversarial networks* and the *evaluation metrics*.

CHAPTER 4

Design and Implementation

This coming chapter will cover the design and implementation of the solution for the problem defined in section 1.2. It will start by uncovering the use cases that it should support and see which steps are necessary to support these in the best possible way. The design will be done in the context of an operational setting, meaning that it is designed and implemented such that it can be used in operation. It will be described how the steps are designed and implemented and then present the design and implementation of the different classification models as well as the generative model.

4.1 Complete System Overview

The complete solution should support two elementary things, namely training the model and making predictions with the model. To support those two cases a pipeline has to be designed which should be able to collect some data, preprocess it, and then either train a model on it or make a prediction for the data.

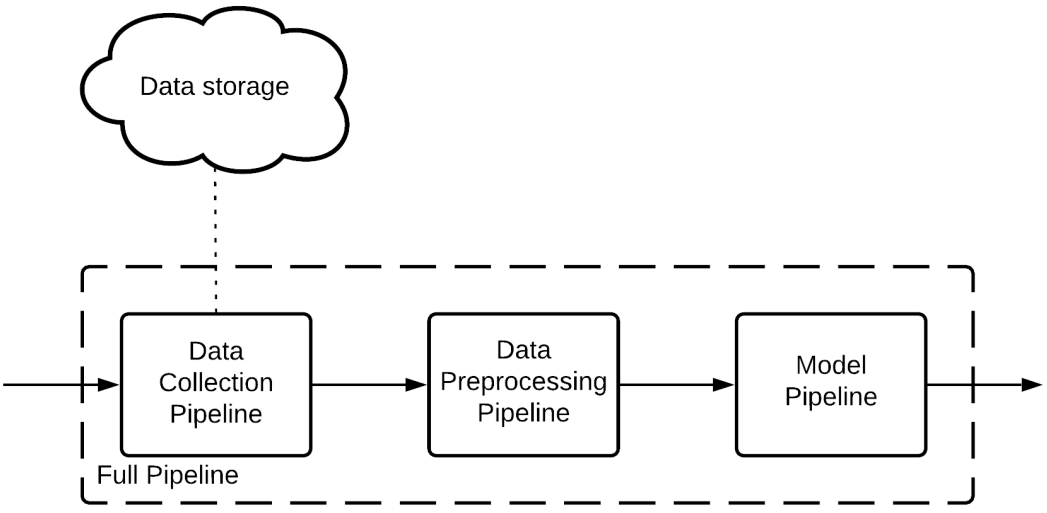

Figure 4.1: The high level pipeline that is needed to support the training of a model and also use it to make predictions.

As seen from figure 4.1 there should be some kind of event that triggers the pipeline. This event could be a technician who tells the system that he has experienced some odd behaviour or it could be the system itself that has detected an anomaly in the sensor readings as described in chapter 2. When

the pipeline gets triggered it should get the sensor data from around the stated time for the given location. It should then preprocess the data such that it is consumable for the classification model. The model should then make a fault prediction by outputting the probabilities for the different known faults. The pipeline could also be triggered to retrain when new data and/or faults are registered.

4.2 Pipeline

4.2.1 Pipeline Design

As seen from figure 4.1, the full pipeline consists of three steps, where each step is another pipeline itself. This means that the pipeline should be able to support nested pipelines. It can also be seen that the pipeline should support multiple steps, where it will send the output data from one step into the next one. Finally, the pipeline should also be able to support a step that includes a classification model, which should be able to either be trained on the input data or make a prediction on the input data. To sum up, the pipeline should be able to support the following operations:

- Fit
- Transform
- Predict
- Save
- Load

fit has the purpose of fitting the pipeline to the input dataset, which is equivalent to fitting each step in the pipeline. The *transform* method serves the purpose of transforming the input data based on the fitted pipeline. This split of fitting and transforming is necessary to be able to reuse the statistics of the train data to transform the validation data. This could e.g. be normalizing the validation data based on the mean and standard deviation of the training data. The *predict* operation, as the name implies, will make it possible to make a prediction based on the input data which will output the probability distribution of the different faults. The *save* operation can then save the calculated statistics and the parameters of the classification model derived in the *fit* operation, such that it can be loaded again with the *load* operation, without having to refit the pipeline.

A technical design that supports these needs, are presented in the UML diagram in figure 4.2. It can be seen that the *Pipeline* object contains the five methods along with the attribute, *steps*. The *steps* attribute is the list that holds all the pipeline steps that exist in the pipeline in sequential order. It can be seen from the UML diagram that the steps can also include a whole new pipeline. The *save* and *load* methods will call the *save* and *load* methods of the steps, but it will also make sure the files are saved and loaded in a folder for the particular *Pipeline* object.

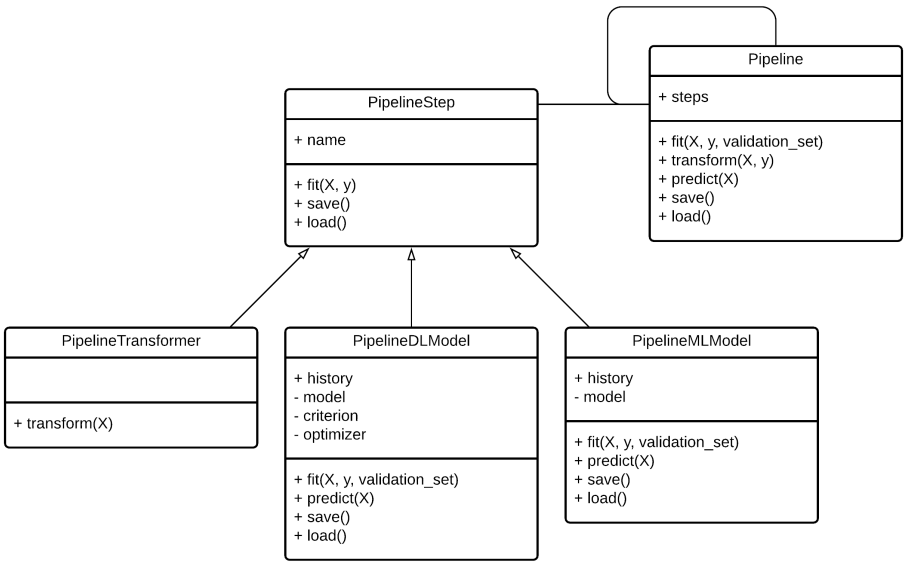

Figure 4.2: The UML diagram of pipeline and the steps in the pipeline.

From figure 4.2 it can also be seen there can exist three types of *PipelineStep*, namely a *PipelineTransformer*, *PipelineDLModel*, and a *PipelineMLModel*. The fundamental difference between the objects is that the *PipelineTransformer* should implement a *transform* method, while the *PipelineModels* both should implement a *predict* method. The difference between the *PipelineDLModel* and the *PipelineMLModel* is the fact that a deep learning model uses a criterion and an optimizer to train the model, which the methods of the object must accommodate for. The *PipelineModels* also include a *history* attribute, which is used to store the performance of the model, such that it can be analysed after training. The ability to handle both traditional machine learning models and deep learning models are part of the reason why the pipeline system was implemented. This ensures flexibility to deploy different models in an operational context. One could argue that there is no difference between the *transform* and *predict* methods except the naming, but this provides a better understanding of what should go into the different methods in the implementations [Bui+13].

4.2.2 Pipeline Implementation

The implementation of this design follows the UML diagram in figure 4.2, where the main elements will be covered in the code listings in this section.

The *Pipeline* class has a constructor which among others sets the *steps* and *step_names* attributes. These attributes are set with the *steps* parameter in the constructor, which is given by a list of tuples, each consisting of a name for the step and the *PipelineStep* itself to follow the same convention as the Scikit-learn Pipeline [Bui+13]. This parameter gets unzipped into two different attributes, namely one for the names and one for the steps themselves. The *Pipeline* class also contains the *transform* method, which can be seen in listing 4.1.

```
1 def transform(self, X, y):
2     for step in self.steps[:-1]:
3         X = step.transform(X, y)
4
5     last_step = self.steps[-1]
6     if isinstance(last_step, (Pipeline, PipelineTransformer)):
7          X = last_step.transform(X, y)
8
```

```
9       return X
```

Listing 4.1: The *transform* method of the *Pipeline* class.

As the code in 4.1 shows, the *transform* method of the *Pipeline* class starts by calling the transform method for all the steps in the pipeline, except for the last step. For each of the steps, it overrides the X variable with the return value of the transformation of each step. This is done so each step transforms based on the output from the previous step. It can furthermore be seen that a transformation also will be done for the last step in the pipeline if the step is of type *Pipeline* or *PipelineTransformer*, otherwise, it will just return the transformed variable.

The *predict* method of the *Pipeline* class follows the exact same structure as the *transform* method, except that it will call *predict* on the last step, as it is assumed the last step of the pipeline is of type *PipelineModel*.

The last method within the *Pipeline* class is the *fit* method. It follows the same principles as the *transform* method but with some additions to it to allow it to pass a validation dataset along the pipeline. This is so the performance of the *Pipeline* can be measured on validation data and make it possible to determine an early stopping criteria for the *PipelineDLModel*. The *fit* method of the *Pipeline* class can be seen in listing 4.2:

```
1  def fit(self, X_train, y_train, validation_set=None):
2
3      if validation_set:
4          X_validation, y_validation = validation_set
5
6          for step in self.steps:
7              if isinstance(step, PipelineTransformer):
8                  step.fit(X_train, y_train)
9                  X_train, y_train = step.transform(X_train, y_train)
10                 if not isinstance(step, Synthesizer):
11                     X_validation, y_validation = step.transform(X_validation, y_validation)
12
13             elif isinstance(step, Pipeline):
14                 X_train, y_train, X_validation, y_validation = step.fit(X_train, y_train,
                       validation_set=(X_validation, y_validation))
15
16             elif isinstance(step, (PipelineDLModel, PipelineMLModel)):
17                 step.fit(X_train, y_train, validation_set=(X_validation, y_validation))
18
19         return X_train, y_train, X_validation, y_validation
20
21     else:
22         for step in self.steps:
23             if isinstance(step, PipelineTransformer):
24                 step.fit(X_train, y_train)
25                 X_train, y_train = step.transform(X_train, y_train)
26
27             elif isinstance(step, Pipeline):
28                 X_train, y_train = step.fit(X_train, y_train)
29
30             elif isinstance(step, (PipelineDLModel, PipelineMLModel)):
31                 step.fit(X_train, y_train)
32
33         return X_train, y_train
```

Listing 4.2: The *fit* method of the *Pipeline* class.

As listing 4.2 shows, the method looks more comprehensive. The method is essentially divided into two parts, where one part is where the validation dataset has been set in the parameter, and one part where it has not. The part where the validation dataset is not set begins on line 21. Here it can be seen that for each of the steps in the pipeline it will call the *fit* method of the step. If the step is of type *PipelineTransformer* it will also call *transform* afterwards and overwrite the variables with the outcome of the transformation. This is done to make sure that the next step in the pipeline will fit according

to the expected input, which is the transformed input from the step. The reason that it will not call *transform* if the step is of type *Pipeline* is that it is done implicitly in the *fit* method of the pipeline.

The part of the method that handles the case where the validation dataset is passed goes from line 3 to 19. Here it follows the exact same pattern, but it additionally passes the validation data through the transformation if the step is of type *PipelineTransformer*. Otherwise, it just passes the validation data further into the nested pipelines and the *PipelineModel*. Furthermore, it can be seen on line 10-11 that it will only transform the validation dataset if the step is not of type *Synthesizer*. The *Synthesizer* is a preprocessing step, which will be explained in section 4.2.3. But the intention of the *Synthesizer* is to concatenate the input data with synthetic data that will be generated based on a specified generative network. The validation dataset should therefore not pass through that transformation. The full implementation of the *Pipeline* class can be found in the Appendix in listing A.1.

4.2.3 Pipeline Steps

The *PipelineStep* class is then implemented to support the methods of the *Pipeline* class and follow the structure defined in section 4.2.1. The class contains a constructor, which just serves the purpose of setting the name of the object. Furthermore, it contains the *set_parameter* method, which can be used by the *GridSearch* class to set an attribute of the *PipelineStep*. This will be explained in a future section. The *PipelineStep* class also contains methods for saving and loading, where it just saves or loads all the public variables of the object in a pickle file. The whole implementation of the *PipelineStep* class can be found in the Appendix in listing A.2.

The *PipelineMLModel* is a bit more comprehensive as it must support the fitting and performance measuring of the machine learning model. The constructor of the object just sets the following attributes:

- ___model
- history
- best_model
- best_score

The *___model* is the machine learning model itself. The double underscore just indicates that it is a private attribute and it is therefore not saved with the save method of the superclass, as it should be saved in another fashion. The *history* attribute is a dictionary that holds all the performance metrics. The *best_model* holds a copy of the model, and the validation f1-score is stored in the *best_score* attribute. The reason that the class has a *history*, a *best_model*, and a *best_score* attribute, is that it follows the same convention as the *PipelineDLModel* which will be described shortly. Besides the constructor, the class contains the following method for calculating the metrics:

```
1 def calculate_metrics(self, y_pred, y_true):
2     report = sklearn.metrics.classification_report(y_true, y_pred, output_dict=True)
3     report = report.get('macro avg', report)
4     accuracy = sklearn.metrics.accuracy_score(y_true, y_pred)
5 
6     return accuracy, report['precision'], report['recall'], report['f1-score']
```

Listing 4.3: The *calculate_metrics* method of the *PipelineMLModel* class.

The *calculate_metrics* method showed in listing 4.3 takes in two one-dimensional arrays as parameters, where one contains the predicted class indices and the other the true class indices, which it uses to calculate the accuracy, precision, recall, and f1-score. As it can be seen from line 3 in the listing the macro average is calculated for a multi-classification problem. This method for calculating the metrics is used in the *fit* method of the *PipelineMLModel* class, to calculate the performance of the model after training. The skeleton for the *fit* method can be seen in listing 4.4 below:

```
1  def fit(self, X_train, y_train, validation_set=None):
2      # Fit
3      self.__model.fit(X_train, y_train)
4
5      # Calculate metrics for training set
6      y_hat_train = self.__model.predict(X_train)
7      accuracy_train, precision_train, recall_train, f1_train = self.calculate_metrics(
           y_hat_train, y_train)
8
9      # Set history values for training metrics
10     ...
11
12     if validation_set:
13         X_validation, y_validation = validation_set
14
15         # Predict with validation set and set history with f1 score
16         y_hat_validation = self.__model.predict(X_validation)
17         accuracy_validation, precision_validation, recall_validation, f1_validation = self.
               calculate_metrics(y_hat_validation, y_validation)
18
19         # Set history values for validation metrics
20         ...
21
22         self.best_score = f1_validation
```

Listing 4.4: The *fit* method of the *PipelineMLModel* class.

As line 3 in listing 4.4 shows, the *fit* methods starts off by calling the regular fit method for the provided machine learning model. On line 6-7 it then uses the fitted model to make a prediction on the training dataset and then calculate the performance metrics for it. If a validation dataset is provided as a parameter in the *fit* method, then it will do the exact same thing for the validation dataset, as seen on line 16-17. On line 22 it sets the best score of the model to the f1-score for the validation dataset.

The last important method of the *PipelineMLModel* class is the *predict* method, which just calls the *predict_proba* method of the machine learning model. This returns a probability distribution for each of the possible classes, instead of just the index of the class with the highest probability. The full implementation of the *PiplineMLModel* class can be found in appendix A.3.

The structure of *PipelineDLModel* is similar to the structure of the *PipelineMLModel* but with some additions to it. Firstly, it contains the following additional attributes:

- ___optimizer
- ___criterion
- ___epochs
- best_epoch
- batch_size
- shuffle
- patience

The *___optimizer* is used to hold the *PyTorch* optimizer that will be used to optimize the parameters of the model. The optimizer will optimize based on the loss calculated by the *___criterion* attribute, which should be a *PyTorch* loss function. The *___epochs* attribute denotes the maximum number of epochs the model should be trained, where the training can be terminated earlier if the early stopping criteria is met. The early stopping criteria is met when the model has not improved its performance for more than a certain amount of epoch, which is specified by the *patience* attribute. This is why the *best_epoch* attribute is needed, as it will hold the epoch of when the best model was achieved. The

other part that is fundamentally different is the *fit* method, as it implements the whole training loop. The skeleton of the *fit* method for the *PipelineDLModel* can be seen in listing 4.5:

```
1  def fit(self, X_train, y_train, validation_set=None):
2
3      # Create the dataset and dataloader
4      dataset = PytorchTimeSeriesDataset(X_train, y_train)
5      dataloader = DataLoader(dataset, batch_size=self.batch_size, shuffle=self.shuffle)
6
7      if validation_set:
8          X_validation, y_validation = validation_set
9
10     # Run training loop
11     for epoch in range(self.current_epoch, self.__epochs):
12         # Set running performance metrics
13         ...
14
15         for i, (input, labels) in enumerate(dataloader):
16             # Zero the parameter gradients
17             self.__optimizer.zero_grad()
18             # Forward
19             outputs = self.__model(inputs)
20             # Calculate loss
21             loss = self.__criterion(outputs,  labels.argmax(dim=1).long())
22             # Backward + optimize
23             loss.backward()
24             self.__optimizer.step()
25             # Calculate metrics and add to running metrics
26             accuracy, precision, recall, f1 = self.calculate_metrics(outputs, labels)
27             ...
28
29         # Average the metrics overall batches and append values to history
30         ...
31
32         # Also calculate for validation set
33         if validation_set:
34             with torch.no_grad():
35                 # Calculate validation metrics and add to history
36                 y_validation_pred = self.__model(X_validation)
37                 loss_validation = self.__criterion(y_validation_pred, y_validation.argmax(dim
                       =1).long())
38                 accuracy_validation, precision_validation, recall_validation, f1_validation =
                        self.calculate_metrics(y_validation_pred, y_validation)
39                 ...
40
41                 # If validation score is better than best, save that model
42                 if f1_validation > self.best_score:
43                     self.best_epoch = epoch
44                     self.best_score = f1_validation
45
46                 # Early stopping criteria met. Break training loop
47                 elif epoch > self.best_epoch + self.patience:
48                     break
```

Listing 4.5: The *fit* method of the *PipelineDLModel* class.

It can be seen from line 4-5 in listing 4.5 that the *fit* method starts by creating a dataloader that uses the batch size set for the *PipelineDLModel* as well as shuffles the dataset if set. On line 7-8 it then unpacks the validation dataset into separate variables for the input and labels. On line 11 the training loop starts, which will run for as many epochs as specified. On line 13 the method initializes the running performance metrics to 0 for the training dataset which are used to calculate the average metrics for all the batches in the training data at each epoch. On line 15-27 the training of each batch is done. It starts by making sure the gradients of the optimizer are set to zero. This is to make sure that the optimizer only will take a step according to the gradients of the current batch. Now the input training data is fed forward through the network, and the result of the network is kept in the variable

outputs. Now the loss is calculated between the predicted output and the true labels with the specified loss function as seen on line 21. On line 23 it then backpropagates to compute the gradients for each of the parameters in the model with respect to the loss, such that the optimizer can take a step in the direction that minimizes the loss for the current batch, which can be seen on line 24. On line 26, it then calculates the performance metrics for the current training batch, which will be added to the running performance metrics. After the epoch, the average training performance metrics are calculated and added to the history dictionary.

On line 34 it will then also feed the validation input through the network and calculate the performance metrics for the validation dataset. Here it is important that it gets done without calculating the gradients as this would lead to the network also being trained on the validation dataset and thereby introduce a bias in the validation performance metrics, hence line 37 in the listing. On line 42 the method checks if the achieved validation f1-score is better than the current best score. If it is, then it will set the *best_epoch* and *best_score* attributes to the new values and save the model. If the model does not perform better, then it will check if it has not performed better for more than a certain amount of epochs, specified by the *patience* attribute. The full implementation of the *PipelineDLModel* can be found in appendix A.4.

4.2.4 Grid Search

To ensure that a model reaches a close to optimal minima in its hyperparameter loss landscape a grid search have been created. Here the best combination of hyperparameter values are selected experimentally by testing the performance of a model with different ranges of values. This is illustrated in figure 4.3.

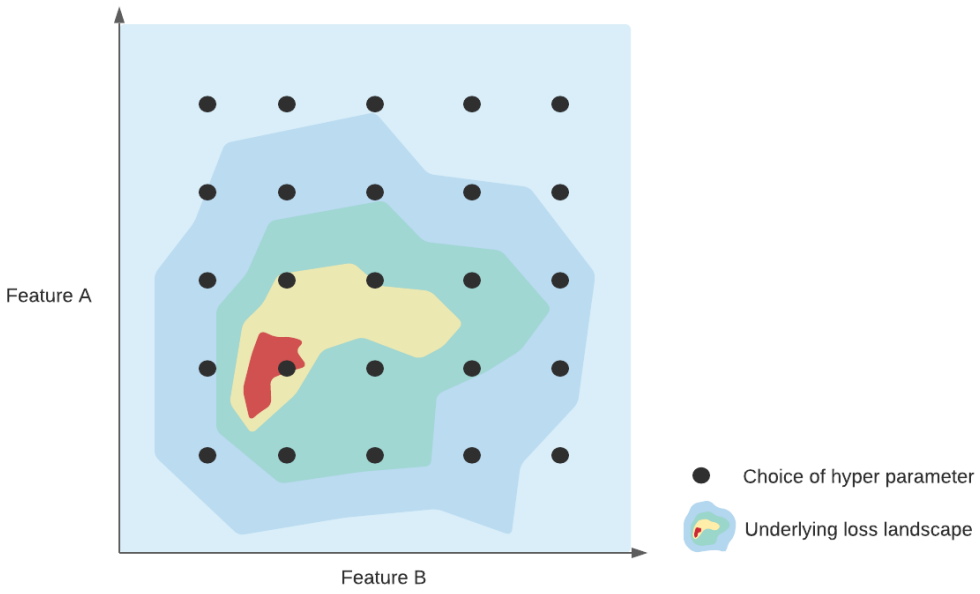

Figure 4.3: A visual illustration of a grid search of selected values of feature A and feature B with their underlying loss landscape.

Grid search is a slow optimization method compared to others, as it tries out all possible combinations. But on the other hand it is possible to control the exact hyperparameters that it should use and it is simple to implement. While hyperparameter optimization tools like *Optuna* [Aki+19] are

available, grid search serves the purpose to obtain benchmark results to compare results of the different classification models.

Grid search not only helps to find the close to optimal hyperparameters, but it also makes it possible to include the search of them in a continuous learning setup, where the pipeline should be retrained regularly.

The grid search should be implemented in such a way, that it will by itself discover all possible combinations of hyperparameters, given a dictionary where the name of the hyperparameters are the keys, and the values are lists containing the values the particular hyperparameter can be. An example of such a dictionary can be seen in listing 4.6. Here the characters before the first double underscores refer to the name of the pipeline step, and the rest of the characters refer to the name of attribute it should set for the given step.

```
params = {
    'resampler__length' : [144, 100, 50],
    'scaler__method' : ['standard', 'mean'],
    'IT__batch_size' : [128, 64],
    'IT___PipelineDLModel__epochs' : [500],
    'IT__patience' : [50]
}
```

Listing 4.6: An example of a dictionary used by the gridsearch class.

From the dictionary seen in listing 4.6, the *GridSearch* class should try out all 12 combinations of the hyperparameters. It does that by taking the product of the parameters and provide them as a list of dictionaries, each containing the hyperparameter names and their specific value for the given setting. This list of hyperparameter settings is referred to as *parameter_settings*. The *GridSearch* class should also contain a *fit* method. The skeleton for this method can be seen in listing 4.7:

```
def fit(self, X, y, validation_set):

    for i, parameters in enumerate(self.parameter_settings):

        pipeline = copy.deepcopy(self.pipeline_template)

        # Set parameters of pipeline
        for key, value in parameters.items():
            pipeline.set_parameter(key, value)

        # Fit pipeline
        pipeline.fit(X, y, validation_set)
        pipeline.save()

        # Save pipeline as best, if better than best
        if self.best_pipeline:
            if pipeline.steps[-1].best_score > self.best_pipeline.steps[-1].best_score:
                self.best_pipeline = pipeline
        else:
            self.best_pipeline = pipeline
```

Listing 4.7: An example of a dictionary used by the gridsearch class.

So basically what the *fit* method of the *GridSearch* class does, is that it loops through all the parameter settings, create a copy of the pipeline that the class was created with, sets the hyperparameters of the model, and then fits the pipeline for the input data. As it can be seen on line 16-20 in listing 4.7, it will also keep track of the best pipeline.

4.3 InceptionTime

One of the classification models that will be used is the InceptionTime algorithm. The algorithm is implemented using the PyTorch library [Pas+19]. As described in section 3.3.1 the InceptionTime

network is build-up of inception blocks which themselves are build-up of inception modules. The authors of the InceptionTime algorithm suggest using three modules per block[Ism+19]. The *init* method of the InceptionTime block can be seen in listing 4.8. Here it can be seen how the different layers in a module are defined. Here a layer is defined for each of the boxes in figure 3.13, which can be found on line 7, 8, 10, 11, 12, and 14, except for the *concatenate* box. It can be seen that each box is created three times in a list, as each block consists of three modules. This is because the same type of layers should be used in each module.

```
1  def __init__(self, in_channels, out_channels=32, dropout=0.0):
2      super(InceptionBlock, self).__init__()
3      self.activation = [nn.ReLU() for _ in range(3)]
4      self.dropout = [nn.Dropout(dropout) for _ in range(3)]
5      self.batchnorm = [nn.BatchNorm1d(out_channels*4) for _ in range(3)]
6
7      self.bottleneck = [nn.Conv1d(in_channels=in_channels, out_channels=out_channels,
           kernel_size=1, stride=1, padding='same') for _ in range(3)] # First bottleneck
8      self.max_pool_layer = [nn.MaxPool1d(kernel_size=3, stride=1, padding=1) for _ in range(3)
           ] # Maxpool
9
10     self.layer_10 = [nn.Conv1d(in_channels=out_channels, out_channels=out_channels,
           kernel_size=10, stride=1, padding='same', padding_mode='replicate') for _ in range(3)
           ] #10
11     self.layer_20 = [nn.Conv1d(in_channels=out_channels, out_channels=out_channels,
           kernel_size=20, stride=1, padding='same', padding_mode='replicate') for _ in range(3)
           ] #20
12     self.layer_40 = [nn.Conv1d(in_channels=out_channels, out_channels=out_channels,
           kernel_size=40, stride=1, padding='same', padding_mode='replicate') for _ in range(3)
           ] #40
13
14     self.bottleneck_after_max_pool = [nn.Conv1d(in_channels=in_channels, out_channels=
           out_channels, kernel_size=1, stride=1, padding='same') for _ in range(3)] # Second
           bottleneck
```

Listing 4.8: The init method in the InceptionTime block.

The actual definition of how the data should flow through the block is defined in the *forward* method of the class. This method can be seen in listing 4.9 where the input x flows according to the arrows from figure 3.13. Here x will flow through the bottleneck layer and the maxpool layer, as seen on line 3 and 4. The output of the maxpool layer will then flow through the other bottleneck layer on line 10. The output of the first bottleneck layer will then flow through the three convolutional layers in parallel, as seen on line 6-8. Finally, the output of the three convolutional layers and the second bottleneck layer will be concatenated followed by a batchnorm layer and a dropout layer as seen on line 13-15. This output will then be fed through the exact same type of layers two more times and then return. It should be noted that each of the layers are acceded with the index, i, to get the layer for the correct module.

```
1  def forward(self, x):
2      for i in range(3):
3          x_bottle_1 = self.activation[i](self.bottleneck[i](x))
4          x_max = self.max_pool_layer[i](x)
5
6          x_10 = self.activation[i](self.layer_10[i](x_bottle_1))
7          x_20 = self.activation[i](self.layer_20[i](x_bottle_1))
8          x_40 = self.activation[i](self.layer_40[i](x_bottle_1))
9
10         x_bottle_2 = self.activation[i](self.bottleneck_after_max_pool[i](x_max))
11
12         # Concatenate
13         x_return = torch.cat((x_10, x_20, x_40, x_bottle_2), 1)
14         x_return = self.batchnorm[i](x_return)
15         x = self.dropout[i](x_return)
16
17     return x
```

Listing 4.9: The forward method in the InceptionTime block.

The InceptionTime blocks then have to be put sequentially with residual connections between each block, as illustrated in figure 3.14. This implementation can be seen in listing 4.10. The residual connections created on line 12 and 15 simply contains a convolutional layer and a batchnorm layer. The for loop on line 10 adds as many residual connections and blocks as needed and append them to a module list. In all but the first module of the first block, the number of *in_channels* is the result of the four different paths in the preceding module. On line 18 the last linear layer is defined, which outputs as many features as there are classes it should predict. The input, x, then flows through each of the blocks and each of the residual connections. The output of the block is then added to the output of the residual connection, as seen on line 23-28. On line 30 the global average pooling is performed where the mean across the time dimension is calculated. This is followed by the linear layer on line 31 that reduces the number of features down to as many classes there exists. This is then followed by a softmax on line 33 to provide the predicted probability of that instance belonging to the different classes. It can furthermore be seen that it is possible to specify the depth of the InceptionTime as a hyperparameter by specifying the number of blocks it should use.

```
1  class InceptionTime(nn.Module):
2
3      def __init__(self, in_channels, num_pred_classes, out_channels=32, num_blocks=2, dropout
           =0.0):
4          super(InceptionTime, self).__init__()
5          self.activation = nn.ReLU()
6          self.num_pred_classes = num_pred_classes
7          self.num_blocks = num_blocks
8
9          self.layers = nn.ModuleList()
10         for block in range(self.num_blocks):
11             if block == 0:
12                 self.layers.append(residual(in_channels, out_channels*4, self.activation))
13                 self.layers.append(InceptionBlock(in_channels, out_channels, dropout=dropout)
                       )
14             else:
15                 self.layers.append(residual(out_channels*4, out_channels*4, self.activation))
16                 self.layers.append(InceptionBlock(out_channels*4, out_channels, dropout=
                       dropout))
17
18         self.linear = nn.Linear(in_features=out_channels*4, out_features=self.
               num_pred_classes)
19
20
21     def forward(self, x):
22         index_control = 0
23         for blocks in range(self.num_blocks):
24             x_res = self.layers[index_control](x)
25             x_block = self.layers[index_control+1](x)
26             x = x_res + x_block
27
28             index_control += 2
29
30         x = x.mean(2)
31         x = self.linear(x)
32
33         x = torch.softmax(x, dim=1)
34
35         return x
```

Listing 4.10: The full InceptionTime network that utilizes the InceptionBlock class.

4.4 ROCKET and MiniROCKET

ROCKET and MiniROCKET are both implemented as a *PipelineTransformer* class. This can be seen in listing 4.11, where the implementation of ROCKET extends the *PipelineTransformer* class. It by default gets initialized with 10.000 kernels, but this can be tuned as a hyper parameter.

```
class RocketTransformer(PipelineTransformer):

    def __init__(self, num_kernels=10000):
        super().__init__(self.__class__.__name__)
        self.rocket = Rocket
        self.num_kernels = num_kernels

    def fit(self, X, y):
        self.rocket = self.rocket(self.num_kernels)
        self.rocket.fit(X)

    def transform(self, X, y=None):
        if y is not None:
            return self.rocket.transform(X), y.argmax(axis=1)
        else:
            return self.rocket.transform(X), None
```

Listing 4.11: The implementation of the ROCKET transformer.

The ROCKET transformation is implemented using the Rocket class from sktime [Lön+21]. It can furthermore be seen that the *fit* method just calls the fit method of the Rocket object. Likewise in the *transform* method, it just calls the transform method of the Rocket object. The *transform* method however also transforms the labels from a one-hot encoded vector to a vector of class labels, if the labels are provided.

The implementation of the MiniROCKET looks very similar as it is also implemented using sktime [Lön+21]. The only difference is that it is not a possibility to change the number of kernels as they are already specified, as described in section 3.3.2.

4.5 uTSGAN

In the following section the implementation of the uTSGAN architecture is explained. In section 5.2 the experimental setup is explained to test the proposed GAN architectures. The following implementation is of the best performing GAN. The variation of the GANs will be explain in section 5.2.

As described in section 3.4.3, the uTSGAN consists of two Wasserstein GANs, *WGAN X* and *WGAN Y*, each consisting of a generator and discriminator, referred to as *generatorX*, *generatorY*, *discriminatorX*, and *discriminatorY*.

