Generative Adversarial Networks for the Synthesis of Unbalanced Irregular Time Series

Thomas Poudevigne Durance

School of Mathematics

Cardiff University

A thesis submitted for the degree of

Doctor of Philosophy

13 March 2024

Abstract

Good quality data are key to informed decision making. Yet real-world data collection can be challenging and resource demanding, so that good quality data are not always available. This thesis explores synthetic data generation, focusing on Generative Adversarial Networks (GANs) which are a class of machine learning models. Real-life environmental data such as water quality data brings added challenges in that they are often datasets with missing values, time dependencies and rare events. The aim of the thesis was to investigate the potential of GANs to synthesise such data.

Identification of the data synthesis techniques available for such datasets showed that while GANs are complex models with multiple parameters to optimise, they could be well suited for this purpose. To address the challenge of synthetising unbalanced irregular time series, first a novel GAN was built to create synthetic data directly from datasets with missing values. Called MaWGAN, it is based on the Wasserstein distance (like WGAN) and uses a mask to hide missing values from the Critic to enable the GAN to run.

Then to capture the dependency in the data, two routes were developed: a new algorithm (Force-GAN) that embedded a third neural network in the MaWGAN architecture, and an alternative option where missing data in the irregular time series are first imputed with a novel technique (Hankel Imputation) that preserves the noise in the data. These insights are then used to illustrate the value of the novel GANs developed in this thesis to address complex real-world challenges such as taste and odour issues in drinking water.

Research contributions to the field include: a GAN that can handle missing values in the dataset (MaWGAN), an expansion that handles time series as well (Force-GAN), a method to impute data while keeping the original noise and trend of the dataset (Hankel Imputation). The value of these novel methods to predict rare events in time series is also demonstrated using water quality unbalanced datasets with missing values.

Acknowledgement

Firstly, we would like to express my sincere gratitude to my supervisors, Prof. Owen Jones and Prof. Yeping Qin for their guidance throughout my research journey. Your supportive encouragements have made such a difference to my journey. we would also like to thank Dr Vincent Knight for helping me in the first year of my PhD and teaching me how to code professionally.

we would also like to acknowledge the financial support provided by KESS2, along with Dŵr Cymru/Welsh Water, which made this research possible. we would like to especially thank Kevin Parry, Willow Smallbone, Liam Buttler for taking the time to comment on the thesis as it developed.

we am grateful to the School of Mathematics and computing staff and my fellow PhD students for creating a supportive community and for making my PhD thoroughly enjoyable including my companions in arms (Tim, Lizz, Mike) for helping me with general knowledge and fun stuff as well,my climbing group for keeping me in shape, and all those who came up with great ideas for games, tea, dinners, quiz nights that make the life of a PhD student so much more worthwhile.

we would like to thank my family; you know it, we would not have done it without your every day support especially my dog Leo, for the unconditional moral support.

we would also like to thank Learning Support at Cardiff University for all their help throughout the years, and the creators of my several assitance software for their brilliant work.

Publications

From Chapter 4: Poudevigne-Durance, T., Jones, O.D. and Qin, Y., 2022. MaWGAN: A generative adversarial network to create synthetic data from datasets with missing data.*Electronics, 11(6),* p.837.

From Chapter 5: Jones, O., Poudevigne-Durance, T. and Qin, Y., 2023. Synthesis of time-series with missing observations using generative adversarial networks. *In Proceedings of the 34th Panhellenic Statistics Conference* (pp. 154-166). Greek Statistical Institute.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This Chapter starts by briefly explaining the general context of the research challenges this thesis addresses, namely the challenge of working with 'real-world' data (1.1), and the specific challenges that stem from data collected by water utilities (1.2). It then provides an overview of the research aims of this thesis and the structure adopted (1.3).

1.1 The challenge of Real data

Organisations collect data for different purposes, including for example data to support decision making like data provided by the Office for National Statistics to support goverment investment, or to monitor key processes such as the water quality of the drinking water that water utilities distribute. Data collected through time can be analysed either to monitor trends in variables of interest, or to model for predictive purposes. For example, data on temperature and rainfall is collected to predict weather conditions in the near future and the trends are used to understand long term changes in climate patterns. Data can be collected by automatised probes or samplers on a regular time step, or samples can be manually collected to be analysed locally or in a laboratory.

Manual data collection presents its own challenges because in practice people don't work every day of the year and don't necessarily access the sampling site at the same time every day. This means that samples and therefore data is in the form of **irregular time series**.

There are four main challenges with this type of 'real-world' data:

- Real-world data often have missing values (e.g. because of automatic sensor failures or irregular sampling) and many models do not work directly with **missing values** (e.g. [1, 2]) so that preaparatory work is necessary.
- Environmental systems evolve over time, for example bacterial populations tend to grow exponentially [3], others like insect populations might vary seasonaly (e.g. [4]). There are thus **time dependencies** in these datasets because the population at any time is generated (and therefore dependent) by the previous population.
- Environmental monitoring data have sometimes rare events (e.g. Cryptosporidium outbreaks in drinking water), and some of these can have significant impact (e.g. Cryptosporidium outbreaks in drinking water cost millions to water utilities when they occur). Due to the rarity of such events, datasets are **unbalanced**, and this makes prediction of future events challenging.
- **Private** data sets may contain sensitive, personal, and identifiable information. If the data needs to be analysed or shared, the original data must therefore be modified in some way so that the individual private information is not visible.

1.2 Water data challenges

A good illustration of the challenges faced by organisations monitoring environmental variables, is the case of the water sector. Water utilities have the responsibility of delivering safe drinking water and of collecting and disposing appropriately of wastewater. This thesis illustrates the challenges in modelling real life environmental monitoring data through a collaboration with Dŵr Cymru Welsh Water (DCWW), the main water utility in Wales.

DCWW is a water utility with an unusual financial model as it is non for profit. This means that the company does not have shareholders and that profits go back into keeping bills down and looking after freshwaters and their environment. DCWW serves over three million people with drinking water and is also responsible for collecting, treating and appropriately disposing of the wastewater. As a water utility, DCWW has customer data which are sensitive in nature and would require some form of anonymisation if they need to be analysed outside of the facility.

To ensure that the drinking water provided is constantly of the highest quality, DCWW monitor on a regular basis a range of water variables. Water quality variables for drinking water are determined by the Drinking Water Inspectorate. The company tracks physical, chemical and biological variables of water quality both at abstraction points and at the tap. Abstraction points are located either in rivers, lakes or purpose-built reservoirs.

If the levels of monitored drinking water quality are poor, the company has to intervene, and sometimes even cease to supply that water. For example, in 2005, DCWW had to discontinue water supply for over 70,000 people for several days because of an outbreak of Cryptosporidium, a gut parasite that causes severe diarrhea. The economic consequences for DCWW of such events are significant. DCWW paid their customers over £1m in compensation for this event alone. These events are usually rare and therefore data on them are scarce. This means that DCWW struggles to build models that would allow to predict, and therefore solve, such issues.

In the past, DCWW had tried to use a method to artificially augment the amount of data called SMOTE 'Synthetic Minority Over-sampling TEchnique' invented by Chawla et al. [5]. However, while this is an often cited approach to unbalanced data [6], this method did not produce satisfactory results with the complex DCWW datasets. Others have also concured that SMOTE only performs on a narrow set of models [6] and in particular do not generalise well to high-dimensional datasets such as the DCWW datasets [7]. DCWW therefore needs a better solution, tailored to their data.

The data collected by DCWW is also usually comprises a series of variables that together inform on the overall water quality and are often linked. Datasets are therefore multivariate. Common water quality variables include chemical measurements like pH or Magnesium, physical measurements like turbidity or temperature, and biological measurements such as algal content or bacterial content. Concern over water quality is often linked to biological measures and the water company needs to better understand what causes these biological events that can have serious financial consequences. DCWW is thus keen to model these events so that they can predict when these are likely to occur and prepare for this event.

1.3 Research aims and thesis structure

The main research question proposed by DCWW can be summarised as: *Can GANs be used to augment data to provide better models for diagnosing or predicting risk situations?*

The Thesis was structured in a sequential way reflecting the need to address three different challenges with the data, namely their irregular, time-dependent and unbalanced nature. This thesis will acknowledge the potential data privacy aspect of synthetic data and GANs (see section 2.2.4 and 3.6), but will only focus on the data imputation and augmentation aspect.

- Chapter 2 investigates the value of data synthesis and its applications to address the challenge of unbalanced datasets through a literature overview. It explores and compares data synthesis techniques to augment tabular datasets, and their limitations. It identifies the evidence to support the use of GANs.
- Chapter 3 investigates the literature around GANs and provides insights into the benefits and limitations of GANs for a range of data, including irregular time series.
- Chapter 4 simplifies the research question by building on the assumption that data are independent and explores the challenge of running GANs on data with missing values.
- Chapter 5 builds on the 'GAN for missing values' developed in Chapter 4, and reintroduces the dependencies that were ignored in Chapter 4.
- Chapter 6 explores an alternative route, where missing data in the irregular time series are first imputed with a novel technique that preserves the noise in the data, before it can be run through a GAN.

- Chapter 7 uses the insight gained in Chapters 4,5 and 6 to address the main research question. To illustrate the approach, data from DCWW on Taste and Odour issues in drinking water are used.
- Chapter 8 reflects on the thesis and highlights its contributions.

Chapter 2

The case for data synthesis

2.1 Overview

This Chapter sets out to understand the main options to address the four data challenges posed by the DCWW multivariate datasets summarised in Chapter 1: missing data, time series, unbalanced data and private data. While simple solutions to some of these challenges have been developed, these often only consider one problem at a time and have a range of shortfalls which are detailed in section 2.2. Section 2.3 shows how data synthesis using statistical approaches can address some of these challenges together. Section 2.4 then introduces the concepts and building blocks for data synthesis using machine learning that will provide the context for this thesis.

2.2 Data challenges

This section explores in depth the challenges indentified in Chapter 1, investigating the challenge of missing data (2.2.1), and how they occur in timeseries (2.2.2), and how unbalanced data challenges are usually addressed (2.2.3). The section then briefly explores the case of private data (2.2.4).

2.2.1 Missing data

Missing data is a frequent issue as demonstrated by a quick search on Web of Science on the 7th October 2023 which showed there were 109,716 peer reviewed papers attempting to solve the issue. Missing values are common and can happen for a range of reasons. Rubin (1976) [8] defines three types of missing data: random (MAR - Missing at Random), completely random (MCAR - Missing Completely At Random), and non-random (NMAR - Not Missing at Random). Considering the independent observations $x_i = (x_{i1}, \ldots, x_{id})^T$, a *mask* $m_i = (m_{i1}, \ldots, m_{id})^T$ corresponding to the observations, where $m_{ij} = 0$ if x_{ij} is missing and $m_{ij} = 1$ if it is present. In the case of MAR, m_{ij} is independent of x_{ij} but dependent on some x_{ik} for $k \neq j$, in the case MCAR, for any j, m_{ij} is independent of x_i, in the case of NMAR, m_{ij} is dependent on x_{ij}.

When data is MAR, there is usually an explanation as to why the data is missing, and it is often event-based. For example, in a recent paper analysing data from water reservoirs to predict taste and odour events [9], weather conditions or restrictions such as lockdowns that did not allow access to the sampling sites were a common cause for missing data. In this case, the probability that a sample value is missing does not depend on the missing value but does depend on some other characteristic.

MCAR data are also randomly distributed in the dataset, but unlike MAR, the probability that a sample value is missing depends both on the missing value and on some other characteristics. For example, in the reservoir study cited above, they noticed that the data recorder may have forgotten to record the data in some instances.

In the NMAR case, the explanation of the missing value depends on a known characteristic. For example, some instruments used by water companies to measure environmental variables such as Amonium or Phosphorous have a limit of detection. This means the value recorded is put down as '$< LOD$' where LOD represents the Limit Of the Detection for that variable. Unless it is replaced by an agreed value (usually half the LOD), the dataset also shows these cases as missing values.

2.2.2 Irregular time series in a multivariate context

Irregular time series occur when data are not recorded regularly. This can happen for example because staff collecting daily variables might not be able to collect on a given day (e.g. because of illness or challenging weather conditions) and might not necessarily access the sampling site at the same time every day, or sample more frequently when there is an event that needs to be more closely monitored. When analysing multiple variables that are irregular time series, one of the challenges is that their collection times might not match. To achieve a common timeline, the set of irregular time series can be transformed into a set of regular time series with missing values [10].

This reformatting can be done by picking a time step frequency that suits all the time series (for example, daily or monthly or hourly), then assigning each observation to the closest timestamp, and resolving any conflict by either taking the mean, maximum, or minimum of the conflicting observations (for example, the value of week 5 in Figure 2.1 is the mean of the two adjacent days). There is a trade-off between a larger time step that removes more missing values but reduces observations and a shorter time step that removes fewer missing values but retains more observations [11]. For example, transforming the irregular timeseries in Figure 2.1 to weekly you retain most of the observations but week 8 is missing, on the other hand doing fortnightly will have no missing values but will lose observations around week 2,4 and 6.

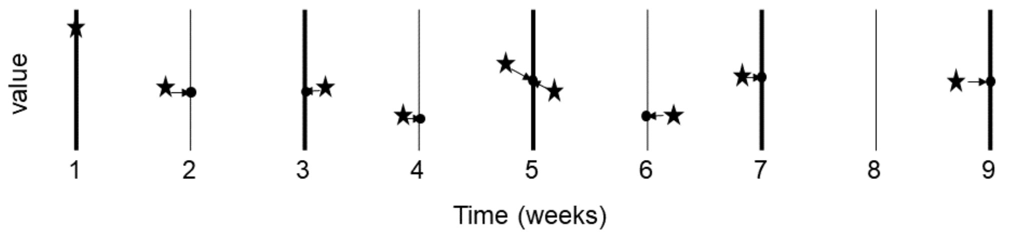


Figure 2.1: Example of reformatting irregular timeseries into regular with missing values, stars indicate the original irregular timeseries, bars show the chosen timestep

While the challenge of irregular time series in a multivariate context can be solved by reformatting the data into a regular time series, a remaining challenge is the presence of missing data in these time series. Resolving the issue of missing data in time series is more complex because of the time dependency between each observation.

2.2.3 Unbalanced data

Data sometimes needs to be categorised, for example to separate taste and odour events from baseline conditions in water reservoirs (see Chapter 7). When there are categories with little data (minor categories), as is the case with rare events, the ratio of categories is unequal, and these datasets are **unbalanced**. Minor categories tend to be underestimated when using predictive models such as logistic regressions on a Unbalanced dataset [12]. This means that the major categories are well modelled, but minor categories are not.

There are many different approaches to mitigate the impact of unbalanced datasets [13]. Those approaches can be categorised into:

- Strategic data Sampling,
- Strategic model use (ensemble learning)
- and Synthetic data generation (section 2.3).

Sampling strategies where various oversampling and/or undersampling techniques aim to compensate for unbalanced distributions. Ensemble learning is combining the predictions from multiple models. This includes: Bagging which involves fitting several models of the same type (e.g. Random Forest models) on different samples of a dataset and averaging the predictions, Stacking which involves fitting many different model types on the same samples and using another model to learn how to best combine the predictions, and Boosting which involves sequentially adding models that correct the predictions made by previous models and outputs a weighted average of the predictions.

2.2.4 The case of Private data

Data sets may contain sensitive, personal, and identifiable information. For example, data may record names or addresses that allow identifying the person it relates to and data on health that are of sensitive nature (e.g. genetic data, drug use...). For legal and ethical reasons this information needs to be protected from non-authorised access. In the UK, the confidentiality of personal information is in legislation under the Data Protection Act (1998) [14] and the Common Law. There are also similar laws worldwide; for example, the EU Data Protection Directive (1995) - [15].

If this sensitive data needs to be shared, the original data must therefore be modified in some way so that the individual private information is not visible whilst still allowing access to the core data. As an example, those modifications can be important for medical trial data that needs to be shared between researchers so that medical progress can be made [16]. It is also important in many other areas where data sharing is essential because it means that work can be done on more datasets, or new methods can be tried.

Private data contains three different types of attributes [14]:

- attributes that are direct identifiers such as a name or phone number. These unique values are often called identifiers (ID)
- attributes that are indirect identifiers because when combined with other data, they may still allow identifying the person. Some are more sensitive (like diseases) and others less (like age and sex).
- other attributes in the dataset that do not need to be protected.

These data can be made anonymous by deleting direct identifiers and modifying indirect identifiers (e.g. Keerie 2018 et al [16]).

However, there needs to be a balance between modifying the data enough that no-one can extract the private data, and modifying the data too much that the data's accuracy and therefore the utility of the data is compromised. It is challenging to achieve that balance. Keerie et al. [16], for example, suggest that relatively simple facts, such as age and country of residence, can identify a person if they are old or famous, since the names, dates of birth and countries of those living people are published in Wikipedia.

Other simple approaches to anonymise data include [17, 18]:

- Generalisation: A value is replaced with a generic value (for example the age of all the records between 30 and 40 is replaced by the average value of 35 years).

- Suppression: where values are removed from data (for example a postcode like CY27 2GT could be replaced by CY** **T).
- Permutation or data randomisation: where values from a group are randomly mixed up
- Perturbation: where noise is added to the data or replaced by creating synthetic data.

The main challenge with all these anonymisation processes is that the more data are made anonymous, the more information is lost, and the more their utility decreases [19].

2.3 Data synthesis using statistical approaches

This section looks at the current statistical approaches to the challenges outlined in section (2.2) and explores: the nature of synthetic data (2.3.1), imputation approaches to solve the challenge of missing values (2.3.2), including for timeseries (2.3.3), as well as the case of data augmentation to address unbalanced datasets(2.3.4). This assesment of statistical approaches to data synthesis form the comparative basis for the exploration of machine learning approaches detailed in section 2.4.

2.3.1 Synthetic data

Data synthesis is the process of generating data [20]. **Generative models** are a class of models that capture the distribution of data, so that this distribution can be sampled to create synthetic data [21]. Generative models are useful for numerous applications, but are particularly interesting to address problems where data is scarce [22]. Synthetic datasets can be categorised into two types, partially synthetic datasets and fully synthetic datasets. Partially synthetic datasets are a mix of synthetic and original data. These are mainly generated when missing values or specific private variables need to be replaced. Fully synthetic datasets do not contain any original data [23].

The following section details how Generative models using statistical approaches can offer a way to address some of the challenges detailed in the previous section, specifically the issue of imputing missing values in independent data or time series, and the issue of unbalanced datasets.

2.3.2 Imputation for Independent data with missing values

Imputation is the process of completing datasets that have missing values. Imputation methods are often based on the analysis of the non-missing data.

The different strategies for implementation include for example: Mean, Hot-Deck, linear regression, Stochastic regression and Multivariate Imputation by Chained Equations (MICE) [24]. The Hot Deck techniques involve finding a similar or closely matched dataset, for example, the Last Observation Carried Forward (LOCF) method imputes the missing value from the last observation in the dataset. Linear regression imputes new numbers predicted from the regression of the whole dataset, other regression forms can be used such as quadratic, cubic as well as non-polynomial [24]. Stochastic regression imputation additionally involves augmenting the regression predicted value with a residual term that is normally distributed with a mean of zero and a variance equal to the residual variance. This has an advantage over a simple regression imputation in that it does not overestimate correlations and underestimate variances and covariances [25]. Each variable is expresssed as a linear function of the other variables, and the regressions are used to impute the missing values in each variable. The MICE procedure only works on MAR datasets [26].

2.3.3 Imputation for time series with missing values

To address the challenge of temporal dependencies characteristic of timeseries, other methods have been developed to address the specific problem of imputing missing values in time series, and a brief overview is provided here. As for imputation methods for independent data, they all have their advantages and disadvantages. For example, some, like linear interpolation are easy to compute, but outputs are too simplified. Spline uses piecewise polynomials and produces more accurate outputs [27]. The Stineman's method provides an interpolating function that prevents abrupt steps in the sequence [28].

More complex methods tend to be specific to certain types of time series, such as the Seasonally Decomposed Missing Value Imputation method [29] which allows for seasonal effects. Most imputation methods impute an expected value based on the mean of the distribution, so they do not capture the distribution as a whole as they are often based on smoothing techniques.

2.3.4 Data Augmentation to address unbalanced datasets

Data augmentation is the process of artificially increasing the dataset, and it is a type of Partial data synthesis. Examples of data augmentation methods include the "Synthetic Minority Over-sampling TEchnique (SMOTE)" or the "Adaptive Synthetic Sampling Technique (ADASYN)". SMOTE was invented by Chawla et al. [5]. This Data Augmentation method creates new data along the lines that link neighbouring data points. ADASYN proposes a similar approach to SMOTE but with additional noise to avoid the linearity issues [30]. The essential idea of ADASYN is to use a weighted distribution for different minority class examples according to their level of difficulty in learning. The main challenges with ADASYN are: overfitting, as generated samples may not accurately represent the distribution of the minority class, and sensitivity to noise and outliers in the dataset.

2.4 Data Synthesis using Machine learning

This section looks at current Machine learning approaches to the challenges detailed in section 2.2 and some advantages over the approaches described in 2.3.

2.4.1 What is Machine learning

Machine learning is where a computer automatically detects patterns in data and then uses these patterns for a range of applications such as predicting future data or generating new data [31]. There are many ways to classify Machine learning problems, and Figure 2.2 summarises three main classes:

- Supervised: where the computer is given labelled training data and there is an expected output, this is mainly regression and classification problems
- Unsupervised: where the pattern of the data and the output of the model is undetermined so the computer tries to fit a distribution to data that is not labelled. This includes a range of different problems: dimension reduction methods like Principle Component Analysis which are used in Chapter 7, clustering, density estimation and Generative models which are at the core of this thesis.
- Reinforcement: where the computer estimates the model parameters by interacting with the environment through sequential decision making.

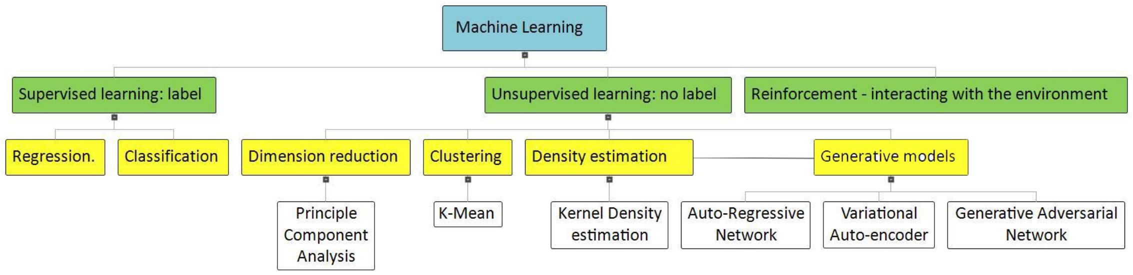

Machine Learning
Supervised learning: label
Unsupervised learning: no label
Reinforcement - interacting with the environment
Regression.
Classification
Dimension reduction
Clustering
Density estimation
Generative models
Principle Component Analysis
K-Mean
Kernel Density estimation
Auto-Regressive Network
Variational Auto-encoder
Generative Adversarial Network

Figure 2.2: Types of machine learning. Problem classes are in green, problems are in yellow, and a few examples of models that can address these problems are in white.

There are a range of Generative machine learning models, including: Gaussian Mixture Models, Hidden Markov Models, Latent Dirichlet Allocation, Boltzmann Machines, Deep Belief Networks, Variational Autoencoders (VAEs), Diffusion Models (DM) and Generative Adversarial Networks (see [32] for an overview).

Three machine learning models stand out for the quality of their synthetic data and these are built on neural networks: Diffusion Models which consist of a process that gradually adds noise to the dataset, VAEs which are likelihood-based models based on Bayesian networks [33] inspired by the Helmholtz machine, and GANs (Generative Adversarial Networks) which are combinations of neural networks, a generator and a discriminator [34]. Each of these have different strenghs and weaknesses (see Figure 2.3).

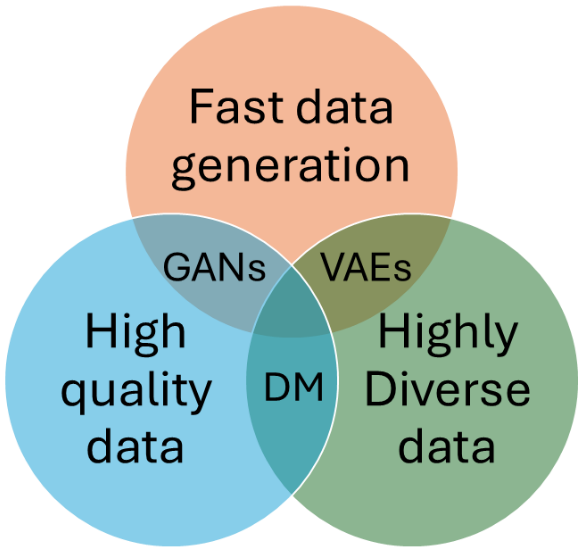


Figure 2.3: Strengths and weaknesses of models base on neural networks.

VAEs can be easily trained and generate fast and highly diverse data, but tend to generate low fidelity synthetic data [35]. Diffusion Models are also easy to train, and generate highly diverse data of high fidelity. However the process is slow because it requires multiple runs of the model to gradually generate samples [36]. GAN architectures, on the other hand, can generate data quickly and an infinite amount of times, while also generating high fidelity data. GANs are trained until the discriminator can't distinguish between the original samples and the samples generated by the generator. This leads to very realistic samples. At the start of this thesis in 2019, GAN research had matured for six years, mostly focusing on synthesising images. The high fidelity and fast generation, compared to other statistical or machine learning approaches, makes them the preferred candidate for the problem set by DCWW.

2.4.2 Related work and motivation for the exploration of machine learning approaches to data synthesis

This brief overview shows that statistical-based data synthesis has been successful to individually address the key data challenges posed by the DCWW multivariate datasets summarised in Chapter 1: missing data, time series, unbalanced data (section 2.2). There has also been some work to use statistical-based data synthesis to address some of these challenges in combination, for example imputation approaches to solve the challenge of missing values for time series (2.3.3).

While data synthesis using statistical approaches outlined in the section 2.3 have been successful, and have the advantage of being relatively easy to compute, they tend to be less flexible than machine learning approaches [37]) since statistical models often require a range of specific assumptions. When datasets or models are complex, as in the DCWW datasets, the flexibility offered by machine learning approaches like GANs offers a promising avenue to synthesise data in a way that could address together all the key challenges of the DCWW data detailed in section 2.2. In 2019, at the start of this thesis, GANs had been mostly developed for images, including the code, and no GAN was able to generate synthetic data with missing values. Moreover, none could deal with all the key challenges of DCWW data.

This provides the motivation to explore the relevance of GANs to address the challenges highlighted in Section 2.3. The next Chapter further explains why GANs were chosen as the preferred machine learning data synthesis investigation route.

Chapter 3

Background on Generative Adversarial Networks

As identified in the previous Chapter, data synthesis based on machine learning is likely to be well-suited to address the objectives set out in this thesis (see Chapter 1). This Chapter sets out the motivation for exploring the relevance of GANs, as a promising machine learning data synthesis approach. The chapter starts by focusing on the building blocks of GANs (section 3.1), what applications these GANs have been used for (section 3.2), how GANs work (section 3.3), improvements to the GAN original architecture (section 3.4,3.6 and 3.5), and how GANs can be evaluated (section 3.7). The Chapter then explores how GANs can be used in an applied framework such as the one described with DCWW (section 3.8).

3.1 Building blocks: Neural Networks

Artificial Neural Networks, NN for short, are structured as networks of **Nodes**, where nodes can be thought as neurons in a brain [38]. Nodes are connected to each other through a series of links, in the simlar fashion as neurons are connected in the brain. Each link has an attached **weight** which is specific to its two nodes, and expresses the importance of the information coming from that link. NN structure can vary, but their nodes usually perform the following operations (Figure 3.1):

$$a(\sum_{i=0}^{n} x_i\theta_i) \tag{3.1}$$

where in the vector of input $(x_1, ..., x_n)$, the set of weights $(\theta_0, ..., \theta_n)$ and $a()$ is the activation function.

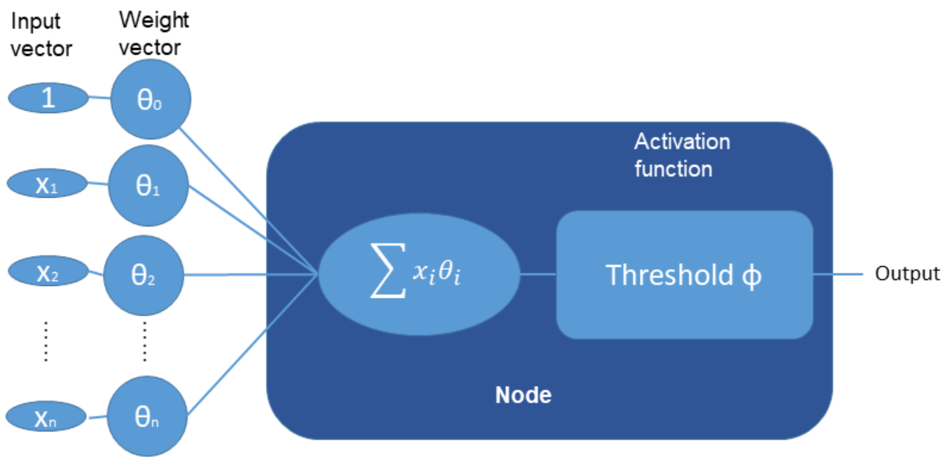


Figure 3.1: A singular node

The information treated by each node can come from the initial data input or from the outputs of other nodes. This sum includes a bias ($x_0 = 1$), which is similar to a constant for a linear function as it increases the flexibility of the node function. The sum is then passed through an activation function which gives a single value output. There are many activation functions to choose from, for example:

- The logistic function:

$$a(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

- Tanh is similar to the logistic function, but the range extends over -1, +1 instead of 0,1

$$a(x) = \frac{2}{1+e^{-2x}} - 1 \tag{3.3}$$

- ReLu stands for rectified linear unit, and is a piecewise function were all negative values become 0

$$a(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{3.4}$$

Softmax node, often used in Hidden layers, is based on a logistic function, and works better with multiple inputs than with normal logistics

$$a(\mathbf{x}) = \frac{e^{\mathbf{x}_i}}{\sum_{j=0}^{n} e^{\mathbf{x}_j}} \tag{3.5}$$

where $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$.

In artificial neural networks, 'Input' nodes take information into the network, the information can then go through a network of 'Hidden' nodes, before exiting the network through the 'Output' nodes that compute the final output of the network [38]. More generally a Neural Network can be represented by a function that is parametrised by a weight (θ), noted as $N(\mathbf{x}; \theta)$ where $\mathbf{x}$ is the inputs to the network, and θ the weights of the network.