GeneratorX is a convolutional network to generate synthetic spectrograms. It consists of transposed convolutional blocks to go from a Zx1x1 shape to a Cx128x128 shape, where Z is the latent size and C is the number of time series it should produce a spectrogram for. The implementation of the init method of generatorX can be seen in listing 4.12:

```
def __init__(self, latent_size, channels):
    super(GeneratorX, self).__init__()

    # Input shape (B, L, 1, 1)
    self.net = nn.Sequential(
        self._block(latent_size, channels*128, kernel_size=4, stride=1, padding=0),  # 4x4
        self._block(channels*128, channels*64, kernel_size=4, stride=2, padding=1),  # 8x8
        self._block(channels*64, channels*32, kernel_size=4, stride=2, padding=1),  # 16x16
        self._block(channels*32, channels*16, kernel_size=4, stride=2, padding=1),  # 32x32
        self._block(channels*16, channels*8, kernel_size=4, stride=2, padding=1),  # 64x64
        nn.ConvTranspose2d(channels*8, channels, kernel_size=4, stride=2, padding=1), # 128
            x128
        nn.Tanh(),
```

```
13      )
```

Listing 4.12: The implementation of the init method for generatorX.

Each *self._block* in listing 4.12 is a transposed convolution followed by batchnorm and a leaky relu activation. It can be seen that the first block creates a 4x4 spectrogram, and from there doubles in size for each block until it is of size 64x64. The last transposed convolution seen on line 11, is the same transposed convolution that is in each block but without the batchnorm and leaky relu. On line 12 it can be seen that the tanh activation is used to get all values between -1 and 1. It is therefore important in the training of the uTSGAN to normalize the spectrogram values to be between -1 and 1. The forward method of generatorX just passes the input through the *self.net* defined in the init method. The full implementation of generatorX can be found in appendix A.5.

DiscriminatorX, which should learn to differentiate between true spectrograms and the synthetic spectrograms generated by generatorX looks similar to generatorX. It is basically just a mirror of generatorX, so it uses normal convolutional layers to get the size of the input spectrogram down to a size of 1x1. The init method of discriminatorX can be seen in listing 4.13:

```
1  def __init__(self, channels):
2      super(DiscriminatorX, self).__init__()
3
4      # Input shape (B, C, 128, 128)
5      self.net = nn.Sequential(
6          self._block(channels, channels*2, kernel_size=4, stride=2, padding=1),  # 64x64
7          self._block(channels*2,channels*4, kernel_size=4, stride=2, padding=1),  # 32x32
8          self._block(channels*4,channels*8, kernel_size=4, stride=2, padding=1),  # 16x16
9          self._block(channels*8,channels*16, kernel_size=4, stride=2, padding=1),  # 8x8
10         self._block(channels*16,channels*32, kernel_size=4, stride=2, padding=1),  # 4x4
11         nn.Conv2d(channels*32, 1, kernel_size=4, stride=2, padding=0), # 1x1
12     )
```

Listing 4.13: The implementation of the init method for discriminatorX.

From listing 4.13, the discriminatorX looks just like generatorX. The only difference was that the blocks use normal convolutions instead of transposed convolutions, and in the end, it uses just a convolution without any activation function. This makes it able to output a value that is not restricted and is a direct indication of how sure the discriminator is that the input is a true instance. It can also be seen that the discriminator expects the input spectrograms to be of shape 128x128 to comply with the synthetic generated spectrograms. The forward method of discriminatorX is again passing the input through the *self.net* defined in the init method, but before it does, it adds some Gaussian noise to the input to make it more robust towards variance. The full implementation of discriminatorX can be found in appendix A.6.

While the generated synthetic spectrograms from generatorX are used as input to discriminatorX, they are also used as input to generatorY. Here the spectrograms are converted into time series. As the uTSGAN paper[SS21] did not specify how this should be implemented, it was implemented in the attempt to mimic a learnable inverse discrete Fourier transform (IDFT) for each point in time. This means implementing a convolution using a stride of 2x1 that ”compresses” the spectrogram along the frequency dimension as illustrated in figure 4.4. This method is what makes the proposed method unique as it turned out that the original uTSGAN implemented this conversion from a spectrogram to a time series in a different manner which will be explained in section 5.2.

It is important to note here that the spectrograms generated in generatorX are not conditioned and therefore the inverse STFT cannot be used to go from the synthetic spectrograms to synthetic time series.

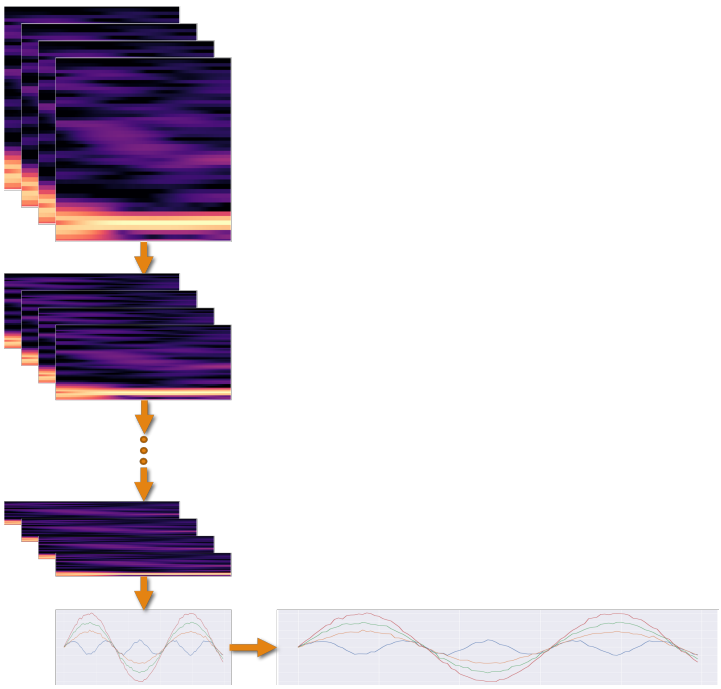

Figure 4.4: A visual representation of how generatorY shrinks the spectrograms in the attempt to mimic a learnable inverse DFT, and then upsample the timeseries to the correct length. The orange arrows represent convolutional layers.

Figure 4.4 shows that it uses convolutional layers to half the frequency dimension till the spectrograms are shrunk down to a size of 1x128. In each of the convolutions, the amount of channels increases by a factor of two to compensate for the information lost in the frequency dimension. This process is essentially trying to estimate the expression $e^{i\frac{2\pi}{N}kn}$ from equation 4.1 for each point in time. In the equation, x_n is the value of the time series at time n and N is the number of different frequency bins. The reason why it is not possible to directly use the ISTFT is the fact that the spectrograms are not conditioned and the spectrograms do therefore not directly represent a conditioned time series.

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k * e^{i\frac{2\pi}{N}kn} \tag{4.1}$$

The implementation of the different parts in generatorY can be seen in listing 4.14. Here it can be seen that the init method defines three different parts. On line 5 it creates an embedding layer, which is the layer that should make the model take the condition into consideration. The next part that is initialized is the network that consists of the convolution blocks that uses asymmetric stride to get the spectrograms down to a size of 1x128. It can be seen on line 15 that the dimensionality is reduced to a 1-dimensional time series. The last part that is initialized is the network that upsamples the time series to a length of 512. It can be seen on line 19 and 20, that it applies a convolutional layer followed by the tanh activation before upsampling.

```
1 def __init__(self, embed_size, channels, num_labels):
2     super(GeneratorY, self).__init__()
3
4     self.embed_size = embed_size
5     self.embedding = nn.Embedding(num_labels, embed_size*128*128)
6
7     # Input shape: (B, channels+embed, 128, 128)
8     self.net = nn.Sequential(
```

```
9           self._block(channels+embed_size, channels*2, kernel_size=3, stride=(2,1), padding=1),
                # 64x128
10          self._block(channels*2,channels*4, kernel_size=3, stride=(2,1), padding=1),  # 32x128
11          self._block(channels*4,channels*8, kernel_size=3, stride=(2,1), padding=1), # 16x128
12          self._block(channels*8,channels*16, kernel_size=3, stride=(2,1), padding=1), # 8x128
13          self._block(channels*16,channels*32, kernel_size=3, stride=(2,1), padding=1), # 4x128
14          self._block(channels*32, channels*16, kernel_size=3, stride=(2,1), padding=(0,1)), #
                1x128
15          nn.Flatten(start_dim=2),
16      )
17
18      self.upnet = nn.Sequential(
19          nn.Conv1d(channels*16, channels, kernel_size=3, stride=1, padding=1, bias=False),
20          nn.Tanh(),
21          nn.Upsample(size=512, mode='linear', align_corners=False)
22      )
```

Listing 4.14: The implementation of the init method for generatorY.

The forward method of generatorY is a bit different from the others as it takes in the labels to learn the condition. It can be seen from listing 4.15 that it starts by passing the labels through the embedding layer where it is reshaped to the size of Ex128x128 and gets concatenated with the synthetic spectrograms. Here E refers to the embed size. On line 4 and 5, it then passes that input through the rest of the network. The full implementation of generatorY can be found in appendix A.7

```
1 def forward(self, X, y):
2     embedding = self.embedding(y.long()).view(y.shape[0], self.embed_size, 128, 128)
3     X = torch.cat([X, embedding], dim=1)
4     X = self.net(X)
5     X = self.upnet(X)
6     return X
```

Listing 4.15: The implementation of the forward method for generatorY.

The synthetic generated time series by generatorY is then processed by discriminatorY. The architecture is almost identical to discriminatorX expect it uses 1-dimensional convolutions instead of 2-dimensional. DiscriminatorY also needs to be conditioned. This is again done by using an embedding layer. The implementation of discriminatorY can be found in appendix A.8.

To train the uTSGAN, a quite comprehensive training loop had to be implemented, due to the four different networks that had to be trained. The uTSGAN is implemented as a *PipelineTransformer*, where the whole training process is located in the fit method. The skeleton of the implementation of the training loop can be seen in listing 4.16. Here it goes into the training loop that will run for the specified amount of epochs, as seen on line 4. The for loop on line 5 then iterates through each batch, where the inputs, labels, and spectrograms of the input get extracted from the batch. On line 8 to 43 the loop that trains the discriminator is located. Here it starts by creating the latent vector, z, which will be used to create fake spectrograms. On line 13 it then passes the true spectrograms through discriminatorX. It then generates some synthetic spectrograms based on the latent vector, z, with generatorX on line 15, which it also passes through discriminatorX. On line 18 to 21 it does the same, just for generatorY and discriminatorY, where it uses the synthetic generated spectrograms from generatorX to pass through generatorY instead of a latent vector. On line 24 to 28 it then calculates the loss for each discriminator according to equation 3.26 and 3.27. The optimizer will try to minimize this loss, and therefore a minus sign is put in front of the loss to ultimately maximizing the distance between the two distributions. On line 33 the two losses get unified as in equation 3.28. Now the gradients of the discriminators are zeroed followed by a backpropagation on line 37. Here it can be seen that the gradient graph is retained as it should be reused for training the generators later. On line 40 a step is taken on each of the optimizers for the discriminators, to make the discriminators update their weights. Note that the discriminator is trained for more iterations than the discriminators. This comes from the WGAN-GP paper [Gul+17]. On line 43 and 44, the synthetic generated spectrograms get passed through the updated discriminatorX and the loss gets calculated as the negative mean of

the discriminator output, as the generators should learn to maximize the loss of the discriminators. On line 47 and 49, the same thing happens, just for generatorY and discriminatorY. It will then zero the gradients of the generators and then call backwards on both generators, as seen on line 54 and 55. Here the graph is retained after calling backward on the loss from generatorX, as parts of the graph is reused in the backward call on generatorY loss. Lastly, a step is taken for each of the generator optimizers to update their weights. The full implementation of the fit method can be found in appendix A.9.

```
1  def fit(self, X, y):
2
3      # Run training loop
4      for epoch in range(self.__epochs):
5          for batch in dataloader:
6              inputs, labels, spectrograms = batch
7
8              for _ in range(self.discriminator_iterations):
9                  # Create latent vector
10                 z = torch.randn(labels.shape[0], self.__generatorX.latent_size, 1, 1)
11
12                 # Feed discriminatorX with true spectrograms
13                 output_discriminatorX_true = self.__discriminatorX(spectrograms).view(-1)
14                 # Feed discriminatorX with fake spectrograms
15                 g_X_z = self.__generatorX(z)
16                 output_discriminatorX_fake = self.__discriminatorX(g_X_z).view(-1)
17                 # Train discriminatorY with true time series
18                 output_discriminatorY_true = self.__discriminatorY(inputs, labels).view(-1)
19                 # Train discriminatorY with fake time series
20                 g_Y_z = self.__generatorY(g_X_z, labels)
21                 output_discriminatorY_fake = self.__discriminatorY(g_Y_z, labels).view(-1)
22
23                 # Calculate loss for discriminatorX
24                 GP_X = self.gradient_penalty_spectrogram(self.__discriminatorX, spectrograms,
                       g_X_z)
25                 loss_discriminatorX = -(torch.mean(output_discriminatorX_true) - torch.mean(
                       output_discriminatorX_fake)) + self.lambda_penalty_x * GP_X
26                 # Calculate loss for discriminatorY
27                 GP_Y = self.gradient_penalty_timeseries(self.__discriminatorY, inputs, g_Y_z,
                       labels)
28                 loss_discriminatorY = -(torch.mean(output_discriminatorY_true) - torch.mean(
                       output_discriminatorY_fake)) + self.lambda_penalty_y * GP_Y
29
30                 # Calculate the unified loss for discriminators
31                 unified_loss_discriminator = (loss_discriminatorX + loss_discriminatorY) / 2
32
33                 # Zero the gradients for the discriminators
34                 ...
35
36                 # Backwards on discriminator
37                 unified_loss_discriminator.backward(retain_graph=True)
38
39                 # Optimize Discriminator
40                 ...
41
42             # Calculate loss generatorX
43             genX_ = self.__discriminatorX(g_X_z)
44             loss_generatorX = -torch.mean(genX_)
45
46             # Calculate loss generatorY
47             genY_ = self.__discriminatorY(g_Y_z, labels)
48             loss_generatorY = -torch.mean(genY_)
49
50             # Zero the gradients for the generator
51             ...
52
53             # Backwards on both generators
54             loss_generatorX.backward(retain_graph=True)
```

```
            loss_generatorY.backward()

            # Optimize generator
            ...
```

Listing 4.16: The implementation of the fit method the uTSGAN.

The design and implementation of the infrastructure for fitting models and predicting failures have now been implemented. The implemented pipeline class with associated methods and models allows for easy tuning of hyperparameters in both the coming experiments but also for future operational settings.

CHAPTER 5
Experimental Method

The following chapter will describe the experiments performed to answer the stated research questions. The experiments are divided into three, a GAN experiment, a simulated experiment, and a real-world experiment. But first, the datasets used in the experiments are presented along with preliminary data exploration.

A series of generative adversarial models (GANs) will be presented with varying complexity. They will be investigated and their ability to create realistic time series data will be compared. The general learnings and limitations of the GANs will also be presented. Next, an extensive experiment of different time series classification methods is conducted on three open-source datasets. One of the datasets will be reduced and the most promising classification models will be trained on the reduced dataset to simulate a small data foundation. A GAN will be used to generate synthetic time series data and thereby increase the size of the dataset again. The most promising classification models will be trained on a combination of the true and generated data to see if the addition of synthetic data positively benefits the classification models.

The "real-world" experiment is based on a relevant dataset from real wind turbines. Here the most promising classification models from the simulated experiments are used to obtain a result. The purpose of the experiments is to investigate to what extent it is possible to differentiate the failures occurring in a wind turbine based on its sensor values. Furthermore, to investigate if it is possible to increase the performance of a model by doing complex data augmentation using a GAN.

5.1 Data foundation

In the following sections five different datasets will be presented. Three of them are open-source datasets and one of them is a real-world dataset. The last dataset is a constructed one, where the underlying functions of the time series are known.

5.1.1 Sinusoidal dataset

The constructed dataset serves the main purpose of being a simple dataset to validate if a GAN was able to learn the underlying function of a dataset while being able to differentiate between different conditions. The dataset that is constructed consists of four different sine waves with different amplitudes, where one of them has a smaller frequency than the other, depending on the condition. An example of each condition can be seen in figure 5.1:

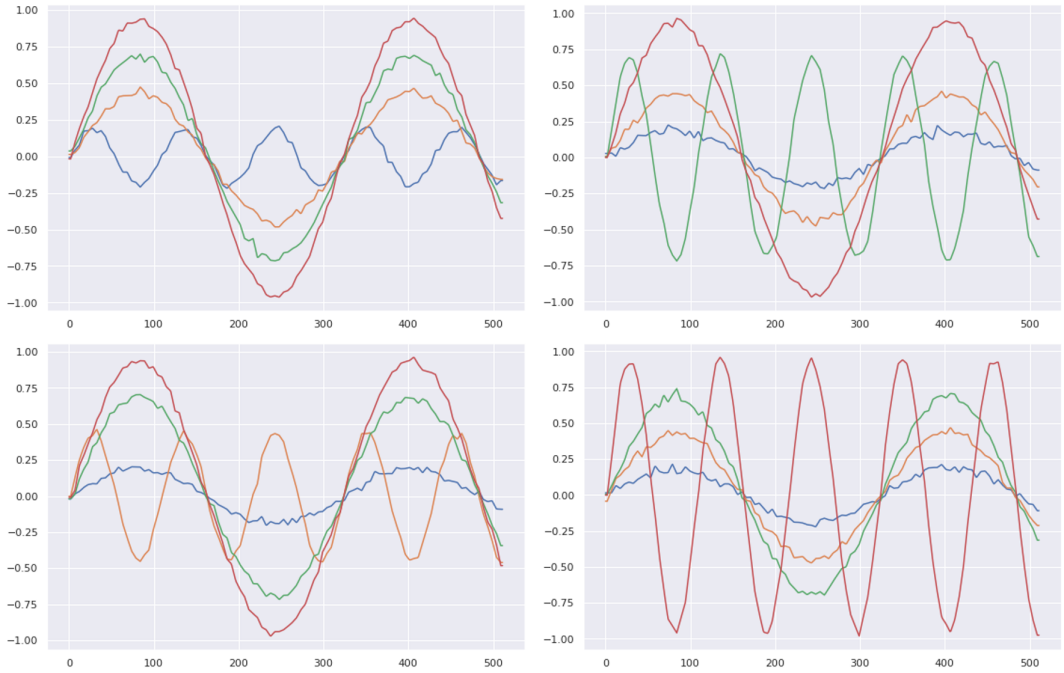

Figure 5.1: An example of each condition in the fictive sinusoidal dataset.

As seen from figure 5.1 there has also been added some random noise to the sine waves so the different samples vary slightly.

5.1.2 Articulary Word Recognition - AWR

One of the open-source datasets is the *Articulary Word Recognition* dataset, also referred to as AWR [J W21]. The AWR dataset is about classifying 25 differently pronounced words, where the data is recorded from 9 sensors attached to a subjects face. Due to its large amount of classes and number of sensors it is interesting to see how both the GAN and the classification models will perform on the dataset. As seen from figure 5.2 the dataset is also perfectly balanced with 11 instances of each class. With its benchmark accuracy of 99.56% [Rui+20] it is expected that the classification models in this project will achieve good results.

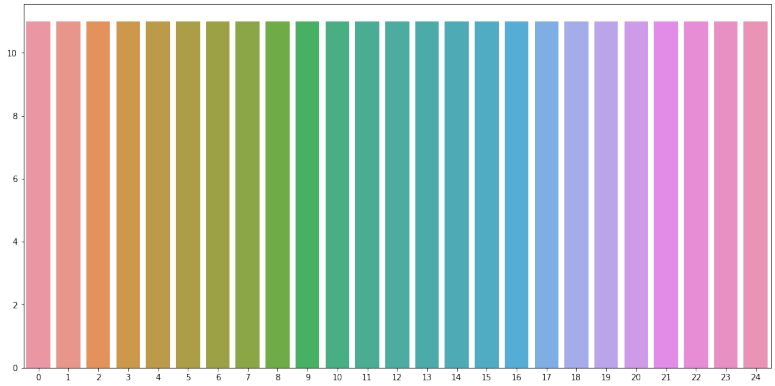

Figure 5.2: The class distribution of the AWR dataset.

The spectrograms of one example for the three first classes in the AWR dataset can be seen in figure 5.3. The spectrograms have been generated with the *Hann* window, a window size of $\frac{1}{5}$ of the time series length (144), and a hop size of one, as this seemed to provide the best resolution trade-off. These parameters will be used when creating spectrograms for the uTSGAN model.

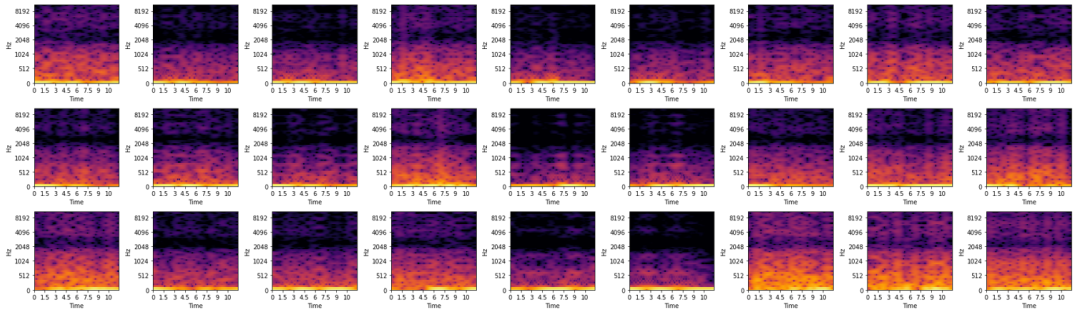

Figure 5.3: The spectrograms for each of the sensors(columns) for the first three classes(rows) in the AWR dataset. The spectrograms for all the classes can be found in appendix B.1 and B.2.

To see the distribution of the different sensors for the different classes the plot in figure 5.4 has been created. Here all the normalized instances for each of the sensors have been plotted for the first three classes.

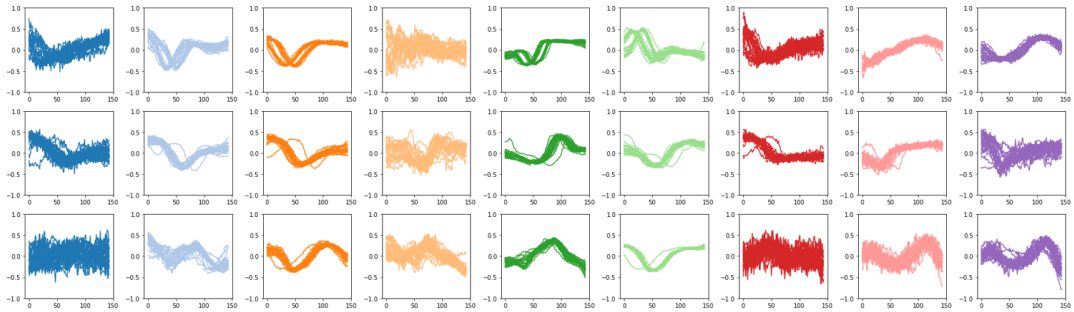

Figure 5.4: All the time series data plotted for each of the sensors (columns) for the first three classes (rows) in the AWR dataset. The plot for all the classes can be found in appendix B.3 and B.4.

From figure 5.4 it can be seen that the instances in each of the plots mostly look quite similar. It, therefore, seems like the classes are easily distinguishable from each other which also explains the high benchmark accuracy.

5.1.3 LSST

The next open-source dataset that is used is the LSST dataset[Kag21]. The dataset contains 6 different sensors, each a different astronomical filter. The sensors record the brightness over time for different regions of the light spectrum. The goal is then to classify the astronomical object among 14 different classes. The benchmark for this dataset is an accuracy of 63.62%, which makes it a harder dataset compared to the AWR dataset. The noise, imbalanced data from the real-world sensors makes this data set interesting to investigate as many of the same characteristics are expected with data coming from industrial assets. The distribution of the dataset can be seen in figure 5.5:

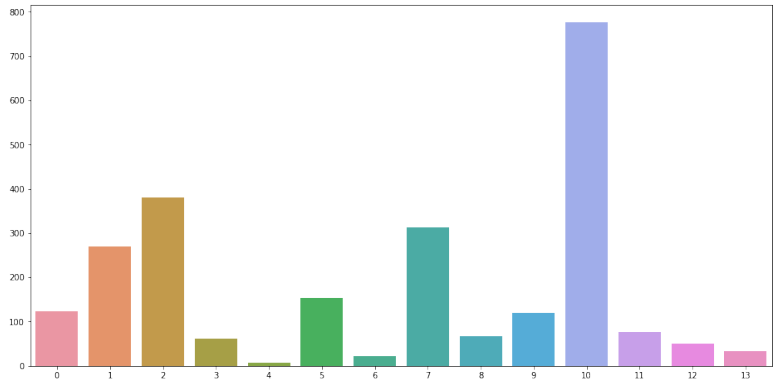

Figure 5.5: The class distribution of the LSST dataset.

The normalized time series data from the LSST dataset can be seen in figure 5.6 where the first three classes are shown. It can here be seen that the time series across the different sensors are much harder to distinguish from each other compared to the AWR dataset, which also explains the lower benchmark. It can be seen that across all the instances that the majority of the time series are around -0.4 with only a few deviations. And most of them seem to rarely change over time, which makes their spectrograms relatively information lose. The spectrograms can be found in appendix B.5.

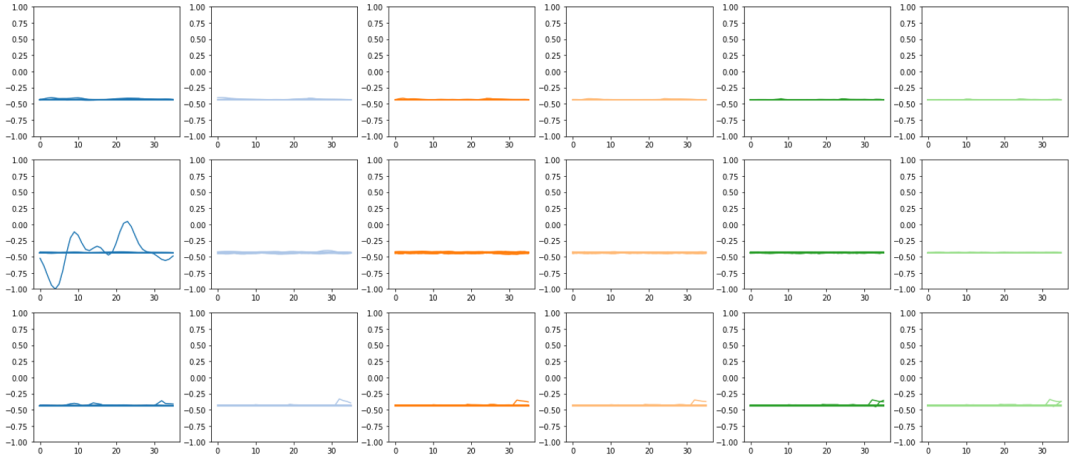

Figure 5.6: All the time series data plotted for each of the sensors(columns) for the first three classes(rows) in the LSST dataset. The plot for all the classes can be found in appendix B.6.

5.1.4 Hydraulic Condition Monitoring

The last open-source dataset that has been used in the following experiments, is the hydraulic condition monitoring dataset [gGm21]. It consists of time series from 16 different sensors that samples at different rates. As this dataset is about hydraulic monitoring, it also seems to be similar to the real-world data that could be recorded on assets. The distribution of the classes can be seen in figure 5.7:

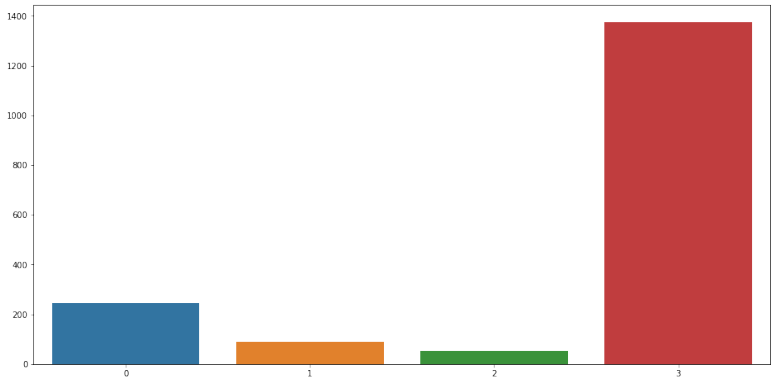

Figure 5.7: The class distribution of the hydraulic condition monitoring dataset.

The time series data for the last three sensors for each of the classes can be seen in figure 5.8

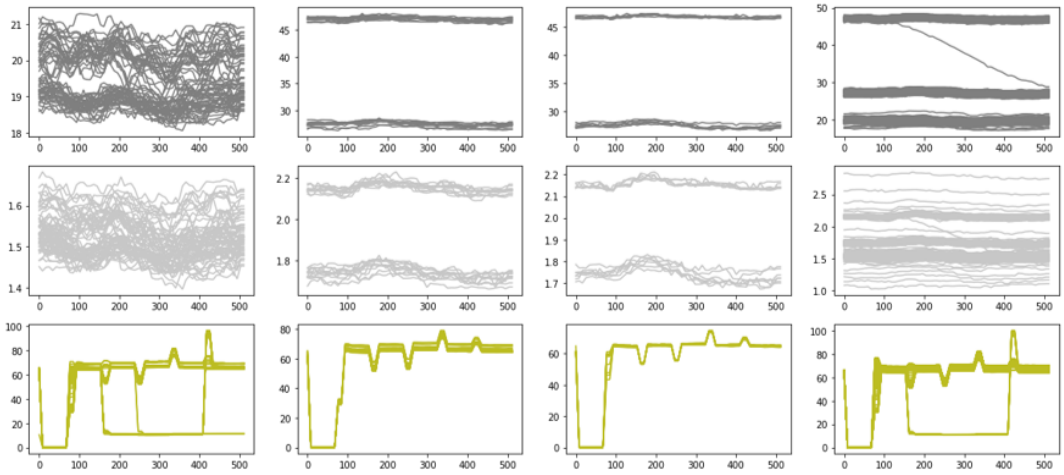

Figure 5.8: All the time series data plotted for each of the classes(columns) for the last three sensors(rows) in the hydraulic condition monitoring dataset. The plot for all the sensors can be found in appendix B.7 and B.8.

5.1.5 Real-World Wind Turbine

The last dataset used in this project is the wind turbine dataset, which was obtained from the company *EDP* in relation to a data analysis competition. The purpose of the competition is to detect the fault that occurs on wind turbines when in operation. The data is from 5 wind turbines over a 2-year period (2016-2017). The data set is divided into 3 different parts:

- The sensor data for all the wind turbines.
- Meteorological data describing the weather conditions.
- Failure records with the turbine's failure, the component in question, and a short remark.

The dataset contains *Supervisory Control And Data Acquisition* (SCADA) data for 5 turbines in Switzerland. 83 features are collected, where a data point is sampled every tenth minute. The sensors measure a large amount of data within the 10 minutes but are then aggregated to the average, standard

deviation, min, and max over the ten minutes period. A clear limitation with the data set is therefore that it is impossible to say anything about the fluctuation of the sensor readings in between the 10-minute intervals. This naturally haves a ”smoothing” effect on the data that most likely influences the classifiers ability to differentiate the failures. A sample of the sensors can be found in figure 5.9 and the full explanatory list of the sensor values can be found in appendix C.

Descriptor	Type	Description	Component
Gen_RPM_Max [rpm]	FLOAT	Maximum generator rpm in latest average period	Generator
Gen_RPM_Min [rpm]	FLOAT	Minimum generator rpm in latest average period	Generator
Gen_RPM_Avg [rpm]	FLOAT	Average generator rpm	Generator
Gen_RPM_Std [rpm]	FLOAT	Std. generator rpm in latest average period	Generator
Gen_Bear_Temp_Avg [°C]	INT	Average temperature in generator bearing 1 (Non-Drive End)	Generator
Gen_Phase1_Temp_Avg [°C]	INT	Average temperature inside generator in stator windings phase 1	Generator
Gen_Phase2_Temp_Avg [°C]	INT	Average temperature inside generator in stator windings phase 2	Generator
Gen_Phase3_Temp_Avg [°C]	INT	Average temperature inside generator in stator windings phase 3	Generator
Hyd_Oil_Temp_Avg [°C]	INT	Average temperature oil in hydraulic group	Hydraulic
Gear_Oil_Temp_Avg [°C]	INT	Average temperature oil in gearbox	Gearbox

Figure 5.9: Sample of the explanation of the wind data sensor.

The failures are presented with a turbine id, timestamp, a component for where the failure occurred, and a remark. The full list of failures can be found in table 5.1. Here it can be seen that many unique failures exist meaning only one example of a failure is available. Furthermore is the number of failures relatively small meaning not many failures did occur during the two years of sampling data from the 5 wind turbines. This is backing up the observed problems of few failure states to many normal states explained in section 2.5.

Turbine_ID	Component	Timestamp	Remarks
T11	GENERATOR	2016-03-03T19:00:00+00:00	Electric circuit error in generator
T06	HYDRAULIC_GROUP	2016-04-04T18:53:00+00:00	Error in pitch regulation
T07	GENERATOR_BEARING	2016-04-30T12:40:00+00:00	High temperature in generator bearing (replace...
T09	GENERATOR_BEARING	2016-06-07T16:59:00+00:00	High temperature generator bearing
T07	TRANSFORMER	2016-07-10T03:46:00+00:00	High temperature transformer
T06	GENERATOR	2016-07-11T19:48:00+00:00	Generator replaced
T01	GEARBOX	2016-07-18T02:10:00+00:00	Gearbox pump damaged
T06	GENERATOR	2016-07-24T17:01:00+00:00	Generator temperature sensor failure
T09	GENERATOR_BEARING	2016-08-22T18:25:00+00:00	High temperature generator bearing
T07	TRANSFORMER	2016-08-23T02:21:00+00:00	High temperature transformer. Transformer refr...
T06	GENERATOR	2016-09-04T08:08:00+00:00	High temperature generator error
T06	GENERATOR	2016-10-02T17:08:00+00:00	Refrigeration system and temperature sensors i...
T09	GEARBOX	2016-10-11T08:06:00+00:00	Gearbox repaired
T09	GENERATOR_BEARING	2016-10-17T09:19:00+00:00	Generator bearings replaced
T11	HYDRAULIC_GROUP	2016-10-17T17:44:00+00:00	Hydraulic group error in the brake circuit
T06	GENERATOR	2016-10-27T16:26:00+00:00	Generator replaced
T09	GENERATOR_BEARING	2017-01-25T12:55:00+00:00	Generator bearings replaced
T11	HYDRAULIC_GROUP	2017-04-26T18:06:00+00:00	Hydraulic group error in the brake circuit
T07	HYDRAULIC_GROUP	2017-06-17T11:35:00+00:00	Oil leakage in Hub
T01	TRANSFORMER	2017-08-11T13:14:00+00:00	Transformer fan damaged
T06	HYDRAULIC_GROUP	2017-08-19T09:47:00+00:00	Oil leakage in Hub
T07	GENERATOR_BEARING	2017-08-20T06:08:00+00:00	Generator bearings damaged
T07	GENERATOR	2017-08-21T14:47:00+00:00	Generator damaged

Table 5.1: List of failures on the 5 turbines on a two year period.

A correlation and PCA analysis was performed to investigate the internal dependencies and to see which features had the most influence on the variance of the data. In the study by Liu et al.[Liu+18] explained in section 2.3, where a similar problem was investigated. Here values with very high correlation were eliminated. Furthermore, using the principle components analysis it was investigated if the whole dataset could potentially be transformed to a lower-dimensional space. From figure 5.10 it is clear that a lot of the values are highly correlated indicating the potential to remove features that are closely correlated with others in the dataset.

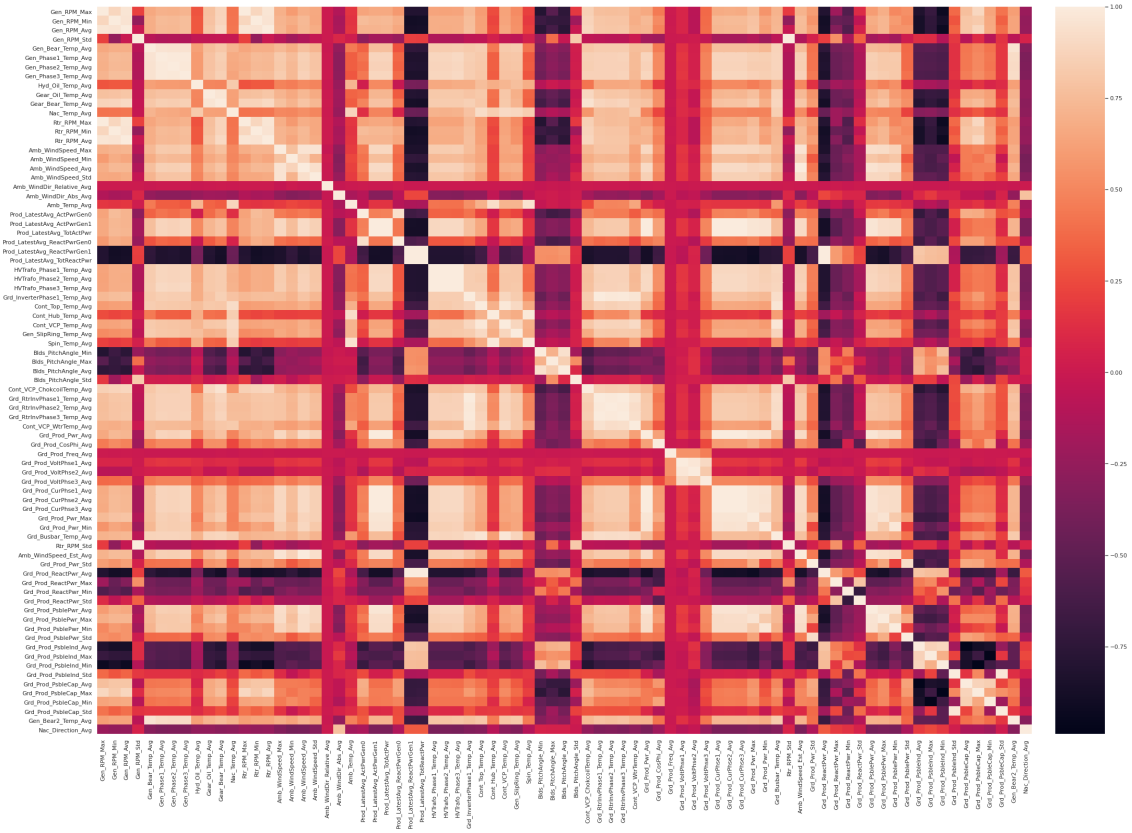

Figure 5.10: The Pearson correlation between all the SCADA sensor data from the wind turbine data set.

The PCA analysis shows that around 97% of the variance of the data set could be explained by only the 20 first principle components as seen in figure 5.11. Even though the PCA analysis suggested that a lot of the dataset's variance could be explained with a smaller set of features in the form of principal components, the data was kept in its original form. It was hypothesised that these small differences in the data would allow a classification model to differentiate between the different failures. It was however decided to only look a the *average* features from the data. This was done as an empirical test was carried out where a classifier was fitted on only the *average* features and all features. The results from these showed that no real difference in performance came from the min, max, std features.

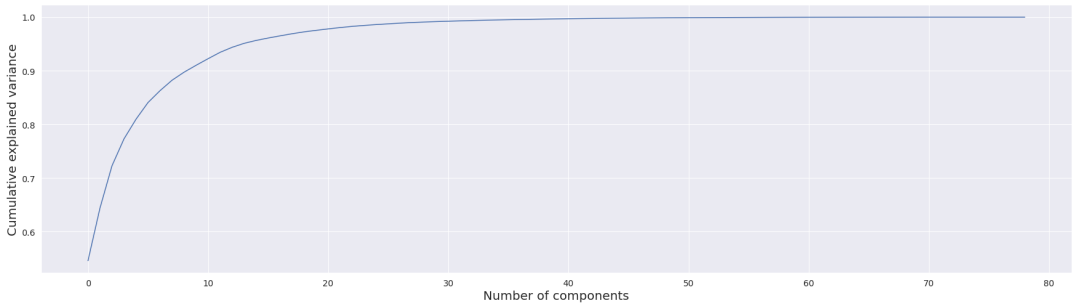

Figure 5.11: The accumulated variance for the number of principle components.

5.2 GAN Experiment

The purpose of this experiment is to investigate different GAN architectures ability to generate synthetic time series data. The generated data should be used to improve the performance of a classification model, as the synthetic data can be used to balance the dataset and in general increase the size of the dataset. In the following sections, the methods used to implement four different GAN architectures is presented. The experimental method used to investigate their ability to create realistic time series data will be presented along with general limitations.

5.2.1 Presentation of the GAN Models

In general, when training a GAN it is really hard to determine if it continues to get better. When it comes to generating synthetic images it can be determined visually by looking at the generated images to see if they form true looking objects. This is not exactly possible to do with time series as humans are not designed to interpret this kind of data. Synthetic time series can however be compared with some true examples to see if the GAN is able to capture the characteristics of the data distribution. Therefore, it is beneficial to start out with a problem where the underlying function or distribution of the time series data is known. To visually evaluate the proposed GAN models the constructed sinusoidal dataset explained in section 5.1 has been used. This dataset will be used for the first model after the AWR dataset will be used.

5.2.1.1 Conditional DCWGAN-GP

To start with, a vanilla GAN was created mainly based on the original GAN by Ian Goodfellow [Goo+14]. The true problem is known to be a multi-class problem, so the GAN has been implemented as a conditional GAN where the labels from the data sample have been given as an input to the networks as explained in section 3.4. Furthermore, convolutional layers are known to improve the GAN for image generation and as time series has temporal characteristics which are similar to the spatial characteristics in images. Therefore convolutional layers are used in both the generator and discriminator. The initial implementation will therefore be characterised as a conditional DCGAN [RMC15]. To make sure the conditional DCGAN is capable of learning a time series underlying data distribution the model is fitted on the constructed sinusoidal dataset.

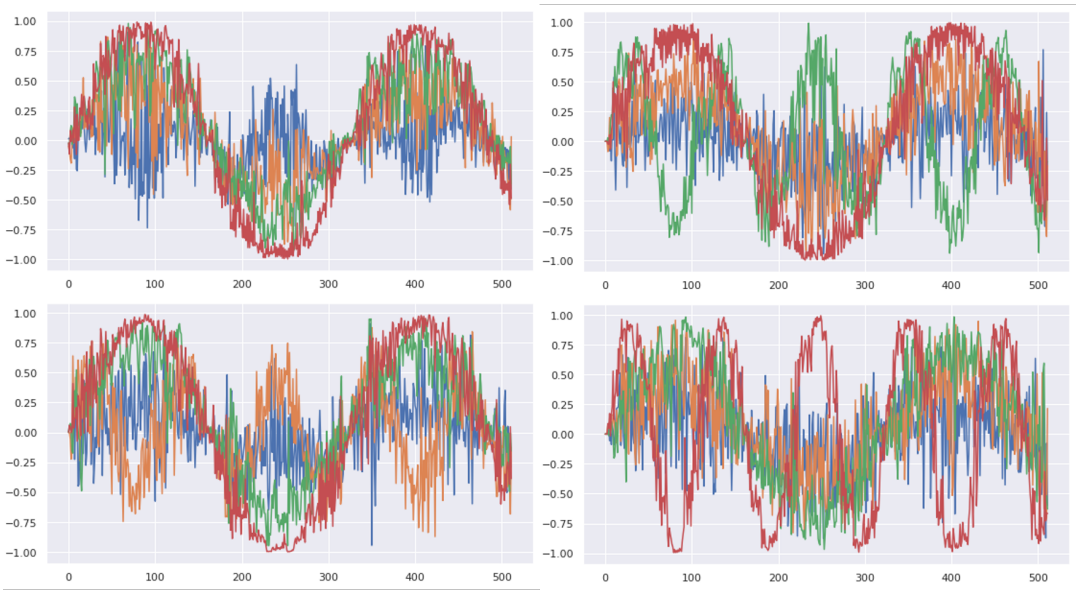

Figure 5.12: A synthetic example of each condition in the fictive sine dataset produced by the conditional DCGAN.

In figure 5.12 it can be seen that the conditional DCGAN understands the overall pattern in the data and also seems to catch the condition fairly well. But the conditional DCGAN also generates an extreme amount of noise compared to the noise in the true data. However, this is not the main concern with this model. From figure 5.13 where 25 examples have been plotted on top of each other, it can be seen that it looks like only one example has been plotted even though all of the instances have been generated using different random latent vectors. This indicates mode collapse and the generator, therefore, are able to trick the discriminator by only outputting one example per label.

Figure 5.13: The mode collapse of the conditional DCGAN. Here 25 examples are plotted for one condition.

To handle the problem of mode collapse the model was ”upgraded” to a conditional DCWGAN-GP. It was expected that this improvement would allow the model to generate synthetic data from the whole distribution due to the Wasserstein distance as explained in section 3.4.2. The new model is again fitted on the synthetic sinusoidal dataset. The generated samples of the conditional DCWGAN-GP can be seen in figure 5.14. Here it can be seen that the synthetic data generated by the conditional DCWGAN-GP seem to catch the underlying function and condition of the sine waves. But the data is also extremely noisy. The results actually look worse compared to the conditional DCGAN. This is probably due to the more complex Wasserstein loss function, that requires more training.

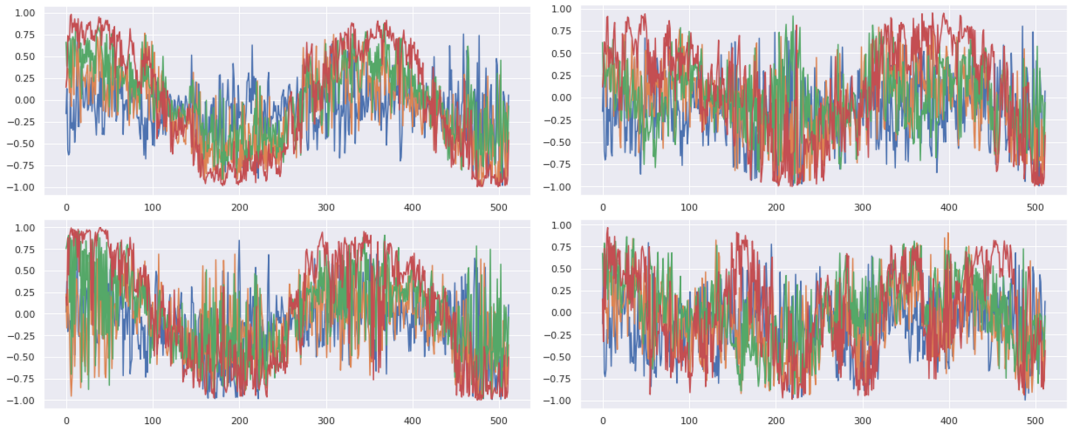

Figure 5.14: A synthetic example of each condition in the fictive sine dataset produced by the conditional DCWGAN-GP.

When inspecting for mode collapse in figure 5.15, the lines forming the noisy sine waves are thicker compared to the lines in figure 5.13, which indicates that this conditional DCWGAN-GP does not suffer as much from mode collapse. It can also be seen around timestamp 450 in the figure, that some of the red waves can be distinguished from each other which further indicate that the model is not mode collapsing.

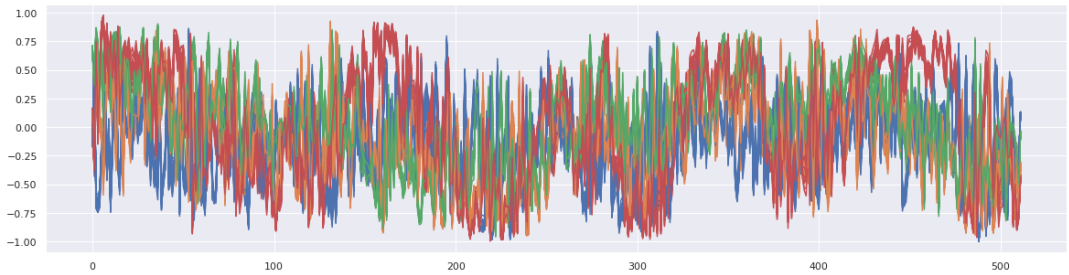

Figure 5.15: 25 synthetic examples of one condition generated by the conditional DCWGAN-GP.

Now that it is known that the Wasserstein distance helps to eliminate mode collapse but the basic conditional DCWGAN-GP does not achieve great results, the uTSGAN explained in section 3.4.3 will be explored. First, the model based on the authors' code will be presented. As the uTSGAN is expected to perform better, the experiment is continued on the AWR dataset. The performance of the conditional DCWGAN-GP can be seen in figure 5.16. Here it can be seen that it continues to produce very noisy data. It however seems that follows the correct pattern of the true data.

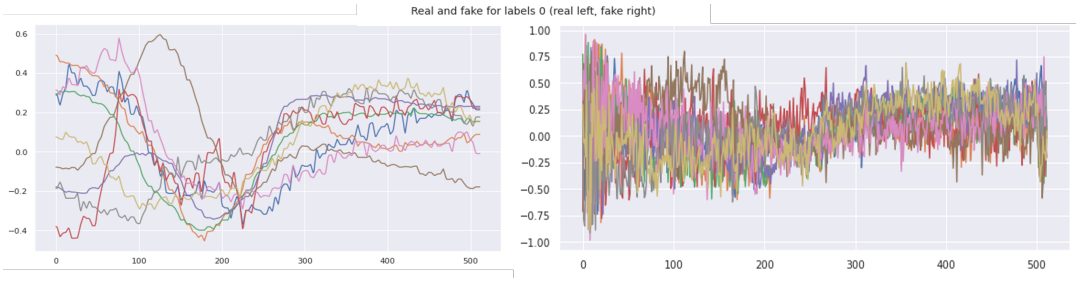

Figure 5.16: A synthetic example of a condition from the AWR dataset produced by the conditional DCWGAN-GP. True data on the left, generated on the right.

5.2.1.2 uTSGAN original

After conversations with Kaleb Earl Smith, the author of the uTSGAN papers, the code behind the papers were acquired. The code was written in TensorFlow so it had to be implemented in PyTorch. The code varied a little from the description in the paper. First of all, the use of a patch loss was added to the unified loss to help generatorX to produce better spectrograms. The true and generated spectrograms were divided into patches. The mean squared error between the patches was calculated to see how far the true spectrograms were from the generated. The average mean squared error between all the patches was added to the loss function in equation 3.28.

The main differences were how he had implemented the generatorY. Instead of using stride in the frequency domain to downscale the spectrograms to time series he used stride in both directions to end up with a very small spectrogram. He then used a linear layer before transforming it back to a time series and then upsample it by alternating between a convolution layer and an upsampling until the desired length was reached. This architecture achieved good results with this approach in the univariate case, and it also seems to work in the multivariate case. The generated samples from this model on the AWR dataset can be seen in figure 5.17.

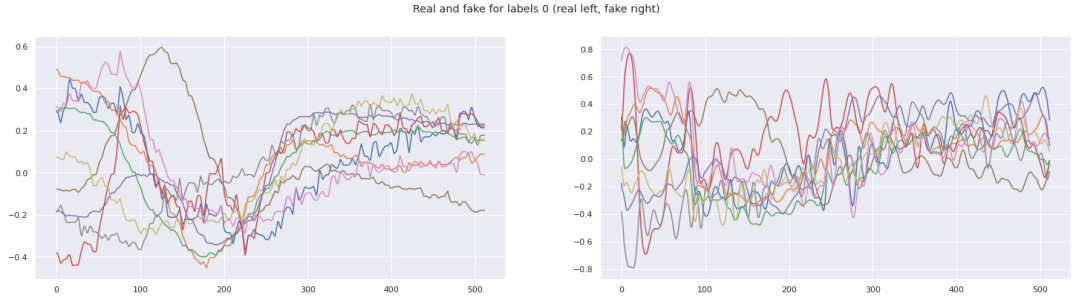

Figure 5.17: A synthetic example of a condition from the AWR dataset produced by the original uTSGAN. True data on the left, generated on the right.

From the figure it can be seen that the generated samples look promising as the general pattern of the time series had been captured.

5.2.1.3 uTSGAN First Iteration

A simpler version of the original uTSGAN architecture was then implemented. This model was exclusively based on the description in the published paper. Here the patch loss was removed so only the stated unified loss in equation 3.28 was used to train the model. In general, the number of features

extracted from each convolutional layer was reduced significantly to make the model simpler and easier to train.

Most importantly the encoding in generatorY was performed only in the frequency dimension as described in section 4.5. The time series was then decoded using a single convolution and upsampled to the required length.

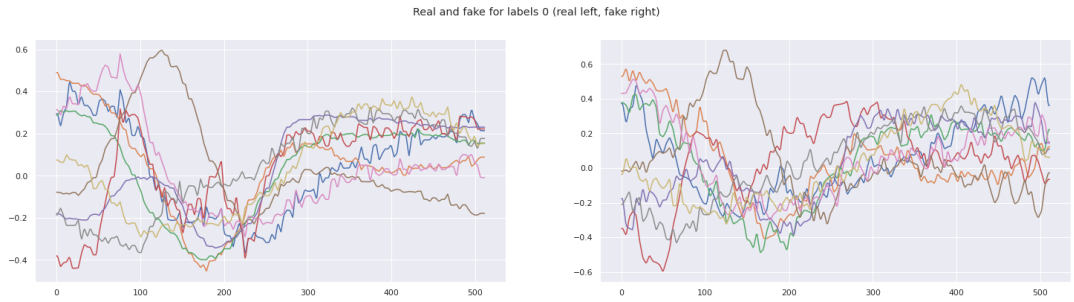

Figure 5.18: A synthetic example of a condition from the AWR dataset produced by the uTSGAN first iteration. True data on the left, generated on the right.

Even though the complexity is significantly reduced it is possible to see from figure 5.18 that the generated samples look very realistic.

5.2.1.4 uTSGAN Second Iteration

The final version combines the two above described versions by increasing the complexity once more by extracting more features from each convolutional layer. Patch loss was again added to the unified loss function. Instead of using the original encoding in the generatorY the encoding from the first iteration uTSGAN was used. The decoding approach from the original uTSGAN, which alternated between upsampling and convolutions, was used to extract even more features. The generated samples from this model can be seen in figure 5.19.

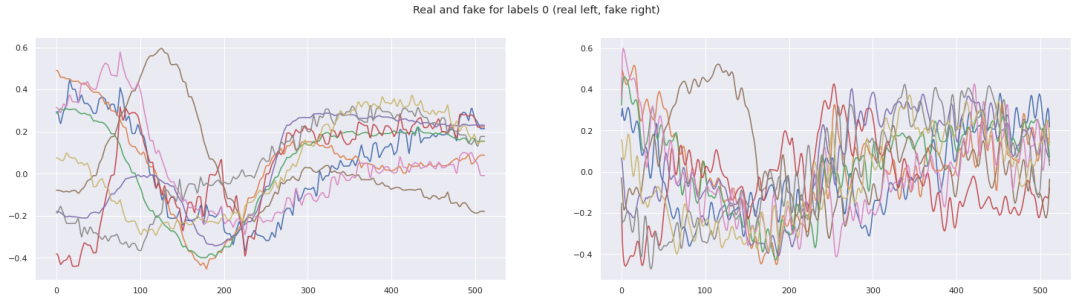

Figure 5.19: A synthetic example of a condition from the AWR dataset produced by the uTSGAN second iteration. True data on the left, generated on the right.

5.2.2 Experimental Method

Now that the four different models (conditional DCWGAN-GP, original uTSGAN, uTSGAN first iteration, and uTSGAN second iteration) have been explored, the experimental method will be explained. During the test of the four models, limitations of the GANs were realized that are necessary to be familiar with to understand the decisions taken in the following experiments. The GANs were in general hard to train both in terms of tuning the hyperparameters to the individual dataset, but also

in terms of computational resources as a model easily could take 8-20 hours to train. This fact has limited the number of datasets to perform the experiments on. The ability to differentiate the samples in the dataset to a class has a big influence on the quality of the data produced by the GANs. As it is not possible to perfectly differentiate a sample from one class to another, the GAN might be asked to produce a sample from class X, but as the distribution of class X overlaps with the distribution of class Y, the generated sample might look more like a sample from class Y. This would especially be the case if the GAN has not perfectly learned the distribution for each class, as illustrated in figure 5.20. This will in the end confuse the classification model that needs to train on the generated data. A very well-performing GAN is therefore necessary for the classification model to be able to benefit from the generated data. If the GAN almost perfectly fits the data distribution its generated samples will "fill in the gaps" of the distribution, resulting in a more robust classification model. For the GAN- and simulated experiments the AWR dataset is therefore used as it is relatively small in size, which enables more models to be tested, and the problem is easy to differentiate into different classes.

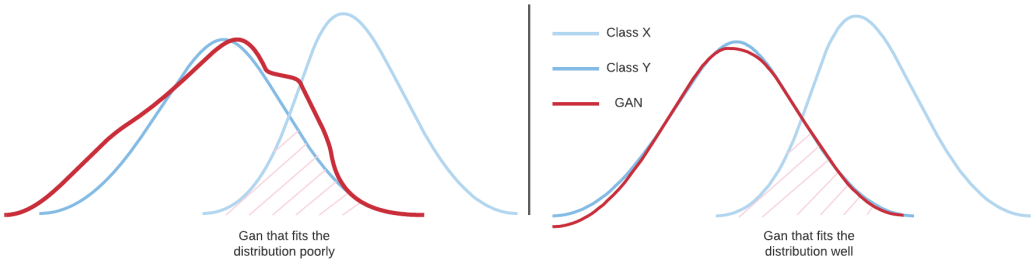

Figure 5.20: A visual description of a GAN that are poor at capturing the distribution of the data thereby increasing the possibility of confusing the classification model (left). And a GAN that fits the distribution of the data well thereby minimizing the possibility of confusing the classification model.

Each of the four GANs was fitted on the AWR dataset. The models were trained for a different number of epochs as they converged at different times. The uTSGAN based models were trained for 1000 epochs while the conditional DCWGAN-GP was trained for 5000 epochs. After the model was fitted to the dataset, different evaluations were used, both to investigate the quality of the data generated and to investigate if the models had mode collapsed. Evaluation metrics of generative models within the time series domain are in general very weak. Within the image domain, the *FID* metric is used as it has empirically been shown to correlate well with humans' perception of how realistic an object look [Heu+17]. This is however not possible to use on time series as big difference in the nature of the data is present from domain to domain. Smith et al. [SS21] developed a FID measure where the FID was calculated for each dataset. It was however determined that the DTW would be a more universal measure of similarity between two time series. To investigate the quality of the generated data, 250 synthetic samples of the time series of each class were generated. For each of the generated samples, the DTW was calculated for each of the true data samples belonging to the same class. The DTW was not restricted as the time series in the dataset are not phase-shifted between samples. The DTW between the generated sample and the true sample with the smallest value is reported back as the one sample that looked the most like the generated sample. The average and standard deviation between all the 250 generated samples will be the resulting quantitative values reported. The smaller the average value is, the more the generated data looks like the true data.

To investigate if the model had mode collapsed, a self-similarity score is calculated. 250 samples from each class are generated. A DTW self-similarity matrix was then calculated for each class, where the values in the lower triangular matrix were selected as seen in figure 5.21. The variance between those distances was then calculated, to see how much variation there is in the synthetic generated samples.

	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7	Sample 8	Sample 9	Sample 10
Sample 1	0									
Sample 2	DTW (1,2)	0								
Sample 3	DTW (1,3)	DTW (2,3)	0							
Sample 4	DTW (1,4)	DTW (2,4)	DTW (3,4)	0						
Sample 5	DTW (1,5)	DTW (2,5)	DTW (3,5)	DTW (4,5)	0					
Sample 6	DTW (1,6)	DTW (2,6)	DTW (3,6)	DTW (4,6)	DTW (5,6)	0				
Sample 7	DTW (1,7)	DTW (2,7)	DTW (3,7)	DTW (4,7)	DTW (5,7)	DTW (6,7)	0			
Sample 8	DTW (1,8)	DTW (2,8)	DTW (3,8)	DTW (4,8)	DTW (5,8)	DTW (6,8)	DTW (7,8)	0		
Sample 9	DTW (1,9)	DTW (2,9)	DTW (3,9)	DTW (4,9)	DTW (5,9)	DTW (6,9)	DTW (7,9)	DTW (8,9)	0	
Sample 10	DTW (1,10)	DTW (2,10)	DTW (3,10)	DTW (4,10)	DTW (5,10)	DTW (6,10)	DTW (7,10)	DTW (8,10)	DTW (9,10)	0

Figure 5.21: A visual representation of the similarity score between the generated samples. The self similarity score is calculated by finding the variance of the blue area.

The uTSGAN based model depends on generatorX producing realistic-looking spectrograms to give generatorY the best starting point for generating realistic time series data. Therefore will the quality of the spectrograms be investigated. This was done in a similar fashion to the time series. 250 synthetic spectrograms were generated for each sensor. The mean squared error was then calculated between each of the generated spectrograms and all of the true. The true spectrogram with the lowest value to the generated was selected as the most similar. The average and standard deviation between all the generated spectrograms is reported back.

The model that produces the best synthetic time series data will be used for both the simulated experiments and for the real-world experiments.

5.3 Simulated Experiments

A series of different classification algorithms were used to investigate which methods result in the most accurate classification of the different open-source datasets. Every proposed model was trained using a grid search where the most optimal selection of hyperparameters was sought out. The investigated dataset was then reduced and the two best classification models were again fitted to the new reduced dataset to set a new benchmark result. Finally, a GAN model was fitted on the reduced dataset to generate synthetic data. The two best classification models were then trained on the synthetic dataset. The purpose was to investigate whether or not it is possible to increase the performance of a model by doing complex data augmentation using a generative model. This process will be explained in further detail in the coming sections.

5.3.1 Phase 1: Benchmark Classification Models

Four different overall classification models are proposed for the time series classification problem. These are the following:

- InceptionTime
- ROCKET with logistic regression with ridge regularization
- ROCKET with logistic regression optimized with SGD
- MiniROCKET with logistic regression with ridge regularization
- MiniROCKET with logistic regression optimized with SGD
- Support vector machine (SVM)

The *random initialized convolutional kernel* models (MiniROCKET and ROCKET) explained in section 3.3.2 have been included because they currently are among the best performing models in the

literature, both in relation to performance metrics but also training times. InceptionTime is included as it also is among the best performing models in the literature [Rui+20]. The results of these models can be seen in table B.7 in the appendix. These results will be compared with the results obtained in the following experiments. These three models are all relatively new (no more than two years at the publishing of this paper). To compare the results a widely used classification algorithm was selected, the support vector machine [BGV92]. The pros and cons related to the different models will be discussed in section 6.

To train a model that most optimally fits a dataset without overfitting is a time-consuming process. A series of hyperparameters can be selected for each of these models. Therefore a grid search algorithm from section 4.2.4 was used to explore multiple combinations of the hyperparameters. Besides making the hyperparameter tuning process considerable easier in the experiment does the grid search process also allow for dynamic hyperparameter tuning and automated continuous training in an operational context, where datasets are updated regularly. A full list of hyperparameters can be found in table 5.2.

Model	**Hyperparameters**
SVM	Alpha: Float
InceptionTime	Epochs: Int
	Learning rate: Float
	Batch size: Int
	Dropout: Float
	Weight decay: Float
ROCKET with logistic regression with ridge regularization	Number of kernels: Int
	Alpha: Float
ROCKET with logistic regression optimized with SGD	Number of kernels: Int
	Alpha: Float
MiniROCKET with logistic regression with ridge regularization	Alpha: Float
MiniROCKET with logistic regression optimized with SGD	Alpha: Float

Table 5.2: List of hyperparameters for each model.

Besides the hyperparameters presented in table 5.2, each model also had a series of prepossessing options like resampling the length of the time series to reduce the number of samples in a given interval. Different scaling methods were implemented to normalize the data with either a mean-, min/max-, robust-, or standard scaler.

The ROCKET models are as explained in section 3.3.2 a transformation of the dataset followed by a linear classification model. As seen in table 5.2, two different classification methods are used based on the authors' recommendations [DSW20], namely a logistic regression with ridge regularization or a logistic regression optimized with stochastic gradient descent. The choice between these two can simply be seen as a hyperparameter that needs to be tuned.

The procedure of finding the best set of hyperparameters, and thereby setting a benchmark for each of the datasets, follows the same structure. For each of the open-source datasets presented in section 5.1 a grid search was defined, where a large interval was chosen for each of the hyperparameters. This was done to get a rough sense of the range each hyperparameter value should be in for that particular dataset. The results of each of the combinations were investigated by plotting the four metrics; accuracy, precision, recall, and an f1-score, explained in section 3.5. Here the models' tendency to overfit is taken into account by evaluating the models' performance on the validation set compared with the training

set. An example of this can be seen in figure 5.22.