The nature of the NN is determined by the way that each node is linked to the others. The most basic of NN is a **feed-forward** network structure (Figure 3.2) where there is a direct flow of information from input nodes to output nodes [39]. In such a network, the links between nodes are structured in layers (acyclic network). Feed-forward NN are mostly used for classification or regression problems. As an example, NN could be used to predict rain (Figure 3.2): the network would be given as input information several variables (for example, air temperature, humidity, wind direction, wind speed). The information flows through the network and is weighted and summed at each node, then, it runs through an activation function before flowing to the next node. The process is repeated from node to node until it reaches the output layer where it suggests if it will rain or not.

The **loss function** assesses the difference between the current output of the model and the 'desired' output of the model. The performance output is used to train the network, and this is done by optimising the set of weights to reduce loss. Optimisation

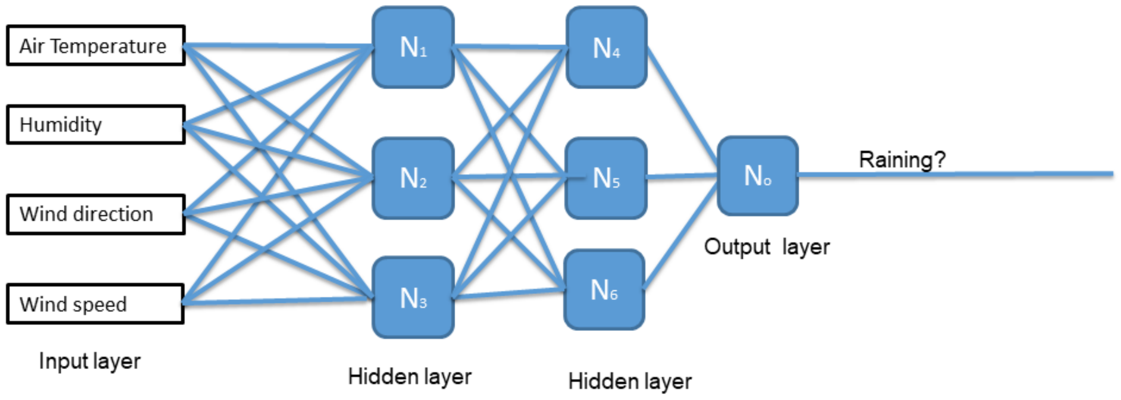


Figure 3.2: An example of a simple feed-forward neural network to deduce rainfall. Information flows from input to the output through layers of nodes (N).

is mostly done by calculating the gradient of the loss function using techniques like **Backpropagation** algorithms [40].

3.2 The advent of GANs

The concept of GAN was first published by Goodfellow in 2014 [34]. At the start of this thesis, a literature review done on the 12th of December 2019 with the search terms (Generative adversarial network* OR GAN) showed 2,317 papers on the Web of Science database. The large majority (76%) of these were focused on images. Indeed, GANs were first developed for images, and many applications in this field have become well-established. As an example, GANs are used to alter the style or texture of images [41], to increase image resolution [42].

Over the course of the thesis, the number of publications in this field has significantly increased, with a total on the 1st September 2023 of 84,389 articles on the Web of Science database with the same search terms. This highlights the worldwide interest in GAN applications. This has been particularly true for GAN applications not focused on images, such as time series, data privacy and numerical databases, which in September 2023 constituted circa 12% each of GAN publications.

3.3 The GAN architecture

3.3.1 The basics of GANs

The idea behind Generative Adversarial Networks (GANs) can be conceived as a game [34]. The 'game' of GAN can be seen as a game where two players, G (the Generative Neural Network) and D (the Discriminative Neural Network), take turns trying to follow opposite objectives.

The objective of G is to replicate an original data set and create a synthetic dataset in all ways similar to the original. The opposed objective of D is to discriminate between the original dataset and a synthetic dataset generated by G.

Each Neural Network adjusts, in turn, their weights to achieve their objective. Each round, commonly referred to as an 'epoch' in the GAN literature [34], involves the following sequence:

1. G adjusts its weights to get closer to its objective of replicating the original data distribution.

2. D adjusts its weights to get closer to its objective of classifying correctly original and synthetic data.

3. At the end of each epoch, the overall losses are calculated, using the cross-entropy loss for D and the outcome of D as the loss for G.

4. The players use the overall losses to help them adapt their weights in the next round, and the game continues.

After a pre-determined number of epochs, the game converges towards three possible outcomes:

1. The Discriminator can classify some (about 50%) of the data. This outcome is referred to as a joint-equilibrium [43], and it is considered the best outcome.

2. The Discriminator can classify all the data and dominate/win the game which means that the Discriminator does not give any useful information to the Generator

as it always detects if the input is synthetic. This outcome is called the **vanishing gradient** outcome or **overfitting**.

3. The Discriminator cannot classify any data, and the Generator dominates/wins the game. This outcome is called **modal collapse** because the Generator always produces the same synthetic data that fools the Discriminator and is, therefore, unable to produce any more realistic data.

3.3.2 The GAN optimisation process

More broadly, when models are put into such a contest game, this is called **adversarial machine learning** and can be used with other models than Neural Networks (NN) [44]. The aim is to train the two models alternatively, so they try to be better than the other, but none of them wins (joint equilibrium outcome). This process is referred to as the 'GAN optimisation'. As the training progresses, the Discriminator gets better at discriminating the characteristics of the original distribution and the synthetic (generated) distribution, and the Generator uses the feedback to get better at generating data with characteristics that the Discriminator does not discriminate.

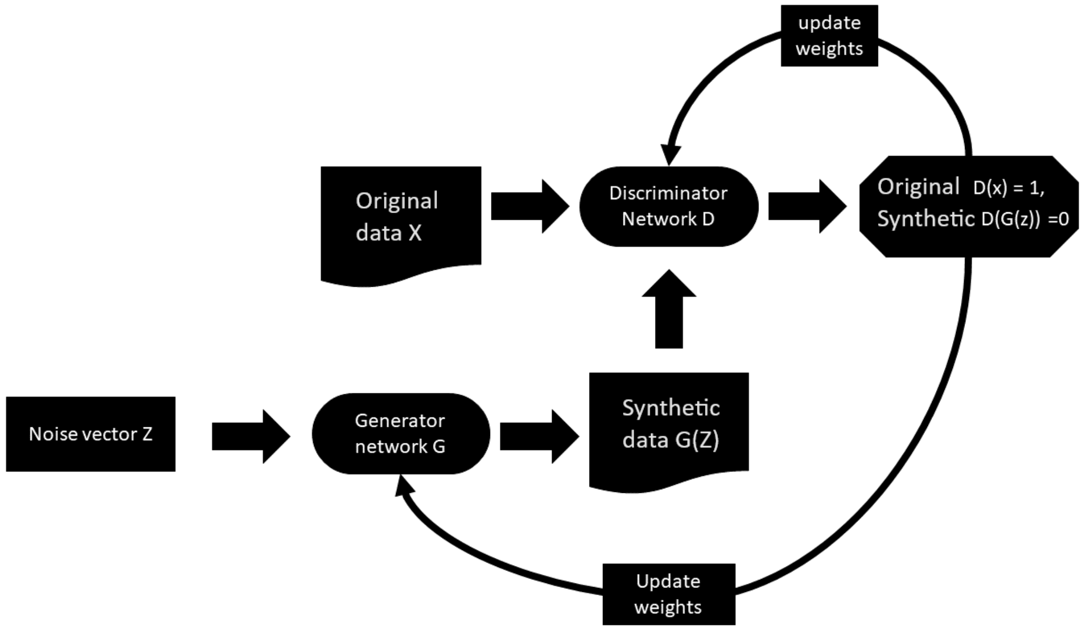


Figure 3.3: Flowchart of GAN training

Training a GAN is difficult because there are two models to train. The process is broadly outlined in Figure 3.3.

1. The Generator G is a generative NN. The Generator is first partially trained to generate synthetic data that looks similar to the original. It generates this synthetic data ($G(z)$) from a random noise vector (z) input.

2. The Discriminator is then partially trained to discriminate the current learned distribution from the original. The Discriminator is a feed-forward neural network (see section 3.1) that uses as input both the original data (X) and the synthetic data ($G(z)$). The Discriminator network is built with a sigmoid activation function (see section 3.1) for the output layer and thus provides a probability output with a probability of original as 1 and probability of synthetic as 0.

3. The training process includes a loss function that provides a performance measure and determines how the weights of the G or D need to be updated. The loss function is described in the previous section (|refnns).

4. At the start of each epoch, the weights of the two models are updated each in turn, based on the loss.

5. Ideally, through continous monitoring, the GAN is trained until the Generator and Discriminator converge. In practice it is very hard to contiuously monitor, so a pre-determined number of epochs is generally selected. At this predetermined number of epochs, convergence is checked, and if not realised, a higher number of epochs is selected. Therefore, the pre-determination of the number of epochs is a key tuning parameter of GANs. Experience shows that when the number of epochs chosen is too small the GAN will be insufficiently trained, and when too large tends to result in one of the poorer outcomes (ie overfitting or modal collapse). Those poorer outcomes as also referred to as model instability.

3.3.3 In mathematical terms

The below describes in simple mathematical terms the GAN structure and optimisation process.

The Generator $G(z;\theta_g)$ is a function that takes z from a latent space$U(0,1)^n$ so that $z = (z_1, z_2, ..., z_n)$, and that then maps to R^n, where θ_g are the weight parameters (see section 3.1). The Discriminator $D(\gamma;\theta_d)$ is a function that takes an input $\gamma \in R^n$ and outputs a single scalar [0,1], that is the probability that $\gamma \in X$, where X is the original dataset, γ is the input (either from the synthetic or the original dataset), and θ_d are the weight parameters of D [34]. $D(\gamma;\theta_d)$ measures how likely γ is to be the realisation of the data distribution. For the loss function of the Generator and the Discriminator networks, the binary cross-entropy is used which expresses the difference between the original and the synthetic probability distributions:

$$\frac{1}{n}\sum_{i=1}^{n} y_i \log(p_i) + (1-y_i)\log(1-p_i) \tag{3.6}$$

where y is the class of data (y=1 for original data and y=0 for generated data), p is the probability that the data belongs to class y and n is the number of data.

In the case of the Generator, the aim is to train it so that synthetic data $G(z)$ is discriminated as original (y=1). The binary cross-entropy L_G for the Generator is:

$$\frac{1}{n}\sum_{i=1}^{n} \log(D(G(z_i;\theta_g);\theta_d) \tag{3.7}$$

In the case of the Discriminator, the aim is to train it so that: i) the synthetic data $G(z)$ is treated as synthetic (y=0),

$$\frac{1}{n}\sum_{i=1}^{n} \log(1 - D(G(z_i;\theta_g);\theta_d) \tag{3.8}$$

ii) the original data X is treated as original (y=1).

$$\frac{1}{n}\sum_{i=1}^{n} \log(D(x_i);\theta_d) \tag{3.9}$$

The cross-entropy loss for the Discriminator L_D is therefore (3.8) + (3.9):

$$\frac{1}{n}\sum_{i=1}^{n}\log(D(x_i;\theta_d) + \frac{1}{n}\sum_{i=1}^{n}\log(1 - D(G(z_i;\theta_g);\theta_d)) \tag{3.10}$$

Each of the networks is trained in turn with its corresponding loss function described above. Sometimes, the Discriminator is not performing as well as the Generator, so it requires more training loops than the Generator. This happens when the Discriminator starts to struggle with categorising the data generated by the Generator. This is identified by looking at the loss graph which shows the progress of the networks. Lack of progression by the Discriminator is characterised by a G loss that improves while the D loss stays flat.

GAN pseudocode

Below is the pseudocode for the first GAN developed by Goodfellow et al ([34]), and commonly referred to as Vanilla GAN. This pseudocode, transcribed nearly as written by the author, constitutes the core pseudocode on which the rest of the thesis is built.

```
Require: initial Generator weights $\theta_G$ and Discriminator weights $\theta_D$
Require: num. epochs $t_G$, Discriminator iterations $t_D$, batch size $\beta$.
 1: for $s = 1, \dots, t_G$ do
 2:     for $t = 1, \dots, t_D$ do
 3:         sample a batch of size $\beta$ from $\{1, \dots, n\}$
 4:         sample $z \sim U(0,1)^n$
 5:         $L_D \leftarrow \frac{1}{n}\sum_{i=1}^{n} \log(D(x_i;\theta_d) + \frac{1}{n}\sum_{i=1}^{n} \log(1 - D(G(z_i;\theta_g);\theta_d))$  ▷ calculate
       Discriminator loss $L_D$
 6:         $\theta_C \leftarrow \theta_C +$ opti$(\theta_C, L_C)$          ▷ update Discriminator weights
 7:     end for
 8:     sample $z \sim U(0,1)^n$
 9:     $L_G \leftarrow \frac{1}{n}\sum_{i=1}^{n} \log(1 - D(G(z_i;\theta_g);\theta_d)$          ▷ calculate Generator loss $L_G$
10:     $\theta_G \leftarrow \theta_G +$ opti$(\theta_G, L_G)$          ▷ update Generator weights
11: end for
```

opti() is any optimiser function

3.4 WGAN and WGAN-GP

To address both of GAN's biggest problems: overfitting and modal collapse, two forms of model instability, WGAN models were developed. Arjozky et al. 2017 [45] proposed a new loss function based on the Wasserstein distance to mitigate these problems. The Wasserstein distance, also known as earth-mover, is calculated as the minimum cost of transforming one distribution into another (like moving earth from one mound to another). This measures the distance between two different distributions. In the case of a GAN, the Wasserstein distance is used to measure the distance between the original and synthetic distributions as a Discriminator would do in a classic GAN. The WGAN is optimised when the distance between the original and synthetic data is minimal (approaches zero). An advantage of this approach is that it is thus possible to have a measurement of how close the model is to the real data, as well as the generated data quality.

Let $\mathcal{P}$ and $\mathcal{Q}$ be probability measures on $\mathbb{R}^d$, and $\Pi(\mathcal{P}, \mathcal{Q})$ the set of measures on $\mathbb{R}^d \times \mathbb{R}^d$ with marginals $\mathcal{P}$ and $\mathcal{Q}$, then the Wasserstein distance between $\mathcal{P}$ and $\mathcal{Q}$ is

$$W(\mathcal{P}, \mathcal{Q}) = \inf_{\Gamma \in \Pi(\mathcal{P}, \mathcal{Q})} \mathbb{E}_{(X,Y)\sim\Gamma} \|X - Y\|_2. \tag{3.11}$$

The Kantorocich-Rubinstein duality [46, 47] gives an equivalent formulation:

$$W(\mathcal{P}, \mathcal{Q}) = \sup_{f \,:\, \|f\|_L \leq 1} \mathbb{E}_{X\sim\mathcal{P}} f(X) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y). \tag{3.12}$$

Here the supremum is over continuous functions $f : \mathbb{R}^d \to \mathbb{R}$ with Lipschitz constant at most 1. That is, f satisfies

$$\sup_{x,y\in\mathbb{R}^d} \frac{|f(x) - f(y)|}{\|x - y\|_2} \leq 1.$$

In this setting $\mathcal{Q}$ will be the measure on $\mathbb{R}^d$ induced by the generator, and $\mathcal{P}$ the measure on $\mathbb{R}^d$ from which the data was sampled. Let $\{x_i\}_{i=1}^n$ and $\{y_i\}_{i=1}^n$ be samples from $\mathcal{P}$ and $\mathcal{Q}$, then approximating $\mathcal{P}$ and $\mathcal{Q}$ with the discrete measures implied by their respective samples, the RHS of (3.12) becomes

$$\sup_{f \,:\, \|f\|_L \leq 1} \frac{1}{n} \sum_{i=1}^n f(x_i) - \frac{1}{n} \sum_{i=1}^n f(y_i). \tag{3.13}$$

Let $G = G(z; \theta_g)$ be our *generator*, with inputs $z \in (0,1)^d$ and parameters θ_g, $y_i = G(z_i; \theta_g)$ can be taken where z_i is a sample of i.i.d. $U(0,1)$ r.v.s. Approximating f with a net C parameterised by θ_c, (3.13) now becomes

$$\sup_{\theta_c \,:\, \|C(\cdot;\theta_c)\|_L \leq 1} \frac{1}{n} \sum_{i=1}^n C(x_i; \theta_c) - \frac{1}{n} \sum_{i=1}^n C(G(z_i; \theta_g); \theta_c). \tag{3.14}$$

C is called the *critic*.

The constraint $\|C(\cdot; \theta_c)\|_L \leq 1$ is not straight-forward. Arjozky et al. [45] used an ad-hoc method they termed "weight clipping" to try and enforce the constraint, whereby the parameters θ_c are constrained to the interval $[-\lambda, \lambda]$ for some factor λ. Gulrajani et al. [48] found that weight clipping is very sensitive: if λ is too high then the gradient can vanish, and if it is too low the convergence is too slow. In addition weight clipping can be unstable when using momentum-based optimisers. Therefore they proposed replacing the Lipschitz constraint with a gradient penalty, so that (3.14) becomes

$$\sup_{\theta_c} \frac{1}{n}\sum_{i=1}^{n} C(x_i;\theta_c) - \frac{1}{n}\sum_{i=1}^{n} C(G(z_i;\theta_g);\theta_c) - \lambda\frac{1}{n}\sum_{i=1}^{n}(\|\nabla C(h_i;\theta_c)\|_2 - 1)^2, \quad (3.15)$$

where λ is a user chosen penalty scaling, and $h_i = t_i x_i + (1-t_i)G(z_i;\theta_g)$ for $t_i \sim U(0,1)$. Note that here the gradient of $C = C(z;\theta_c)$ is taken w.r.t. the input z and not the parameter θ_c.

Like the Vanilla GAN, a noise vector goes through the Generator to create synthetic data. Then, an average of the original and synthetic data is calculated with a random weight, and the output will be the interpolated data. All 3 data sets (original, synthetic, interpolated) are put through the Critic. The three datasets are used to calculate the Critic distance, and part of this distance is used to update the weights of the Generator. Arjozky et al. [45] suggest performing, as a base, more iterations of the Critic per Generator iteration. This improves the calculations of the gradients used to train the Generator, so having a well-trained critic reduces the risk of modal collapse (see section 3.3). The gradient penalty prevents the critic from overfitting and balances the system.

Below is the pseudo-code for WGAN-GP as written by Gulrajani et al [48] but with a slightly modified notation to facilitate comparison with WGAN:

```
Require: initial Generator weights $\theta_G$ and critic weights $\theta_C$, learning rate $\alpha$
Require: num. epochs $t_G$, critic iterations $t_C$, batch size $\beta$, critic regularisation $\lambda$
 1: for $s = 1, \ldots, t_G$ do                                        ▷ update the Generator
 2:     for $t = 1, \ldots, t_C$ do                                    ▷ update the critic
 3:         choose a batch $\sigma$ of size $\beta$ from $\{1, \ldots, n\}$
 4:         for $i = 1, \ldots, \beta$ do                              ▷ calculate critic loss
 5:             sample $u \sim U(0,1)^d$
 6:             sample $\epsilon \sim U(0,1)$
 7:             $h_i \leftarrow \epsilon x_i + (1-\epsilon) y_i$
 8:             $L_C^i \leftarrow C(x_i) - C(y_i) - \lambda(\|\nabla C(h_i)\|_2 - 1)^2$
 9:         end for
10:         $L_C \leftarrow \frac{1}{\beta} \sum_{i=1}^{\beta} L_C^i$
11:         update $\theta_C$ using gradient of $L_C$ (increasing $L_C$)
12:     end for
13:     for $i = 1, \ldots, \beta$ do                                  ▷ calculate Generator loss
14:         sample $u \sim U(0,1)^d$
15:         $L_G^i \leftarrow C(G(u))$
16:     end for
17:     $L_G \leftarrow \frac{1}{\beta} \sum_{i=1}^{\beta} L_G^i$
18:     update $\theta_G$ using negative gradient of $L_G$ (decreasing $L_G$)
19: end for
```

3.5 Adapting GANs for time series

Time series exhibit dependencies between consecutive observations. Classic GANs such as Vanilla GAN or WGAN sample distributions as independent and therefore are not designed to take into account these dependencies. The outcomes cannot be considered as time series.

3.5.1 Types of GANs for time series

GANs that are currently designed for time series use either recurrent layers or convolutional layers to capture the dependencies in time series.

A **Recurrent layer** is a layer which 'remembers' previous observations, it does this by saving the output of the layer and feeding this back to the input in order to predict the output of the same layer. and this captures the time dependency. This layer can be inserted into a GAN, so the GAN can be used for time series. Examples include: C-RNN-GAN by Mogren 2016 [49], Recurrent GAN (RGAN) and Recurrent Conditional GAN (RCGAN) by Stephanie et al [50].

Convolutional layers are usually used in image processing as they capture relationships between nearby pixels so they can also be used to capture time dependency. Applied to time series, this means convolutional layers can capture the relationship between nearby time steps, this is done by compressing the infomation by dot product of a subsequence and the weights of the layer. The subsequence is shifted by one until it has processed all the sequence [51]. A good illustration is AnoGAN [52] which uses a deep convolutional generative adversarial network to analyse anatomical anomalies in medical images through time.

There are some GANs that are composed of a combination of these layers to create the best outcomes. Examples include: LSTM-FCN cGAN by Maximilian Ehrhart et al [53], which uses a mix of Convolutional and Recurrent layers.

3.5.2 Yoon's Time-GAN

The latest addition to the GAN models capable of dealing with time series is Time-GAN. Yoon's Time-GAN has 4 networks: an encoder, a decoder, a Discriminator, a Generator [54]. The GAN operate as follows: First, data is mapped to a feature space using the encoder. A **Feature Space** is the space that contains features to reduce the complexity of datasets, namely time series. For example this can be done using a dimension reduction algorithm or encoder network. A feature space is a lower-dimensional representation of the real world and this enables the GAN to learn the underlying temporal dynamics of the data, and separating the signal from the noise. The GAN, which contains a Generator and a Discriminator, is then trained in this feature space. This GAN contains Recurrent layers such as those found in RGAN [50]. Once the GAN is trained, the Generator can create data in the feature space which can be mapped to the real space with the decoder. This GAN uses 3 losses: the Embedding loss for the encoder and decoder, the Supervised Loss for the generator to learn time dependency, and the Unsupervised loss for the overall distribution. Training, detailed in [54], can be summarised into 3 phases:

1. training the encoder and decoder with the Embedding loss;
2. training the generator with the Supervised Loss only;
3. training the encoder, decoder, generator and discriminator together.

This GAN is later utilised in Chapter 7 to synthesise time series.

3.6 Adapting GANs for Anonymisation

This section briefly explores the potential of GAN for Anonymisation since this was one of the interests of the sponsor DCWW. As a reminder (see section 3.3), GANs generate synthetic data using the learned distribution of the original data set rather than the original dataset itself. This could mean that the likelihood of relating this synthetic data to the original data is low, unless perhaps there is a probablity that the GAN can detect points within clsutered data. Also, compared to other ways of synthesizing data, GANs retain the distribution of the original dataset and therefore analysis of the synthetised data set should yield similar results to the original dataset. As a consequence, the hypothesis is that GANs could provide a secure way to anonymise data.

There has already been some limited exploration of the potential for using GAN to anonymise data for different purposes [55]. In common with other approaches, one of the most significant issues to resolve when using GANs, is the way that the distribution of the generated synthetic data could be concentrated on some data points and this concentration could therefore reveal information on the original data points. For example, the centroid of aggregated synthetic data points could reveal one of the original training points [56].

Xie et al. [56] tried to solve this issue by creating an add-on to the GAN algorithm that would reduce the risk of generating synthetic data points that were too aggregated around original points (Differentially Private GAN, DP-GAN). It achieved this by adding noise to the gradient of the Discriminator during training to avoid data aggregation. It was considered a benchmark in anonymising data using GANs after being successfully used to anonymise patient medical reports.

Jordon et al. [57] then tried using Private Aggregation of Teacher Ensembles (PATE) as an add-on to the GAN, to tightly bound the influence of any individual sample on the model. They suggested that PATE-GAN consistently produced synthetic datasets that were more realistic than the synthetic datasets produced by the benchmark DP-GAN, and were able to test the performance of PATE-GAN on continuous, discrete, and binary variables. They also concluded that extending the PATE framework to the regression setting, for example using the recently developed Wasserstein GAN (WGAN) could be promising. One of the issues of both DP-GAN and PATE-GAN algorithms is that they work by adding noise during the training process and this reduces the quality of the synthetic dataset.

While writing this Chapter, Yoon et al. [58] also developed a new GAN (ADS-GAN) for anonymising numerical patient data using WGAN-GP combined with a conditional GAN framework [59]. However, this model was only developed for numerical and binary data. Also, another challenge with ADS-GAN is that, at the optimisation step, ADS-GAN adds a penalty that uses 'discrete entropy' to increase anonymisation. However, as this penalty is discrete, in order to make ADS-GAN work for continuous features, ADS-GAN quantizes these features in discrete chunks. This rounding is likely to make the training calculations less accurate.

3.7 Evaluating the GANs

3.7.1 Purpose

Some models perform better than others, but there is no one best way of evaluating the trained GAN. A good evaluation measure should at least be able to show the difference between real and synthetic distributions, and also show whenever the model is in modal collapse or vanishing gradient mode. However, there are many metrics used depending on what the GAN is used for. Borji [60] review 29 different quantitative and qualitative ways of evaluating GANs and propose a list of 7 criteria to compare them. There is still research needed to assess their strength and limits. Among those, the most adapted metrics to this thesis included Fréchet distance and the Likeness score. These were thus selected for this thesis.

3.7.2 Fréchet distance

The Fréchet distance F is based on the Wasserstein-2 distance and uses the estimated mean μ and covariance $\mathcal{C}$ of the generated data y and the real data x [61]. $F^2 = \|\mu_x - \mu_y\|^2 +$ Tr$(\mathcal{C}_x + \mathcal{C}_y - 2(\mathcal{C}_x\mathcal{C}_y)^{\frac{1}{2}})$, where μ_x and $\mathcal{C}_x$ are the sample mean and sample covariance matrix of $x_1, \ldots, x_n$, and similarly for μ_y and $\mathcal{C}_y$. Tr is the trace of the matrix.

The scale is well bounded; a number close to 0 is very similar to the original and a number that tends to infinity is very different. It is not complex to compute and can handle most noise in the data well. In this thesis, the Fréchet distance was used to evaluate GANs by calculating the distance between the generated and original data. Well performing GANs showed distances closer to zero. Note that for images it is common to calculate F not using the original and synthetic data directly but instead by first applying a feature extracting transform [62]. As the thesis is focused on tabular data, this was not necessary.

3.7.3 Likeness score

The Likeness score was introduced by Shuyue Guan and Murray Loew [63]. This score is built from a set of three distance measures which are measured independently and in combination. The combined score provides an assessment of the overall performance of the GAN. Comparison between the individual scores that contribute to the combined score provide more specific information on how the GAN performs.

Mathematically, given $x_1, \dots, x_n$ as the original data observations and $y_1, \dots, y_m$ as the synthetic data observations, to calculate the Likeness score L, Shuyue Guan and Murray Loew [63] first calculate the three individual distances:

- the set of distances between any two points in the original data $S_x = \{\|x_i - x_j\|_2\}_{i \neq j}$ (original intra-class distance)
- the set of distances between any two points in the synthetic data $S_y = \{\|y_i - y_j\|_2\}_{i \neq j}$ (synthetic intra-class distance)
- the set of distances between any point in the synthetic data and any point in the original data $S_{x,y} = \{\|x_i - y_j\|_2\}_{i,j}$ (between-class distance)

The likeness score is expressed as:

$$L = 1 - \max\{\kappa(S_x, S_{x,y}), \kappa(S_y, S_{x,y})\}.$$

where κ is the Kolmogorov-Smirnov distance and $S_x, S_y, S_{x,y}$ are vetcors Note that $L \in [0, 1]$ and the two sets of observations have a likeness score of 1 if and only if they are identical, with lower scores indicating greater dissimilarity. The score components inform on what the GAN is lacking:

- the GAN can be described as lacking Creativity when the synthetic data are very close to the original,
- the GAN can be described as lacking Diversity when the synthetic data distribution appears in clusters

- the GAN can be described as lacking Inheritance when the synthetic data distribution is not in the same range as the original,

The score components are described in further detail in [63].

3.8 In practice: Coding and testing the GAN

3.8.1 Coding the GANs

To use the GAN algorithm, python codes were first developed for the three different generalist GANs described in section 3.4: GAN(Vanilla), WGAN, WGAN-GP.

For this thesis, code was done in Python using the PyTorch library [64]. The structure of this library is closer to a pseudo-code and can therefore be more easily modified. As all the WGANs at time of writing were coded for the treatment of images, the flexibility of this library was needed so that it could adapt exiting codes to run on tabular data instead of images. Code sources include:

- for Vanilla GAN from [65];
- for WGAN and WGAN-GP from [66].

However, both these codes were developed for images, and therefore the code had to be adpated to tabular data by replacing the Convolutional layers, which are only adapted to images, with Regular feed-forward layers that are more adapted to any type of input. Regular feed-forward layers are the default layer type for tabular data. Codes for the WGANs are detailed in Appendix A.2

3.8.2 Testing the GANs

To establish which GAN algorithm architecture was the most useful to achieve the objectives set out in Chapter 1, three different generalist GANs (GAN(Vanilla), WGAN, WGAN-GP) were tested on a simple dataset. The Iris dataset [67] is a simple dataset with four variables that record the length and width of the sepals and petals of irises. The

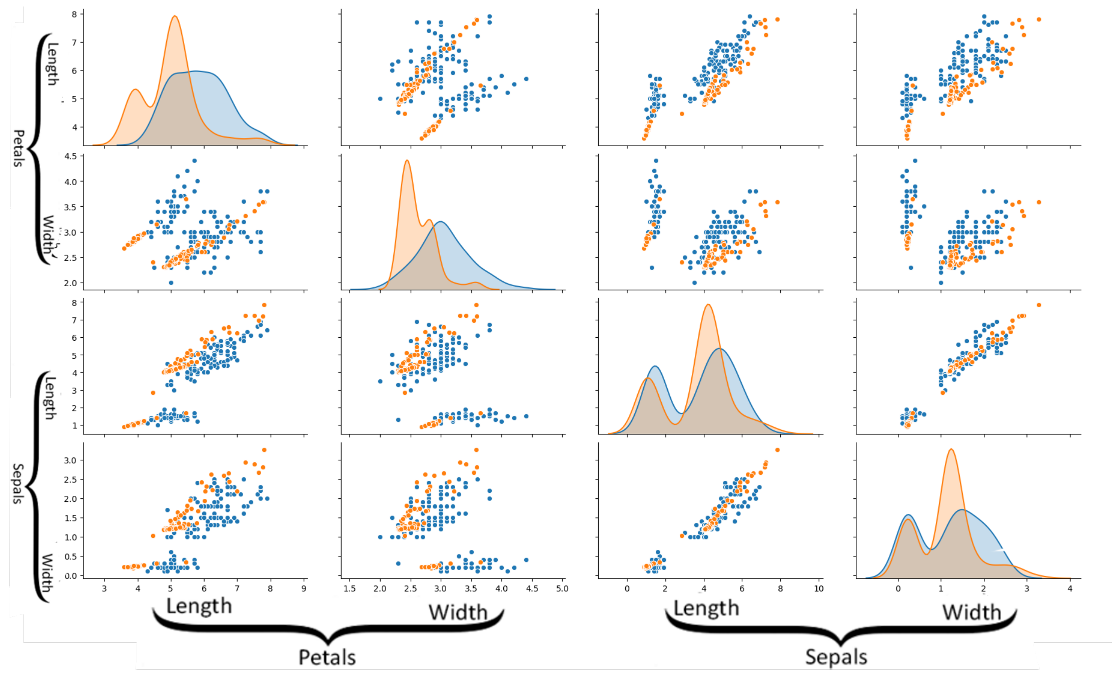


Figure 3.4: Comparison of the original Iris data (in blue), and the one generated by the Vanilla GAN (in orange)

original dataset only contains data of 150 flowers. This is a simple multivariate tabular dataset that provides a first insight into the relative performance of the three GANs.

A typical Data Augmentation problem (see section 2.3.4) was used to test the GANs and resulted in the creation of another 150 flowers. All the GANs were trained on 17000 epochs to ensure the tests were comparable, this pre-determined number of epochs was chosen to ensure none of the GANs overfitted or collapsed. Comparisons were made visually first using a pair plot of generated and original data distributions and then checked using the FD score described in section 3.7.2, where the lower the FD score, the better the performance of the GAN.

The code for Vanilla GAN contained 4 layers for the Generator and 4 for the Discriminator. Figure 3.4 shows the distributions of the original and generated data along the 4 variables in the dataset (sepals and petals, length and width). The graphs show that the generated data points are clumped rather than distributed in a simlar way to the original, and fit is confirmed by a FD score of 1.001.

For the WGAN test, choosing a good weight clipping variable is done by trial and error. After some experimentation, the WGAN with a weight clipping of 0.2 gave the lowest

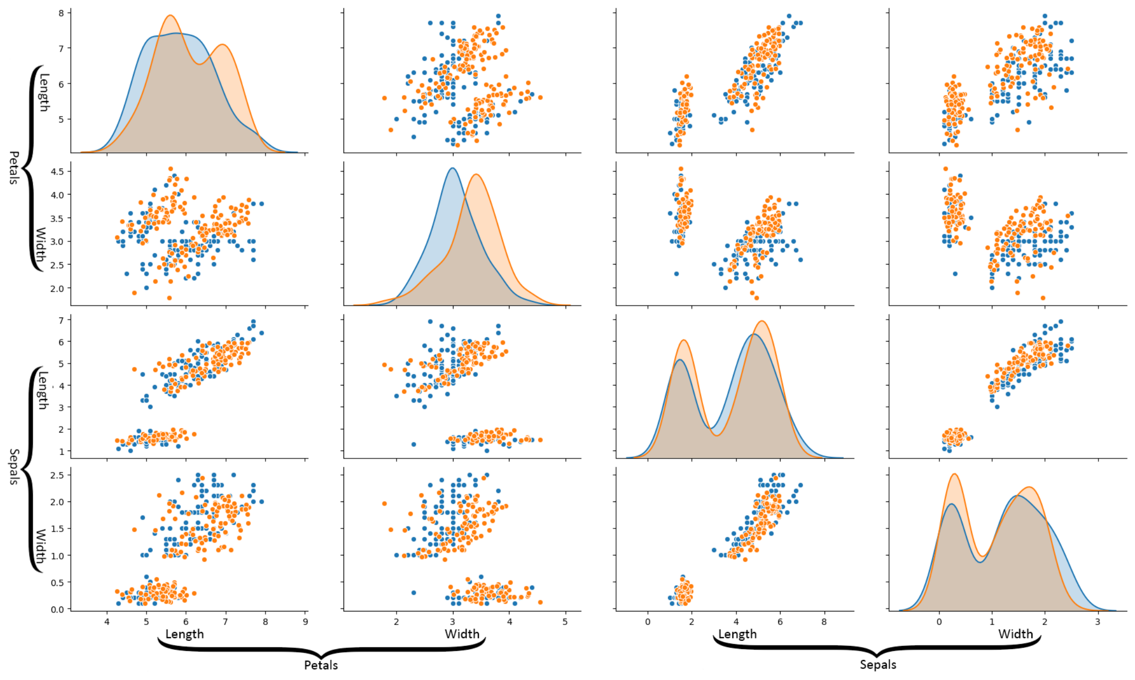


Figure 3.5: Comparison of the original Iris data, and the one generated by the WGAN with a weight clipping of 0.2

FD score for the Iris dataset Figure 3.5. The results in Figure 3.6 show a better overlap between original and generated data distributions than Figure 3.4. This is confirmed by the lower FD score of 0.189 that is closer to 0, where values closer to zero are seen as better.

For WGAN-GP the FD score was 0.010 and this was a high performance score by comparison with the WGAN score of 0.189, and the GAN score of 1.001.

This exploratory section on the relative performance of the three different GANs on a simple multivariate tabular dataset suggested that WGAN-GP was the best candidate for the thesis since the data mentioned in Chapter 1 and analysed in Chapter 7, although more complex, were also multivariate tabular data.

In summary, compared to the simplest form of GAN (Vanilla GAN - Section 3.3), Figures 3.4, 3.6 and 3.5 shows WGAN-GP performed the best.

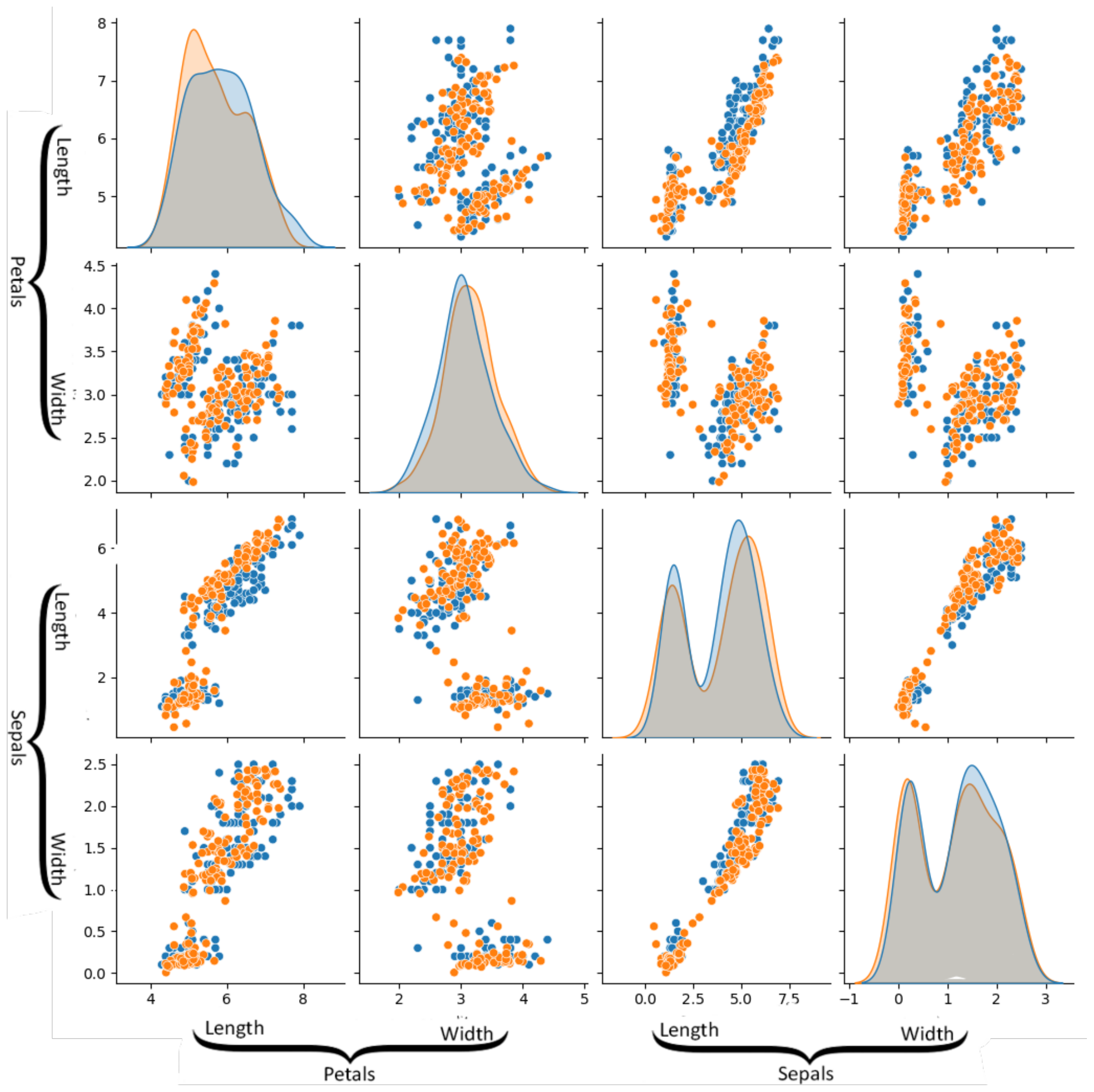


Figure 3.6: Comparison of the original Iris data (in blue), and the one generated by the WGAN-GP (in orange) with a λ of 10

3.9 Using GANs in a business context

3.9.1 Limitations in a business context

One of the limitations of GANs is that they are challenging to work with and require specialist expertise given the level of computation and fine tuning necessary when using GANs. This means that the solution proposed here cannot easily be realised in the organisation that needs it, and the data, therefore, needs to be transferred to a specialist data organisation. This can be an issue when dealing with sensitive data.

To address this issue, an interface was developed for the GAN to make it more accessible to non-specialists. While this is helpful, it is important to keep in mind that multiple parameters still need to be optimised, and therefore, staff need to be trained to be able to use this solution effectively. The interface created addresses the main challenges encountered by non-specialists. The first challenge is to make sure the data are all in the right format, and this includes being able to input any tabular data type, including categorical data (a characteristic that is not well supported by neural networks). The second challenge is to create the GAN so that all the networks are properly constructed. The next challenge when training the GAN is to be able to get the GAN working, as there are different components of the algorithm that need to be used in a specific sequence (one script that trains, one script that generates data, one script that saves/loads the model for further use). The last challenge is to revert the generated data back into a usable format.

While all the tasks described here have straightforward solutions, a choice had to be made for the challenge of handling categorical data. There are three ways of handling categorical data [68]:

Ordinal Encoding: This is the most basic way of handling categorical data. For example, apples could be coded as 1, pears as 2, oranges as 3. The issue with this type of encoding is that it orders the data, even when there is no order.

One Hot Encoding: This puts data in a matrix, where each categaories is defined though a set of binary sequences. For example, apples would be encoded as 001, pears as 010, and oranges as 100. The advantage of this method is that it handles missing data very well because missing data can be represented as 000, for example. There are, however, two issues with this method. First, all possible outcomes need to be known beforehand, as this constrains the matrix size. Secondly, this method uses a lot of memory when categories have lots of unique values. A last minor issue is that the potential relationship between categories cannot be represented; for example, apples and pears might both be green.

Embedding Categorical data: This method puts data in a matrix that also contains semantic meaning, this means it stores relationships between categories. Embedding provides two advantages. First, the number of columns per category is limited. Second, embeddings by nature group similar variables together. A disadvantage, however, of this method is that it does not deal very well with missing data. For this interface, One Hot Encoding is chosen as it handles categorical data well, and the missing numerical data can be deleted, or imputed using other techniques.

Below is a detailed overview of how the code was therefore structured to address all the challenges. The code (see Appendix A.1) does the following actions:

1. Data import: Load the database into Python

2. Data preparation: For numerical values Normalises by performing:

$$n_{ij} = \frac{d_{ij} - \mu_j}{\sigma_j} \tag{3.16}$$

   where n is the normalised data, d is the database with row i and column j, μ is the mean, and σ is the standard deviation of the column. For categorical values, use One Hot Encoding for each separated categorical variable and encode to each data point

3. GAN creation: Creates the GAN in 3 steps: building Discriminator, the Generator, and compiling the model ready to run

4. Training GAN: Trains the GAN as explained in section 3.3.2

5. Performance testing: Calculates performance (see section 3.7.2) to compare the generated and original dataset

6. Creates and Un-normalises generated data and argmax for decoding categorical values and saves it. It also saves the model weights in a file.

3.9.2 Trialling the interface

A series of tests to assess the software were performed. The tests need to demonstrate that:

1. the synthetic and original datasets have a similar distribution,
2. the same result is obtained when the analysis is performed with the synthetic and original data,

To test the interface, a small scale 'Penguin dataset' by Gorman et al. [69] was used. The dataset records information on adult penguins in the Palmer Archipelago near Antarctica. It contains three categorical variables (species, Island location, sex) and four continuous variables (Structural size measurements and isotopic signatures of foraging among adult male and female Adele, Gentoo and Chinstrap penguins which are culmen length and culmen depth, flipper length, body mass). The dataset is small enough to be rapidly analysed but has categorical and continuous data. so comparable with the DCWW data

To evidence that the synthetic and original datasets have a similar distribution, the FD was used (see section 3.7.2). To test that the same result is obtained when analysis is performed with the synthetic and original data, a model with the GAN generated data and another model with the original data were computed, and then the two models were compared. A k-mean clustering algorithm was performed on both the synthetic and real database. This algorithm discriminates between the three species of penguin.

Table 3.1: k-mean clusters of penguin original and synthetic data

	real			synthetic			difference		
Final Cluster Centers	1	2	3	1	2	3	1	2	3
culmen_length_mm	48.9	44.7	41.1	48	44.2	42.4	2%	1%	-3%
culmen_depth_mm	15.5	17.1	18	15.2	16.5	18.4	2%	4%	-2%
flipper_length_mm	220.5	203.6	189.6	225	208	189	-2%	-2%	0%
body_mass_g	5421.2	4387.8	3490.5	5345	4468	3580	1%	-2%	-3%

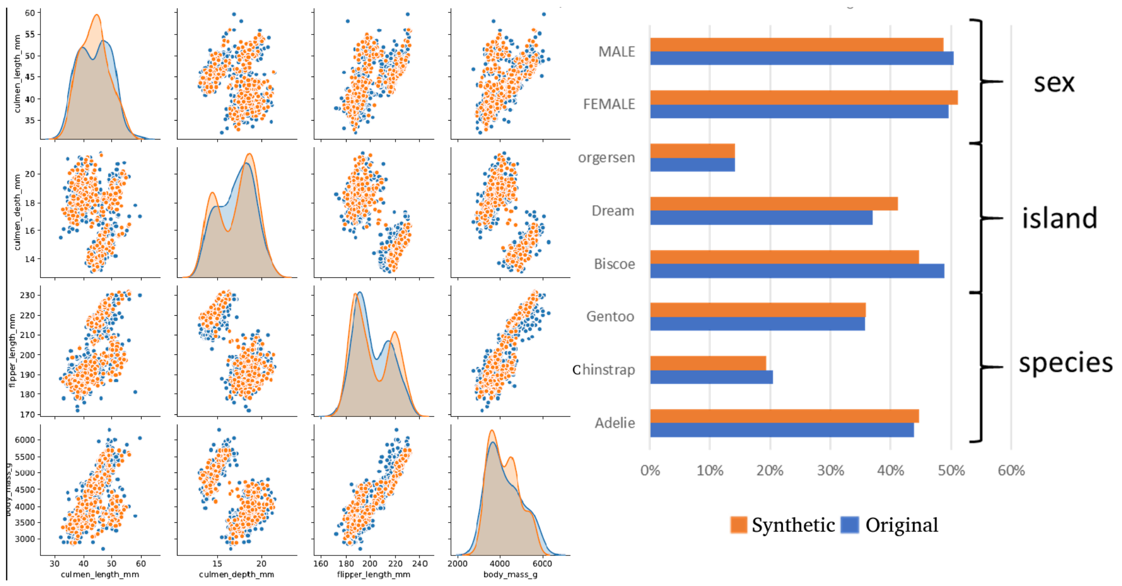


Figure 3.7: Comparison of original and synthetic data from the penguin dataset

Figure 3.7 provides an illustration of how the original and synthetic are distributed. The overlap pair plots show that both the continuous and categorical data synthesised are similar to the original data. The k-mean results (see Table 3.1) show that the quality of the synthetic dataset is high. Importantly, it also shows that the relationships between the variables are preserved.

Chapter 4

MaWGAN: GAN for data synthesis with missing values

4.1 Introduction

Missing data is a common problem for modelling and can arise for a range of reasons (see section 2.2.1). Until now, missing data has also been a problem for data synthesis methods such as GANs as existing algorithms require complete observations. This means users have to either impute the missing data or just discard incomplete observations before using the algorithm.

In this chapter, a novel GAN algorithm is proposed that can directly train a synthetic data generator from datasets with missing values. This is to my knowledge the first such attempt. It was called MaWGAN (for Masked Wasserstein GAN) as it is based on modifying the Wasserstein distance.

4.2 Related work

The work presented here builds on:

i) the GAN frameworks (see section 3.3),

ii) the use of a mask to locate missing data as developed in the GAIN strategy (see section 4.2.2)

iii) the stable optimisation process of the WGAN-GP (see section 3.4).

4.2.1 GAN frameworks

GANs are increasingly recognised as a powerful tool for data synthesis (see section 3.3). As a reminder, GANs use two neural nets, one to generate synthetic data, and the other to build a critic, which is used to train the generator (also called a discriminator). The generator and critic are trained iteratively, so that as the generator improves the critic becomes more discerning, allowing further refinement of the generator. The novel idea implemented in this Chapter is to modify the GAN algorithm so that comparing the generator output with the original data does not require the user to discard incomplete observations.

4.2.2 GAIN

Generative Adversarial Imputation Nets [70] have been proposed as one of the methods for imputing missing data based on a GAN framework where the Generator's goal is to impute missing data, and the Discriminator's goal is to distinguish between observed and imputed data. The output of the algorithm is thus a dataset that is composed of the original data and where missing data has been replaced by new synthesised data. It is important to note that the outcome dataset is therefore not a synthetic dataset but a completed dataset since all the original values are preserved. The GAIN approach creates a mask that represents the location of the missing data by assigning missing values to '0' and non-missing to '1'. To help the Discriminator distinguish between the observed and imputed data, GAIN utilises a 'hint mechanism' which reveals to the Discriminator partial information about the missingness in the original sample. This means that the Generator can focus on filling missing locations. To create a whole new dataset, rather than just filling in missing values, MaWGAN builds on the idea of a mask to hide missing data from the Discriminator so that it can compare the original data set and the synthetic one without being challenged by missing data (section 4.4).

4.2.3 WGAN-GP

One of the main issues with using GANs to create synthetic data is that the training can be unstable so that Discriminator and Generator may go into modal collapse (see section 3.3.2), where it does not produce anything useful. WGAN is a GAN that is more stable in training. Arjozky et al. [45] proposed a new model based on the Wasserstein distance to solve the stability issue. WGAN-GP was then developed to improve WGAN (section 3.4) by including a better Lipschitz function restriction. MaWGAN builds on the WGAN-GP algorithm [48, 71].

4.3 Theoretical basis

Recall from (3.11) and (3.12) that for probability measures $\mathcal{P}$ and $\mathcal{Q}$ on $\mathbb{R}^d$, the Wasserstein distance is

$$\begin{aligned} W(\mathcal{P}, \mathcal{Q}) &= \inf_{\Gamma \in \Pi(\mathcal{P}, \mathcal{Q})} \mathbb{E}_{(X,Y)\sim\Gamma} \|X - Y\|_2 \\ &= \sup_{f \,:\, \|f\|_L \le 1} \mathbb{E}_{X\sim\mathcal{P}} f(X) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y). \end{aligned}$$

Let $\mathcal{M}$ be a measure on $\{0, 1\}^d$, then the $\mathcal{M}$-Wasserstein distance is defined as

$$W_{\mathcal{M}}(\mathcal{P}, \mathcal{Q}) = \sup_{f \,:\, \|f\|_L \le 1} \mathbb{E}_{M\sim\mathcal{M}} \left(\mathbb{E}_{X\sim\mathcal{P}} f(X \odot M) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y \odot M) \right), \tag{4.1}$$

where $M \perp\!\!\!\perp X$, $M \perp\!\!\!\perp Y$, and $\odot$ indicates pointwise multiplication. Interpreting M as a random mask, $X \odot M$ is just the operation of replacing the unmasked entries of X with zeros.

The following lemma shows that $W_{\mathcal{M}}$ is equivalent to W in the topological sense (meaning they generate the same topology on the space of measures on $\mathbb{R}^d$). The practical consequence of the lemma is that a sequence of measures $\mathcal{Q}_i$ (representing a sequence of improving generators) will converge to $\mathcal{P}$ with respect to the Wasserstein distance if and only if they converge to $\mathcal{P}$ with respect to the $\mathcal{M}$-Wasserstein distance.

Lemma Let $\mathcal{M}$ be a random mask, then provided $\mathcal{M}((1,\dots,1)) > 0$ there exists a constant $c \in (0,1]$ such that

$$cW(\mathcal{P},\mathcal{Q}) \le W_{\mathcal{M}}(\mathcal{P},\mathcal{Q}) \le W(\mathcal{P},\mathcal{Q}).$$

Proof *Upper bound.* For any $M \in \{0,1\}^d$ and $\mathbf{x} \in \mathbb{R}^d$ there is $\|\mathbf{x}\|_2 \ge \|\mathbf{x} \odot M\|_2$, so

$$\inf_{\Gamma\in\Pi(\mathcal{P},\mathcal{Q})} \mathbb{E}_{(X,Y)\sim\Gamma}\|X-Y\|_2 \ge \inf_{\Gamma\in\Pi(\mathcal{P},\mathcal{Q})} \mathbb{E}_{(X,Y)\sim\Gamma}\|(X-Y)\odot M\|_2$$

and, thus, integrating M with respect to $\mathcal{M}$, provides

$$\begin{aligned}
W(\mathcal{P},\mathcal{Q}) &\ge \mathbb{E}_{M\sim\mathcal{M}} \inf_{\Gamma\in\Pi(\mathcal{P},\mathcal{Q})} \mathbb{E}_{(X,Y)\sim\Gamma}\|(X-Y)\odot M\|_2 \\
&= \mathbb{E}_{M\sim\mathcal{M}} \sup_{\|f\|_L\le 1} \left(\mathbb{E}_{X\sim\mathcal{P}} f(X\odot M) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y\odot M)\right) \\
&\ge \sup_{\|f\|_L\le 1} \mathbb{E}_{M\sim\mathcal{M}} \left(\mathbb{E}_{X\sim\mathcal{P}} f(X\odot M) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y\odot M)\right) \\
&= W_{\mathcal{M}}(\mathcal{P},\mathcal{Q}).
\end{aligned}$$

Here the second line follows because $X \odot M$ can be viewed as a realisation of $\mathcal{P}$ projected onto the subspace corresponding to the non-zero co-ordinates of M, and similarly for $Y \odot M$.

Lower bound. For any function f there is

$$\begin{aligned}
&\mathbb{E}_{M\sim\mathcal{M}} \left(\mathbb{E}_{X\sim\mathcal{P}} f(X\odot M) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y\odot M)\right) \\
&\quad = \sum_{M\in\{0,1\}^d} \mathcal{M}(M) \left(\mathbb{E}_{X\sim\mathcal{P}} f(X\odot M) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y\odot M)\right) \\
&\quad \ge \mathcal{M}((1,\dots,1)) \left(\mathbb{E}_{X\sim\mathcal{P}} f(X) - \mathbb{E}_{Y\sim\mathcal{Q}} f(Y)\right)
\end{aligned}$$

where $W_{\mathcal{M}}(\mathcal{P},\mathcal{Q}) \ge \mathcal{M}((1,\dots,1))W(\mathcal{P},\mathcal{Q})$. $\square$

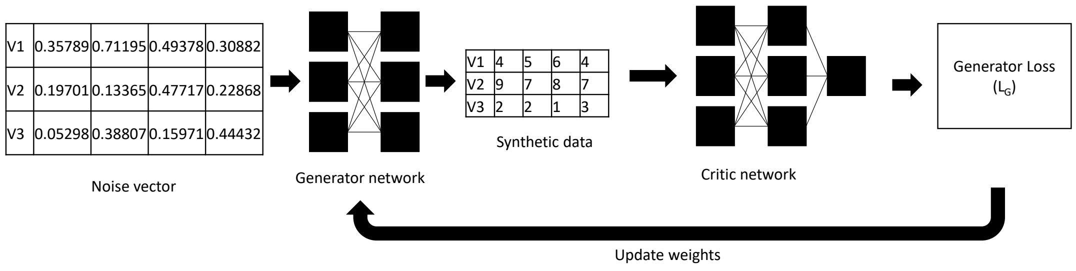
As for WGAN-GP, the $\mathcal{M}$-Wasserstein distance can be approximated by using samples from $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{M}$, using a net for f, and replacing the Lipschitz constraint with a gradient penalty. Let $\{x_i\}_{i=1}^n$ be a data sample from $\mathcal{P}$, $\{m_i\}_{i=1}^n$ a sample of masks from $\mathcal{M}$, $G = G(\cdot;\theta_g)$ the generator and $\{z_i\}_{i=1}^n$ a sample of $U(0,1)^d$, and $C = c(\cdot;\theta_c)$ the critic, then the RHS of (4.1) becomes

$$\sup_{\theta_c} \frac{1}{n}\sum_{i=1}^{n} C(x_i \odot m_i;\theta_c) - \frac{1}{n}\sum_{i=1}^{n} C(y_i \odot m_i;\theta_c) - \lambda\frac{1}{n}\sum_{i=1}^{n}(\|\nabla C(h_i \odot m_i;\theta_c)\|_2 - 1)^2, \tag{4.2}$$

where λ is a user chosen penalty scaling, $y_i = G(z_i;\theta_g)$, and $h_i = t_i x_i + (1 - t_i)y_i$ for $t_i \sim U(0,1)$.

4.4 Implementation

In this section the MaWGAN implementation is explained step by step. Figures 4.1 and 4.2 illustrate the flow of information in a single training step for the generator and critic respectively.



V1	0.35789	0.71195	0.49378	0.30882
V2	0.19701	0.13365	0.47717	0.22868
V3	0.05298	0.38807	0.15971	0.44432

V1	4	5	6	4
V2	9	7	8	7
V3	2	2	1	3

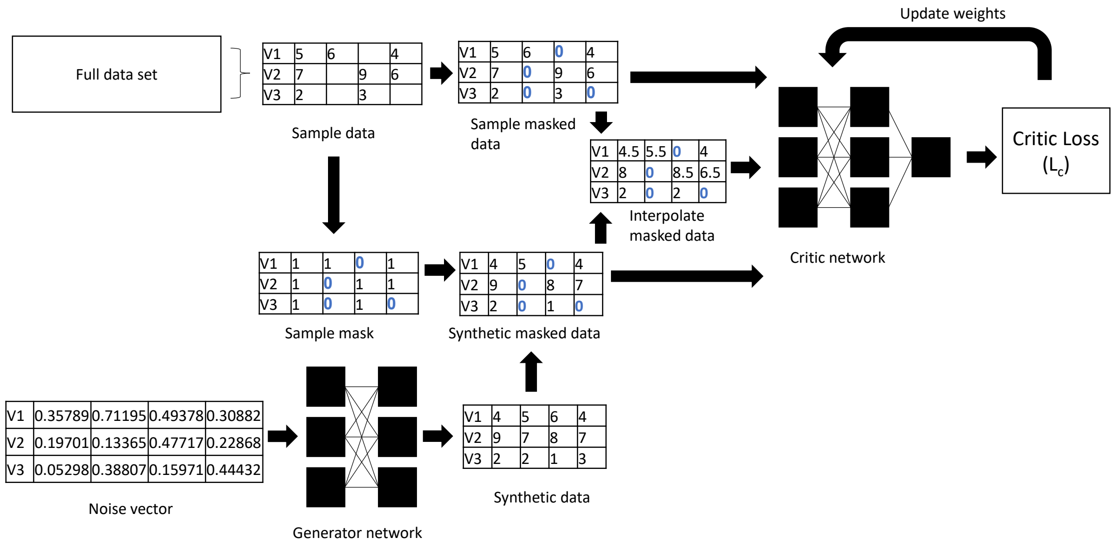
Figure 4.1: Flowchart for a single training step of the Generator

- In both cases, calculate a loss which measures the performance of the generator/critic.
- Given the loss, calculate its gradient with respect to the weights (parameters) of the generator/critic, then update the weights in the direction of the gradient.
  - Feed an array of random numbers into the generator one row at a time, to obtain an array of synthetic data (each row represents an independent realisation).

  – Feed the synthetic data into the critic one row at a time to obtain a vector of performance evaluations, which average to obtain the loss.

- Training the critic needs two sets of inputs: a sample (or batch) from the original dataset and a synthetic dataset of the same size produced by the generator. From the original dataset generates a mask indicating which data are missing, which is used to both replace the missing data with zeros, and replace the corresponding entries in the synthetic data array with zeros.

- Also generate an interpolated data array, which is just a linear combination of the masked original and masked synthetic data. Each row of the original and synthetic data is fed into the critic, each row of the interpolated data array is fed into the gradient of the critic, and these are averaged as per Equation (4.2) to give the loss.