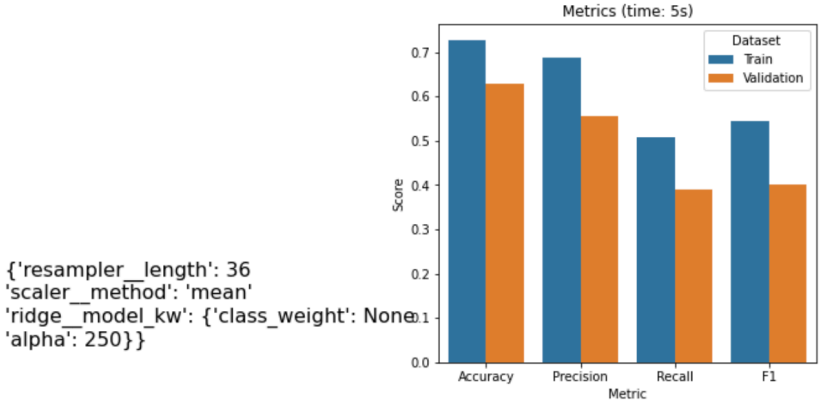

Figure 5.22: An example of an element in the grid search from the LSST dataset training a logistic regression with ridge regularization on a MiniROCKET transformation. On the left the parameters for that model can be found and on the right the metrics on the training and validation set can be found.

After the most promising interval for each hyperparameter was identified, a more narrow range for each of the hyperparameters was selected and a new grid search was defined and executed in the same manner as above. The final tuning of the model results in the closest to optimal hyperparameters. A visual representation of the tuning process can be found in figure 5.23.

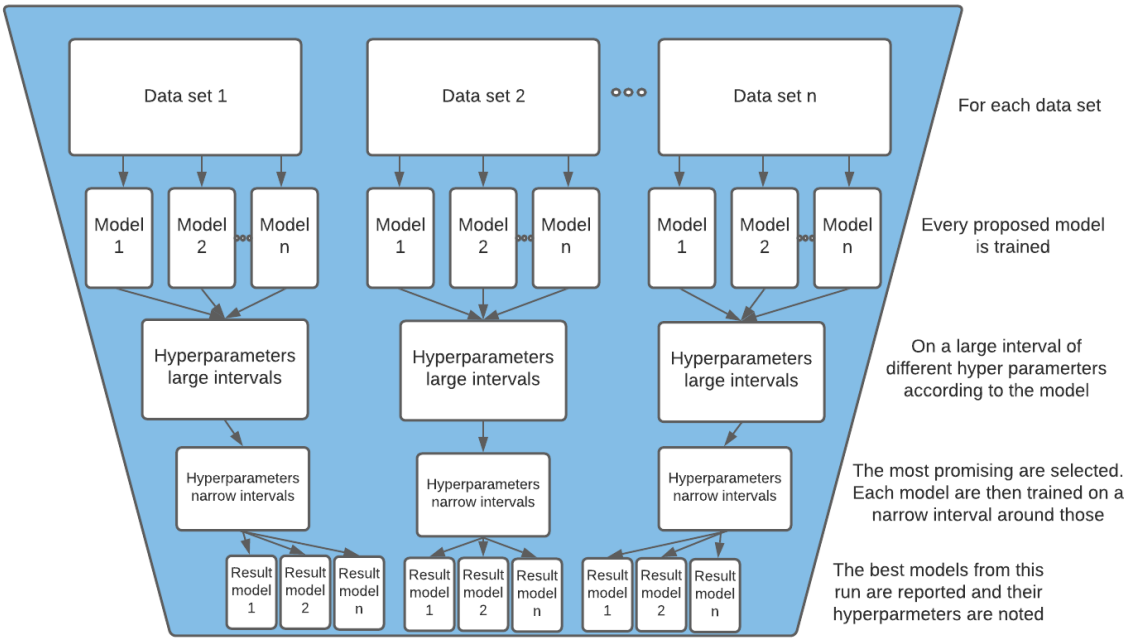

Figure 5.23: The experimental setup for the academic experiments..

Cross-validation was not used as the fundamental architecture of the models was not changed for

each hyperparameter setting. The use of cross-validation would further exponentially increase the number of computational resources needed in the experiment. If more computational resources had been available this step would have been included in the method.

Both the proposed deep learning model (InceptionTime) and the *random initialized convolutional kernel* models (MiniROCKET and ROCKET) seeks to extract features from the time series via transformations that explain the characteristics for the individual class of the data. These transformations are often hard for humans to understand and the models therefore becomes a *black box*. To be able to evaluate the quality of the models' ability to extract relevant features from the time series, manual feature extraction was carried out for comparison. To do this the python library *Time Series Feature Extraction on basis of scalable Hypothesis tests* (tsfresh) was used [Chr+18]. With this, it was possible to calculate 1200 different features from the provided time series, ranging from Fourier transformations to *number of peaks*. The SVM was trained on these features as the SVM is not designed for raw time series data but instead tabular data and would therefore not understand temporal information. The result of these experiments will be discussed in section 6.

5.3.2 Phase 2: Reduce Dataset and Benchmark

The two best models from the first phase were selected for the next phase. Here the datasets were reduced using a stratified sampling method [Ped+21] which ensured that all classes in the original dataset is represented in the reduced dataset and roughly keeps the class distribution. The datasets were reduced to only 20% of the original dataset. The purpose of this reduction was to simulate a smaller data foundation that is expected from an operational setting as explained in section 2.5. The two best classifiers were fitted. This was done to be able to measure the effect of adding the synthetic generated data. This benchmark on the reduced dataset is again found via a wide followed by a narrow grid search.

5.3.3 Phase 3: Train Models on Synthetic Data

The most promising GAN from the GAN experiments is fitted on the reduced data set. Due to the GANs limitations explained in section 5.2 only the AWR dataset has been reduced as the GAN can not perfectly segment the time series based on the condition. It was therefore necessary to look at a dataset where the classification models already had a very good performance. Here the AWR dataset was the best match as the performance was very good for all the models and was a reasonable size so the GAN models could probably train, in contrast to the hydraulic dataset which also had a good performance but very large size. The GAN was fitted for 1000 epochs with the hyperparameters and architecture described in section 4.5. The resulting synthetic data can be seen in section 6.1. The two most promising classification algorithms were again fitted on a dataset containing both the 20% of the original dataset that the GAN was fitted on along with the synthetic data. Theoretically, it is possible to add infinite amounts of synthetic data to the original data. It was however empirically tested how many generated samples per class that was necessary for a model to correctly classify the class of a sample. This was done by fitting a classifier on 1, 2, 5, 10.. 200 generated samples for each class and plotting the relative marginal improvement by adding extra samples. From this, the most reasonable amount of generated samples was selected by assessing the balance between the added computational cost of more samples against the added performance of the model. This point is illustrated in figure 5.24.

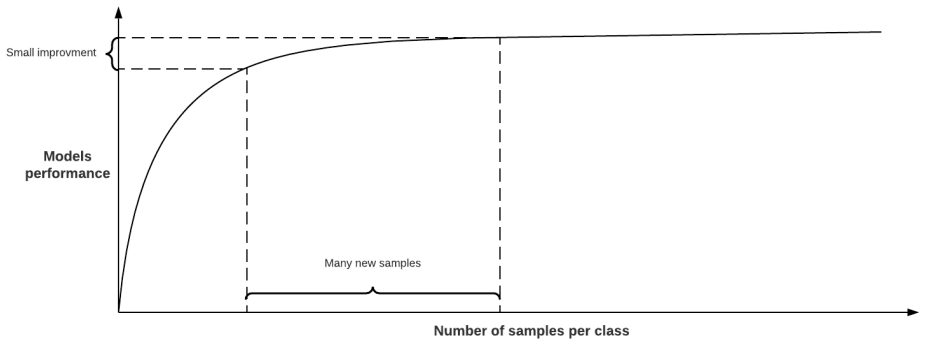

Figure 5.24: Number of samples generated vs. the increase in model performance is balanced in accordance with the increase in computational cost of training the model..

The two classifiers were then fitted with the original data along with the identified number of extra generated samples. The results from these experiments will be discussed in section 6.2 where benchmarks set by other researchers and experiments will be included to compare the proposed method.

5.4 Real-World Experiments

The following section will present the methodology used to conduct the experiment with the data from real-world wind turbines. First, it will be explained how the training and validation data was sampled along with the limitations of the dataset. Finally, the steps of the experiments will be explained.

5.4.1 Construction of the Training- and Validation Dataset

To create a training set for the classification model, resampling of the original dataset is necessary. It is assumed that a failure starts to occur two days before the time noted in the dataset and continues two days after. This assumption will be discussed further in section 6.3. This means that each failure period is of length 4 days with a sampling rate of 10 minutes resulting in 576 samples. Four different ways to organize failures were constructed and tested.

- Binary: An observation is either set to be a failure or no failure.
- Component: The failure occurring in a component is set to the same failure class. Five components are represented in the dataset; Transformer, Generator, Hydraulic System, Generator barring and Gearbox.
- Unique remarks: Several failures had the same remarks, meaning the same failure. These were assigned the same failure class.
- Manually assigned: Each failure was assessed manually into a failure class by assessing the similarities between the remarks. A sensor failure in the transformer component was for example assigned the same failure class as another sensor failure.

The low number of failures (23 data points) made it infeasible to use the *unique remarks* or *manually assigned* methods as this would mean that some of the classes only would have one example. The *component* method was therefore used as a multi-class problem was desired and therefore excluding the *binary* method.

To get a larger number of shorter data samples the four day period was split into equally sized 12 hour periods without any overlap. This resulted in each failure timestamp was converted into 8 samples with a length of 72 each. Some of the intervals had some missing values. An interpolation was therefore performed to fill in the missing values.

The experiment was conducted in the context of optimized troubleshooting. In chapter 1 and 2 the general steps of a guided troubleshooting system was defined along with the scope of this project. The diagnostic of a failure came as a result of a detection of a failure. Therefore it was decided that the data points where no failure have been registered would not be included. This means that the classification algorithm only had to differentiate between failure classes. This decision will be discussed further in section 6.3.

The resulting dataset contains 178 samples. To validate the performance of the proposed model, a sub-sample of the data was kept as a validation set. Because the dataset is relatively small, only 10% of the data was used for validation. It could be argued that the validation data should be sampled from only instances occurring after the training instances, as a model in an operational setting would only predict based on data in the future. This was however ignored for the following experiments, as the scope of the experiment was simply to investigate the different classification algorithms ability to differentiate between a failure in one component compared to another.

5.4.2 Phase 1: Benchmark Classification Models

The two most promising classification algorithms from section 5.3 were fitted on the sampled data. A grid search was used to find the most optimal hyperparameters. As with the simulated experiments, a wide grid search was conducted followed by a narrow to focus on the most optimal range of values for the hyperparameters.

5.4.3 Phase 2: Train Models on True- and Synthetic Data

The most promising GAN was fitted on the training data. Due to the limitations of the GANs explained in section 5.2 it was decided to fit a unique GAN for each of the five classes of the component sampling. This was done to force the different GAN models to find the distribution of a particular class. Due to computational limitations, the GANs were only trained for 500 epochs instead of 1000 as the simulated experiments. The small number of samples resulted in a GAN where only 14 data points were available for fitting the model, as seen in figure 5.25. This will influence the GANs ability to capture the distribution of the data.

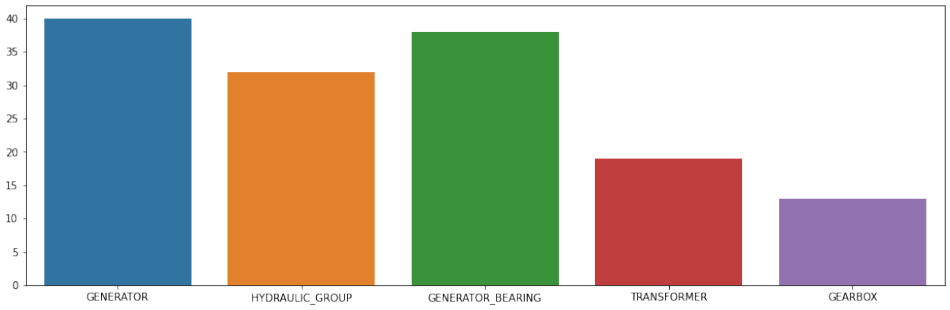

Figure 5.25: The number of samples for the five classes in the real-world experiment.

Each class was then enriched with synthetic samples. The number of samples was determined as described in section 5.3.3. For each class, 50 new samples were generated which balances out the number of samples for each class. The two classification algorithms were then fitted on the true data with the added synthetic data using a wide and narrow grid search.

The experimental method for three different experiments has now been presented. These include a GAN experiment where the aim is to investigate different GAN architectures' ability to create realistic time series data, a simulated experiment where open-source datasets have been used to compare the results of the proposed methods with public published results, and finally a dataset from the real-world to investigate the methods' ability to differentiate between faults in a wind turbines components. The limitations of the different methods and datasets have been presented.

CHAPTER 6

Results and Discussion

In the following section the results from the GAN, simulated- and real-world experiments will be presented and discussed. The results will be put in perspective to the theory and the related work presented in section 3 and 2.

6.1 GAN Experiment

The results of the different GANs explored in section 5.2 will in this section be presented and discussed to see how well the different architectures managed to generate realistic time series as well as their limitations.

6.1.1 Conditional DCWGAN-GP

The first GAN architecture was the conditional DCWGAN-GP, which seemed to produce quite noisy synthetic data. This can be seen in figure 6.1, where the 9 sensors for the first 3 classes are plotted with both the true- and generated data. It can however also be seen that it catches the high-level characteristics of each sensor for each class well.

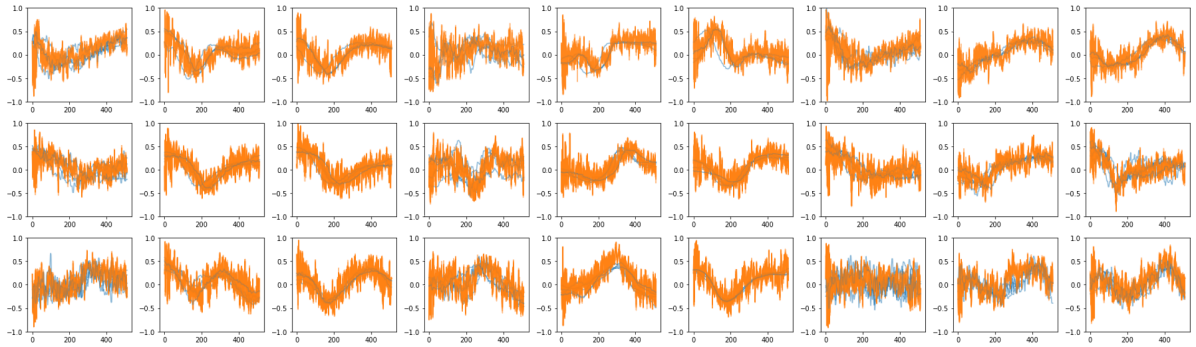

Figure 6.1: 5 synthetic samples(orange) for the first 3 classes(rows) of the AWR dataset generated by the conditional DCWGAN-GP compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor.

To view the results more quantitatively, 250 synthetic time series of each class has been generated and the DTW distance against each sample in the true dataset has been calculated. These results can be seen in table 6.1. The results in the table show that the GAN architecture consistently produces synthetic data that has a DTW distance of around 14, as the standard deviation for each of the classes is low.

Class	0	1	2	3	4	5	6	7	8	9	10	11	12
Mean	14.2	13.5	16.5	16.1	13.0	14.6	14.6	14.7	16.0	13.4	16.2	14.1	15.8
Std	0.09	0.09	0.23	0.14	0.11	0.17	0.14	0.12	0.12	0.10	0.18	0.14	0.14

Class	13	14	15	16	17	18	19	20	21	22	23	24
Mean	14.5	15.2	15.6	14.6	15.9	15.6	14.6	14.5	14.1	14.4	12.7	13.3
Std	0.16	0.16	0.15	0.13	0.17	0.16	0.11	0.17	0.14	0.14	0.11	0.11

Table 6.1: The mean and standard deviation of the DTW distances between each of the generated classes by the conditional DCWGAN-GP to the true classes in the AWR dataset.

From these results, it can be hard to determine if the GAN actually produces different results each time or if it has mode collapsed. Therefore the DTW self-similarity of the generated synthetic data was calculated for each class. The standard deviation of the DTW distances was calculated and can be viewed in table 6.2. Here it can be seen the standard deviation for each class is around 0.4, which indicates that some variance exists between the synthetic samples generated by the conditional DCWGAN-GP. It is however important to note that the generated samples could be more different than described by the variance in the DTW distances. This is because multiple time series, which are very different, can have the same DTW distance as the true time series. Therefore these scores only provide an indication of a potential mode collapse. However, a high DTW self-similarity score does mean that mode collapse does not exist.

Class	Var	Class	Var	Class	Var	Class	Var	Class	Var
0	0.54	5	0.57	10	0.55	15	0.47	20	0.46
1	0.42	6	0.48	11	0.43	16	0.42	21	0.46
2	0.50	7	0.53	12	0.48	17	0.52	22	0.46
3	0.53	8	0.51	13	0.48	18	0.54	23	0.39
4	0.44	9	0.48	14	0.50	19	0.51	24	0.46

Table 6.2: The variance of the DTW self similarity distances of the 250 synthetic samples generated by the conditional DCWGAN-GP.

6.1.2 Original uTSGAN

The next GAN architecture that was explored was the architecture which was based on the implementation provided by the author of uTSGAN K. Smith. Five of the synthetic samples generated by this architecture can be seen for the first three classes in figure 6.2, and it is clear to see how the noise has been reduced and it now has caught the high-frequency characteristics better.

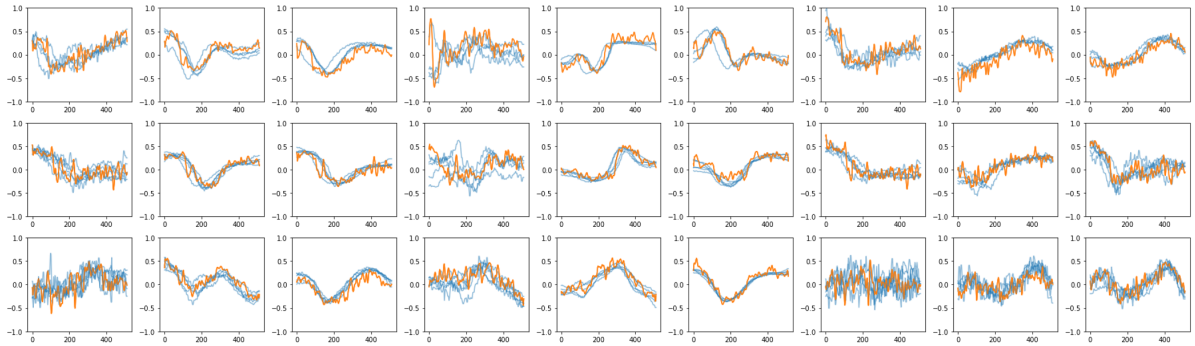

Figure 6.2: 5 synthetic samples(orange) for the first 3 classes(rows) of the AWR dataset generated by the original uTSGAN compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor.

This architecture resulted in a lower DTW distance to the samples it was trained on, which can be seen in table 6.3. It can also be seen the standard deviation for the different classes is in general lower than the standard deviations produced by the conditional DCWGAN-GP. By the look at the synthetic examples in figure 6.2 and the low standard deviations from table 6.3 it seems this architecture did mode collapse.

Class	0	1	2	3	4	5	6	7	8	9	10	11	12
Mean	10.3	8.8	11.3	8.8	7.3	9.7	9.9	10.5	12.3	8.4	11.7	8.4	12.0
Std	0.03	0.01	0.01	0.03	0.02	0.03	0.02	0.03	0.02	0.01	0.02	0.00	0.02

Class	13	14	15	16	17	18	19	20	21	22	23	24
Mean	10.1	11.3	11.1	12.4	10.6	9.9	11.0	8.9	9.1	8.6	7.6	8.0
Std	0.00	0.02	0.05	0.02	0.03	0.02	0.02	0.01	0.02	0.02	0.01	0.02

Table 6.3: The mean and standard deviation of the DTW distances between each of the generated classes by the uTSGANvSmith to the true classes in the AWR dataset.

To verify if the model has mode collapsed, the DTW self-similarity distance has been calculated for 250 samples of each class. The standard deviation for each of the classes can be found in table 6.4, where the really low standard deviations confirm the mode collapse. In fact, it is also much smaller compared to the variance of the DTW self-similarity distances from the conditional DCWGAN-GP.

The reason for this mode collapse is believed to be due to the low batch size that was used to train this architecture. This low batch size was necessary because of the high complexity of the architecture, which meant that each additional sample would consume more memory on the GPU. This low batch size (14 samples for each iteration) meant that on average less than one sample of each class is being trained on as there are 25 classes in the AWR dataset. This means that when the loss is calculated with the Wasserstein distance it cannot estimate the distribution properly, and therefore the GAN does not benefit from this Wasserstein distance in the loss function.

Class	Var	Class	Var	Class	Var	Class	Var	Class	Var
0	0.07	5	0.08	10	0.06	15	0.09	20	0.07
1	0.07	6	0.06	11	0.05	16	0.05	21	0.06
2	0.07	7	0.07	12	0.07	17	0.08	22	0.08
3	0.08	8	0.07	13	0.06	18	0.07	23	0.07
4	0.07	9	0.04	14	0.06	19	0.06	24	0.04

Table 6.4: The variance of the DTW self similarity distances of the 250 synthetic samples generated by the uTSGANvSmith.

To further investigate the original uTSGAN, the synthetic spectrograms it produced were compared with the true ones. In figure 6.3 the synthetic spectrograms look similar to the true ones from figure 5.3. But they also seem to be more smoothed out. To compare them more quantitatively, 250 synthetic spectrograms have been generated for each sensor and the mean squared error between each synthetic sample and all the true samples were calculated. The results can be seen in table 6.5.

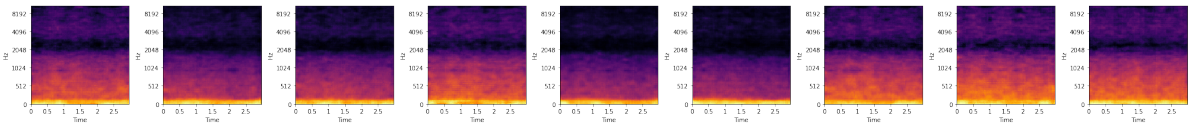

Figure 6.3: A synthetic example of the spectrograms generated by the uTSGANvSmith. Each plot represents a different sensor.

Sensor	0	1	2	3	4	5	6	7	8
Mean	0.018	0.016	0.015	0.018	0.014	0.015	0.019	0.018	0.016
Std	0.0040	0.0036	0.0030	0.0047	0.0027	0.0033	0.0037	0.0045	0.0041

Table 6.5: The mean and standard deviation of the mean squared errors for each sensor spectrogram generated by the uTSGANvSmith.

From table 6.5 it can be seen that the average mean squared error for the spectrograms of each sensor is around 0.017, which is really low and therefore the spectrograms appear to be very true looking.

6.1.3 First Iteration uTSGAN

The architecture for the first iteration uTSGAN was a simpler version of the original uTSGAN. Due to this simplicity, it was possible to increase the batch size to 128, and it was therefore expected that this architecture would be able to better catch the distribution of the data. In figure 6.4 the synthetic data generated by the GAN is shown. It is possible to see how the model again catches the low- and high-frequency patterns of the data along with the condition. Furthermore, it can be seen that multiple variations of each sensor type are visible which indicates that the model has not mode collapsed. This can for instance be seen for the plot in the second column for the first row, where it has produced time series that are completely separable.

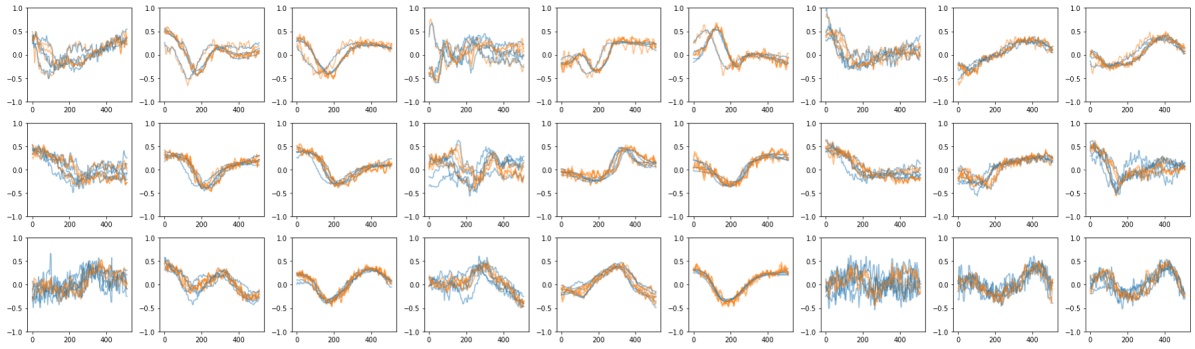

Figure 6.4: 5 synthetic samples(orange) for the first 3 classes(rows) of the AWR dataset generated by the first iteration uTSGAN compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor. A zoom on the first three plots for the first class can be found in appendix B.11, and all plots can be found in appendix B.9 and B.10.

When 250 samples of each class have been produced, it can be seen in table 6.6 that this architecture in general produces times series that are much closer to the true dataset, as the mean values are much lower. It can also be seen how the standard deviations are higher compared to the other architectures.

Class	0	1	2	3	4	5	6	7	8	9	10	11	12
Mean	5.0	5.2	6.6	7.0	5.8	5.5	6.7	7.7	7.8	5.9	7.7	5.6	6.1
Std	0.90	0.98	0.76	0.82	0.84	1.03	0.82	0.87	1.28	0.79	1.28	0.67	1.18

Class	13	14	15	16	17	18	19	20	21	22	23	24
Mean	6.7	7.1	7.8	6.7	6.1	6.5	8.3	6.5	5.4	5.3	4.5	5.3
Std	0.69	0.97	1.07	1.42	0.92	0.96	1.78	0.79	0.82	0.67	0.56	0.80

Table 6.6: The mean and standard deviation of the DTW distances between each of the generated classes by the first iteration uTSGAN to the true classes in the AWR dataset.

As expected the variance of the DTW self-similarity distances is much higher compared to the other architectures. This confirms that this architecture did not mode collapse and it further supports the hypothesis of a small batch size leads to mode collapse.

Class	Var	Class	Var	Class	Var	Class	Var	Class	Var
0	3.7	5	4.5	10	4.0	15	4.7	20	3.5
1	3.3	6	4.4	11	3.4	16	4.5	21	4.3
2	3.4	7	3.4	12	4.0	17	3.3	22	3.2
3	3.3	8	3.9	13	3.7	18	4.1	23	3.1
4	2.6	9	3.4	14	4.3	19	3.7	24	3.2

Table 6.7: The variance of the DTW self similarity distances of the 250 synthetic samples generated by the first iteration uTSGAN.

The spectrograms generated by the first iteration uTSGAN can be seen in figure 6.5. By the look of it, the spectrograms also seem to look realistic. They actually seem to be more realistic looking due to the increased amount and size of the black spots in the spectrograms which are visible on the real spectrograms, see figure 5.3. But when looking at the statistics in table 6.8, the average mean squared

error of around 0.021 is significantly higher compared to the results of the original uTSGAN. As the only difference affecting the spectrograms between those variations is the added patch loss, it indicates that patch loss quantitatively improves the generation of spectrograms.

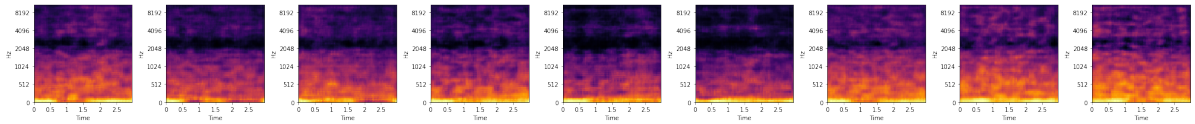

Figure 6.5: A synthetic example of the spectrograms generated by the first iteration uTSGAN. Each plot represents a different sensor.

Sensor	0	1	2	3	4	5	6	7	8
Mean	0.022	0.018	0.018	0.023	0.020	0.018	0.021	0.024	0.025
Std	0.004	0.004	0.005	0.004	0.005	0.006	0.004	0.006	0.009

Table 6.8: The mean and standard deviation of the mean squared errors for each sensor spectrogram generated by the first iteration uTSGAN.

This first iteration of the uTSGAN showed great potential as it in general generated synthetic data with a low DTW distance to the true data, as seen from table 6.6. It also managed to create synthetic data with variation which can be seen in figure 6.4 and is supported by table 6.7. Therefore it was desired to look into the differences between the synthetic data generated by the first iteration of the uTSGAN and true data. To do this, 250 synthetic samples were produced and the true sample they were most similar to was subtracted for each of them. This left behind the noise generated by the generative model which was used to investigate its spectrum. One sample of the noise on the first three sensors can be seen at the top of figure 6.6, and at the bottom, the spectrum of 250 samples can be seen for the same first three sensors.

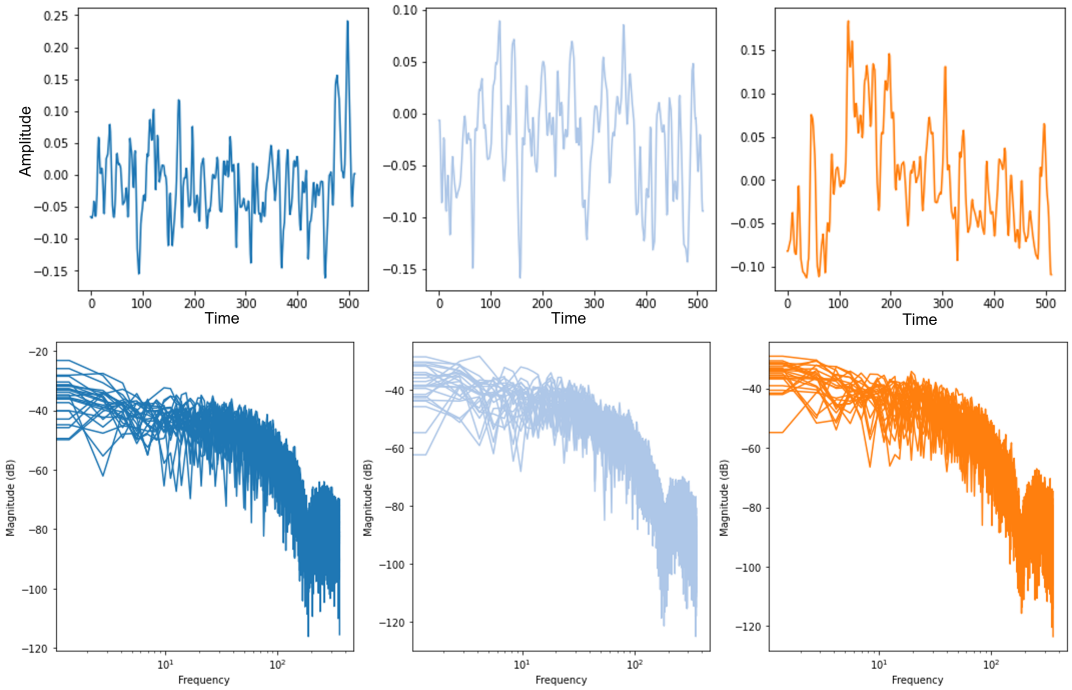

Figure 6.6: The noise generated by the first iteration uTSGAN on the first three sensors. In the top one noise sample is shown per sensor. In the bottom 250 samples and their spectrum is shown per sensor.

From figure 6.6 it can be seen that across the sensors the same noise is generated. If these noise spectra are compared with the different colours of noise [Wik21], it is seen that the first iteration of the uTSGAN does not learn just to add coloured noise. It is however still the same type of noise it learns to add to the true data across the time series samples, which could indicate that if it was possible to sample noise of this type, that noise could just have been added to the true data. It is however not known if the generative model would learn to generate the same type of noise across different datasets.

6.1.4 Second Iteration uTSGAN

The second iteration of uTSGAN architecture was a bit different from the original uTSGAN architecture but was also quite complex in the upsampling in generatorY. This meant that this architecture also had to be trained with a low batch size of 25. The results of five synthetic time series for the first three conditions can be seen in figure 6.7. It can again be seen that it catches the overall characteristics of each sensor fairly well, however, it also shows some spikes around the middle of the time series for each sensor for the second class. The spikes indicate failures within the GAN to generate realistic time series. During the development of the models, the same spikes were observed but disappeared once the model started to converge probably.