Sample data

V1	5	6		4
V2	7		9	6
V3	2		3	

Sample masked data

V1	5	6	0	4
V2	7	0	9	6
V3	2	0	3	0

Interpolate masked data

V1	4.5	5.5	0	4
V2	8	0	8.5	6.5
V3	2	0	2	0

Sample mask

V1	1	1	0	1
V2	1	0	1	1
V3	1	0	1	0

Synthetic masked data

V1	4	5	0	4
V2	9	0	8	7
V3	2	0	1	0

Noise vector

V1	0.35789	0.71195	0.49378	0.30882
V2	0.19701	0.13365	0.47717	0.22868
V3	0.05298	0.38807	0.15971	0.44432

Synthetic data

V1	4	5	6	4
V2	9	7	8	7
V3	2	2	1	3

Figure 4.2: Flowchart for a single training step of the Critic

The pseudocode below shows how the generator and critic steps are interwoven. For each update step of the gradient, several updates of the critic are performed t_C times to keep the critic as a good approximation of the $\mathcal{M}$-Wasserstein distance (see explanation at the bottom of section 3.3.3)

The following notations were chosen for the pseudocode: $\mathbf{x}_i \in \mathbb{R}^d$ are the observations for $i = 1, \dots, n$, which collect into an $n \times d$ matrix X, where the i-th row of X is $\mathbf{x}_i^T$. $\mathbf{m}_i$ is the mask corresponding to $\mathbf{x}_i$ and M is the $n \times d$ matrix whose i-th row is $\mathbf{m}_i^T$. $G : (0,1)^d \to \mathbb{R}^d$ is the generator and $C : \mathbb{R}^d \to \mathbb{R}_+$ the critic. $\boldsymbol{\theta}_G$ are the weights that parameterise the generator G, and $\boldsymbol{\theta}_C$ for the critic weights. It is $\boldsymbol{\theta}_G$ and $\boldsymbol{\theta}_C$ that update when training G and C. The update steps require a learning rate α, which is not explicitly included in the pseudocode. In the algorithm the generator is updated t_G times, which are called epochs. For each epoch the critic is updated t_C times, and a batch of data size β is used. The items in the batch are noted $\sigma \subset \{1, \dots, n\}$ and $\sigma(i)$ for the i-th item. $\lambda > 0$ is the regularisation parameter for the critic loss, which also needs to be set before hand.

```
Require: initial generator weights $\boldsymbol{\theta}_G$ and critic weights $\boldsymbol{\theta}_C$, learning rate $\alpha$
Require: num.epochs $t_G$, critic iterations $t_C$, batch size $\beta$, critic regularisation $\lambda$
 1: for $s = 1, \dots, t_G$ do                                          ▷ update the generator
 2:     for $t = 1, \dots, t_C$ do                                          ▷ update the critic
 3:         choose a batch $\sigma$ of size $\beta$ from $\{1, \dots, n\}$
 4:         for $i = 1, \dots, k$ do                                      ▷ calculate critic loss
 5:             $\bar{\mathbf{x}}_i \leftarrow \mathbf{x}_{\sigma(i)} \odot \mathbf{m}_{\sigma(i)}$
 6:             sample $\mathbf{u} \sim U(0,1)^d$
 7:             $\mathbf{y}_i \leftarrow G(\mathbf{u}) \odot \mathbf{m}_{\sigma(i)}$
 8:             sample $h \sim U(0,1)$
 9:             $\mathbf{h}_i \leftarrow h\bar{\mathbf{x}}_i + (1-h)\mathbf{y}_i$
10:             $L_C^i \leftarrow C(\bar{\mathbf{x}}_i) - C(\mathbf{y}_i) - \lambda(\|\nabla C(\mathbf{h}_i)\|_2 - 1)^2$
11:         end for
12:         $L_C \leftarrow \frac{1}{k}\sum_{i=1}^{k} L_C^i$
13:         update $\boldsymbol{\theta}_C$ using gradient of $L_C$ (increasing $L_C$)
14:     end for
15:     for $i = 1, \dots, k$ do                                      ▷ calculate generator loss
16:         sample $\mathbf{u} \sim U(0,1)^d$
17:         $L_G^i \leftarrow C(G(\mathbf{u}))$
18:     end for
19:     $L_G \leftarrow \frac{1}{k}\sum_{i=1}^{k} L_G^i$
20:     update $\boldsymbol{\theta}_G$ using negative gradient of $L_G$ (decreasing $L_G$)
21: end for
```

Note that, compared to the orignial WGAN-GP, a mask is added lines 5 and 7.

4.5 Numerical testing

4.5.1 Datasets

To test the performance of MaWGAN three datasets were used of varying sizes and complexity. These are multivariate tabular datasets that vary from 150 to 20,000 observations and four to sixteen variables.

- The well-known **Iris** dataset which records the length and width of the sepals and petals of the flowers of three different iris species [72, 73]. The dataset comprises 150 observations and four variables. This dataset is further detailed in Section 3.8.2.
- The 2019 values of the **Welsh Index of Multiple Deprivation (WIMD)** which is the Welsh Government's official measure of relative deprivation in Wales (UK) [74]. For 1904 separate regions the WIMD provides measures of income, employment, education, and health. One region had a missing value that was removed from the dataset, leaving 1903 observations of 11 numerical variables.
- The **Letter** dataset which was generated by Frey and Slate [75] and records 16 measured characteristics of images of the capital letters in the English alphabet. Letters were selected from 20 different fonts and randomly distorted a number of times. The dataset comprises 20,000 observations of 16 numerical variables.

4.5.2 Simulated MCAR datasets

To simulate MCAR datasets, nine additional versions of each dataset were generated with 10%, 20%, ..., 90% missing data. Points were removed at random with equal probability until the required percentage was reached. The additional datasets are nested in the sense that if an element is missing from one then it is missing from all versions with higher levels of missing data. By artificially removing data the performance of the synthetic data generator with the complete dataset could be compared even when it was trained with missing data.

4.5.3 Competing methodologies

MaWGAN was compared to two other approaches. The first is a two-step process where the GAIN imputation method is applied and then used WGAN-GP to train a generator on the completed data. The second alternative was to discard incomplete observations then use a WGAN-GP to train a generator on what remained. The number of remaining observations at the different levels of missingness are given in Table 4.1.

Table 4.1: Number of complete observations remaining in each dataset after different proportions of data were removed at random.

Percentage	Dataset		
missing	Iris	WIMD	Letter
0%	150	1093	20000
10%	95	666	3723
20%	57	218	564
30%	35	59	67
40%	16	11	5
50%	5	2	0
60%	1	0	0
70%	0	0	0
80%	0	0	0
90%	0	0	0

4.5.4 Performance metric

To assess the performance of the three methods the *likeness* score L was used see section 3.7.3. To reduce the variation due to sampling from the generator, L was calcuated 100 times using different sets of synthetic data. The average value was retained.

4.6 Algorithmic details

The MaWGAN, GAIN and WGAN-GP algorithms were implemented in Python using the PyTorch library [64]. The MaWGAN and WGAN-GP implementations incorporated publicly available GitHub code [66] (see appendix A.2), and the GAIN implementation used the code provided by the original authors [70]. For both MaWGAN and WGAN-GP, the neural network architecture of both the generator and Critic had five layers. For the generators, the input and output layers had nodes equal to the number of variables, and 150 nodes were used per hidden layer. The number of nodes was chosen by sequential approximation. For the Critic the input layer has nodes equal to the number of variables, output layer size 1, and 150 nodes were used per hidden layer. For training, $t_G = 15,000$ epochs were used with $t_C = 5$ training steps for the Critic each time. The number of steps reflects the value at which the Critic continues to perform without collapsing. A batch size of $k = 30$ was used, and a learning rate of $\alpha = 0.0001$ and critic regularisation $\lambda = 10$. These hyperparameters were chosen based on previous experience with WGAN-GP.

4.7 Results

Because GAN training is stochastic, the performance of the resulting generator can vary. Accordingly for each combination of method, dataset and missingness, the generator was fitted 20 times, calculating the Likeness score each time. The results are summarised in Figure 4.3. For each combination of method, dataset and missingness the average Likeness score is given and a 95% confidence interval for the mean.

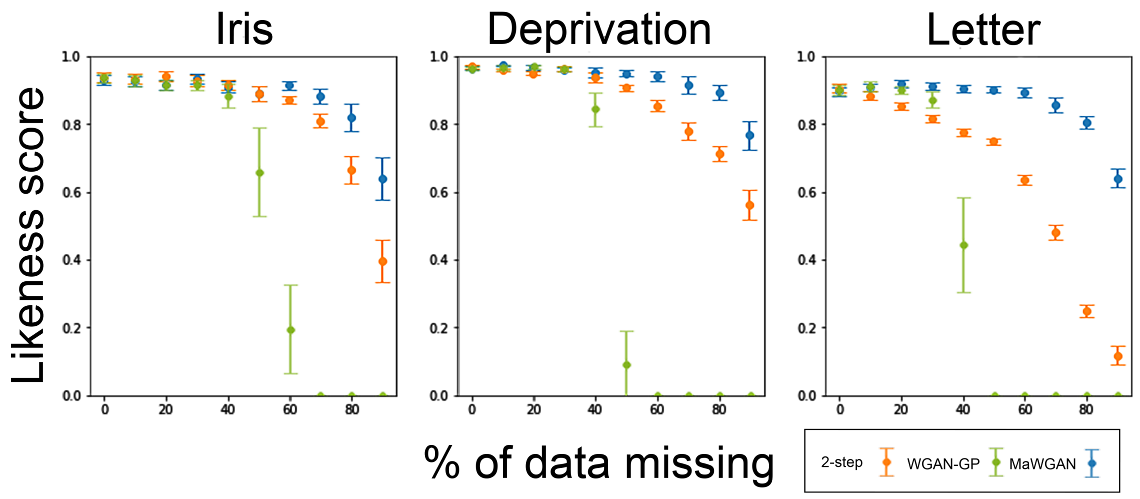


Figure 4.3: Likeness scores for each method on the three datasets with different levels of missingness.

Overall, the results show that, for these datasets, MaWGAN performs consistently well with levels of missing data up to 50%. MaWGAN also performs significantly better than both the two-step method and the complete observations method with moderate to high levels of missing data, and never performed any worse than either alternative. It is also notable that MaWGAN's performance is less variable than the two-step method. For both methods there is some variability in the training of the generator, however for the two-step method, additional variability is introduced in the training of the GAIN imputation function.

4.8 Discussion

MaWGAN is a generalisation of WGAN-GP, since in the absence of missing data it is exactly a WGAN-GP, yet it requires no more parameter tuning than a WGAN-GP. Moreover, the masking step that implements MaWGAN is simple to add to existing code and all runs showed it had a marginal impact (<2 seconds over 115 runs) on the running time (calculating the weight-gradient for the generator and critic remain the most expensive steps). As in most GANs, the bigger datasets needed more training [76]. It is possible that training needs also increase with the proportion of missing data. For example, looking at the large Letters dataset, the Likeness score decreases with the proportion of missing data, and it is possible that this reduction in performance could be managed by increasing training. Note that the theory and implementation apply equally well to the original WGAN formulation as the WGAN-GP approach, though the latter is recommended as its approach to training the critic was much more stable. Note also that the datasets used to test MaWGAN are missing completely at random. This is not always the case in real-life scenarios, namely when the cause of missing data is linked to non-random events.

Overall, the experimental results described here demonstrate that for dealing with data missing completely at random (MCAR), MaWGAN has superior performance to the alternatives of separately imputing missing data or discarding incomplete observations.

Chapter 5

Data synthesis of time series with missing data using GANs

5.1 Overview

The aim of this chapter is to generate synthetic data from time series with missing data. There are existing methods for creating synthetic data from regular time series using GANs (see Chapter 3), but these do not work on time series with missing data. On the other hand, the MaWGAN algorithm described in Chapter 4 can handle missing data, but there is an assumption that the data are independent. Here a new algorithm is proposed using the MaWGAN backbone, called Force-GAN, that generates synthetic data from regular time series with missing data.

5.2 Methods

5.2.1 Preliminary exploration using RNN-nodes in MaWGAN

Methods for creating synthetic sequences from regular time series using GANs rely on the use of recurrent neural network nodes (RNN-nodes) to model the time dependencies of the sequence. It does this by learning a temporal pattern using previous observations in the sequence. An initial exploration based on RNN was therefore trialed as a logical approach to resolving the challenge described in the overview. The Feed Forward nodes in the Generator and Critic hidden layers in the MaWGAN algorithm (see pseudocode in Chapter 4) were simply replaced with Recurrent Neural Nodes. To be more specific, the GRU (Gated Recurrent Unit) node was used since it exhibits a better memory than the traditional RNN or LSTM (Long Short Term Memory). This means replacing regular backpropagation with backpropagation through time [77].

This approach was tested using two datasets: a simple Auto-regressive simulated dataset (section 5.3.1) and a real multivariate dataset (5.3.2). Both tests resulted in outputs that were not in any way similar to the original time seriesso this method was not retained. This result can be explained by the fact that the masking process in MaWGAN means that missing values are replaced by '0', so that the memory from the RRN node is corrupted.

5.2.2 Development of Force-GAN

To capture time dependencies, a state space u was created which lists all the possible subsequences with a length k (see section 5.2.3). This subsequence, called the lag, must be large enough to incorporate the longest history. The GAN is trained on u and learns the joint distribution of current and past observations. The Generator produces at least one subsequence of length k, and each subsequence is independent from each other, so cannot be stitched together. The solution is to build a forecaster model to expand the length of the subsequence that the Generator produces. This Forecaster model is a generator NN of a conditional GAN, conditioned on the previous observations [78]. The Generator and the Forcaster share the same Critic. The Critic training used is the same as the Critic used in MaWGAN (see chapter 4) as it can use time-series with missing values. In summary, three neural networks are used in the architecture: the Generator G, the Critic C and the Forecaster F.

The resulting Force-GAN algorithm can be broken down into 5 steps:

1. Normalise the time series as neural networks tend to struggle with large variances. The time series size n with dimentions d and their observations $x_{1d}, \dots, x_{nd}$ are

normalised, $\tilde{x}_{ij} = \frac{x_{ij}-\mu_j}{\sigma_j}$ where μ_j the sample mean of $\{x_{ij} : i = 1, \ldots, n\}$ and its sample deviation is σ_j.

2. Reformat the data by extracting observations to a lag (u) matrix (see section 5.2.3).

3. Use MaWGAN to train a Generator G with the Critic C (see section 5.2.4) on the lag matrix.

4. Train the Forecaster F with the Critic C on the data synthesised by the Generator, but don't continue training G (see section 5.2.5).

5. Generate synthetic series by starting with a single sample from G and using F to forecast the rest of the sequence (see section 5.2.6).

5.2.3 Reformatting for Force-GAN

The normalised time series dataset is converted into a lag matrix (Figure 5.1). Let the original dataset have d dimensions, and contain n observations. All the observations $x_i \in \mathbb{R}^d$ require to be regularly spaced, but observations may be missing.

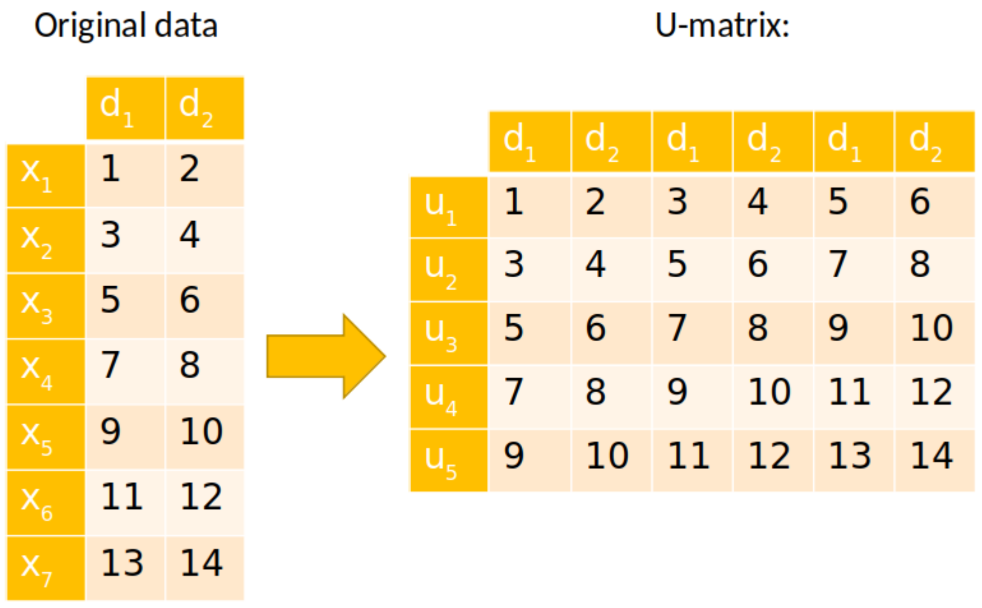
For a lag k, u_i is defined as:

$$u_i = (\tilde{x}_i | \tilde{x}_{i+1} | \ldots | \tilde{x}_{i+k}) \in \mathbb{R}^{(k+1)d}$$

for $i = 1, \ldots, n-k$ and then put the u_i into a matrix u:

$$u = \begin{pmatrix} u_1 \\ - \\ \vdots \\ - \\ u_{n-k} \end{pmatrix}$$

The mask m corresponding to u_i is a matrix of similar shape u, and is defined as: $m_i \in \{0,1\}^d$, $i = 1, \ldots, n$. Like MaWGAN, the mask is used to capture the location of the missing values.


Original data

	d_1	d_2
x_1	1	2
x_2	3	4
x_3	5	6
x_4	7	8
x_5	9	10
x_6	11	12
x_7	13	14

U-matrix:

	d_1	d_2	d_1	d_2	d_1	d_2
u_1	1	2	3	4	5	6
u_2	3	4	5	6	7	8
u_3	5	6	7	8	9	10
u_4	7	8	9	10	11	12
u_5	9	10	11	12	13	14

Figure 5.1: In this original dataset, the two dimensions, d1 and d2, are converted into a 2-lag Matrix, which therefore contains six dimensions, where the first two dimensions hold the original sequence, the next two hold the 1-lag sequence, and the next the 2-lag sequence.

5.2.4 Train the Critic with the Generator

Once the pre-processing is done, the MaWGAN is ready to be trained on the u matrix (see Chapter 4 for the training process). The Critic and Generator are structured as: Critic $C : \mathbb{R}^{(k+1)\times d} \to \mathbb{R}$ with the Generator $G : (0,1)^{(k+1)\times d} \to \mathbb{R}$

5.2.5 Train the Forecaster

A synthetic sample $w \in \mathbb{R}^{(k+1)\times d}$ is produced by the generator a row of u. Then $w = (w(*)|w(-))$, where $w(*)$ are the first kd elements of w and $w(-)$ are the last elements. The Forecaster $F : \mathbb{R}^{k\times d} \times (0,1)^d \to \mathbb{R}$, and takes a sequence of random numbers $b \sim U(0,1)^d$ and $w(-)$ from the generator. to produce a replacement for $w(-)$ (see Figure 5.2). The w is then fed into the critic for training.

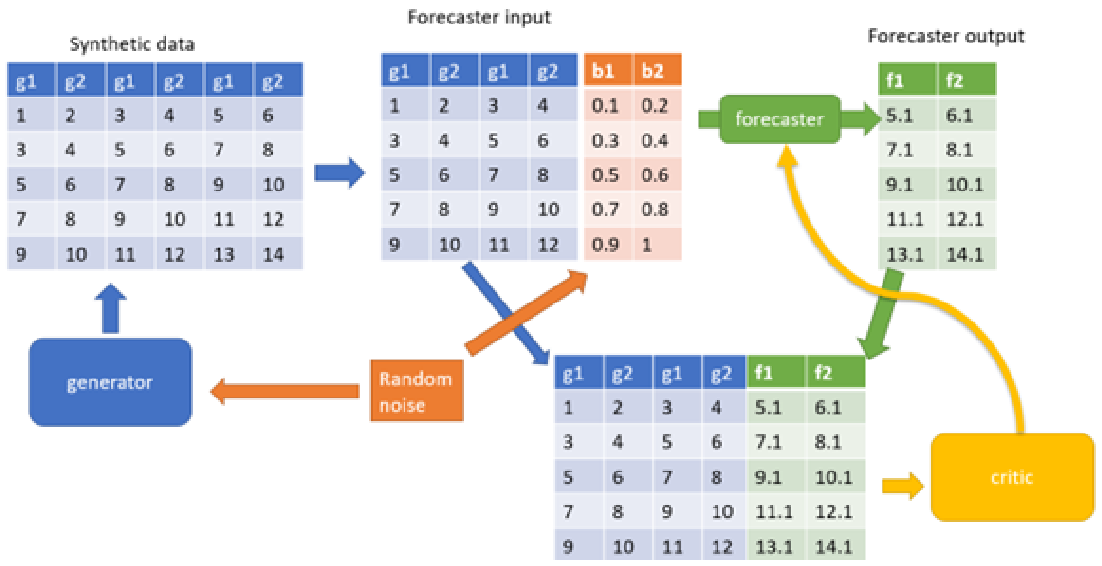


Figure 5.2: Flowchart of training the Forecaster

A vector of performance evaluations, which is averaged to obtain the loss, is then used to update the weight of the Forecaster. The Critic must continue to be trained with the Forecaster, which is achieved by using a training approach that blends the MaWGAN and Forecaster training. This training encourages the Critic to concentrate on the distribution of the last element of the sample given the other k elements (because the training of F will not affect the distribution of the first k elements passed to the Critic).

Below is the pseudocode for the Forecaster training:

```
Require: Forecaster weights $\Theta_F$ and critic weights $\Theta_C$, learning rates $\alpha_F$ and $\alpha_C$.
Require: num. epochs $t_F$, forecaster batch size $\beta_F$, critic iterations $t_C$, critic batch size
  $\beta_C$, critic regularisation $\lambda$.
  for $s = 1, \dots, t_F$ do                                          ▷ update the Forecaster
      for $t = 1, \dots, t_C$ do                                      ▷ update the critic
          choose a batch $\sigma$ of size $\beta_C$ from $\{1, \dots, n-k\}$
          for $i = 1, \dots, \beta_C$ do                              ▷ calculate critic loss
              $\bar{u}_i \leftarrow u_{\sigma(i)} \odot m_{\sigma(i)}$
              $w_i = (w_i(*)|w_i(-)) \leftarrow G(z)$ for $z \sim U(0,1)^{(k+1)\times d}$
              $w_i(-) \leftarrow F(w_i(*), b)$ for $b \sim U(0,1)^d$
              $\bar{w}_i \leftarrow w_i \odot m_{\sigma(i)}$
              $z_i \leftarrow \epsilon\bar{u}_i + (1-\epsilon)\bar{w}_i$ for $\epsilon \sim U(0,1)$
              $L_C^i \leftarrow C(\bar{u}_i) - C(\bar{w}_i) - \lambda(\|\nabla C(z_i)\|_2 - 1)^2$
          end for
          $L_C \leftarrow \frac{1}{\beta_C}\sum_{i=1}^{\beta_C} L_C^i$
          update $\Theta_C$ using gradient of $L_C$ (increasing $L_C$) and learning rate $\alpha_C$
      end for
      for $i = 1, \dots, \beta_F$ do                                  ▷ calculate Forecaster loss
          $w_i = (w_i(*)|w_i(-)) \leftarrow G(z)$ for $z \sim U(0,1)^{(k+1)\times d}$
          $w_i(-) \leftarrow F(w_i(*), b)$ for $b \sim U(0,1)^d$
          $L_F^i \leftarrow C(w_i)$
      end for
      $L_F \leftarrow \frac{1}{\beta_F}\sum_{i=1}^{\beta_F} L_F^i$
      update $\Theta_F$ using negative gradient of $L_F$ (decreasing $L_F$) and learning rate $\alpha_F$
  end for
```

5.2.6 Data generation

To create synthetic data, the Generator creates an initial window w_1 which is a sequence $k + 1 \times d$ observations long (Figure 5.3). The first k observations of the w_i together with a random number b forms a window that is fed into the Forecaster to obtain the next values in the sequence y. The window then moves along. This process repeats until it reaches the desired length. Below is the pseudocode of the generation process:

```
**Require:** simulation run length $l$
    let $G(z) = (w_1| \cdots |w_{k+1}, \in \mathbb{R}^d$ for $z \sim U(0,1)^{(k+1)\times d}$
    **for** $i = k+2, \ldots, l$ **do**
        $w_i \leftarrow F((w_{i-k}|w_{i-k+1}| \cdots |w_{i-1}), b)$ for $b \sim U(0,1)^d$
    **end for**
```

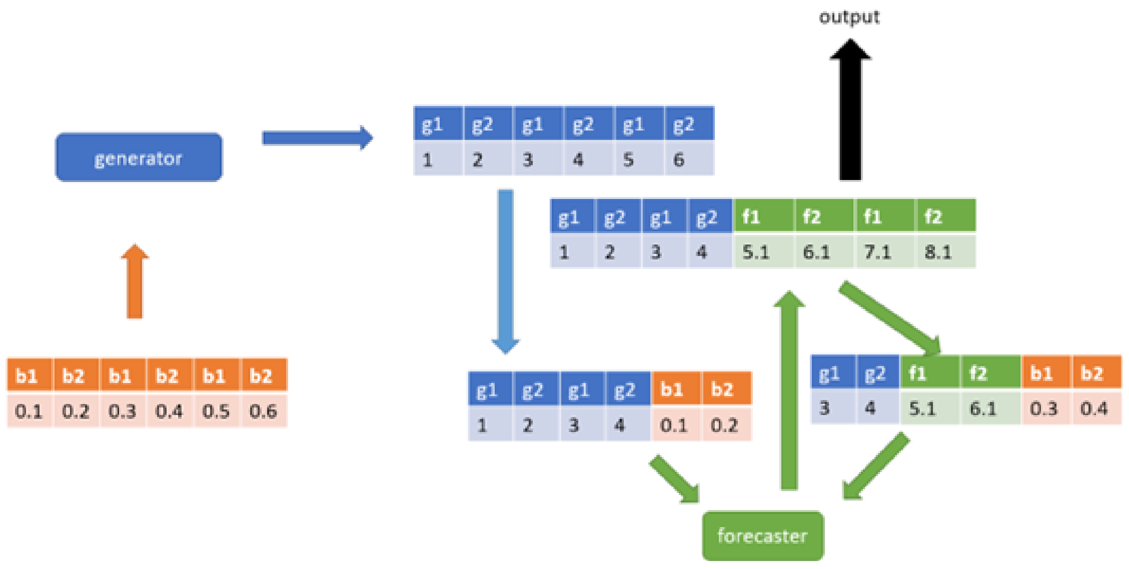


Figure 5.3: Flowchart of data generation after training with the Generator creating the initial sequence in blue and the Forecaster extending it in green. The latent space is in orange.

5.3 Testing Force-GAN

To test the performance of Force-GAN, four different tests on two datasets were performed: a sequence generated using a simple auto-regressive model, and a multivariate time-series of measured environmental variables [79]. Data points were removed randomly from each dataset to simulate a Missing Completely at Random dataset (MCAR, see section 2.2.1). Force-GAN was trained thirty times for each dataset, and 100 sequences were generated each time. Two different performance measures were used (see section 5.3.3).

5.3.1 Sequence from simple auto-regressive model

An Auto-Regressive (AR) model was used with three parameters is used to generate a sequence using the following steps:

1. 100 samples of the AR model, with $\phi = (0.1, -0.3, 0.9)$ as coefficients, were created to be used as the base to train Force-GAN.
2. The parameters used for training Force-GAN were: a Lag of 4, each neural network has one hidden layer, with 100 nodes per layer, and a learning rate of 0.0001.
3. From the trained Force-GAN, 100 samples of synthetic data are generated.
4. A new AR model was trained with this synthetic data.
5. The parameters of the new AR model and the original were compared.

To assess the effect of missing data, for each original AR(3) sequence, six auxiliary sequences were generated with increasing levels of missing data: 10%, 20%, ..., 60%. For the experiments, $k = 3$ was selected, and the Critic and Generator both had 1 hidden layer with 100 nodes. For the initial training of the Generator and Critic, a learning rate $\alpha = 0.0001$; batch size 30; iterations $t_G = 5000$ and $t_C = 10$; and Critic regularisation $\lambda = 10$ was used. For the training of the Forecaster, the same parameters worked, namely the learning rates were $\alpha_F = 0.0001$ and $\alpha_C = 0.0001$; batch sizes were $\beta_F = 30$ and $\beta_C = 30$; iterations were $t_F = 5000$ and $t_C = 10$; and the Critic regularisation was $\lambda = 10$. As for previous experiments (see Chapter 4), the hyperparameters were selected based on trial and error. The parameters were optimised as for any WGAN-GP (see Chapter 3).

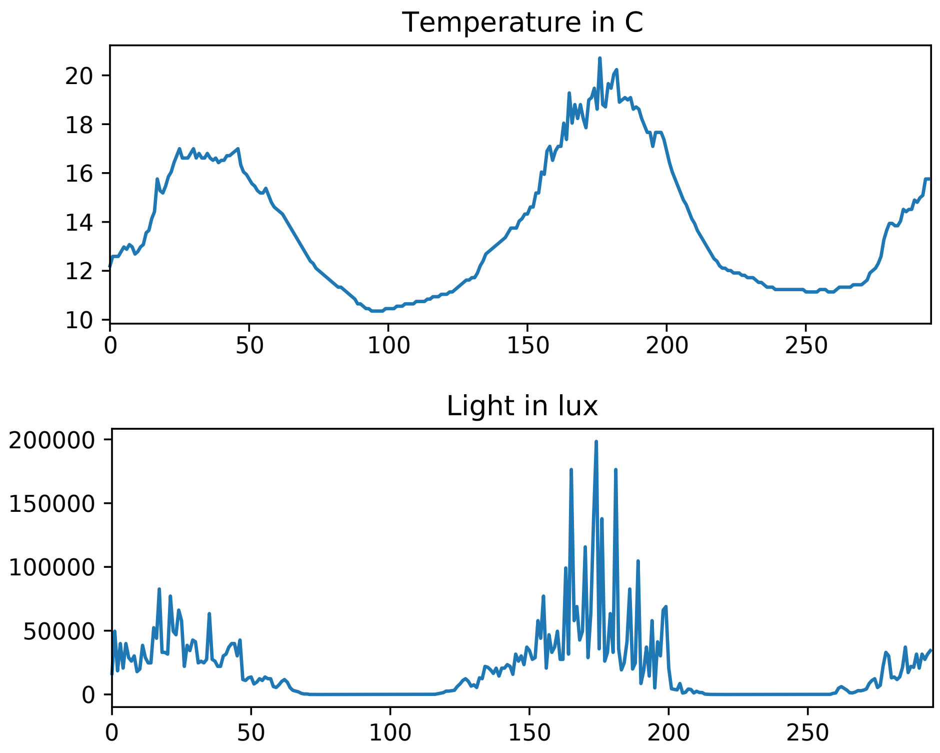


Figure 5.4: Temperature vs light original dataset

5.3.2 Temperature vs Light dataset

This dataset was originally used to find a relationship between light levels and the water temperature of streams in Wales [79]. The data was recorded every 10 minutes and shows a cycle pattern linked to day and night.

To capture all the complexity of the data, the number of nodes was increased compared to the last example. The parameters used were: Lag of 110 to capture the large time dependency, each neural network has 2 hidden layers, with 300 nodes per layer, and a learning rate of 0.0001.

The Temperature vs Light data is far more complex than the AR model and therefore more sensitive to the amount of missing values. So for this dataset, the increasing levels of missing data were reduced to 5%, 10%, ..., 50%.