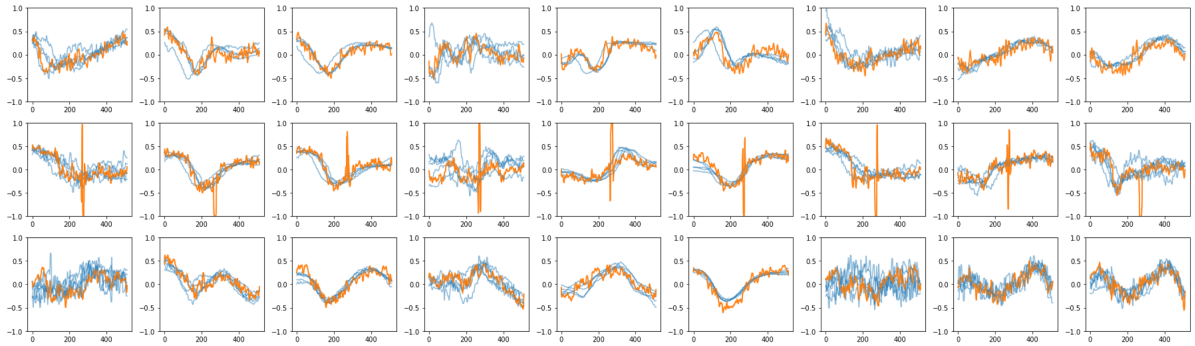

Figure 6.7: 5 synthetic samples(orange) for the first 3 classes(rows) of the AWR dataset generated by the second iteration uTSGAN compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor.

The average DTW distance to the true training set is around 10 with a very low standard deviation, as seen in table 6.9. By the look of it, it is hard to spot any variations in the synthetic time series from figure 6.7, which could indicate that this architecture has mode collapsed as well.

Class	0	1	2	3	4	5	6	7	8	9	10	11	12
Mean	8.5	12.2	9.1	9.2	7.7	11.4	8.3	9.6	11.9	9.1	10.8	7.5	9.1
Std	0.05	0.08	0.02	0.03	0.06	0.12	0.06	0.05	0.07	0.05	0.08	0.04	0.04

Class	13	14	15	16	17	18	19	20	21	22	23	24
Mean	10.7	11.1	13.8	11.4	11.1	9.4	10.3	9.3	10.4	8.7	7.1	8.2
Std	0.03	0.05	0.05	0.03	0.06	0.06	0.06	0.06	0.03	0.05	0.05	0.05

Table 6.9: The mean and standard deviation of the DTW distances between each of the generated classes by the second iteration uTSGAN to the true classes in the AWR dataset.

To confirm if the architecture has mode collapsed, the variance of the DTW self-similarity distances for each class has been calculated and put into table 6.10. Here it can be seen that the standard deviations are significantly higher compared to the ones produces by the original uTSGAN. This also supports the hypothesis of the low batch size introducing mode collapse, as this architecture only uses a slightly higher batch size compared to the original uTSGAN.

Class	Var	Class	Var	Class	Var	Class	Var	Class	Var
0	0.48	5	0.66	10	0.51	15	0.54	20	0.51
1	0.45	6	0.55	11	0.45	16	0.55	21	0.52
2	0.54	7	0.59	12	0.54	17	0.55	22	0.58
3	0.49	8	0.59	13	0.49	18	0.48	23	0.44
4	0.46	9	0.51	14	0.51	19	0.57	24	0.42

Table 6.10: The variance of the DTW self similarity distances of the 250 synthetic samples generated by the second iteration uTSGAN.

The spectrograms produced by the second iteration uTSGAN can be found in figure 6.8. It can be seen that they also look very realistic compared to the true spectrograms, and they actually seem to understand that they are not smooth. But when looking at the statistics in table 6.11, it can be seen that

the average mean squared error for each sensor, in general, is a bit higher compared to the spectrograms of the original uTSGAN. This indicates that the uTSGAN architecture benefits from the last linear layer in discriminatorX, as that is the only difference between the original uTSGAN and second iteration uTSGAN in generatorX and discriminatorX. It is interesting to see how these spectrograms actually are more true looking, but quantitatively are more different from the true spectrograms. This could be due to some of the small black spots in the spectrograms are slightly shifted in time or frequency, which results in a higher error, but still looking realistic.

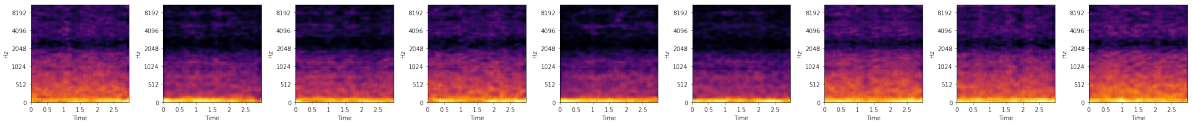

Figure 6.8: A synthetic example of the spectrograms generated by the second iteration uTSGAN. Each plot represents a different sensor.

Sensor	0	1	2	3	4	5	6	7	8
Mean	0.019	0.016	0.015	0.020	0.014	0.014	0.020	0.020	0.018
Std	0.0046	0.0042	0.0040	0.0050	0.0041	0.0045	0.0048	0.0045	0.0043

Table 6.11: The mean and standard deviation of the mean squared errors for each sensor spectrogram generated by the second iteration uTSGAN.

6.1.5 General GAN Discussion

As seen previously in this section the different GAN architectures produce different results. In general the utilization of the uTSGAN as the backbone architecture resulted in much more true looking time series compared to the conditional DCWGAN-GP.

While the first iteration uTSGAN provided the best synthetic time series and the most varying time series there is still room for improvements. In general it is generatorY that seems to be the bottleneck in the architecture as it seems to infer some high frequency noise. This could be affected by the frequency leaking due to the choice of window size. Therefore, the architecture could benefit from multiple spectrograms per sensor, each with a different window size. In that way it could maybe more easily learn the exact frequencies and exact point in time it changes.

In the AWR dataset, where the GAN architectures were validated on, the spectrograms look quite similar for each sensor across the different classes. In datasets where the spectrograms for each sensor looks very different for each class, it would be expected that it will not perform as well. This is due to generatorX and discriminatorX not being conditional, and therefore the spectrograms generated by generatorX would just be a random sample and not necessarily look like the class it should generate the time series for. This means that the synthetic spectrogram would be less ”hinting” which class it should generate a sample for. In the uTSGAN article, the authors argues that the generation of realistic looking spectrograms gives generatorY much better prerequisites for producing realistic time series [SS21]. It would therefore be expected that making generatorX and discriminatorX conditional, would make the architecture more robust to spectrogram variation across the classes.

Even though the uTSGAN produces fairly good conditioned multivariate times series, it certainly has its limitations. The first iteration uTSGAN produces time series data with noise of higher frequencies. This is fine for the AWR dataset, as the separation of the different classes does not seem to be dependent on the higher frequencies but instead the lower frequencies and thereby the higher-level characteristics. It could be argued that this would not be suitable for generating synthetic data for high frequency

signals, such as vibration measurements. But as much signal data from sensors on assets is aggregated and sent every tenth minute, it is not expected that the change in the signals are of higher frequency and thereby the different conditions should be separable by the general characteristics of the time series.

The first iteration uTSGAN also has more general limitations in form of scalability. Firstly the amount of sensors affects the complexity of the first iteration uTSGAN by increasing the initial amount of channels. But the real scalability limitation is regarding the number of different classes. Because when increasing the amount of different classes, the average amount of classes that will be present in each batch gets smaller. As described above this leads to mode collapse and will not let the model generate synthetic data from the whole distribution. A possible way to compensate for this, is to use batches with samples of only a few classes at a time. This should lead to a better estimation of the distribution within each class. This however potentially results in the gradients not being estimated accurately and thereby possibly not being able to distinct between the different classes.

Another way to potentially work around this issue is to train model for each of the classes. That ensures the classes will be generated correctly and estimate the distribution better. However, this potentially leads to a lot of models to train and might therefore not be a desired option, especially in a continuous learning setup. This hypothesis is tested in the real world experiment in section 5.4.

6.2 Simulated Experiment

The following section will present the results obtained from the three different phases of the simulated experiments. The results will be discussed and compared to benchmark results from the literature.

6.2.1 Results Phase 1: Benchmark Classification Models

After doing a grid search for all the models and datasets the following results were obtained. The presented tables of results show the most optimal hyperparameters after the wide and narrow grid search. In table 6.12 the results from the dataset "Articular word recognition" is shown. The first observation is that all model, in general, performs well. The time series from the individual classes are also relatively easy to tell apart and do not vary much between samples, see figure 5.4 for an example.

Method	**Time**	**F1**	**Accuracy**	**Hyperparameters**	**Comment**
InceptionTime	1 m	0.995	0.999	Epochs: 100 Learning rate: 0.0001 Batch size: 32 Dropout: 0.5 Weight decay: 0.001	
ROCKET with logistic regression with ridge regularization	37 s	0.998	0.999	Alpha: 11 Number of kernels: 10000	
MiniROCKET with logistic regression with ridge regularization	1 s	0.990	0.993	Alpha: 2.5	
SVM on tsfresh feature extraction	8 h	0.986	0.990	Alpha: 0.1 Kernel: RBF	Feature extraction took 8 hours SVM took 10 seconds

Table 6.12: The results from the AWR dataset using the proposed classification algorithms.

The best performing models were the deep learning model InceptionTime explained in section 3.3.1 and the ROCKET transformer with a logistic regression with ridge regularization explained in section 3.3.2. Here an almost perfect segmentation of the classes was possible to achieve as seen in figure 6.9.

Note here how well the model generalizes to the data showing no signs of overfitting, this is partly due to the use of dropout and weight decay.

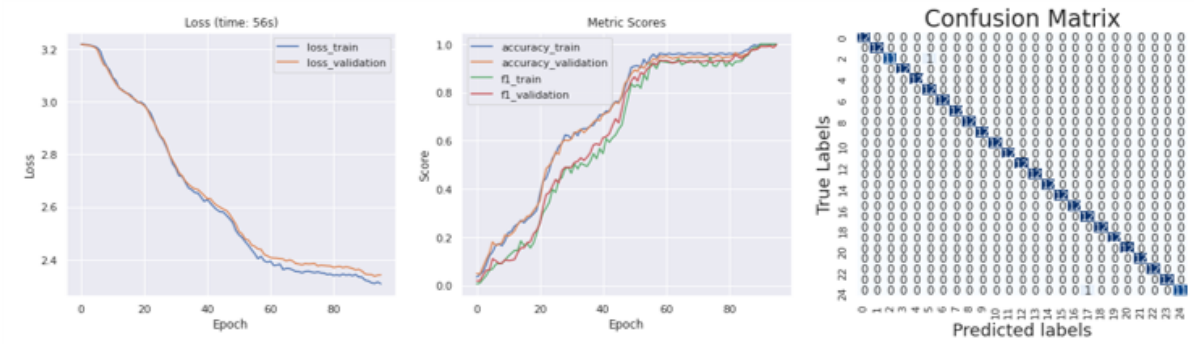

Figure 6.9: The loss curvec (left), performance metrics (middle) and confusion matrix (right) for the InceptionTime model trained on the AWR dataset.

The difference between the performance for the ROCKET and MiniROCKET transformer is very small. It is however apparent that the ROCKET transformer outperforms with a small margin. This is possibly due to double the amount of extracted features from the times series. When taking the fitting time into consideration the MiniROCKET transformer is superior. When comparing these results to the SVM on the feature extraction, the advantages with the ROCKET transformers become apparel as the manual feature extraction is 28.800 times slower than the MiniROCKET transformer without any difference in performance.

In the article by Ruiz. A. et al. [Rui+20] the recent state of the art methods within Time series classification have been discussed. Here some of the proposed methods have also been benchmarked against the same datasets. This enables a comparison to validate the results achieved in the presented experiments. They present the result from 12 algorithms including the InceptionTime and ROCKET. For the AWR dataset, the best performance was achieved with the ROCKET algorithm with an accuracy of 99.56 closely followed by the InceptionTime algorithm with 99.10 [Rui+20]. This corresponds well with the presented results. It is therefore fair to conclude that the proposed grid search method to tune the hyperparameters have succeeded and can obtain similar results as benchmarks from the literature. It is however important to focus on the fact that the results presented in [Rui+20] is a average over 30 independent runs where only one run were performed in the presented experiment due to hardware limitations.

The LSST dataset contains a harder problem to solve than the AWR which can be seen in the results in table 6.13. In general, all the models performed in a similar fashion. Interesting to notice is the large difference in performance between the ROCKET and MiniROCKET transformers. The MiniROCKET transformer has better performance on every parameter, furthermore did it not overfit as much as the ROCKET transformer. Overfitting was in general an issue with this dataset which can be seen in the large values for the regularizing parameters (alpha and dropout).

When comparing the results with the benchmark results from [Rui+20] the same picture emerges as with the AWR dataset. One interesting observation is however with the IT algorithm. Here the benchmark accuracy is only 33.97. It would therefore be necessary to rerun the experiments for several more iterations to see if the average accuracy is reduced as the presented results significantly outperform the benchmark scorers. The ROCKET algorithm achieves the best results in the benchmark with an accuracy of 63.15 very similar to the results from the MiniROCKET.

Method	Time	F1	Accuracy	Hyperparameters	Comment
InceptionTime	15 m	0.44	0.60	Epochs: 5000 Learning rate: 0.0001 Batch size: 32 Dropout: 0.7 Weight decay: 0.0001	
ROCKET with logistic regression with ridge regularization	45 s	0.35	0.55	Alpha: 4500 Number of kernels: 10000	Overfitted
MiniROCKET with logistic regression with ridge regularization	16 s	0.45	0.65	Alpha: 220	
SVM on tsfresh feature extraction	16 h	0.45	0.62	Alpha: 50 Kernel: RBF	Overfitted Feature extraction took 16 hours SVM took 10 seconds

Table 6.13: The results from the LSST dataset using the proposed classification algorithms.

The large size of the Hydraulic dataset resulted in some issues. As the only dataset, it was necessary to resample each time series to a smaller size which is indicated in table 6.14 as *resampling length*. This did however not seem to affect the algorithm's ability to correctly classify the samples. Especially did the InceptionTime and MiniROCKET algorithms manage to almost perfectly separate the classes as seen in figure 6.10.

To perform the feature extraction the dataset had to be resampled to a length of 500 as the dataset was too large to calculate the features. This did reduce the performance significantly of the SVM on this feature extraction.

Method	Time	F1	Accuracy	Hyperparameters	Comment
InceptionTime	60 m	0.999	0.999	Epochs: 250 Learning rate: 0.008 Batch size: 128 Dropout: 0.0 Weight decay: 0.005 Resampler length: 3000	
ROCKET with logistic regression optimized with SGD	28 m	0.810	0.898	Alpha: 0.0002 Number of kernels: 10000 Resampler length: 3000	
MiniROCKET with logistic regression with ridge regularization	7 m	0.993	0.993	Alpha: 0.6 Resampling length: 3500	
SVM on tsfresh feature extraction	20 h	0.242	0.750	Alpha: 1.0 Kernel: RBF	Downsampled times series from 6000 to 500 before feature extraction. Feature extraction took 20 hours SVM took 10 seconds

Table 6.14: The results from the Hydraulic dataset using the proposed classification algorithms.

Figure 6.10: The loss curve (left), performance metrics (middle) and confusion matrix (right) for the InceptionTime model trained on the Hydraulic dataset.

From the results presented in this section, InceptionTime and MiniROCKET have shown to provide the best results and was therefore used for the remaining experiments.

6.2.2 Results Phase 2: Reduce Dataset and Benchmark

From the learning acquired from the GAN experiments, it was decided that only the AWR dataset would be investigated further. This means that only this dataset will be reduced so a GAN can be fitted and synthetic data can be generated. The two best algorithms (InceptionTime and MiniROCKET) was fitted on a subset corresponding to 20% of the original dataset, again using a grid search to find the most optimal parameters. The results from this can be seen in table 6.15.

Method	**Time**	**F1**	**Accuracy**	**Hyperparameters**	**Comment**
MiniROCKET with logistic regression with ridge regularization	2 s	0.956	0.970	Alpha: 0.1	Overfitted more than with the full dataset
InceptionTime	4 m	0.970	0.971	Epochs: 200 Learning rate: 0.0001 Weight decay: 0.001 Dropout: 0.5 Batch size: 64	Overfitted more then with the full dataset

Table 6.15: The two best algorithms trained on only 20 % of the AWR dataset and validated against the other 80 %.

Most noticeable is the algorithms' ability to classify the time series correctly despite only having $\frac{1}{5}$ the original data. The models' ability to generalize is however reduced leading to overfitting which can be seen in figure 6.11.

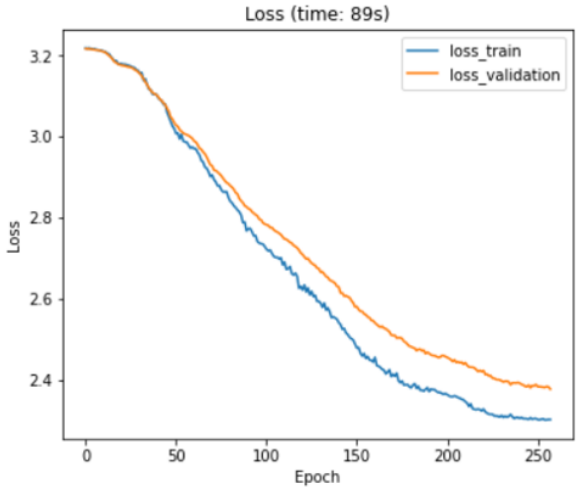

Figure 6.11: The loss curve for the reduced AWR dataset indicating overfitting.

6.2.3 Results Phase 3: Train Models on Synthetic Data

Finally, two new classification models were fitted where synthetic data have been added to the reduced dataset to investigate if the models would benefit from synthetic data.

The number of generated samples per class was determined by fitting a classification model (InceptionTime) on different numbers of only synthetic data samples and validated on the 80% original data, see figure 6.12. From the figure, the quality of the data produced seems promising. The plot shows a quick increase in performance with a following slower increase after reaching a score of around 0.95. This shows that adding more instances of a label increases the score, but the computational cost vs. performance should be balanced. 200 samples of every class were therefore chosen to be added to the existing data. Furthermore does the plot show that the classifier do not necessary need samples from the true dataset to perform well on unseen true data, indicating a good performing GAN model.

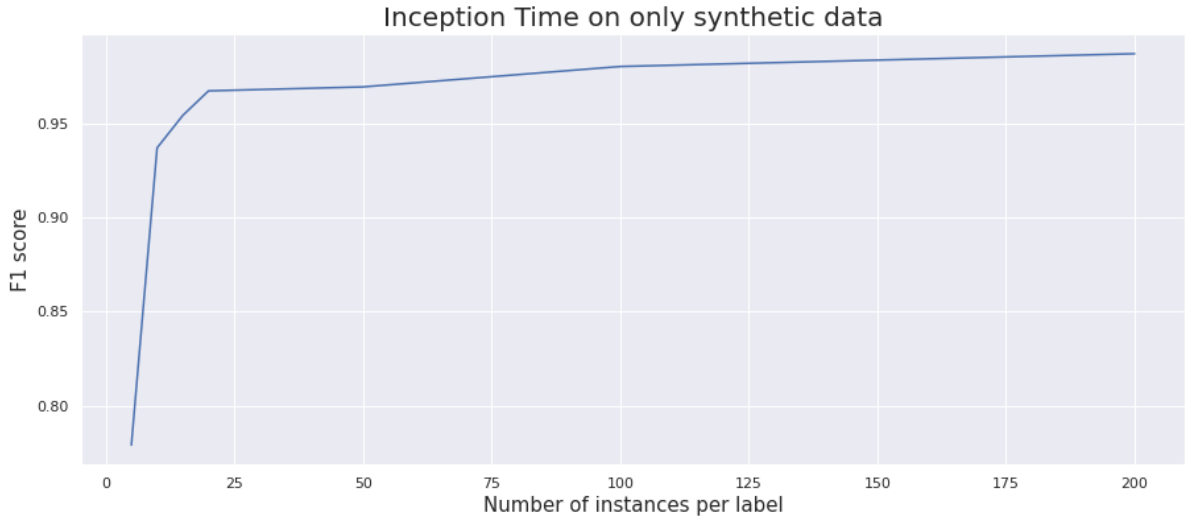

Figure 6.12: The performance of the InceptionTime Model only fitted on synthetic data and validated against all of the omitted true data for different number of instances of the labels..

In table 6.16 the results from a grid search for the MiniROCKET and InceptionTime algorithm trained on the reduced dataset with synthetic data included is shown.

Method	Time	F1	Accuracy	Hyperparameters	Comment
InceptionTime	20 m	0.994	0.996	Epochs: 300 Learning rate: 0.0001 Batch size: 32 Dropout: 0.5 Weight decay: 0.0001	
MiniROCKET with logistic regression with ridge regularization	16 s	0.853	0.864	Alpha: 0.1	

Table 6.16: The two best algorithms fitted on the reduced dataset with synthetic data added. The score is from the validation set which contains the data that was omitted.

The MiniROCKET algorithm did not benefit from the added data as the performance is reduced when comparing it to the MiniROCKET when only fitting on the reduced data. This can perhaps be explained by the MiniROCKET's almost deterministic nature. The more "unnatural" data points, caused by some of the generated data, therefore may have conflicted with the majority of the correct data points from both the generated and true. The InceptionTime however may have been able to learn that some of the more "unnatural" data points should be put less weight on. This is clear from InceptionTime's general performance. With synthetic data, the classifier manages to get a slightly better performance score. It was expected that the addition of synthetic data would decrease the models' tendency to overfit as the increase of examples from the whole distribution would help the models understand the general behaviour of the data. This was however not possible to see when inspecting the loss from the model fitted on both the synthetic and true data. This is probably due to the fact that the GAN model did not perfectly fit the training data as discussed in section 6.1. It is speculated that the generated distribution did not match the true distribution resulting in the model being "pushed" in the wrong direction where samples were misclassified because the overlap between the two underlying distributions was to similar.

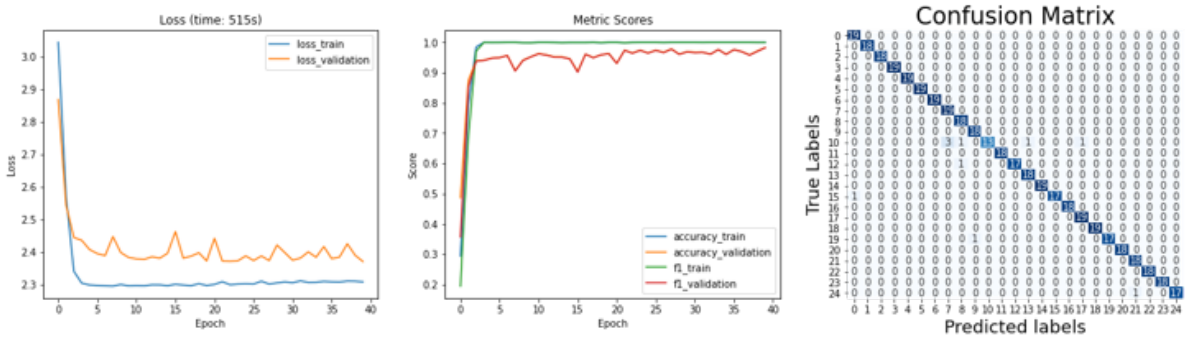

Figure 6.13: The loss curve (left), performance metrics (middle), and confusion matrix (right) for the InceptionTime model trained on the reduced + synthetic AWR dataset.

The small increase in performance when fitting the classifier with synthetic data is relevant to discuss. First of all, a statistical test would need to be performed to probably validate whether or not the model's performance increase is random or consistently better. Here a t-test could be used. Secondly, it should be investigated if a simple data augmentation would yield similar results as the performance increase

is so small. Here some simple Gaussian noise could be added to the input time series. As discovered in section 6.1.3, the noise the generative model learned to add to the true data is not a known type of coloured noise. This could indicate that simple Gaussian noise or any coloured noise would not have the same impact on the performance of the classification model. This should however be validated. The GAN's effect on the models' performance is not significant.

It can, in general, be said that the GAN in all managed to capture the underlying distribution of the data especially the loe frequency patterns. The results from the MiniROCKET however suggest that some improvements still can be done. These few inconsistencies in the generated data can most likely be explained by the quality of the GAN. It was hoped that the GAN would fill in the distribution where values in the dataset was missing, as illustrated in figure 6.14.

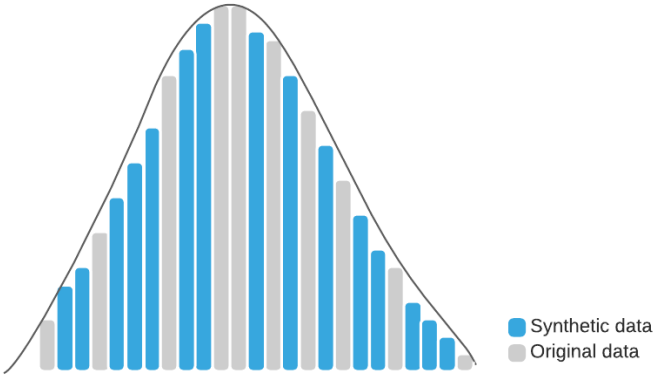

Figure 6.14: The GAN's ability to capture the underlying data distribution of a dataset helps the classification model to generalize better based on the synthetic data..

6.3 Real-World Experiment

The following section will present the results from the experiments performed on asset data from the real-world explained in section 5.4. The section will be divided into two. First, the results from the two best classification algorithms identified in section 6.2 will be presented. Next, the results from applying a GAN on the real-world dataset is presented.

6.3.1 Benchmark Results without Synthetic Data

The MiniROCKET and InceptionTime algorithms were fitted on the dataset using a wide followed by a narrow grid search to find the most optimal set of hyperparameters. The results from this can be found in table 6.17.

Model	Time	F1	Accuracy	Hyperparameters	Comment
InceptionTime	23 s	0.40	0.45	Epochs: 51 Learning rate: 0.001 Batch size: 32 Dropout: 0.7 Weight decay: 0.0005	Overfitted some
MiniROCKET with logistic regression with ridge regularization	1 s	0.58	0.65	Alpha: 0.1	Overfitted some

Table 6.17: The results from the real-world dataset using the proposed classification algorithms.

The MiniROCKET algorithm achieves the highest score of 0.58 over the InceptionTime's 0.40. This was an unexpected result as the two algorithms in general have had similar performance in the simulated experiments. The MiniROCKET did overfit more than InceptionTime which can give a skewed perception of the results. The general overfitting was combated with extensive regularization parameters on the InceptionTime (note 0.7 dropout and 0.0005 weight decay) which did help. Regularization on the MiniROCKET did however not have a big effect. The general overfitting is most likely coursed by the small data sample for both training and validation and the sampling technique described in section 5.4. The model only sees a few examples for a given failure in a small number of surrounding circumstances. This makes it difficult for the model to figure out exactly what patterns in the data to identify as being associated with a given failure. The sensor data from a given failure will e.g. look very different in a storm versus a beautiful summer day. For the model to correctly identify the failure it must see more examples in very different circumstances. Another limitation is the fact that the data comes from five different wind turbines. The different conditions an individual wind turbine experiences are influenced by the location of the wind turbine. The sensor data coming from these turbines would therefore vary by some margin. In an ideal situation, it would therefore be most optimal to either have a model for each of the turbines or group turbines that operate in similar conditions. In the case of this experiment, the wind turbines are located at the same wind farm, so the conditions are assumed to be similar. Research however indicates that large differences in failure rate can occur internally in a wind park as some turbines are experiencing heavier forces if they e.g. are located on the edge of the wind turbine array [Tav+13]. This information would be relevant to include in the *strategy decision model* explained in chapter 1. A general discussion point is the fact that the proposed model, in general, can be considered *black-box* models. This means that it is near impossible to identify how the model actually made its decision. It is expected that some operator or a technician would have trouble following advice from a model where it maybe would not be completely clear why it suggested what it did. Making a model more transparent could therefore be an interesting subject for future studies. A start could be to report how certain a model was on its diagnosis.

The sampling of the data is also relevant to discuss. As the exact interval for where the failure occurred to when it was resolved is not given, some assumptions had to be made. It was assumed that failures were represented in the data 2 days before and after a failure was detected. It is however possible that a failure was either resolved much faster or slower, meaning that a data sample represented as a failure was actually a "normal" state leading to confusion in the model. For the algorithms to be used in an operational setting more detailed logging and sampling of the data is therefore needed to get a more accurate picture of the duration of the failure.

The obtained results indicate that it is possible for the algorithms to differentiate the failures on the different components to a wide degree. It is however also doubtable that a perfect segmentation of the failures on the failure of the component is achievable. This is partly due to the uncertainty in sensors readings where noise are introduced into the measures.

For this project, it has been assumed that only one failure can occur simultaneously. This is of course not the case in the real-world. This problem has been discussed and explored extensively in the

literature [Sha+94; Ise05; Ott12]. The current methods proposed in the literature rely on traditional algorithms like decision trees and graph searches that gets exponentially more complex with more failures at the same time. Shakeri et al. argue that by testing a system often enough, the probability of multiple faults occurring simultaneously is reduced [Sha+94]. The possibility can however not be eliminated completely. The explored InceptionTime algorithm uses the softmax activation function in the final layer to get the probability of the input belongs to the individual class. these probabilities sum to one across the classes. This activation function could be replaced with a sigmoid activation which outputs a number between zero and one for every class. This would allow the model to output more than one prediction for each input. This could therefore be an interesting change when maybe combined with a *strategy decision model*.

It was decided that classifying the components for where the failure occurs would be more beneficial than classifying the failures themselves as some failures were only registered once, making it impossible to train and validate. A failure in one part of the component might present itself in the sensor data very differently than another failure in another part in the same component. It could therefore be interesting if more data became available to investigate if the performance of the classification model could be maintained by looking at the individual failure over the component as a whole. Even though the model is not able to tell exactly what failure has occurred, the ability to suggest what component to investigate first is still a big advantage compared to starting from scratch.

In section 5.4 it was explained that it was assumed that these models would only be deployed once a failure had been detected by some classification model or operator/technician. This meant that no data point from normal operation was included in the model. It could however be argued that a model should be able to return without a suggestion for a fault. This would especially be in the case where an operator or technician have observed something unusual and wants to investigate if the system could identify a failure. Here the model should have the opportunity to return without a suggestion. It would however be unnecessary if an anomaly detection model triggered the diagnosis model that then returns without a suggestion. The addition of a *normal class* would further complicate the model.

It is in general hard to evaluate how good the models actually perform as no other results from this dataset exist. By taking the limited amount of failures in the dataset into account, the results are promising as they clearly indicate that there are some sensor signals that allows the classification models to differentiate between the components from where the failure occurs. This performance is interesting to compare with the medical studies presented in [Cli+13]. The performance for the predictive model was around 90 %. The high accuracy is in general expected with medical models as wrong predictions potentially could be lethal. The accuracy of the proposed models on industrial assets do not reach as high of an accuracy. It can be argued that the necessity of a high accuracy is not that important as with medical devices as the worst that can happen with a false positive is that a technician would have to keep looking for the failure. In the experiment conducted in [Cli+13] not only sensor values was used to predict the diagnosis of a patient but also clinical observation. This unification of data types is very similar to what is expected when a *strategy decision model* is implemented as described in chapter 1. It is therefore also expected that a higher accuracy can be achieved on the long term, both because more data would become ideal but also because a model capable of also handling unstructured data could be implemented. The logging of more data is relevant to discuss. The proposed infrastructure is designed with continuous training in mind as it allows for retraining of the classification models. The proposed method for automatic creating labeled datasets for automatic debugging in [ESK18] (presented in 2.4) could be applied to the problem at hand. If the operators or technicians were asked to log the fault and steps eventually performed to fix the problem, a dataset could be generated automatically. The sensor data from the point in time where the fault was first detected up to the fix would be the training data and the label would be the log created by the technician.

6.3.2 Results with Synthetic Data

The MiniROCKET and InceptionTime algorithms were fitted on the dataset with synthetic data using a wide followed by a narrow grid search to find the most optimal set of hyperparameters. The results from this can be found in table 6.18.

Method	Time	F1	Accuracy	Hyperparameters	Comment
InceptionTime	3.5 m	0.39	0.47	Epochs: 320 Learning rate: 0.001 Batch size: 64 Dropout: 0.5 Weight decay: 0.005	Overfitted a lot
MiniROCKET with logistic regression with ridge regularization	1 s	0.31	0.36	Classifier: Ridge Alpha: 0.8	Overfitted a lot

Table 6.18: The results from the real-world dataset with added generated data using the proposed classification algorithms.

As with the simulated experiments, the MiniROCKET algorithm decreases in performance with the added synthetic data. This result further indicates that the deterministic nature of MiniROCKET causes confusion to the model when ”unnatural” samples are introduced into the dataset. A better GAN producing more realistic samples is therefore necessary. The InceptionTime algorithm does however still maintain a good performance with the added synthetic samples. In contrast to the simulated experiments, the F1 score does not increase with the synthetic samples. It is hard to identify if any particular classes were harder to identify than others. From figure 6.15 it can maybe be seen that the model have an easier time identifying the classes with more samples from the true dataset. This indicates that the generated samples for the classes with few examples do not benefit the training of the classifier. It can also be seen that the model have a hard time to differentiate between class 0, 1, and 2. The differentiation between class 0 and 2 is expected to be hard as class 0 is the generator and class 2 is the generator bearing. It could therefore be expected that some of the features for one of the failures look similar to the other. It was not possible to conclude that the training and validation performance were correlated by looking at the F1 score on the training and validation set over the epochs which further indicate the inconsistency in the model. The generative model was developed with the AWR dataset as a validation set, meaning that the architecture of the generative model might not be optimal for the real-world dataset. This has maybe influenced the quality of the generated samples and thereby the performance of the classification model.

The results indicate that the InceptionTime algorithm is capable of adjusting in accordance with the variation of quality in the synthetic data. It is hypothesised that this might be due to the similarity between the GAN and InceptionTime. Both models are based on many convolutional layers that try to extract features from the input. It could therefore be argued that some of the layers in the two models learn similar features. Especially discriminatorY whose job is to both tell if the time series is true or not, but also if the outputted time series meets the condition that is put on the generator. This is in principle a classifier just like the InceptionTime. The other deep learning method presented in chapter 2 were highly specialized to a single type of problem [Pan+17; Zha+17]. The InceptionTime algorithm has shown to be very diverse across many different types of datasets and are therefore suitable in an operational setting.

The quality of the synthetic data did not seem to be as realistic such that the classifier could benefit from it. This is partly due to the size of the dataset and the sampling technique. Even though the aim with the use of GAN was to create robust models, even with little training data, a GAN must

encounter a certain amount of samples to be able to capture the underlying distribution of the data. When comparing the results with the GAN study on wind turbines presented in section 2.3 [Liu+19] it is clear that more work is needed to get the benefit from the GANs. Some key differences are however presented that are relevant to discuss. In the study the GAN was helped considerably by actually manually creating low-frequency data that the GAN should then fine-tune. This means that actually new samples were introduced into the dataset and not filling in the distribution as was proposed with this project use of GANs. This method would not be desirable as it would introduce a significant amount of manual work from domain experts which was exactly what is trying to be avoided. Secondly, the problem they worked on was a detection problem where the classification model only had to identify if a fault had occurred or not. This made the GANs job considerable easier as it did not have to learn the same amount of conditions as the one presented in this experiment.

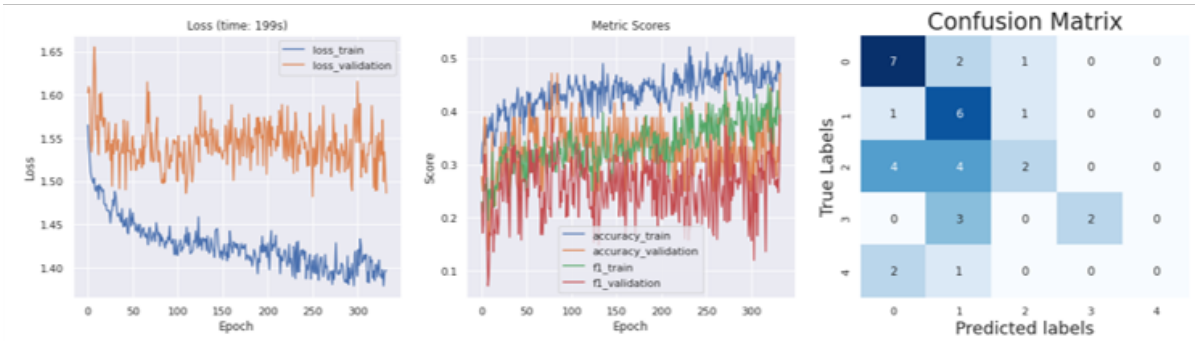

Figure 6.15: The loss curve (left), performance metrics (middle) and confusion matrix (right) for the InceptionTime model trained on the real-world data + synthetic wind turbine dataset.

Overfitting was not reduced with the synthetic data which was also stated in the simulated experiments. This is possibly due to the fact that the additional synthetic data expands the range of the data distribution. When a sample is added that does not fit into the original data distribution because of the GAN's imperfections, the model has to take these samples into account. When validating the model, the samples from outside the true distribution pushes the model in an incorrect direction. With a better GAN, it would be expected that the addition of samples within the true distribution would smooth out the distribution, as shown in figure 6.14. Because of this fact it is not recommended to use GANs in an operational setting as the uncertainty that the model would do more harm than good is too high. More work, therefore, needs to be done in the research community. It is here relevant to mention that time series has not been the main study research subject within deep learning. Focus in the community has mainly been on image data and natural language processing. It is however expected that more focus will be put upon the analysis of time series data within the coming years [Ism+19]. This will hopefully mean that both new classification algorithms will be developed but also more effort will be put into developing generative models that can be used to generate realistic time series.

The results from the conducted experiments have now been presented. It was shown that it was possible to achieve state of the art results on open-source datasets using the most recent classification algorithms. Especially the deep learning model Inception Time performed consistently well. These learnings were applied to a "real-world" dataset where promising results were achieved. The use of GANs to generate realistic synthetic data have been investigated. The GAN experiment showed that it was possible to generate realistic synthetic data from the AWR dataset. A series of limitations with using a GAN was observed. It was concluded that the GANs ability to create realistic time series for a specific class was highly limited. When creating a model for each of the classes in the real-world dataset the size and sampling of the dataset limited the quality of the generated synthetic data. Furthermore

did the lack of computational resources limit the number of epochs a GAN model could fit the input data.

CHAPTER 7

Conclusion and Future Work

To what extent is it possible to create a model for fault diagnostics of industrial assets by classifying time series?

The simulated experiments showed that the proposed state of the art classification models combined with automatic hyperparameter tuning managed to achieve benchmark results when comparing them with results presented in the literature. In the real-world experiments, the classification models to a long extent were able to differentiate in what component a failure had occurred. However, the number of failures logged did restrict the model in capturing the exact pattern associated with a failure leading to wrong classifications. Comparing the methods to similar problems from the medical industry the need for a model that combines structured data from sensors with unstructured observations is needed to get better results.

Could the performance of a classification model be improved by enriching the training dataset with synthetic data?

A variation of the uTSGAN architecture has been developed that produces realistic time series samples. The experiments on the AWR dataset showed that training the InceptionTime algorithm solely on this synthetic generated data and validated on 80% of the real dataset resulted in benchmark performance. These results indicate that the GAN managed to capture the general patterns of the underlying distribution of the dataset. When training on both the 20% and synthetic samples, the model achieved a better performance than only training on the 20%. This indicates that it is possible to improve the performance of the classification model by doing advanced data augmentation. More experiments do however need to be performed to validate the statistical significance of the results. The limitation of the GAN was however clear when training a GAN on the real-world dataset. Here the additional synthetic data did not improve the performance of the classification model. The GAN models need to almost perfectly fit the underlying distribution of the data to benefit the classification model. Therefore it is not recommended to use advanced data augmentation using GANs in an operational setting as too much uncertainty is present in the quality of the GAN. More research must be conducted for multivariate time series generation using generative models to reduce this uncertainty.

How should a system be implemented to allow for dynamic prepossessing, training, and continuous improvement?

A pipeline was implemented that allowed for automatic prepossessing, hyperparameter tuning, predictions and continuous training. The pipeline was implemented so both traditional machine learning models and deep learning models could be trained using a grid search. The results obtained with the method matches the benchmark results presented in relevant studies. It is therefore concluded that the method could be used in an operational setting.

7.1 Future work

While good results were achieved for the classification models and the generative model in the simulated experiment, there is still work to be done to further validate the classification models and improve the generative model. First, cross-validation should be done for all the different classification models, followed by a t-test to statistically compare if one is better than the others. A more sophisticated hyperparameter optimization method would also be an improvement to make the optimal hyperparameters more fine-tuned. This would also benefit the retraining time of the pipeline in an operational context. Therefore, hyperparameter optimization tools like *Optuna* [Aki+19] should be explored.

Regarding the generative model, it would be interesting to see the difference in performance, if the $WGAN_X$ in the uTSGAN was also conditioned to make the synthetic spectrograms more directly related to the synthetic time series. As discussed, it would also be worth experimenting with the sampling of the batches, such as sampling only a few classes in each batch to ensure the model can estimate the distribution in each class correctly and allow for a more complex model. A more rigorously comparison of the original uTSGAN with the second iteration of the uTSGAN should also be done to validate the impact of the "reinvented" way of generating a time series from a spectrogram. Again, this should also be cross-validated. The generative model should also be validated against other datasets to investigate if the noise it learned in the AWR dataset is the same or different from other datasets.

In the operational context, it would be interesting to see if the performance of the models would increase as more data would become available. In general, there should be experimented with the sampling of the dataset to cover the entire period of a failure without overlapping with a normal state. This should then be combined with simple data augmentation to provide the classification models with the best possible data foundation. The scope of this problem has been on the diagnostic of failures with a narrow focus on the sensor data from the assets. There are both the detection model that needs to be implemented along with a system for guiding the technicians to solve the failure the diagnostic model has suggested. Before this, the operational data must be combined with the predicted outputs from the classification models presented in this thesis. This has been described as a strategy decision model throughout the thesis. One kind of model that has been discussed to be suitable for this task is a reinforcement learning model. These types of models are at the end of their maturity phase and are starting to be deployed in operational settings. It could therefore be interesting to investigate the compatibility of these models with the guided troubleshooting problem.

Bibliography

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." In: *International conference on machine learning*. PMLR. 2017, pages 214–223.

[Aga21] Rahul Agarwal. *The 5 Classification Evaluation metrics every Data Scientist must know*. `https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226`. Accessed: 2021-05-05. 2021.

[Aki+19] Takuya Akiba et al. "Optuna: A Next-generation Hyperparameter Optimization Framework." In: *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.

[AZV09] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. "Spectrum-based multiple fault localization." In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2009, pages 88–99.

[BGV92] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. "A training algorithm for optimal margin classifiers." In: *Proceedings of the fifth annual workshop on Computational learning theory*. 1992, pages 144–152.

[Bui+13] Lars Buitinck et al. "API design for machine learning software: experiences from the scikit-learn project." In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pages 108–122.

[Chr+18] Maximilian Christ et al. "Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package)." In: *Neurocomputing* 307 (2018), pages 72–77.

[Cli+13] Lei Clifton et al. "Predictive monitoring of mobile patients by combining clinical observations with data from wearable sensors." In: *IEEE journal of biomedical and health informatics* 18.3 (2013), pages 722–730.

[DPW20] Angus Dempster, François Petitjean, and Geoffrey I Webb. "ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels." In: *Data Mining and Knowledge Discovery* 34.5 (2020), pages 1454–1495.

[DSH19] Colin Dixon, Corey T Schimpf, and Sherry H Hsi. "Beyond Trial & Error: Iteration-to-learn using computational papercrafts in a STEAM camp for girls." In: *2019 ASEE Annual Conference & Exposition*. 2019.

[DSW20] Angus Dempster, Daniel F Schmidt, and Geoffrey I Webb. "MINIROCKET: A Very Fast (Almost) Deterministic Transform for Time Series Classification." In: *arXiv preprint arXiv:2012.08791* (2020).

[ESK18] Amir Elmishali, Roni Stern, and Meir Kalech. "An artificial intelligence paradigm for troubleshooting software bugs." In: *Engineering Applications of Artificial Intelligence* 69 (2018), pages 147–156.

[Faw+20] Hassan Ismail Fawaz et al. "Inceptiontime: Finding alexnet for time series classification." In: *Data Mining and Knowledge Discovery* 34.6 (2020), pages 1936–1962.

[Gér19] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow*. O'Reilly Media, 2019.

[gGm21] ZeMA gGmbH. *Condition monitoring of hydraulic systems Data Set*. https://archive.ics.uci.edu/ml/datasets/Condition+monitoring+of+hydraulic+systems. Accessed: 2021-04-29. 2021.

[Gho+19] Sudipto Ghoshal et al. "An Integrated model-based Approach for FMECA Development for Smart Manufacturing Applications." In: *Annual Conference of the PHM Society*. Volume 11. 1. 2019.

[Goo+14] Ian J Goodfellow et al. "Generative adversarial networks." In: *arXiv preprint arXiv:1406.2661* (2014).

[Gul+17] Ishaan Gulrajani et al. "Improved training of wasserstein gans." In: *arXiv preprint arXiv:1704.00028* (2017).

[Gup+18] Abhirut Gupta et al. "Mining procedures from technical support documents." In: *arXiv preprint arXiv:1805.09780* (2018).

[Hay+00] Paul Hayton et al. "Support vector novelty detection applied to jet engine vibration spectra." In: *NIPS*. Citeseer. 2000, pages 946–952.

[Heu+17] Martin Heusel et al. "Gans trained by a two time-scale update rule converge to a local nash equilibrium." In: *Advances in neural information processing systems* 30 (2017).

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *arXiv:1502.03167* (2015).

[Ise05] Rolf Isermann. *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2005.

[Ise97] Rolf Isermann. "Supervision, fault-detection and fault-diagnosis methods—an introduction." In: *Control engineering practice* 5.5 (1997), pages 639–652.

[Ism+19] Hassan Ismail Fawaz et al. "InceptionTime: Finding AlexNet for Time Series Classification." In: *arXiv e-prints*, arXiv:1909.04939 (September 2019), arXiv:1909.04939. arXiv: 1909.04939 [cs.LG].

[J W21] UCR J Wang. *ArticularyWordRecognition*. https://sites.google.com/site/dtwadaptive/home. Accessed: 2021-04-29. 2021.

[Kag21] UEA Kaggle. *PLAsTiCC Astronomical Classification*. https://www.kaggle.com/c/PLAsTiCC-2018. Accessed: 2021-04-29. 2021.

[Kon01] Igor Kononenko. "Machine learning for medical diagnosis: history, state of the art and perspective." In: *Artificial Intelligence in medicine* 23.1 (2001), pages 89–109.

[Lab04] Ashraf W Labib. "A decision analysis model for maintenance policy selection using a CMMS." In: *Journal of Quality in Maintenance Engineering* (2004).

[Liu+18] Jinhai Liu et al. "A small-sample wind turbine fault detection method with synthetic fault data using generative adversarial nets." In: *IEEE Transactions on Industrial Informatics* 15.7 (2018), pages 3877–3888.

[Liu+19] J. Liu et al. "A Small-Sample Wind Turbine Fault Detection Method With Synthetic Fault Data Using Generative Adversarial Nets." In: *IEEE Transactions on Industrial Informatics* 15.7 (2019), pages 3877–3888. DOI: 10.1109/TII.2018.2885365.

[Lön+21] Markus Löning et al. *alan-turing-institute/sktime: v0.6.1*. Version v0.6.1. May 2021. DOI: 10.5281/zenodo.4763769. URL: https://doi.org/10.5281/zenodo.4763769.

[Lop+16] Isabel Lopes et al. "Requirements specification of a computerized maintenance management system–a case study." In: *Procedia Cirp* 52 (2016), pages 268–273.

[Mis20] Abhishek Mishra. *Time Series Similarity Using Dynamic Time Warping -Explained*. https://medium.com/walmartglobaltech/time-series-similarity-using-dynamic-time-warping-explained-9d09119e48ec. Accessed: 2021-06-04. 2020.

[MO14] Mehdi Mirza and Simon Osindero. "Conditional generative adversarial nets." In: *arXiv preprint arXiv:1411.1784* (2014).

[Nie13a] Olli Niemitalo. *Blackman window function*. https://en.wikipedia.org/wiki/Window_function#/media/File:Window_function_and_its_Fourier_transform_-_Blackman_(n_=_0...N).svg. Accessed: 2021-06-03. 2013.

[Nie13b] Olli Niemitalo. *Hann window function*. https://en.wikipedia.org/wiki/Window_function#/media/File:Window_function_and_its_Fourier_transform_%E2%80%93_Hann_(n_=_0...N).svg. Accessed: 2021-06-03. 2013.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[Ott12] Thorsten Jørgen Ottosen. "Solutions and Heuristics for Troubleshooting with Dependent Actions and Conditional Costs." Dansk. PhD thesis. March 2012.

[Pan+17] Jun Pan et al. "LiftingNet: A novel deep learning network with layerwise feature learning from noisy mechanical data for fault classification." In: *IEEE Transactions on Industrial Electronics* 65.6 (2017), pages 4973–4982.

[Pas+19] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library." In: *arXiv preprint arXiv:1912.01703* (2019).

[Ped+21] F. Pedregosa et al. *Scikit-learn: train_test_split*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html. Accessed: 2021-06-06. 2021.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." In: *arXiv preprint arXiv:1511.06434* (2015).

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), page 386.

[Rui+20] Alejandro Pasos Ruiz et al. "The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances." In: *Data Mining and Knowledge Discovery* (2020), pages 1–49.

[San+18] Shibani Santurkar et al. "How Does Batch Normalization Help Optimization?" In: *arXiv:1805.11604* (2018).

[Sha+94] M. Shakeri et al. "Near-optimal sequential testing algorithms for multiple fault isolation." In: *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*. Volume 2. 1994, 1908–1913 vol.2. DOI: 10.1109/ICSMC.1994.400130.

[SS20] Kaleb E Smith and Anthony O Smith. "Conditional GAN for timeseries generation." In: *arXiv preprint arXiv:2006.16477* (2020).

[SS21] Kaleb E Smith and Anthony O Smith. "A Spectral Enabled GAN for Time Series Data Generation." In: *arXiv preprint arXiv:2103.01904* (2021).

[Sze+15] Christian Szegedy et al. "Going deeper with convolutions." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pages 1–9.

[Tav+13] PJ Tavner et al. "Study of weather and location effects on wind turbine failure rates." In: *Wind energy* 16.2 (2013), pages 175–187.

[TRY21] TRYQA. *What is Test Strategy? Types of strategies with examples*. http://tryqa.com/what-is-test-strategy-types-of-strategies-with-examples/. Accessed: 2021-02-22. 2021.

[tsl17] tslearn. *tslearn.metrics.dtw_path*. https://tslearn.readthedocs.io/en/latest/gen_modules/metrics/tslearn.metrics.dtw_path.html#tslearn.metrics.dtw_path. Accessed: 2021-06-04. 2017.

[WHV16] Michael Wienker, Ken Henderson, and Jacques Volkerts. “The computerized maintenance management system an essential tool for world class maintenance.” In: *Procedia Engineering* 138 (2016), pages 413–420.

[Wik21] Wikipedia contributors. *Colors of noise — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Colors_of_noise&oldid=1022510242`. [Online; accessed 9-July-2021]. 2021.

[Xan11] XantaCross. *Dynamic Time Warping example*. `https://commons.wikimedia.org/wiki/File:Euclidean_vs_DTW.jpg#/media/File:Euclidean_vs_DTW.jpg`. Accessed: 2021-06-04. 2011.

[Zha+17] Ran Zhang et al. “Fault diagnosis from raw sensor data using deep neural networks considering temporal coherence.” In: *Sensors* 17.3 (2017), page 549.

APPENDIX A

Appendix - Code Listings

A.1 Pipeline Implementation

```
class Pipeline:

    def __init__(self, name:str, steps:list):
        self.name = name
        self.step_names, self.steps = zip(*steps)
        self.folder_path = 'drive/MyDrive/Speciale_2021/Kode/Saved_Pipelines' # Default save path
        self.fitting_time = -1


    def get_step_by_name(self, name):
        idx = self.step_names.index(name)
        return self.steps[idx]


    def fit(self, X_train, y_train, validation_set=None):
        time_start = round(time.time()) # Start time in seconds

        # If validation set is specified run train and validation data through pipeline and provide to PipelineModel
        if validation_set:
            X_validation, y_validation = validation_set

            for step in self.steps:
                if isinstance(step, PipelineTransformer):
                    step.fit(X_train, y_train)
                    X_train, y_train = step.transform(X_train, y_train)
                    if not isinstance(step, Synthesizer):
                        X_validation, y_validation = step.transform(X_validation, y_validation)

                elif isinstance(step, Pipeline):
                    X_train, y_train, X_validation, y_validation = step.fit(X_train, y_train, validation_set=(X_validation, y_validation))

                elif isinstance(step, (PipelineDLModel, PipelineMLModel)):
                    step.fit(X_train, y_train, validation_set=(X_validation, y_validation))

            time_end = round(time.time()) # End time in seconds
            self.fitting_time = time_end - time_start

            return X_train, y_train, X_validation, y_validation

        # If validation set is not specified, only run train data through pipeline
        else:
            for step in self.steps:
                if isinstance(step, PipelineTransformer):
                    step.fit(X_train, y_train)
                    X_train, y_train = step.transform(X_train, y_train)

                elif isinstance(step, Pipeline):
```

```
                    X_train, y_train = step.fit(X_train, y_train)

                elif isinstance(step, (PipelineDLModel, PipelineMLModel)):
                    step.fit(X_train, y_train)

            time_end = round(time.time()) # End time in seconds
            self.fitting_time = time_end - time_start

            return X_train, y_train


    def transform(self, X, y):
        # Transform
        for step in self.steps[:-1]:
            X, y = step.transform(X, y)

        last_step = self.steps[-1]
        if isinstance(last_step, (Pipeline, PipelineTransformer)):
             X, y = last_step.transform(X, y)

        return X, y


    def predict(self, X):
        # Predict last step, transform all steps before
        for step in self.steps[:-1]:
            if not isinstance(step, Synthesizer):
                X, _ = step.transform(X, None)

        last_step = self.steps[-1]
        y_ = last_step.predict(X)

        return y_


    def set_parameter(self, key, value):
        name, key = key.split('__', 1)
        step = self.get_step_by_name(name)
        step.set_parameter(key, value)


    def save(self, save_path:str=''):

        save_path = f'{self.folder_path}/{self.name}' if not save_path else f'{save_path}/{
            self.name}'

        # Create folder for pipeline if not exist
        if not os.path.exists(save_path):
            os.makedirs(save_path)

        # Save all pipeline steps to folder
        for step in self.steps:
            step.save(save_path)

        # Save pipeline info
        values = {key : value for key, value in self.__dict__.items() if not '__' in key and
            not callable(key)}
        with open(f'{save_path}.pickle', 'wb') as handle:
            pickle.dump(values, handle, protocol=pickle.HIGHEST_PROTOCOL)


    def load(self, load_path:str=''):

        load_path = f'{self.folder_path}/{self.name}' if not load_path else f'{load_path}/{
            self.name}'

```

```
        # Load all steps in pipeline from folder
        for step in self.steps:
            step.load(load_path)

        # Load pipeline info
        with open(f'{load_path}.pickle', 'rb') as handle:
            values = pickle.load(handle)
            for key, value in values.items():
                setattr(self, key, value)
```

Listing A.1: The full implementation of the *Pipeline* class.

A.2 PipelineStep Implementation

```
class PipelineStep:

    def __init__(self, name:str):
        self.name = name


    def set_parameter(self, key, value):
        setattr(self, key, value)


    def save(self, folder_path:str):
        values = {key : value for key, value in self.__dict__.items() if not '__' in key and
            not callable(key)}
        with open(f'{folder_path}/{self.name}.pickle', 'wb') as handle:
            pickle.dump(values, handle, protocol=pickle.HIGHEST_PROTOCOL)


    def load(self, folder_path:str):
        with open(f'{folder_path}/{self.name}.pickle', 'rb') as handle:
            values = pickle.load(handle)
            for key, value in values.items():
                setattr(self, key, value)
```

Listing A.2: The full implementation of the *PipelineStep* class.

A.3 PipelineMLModel Implementation

```
class PipelineMLModel(PipelineStep):

    def __init__(self, name:str, model):
        super().__init__(name)
        self.__model = model
        self.history = {
            'accuracy_train' : [0],
            'precision_train' : [0],
            'recall_train' : [0],
            'f1_train' : [0],
            'accuracy_validation' : [0],
            'precision_validation' : [0],
            'recall_validation' : [0],
            'f1_validation' : [0]
        }
        self.best_folder_path = ''
        self.best_score = 0
        self.best_model = None
```

```
19
20
21      def init_objects(self):
22          # Init model and optimizer if keyword arguments are avaialble
23          if hasattr(self, 'model_kw'):
24              try:
25                  self.__model = self.__model(**self.model_kw, probability=True)
26              except:
27                  self.__model = self.__model(**self.model_kw)
28
29
30      def fit(self, X_train, y_train, validation_set=None):
31          # Init objects
32          self.init_objects()
33
34          # Fit
35          self.__model.fit(X_train, y_train)
36
37          # Calculate metrics for training set
38          y_hat_train = self.__model.predict(X_train)
39          accuracy_train, precision_train, recall_train, f1_train = self.calculate_metrics(
                y_hat_train, y_train)
40
41          # Set history value
42          self.history['accuracy_train'] = [accuracy_train]
43          self.history['precision_train'] = [precision_train]
44          self.history['recall_train'] = [recall_train]
45          self.history['f1_train'] = [f1_train]
46
47
48          if validation_set:
49              X_validation, y_validation = validation_set
50
51              # Predict with validation set and set history with f1 score
52              y_hat_validation = self.__model.predict(X_validation)
53              accuracy_validation, precision_validation, recall_validation, f1_validation =
                    self.calculate_metrics(y_hat_validation, y_validation)
54
55              # Set history value
56              self.history['accuracy_validation'] = [accuracy_validation]
57              self.history['precision_validation'] = [precision_validation]
58              self.history['recall_validation'] = [recall_validation]
59              self.history['f1_validation'] = [f1_validation]
60
61              if self.best_folder_path:
62                  self.best_score = f1_validation
63                  self.save(self.best_folder_path)
64
65
66      def predict(self, X):
67          try:
68              y_hat = self.__model.predict_proba(X)
69          except:
70              y_hat = self.__model.predict(X)
71
72          return y_hat
73
74
75      def calculate_metrics(self, y_pred, y_true):
76          report = classification_report(y_true, y_pred, output_dict=True, zero_division=0)
77          report = report.get('macro avg', report)
78          accuracy = accuracy_score(y_true, y_pred)
79
80          return accuracy, report['precision'], report['recall'], report['f1-score']
81
82
```

```
    def save(self, folder_path:str):

        # Save all variables
        super().save(folder_path)

        # Make directory for best model
        if not os.path.exists(self.best_folder_path):
            os.makedirs(self.best_folder_path)

        # Save model
        f = open(f'{folder_path}/{self.name}.pt', 'wb')
        pickle.dump(self.__model, f)


    def load(self, folder_path:str):
        # Load all variables
        super().load(folder_path)

        self.init_objects()

        # Load model
        f = open(f'{folder_path}/{self.name}.pt', 'rb')
        self.__model = pickle.load(f)
```

Listing A.3: The full implementation of the *PipelineMLModel* class.

A.4 PipelineDLModel Implementation

```
class PipelineDLModel(PipelineStep):

    def __init__(self, name:str, model=None, criterion=None, optimizer=None, epochs=100,
        batch_size=16, shuffle=True, patience=50):
        super().__init__(name)
        self.__device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.__model = model
        self.__criterion = criterion
        self.__optimizer = optimizer
        self.__epochs = epochs
        self.current_epoch = 0
        self.batch_size = batch_size
        self.shuffle = shuffle
        self.patience = patience
        self.history = {
            'loss_train' : [],
            'accuracy_train' : [],
            'precision_train' : [],
            'recall_train' : [],
            'f1_train' : [],
            'loss_validation' : [],
            'accuracy_validation' : [],
            'precision_validation' : [],
            'recall_validation' : [],
            'f1_validation' : []
        }
        self.best_folder_path = ''
        self.best_score = 0
        self.best_model = None
        self.best_epoch = -1


    def init_objects(self):
        # Init model and optimizer if keyword arguments are avaialble
        if hasattr(self, 'model_kw'):
```

```
            self.__model = self.__model(**self.model_kw)
        self.__model.to(self.__device)
        if hasattr(self, 'optimizer_kw'):
            self.__optimizer = self.__optimizer(self.__model.parameters(), **self.
                optimizer_kw)


    def fit(self, X_train, y_train, validation_set=None):

        # Create the dataset and dataloader
        dataset = PytorchTimeSeriesDataset(X_train, y_train)
        dataloader = DataLoader(dataset, batch_size=self.batch_size, shuffle=self.shuffle)

        # If validation set is specified, set X_validation and y_validation.
        if validation_set:
            X_validation, y_validation = validation_set
            X_validation = torch.tensor(X_validation.astype(np.float32)).to(self.__device)
            y_validation = torch.tensor(y_validation.astype(np.float32)).to(self.__device)

        # Init objects
        self.init_objects()

        # Set model to training mode
        self.__model.train()

        # Run training loop
        for epoch in range(self.current_epoch, self.__epochs):

            running_loss = 0.0
            running_accuracy = 0.0
            running_precision = 0.0
            running_recall = 0.0
            running_f1 = 0.0

            for i, batch in enumerate(dataloader):
                inputs, labels = batch
                inputs, labels = inputs.to(self.__device), labels.to(self.__device)
                # Zero the parameter gradients
                self.__optimizer.zero_grad()

                # Forward
                outputs = self.__model(inputs)

                # Calculate loss
                loss = self.__criterion(outputs,  labels.argmax(dim=1).long())

                # Backward + optimize
                loss.backward()
                self.__optimizer.step()

                # Calculate metrics
                accuracy, precision, recall, f1 = self.calculate_metrics(outputs, labels)
                running_loss += loss.item()
                running_accuracy += accuracy
                running_precision += precision
                running_recall += recall
                running_f1 += f1

            # Average the metrics over all batches
            loss_train = running_loss / (i+1)
            accuracy_train = running_accuracy / (i+1)
            precision_train = running_precision / (i+1)
            recall_train = running_recall / (i+1)
            f1_train = running_f1 / (i+1)

            # Update history
```

```
            self.history['loss_train'].append(loss_train)
            self.history['accuracy_train'].append(accuracy_train)
            self.history['precision_train'].append(precision_train)
            self.history['recall_train'].append(recall_train)
            self.history['f1_train'].append(f1_train)

            # Also calculate for validation set
            if validation_set:
                with torch.no_grad():
                    # self.__model.eval()
                    y_validation_pred = self.__model(X_validation)
                    loss_validation = self.__criterion(y_validation_pred, y_validation.argmax
                        (dim=1).long())
                    accuracy_validation, precision_validation, recall_validation,
                        f1_validation = self.calculate_metrics(y_validation_pred,
                        y_validation)
                    self.history['loss_validation'].append(loss_validation.item())
                    self.history['accuracy_validation'].append(accuracy_validation)
                    self.history['precision_validation'].append(precision_validation)
                    self.history['recall_validation'].append(recall_validation)
                    self.history['f1_validation'].append(f1_validation)

                    # If validation score is better than best, save that model
                    if f1_validation > self.best_score:
                        self.best_epoch = epoch
                        self.best_score = f1_validation
                        self.save(self.best_folder_path)
                        self.confusion_matrix = confusion_matrix(y_validation.cpu().squeeze()
                            .argmax(dim=1), y_validation_pred.cpu().squeeze().argmax(dim=1))

                    # Early stopping criteria met. Break training loop
                    elif epoch > self.best_epoch + self.patience:
                        break


            # Plot for every 500th epoch
            if self.current_epoch % 25 == 0:
                if validation_set:
                    self.plot(validation_set=(y_validation_pred, y_validation))
                else:
                    self.plot()

            self.current_epoch = epoch + 1

        # Plot after full training
        if validation_set:
            self.plot(validation_set=(y_validation_pred, y_validation))
        else:
            self.plot()


    def calculate_metrics(self, y_pred, y_true):

        y_pred = y_pred.cpu().squeeze().argmax(dim=1)
        y_true = y_true.cpu().squeeze().argmax(dim=1)

        report = classification_report(y_true, y_pred, output_dict=True, labels=torch.unique(
            torch.cat((y_true, y_pred), dim=0)), zero_division=0)
        report = report.get('macro avg', report)
        accuracy = accuracy_score(y_true, y_pred)

        return accuracy, report['precision'], report['recall'], report['f1-score']


    def plot(self, validation_set=None):

```

```
        clear_output(wait=True)
        sns.set_theme()
        plt.figure(figsize=(20,10))

        # Plot loss
        plt.subplot(1,2,1)
        plt.plot(range(len(self.history['loss_train'])), self.history['loss_train'], label='
            Train loss')
        if 'loss_validation' in self.history:
            plt.plot(range(len(self.history['loss_validation'])), self.history['
                loss_validation'], label='Validation loss')
        plt.legend()
        plt.title('Loss', fontsize=24)
        plt.xlabel('Epoch', fontsize=18)
        plt.ylabel('Loss', fontsize=18)

        # Plot f1 score
        plt.subplot(1,2,2)
        plt.plot(range(len(self.history['f1_train'])), self.history['f1_train'], label='Train
             f1 score')
        if 'f1_validation' in self.history:
            plt.plot(range(len(self.history['f1_validation'])), self.history['f1_validation'
                ], label = 'Validation f1 score')
        plt.legend()
        plt.title('F1 Score', fontsize=24)
        plt.xlabel('Epoch', fontsize=18)
        plt.ylabel('F1 score', fontsize=18)
        plt.tight_layout()


        # Plot validation confusion matrix
        if validation_set:

            y_validation_pred = validation_set[0].cpu().squeeze().argmax(dim=1)
            y_validation = validation_set[1].cpu().squeeze().argmax(dim=1)

            fig = plt.figure(figsize=(14,14))
            ax = fig.add_subplot(111)

            cm = confusion_matrix(y_validation, y_validation_pred)

            sns.heatmap(cm, annot=True, cmap='Blues', cbar=False, fmt='d')
            ax.set_title('Confusion Matrix', fontsize=24)
            ax.set_xlabel('Predicted labels', fontsize=18)
            ax.set_ylabel('True Labels', fontsize=18)


        plt.tight_layout()
        plt.show()


    def predict(self, X):
        # Set the model to evaluation mode
        self.__model.eval()
        X = X.to(self.__device)

        # Predict for the input batch
        y_pred = self.__model(torch.tensor(X.astype(np.float32)))

        return y_pred


    def save(self, folder_path:str):

        # Save all variables
        super().save(folder_path)
```

```
223
224         # Make directory for best model
225         if not os.path.exists(self.best_folder_path):
226             os.makedirs(self.best_folder_path)
227
228         # Save model
229         f = open(f'{folder_path}/{self.name}.pt', 'wb')
230         torch.save(self.__model.state_dict(), f)
231
232
233     def load(self, folder_path:str):
234         # Load all variables
235         super().load(folder_path)
236
237         self.init_objects()
238
239         # Load model
240         self.__model.load_state_dict(torch.load(f'{folder_path}/{self.name}.pt'))
```

Listing A.4: The full implementation of the *PipelineDLModel* class.