5.3.3 Performance measures

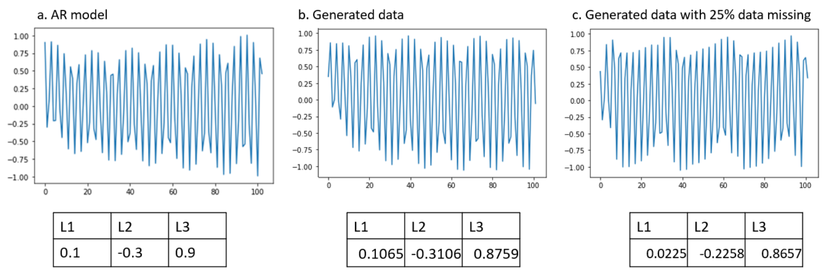
Two performance measures were used to quantify the performance of the Force-GAN. The first is the Likeness Score Ls introduced by Guan & Loew (see section 3.7.3), used for both datasets. The score compared the original datasets and the output from the Forecaster. The second performance measure used was ad-hoc, which differed for each dataset. For the AR(3) process, the performance measure used was just the mean-squared error $M := \|\boldsymbol{\phi} - \hat{\boldsymbol{\phi}}\|_2^2/3$, where $\hat{\boldsymbol{\phi}}$ is the usual maximum likelihood estimate of $\boldsymbol{\phi}$ calculated using a synthetic sample from the Force-GAN. This measure was called the AR Coefficient Score, and smaller values indicate a better outcome.

For the Temperature vs Light dataset, the performance measure required three steps. First, a sample of sequence of 1000 was generated by Force-GAN. Then, two regressions between light and temperature are run using the generated and the original dataset. Third, the difference between original data and data predicted from regression models of Temperature vs Light based on Force-GAN is calculated. The difference was used as the performance measure.

5.4 Results

5.4.1 AR sequence

The performance of the Force-GAN is summarised in Figures 5.6 (Likeness Score and AR Coefficient Score). For both figures, the average performance of the Force-GAN for different levels of missing data (with 95% confidence intervals) is given. For both measures, the performance of the Force-GAN is not significantly affected by the levels of missing data until up to 40%. Each point is the mean of 30 calculations. For M, the average squared distance between the true parameters of an AR(3) process and the estimated parameters from a synthetic sample generated using a Force-GAN was used. The GAN was trained using a sample of size 100 from the AR(3) model. For the Likeness score, a sequence of length 100 from an AR(3) process with synthetic data generated using a Force-GAN was used. The GAN was trained using the AR(3) sample. The level of missing data is the proportion of observations removed at random from the original sample before training the Force-GAN, though note that the Likeness Score is always calculated using the original data with no missing observations.



L1	L2	L3
0.1	-0.3	0.9

L1	L2	L3
0.1065	-0.3106	0.8759

L1	L2	L3
0.0225	-0.2258	0.8657

Figure 5.5: Comparision of original vs generated data. a is the original dataset (AR model), b is the generated dataset based on a. c is the generated dataset from a dataset were 25% of the a data was removed.

To have a basis for comparison, an AR(3) process was fitted to each of the original samples generated, and then Ls and M were calculated as above (see figure refcomapre). For the AR Coefficient Score, a mean of 0.0027 with 95% CI (0.0012, 0.0042) was obtained, and for the Likeness Score a mean of 0.809 and 95% CI (0.751, 0.867). According to the AR Coefficient Score, the original data gave a better fit, which is to be expected as a correctly specified model was used. However, according to the Likeness Score, the data generated by Force-GAN provides a better fit. In interpreting this result, it is important to remember that the Force-GAN fits to the sample it is given, while the Likeness Score measures how well that sample is matched, rather than the parameters of the original process. The Likeness score in this case is thus not as useful as comparing the AR coefficient scores.

5.4.2 Temperature versus light data

For the regression coefficient scores (see Figure 5.7), performance on the Temperature vs Light data decreases like the AR(3) process. In the case of the Temperature vs Light dataset, the missing data appears to have a much larger effect than in the AR(3) dataset. This is probably due to the larger time dependency in the Temperature vs Light data. For the Likeness score shown in Figure 5.8, the score without missing data is less than the AR(3) process, and as the missing data ratio increases, the scores gradually decrease.

5.5 Discussion

This chapter set out to develop a GAN that could handle time series with missing values. Force-GAN produces some promising results in that it: i) performed well on time series with a clear pattern such as those generated by the AR(3) process, and ii) the method easily generalised to multivariate time-series, and could be used to model any conditional distribution.

However, Force-GAN showed some limitations. Applied to the more complex real datasets, such as the Temperature vs Light time series, the performance was not as good. This limitation in performance could be linked to different causes. One is that the initial training of the Critic may mean it is too specialised for the early training of the Forecaster, causing it to focus on details rather than broad features. One way of alleviating this problem somewhat could be to pause the training of the Critic while the Forecaster 'catches up'. Another cause could be the use of a set lag which means that a dataset with a known lag will yield a good performance but that the GAN struggles with datasets with less clear patterns. It is clear that in practice the choice of lag is important and may not be as straightforward as for these two datasets. The lag needs to be large enough to encompass any features in the data, such as seasonal effects or irregular cycles, however the larger the lag the more complicated the Forecaster has to be, which translates into more and larger internal layers for all the GAN networks (Generator, Critic and Forecaster), making the fitting slower and more temperamental.

To ensure the convergence of a GAN requires a balance between the speed at which the Generator and Critic converge. Force-GAN is more challenging as it requires the convergence of three networks: the Generator, the Critic and the Forecaster. To simplify this, the Generator and Critic were trained first, then switching to the Critic and Forecaster. It is possible that simultaneous training of all three networks could improve the overall speed at which they converge and therefore improve their performance, but this creates additional parameter tuning.

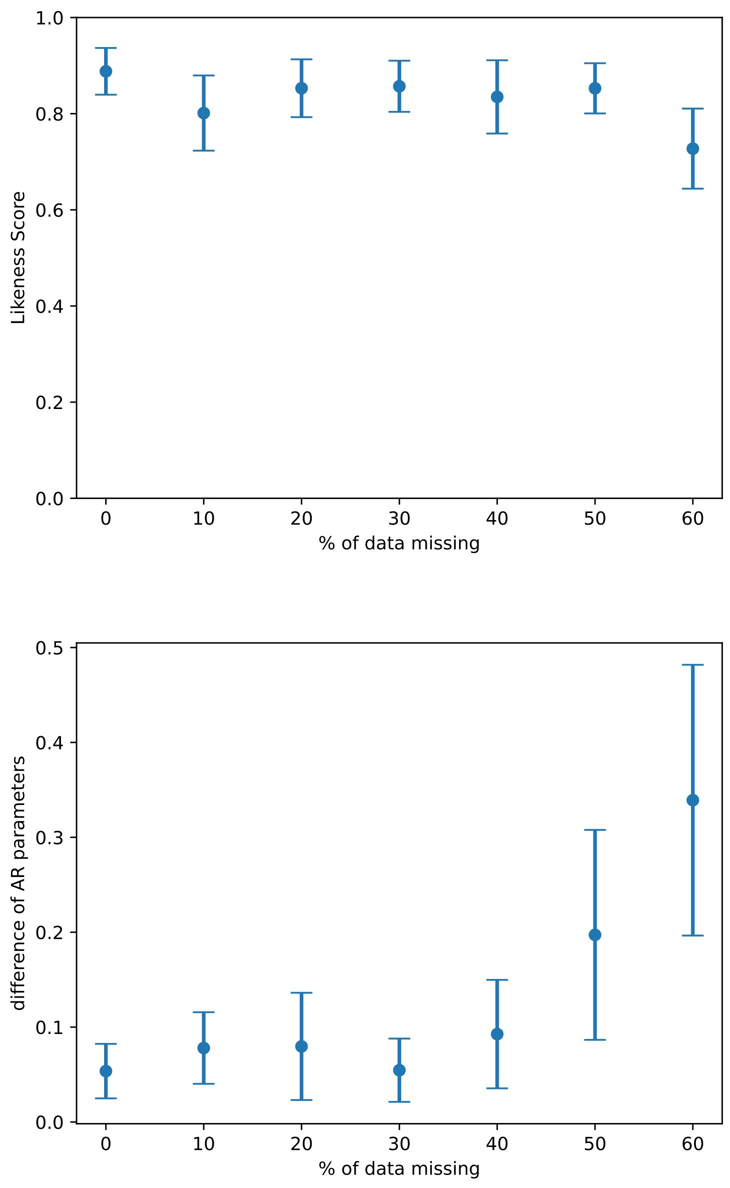


Figure 5.6: Likeness Scores on the bottom panel and AR Coefficient Scores on top panel. The error bars give 95% confidence intervals

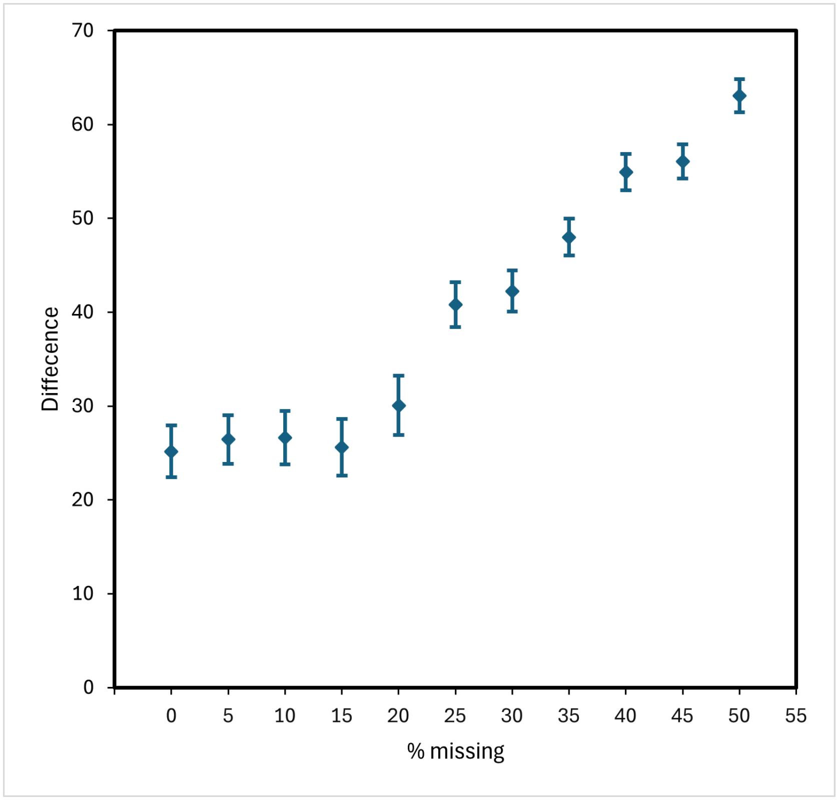


Figure 5.7: Difference between original data and data predicted from regression models of Temperature vs Light based of Force-GAN

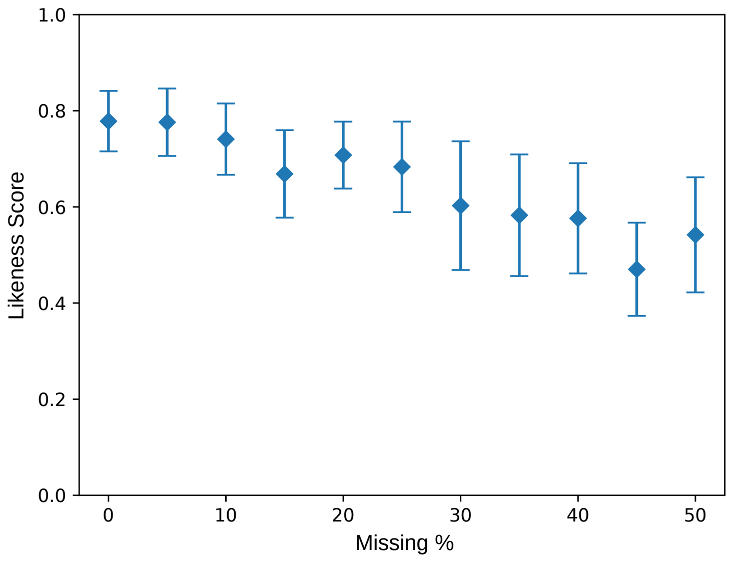


Figure 5.8: Likeness score of Temperature vs Light between original datasets

Chapter 6

Hankel Imputation for time-series data

As discussed in section 2.2.1, natural time series may exhibit missing data like any other data types. For example, instrumentation issues or data recorder difficulties may lead to discontinuities in time series. Many analytical tools require regularly spaced time series, whether simple tools like statistical tests, or more complex tools such as General Adversarial Networks. However, imputing missing data in time series is challenging because of the time dependency between each observation.

Here, a novel time series imputation method was developed that is based on a modified low rank approximation of Hankel Matrices [80] that is named the Hankel Imputation method (HI). Its effectiveness was compared against 5 recognised imputation techniques with varying levels of computational effort. The hypothesis is that this approach will better capture real world time series. This new method aims to capture both trend and noise from the original data when filling in the missing data. This offers the potential for more realistic outputs.

6.1 Contributions

A Hankel matrix is a matrix with constant values along its anti-diagonals and is one way of representing a time series in a matrix format [81].

Let X be a time series with η variables $x_{i,1} \cdots x_{i,\eta}$ where x_i contain n observations $x_{1,} \cdots x_{n,}$ and i is the index of the ith observation.

The Hankel matrix is then defined as follows:

$$H = \begin{pmatrix} x_{1,1} & \cdots & x_{1,\eta} & x_{2,1} & \cdots & x_{2,\eta} & \cdots & x_{k,\eta} \\ x_{2,1} & \cdots & x_{2,\eta} & x_{3,1} & \cdots & x_{3,\eta} & \cdots & x_{k+1,\eta} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{n-k+1,1} & \cdots & x_{n-k+1,\eta} & x_{n-k+2,1} & \cdots & x_{n-k+2,\eta} & \cdots & x_{n,\eta} \end{pmatrix}$$

where k is the lag, and the effects are explained in the parameter tuning section 6.3.

The problem of matrix completion is about completing a matrix where only some values have been observed. As one way of representing a time series is in a matrix format, time series imputation can therefore be solved as a matrix completion problem. Low rank approximation of a matrix is the processes of reducing rank subject to a given constraint. Gillard and Usevich [80], considered a matrix completion using the nuclear norm of a Hankel matrix.

To forecast time series, Gillard and Usevich [80]embedded the time series into a Hankel matrix, which they then enlarge to store data that they want to forecast in the bottom-right hand corner of the matrix. An indicator signals where the forecast needs to start. Last, the nuclear norm (described below) of the Hankel matrix is reduced with the constraint of keeping the difference between the forecast approximation and the original matrix smaller than ϵ.

6.2 Methods

This novel time series imputation method is based on a modified low rank approximation of the Hankel Matrices. The Hankel Imputation method (HI) first embeds the time series into a Hankel matrix. Then the indicator that signals the forecasting start with a mask that signals where the missing data are distributed in the matrix using a similar approach to that developed in MaWGAN (see Chapter 4). The mask function M is defined as:

$$M(x_{i,j}) = \begin{cases} 0 & \text{if } x_{i,j} = \text{is missing} \\ 1 & \text{otherwise} \end{cases}$$

This novel imputation method can be done by solving the following optimising problem:

$$\min \left\| \begin{matrix} y_{1,1} & \cdots & y_{k,\eta} \\ \vdots & \ddots & \vdots \\ y_{n-k+1,1} & \cdots & y_{n,\eta} \end{matrix} \right\|_N$$

Subject to:

$$\left\| \begin{pmatrix} x_{1,1} \cdots x_{n,1} \\ \vdots \ddots \vdots \\ x_{1,\eta} \cdots x_{n,\eta} \end{pmatrix} \odot \begin{pmatrix} M(x_{1,1}) \cdots M(x_{n,1}) \\ \vdots \ddots \vdots \\ M(x_{1,\eta}) \cdots M(x_{n,\eta}) \end{pmatrix} - \begin{pmatrix} y_{1,1} \cdots y_{n,1} \\ \vdots \ddots \vdots \\ y_{1,\eta} \cdots y_{n,\eta} \end{pmatrix} \odot \begin{pmatrix} M(x_{1,1}) \cdots M(x_{n,1}) \\ \vdots \ddots \vdots \\ M(x_{1,\eta}) \cdots M(x_{n,\eta}) \end{pmatrix} \right\|_F \leq \epsilon$$

where X is the Time Series, Y and $y_{i,1} \cdots y_{i,\eta}$ is the estimated data that the optimiser can adjust and creates a model of the time series, n is the length of the time series and k is the lag, ϵ is determines the tolerance , and $\epsilon > 0$, $\odot$ is the pointwise multiplication, $\|_F$ is the Frobenius norm:

$$\|A\|_F = \sqrt{\sum_i^{\eta} \sum_j^{n} [a_{ij}]^2}$$

and $\|A\|_N$ is the Nuclear norm of A which is the sum of the singular values of A (equivalent to the square roots of the eigenvalues of $A^T A$). Ideally the Frobenius norm of the difference matrix (size difference matrix) between the non-missing real data and the estimated data y should be zero. Realistically it is very difficult to get it to zero so a number ϵ close to zero is chosen as the upper bound of size difference matrix. To calculate this size difference matrix, first the missing values are "removed" by applying a mask to the each data x and y by doing piece wise multiplication. Then take the difference of what is remaining which creates a difference matrix. calculate the size of that matrix by using the Frobenius norm, which is the sum of the absolute squares of its elements and the most basic norm. Using the Nuclear Norm for the Low rank approximation has a nice property of being a convex problem which is usually not the case [82]. The problem can be solved with a generalist convex solver (SCS [83])and the code is found in appendix A.5. The values $y_{i,j}$ from the model are used to impute the missing values.

6.2.1 Comparison method

To test the value of the HI method and compare it with existing methods, two approaches were used: two auto-regressive models were created and the models sampled to create two time series, and a real world time series publicly available on the Internet was utilised (6.2.1). For all three time series, values in the time series were randomly extracted to create replicate time series with missing data. To create the two auto-regressive models with different properties, two different parameter sets were used (Figure 6.1):

- Simple AR(3) model defined by the parameters: (0.1,-0.3,0.9)
- A Multi-AR(7) model with the parameters:

$$\begin{pmatrix}
0.6 & 0.22 & 0.13 & 0.02 & 0.05 & 0.003 & 0.0004 \\
0.6 & 0.12 & 0.19 & 0.03 & 0.03 & 0.004 & 0.00041 \\
0.5 & 0.15 & 0.12 & 0.07 & 0.04 & 0.007 & 0.00042 \\
0.6 & 0.13 & 0.19 & 0.04 & 0.03 & 0.003 & 0.00043 \\
0.4 & 0.122 & 0.15 & 0.07 & 0.02 & 0.001 & 0.00044 \\
0.55 & 0.162 & 0.17 & 0.13 & 0.03 & 0.0045 & 0.00045 \\
0.45 & 0.152 & 0.12 & 0.07 & 0.01 & 0.0082 & 0.00046
\end{pmatrix}$$

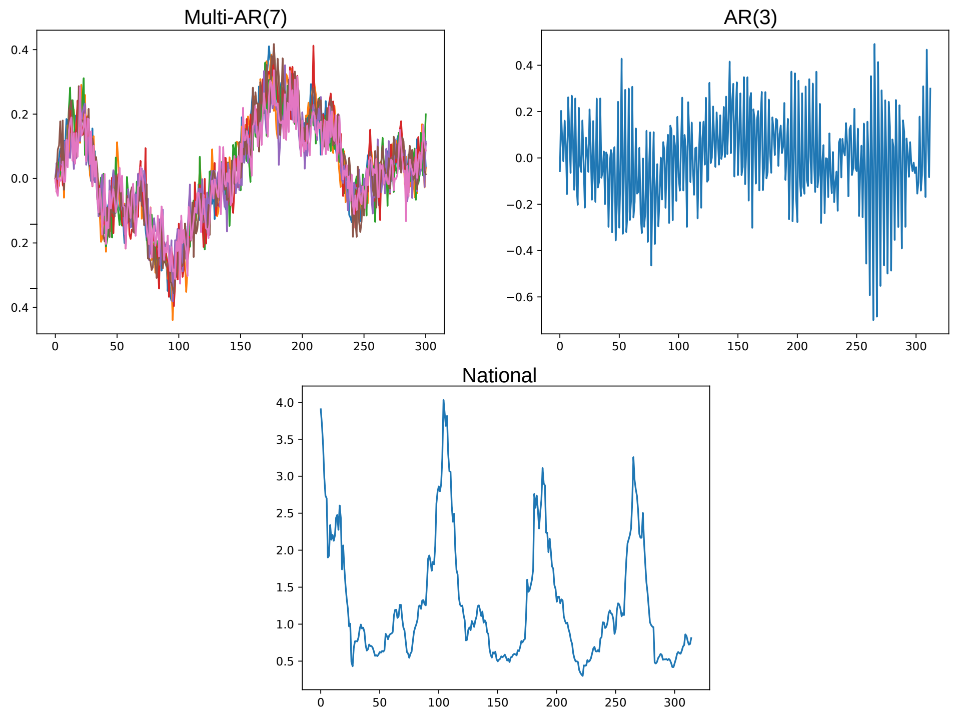


Figure 6.1: Original (no missing data) auto-regressive models and the National signal dataset used to test and compare the imputation methods. Note that the different colours in the Multi-AR(7) represent the different variables of that model.

National signal dataset

For the real time series, a dataset of SARS-CoV-2 levels in wastewater from the Welsh Government Wastewater Surveillance programme was used. A time series of 315 daily data points was extracted from the national signal. The time series provides the average of SARS-CoV-2 genes copies per 100,000 people across 47 wastewater treatment works covering 75% of the Welsh population [84].

The HI method was tested against 5 different imputation methods. These methods were chosen [29] to represent the main approaches often used. This included three interpolation methods: Linear interpolation (Linear), Spline interpolation (Spline), Stineman interpolation (Stine). One method was based on the Kalman filtering: Structural Model and Kalman smoothing (Kalman). One was based on the exponential weighted moving average (EWMA).

6.2.2 Quantifying performance

It is common to decompose a time-series into trend and noise components for forecasting and imputation methods [85]. using least-squares for fitting and/or evaluation discourages the presence of noise. A feature of the Hankel Imputation method is that it picks up both (low-frequency) trend components and (high-frequency) noise components. Accordingly, performance metrics was developed which allow to quantify how well the method matches each component.

To test the performance of all the imputation methods, values were removed in the test time series x using the MCAR process described in chapter 4.5.2 and designated t to be the observations missing, with x' becoming the time series with MCAR values. x' was imputed with the selected imputation method (in section 6.2.1) with y as the outcome timeseries of the imputation.

This new performance measures both trend components and noise components of a single variable. The time-series can be decomposed into $x_n = \tau(x_n) + \epsilon(x_n)$ where the trend $\tau(x_n)$ is a simple smooth with a moving window of $2m + 1$ long:

$$\tau(x_n) = \frac{1}{2m+1} + \sum_{s=n-m}^{n+m} x_s$$

A similar decomposition can be done with imputed values. the imputed data fits the trend was measured using

$$T = \sqrt{\frac{1}{t}\sum_{i=1}^{t}(\tau(x_i) - \tau(y_i))^2}$$

and how well it fits the noise using

$$E = \left| \sqrt{\sum_{i=1}^{t}(\epsilon(x_i)^2} - \sqrt{\sum_{i=1}^{t}(\epsilon(y_i)^2)} \right|$$

these was referd the Trend Score and Noise Score. In both cases the smaller the better. For the multi-AR(7) dataset which has several variables an average score was used.

6.3 Parameter tuning

Hankel Imputation methods have two parameters, the lag (k) and (ϵ). $k \times \eta$ is the length of the rows of the Hankel matrix.

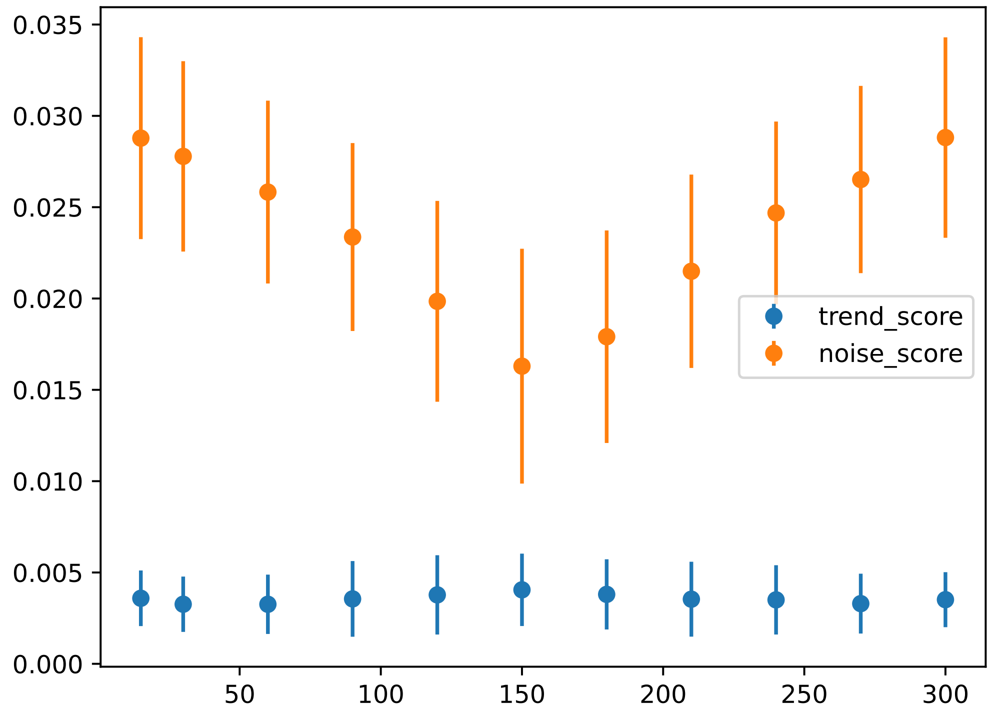
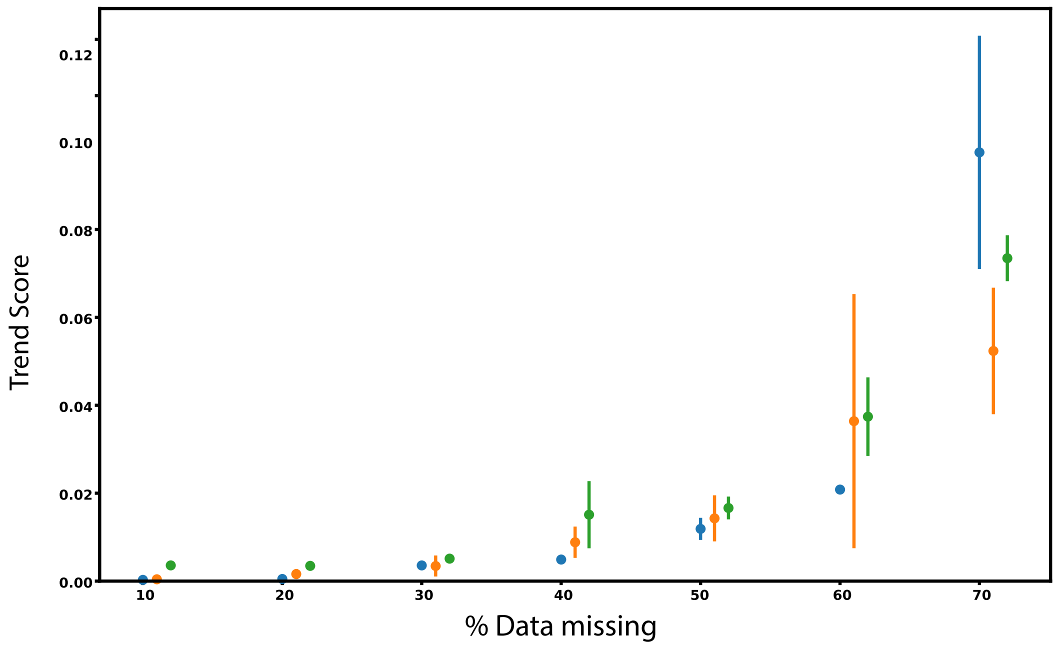


Figure 6.2: Parameter tuning of k

The lag has some impact on the noise imputed. Figure 6.2 illustrates how changes in the lag impact the noise. When the lag is set at $k = 150$, the matrix is approximatively square, and this is when the HI performs the best. This is confirmed by the forecasting method described in [80] which also performed better on square shaped matrices. With the known time series length of 300 in example (figure 6.2), the optimal lag can be deduced as $k = \frac{n+1}{\eta+1}$

The forecasting method described in [80] performed better on square-shaped matrices. The imputation method described here also uses approximatively square-shaped matrices, making $k = \frac{n}{\eta}$. The ϵ parameter indicates the tolerance of the difference between the



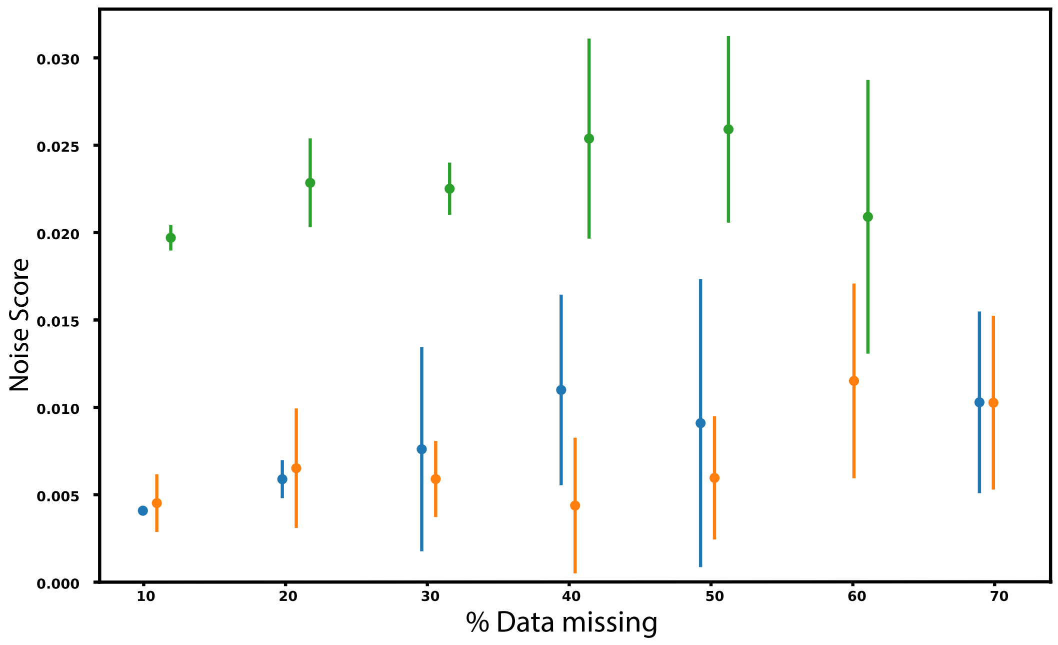
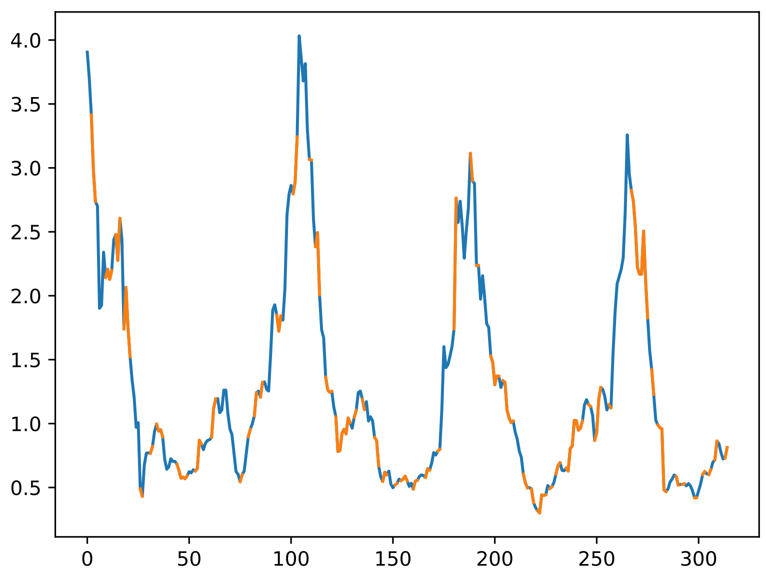


Figure 6.3: Parameter tuning of ϵ using 100 runs. Each point is a mean and bars represent a 95% confidence interval. ϵ=1 are green, ϵ=0.1 are orange, ϵ=0.01 are blue

original and the low-rank approximation. Tests were performed on the National signal dataset to see how the parameters behaved. The tests reveal that there is a trade-off between diminishing returns of accuracy and resource costs. In the tests, decreasing the epsilon parameter improves performance but with diminishing returns as it decreases, and at the expense of increased run time. Figure 6.3 illustrates the effect of changing ϵ using the National signal dataset. If the aim is to capture random noise to yield a realistic model, it is probably not necessary or advantageous to fit the model too closely with the data, therefore higher precision may not be required. The ϵ parameter was chosen as 0.1 because it gives a good balance between accuracy and resource cost.