A.5 uTSGAN Implementation

```
1  class GeneratorX(nn.Module):
2
3      def __init__(self, latent_size, channels):
4          super(GeneratorX, self).__init__()
5
6          # Input shape (B, L, 1, 1)
7          self.net = nn.Sequential(
8              self._block(latent_size, channels*128, kernel_size=4, stride=1, padding=0),
9              self._block(channels*128, channels*64, kernel_size=4, stride=2, padding=1),
10             self._block(channels*64, channels*32, kernel_size=4, stride=2, padding=1),
11             self._block(channels*32, channels*16, kernel_size=4, stride=2, padding=1),
12             self._block(channels*16, channels*8, kernel_size=4, stride=2, padding=1),
13             nn.ConvTranspose2d(channels*8, channels, kernel_size=4, stride=2, padding=1),
14             nn.Tanh(),
15         )
16
17     def _block(self, in_channels, out_channels, kernel_size, stride, padding):
18         return nn.Sequential(
19             nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding, bias=
                   False),
20             nn.BatchNorm2d(out_channels),
21             nn.LeakyReLU(0.2),
22         )
23
24     def forward(self, X):
25         X = self.net(X)
26         return X
```

Listing A.5: The full implementation of generatorX.

```
1  class DiscriminatorX(nn.Module):
2
3      def __init__(self, channels):
4          super(DiscriminatorX, self).__init__()
5
6          # Input shape (B, C, 128, 128)
7          self.net = nn.Sequential(
8              self._block(channels, channels*2, kernel_size=4, stride=2, padding=1),
9              self._block(channels*2,channels*4, kernel_size=4, stride=2, padding=1),
```

```
            self._block(channels*4,channels*8, kernel_size=4, stride=2, padding=1),
            self._block(channels*8,channels*16, kernel_size=4, stride=2, padding=1),
            self._block(channels*16,channels*32, kernel_size=4, stride=2, padding=1),
            nn.Conv2d(channels*32, 1, kernel_size=4, stride=2, padding=0),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
        )

    def forward(self, X):
        rand_noise = torch.randn(X.shape).to(X.device)
        X = X + rand_noise
        X = self.net(X)
        return X
```

Listing A.6: The full implementation of discriminatorX.

```
class GeneratorY(nn.Module):

    def __init__(self, embed_size, channels, num_labels):
        super(GeneratorY, self).__init__()

        self.embed_size = embed_size
        self.embedding = nn.Embedding(num_labels, embed_size*128*128)

        # Input shape: (B, channels+embed, 128, 128)
        self.net = nn.Sequential(
            self._block(channels+embed_size, channels*2, kernel_size=3, stride=(2,1), padding
                =1),
            self._block(channels*2, channels*4, kernel_size=3, stride=(2,1), padding=1),
            self._block(channels*4, channels*8, kernel_size=3, stride=(2,1), padding=1),
            self._block(channels*8, channels*16, kernel_size=3, stride=(2,1), padding=1),
            self._block(channels*16, channels*32, kernel_size=3, stride=(2,1), padding=1),
            self._block(channels*32, channels*16, kernel_size=3, stride=(2,1), padding=(0,1)),
            nn.Flatten(start_dim=2),
        )

        self.upnet = nn.Sequential(
            nn.Conv1d(channels*16, channels, kernel_size=3, stride=1, padding=1, bias=False),
            nn.Tanh(),
            nn.Upsample(size=512, mode='linear', align_corners=False)
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
        )

    def forward(self, X, y):
        embedding = self.embedding(y.long()).view(y.shape[0], self.embed_size, 128, 128)
        X = torch.cat([X, embedding], dim=1)
        X = self.net(X)
        X = self.upnet(X)
        return X
```

Listing A.7: The full implementation of generatorY.

```
class DiscriminatorY(nn.Module):

```

```
    def __init__(self, embed_size, channels, num_labels):
        super(DiscriminatorY, self).__init__()

        self.embed_size = embed_size
        self.embedding = nn.Embedding(num_labels, 512*embed_size)

        # Input (B, C+E, 512)
        self.net = nn.Sequential(
            nn.Conv1d(channels+embed_size, channels*2, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2),
            self._block(channels*2, channels*4, kernel_size=3, stride=2, padding=1),
            self._block(channels*4, channels*8, kernel_size=3, stride=2, padding=1),
            self._block(channels*8, channels*16, kernel_size=3, stride=2, padding=1),
            self._block(channels*16, channels*32, kernel_size=3, stride=2, padding=1),
            self._block(channels*32, channels*64, kernel_size=3, stride=2, padding=1),
            self._block(channels*64, channels*128, kernel_size=3, stride=2, padding=1),
            nn.Conv1d(channels*128, 1, kernel_size=4, stride=2, padding=0),
        )

    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv1d(in_channels, out_channels, kernel_size, stride, padding, bias=False)
            nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
        )

    def forward(self, X, y):
        embed = self.embedding(y.long())
        embed = torch.reshape(embed, (X.shape[0], self.embed_size, X.shape[2]))
        X = torch.cat([X, embed], dim=1)
        X = self.net(X)
        return X
```

Listing A.8: The full implementation of discriminatorY.

```
def fit(self, X, y):
    # Create the dataset and dataloader
    dataset = PytorchTimeSeriesDatasetWithSpectogram(X, y)
    dataloader = DataLoader(dataset, batch_size=self.batch_size, shuffle=self.shuffle)
    z_to_plot = torch.randn(y.shape[1], self.__generatorX.latent_size, 1, 1).to(self.__device
        )
    true_idx_to_get = y.argmax(0)
    true_to_plot = dataset.__getitem__(true_idx_to_get)

    # Set model to training mode
    self.__generatorX.train()
    self.__generatorY.train()
    self.__discriminatorX.train()
    self.__discriminatorY.train()

    # Run training loop
    for epoch in range(self.current_epoch, self.__epochs):

        loss_generator_epoch = 0
        loss_generatorX_epoch = 0
        loss_generatorY_epoch = 0
        loss_discriminator_epoch = 0
        loss_discriminatorX_epoch = 0
        loss_discriminatorY_epoch = 0

        for i, batch in enumerate(dataloader):
            inputs, labels, spectrograms = batch
            inputs, labels, spectrograms = inputs.to(self.__device), labels.to(self.__device)
                , spectrograms.to(self.__device)

            loss_discriminator_iteration = 0
```

```
            loss_discriminatorX_iteration = 0
            loss_discriminatorY_iteration = 0

            for _ in range(self.discriminator_iterations):

                with torch.autograd.set_detect_anomaly(True):
                    # Create latent vector
                    z = torch.randn(labels.shape[0], self.__generatorX.latent_size, 1, 1).to(
                        self.__device)

                    # Train with discriminatorX with true spectrograms
                    output_discriminatorX_true = self.__discriminatorX(spectrograms).reshape
                        (-1)
                    # Train discriminatorX with fake spectrograms
                    g_X_z = self.__generatorX(z)
                    output_discriminatorX_fake = self.__discriminatorX(g_X_z).reshape(-1)
                    # Train discriminatorY with true time series
                    output_discriminatorY_true = self.__discriminatorY(inputs, labels).
                        reshape(-1)
                    # Train discriminatorY with fake Timeseries
                    g_Y_z = self.__generatorY(g_X_z, labels)
                    output_discriminatorY_fake = self.__discriminatorY(g_Y_z, labels).reshape
                        (-1)

                    # Calculate loss for discriminatorX
                    GP_X = self.gradient_penelty_spectogram(self.__discriminatorX,
                        spectrograms, g_X_z)
                    loss_discriminatorX = -(torch.mean(output_discriminatorX_true) - torch.
                        mean(output_discriminatorX_fake)) + self.lambda_penalty_x * GP_X
                    # Calculate loss for discriminatorY
                    GP_Y = self.gradient_penelty_timeseries(self.__discriminatorY, inputs,
                        g_Y_z, labels)
                    loss_discriminatorY = -(torch.mean(output_discriminatorY_true) - torch.
                        mean(output_discriminatorY_fake)) + self.lambda_penalty_y * GP_Y

                    # Calculate the unified loss
                    unified_loss_discriminator = (loss_discriminatorX + loss_discriminatorY)
                        /2
                    loss_discriminator_iteration += unified_loss_discriminator.item()
                    loss_discriminatorX_iteration += loss_discriminatorX.item()
                    loss_discriminatorY_iteration += loss_discriminatorY.item()

                    # Zero the gradients for the discriminators
                    self.__discriminatorX.zero_grad()
                    self.__discriminatorY.zero_grad()

                    # Backwards
                    unified_loss_discriminator.backward(retain_graph=True)

                    # Optimize Discriminator
                    self.__optimizer_discriminatorX.step()
                    self.__optimizer_discriminatorY.step()

            # Average loss in discriminator iterations
            loss_discriminator_iteration /= self.discriminator_iterations
            loss_discriminatorX_iteration /= self.discriminator_iterations
            loss_discriminatorY_iteration /= self.discriminator_iterations

            # Calculate loss generatorX
            genX_ = self.__discriminatorX(g_X_z)
            loss_generatorX = -torch.mean(genX_)

            # Calculate loss generatorY
            genY_ = self.__discriminatorY(g_Y_z, labels)
            loss_generatorY = -torch.mean(genY_)

```

```
            # Zero the  gradients for the generator
            self.__generatorX.zero_grad()
            self.__generatorY.zero_grad()

            # Backwards on both generators
            loss_generatorX.backward(retain_graph=True)
            loss_generatorY.backward()

            # Optimize generator
            self.__optimizer_generatorX.step()
            self.__optimizer_generatorY.step()

            loss_discriminator_epoch += loss_discriminator_iteration
            loss_discriminatorX_epoch += loss_discriminatorX_iteration
            loss_discriminatorY_epoch += loss_discriminatorY_iteration
            loss_generator_epoch += unified_loss_generator.item()
            loss_generatorX_epoch += loss_generatorX.item()
            loss_generatorY_epoch += loss_generatorY.item()

        # Logging to history for epoch average
        self.history['loss_discriminator'].append(loss_discriminator_epoch / (i+1))
        self.history['loss_discriminatorX'].append(loss_discriminatorX_epoch / (i+1))
        self.history['loss_discriminatorY'].append(loss_discriminatorY_epoch / (i+1))
        self.history['loss_generator'].append(loss_generator_epoch / (i+1))
        self.history['loss_generatorX'].append(loss_generatorX_epoch / (i+1))
        self.history['loss_generatorY'].append(loss_generatorY_epoch / (i+1))

        self.current_epoch += 1
```

Listing A.9: The full implementation of the fit method of the uTSGAN.

APPENDIX B

Appendix - Figures

B.1 AWR - Spectrograms for Sensors per Class

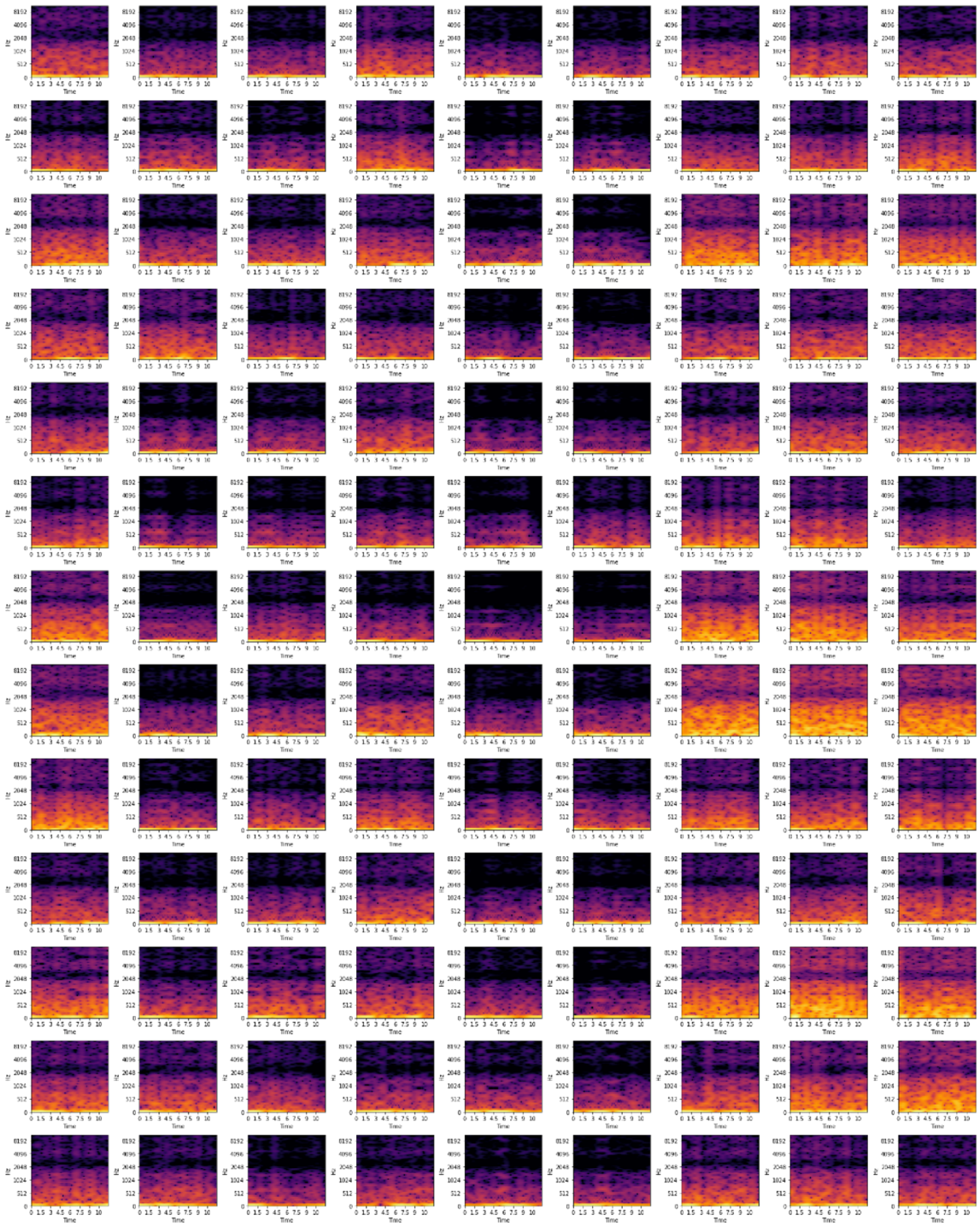

Figure B.1: The spectrograms for each of the sensors(columns) for the first 13 classes(rows) in the AWR dataset.

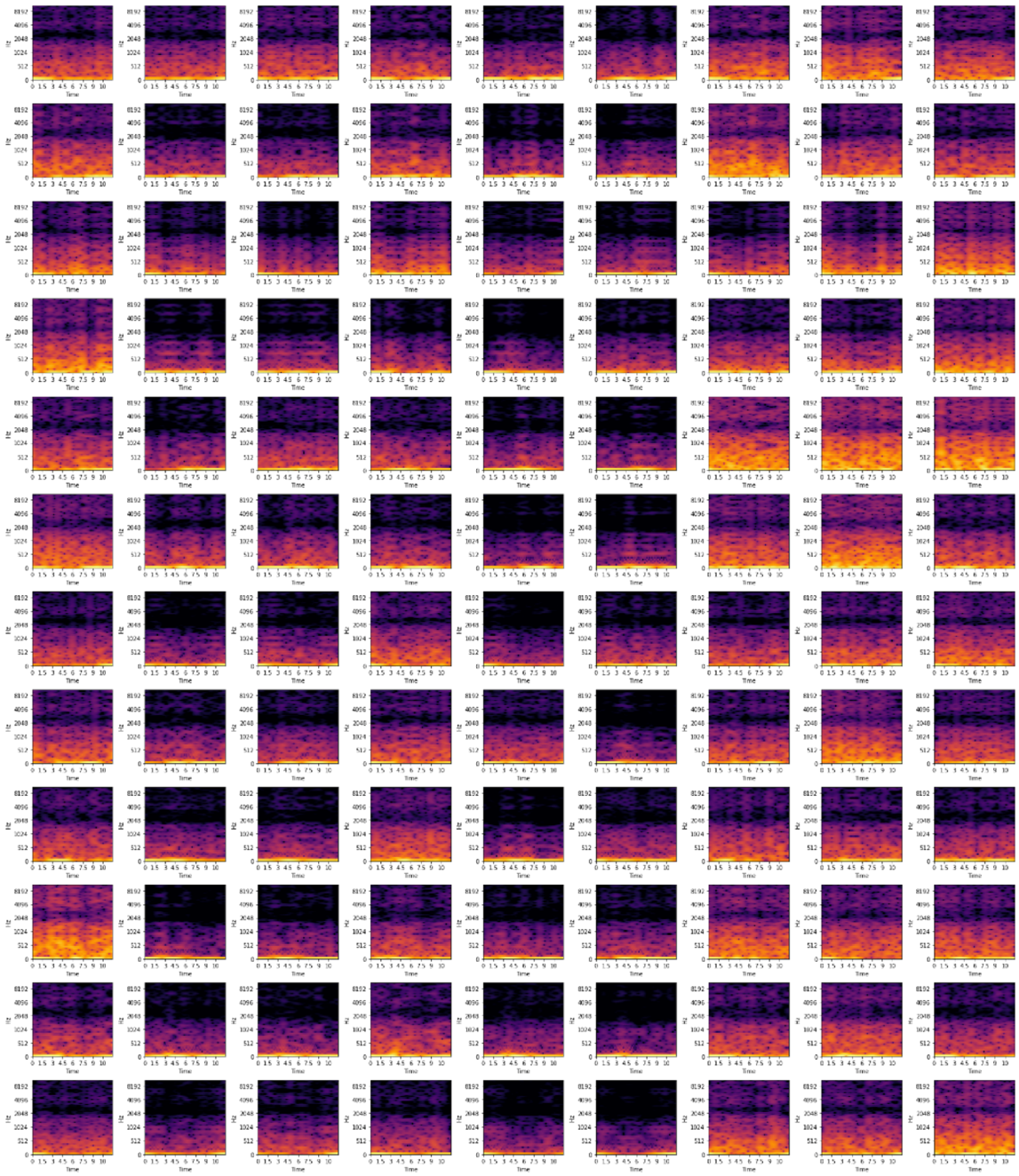

Figure B.2: The spectrograms for each of the sensors(columns) for the last 12 classes(rows) in the AWR dataset.

B.2 AWR - Time Series for Sensors per Class

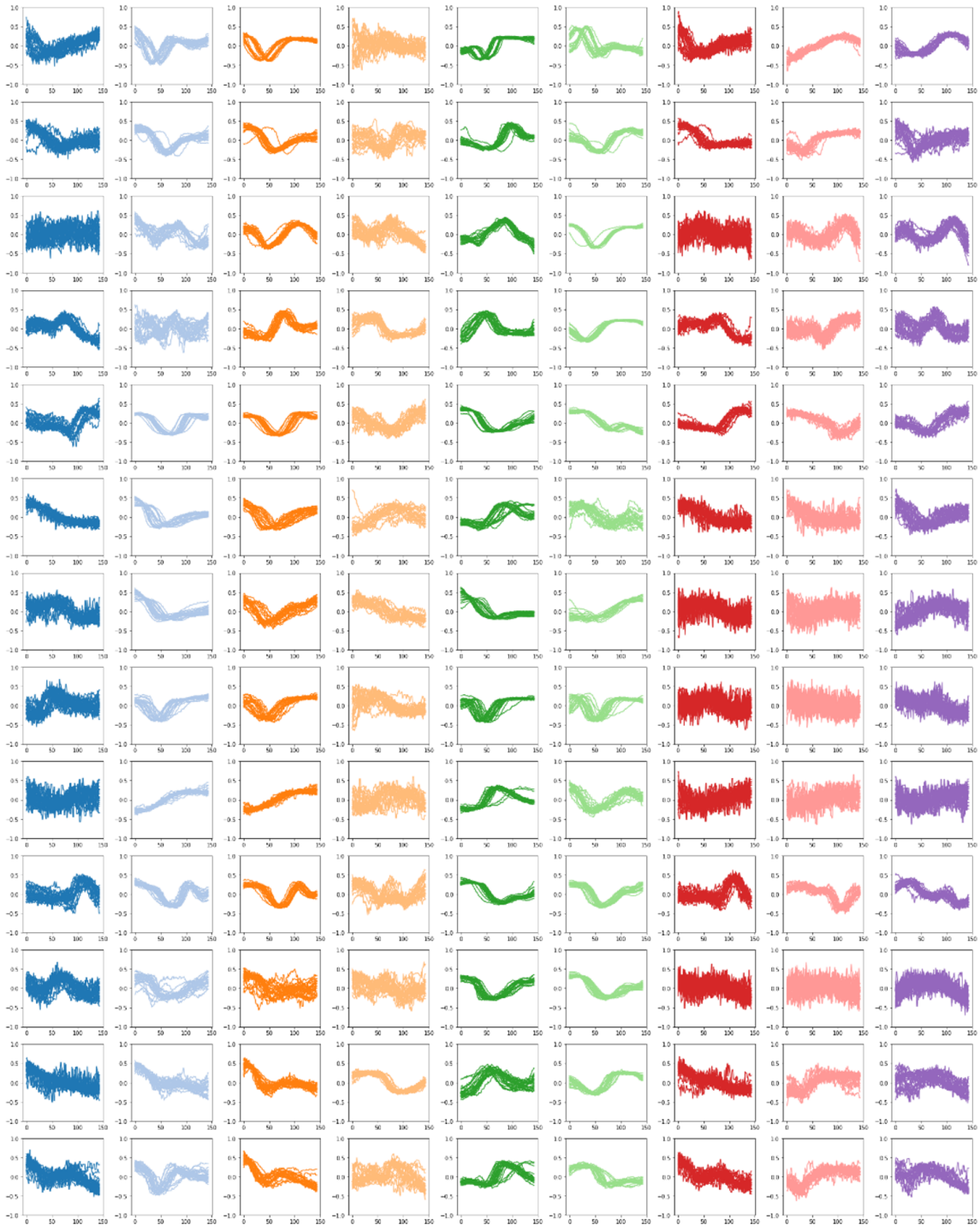

Figure B.3: All the time series data plotted for each of the sensors(columns) for the first 13 classes(rows) in the AWR dataset.

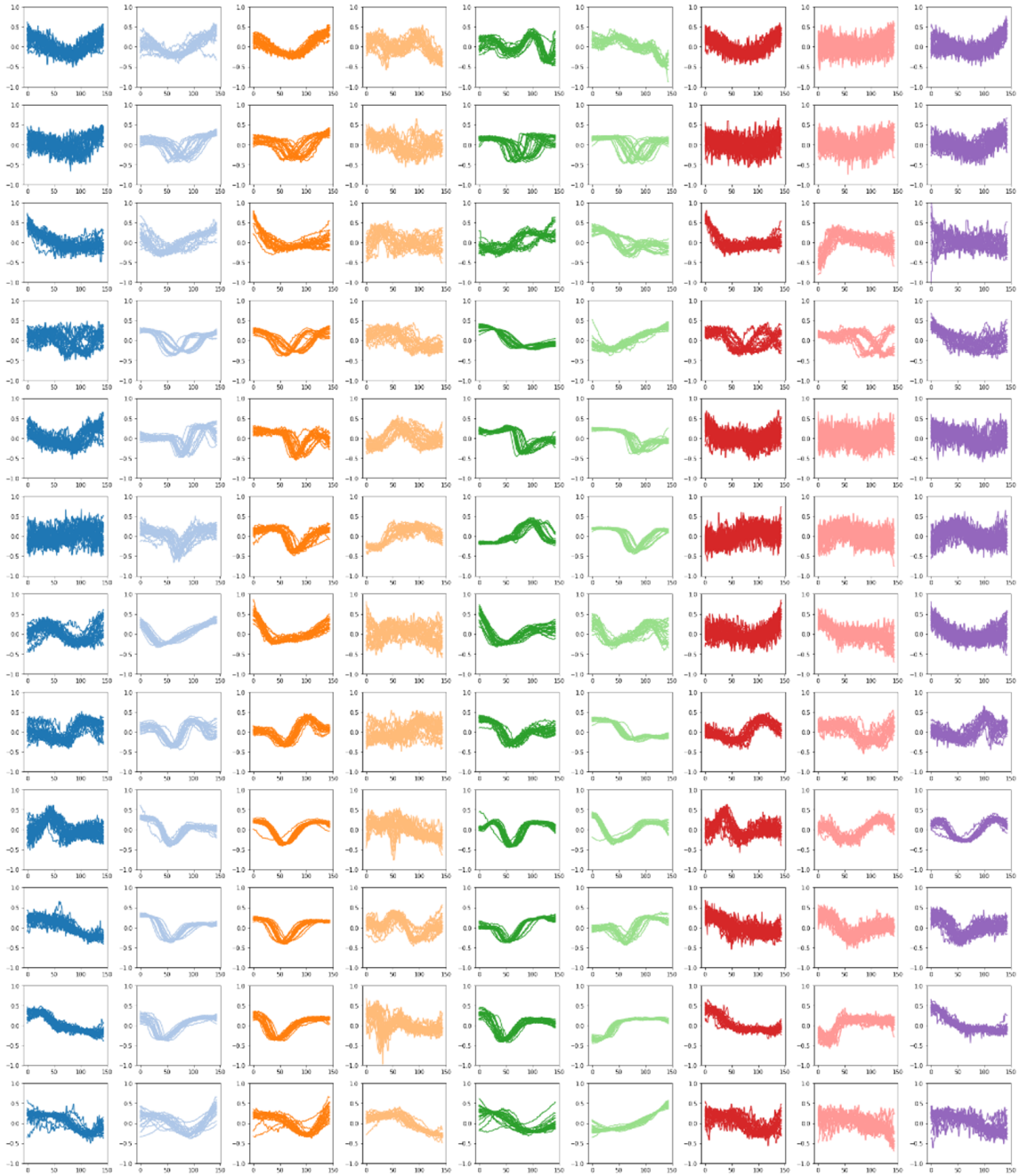

Figure B.4: All the time series data plotted for each of the sensors(columns) for the last 12 classes(rows) in the AWR dataset.

B.3 LSST - Spectrograms for Sensors per Class

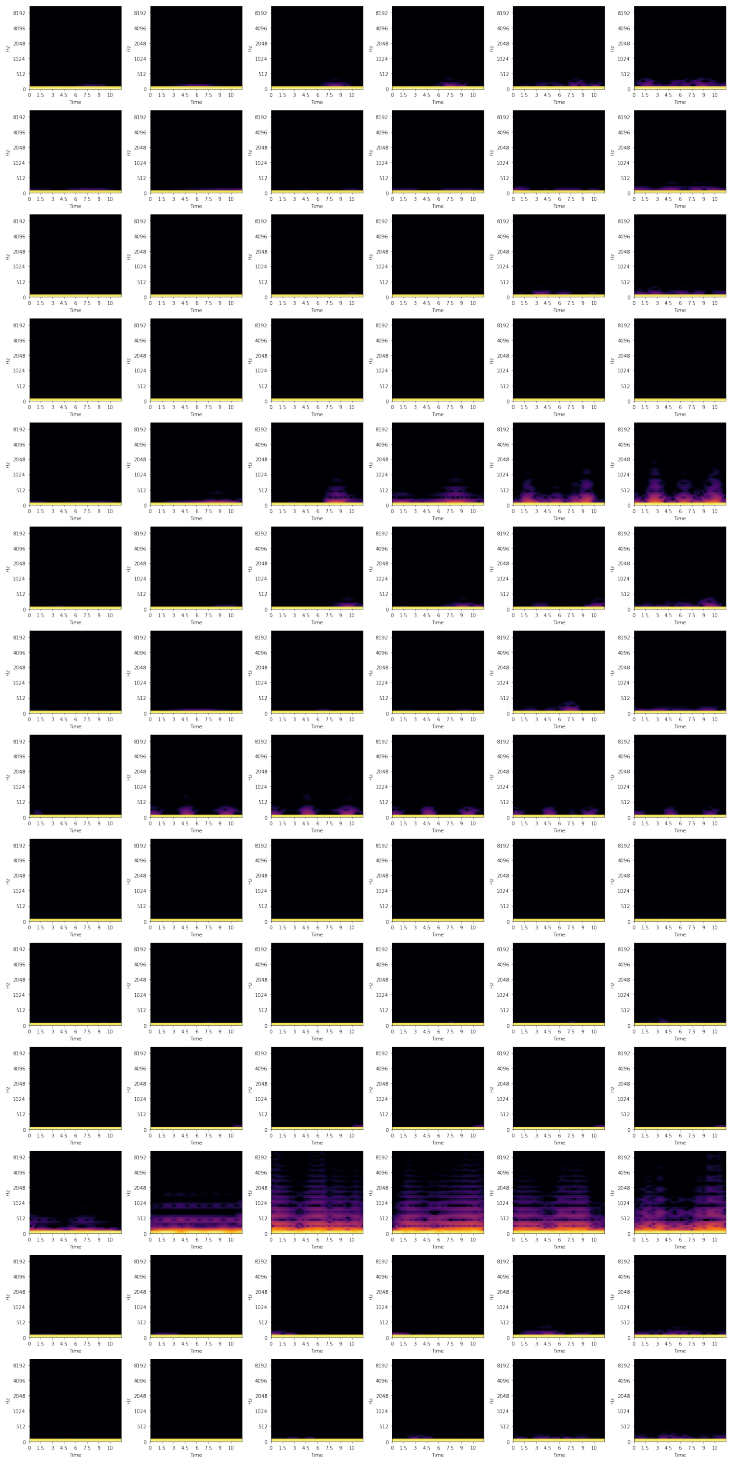

Figure B.5: The spectrograms for each of the sensors(columns) for the classes(rows) in the LSST dataset.