6.4 Results

For the HI method, the results differ for the three datasets (Figure 6.5). For the Simple 3 Variable Auto regression dataset, the Noise score of the HI was low although not the lowest (the Spline method has the lowest score), and the trend score was the lowest as for the HI. For the Multi-AR(7) dataset, the Noise score of the HI was low compared to the other methods, but the Trend score was average. Finally, for the National signal dataset (see Figure 6.4), the Noise score of the HI was the lowest, and the Trend score was high.



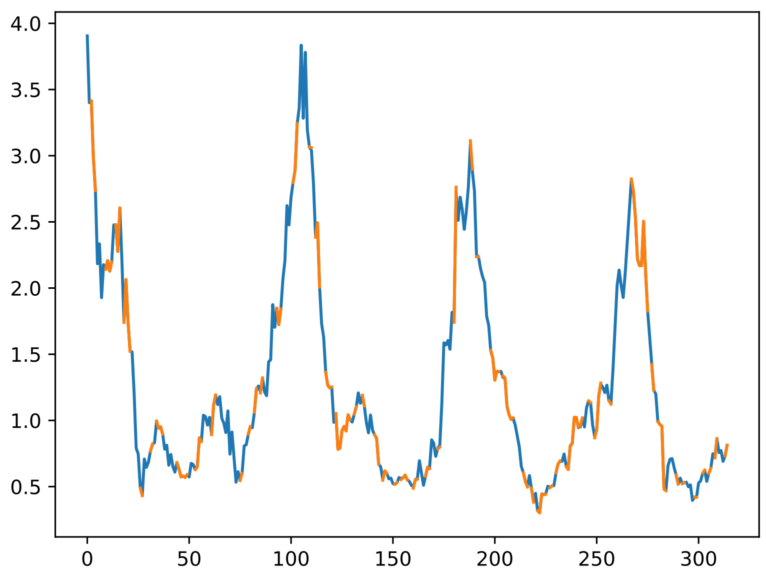


Figure 6.4: HI sample of the National signal dataset. The top plot shows the imputed time series with original values in orange and imputed values in blue. The bottom plot shows the original time series with the values removed in blue and the rest in orange

6.5 Computing efficiency

Longer time series require larger matrices and this dramatically increases the computing resources required. In order to reduce computing cost, the data is split into batches. This section considers how much computing time it saves to the expense of performance. The basis of the Hankel Imputation method is the Hankel matrix H that encodes the time-series.

This is an effective performance of a singular value decomposition H, with computational cost $O((n^{0.5})^3) = O(n^{1.5})$. Thus, if n half and perform two smaller imputations the computation cost expected will scale by $2 \times 2^{-1.5} = 2^{-0.5} = 0.707$ (3 significant figures). The purpose of the experiment was to compare the computational saving to the loss in performance when using batches. For the experiment, the univariate

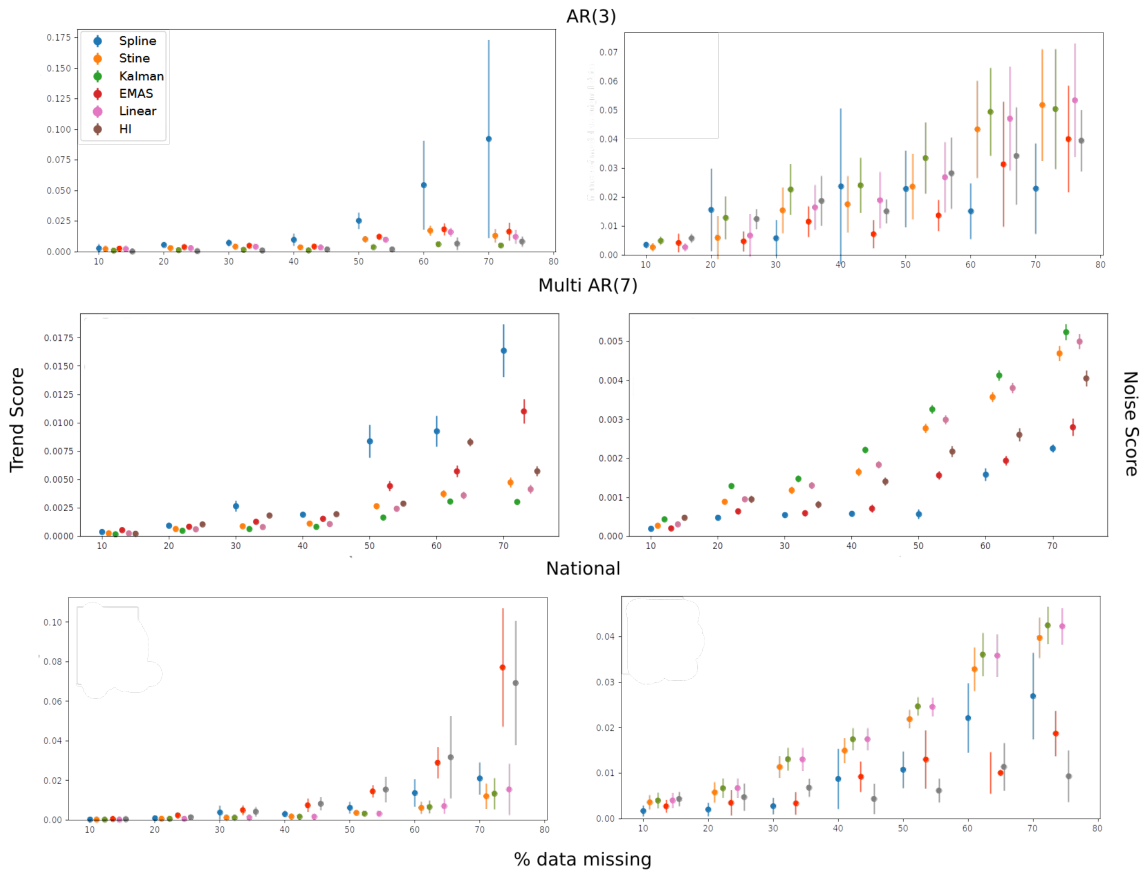


Figure 6.5: Noise score and Trend score of all 3 datasets and 6 different imputing methods (Linear:pink, Spline:blue, Stine:orange, Kalman:green EWMA:red, HI:grey)

National signal dataset time-series was used.

The missing observations were chosen uniformly at random and were nested, so that the observations missing at the 10% level were also missing and the 20% level, and so on. For each of these data sets, imputation was carried out using batches of size 208 (all the data), 104, 52, 26 and 13. The results are given in Figure 6.6: the top panel gives the Trend Score squared, the middle panel the Noise Score, and the bottom panel the computation time. Each point is the average of 10 experiments (each with different sets of missing data) and the vertical bars give 95% confidence intervals for the mean. At all levels of missingness, there is a loss in performance when using smaller batch sizes, though with a reduction in computing time. In conclusion, a large batch size as time allows is recommended.

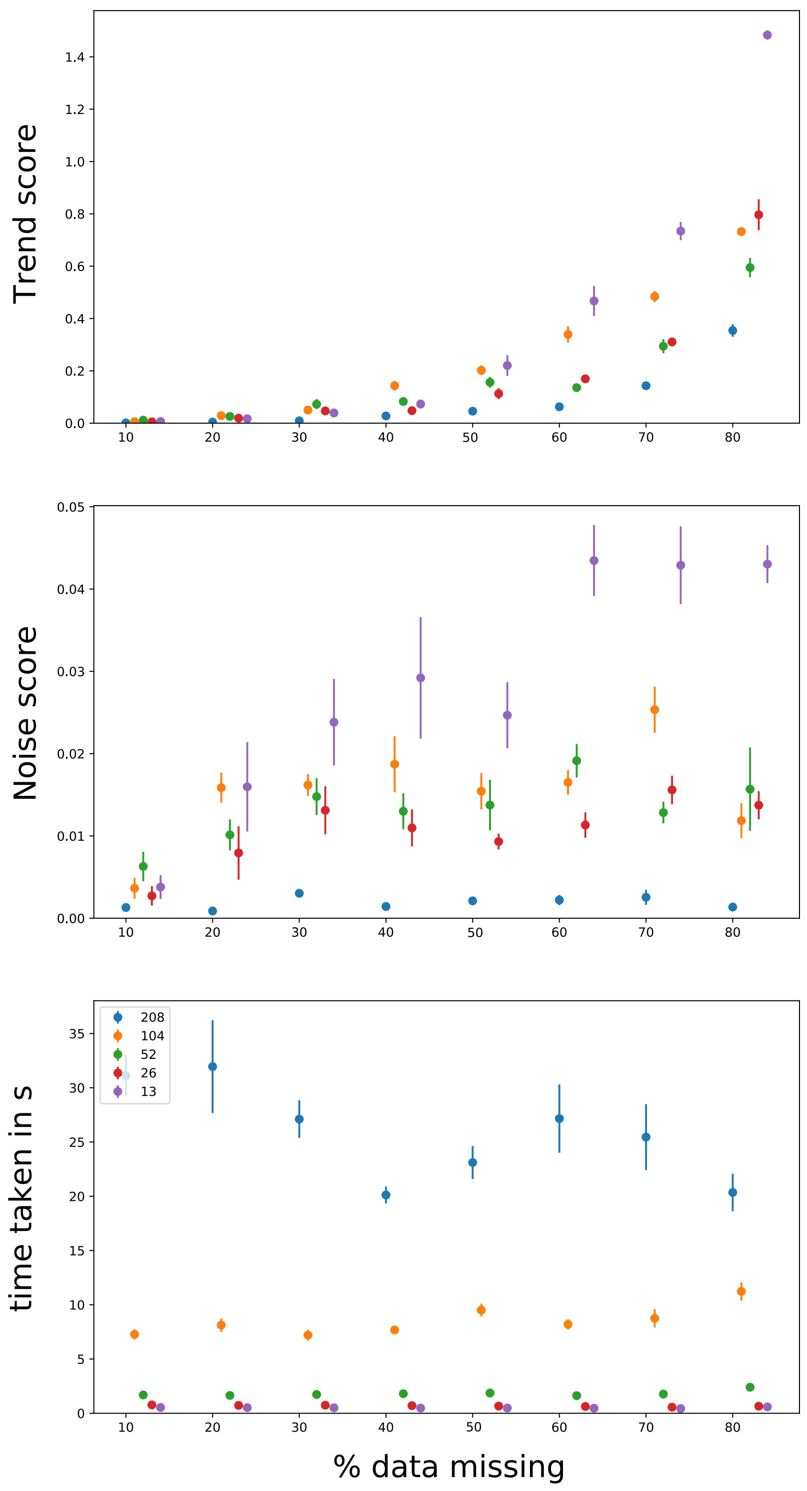


Figure 6.6: Relations between the different batch sizes and the amount of different % of missing values in terms of time taken, noise score and trend score in the National signal data set

6.6 Discussion

This Chapter aimed to test the hypothesis that the novel imputation technique for time series based on a modified low rank approximation of the Hankel Matrices, the HI method, would better capture the time series than existing techniques based on smoothing and thus, simplifying data. The performance of HI is visually very good and performs well using the scores. The fit between the imputed time series and the original time series is good overall but the advantage of the method is most significant when the dataset is noisy. Current imputation methods perform very well for simple datasets with low noise. Given that many time series in the real world tend to be noisy and have missing data, this novel technique provides an efficient way to impute realistic time series. This aspect is important when the imputed data need to be used in complex AI models such as GANs.

Chapter 7

Case study: DCWW water data

7.1 Introduction

7.1.1 Issues in water quality

To guarantee the safety and quality of drinking water, water companies treat the water they extract from rivers, groundwater, lakes and reservoirs using various physical, chemical, and biological processes [86]. Customers often rely on taste and odour to estimate the quality of their water [87]. The main factors worldwide for taste and odour issues are two compounds found naturally in water: Geosmin and 2-MIB [9].

Although these compounds pose no risk to human health [88], they are a main cause of customer dissatisfaction [89].Removing them however is challenging because they are highly resistant to conventional drinking water treatment processes like coagulation, sedimentation and filtration [90, 91]. This means that additional treatment is necessary such as the use of activated carbon. These treatments are very costly and therefore prevention is preferred [92]. Prevention requires understanding the mechanisms that trigger the production of these two compounds.

Geosmin ($C_{12}H_{22}O$) and 2-MIB ($C_{11}H_{20}O$) are produced by algae, particularly Cyanobacteria when they bloom and decay [93]. Over the past two decades, much research has investigated the source of these events. Potential triggers include temperature, nutrient concentrations [94] such as ammonium and phosphorous, and other water chemicals such as ferrous iron and sulphate [95]. These events occur naturally but are normally quite rare. Climate change, namely warmer summers, tend to favour increase in algal biomass, and this means the two compounds responsible for taste and odour increase significantly creating taste and odour events that are becoming a challenge for water companies [96].

Therefore, water companies are increasingly seeking models that would allow them to mitigate or prepare for such events. Prediction models require large datasets to be able to reliably predict these events. However, monitoring of water quality is costly as it requires manual sampling and then a laboratory analysis [97]. As a consequence, water companies often only have small data sets, with infrequent data that only capture a small amount of events over a relatively small period of time.

7.1.2 Rare events

Rare events occur at low frequency, which means that the dataset associated with them is unbalanced (section 2.2.3). In this case, this means that there is a minor category of data that relates to rare events, and a major category of data that relate to non-events.

This is a big issue as infrequent events often have a widespread effect that might destabilise the system. For example, Geosmin and 2-MIB outbreaks are infrequent but the cost for water companies of additional treatments to address the event are high.

Synthetic data generation is an approach that aims to overcome imbalance in an original dataset by artificially generating data samples. Generative Adversarial Networks (GANs) could be a good candidate to address rare event challenges as they produce quality datasets.

This Chapter aims to test the value of GANs to solve the challenge set in this Chapter by synthesizing rare events for data augmentation. Using the case of Geosmin and 2-MIB fluctuations in drinking water abstracted from a drinking water reservoir, this chapter investigates the benefits of using GANs in an industrial context. More specifically, the aim is to answer the question: What does the analysis of a dataset generated with a GAN teach us that the analysis of the original data does not?

To answer the question, a dataset spanning 4 years from a reservoir in Wales (UK) was provided by DCWW (Dwr Cymru Welsh Water). It is the largest water company in Wales serving more than 3 million customers. The dataset includes time series of Geosmin and 2-MIB along with a range of 14 chemical, biological and physical variables that are thought to be associated with Taste and Odour (T&O) events [98].

Because the sampling is done manually, the dataset poses several challenges for modeling. First, the timeseries provided have different time steps for each variable, where some variables are taken on average every fortnight, and some variables are taken daily. This means that the dataset itself is a set of irregular timeseries which causes problems in most modelling approaches. Second, the dataset is unbalanced because it has a few rare events because there are only 26 instances where Geosmin or 2-MIB are high over the 4 years. In summary, the challenge of this dataset is that it is irregular between variables and within variables, and unbalanced.

7.2 Method

7.2.1 The dataset

The dataset provided by DCWW includes 16 quantitative variables: Geosmin and 2MIB, as well as 14 other environmental variables. The timeseries cover the period 25/11/2012 to 12/08/2018. This includes 881 data points for Geosmin and 880 data points for 2-MIB. Other variables have fewer data points, with on average 246 data points. The 14 environmental variables (see table 7.1) cover physical measures (conductivity, temperature, turbidity), chemical measures (aluminium, ammonium, iron, manganese, nitrate, nitrite, phosphorus, pH) and biological measures (Chlorophyll A, total algae). Another issue of the dataset is that the instruments used to measure the variables have limits of detection. Some of the measurements are below the limits of detection of the instruments. In these cases, half of the lowest detection value was assigned to the data point.

Variables	Unit	Count	Mean	Std	Min	Median	Max
Aluminium	mg/L	281	0.068	0.051	0.017	0.058	0.620
Ammonium (as N)	mg/L	279	0.009	0.006	0.003	0.008	0.054
Chlorophyll A	$\mu g/L$	273	7.303	7.582	0.630	4.600	55.460
Conductivity	$\mu S/cm$	283	176.878	36.613	7.600	180.000	520.000
Geosmin	ng/L	881	3.533	2.463	0.300	3.000	24.000
Iron	mg/L	283	0.279	0.183	0.060	0.211	1.200
Manganese	mg/L	284	0.042	0.057	0.007	0.029	0.810
2-MIB	ng/L	880	2.525	2.885	0.570	1.400	19.000
Nitrate (as N)	mg/L	282	2.090	1.306	0.500	1.800	5.529
Nitrite (as N)	mg/L	283	0.006	0.004	0.002	0.004	0.041
Phosphorus	mg/L	159	0.047	0.008	0.020	0.043	0.094
Temperature	$°C$	287	10.713	4.375	3.100	10.600	18.500
Total algae	—	274	3838.612	5762.124	0.000	1708.500	46569.600
Total pesticides	$\mu g/L$	67	0.014	0.022	0.000	0.004	0.080
Turbidity	NTU	282	2.623	1.467	0.100	2.300	13.000
pH	pH	282	7.542	0.172	7.000	7.500	9.000

Table 7.1: Environmental variables measured by DCWW to assess water quality

7.2.2 Defining a T&O event

Geosmin and 2-MIB can be sensed at low concentrations. Detection by humans varies between individuals and with the overall chemistry of the water. For Geosmin, customer complaints start at 12 ngL^{-1} and for 2-MIB start at 7 ngL^{-1} [99]. However, water companies need to start to take action before the complaints start. Dwr Cymru Welsh Water start to increase their water sampling frequency when Geosmin is $> 5ngL^{-1}$ and 2-MIB is $> 2.5ngL^{-1}$, in line with the Drinking Water Directive [100]. These values also correspond roughly to human detection thresholds for taste and odour of these two compounds [101, 102]. The cumulative frequency distribution of the two compounds

(Figure 7.1) also shows that these values correspond roughly to a breakpoint occurring around the 80% cumulative frequency.

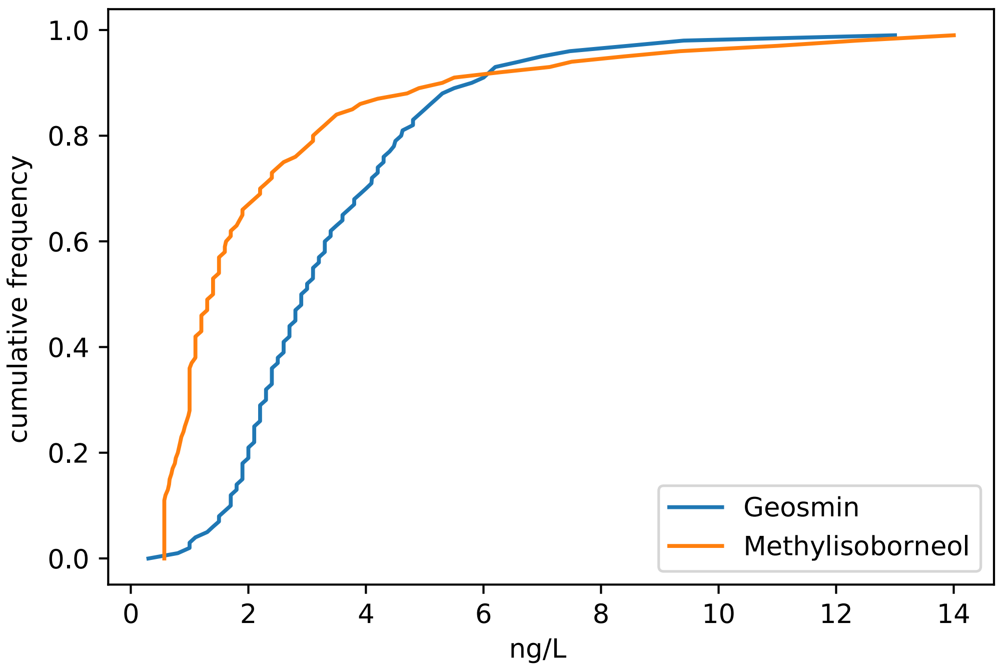


Figure 7.1: Cumulative frequency distribution for sampled levels of Geosmin and 2-MIB expressed in ng/l. The line corresponds roughly to a breaking point in frequency distribution.

When concentrations are high reaching $10ngL^{-1}$ for Geosmin or $5ngL^{-1}$ for 2-MIB, DCWW turns on Activated Carbon filters if they have some. If not, they dose the water with Powdered Activated Carbon at low levels and check customer complaints daily. They only go back to normal when concentrations go below these thresholds.

For this chapter, and based on the above, levels of the two compounds were categorised into three categories (ζ):

$$\zeta = \begin{cases} 2 & \text{if Geosmin} > 10 \text{ or 2MIB} > 5 \\ 1 & \text{if Geosmin} > 5 \text{ or 2MIB} > 2.5 \\ 0 & \text{otherwise} \end{cases} \tag{7.1}$$

Category 2 represents a T&O 'event' and corresponds to the start of customer complaints, category 1 is when DCWW are on 'alert' and increase their sampling frequency, and category 0 represents 'normal' operating conditions.

7.2.3 Analysis

To test the aims of this Chapter, the data was processed using the following steps:

1. Convert from irregular to regular time series
2. Use the Hankel Imputation to impute the missing values see Chapter 6.3). The parameter ϵ set to 1 as it is the best preforming based on the result in figure 6.3
3. Create synthetic data using Time-GAN (see section 3.5)
4. Use a model and cross models on both the imputed and the generated synthetic dataset to gage the effectiveness of this approach

Data discretization and imputation:

Paerl et al. [103] demonstrated that a significant change in Geosmin concentration can occur on a week-to-week basis during spring and summer. Therefore, in this study, the irregular time series is transformed into a regular time series using a weekly frequency.

After transforming to a regular time series, the raw data contained 1140 missing values out of 5382 and therefore needed to be imputed. The data was filled using the H.I method. The H.I method was developed to impute data where it is important to keep the noise as well as the trend and is detailed in Chapter 6.

Data augmentation:

The data was augmented using TimeGAN. TimeGAN is a variation of GAN adapted to time series. TimeGAN was developed by Yoon et al [54]. The Time-GAN model requires a time series with regular time steps.

The GAN generates datasets in the same proportions as the original data, so the relative proportion of categories is maintained. To obtain a more balanced dataset, one with equal amounts of each category, the excess of 'normal' (category 0) data was removed to focus on events. Removal was performed so that every category had more or less of the same amount of data. This is referenced in this Chapter as 'trimming'.

Modelling:

A linear model was first fitted to the data with δ as response variables and the four PCA components as the independent variables, but it proved to have little predictive power. Since there was little evidence of linear relationships, a non-linear model was considered. Among the different options (Table 7.2) Random Forest was chosen for this because it can model non-linear relationships with low computing cost.

Table 7.2: Advantages and disadvantages of different categorical models

Name	Advantage	Disadvantage
CatReg	Easy to compute	Only model linear Relationship
Random Forest	Model non-linear Relationship	Doesn't account for time dependency
Neural network	Flexible	Resource Intensive and Complex parameters

A Random Forest is a group of decision trees with randomly selected data points. The output of the Random Forest is the mode of all the decision trees output. The idea is that it is not as affected by changes in the dataset as a single decision tree, and therefore more robust and less prone to over fitting. Decision trees model categorical data by a sequence of nodes; each node determines which category each observation is by deducing if the variables are above the threshold or not. A Random Forest of the DCWW data was run to predict the event categories using the environmental variables.

To compare the outputs of the Random Forest analysis, an accuracy vector was used. The accuracy vector (AcV) contains the proportion of each category that is mis-predicted.

This is calculated by:

$$\text{AcV} = [v_0, v_1, v_2]$$

$$v_k = \frac{|\{i : P_i \neq A_i, A_i = k\}|}{|\{i : A_i = k\}|}$$

where A_i is the actual category, P_i is the predicted category for observation i.

Since the amount of data available is limited, the comparison is done by splitting the data into a test and train split that is randomly chosen. The training data is used to train the Random Forest decision tree and the test data is used to compare the output of the Random Forest with the known classification. In the software used (Sklearn python library), the number of estimators used by the Random Forest is by default set to 100, and this number was retained. As a random split is used, a Monte Carlo experiment is performed 100 times to reduce the variance linked to the split.

Cross models were used to understand the strength of the models. Cross models are models generated by a dataset to predict another dataset. Three cross-models were tested: the random forest model created using the GAN generated dataset is used to predict the Filled data set (G$\rightarrow$F), the random forest model created using the Filled dataset is used to predict the Raw irregular (Original) dataset with the missing data removed (F$\rightarrow$I), and finally, the random forest model created using the GAN generated dataset is used to predict the Original data (G$\rightarrow$I).

Because of the large array of environmental variables, and the fact that some of them are partially correlated, the variables were first processed through a dimension reduction analysis. A simple Principal Components Analysis was done on the non-standardized data using the correlation matrix to identify a smaller number of components that explain most of the variance observed in the 14 variables (See Table 7.3). This approach is equivalent to a standardised PCA using a covariance matrix [104]. Four components were retained that together explain 67% of the variance observed. Further components only added minimal additional explanatory power (i.e. less than 5%).

Table 7.3: PCA of 13 variables measured to assess water quality, the green showing positive weight on each components and the blue showing negative weights.

	Component			
	1	2	3	4
Aluminium	0.357	0.850	-0.141	-0.175
AmmoniumasN	-0.072	0.063	0.101	0.599
ChlorophyllA	-0.050	0.305	0.827	0.157
Conductivity	-0.807	0.228	-0.067	-0.069
Iron	0.890	0.172	-0.015	-0.018
Manganese	0.717	0.356	-0.011	-0.142
NitrateasN	-0.746	0.555	-0.095	-0.038
NitriteasN	-0.121	0.407	-0.454	0.450
Temperature	0.729	-0.419	0.077	0.141
Totalalgae	-0.024	0.094	0.761	0.202
Turbidity	0.324	0.824	0.018	-0.068
pH	-0.144	0.132	0.601	-0.144
Phosphorus	-0.136	-0.110	0.126	-0.773

The First Principal Component (25% of variance) highlights a scale from high Conductivity and Nitrate levels on the negative side of the scale to high Manganese, Iron and Temperature levels on the positive side. The Second Principal Component (18% variance) represents high Turbidity and Aluminium levels on the positive side of the scale. The third Principal Component (14 % variance) captures the biological measures Total algae and chlorophyll as well as pH. Finally, the Fourth Principal Component (10% variance) represents mostly Phosphorus and Ammonium.

7.3 Results

7.3.1 Performance of the model

The performance of the Hankel Imputation is measured using trend and noise scores described in section 6.2.2. The performance of the Hankel Imputation was reasonable despite the fact that this method had never been tested before with such extensive missing values in the dataset.

The overall noise score was 0.0234. Compared with the "Nat" database described in Chapter 6.2.2, this could be interpreted as a good score (in the Nat database the better scores were close to 0.02). However, comparisons with other datasets may offer limited validation. Similarly the trend score was 0.0322 which also could be interpreted as a good score relative to the Nat database. These interpretations seem however to concur with the results in section 6.4.

Figure 7.2 shows the breakdown of trend and noise scores per variable. Phosphorus shows the worst trend and noise scores, and this is perhaps related to the fact that many Phosphorus data points were below the detection limit. Geosmin and 2-MIB also show poor scores. This is probably due to the presence of events in the data. Variables with low levels of missing values, or no events, such as Iron and Ammmonium, scored highly.

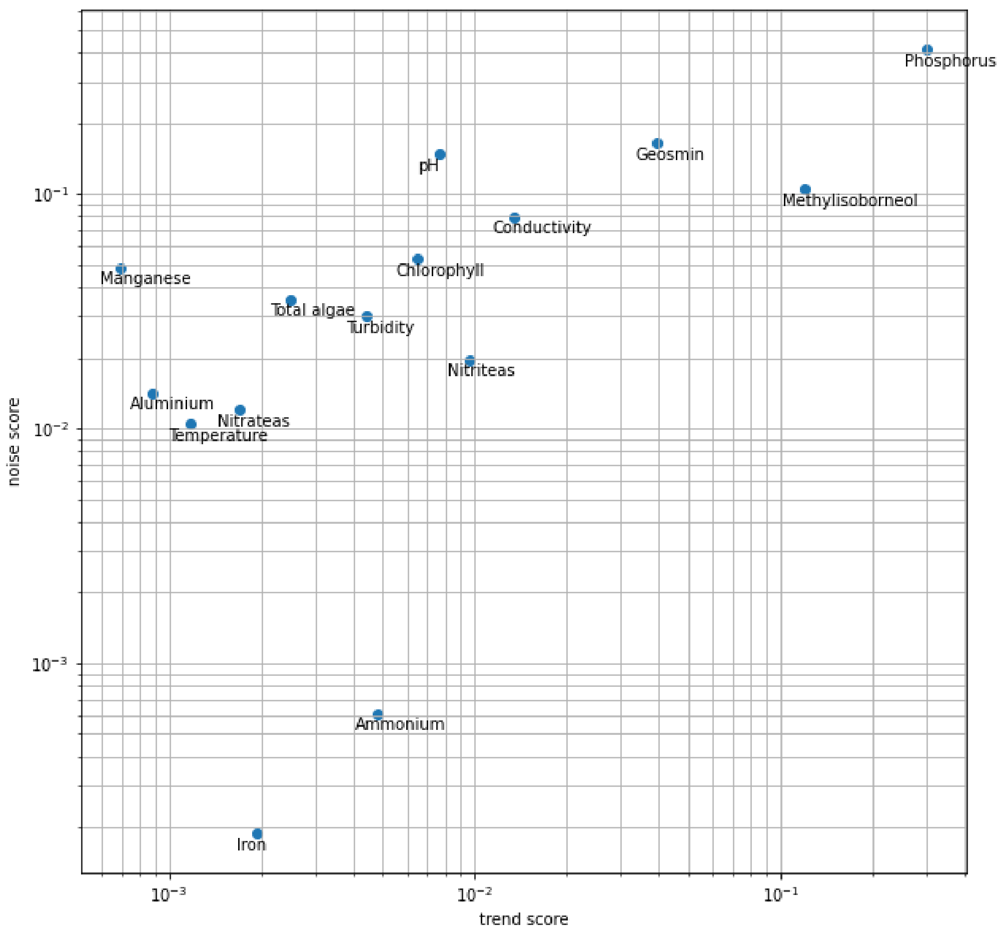


Figure 7.2: Breakdown of Trend and Noise scores per variable

The performance of the TimeGAN model was evaluated using the Likeness score described in section 3.7.3. The Likeness score of the generated data is 0.91 which is good as within the 0.9-1 bracket. The breakdown of the score is: $S_x = 6.45$, $S_y = 5.84$ and $S_{x,y} = 6.26$.

7.3.2 Analysis of data

Table 7.4 and 7.5 show the average proportions of incorrect classification of ζ at each stage described in section 7.2.3 for Geosmin events and 2-MIB events, respectively.

The model based on irregular raw data marked as "Original" shows the decision tree classifies "class 0" well. However, the decision tree classifies class 1&2 incorrectly, which means that the imbalance of the dataset is significant. The model based on Hankel imputed data marked as "Filled" is already an improvement. The model based on GAN's untrimmed synthetic data marked as "Generated" shows that the decision tree classifies all the classes well. The cross-models (Table 7.4 and 7.5) show that using a GAN is beneficial but only when the generated data is trimmed. The "Untrimmed G$\rightarrow$I" model (Figure 7.4) shows an equal performance to the "Original" model with all incorrect predictions of "class 2", while the Trimmed model decreases performance down by a small amount. For the 2-MIB dataset (Figure 7.5), the "Untrimmed G$\rightarrow$I" model shows a slightly better performance in predicting "class 2", compared to the "Original" model. However, the "Trimmed" model shows a more significant decrease. The cross-models also show that although the GAN augmentation used to predict 'Original' or 'Filled' datasets improves the classification of class 1&2, it loses some ability to classify "class 0".

Table 7.4: Average of incorrect classification for each variable, at each stage, for Geosmin, with the yellow highlight showing the best comapring to the original dataset

Model	$\zeta = 0$	$\zeta = 1$	$\zeta = 2$	overall
Irregular (I)	0.027 ± 0.029	0.929 ± 0.099	1 ± 0	0.652 ± 0.043
F→I	0.056 ± 0.03	0.877 ± 0.104	1 ± 0	0.644 ± 0.045
Untrimmed G→I	0.131 ± 0.04	0.795 ± 0.132	1 ± 0	0.642 ± 0.057
Trimmed G→I	0.33 ± 0.056	0.532 ± 0.158	0.768 ± 0.352	0.544 ± 0.189
Filled (F)	0.022 ± 0.018	0.892 ± 0.07	0.932 ± 0.119	0.615 ± 0.069
Generated (G)	0.018 ± 0.002	0.169 ± 0.012	0.209 ± 0.021	0.132 ± 0.012
Untrimmed G→F	0.118 ± 0.026	0.774 ± 0.077	0.89 ± 0.119	0.594 ± 0.074
Trimmed G→F	0.276 ± 0.032	0.546 ± 0.109	0.852 ± 0.139	0.558 ± 0.093

Table 7.5: Average of incorrect classification for each variable, at each stage, for 2-MIB with the yellow highlight showing the best comapring to the original dataset

Model	$\zeta = 0$	$\zeta = 1$	$\zeta = 2$	overall
Irregular (I)	0.082 ± 0.052	0.977 ± 0.052	0.803 ± 0.133	0.62 ± 0.079
F→I	0.236 ± 0.05	0.845 ± 0.123	0.416 ± 0.172	0.499 ± 0.115
Untrimmed G→I	0.331 ± 0.059	0.689 ± 0.12	0.763 ± 0.125	0.594 ± 0.102
Trimmed G→I	0.422 ± 0.062	0.601 ± 0.133	0.682 ± 0.156	0.568 ± 0.117
Filled (F)	0.189 ± 0.055	0.764 ± 0.101	0.618 ± 0.092	0.524 ± 0.083
Generated (G)	0.045 ± 0.004	0.094 ± 0.007	0.084 ± 0.007	0.074 ± 0.006
Untrimmed G→F	0.323 ± 0.042	0.636 ± 0.089	0.557 ± 0.084	0.505 ± 0.072
Trimmed G→F	0.419 ± 0.054	0.519 ± 0.084	0.483 ± 0.082	0.474 ± 0.073

7.4 Discussion

This Chapter aimed to test the value of GANs in a real-world application: the augmentation of irregular time series for event management. Results show that datasets that were filled using an imputation model that conserves noise (Hankel Imputation developed in Chapter 6) already provided a better dataset to predict events. Generated datasets through a GAN provided an even better way of augmenting events in time series, once the datasets had been filled.

The cross models using generated data to predict filled data did not perform as well as the "Generated" non-cross model. This usually means that the GAN used (TimeGAN) needed more training. However, significant efforts to optimise the parameters did not yield a better outcome. This therefore suggest that TimeGAN might not be the most adapted GAN for this purpose, and that a time-dependent GAN that takes account of events could deliver better outcomes. Another consideration, is that the Random Forest algorithm used to classify events, while computer resource efficient, might not be very well adapted to the dependencies of time series. Another option would have been to use a RNN - Recurrent Neural Network- but these are extremely computer resource demanding which is an issue when dealing with a Monte Carlo experiment like the one used here.

The difference of performance between Geosmin events and 2-MIB events means that the number of events in the Original data (8 for Geosmin compared to 18 for 2-MIB) makes a significant difference. This means that efforts to increase the number of events, for example by aggregating data from more sites or by collecting data over a longer period of time, would be beneficial. Time series datasets in real life are often limited in time because of the costs of monitoring and also often have missing values when instrumentation files or when field sampling is not possible. While the solution provided here works relatively well for near-complete datasets, it works less well with very incomplete datasets. This confirms findings in Chapter 4.

While possibly not optimised, the solution proposed here is very useful. First, it means that historical data owned by companies can be utilised to understand events even when they are very patchy since HI imputation delivers a much better set of data that the original raw dataset for modelling purposes. Second, augmentation of data using the GAN proposed here also allows to produce more accurate models, although it is important to note that the efficiency of this solution increases with the original number of events in the dataset. Indeed, with a balance of 18 event points to 281 non-event points 2-MIB augmentation performed better that Geosmin with 8 event points to 291 non-event points.

Chapter 8

Discussion

The aim of this thesis was to utilise GANs, a form of machine learning, to solve problems with complex real-world data. This chapter summarises the research carried out and the methods used to answer the different research questions highlighted in Chapter 1. The Chapter then identifies what are the main contributions of this thesis to the field, the limitations of the work, the impact the work is likely to have, and thoughts on future work.

8.1 Research highlights and limitations

The Introduction to the thesis (Chapter 1) highlights the challenges of 'real-world' data, namely that they are often datasets with missing values, time dependencies and rare events which makes prediction models hard to develop. The structure of the thesis reflects the need to address these different challenges. The main research question addressed in the thesis was: How can data synthesis be used to provide better models for diagnosing or predicting risk situations when datasets have challenges such as missing values, time dependencies or rare events? This is a complex problem that was broken down to address independently the three challenges.

Chapter 2 investigated the challenge through a literature overview of data synthesis and its applications. It asked the following questions: What data synthesis techniques are available for datasets with missing values, time dependencies, and rare events, and what are their limitations? It identified the evidence to support the exploration of GANs since GAN architectures can augment unbalanced data with high fidelity and can generate data quickly and infinitely many times.

Chapter 3 investigated further the literature around GANs and provided more detailed insights into the benefits and limitations of GANs for a range of datasets. The chapter highlights that: i) GANs are complex models with multiple parameters that need optimising, that can easily overfit or collapse, ii) some (like WGAN) have been developed to reduce these risks, iii) GAN models can be adapted for datasets that range from image to time series, iv) there are several ways to evaluate the performance of these models, and the most reliable are used throughout this thesis. The chapter also describes the GAN code that was developed in this thesis and that was used as a baseline for the rest of the thesis.

Chapter 4 explored the first challenge of running GANs on data with missing values. As most models, GANs fail to work with missing values. A novel GAN called MaWGAN (Masked Wasserstein GAN) was therefore created, which generates synthetic data directly from datasets with missing values. MaWGAN introduces a novel methodology for comparing the generator output with the original data that does not require to discard incomplete observations. It is based on a modification of the Wasserstein distance and is easily implemented using masks generated from the pattern of missing data in the original dataset. Numerical experiments demonstrated the superior performance of MaWGAN compared to (a) discarding incomplete observations before using a GAN, and (b) imputing missing values (for example using the GAIN algorithm) before using a GAN. MaWGAN handled missing data, but its main limitations lies in that it works on the assumption that the observations are independent.

Chapter 5 investigates the challenge of time dependencies building on the capabilities to deal with missing data developed in of Chapter 4. A third neural network was embedded in the MaWGAN architecture, that predicts what comes next in the sequence and therefore captures the dependency in the data. The new algorithm developed, called Force-GAN, succeeded in generating synthetic sequences from simple regular time-series with missing values. While promising, Force-GAN had limitations as it did not perform well with more complex time series, and was therefore not the right tool for real-world data challenge of this thesis.

Chapter 6 explored an alternative route, where missing data in the irregular time series were first imputed with a novel technique that preserved the noise in the data, before being run through a GAN. The imputation method developed in this thesis, called Hankel Imputation (HI), was based on a low-rank completion of Hankel Matrices (LDHM) which has previously been used to forecast time series. Unlike other smoothing techniques which fill in missing data with only the signal data, HI captures noise in the time series and therefore better imputes complex time series like the ones this thesis focused on. Limitations to this solution are that with large datasets, processing times are too high and thus require a break down into batches which may affect accurate generation of time dependencies.

Chapter 7 used the insight gained from the shortcoming of Chapters 4 and 5 and the success of Chapter 6 to address the full problem outlined in Chapter 1. To illustrate the approach, data from DCWW on Taste and Odour issues in drinking water were used. Chapter 7 illustrated that GANs can be developed to handle complex real world datasets with missing values, time dependencies and rare events. The work also demonstrates the value of using GAN and HI to augment datasets to solve the unbalanced nature of event datasets.

8.2 Research contributions and insights

This thesis has provided novel contributions to the field of GANs and its application to tabular data. The main contributions are:

1. The literature reviews in Chapter 2 and 3 gave an non-exhaustive overview of current knowledge on Data synthesis with a focus on GANs. It provided novel insight into the limited research on tabular data, especially real world data with missing values and time series, that have guided this thesis.

2. The development of a novel GAN in Chapter 4 that can handle missing values in the dataset. This allowed the GAN to learn a distribution directly without

letting imputation or removal processes influence the training. This was published as "MaWGAN: A generative adversarial network to create synthetic data from datasets with missing data" in *Electronics Journal, 11(6),* in 2022. This paper was based on the work in Chapter 4, and at time of writing cited 9 times.

3. A Novel algorithm was developed in Chapter 5 to expand the uses of MaWGAN to handle time series. This work highlighted an RNN approach was not a suitable option (negative result), but that the forecaster approach allowed basic time series with missing values to be synthesised through a GAN without imputation. Some of the results were presented at a Conference and published in the proceedings as "Synthesis of time-series with missing observations using generative adversarial networks" in *Proceedings of the 34th Panhellenic Statistics Conference Greek Statistical Institute* in 2023.

4. A method to impute data while keeping the original noise of the dataset was developed in Chapter 6, expanding on Gillard and Usevich [80] who used Hankel matrices to forecast time series. This is important for models that require both noise and trend to be reproduced during the imputation process.

5. By bringing together the imputation model developed in Chapter 6 and a known GAN for time series identified in Chapter 3, Chapter 7 demonstrated that GANs can have a significant value to predict rare events in time series. It also demonstrated that GANs can be used for data with unbalanced datasets with missing values.

8.3 Applied limitations linked to the GAN approach

Besides the limitations described in the previous section, this thesis also highlights the technical challenges linked to GAN-based approaches that limit the widespread application of these solutions. Technically, the use of GANs to synthesise data is challenging for the non-expert. First, the use of GAN requires knowledge of python or any other code language. This is an issue that was addressed by creating an interface (see Chapter 3) that removed the requirement of any coding language. However, this did not remove the need for basic knowledge of how GAN parameters work. As a consequence, this means the solutions proposed in this thesis are not immerdiately accesible for the non-experts.

For experts, additional limiations are linked to software and hardware resource. For example, the array of python libraries used is a challenge because there are two possible neural network libraries and they have multiple versions. The challenge lies in that the different libraries used in this thesis are updated at different rates. This means for example that MaWGAN can only run with the library versions it was created with. An additional limitation of GANs is that parameter tuning (see Chapter 3) which not only requires experience but can also be very time consuming. Finally, GANs and most of the models used in this thesis are also extensively resource demanding, for example most training sessions for this thesis required at least 1-2 days to run on a desktop PC.

8.4 Impact

This work was sponsored by DCWW and this collaboration has influenced the direction of this thesis. Chapter 7 utilised DCWW data and demonstrated the value of GANs for synthesising datasets with rare events. DCWW plan to use this to augment the data they have on a range of rare event issues that have significant impact on the business. For example taste and odour issues in reservoirs as in Chapter 7, but also bacteriological events in drinking water supply. DCWW have also expressed an interest in the use of GANs such as MaWGAN to anonymise datasets that they would like to share with others. In addition, GANs can also be used by DCWW for a range of purposes, and Chapter 3,4,5 and 6 showed different ways to synthesise data. The decision tree in Figure 8.1 provides a quick guide to the best options depending on the original data set to be synthesised.

Beyond DCWW, the work of this thesis can also be applied to other industries with similar monitoring data and predictive modelling needs.

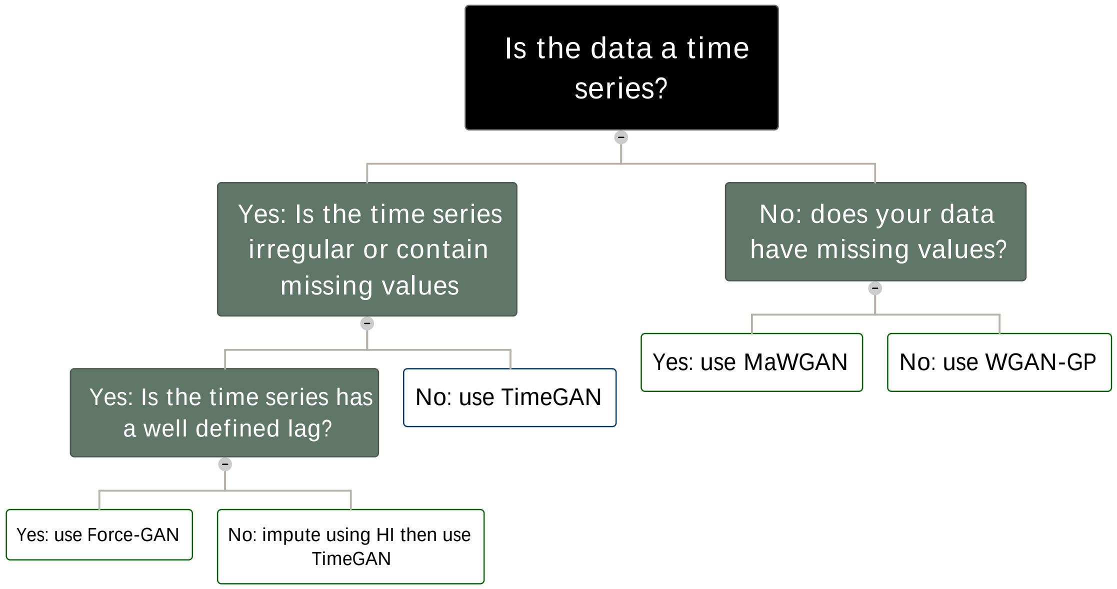

Is the data a time series?
Yes: Is the time series irregular or contain missing values
No: does your data have missing values?
Yes: Is the time series has a well defined lag?
No: use TimeGAN
Yes: use MaWGAN
No: use WGAN-GP
Yes: use Force-GAN
No: impute using HI then use TimeGAN

Figure 8.1: Decision tree

8.5 Future work

There are three areas that could be further explored using this thesis as a basis:

1. In Chapter 4, the performance of MaWGAN is tested with Completely Missing At Random (CMAR) data. It could be further tested with other data types such as Missing At Random (MAR) or Not Missing at Random (NMAR) .

2. The Force-GAN developed in Chapter 5 could potentially be improved to work on more complex datasets. It is possible that a better architecture could be needed, although the option tried unsuccessfully (the subcritic architecture described in section 5.5).

3. The Hankel Imputation method developed in Chapter 6 was time consuming for larger datasets. The generalist convex solver (SCS) used could perhaps be replaced by a more targeted solver. For example, given that the constraint is non-linear, NAG, a solver better designed for second-order cone programs might give better outcomes or require less computing resources.

4. The applied Chapter 7 gave promising results on the use of GAN in the industry

context and could be applied to other types of monitoring data in other industries. One specific potential improvement is the classification of events which in the case studied in Chapter 7 was quite subjective. It is likely that further GANs for time series will be developed in the future that perform better in the synthesising of data (ie synthetic data matches better the original). This would improve the outcomes realised in Chapter 7.

Bibliography

[1] S. W. J. Nijman, A. M. Leeuwenberg, I. Beekers, I. Verkouter, J. J. L. Jacobs, M. L. Bots, F. W. Asselbergs, K. G. M. Moons, and T. P. A. Debray, "Missing data is poorly handled and reported in prediction model studies using machine learning: a literature review," *Journal of Clinical Epidemiology*, vol. 142, pp. 218–229, Feb 2022. [Online]. Available: https://doi.org/10.1016/j.jclinepi.2021.11.023

[2] A. Tsvetanova, M. Sperrin, N. Peek, I. Buchan, S. Hyland, and G. P. Martin, "Missing data was handled inconsistently in uk prediction models: a review of method used," *Journal of Clinical Epidemiology*, vol. 140, pp. 149–158, 2021.

[3] B. Ughy, S. Nagyapati, D. B. Lajko, T. Letoha, A. Prohaszka, D. Deeb, A. Der, A. Pettko-Szandtner, and L. Szilak, "Reconsidering dogmas about the growth of bacterial populations," *Cells*, vol. 12, no. 10, May 2023, pMC10217356. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/37408264

[4] C. C. Lord, "Seasonal population dynamics and behaviour of insects in models of vector-borne pathogens," *Physiological Entomology*, vol. 29, no. 3, pp. 214–222, 2004. [Online]. Available: https://resjournals.onlinelibrary.wiley.com/doi/abs/10.1111/j.0307-6962.2004.00411.x

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, Jun. 2002. [Online]. Available: https://www.jair.org/index.php/jair/article/view/10302

[6] Y. Elor and H. Averbuch-Elor, "To smote, or not to smote?" *CoRR*, vol. abs/2201.08528, 2022. [Online]. Available: https://arxiv.org/abs/2201.08528

[7] R. Blagus and L. Lusa, "Evaluation of smote for high-dimensional class-imbalanced microarray data," in *2012 11th International Conference on Machine Learning and Applications*, vol. 2, 2012, pp. 89–94.

[8] D. B. Rubin, "Inference and missing data," *Biometrika*, vol. 63, no. 3, pp. 581–592, 12 1976.

[9] R. Perkins, E. Slavin, T. Andrade, C. Blenkinsopp, P. Pearson, T. Froggatt, G. Godwin, J. Parslow, S. Hurley, R. Luckwell *et al.*, "Managing taste and odour metabolite production in drinking water reservoirs: The importance of ammonium as a key nutrient trigger," *Journal of environmental management*, vol. 244, pp. 276–284, 2019.

[10] X. Wang and C. Wang, "Time series data cleaning: A survey," *CoRR*, vol. abs/2004.08284, 2020. [Online]. Available: https://arxiv.org/abs/2004.08284

[11] T. C. Mills, "Chapter 2 - transforming time series," in *Applied Time Series Analysis*, T. C. Mills, Ed. Academic Press, 2019, pp. 13–30. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128131176000028

[12] G. King and L. Zeng, "Logistic regression in rare events data," *Political Analysis*, vol. 9, no. 2, p. 137–163, 2001.

[13] L. Wang, M. Han, X. Li, N. Zhang, and H. Cheng, "Review of classification methods on unbalanced data sets," *IEEE Access*, vol. 9, pp. 64 606–64 628, 2021.

[14] Great Britain, *Data Protection Act 1998: Chapter 29.* London: Stationery Office, 1998, oclc: 39736600.

[15] "Data protection in the EU - European Commission," Jul. 2023. [Online]. Available: https://commission.europa.eu/law/law-topic/data-protection/data-protection-eu_en

[16] C. Keerie, C. Tuck, G. Milne, S. Eldridge, N. Wright, and S. C. Lewis, "Data sharing in clinical trials–practical guidance on anonymising trial datasets," *Trials*, vol. 19, no. 1, pp. 1–8, 2018.

[17] L. Xu, C. Jiang, J. Wang, J. Yuan, and Y. Ren, "Information Security in Big Data: Privacy and Data Mining," *IEEE Access*, vol. 2, pp. 1149–1176, 2014, conference Name: IEEE Access.

[18] Y. Lindell and B. Pinkas, "Privacy preserving data mining." *Journal of cryptology*, vol. 15, no. 3, 2002.

[19] C. Eyupoglu, M. A. Aydin, A. H. Zaim, and A. Sertbas, "An efficient big data anonymization algorithm based on chaos and perturbation techniques," *Entropy*, vol. 20, no. 5, p. 373, 2018.

[20] S. A. Assefa, D. Dervovic, M. Mahfouz, R. E. Tillman, P. Reddy, and M. Veloso, "Generating synthetic data in finance: Opportunities, challenges and pitfalls," in *Proceedings of the First ACM International Conference on AI in Finance*, ser. Icaif '20. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3383455.3422554

[21] D. Foster, *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play*. O'Reilly Media, 2019. [Online]. Available: https://books.google.co.uk/books?id=RqegDwAAQBAJ

[22] B. K. Iwana and S. Uchida, "An empirical survey of data augmentation for time series classification with neural networks," *Plos One*, vol. 16, no. 7, pp. 1–32, 07 2021. [Online]. Available: https://doi.org/10.1371/journal.pone.0254841

[23] H. Surendra and H. Mohan, "A review of synthetic data generation methods for privacy preserving data publishing," *International Journal of Scientific & Technology Research*, vol. 6, no. 3, pp. 95–101, 2017.

[24] M. S. Osman, A. M. Abu-Mahfouz, and P. R. Page, "A survey on data imputation techniques: Water distribution system as a use case," *IEEE Access*, vol. 6, pp. 63 279–63 291, 2018.

[25] A. N. Baraldi and C. K. Enders, "An introduction to modern missing data analyses," *Journal of School Psychology*, vol. 48, no. 1, pp. 5–37, 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022440509000661

[26] M. J. Azur, E. A. Stuart, C. Frangakis, and P. J. Leaf, "Multiple imputation by chained equations: what is it and how does it work?" *International journal of methods in psychiatric research*, vol. 20, no. 1, pp. 40–49, 2011.

[27] L. Schumaker, *Spline functions: basic theory*. Cambridge university press, 2007.

[28] R. W. Stineman, "A consistently well-behaved method of interpolation," *Creative Computing*, vol. 6, no. 7, pp. 54–57, 1980.

[29] S. Moritz and T. Bartz-Beielstein, "imputets: time series missing value imputation in r." *R J.*, vol. 9, no. 1, p. 207, 2017.

[30] H. He, Y. Bai, E. A. Garcia, and S. Li, "Adasyn: Adaptive synthetic sampling approach for imbalanced learning," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 1322–1328.

[31] K. Murphy, *Machine Learning: A Probabilistic Perspective*, ser. Adaptive Computation and Machine Learning series. MIT Press, 2012. [Online]. Available: https://books.google.co.uk/books?id=NZP6AQAAQBAJ

[32] H. GM, M. K. Gourisaria, M. Pandey, and S. S. Rautaray, "A comprehensive survey and analysis of generative models in machine learning," *Computer Science Review*, vol. 38, p. 100285, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013720303853

[33] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *CoRR*, vol. abs/1906.02691, 2019. [Online]. Available: http://arxiv.org/abs/1906.02691

[34] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *arXiv:1406.2661 [cs, stat]*, Jun. 2014, arXiv: 1406.2661. [Online]. Available: http://arxiv.org/abs/1406.2661

[35] C. Doersch, "Tutorial on variational autoencoders," 2021. [Online]. Available: https://arxiv.org/abs/1606.05908

[36] F.-A. Croitoru, V. Hondru, R. T. Ionescu, and M. Shah, "Diffusion models in vision: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 9, pp. 10 850–10 869, 2023.

[37] A. Mumuni and F. Mumuni, "Data augmentation: A comprehensive survey of modern approaches," *Array*, vol. 16, p. 100258, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2590005622000911

[38] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017. [Online]. Available: https://books.google.co.uk/books?id=bRpYDgAAQBAJ

[39] W. F. Schmidt, M. A. Kraaijveld, R. P. Duin *et al.*, "Feed forward neural networks with random weights," in *International Conference on Pattern Recognition*. Ieee Computer Society Press, 1992, pp. 1–1.

[40] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[41] Z. Xu, M. Wilber, C. Fang, A. Hertzmann, and H. Jin, "Learning from multi-domain artistic images for arbitrary style transfer," 2019. [Online]. Available: http://arxiv.org/abs/1805.09987

[42] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network," *arXiv:1609.04802 [cs, stat]*, May 2017, arXiv: 1609.04802. [Online]. Available: http://arxiv.org/abs/1609.04802

[43] A. K. Dixit and S. Skeath, *Games of strategy: Fourth international student edition*. WW Norton & Company, 2015.

[44] I. Goodfellow, P. McDaniel, and N. Papernot, "Making machine learning robust against adversarial inputs," *Commun. ACM*, vol. 61, no. 7, p. 56–66, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3134599

[45] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv:1701.07875 [cs, stat]*, Dec. 2017, arXiv: 1701.07875. [Online]. Available: http://arxiv.org/abs/1701.07875

[46] C. Villani, *Optimal transport – Old and new*, 01 2008, vol. 338, pp. xxii+973.

[47] Z. Zhou, Y. Song, L. Yu, H. Wang, W. Zhang, Z. Zhang, and Y. Yu, "Understanding the Effectiveness of Lipschitz-Continuity in Generative Adversarial Nets," Sep. 2018. [Online]. Available: https://openreview.net/forum?id=r1zOg309tX

[48] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/892c3b1c6dccd52936e27cbd0ff683d6-Paper.pdf

[49] O. Mogren, "C-RNN-GAN: continuous recurrent neural networks with adversarial training," *CoRR*, vol. abs/1611.09904, 2016. [Online]. Available: http://arxiv.org/abs/1611.09904

[50] C. Esteban, S. L. Hyland, and G. Rätsch, "Real-valued (medical) time series generation with recurrent conditional GANs." [Online]. Available: http://arxiv.org/abs/1706.02633

[51] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint arXiv:1803.01271*, 2018.

[52] T. Schlegl, P. Seeböck, S. M. Waldstein, U. Schmidt-Erfurth, and G. Langs, "Unsupervised anomaly detection with generative adversarial networks to guide marker discovery," *CoRR*, vol. abs/1703.05921, 2017. [Online]. Available: http://arxiv.org/abs/1703.05921

[53] M. Ehrhart, B. Resch, C. Havas, and D. Niederseer, "A conditional gan for generating time series data for stress detection in wearable physiological sensor data," *Sensors*, vol. 22, no. 16, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/16/5969

[54] J. Yoon and D. Jarrett, "Time-series generative adversarial networks," 2019.

[55] E. Choi, S. Biswal, B. Malin, J. Duke, W. F. Stewart, and J. Sun, "Generating Multi-label Discrete Patient Records using Generative Adversarial Networks," *arXiv:1703.06490 [cs]*, Jan. 2018, arXiv: 1703.06490. [Online]. Available: http://arxiv.org/abs/1703.06490

[56] L. Xie, K. Lin, S. Wang, F. Wang, and J. Zhou, "Differentially Private Generative Adversarial Network," *arXiv:1802.06739 [cs, stat]*, Feb. 2018, arXiv: 1802.06739. [Online]. Available: http://arxiv.org/abs/1802.06739

[57] J. Jordon, J. Yoon, and M. v. d. Schaar, "PATE-GAN: Generating Synthetic Data with Differential Privacy Guarantees," Sep. 2018.

[58] J. Yoon, L. N. Drumright, and M. v. d. Schaar, "Anonymization Through Data Synthesis Using Generative Adversarial Networks (ADS-GAN)," *IEEE Journal of*

Biomedical and Health Informatics, vol. 24, no. 8, pp. 2378–2388, Aug. 2020, conference Name: IEEE Journal of Biomedical and Health Informatics.