B.4 LSST - Time Series for Sensors per Class

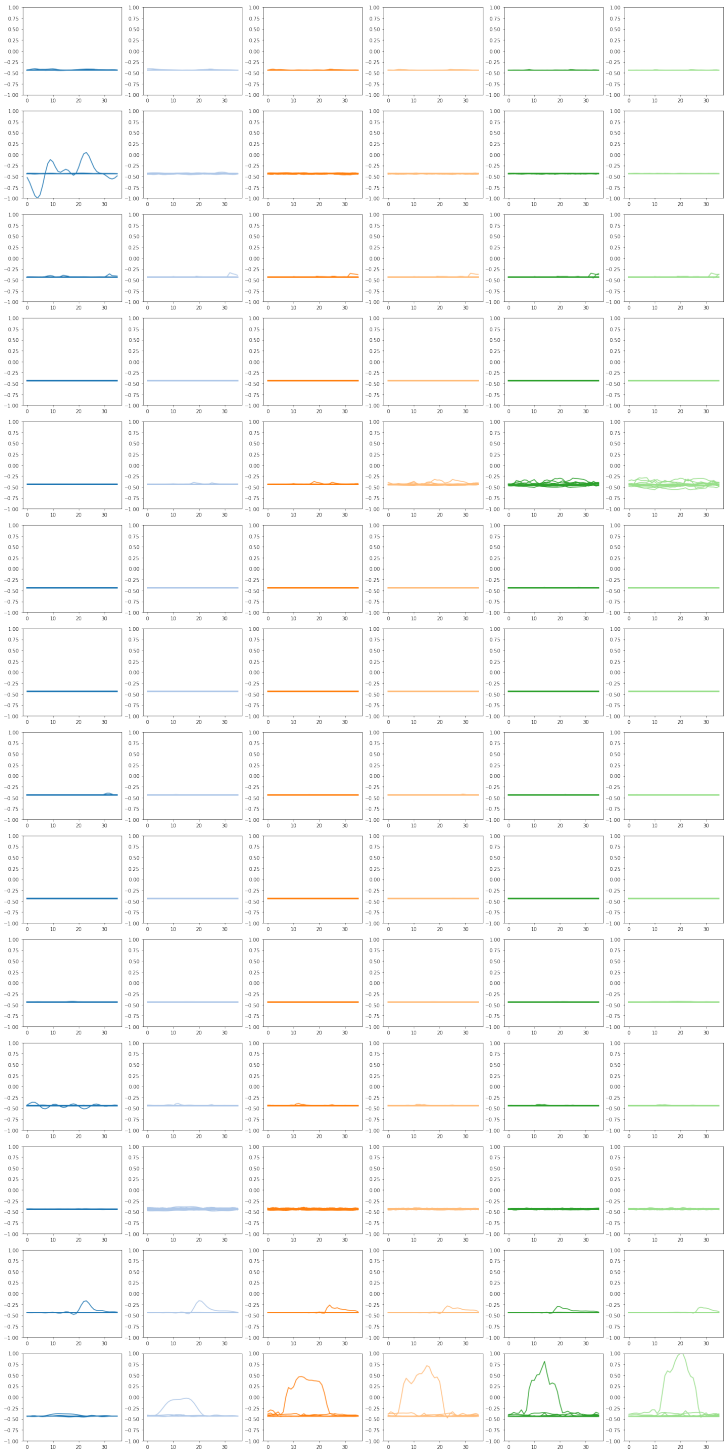

Figure B.6: All the time series data plotted for each of the sensors(columns) for the classes(rows) in the LSST dataset.

B.5 Hydraulic Condition Monitoring - Time Series for Sensors per Class

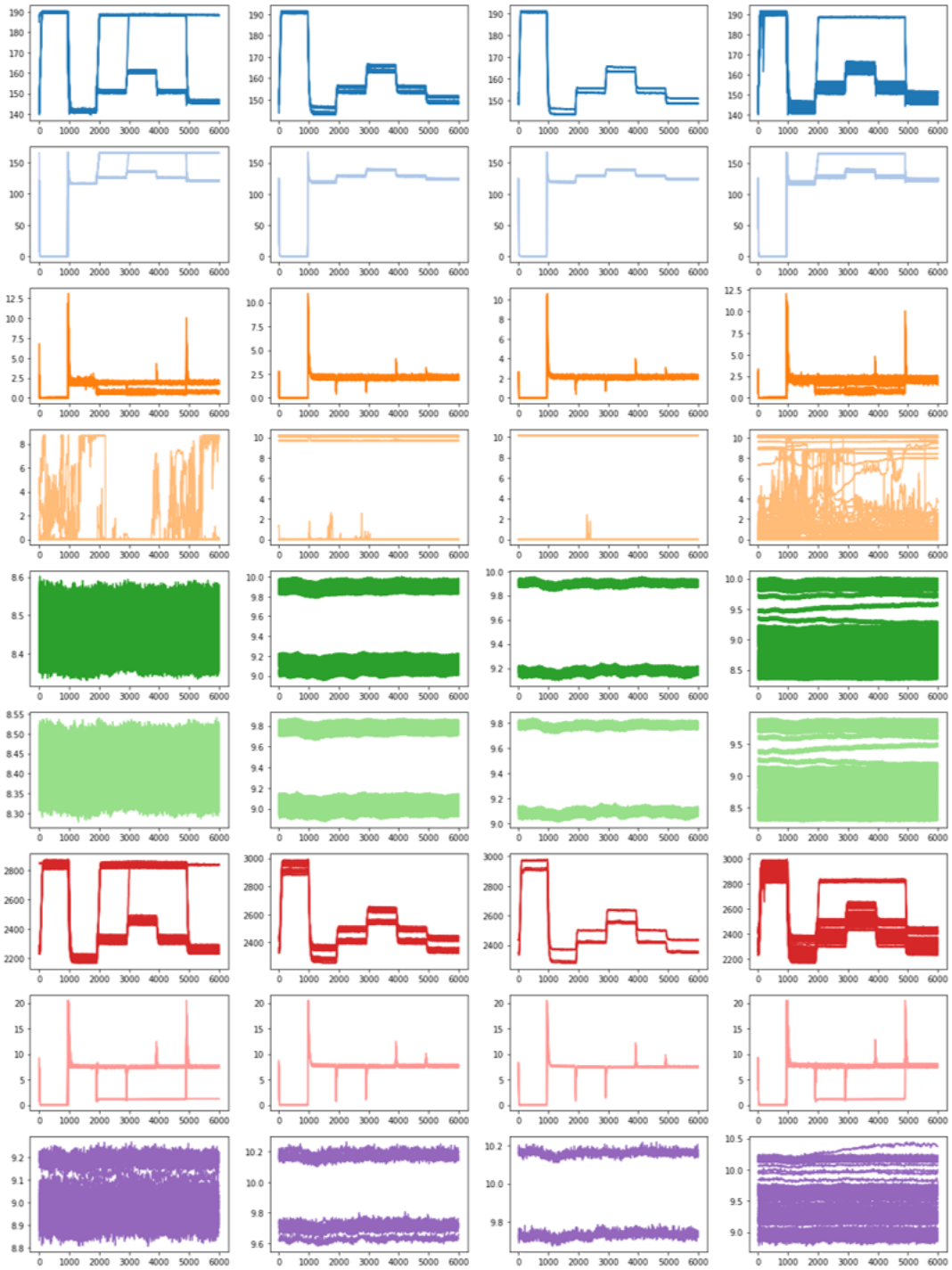

Figure B.7: All the time series data plotted for each of the classes(columns) for the first 9 sensors(rows) in the hydraulic condition monitoring dataset.

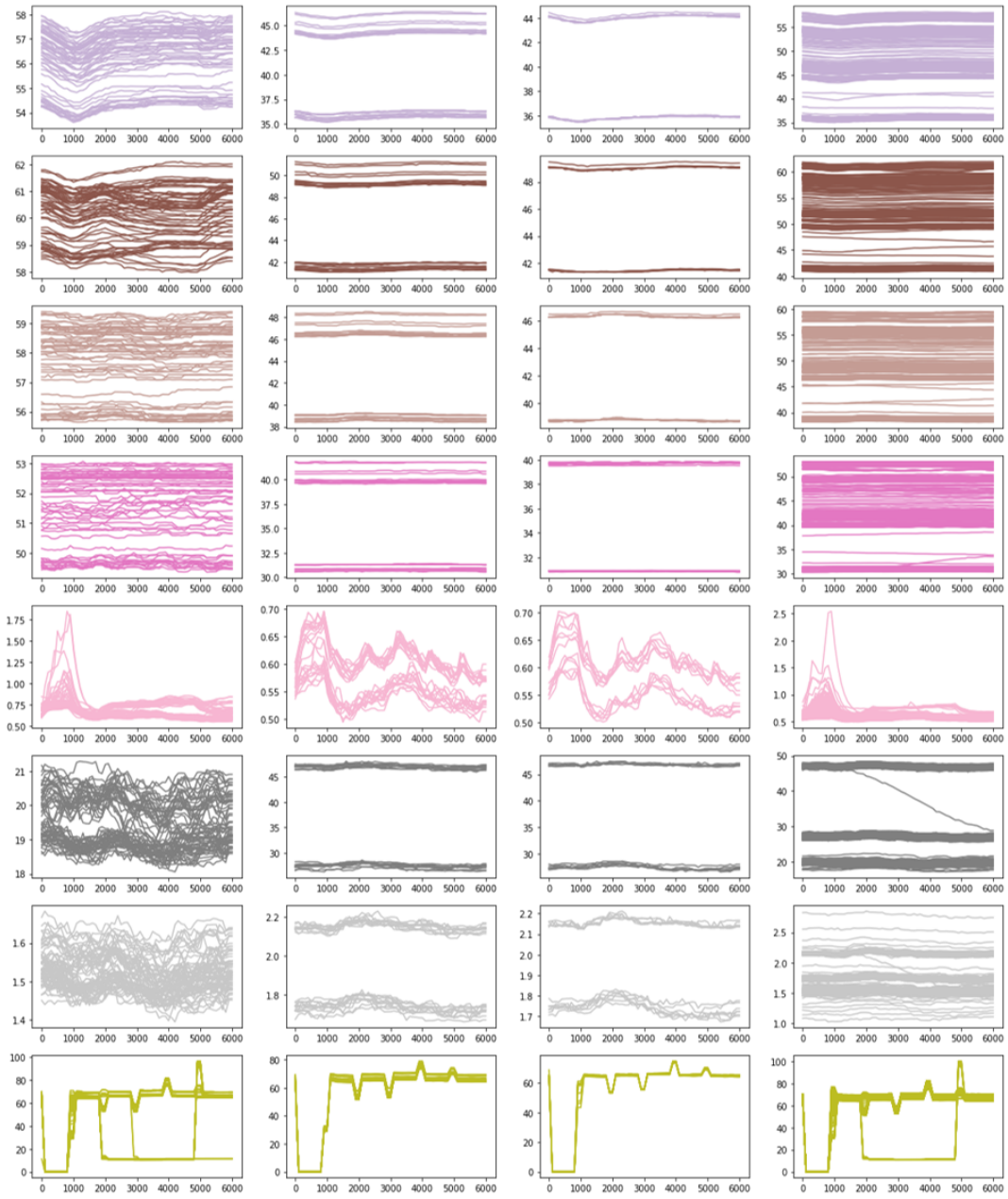

Figure B.8: All the time series data plotted for each of the classes(columns) for the last 8 sensors(rows) in the hydraulic condition monitoring dataset.

B.6 First iteration uTSGAN against true dataset

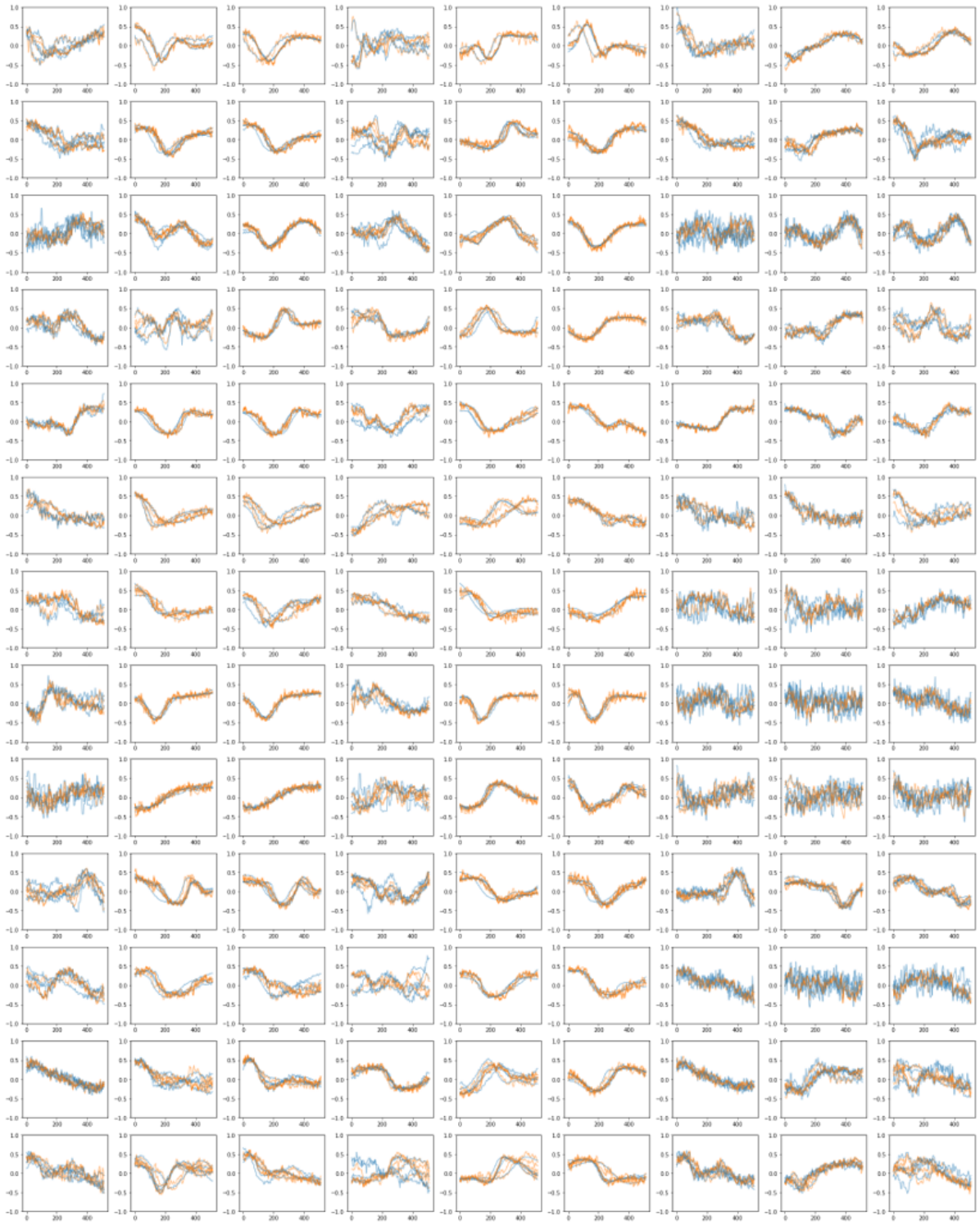

Figure B.9: 5 synthetic samples(orange) for the first 13 classes(rows) of the AWR dataset generated by the first iteration uTSGAN compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor..

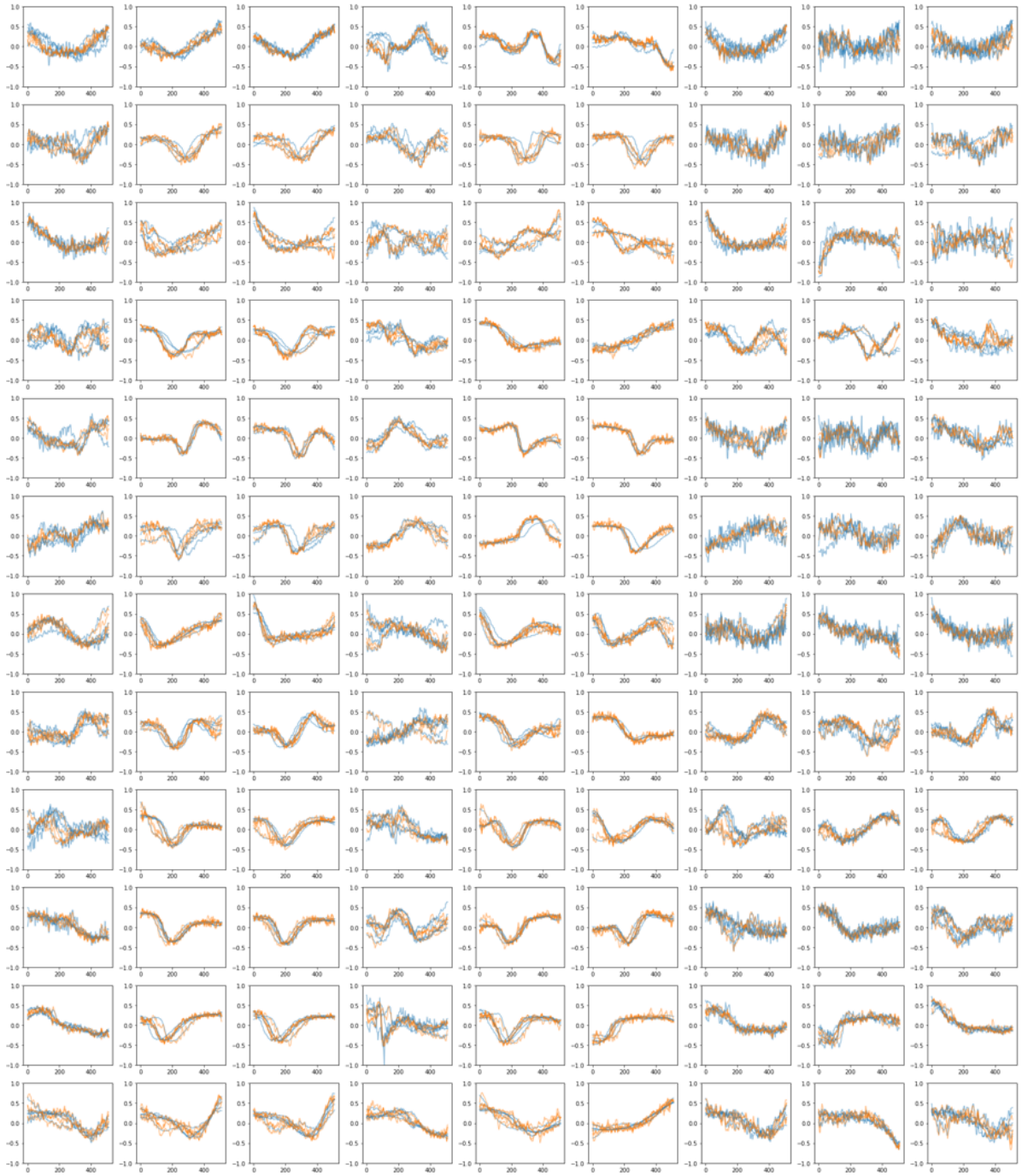

Figure B.10: 5 synthetic samples(orange) for the last 12 classes(rows) of the AWR dataset generated by the first iteration uTSGAN compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor..

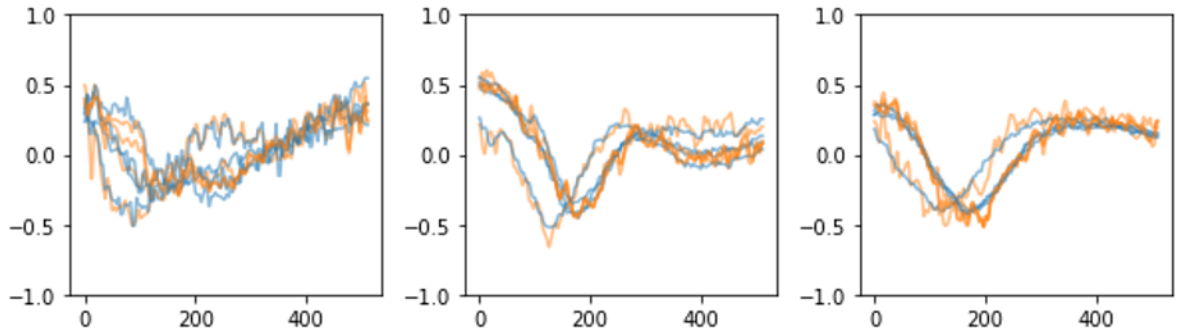

Figure B.11: 5 synthetic samples(orange) for the first class for the first 3 sensors of the AWR dataset generated by the first iteration uTSGAN compared to the true samples(blue) the GAN was trained on. Each column represents a different sensor..

B.7 Benchmark Results from Literature

Table 11 Accuracy of twelve algorithms averaged over thirty stratified re-samples data sets for the UEA MTSC archive

Problem	Default (%)	ROCKET (%)	IT (%)	MUSE (%)	CIF (%)	HC (%)	mrseql (%)	ResNet (%)	DTW_A (%)
AWR	4.00	99.56	99.10	98.87	97.89	97.99	98.98	98.26	98.94
AF	33.3	24.89	22.00	74.00	25.11	29.33	36.89	36.22	22.44
BM	25.0	99.00	100.0	100.0	99.75	100.0	94.83	100.0	99.92
CR	8.33	100.0	99.44	99.77	98.38	99.26	99.21	99.40	100.0
DDG	20.0	46.13	63.47		56.00	47.60	39.27	63.20	56.67
EW	42.0	86.28			90.33	78.17	72.16	45.45	
EP	26.8	99.08	98.65	99.64	98.38	100.0	99.93	99.18	97.37
EC	25.1	44.68	27.92	48.64	72.89	80.68	60.18	28.62	29.57
ER	16.7	98.05	92.10	96.89	95.65	94.26	93.19	87.19	92.89
FD	50.0	69.42	77.24		68.89	69.17		62.97	53.13
FM	49.0	55.27	56.13	54.77	53.90	53.77	55.53	54.70	54.93
HMD	18.9	44.59	42.39	38.02	52.21	37.79	35.23	35.32	30.72
HW	3.8	56.67	65.74	51.85	35.13	50.41	54.04	59.78	60.55
HB	72.2	71.76	73.20	73.59	76.52	72.18	72.52	63.89	68.07
LIB	6.7	90.61	88.72	90.30	91.67	90.28	86.57	94.11	87.85
LSST	31.5	63.15	33.97	63.62	56.17	53.84	60.28	42.94	56.96
MI	50.0	53.13	51.17		51.80	52.17	53.00	49.77	50.37
NATO	16.7	88.54	96.63	87.13	84.41	82.85	86.43	97.11	81.48
PD	10.4	99.56	99.68	98.68	98.97	97.19	97.14	99.64	99.27
PEMS	11.6	85.63	82.83		99.85	97.98	97.15	81.95	78.73
PS	2.6	28.35	36.74		26.56	32.87		30.86	15.39
RS	28.3	92.79	91.69	89.56	89.30	90.64	88.73	91.23	85.79
SRS1	50.2	86.55	84.69	73.58	85.94	86.02	82.86	76.11	81.34
SRS2	50.0	51.35	52.04	49.52	48.87	51.67	49.61	50.24	52.43
SWJ	33.3	45.56	42.00	34.67	45.11	40.67	42.00	30.89	25.56
UW	12.5	94.43	91.23	90.39	92.42	91.31	91.32	88.35	91.51

APPENDIX C

Appendix - EDP

Wind Farm 1 – Signals

Wind turbine SCADA signals for the 5 selected wind turbines

Field Name	Description
Turbine Identifier	Wind turbine ID
TimeStamp	Date and time of the measure
Descriptors + Value	Description and value of the sensor data

Please find below the SCADA signals available for each wind turbine and respective units. The data is given for a 10-minute average period.

Descriptor	Type	Description	Component
Gen_RPM_Max [rpm]	FLOAT	Maximum generator rpm in latest average period	Generator
Gen_RPM_Min [rpm]	FLOAT	Minimum generator rpm in latest average period	Generator
Gen_RPM_Avg [rpm]	FLOAT	Average generator rpm	Generator
Gen_RPM_Std [rpm]	FLOAT	Std. generator rpm in latest average period	Generator
Gen_Bear_Temp_Avg [ºC]	INT	Average temperature in generator bearing 1 (Non-Drive End)	Generator
Gen_Phase1_Temp_Avg [ºC]	INT	Average temperature inside generator in stator windings phase 1	Generator
Gen_Phase2_Temp_Avg [ºC]	INT	Average temperature inside generator in stator windings phase 2	Generator
Gen_Phase3_Temp_Avg [ºC]	INT	Average temperature inside generator in stator windings phase 3	Generator
Hyd_Oil_Temp_Avg [ºC]	INT	Average temperature oil in hydraulic group	Hydraulic
Gear_Oil_Temp_Avg [ºC]	INT	Average temperature oil in gearbox	Gearbox
Gear_Bear_Temp_Avg [ºC]	INT	Average temperature in gearbox bearing on high speed shaft	Gearbox
Nac_Temp_Avg [ºC]	INT	Average temperature in nacelle	Nacelle
Rtr_RPM_Max [rpm]	FLOAT	Maximum rotor rpm in latest average period	Rotor
Rtr_RPM_Min [rpm]	FLOAT	Minimum rotor rpm in latest average period	Rotor
Rtr_RPM_Avg [rpm]	FLOAT	Average rotor rpm	Rotor
Amb_WindSpeed_Max [m/s]	FLOAT	Maximum windspeed within average timebase	Ambient
Amb_WindSpeed_Min [m/s]	FLOAT	Minimum windspeed within average timebase	Ambient
Amb_WindSpeed_Avg [m/s]	FLOAT	Average windspeed within average timebase	Ambient
Amb_WindSpeed_Std [m/s]	FLOAT	Std. windspeed within average timebase	Ambient
Amb_WindDir_Relative_Avg [º]	FLOAT	Average wind relative direction	Ambient
Amb_WindDir_Abs_Avg [º]	FLOAT	Average wind absolute direction	Ambient
Amb_Temp_Avg [ºC]	INT	Average ambient temperature	Ambient

Prod_LatestAvg_ActPwrGen0 [Wh]	INT	Active power - generator disconnected (yaw motor hydraulic motor etc.)	Production
Prod_LatestAvg_ActPwrGen1 [Wh]	INT	Active power - generator connected in delta	Production
Prod_LatestAvg_ActPwrGen2 [Wh]	INT	Active power - generator connected in star	Production
Prod_LatestAvg_TotActPwr [Wh]	INT	Total active power	Production
Prod_LatestAvg_ReactPwrGen0 [VArh]	INT	Reactive power - generator disconnected (yaw motor hydraulic motor etc.)	Production
Prod_LatestAvg_ReactPwrGen1 [VArh]	INT	Reactive power - generator connected in delta	Production
Prod_LatestAvg_ReactPwrGen2 [VArh]	INT	Reactive power - generator connected in star	Production
Prod_LatestAvg_TotReactPwr [VArh]	INT	Total reactive power	Production
HVTrafo_Phase1_Temp_Avg [°C]	INT	Average temperature in HV transformer phase L1	Transformer
HVTrafo_Phase2_Temp_Avg [°C]	INT	Average temperature in HV transformer phase L2	Transformer
HVTrafo_Phase3_Temp_Avg [°C]	INT	Average temperature in HV transformer phase L3	Transformer
Grd_InverterPhase1_Temp_Avg [°C]	INT	Average temperature measured by the IGBT-driver on the grid side inverter	Grid
Cont_Top_Temp_Avg [°C]	INT	Average temperature in the top nacelle controller	Controller
Cont_Hub_Temp_Avg [°C]	INT	Average temperature in the hub controller	Controller
Cont_VCP_Temp_Avg [°C]	INT	Average temperature on the VCP-board	Controller
Gen_SlipRing_Temp_Avg [°C]	INT	Average temperature in the split ring chamber	Generator
Spin_Temp_Avg [°C]	INT	Average temperature in the nose cone	Spinner
Blds_PitchAngle_Min [°]	FLOAT	Maximum angle in latest average period	Blades
Blds_PitchAngle_Max [°]	FLOAT	Minimum angle in latest average period	Blades
Blds_PitchAngle_Avg [°]	FLOAT	Average angle	Blades
Blds_PitchAngle_Std [°]	FLOAT	Std. angle in latest average period	Blades
Cont_VCP_ChokcoilTemp_Avg [°C]	INT	Average temperature in the choke coils on the VCS-section	Controller
Grd_RtrInvPhase1_Temp_Avg [°C]	INT	Average temperature measured by the IGBT-driver on the rotor side inverter phase1	Grid
Grd_RtrInvPhase2_Temp_Avg [°C]	INT	Average temperature measured by the IGBT-driver on the rotor side inverter phase2	Grid
Grd_RtrInvPhase3_Temp_Avg [°C]	INT	Average temperature measured by the IGBT-driver on the rotor side inverter phase3	Grid
Cont_VCP_WtrTemp_Avg [°C]	INT	Average temperature in the VCS cooling water	Controller
Grd_Prod_Pwr_Avg [kW]	FLOAT	Power average according to	Grid
Grd_Prod_CosPhi_Avg	FLOAT	Average actual phase displacement	Grid
Grd_Prod_Freq_Avg [Hz]	FLOAT	Average frequency	Grid

Grd_Prod_VoltPhse1_Avg [V]	FLOAT	Averaged voltage in phase 1	Grid
Grd_Prod_VoltPhse2_Avg [V]	FLOAT	Averaged voltage in phase 2	Grid
Grd_Prod_VoltPhse3_Avg [V]	FLOAT	Averaged voltage in phase 3	Grid
Grd_Prod_CurPhse1_Avg [A]	FLOAT	Averaged current in phase 1	Grid
Grd_Prod_CurPhse2_Avg [A]	FLOAT	Averaged current in phase 2	Grid
Grd_Prod_CurPhse3_Avg [A]	FLOAT	Averaged current in phase 3	Grid
Grd_Prod_Pwr_Max [kW]	FLOAT	Maximum Power in latest average period	Grid
Grd_Prod_Pwr_Min [kW]	FLOAT	Minimum Power in latest average period	Grid
Grd_Busbar_Temp_Avg [ºC]	INT	Average temperature in the busbar section	Grid
Rtr_RPM_Std [rpm]	FLOAT	Std. rotor rpm in latest average period	Rotor
Amb_WindSpeed_Est_Avg [m/s]	FLOAT	Average windspeed within average timebase	Ambient
Grd_Prod_Pwr_Std [kW]	FLOAT	Std. power in latest average period	Grid
Grd_Prod_ReactPwr_Avg [kVAr]	FLOAT	Average grid reactive power	Grid
Grd_Prod_ReactPwr_Max [kVAr]	FLOAT	Maximum grid reactive power	Grid
Grd_Prod_ReactPwr_Min [kVAr]	FLOAT	Minimum grid reactive Power	Grid
Grd_Prod_ReactPwr_Std [kVAr]	FLOAT	Std. Deviation grid reactive power	Grid
Grd_Prod_PsblePwr_Avg [kW]	FLOAT	Average possible grid active power	Grid
Grd_Prod_PsblePwr_Max [kW]	FLOAT	Maximum possible grid active power	Grid
Grd_Prod_PsblePwr_Min [kW]	FLOAT	Minimum possible grid active power	Grid
Grd_Prod_PsblePwr_Std [kW]	FLOAT	Std. possible grid active power	Grid
Grd_Prod_PsbleInd_Avg [kVAr]	FLOAT	Average possible inductive reactive power	Grid
Grd_Prod_PsbleInd_Max [kVAr]	FLOAT	Maximum possible inductive reactive power	Grid
Grd_Prod_PsbleInd_Min [kVAr]	FLOAT	Minimum possible inductive reactive power	Grid
Grd_Prod_PsbleInd_Std [kVAr]	FLOAT	Std. possible inductive reactive power	Grid
Grd_Prod_PsbleCap_Avg [kVAr]	FLOAT	Average possible capacitive reactive power	Grid
Grd_Prod_PsbleCap_Max [kVAr]	FLOAT	Maximum possible capacitive reactive power	Grid
Grd_Prod_PsbleCap_Min [kVAr]	FLOAT	Minimum possible capacitive reactive power	Grid
Grd_Prod_PsbleCap_Std [kVAr]	FLOAT	Std. possible capacitive reactive power	Grid
Gen_Bear2_Temp_Avg [ºC]	INT	Average temperature in generator bearing 2 (Drive End)	Generator
Nac_Direction_Avg [º]	FLOAT	Average nacelle direction	Nacelle

$\sqrt{17}$
$\Omega \int \delta e^{i\pi} = -1$
{2.7182818284}
ϑφερτυθιοπσδφγηξκλ
χ^2
$\sum$