[59] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv:1411.1784 [cs, stat]*, Nov. 2014, arXiv: 1411.1784. [Online]. Available: http://arxiv.org/abs/1411.1784

[60] A. Borji, "Pros and cons of gan evaluation measures," *Computer Vision and Image Understanding*, vol. 179, pp. 41–65, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314218304272

[61] D. Dowson and B. Landau, "The fréchet distance between multivariate normal distributions," *Journal of Multivariate Analysis*, vol. 12, no. 3, pp. 450–455, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0047259X8290077X

[62] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," *arXiv:1706.08500 [cs, stat]*, Jan. 2018, arXiv: 1706.08500. [Online]. Available: http://arxiv.org/abs/1706.08500

[63] S. Guan and M. H. Loew, "Measures to evaluate generative adversarial networks based on direct analysis of generated images," *CoRR*, vol. abs/2002.12345, 2020.

[64] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[65] "PyTorch-GAN implementations gan at master · eriklindernoren PyTorch-GAN." [Online]. Available: https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/gan/gan.py

[66] Green9, "Zeleni9/pytorch-wgan," Dec. 2020, original-date: 2018-02-23T16:32:34Z. [Online]. Available: https://github.com/Zeleni9/pytorch-wgan

[67] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[68] D. Heffernan, "An Introduction to Using Categorical Embeddings," Dec. 2018. [Online]. Available: https://medium.com/@davidheffernan_99410/an-introduction-to-using-categorical-embeddings-ee686ed7e7f9

[69] "Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins (Genus Pygoscelis)." [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0090081

[70] J. Yoon, J. Jordon, and M. van der Schaar, "GAIN: Missing Data Imputation using Generative Adversarial Nets," *arXiv:1806.02920 [cs, stat]*, Jun. 2018, arXiv: 1806.02920. [Online]. Available: http://arxiv.org/abs/1806.02920

[71] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. Pmlr, 06–11 Aug 2017, pp. 214–223.

[72] E. Anderson, "The irises of the gaspe peninsula," *Bulletin of the American Iris Society*, vol. 59, pp. 2–5, 1935.

[73] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[74] "Welsh Index of Multiple Deprivation (full Index update with ranks): 2019." [Online]. Available: https://gov.wales/welsh-index-multiple-deprivation-full-index-update-ranks-2019

[75] P. W. Frey and D. J. Slate, "Letter recognition using Holland-style adaptive classifiers," *Machine Learning*, vol. 6, no. 2, pp. 161–182, Mar. 1991.

[76] S. Hitawala, "Comparative Study on Generative Adversarial Networks," *arXiv:1801.04271 [cs]*, Jan. 2018, arXiv: 1801.04271. [Online]. Available: http://arxiv.org/abs/1801.04271

[77] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[78] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014. [Online]. Available: https://arxiv.org/abs/1411.1784

[79] I. Durance and S. J. Ormerod, "Climate change effects on upland stream macroinvertebrates over a 25-year period," *Global change biology*, vol. 13, no. 5, pp. 942–957, 2007.

[80] J. Gillard and K. Usevich, "Structured low-rank matrix completion for forecasting in time series analysis," *International Journal of Forecasting*, vol. 34, no. 4, pp. 582–597, 2018.

[81] S. Al-Homidan, "Hankel matrix transforms and operators," *Journal of Inequalities and Applications*, vol. 2012, no. 1, pp. 1–9, 2012.

[82] H. Butcher and J. Gillard, "Simple nuclear norm based algorithms for imputing missing data and forecasting in time series," *Statistics and its Interface*, vol. 10, no. 1, pp. 19–25, 2017.

[83] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd, "Conic optimization via operator splitting and homogeneous self-dual embedding," *Journal of Optimization Theory and Applications*, vol. 169, no. 3, pp. 1042–1068, June 2016. [Online]. Available: http://stanford.edu/~boyd/papers/scs.html

[84] H. Wilde, W. B. Perry, O. Jones, P. Kille, A. Weightman, D. L. Jones, G. Cross, and I. Durance, "Accounting for dilution of sars-cov-2 in wastewater samples using physico-chemical markers," *Water*, vol. 14, no. 18, 2022. [Online]. Available: https://www.mdpi.com/2073-4441/14/18/2885

[85] F. Petropoulos, D. Apiletti, V. Assimakopoulos, M. Z. Babai, D. K. Barrow, S. Ben Taieb, C. Bergmeir, R. J. Bessa, J. Bijak, J. E. Boylan, J. Browell, C. Carnevale, J. L. Castle, P. Cirillo, M. P. Clements, C. Cordeiro, F. L. Cyrino Oliveira, S. De Baets, A. Dokumentov, J. Ellison, P. Fiszeder, P. H. Franses, D. T. Frazier, M. Gilliland, M. S. Gönül, P. Goodwin, L. Grossi, Y. Grushka-Cockayne, M. Guidolin, M. Guidolin, U. Gunter, X. Guo, R. Guseo, N. Harvey, D. F. Hendry, R. Hollyman, T. Januschowski, J. Jeon, V. R. R. Jose, Y. Kang, A. B. Koehler, S. Kolassa, N. Kourentzes, S. Leva, F. Li, K. Litsiou, S. Makridakis, G. M. Martin, A. B. Martinez, S. Meeran, T. Modis, K. Nikolopoulos, D. Önkal, A. Paccagnini, A. Panagiotelis, I. Panapakidis, J. M. Pavía, M. Pedio, D. J. Pedregal, P. Pinson, P. Ramos, D. E. Rapach, J. J. Reade, B. Rostami-Tabar, M. Rubaszek, G. Sermpinis, H. L. Shang, E. Spiliotis, A. A. Syntetos, P. D. Talagala, T. S. Talagala, L. Tashman, D. Thomakos, T. Thorarinsdottir, E. Todini, J. R. Trapero Arenas, X. Wang, R. L. Winkler, A. Yusupova, and F. Ziel, "Forecasting: theory and practice," *International Journal of Forecasting*, vol. 38, no. 3, pp. 705–871, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207021001758

[86] E. I. Prest, F. Hammes, M. C. Van Loosdrecht, and J. S. Vrouwenvelder, "Biological stability of drinking water: controlling factors, methods, and challenges," *Frontiers in microbiology*, vol. 7, p. 45, 2016.

[87] M. J. Kehoe, K. P. Chun, and H. M. Baulch, "Who smells? forecasting taste and odor in a drinking water reservoir," *Environmental Science & Technology*, vol. 49, no. 18, pp. 10 984–10 992, 2015, pmid: 26266956. [Online]. Available: https://doi.org/10.1021/acs.est.5b00979

[88] A. S. o. Martins, E. Carvajal, J. A. A. d. Santos, P. G. Moura, N. B. Handam, N. P. Kotowski Filho, R. Jardim, A. d. S. Ferrão Filho *et al.*, "Events linked to geosmin and 2-methylisoborneol (2-mib) in a water supply in the state of rio de janeiro, brazil: a case study," 2021.

[89] M. A. Webber, P. Atherton, and G. Newcombe, “Taste and odour and public perceptions: what do our customers really think about their drinking water?” *Journal of Water Supply: Research and Technology—AQUA*, vol. 64, no. 7, pp. 802–811, 2015.

[90] D. Cook, C. Chow, and M. Drikas, “Laboratory study of conventional alum treatment versus miex® treatment for removal of natural organic matter,” in *19th Federal AWA Convention*, 2001, pp. 1–4.

[91] R. Srinivasan and G. A. Sorial, “Treatment of taste and odor causing compounds 2-methyl isoborneol and geosmin in drinking water: A critical review,” *Journal of Environmental Sciences*, vol. 23, no. 1, pp. 1–13, 2011.

[92] W. Yang, M. Paetkau, and N. Cicek, “Improving the performance of membrane bioreactors by powdered activated carbon dosing with cost considerations,” *Water Science and Technology*, vol. 62, no. 1, pp. 172–179, Jul. 2010. [Online]. Available: https://iwaponline.com/wst/article/62/1/172/14640/Improving-the-performance-of-membrane-bioreactors

[93] S. Suurnäkki, G. V. Gomez-Saez, A. Rantala-Ylinen, J. Jokela, D. P. Fewer, and K. Sivonen, “Identification of geosmin and 2-methylisoborneol in cyanobacteria and molecular detection methods for the producers of these compounds,” *Water Research*, vol. 68, pp. 56–66, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0043135414006678

[94] T. D. Harris, V. H. Smith, J. L. Graham, D. B. V. de Waal, L. P. Tedesco, and N. Clercin, “Combined effects of nitrogen to phosphorus and nitrate to ammonia ratios on cyanobacterial metabolite concentrations in eutrophic midwestern usa reservoirs,” *Inland Waters*, vol. 6, no. 2, pp. 199–210, 2016. [Online]. Available: https://www.tandfonline.com/doi/abs/10.5268/IW-6.2.938

[95] L. Molot, S. Watson, I. Creed, C. Trick, S. McCabe, M. Verschoor, R. Sorichetti, C. Powe, J. Venkiteswaran, and S. Schiff, “A novel model for cyanobacteria bloom

formation: the critical role of anoxia and ferrous iron," *Freshwater Biology*, vol. 59, no. 6, pp. 1323–1340, 2014.

[96] J. Zhang, L. Li, L. Qiu, X. Wang, X. Meng, Y. You, J. Yu, and W. Ma, "Effects of climate change on 2-methylisoborneol production in two cyanobacterial species," *Water*, vol. 9, no. 11, 2017. [Online]. Available: https://www.mdpi.com/2073-4441/9/11/859

[97] R. D. Harmel, H. E. Preisendanz, K. W. King, D. Busch, F. Birgand, and D. Sahoo, "A review of data quality and cost considerations for water quality monitoring at the field scale and in small watersheds," *Water*, vol. 15, no. 17, 2023. [Online]. Available: https://www.mdpi.com/2073-4441/15/17/3110

[98] A. Hooper, P. Kille, S. Watson, S. Christofides, and R. Perkins, "The importance of nutrient ratios in determining elevations in geosmin synthase (geoa) and 2-mib cyclase (mic) resulting in taste and odour events," *Water Research*, vol. 232, p. 119693, 2023.

[99] M. Simpson and B. MacLeod, "Comparison of various powdered activated carbons for the removal of geosmin and 2-methylisoborneol in selected water conditions," in *Proc. Am. Water Works Assoc. Ann. Conf*, 1991.

[100] environment.ec.europa.eu – drinking water. [Online]. Available: https://environment.ec.europa.eu/topics/water/drinking-water_en

[101] P. Ömür-Özbek, J. Little, and A. Dietrich, "Ability of humans to smell geosmin, 2-mib and nonadienal in indoor air when using contaminated drinking water," *Water science and technology*, vol. 55, no. 5, pp. 249–256, 2007.

[102] W. Young, H. Horth, R. Crane, T. Ogden, and M. Arnott, "Taste and odour threshold concentrations of potential potable water contaminants," *Water Research*, vol. 30, no. 2, pp. 331–340, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0043135495001735

[103] R. Paerl, H. Huang, and L. Ehrlich, “Investigating the microbial culprits of taste/odor issues in city of durham drinking water reservoir lake michie and algicidal mitigation tactics,” Nc Wrri, Tech. Rep., 2022.

[104] N. E. Huang, I. Daubechies, and T. Y. Hou, “Adaptive data analysis: theory and applications,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150207, 2016. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2015.0207

Appendix

A code

A.1 Interface

```
import math
import numpy as np
import Mawgan.tools as tools
import Mawgan.gans as gans
import click
import logging

#--------------------------------------------------------------------------------
#_#
#__#1. Inputting parameters
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function reads parameters and prepares the GAN
#_#Reviewer Notes\

#_# Steps\
#_# Document all the parameters needed for the GAN
@click.command()
@click.option("--test",default=None,help="test the intallment")
@click.option(
    "--filepath",
    help=" enter the file name and location of the database and model",
)
@click.option(
    "--epochs", help="choose how long that you want to train"
)
@click.option(
    "--dataset",
    default=None,
    help="choose the dataset/table that the GAN will train on",
)
@click.option("--opti", default=None, help="choose the optimiser you want to use")
@click.option("--nodes", default=None, help="choose the number nodes per layer")
@click.option(
    "--batch", default=None, help="choose how many datapoints is process when traing in one go"
)
@click.option(
    "--layers", default=None, help="choose the number of layers of each network"
)
@click.option("--lambdas", type=float, default=None, help="learning penalty")
```

```
@click.option(
    "--sample",
    default=1,
    type=int,
    help="choose the number of generated data",
)
@click.option(
    "--rate",
    default=None,
    type=float,
    help="choose the learing rate of the model",
)

@click.option(
    "--usegpu",
    default=0,
    type=int,
    help="set to 1 to use gpu",
)

def main(
    dataset,
    filepath,
    epochs,
    opti,
    nodes,
    batch,
    layers,
    lambdas,
    sample,
    rate,
    usegpu,
    test
):
    """
    This is the code for "MaWGAN: a Generative Adversarial Network to create synthetic
    data from datasets with missing data", this script is the interface of the MaGAN
    It pre-processes data, trains gan and creates synthetic datasets.
    """
    #_# check if testing for installment
    if test != None:
    #_#if installation is sucessfull then you should be able run to this point without error
        print("installation is sucessfully")
        #_# exit the script
        return
    #_# Tell user that the computer is processing the request
    click.echo("loading...")
    #_# Extract the extension of the file path
    filename, extention = filepath.split(".")
    if extention == "csv":
        dataset = True
    #_# Create the log file
    tools.setup_log(filename + "_progress.log")
    #_# Save the parameters to the parameters list
    parameters_list = [dataset, opti, nodes, batch, layers, lambdas, rate]
    #_# Load the previous parameters and check if any is missing
    parameters, successfully_loaded = parameters_handeling(filename, parameters_list)
    #_# Convert the epochs into an integer variable
    epochs = int(epochs)
    #_# Attempt to load the dataset and pre-process the dataset
    try:
        database, details, encoded = load_data(parameters[0], filepath, extention)
    except Exception as e:
        #_# State if a file does not exist
        print("Data could not be loaded propely see logs for more info")
        #_# Record the error
        logging.exception(e)
        #_# Exit the function
        return
    #_# Create the GAN using the parameters and train it
    thegan, success = run(filename, epochs, parameters, successfully_loaded, database, bool(usegpu))
    #_# Create an empty variable so it does not produce errors
    fake = None
    #_# Check if the GAN has trained sucessfully
    if success and sample > 0:
        #_# If it has trained sucessfully make a synthetic sample data
        fake = make_samples(
            thegan,
            encoded,
            sample,
            filename,
            details,
            extention,
            bool(usegpu),
        )
    #_# Output the GAN object and the synthetic data
    return thegan, fake
#_#

#--------------------------------------------------------------------------------
#_#
#__#2. Unpack parameters
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function unpacks the parameters into individual variables
#_#Reviewer Notes\

#_# Steps\
#_# Outputs the parameters into individual variables

def unpack(p):
    """
    Unpacks the parameters
    """
    return p[1], int(p[2]), int(p[3]), int(p[4])

#--------------------------------------------------------------------------------
#_#
#__#3. Set-up
#_#
```

```
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function checks missing parameters and prompts to fill gaps

#_#Reviewer Notes\

#_# Steps\
#_#

def setup(parameters_list,sucess,loaded_parameters):
    """
    Creates new parameters
    """
    #_# Create an empty list for checked parameters
    parameters = []
    #_# Setup the questions
    questions = [
        "table? ",
        "opti? ",
        "nodes size? ",
        "batch size? ",
        "layers? ",
        "learning constiction? ",
        "rate? ",
    ]
    #_#Loop over each parameter
    for q in range(len(questions)):
    #_# Check if the parameters exist
        if parameters_list[q] != None:
        #_# If the parameter exists put it in the waiting list
        #_# to be added to the list of checked parameters
            param = parameters_list[q]
        elif sucess:
        #_#  If the parameters was loaded it added to the list of checked parameters
            param = loaded_parameters[q]
        else:
            # If it does not exist, prompt to fill it in, parameters 1 and 2 are strings
            if q < 3:
                param = input(questions[q])
            #_# parameters 3, 4 and 5 are intergers
            elif q < 6:
                param = input_int(questions[q])
            #_# parameters 6 and 7 are floats
            else:
                param = input_float(questions[q])
        #_# Save the answers and transfer the checked parameters into the checked parameters list
        parameters.append(param)
    #_# Save the parameters list
    return parameters


#-------------------------------------------------------------------------------
#_#
#__#3. Checking the responses to questions
#_#
```

```
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function checks if the response to the question is an integer

#_#Reviewer Notes\

#_# Steps\
#_#

def input_int(question):
    """
    makes sure a number is inputed
    """
    #_# Loop until the response to the question is an integer
    while True:
        try:
            #_# Ask the question
            a = input(question)
            #_# Attempt to convert the answer to an integer
            answer = int(a)
            #_# If failed, state that the answer must be an integer
        except Exception:
            print("the answer must be an integer")
            #_# Can exit programme by entering nothing
            if a == "" or None:
                raise RuntimeError
        #_# If the conversion does not fail return the converted answer
        else:
            return answer

#--------------------------------------------------------------------------------
#_#
#__#4. Checking the responses to questions
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function checks if the response to the question is a float

#_#Reviewer Notes\

#_# Steps\
#_#

def input_float(question):
    """
    makes sure a number is inputed
    """
    #_# Loop until the response to the question is an float
    while True:
        try:
            #_# Ask the question
            a = input(question)
            #_# Attempt to convert the answer to an float
            answer = float(a)
            #_# If failed, state that the answer must be an float
        except Exception:
            print("the answer must be an float")
            #_# Can exit programme by entering nothing
            if a == "" or None:
                raise RuntimeError
        #_# If the conversion does not fail return the converted answer
        else:
            return answer
#--------------------------------------------------------------------------------
#_#
#__#5. Loading the data
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function loads the data and pre-processes it

#_#Reviewer Notes\

#_# Steps\
#_#
#_#
def load_data(sets, filepath, extention):
    """
    Loads a dataset from an sql table
    """
#_# Check if the file format is a db file
    if extention == "db":
        #_# Load the file
        raw_data = tools.load_sql(filepath, sets)
        #_# Check if the file format is csv
    if extention == "csv":
        #_# Load the file
        raw_data = tools.pd.read_csv(filepath)
        #_# Pre-process the data
    database, details, encoded = tools.procsses_data(raw_data)
    #_# Output the database and the mapping
    return database, details, encoded

#--------------------------------------------------------------------------------
#_#
#__#6. Loading the GAN weights
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function loads the weights o the GAN

#_#Reviewer Notes\

#_# Steps\
#_#
def load_gan_weight(filepath, mygan):
    """
    Loads weight from previous trained GAN
    """
#_# Attempt to load the weights of the GAN
    try:
        mygan.load_model(filepath)
        #_# If this fails, state so
    except OSError:  # as 'Unable to open file':
        print("No previous gan exsit, starting from scratch")
    finally:
        #_# Ouput the GAN class
        return mygan
#-------------------------------------------------------------------------------
#_#
#__# 7. Creating the GAN class
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function creates the GAN class according to the given parameters

#_#Reviewer Notes\

#_# Steps\
#_#

def create_model(parameters, no_field):
    """
    Builds the GAN using the parameters
    """
#_# Take the learning rate (lr) from the parameter list
    lr = float(parameters[6])
    #_# Umpack the rest of the parameters
    opti, nodes_dim, batch, number_of_layers = unpack(parameters)
    #_# Create the GAN class modelusing the list of parameters
    mygan = gans.MaWGAN(
        optimiser=opti,
        number_of_variables=no_field,
        number_of_nodes=nodes_dim,
        number_of_layers=number_of_layers,
        lambdas=float(parameters[5]),
        learning_rate=lr,
    )
    #_# Print the configuration
    mygan.summary()
    #_# Output the GAN class
    return mygan

#-------------------------------------------------------------------------------
#_#
#__#8. Making samples
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function creates synthetic data and saves them to a file
#_#Reviewer Notes\
```

```python
#_# Steps\
#_#
def make_samples(
    mygan, database, batch, filepath, details, extention, usegpux
):
    """
    Creates a number of samples
    """
    #_# Unpack the detail list
    mean, std, info, col = details
    #_# Create an empty variable called fullset
    fullset = None
    #_# Create a batch of synthetic data
    generated_data = mygan.create_synthetic(batch)
    # sample the dataset
    data_sample = database.sample(batch)
    #_# Un-normalise the batch of synthetic data
    generated_data = tools.unnormalize(tools.pd.DataFrame(generated_data), mean, std)
    #_# Relabel the variables as the previous format could not label them
    generated_data.columns = col
    try:
        #_# calculate ls score
        if usegpux:
            ls = tools.gpu_LS(data_sample.dropna().to_numpy(),generated_data.to_numpy())
        else:
            ls = tools.LS(data_sample.dropna().to_numpy(),generated_data.to_numpy())
        #_# calculate fid score
        fid = tools.calculate_fid(data_sample.dropna(),generated_data)
        #_#print comparison
        print(" LS: ",ls,"\n FID:",fid)
    except Exception as e:
       print("comparison calculation failed")
       logging.error("comparison calculation failed due to" + str(e))
    #_# Restore the categorical variables
    fullset = tools.decoding(generated_data, info)
    #_# Save the synthetic data in the same file format as the original,
    #_# if the file format is db then save in db, if the file format is csv then save in csv
    if extention == "db":
        tools.save_sql(fullset, filepath + ".db")
    if extention == "csv":
        fullset.to_csv(filepath + "_synthetic.csv", index=False)
    #_# Output the synthetic data
    return fullset

#--------------------------------------------------------------------------------
#_#
#__#9. Saving the parameters
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function saves the parameters so that the GAN can be rebuilt the same way

#_#Reviewer Notes\

#_# Steps\
#_#
def save_parameters(parameters, filepath):
    """
    Saves the parameters for the GAN
    """
#_# Create a filename for the parameters to be saved into
    fname = filepath + "_parameters.npy"
    #_# Convert the list to an array
    parameter_array = np.array(parameters)
    #_# Save the array
    np.save(fname, parameter_array)

#-------------------------------------------------------------------------------
#_#
#__#10. Loading the parameters
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function loads previously saved parameters
#_#Reviewer Notes\

#_# Steps\
#_#
def load_parameters(filepath):
    """
    Loads the parameters for the GAN
    """
#_# Attempt to load the parameters
    try:
        parameter_array = np.load(filepath + "_parameters.npy", allow_pickle=True)
        #_# Otherwise print an error message
    except OSError:
        print("parameters file not found, starting from scratch")
        #_# Create a flag to state that no parameters have been loaded
        successfully_loaded = False
        parameter_array = None
    else:
        #_# Create a flag that states that the parameters have been loaded
        successfully_loaded = True
    return parameter_array, successfully_loaded

#-------------------------------------------------------------------------------
#_#
#__#11. Handeling the parameters
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function loads parameters if the file exists otherwise it
#_# takes parameters that the user inputs

#_#Reviewer Notes\

#_# Steps\
#_#
```

```
def parameters_handeling(filepath, parameters_list):
    """
    Load parameters if they exist, otherwise saves new ones
    """
#_# Load parameters from the existing file
    loaded_parameters, successfully_loaded = load_parameters(filepath)
    #_# check if nothing is missing in the parameter list
    parameters = setup(parameters_list,successfully_loaded,loaded_parameters)
    #_# Save the new parameters
    save_parameters(parameters, filepath)
    #_# Print the parameters
    print(parameters)
    #_# Output that the parameters and wether they have been sucessfully loaded
    return parameters, successfully_loaded

#--------------------------------------------------------------------------------
#_#
#__#12. Running the main GAN script
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_# This function runs the main GAN script: It creates and trains a GAN
#_# from the parameters provided.
#_# It will load the weights of the GAN if they exist.

#_#Reviewer Notes\

#_# Steps\
#_#
def run(filepath, epochs, parameters, successfully_loaded, database, usegpu):
    """
    Creates and trains a GAN from the parameters provided.
    It will load the weights of the GAN if they exist.
    """
    #_# Count the number of variables
    no_field = len(database[1])
    try:
        #_# Attempt to create the model
        mygan = create_model(parameters, no_field)
        #_# Otherwise state that the building has failed, exit programme
    except Exception as e:
        print("building failed, check you parameters")
        logging.error("building failed due to" + str(e))
        return None, False
        #_# Do not load weights if parameters not loaded
    if successfully_loaded:
        #_# Load the GAN weights to the network
        mygan = load_gan_weight(filepath, mygan)
        #_# If the epoch is null then do not attempt to train
    if epochs > 0:
        #_#  Calculate a tenth of the total epochs that will be how often the loss will be recorded
        step = int(math.ceil(epochs * 0.1))
        #_# Prepare the checking of missing data
        checkI = tools.pd.DataFrame(database)
        #_#  Check if any data is missing and if so run MAWGAN instead of WGAN-GP
        checkII = checkI.isnull().sum().sum() > 0
        #_# Attempt to train the GAN
        try:
            mygan.train(database, int(parameters[3]), epochs, checkII, step, usegpu=usegpu)
            #_# State that the training has failed to check parameters if the training failed
        except Exception as e:
            logging.exception(e)
            print("training failed check you parameters")
            #_# and exit the programme
            return None, False
        else:
            #_# If training is sucessfull, save the model
            mygan.save_model(filepath)
    #_# Return the GAN class and whether or not the programme was sucessfull
    return mygan, True


if __name__ == "__main__":
    main()
```

A.2 full model

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
from torch import autograd
from .masker import make_mask
import logging


class MaWGAN(object):
#---------------------------------------------------------------------------------
#_#
#__#1.Creates the GAN according to the parameters given
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function creates the class object
#_#
#_#Reviewer Notes\
#_#
#_#

    def __init__(
        self,
        optimiser,
        number_of_variables,
        number_of_nodes,
        number_of_layers,
        lambdas,
        learning_rate
    ):
    #_#Steps\
    #_# Save the number of nodes in a layer to the class object
        self.net_dim = number_of_nodes
    #_# Save the number of variables to the class object
        self.data_dim = number_of_variables
    #_# Create the generator and save it in the class object
        self.Make_Generator(number_of_layers)
    #_# Create the critic and save it in the class object
        self.Make_Critic(number_of_layers)
    #_# Save the lambda_term to the class object
        self.lambda_term = lambdas
    #_# Save the learning rate to the class object
        self.learning_rate = learning_rate
    #_# Create the optimzers used for training the networks
        self.make_optimize(optimiser.lower())

#---------------------------------------------------------------------------------
#_#
#__#2. Create the optimzers used for training the networks
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function creates the optimers for the critic and the generator
#_#that are used for training the networks. There is a choice of five optimsers
#_#(adam, adadelta, adagrad, rmsprop).
#_#Reviewer Notes\
#_#
#_#

    def make_optimize(self,opt):
        #_#Steps\
        #_# Check if the optimser 'adam' has been selected
        if opt == "adam":
        #_# Assign the optimser 'adam' to the Critic parameters
            self.d_optimizer = optim.Adam(self.Critic.parameters(), lr=self.learning_rate)
        #_# Assign the optimser 'adam' to the Generator parameters
            self.g_optimizer = optim.Adam(self.Generator.parameters(), lr=self.learning_rate)
        #_# Check if the optimser 'adadelta' has been selected
        if opt == "adadelta":
        #_# Assign the optimser 'adelta' to the Critic parameters
            self.d_optimizer = optim.Adadelta(self.Critic.parameters(), lr=self.learning_rate)
        #_# Assign the optimser 'adam' to the Generator parameters
            self.g_optimizer = optim.Adadelta(self.Generator.parameters(), lr=self.learning_rate)
        #_# Check if the optimser 'adagrad' has been selected
        if opt == "adagrad":
        #_# Assign the optimser 'adagrad' to the Critic parameters
            self.d_optimizer = optim.Adagrad(self.Critic.parameters(), lr=self.learning_rate)
                #_# Assign the optimser 'adagrad' to the Generator parameters
            self.g_optimizer = optim.Adagrad(self.Generator.parameters(), lr=self.learning_rate)
        #_# Check if the optimser 'rmsprop' has been selected
        if opt == "rmsprop":
        #_# Assign the optimser 'rmsprop' to the Critic parameters
            self.d_optimizer = optim.RMSprop(self.Critic.parameters(), lr=self.learning_rate)
        #_# Assign the optimser 'rmsprop' to the Generator parameters
            self.g_optimizer = optim.RMSprop(self.Generator.parameters(), lr=self.learning_rate)
        #_# Check if the optimser 'SGD' has been selected
        if opt == "sgd":
        #_# Assign the optimser 'sgd' to the Critic parameters
            self.d_optimizer = optim.SGD(self.Critic.parameters(), lr=self.learning_rate)
        #_# Assign the optimser 'sgd' to the Generator parameters
            self.g_optimizer = optim.SGD(self.Generator.parameters(), lr=self.learning_rate)

#-------------------------------------------------------------------------------
#_#
#__#3. Build generator
#_#
#__#Review Decision:
#_#
#_#This makes a Generator network with a given number of layers (number_of_layers) and
#_# a given number of nodes per layer (net_dim).
#_#The input to the generator is a random matrix where the dimensions are the batch size and
#_#the number of variables (data_dim).
#_#The outputs are synthetic data with the same size as the input matrix.
#_#
#_#
    def Make_Generator(self, number_of_layers):
        """
        This makes a generator network with 'number_of_layers' layers and 'net_dim' of nodes per layer.
        It takes in a vector of 'batch_size' length and outputs a vector of data that is 'data_dim' long.
        """
        #_#Steps/
        #_#Create empty neural network object
        self.Generator = nn.Sequential()
        #_# Add the input layers
        self.Generator.add_module(
            str(number_of_layers) + "Glayer", nn.Linear(self.data_dim, self.net_dim)
        )
        #_# Add an activation function
        self.Generator.add_module(str(number_of_layers) + "active", nn.Tanh())
        #_# Reduce the number of layer count by 1
        number_of_layers -= 1
        #_# Loops until number layer count = 0
        while number_of_layers > 1:
        #_# Create a hidden layer
            self.Generator.add_module(
                str(number_of_layers) + "Glayer",
                nn.Linear(self.net_dim, self.net_dim),
            )
            #_# Add an activation function
            self.Generator.add_module(str(number_of_layers) + "active", nn.Tanh())
            #_# Reduce the number of layer count by 1
            number_of_layers -= 1
        #_# Create the ouput layer
        self.Generator.add_module(
            str(number_of_layers) + "Glayer", nn.Linear(self.net_dim, self.data_dim)
        )
#-------------------------------------------------------------------------------
#_#
#__#4. Build the Critic
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This makes a Critic network with a given number of layers (number_of_layers) and
#_# a given number of nodes per layer (net_dim).
#_#It takes in a dataset that has a given number of variables (data_dim)
#_#and outputs a vector that deduces whether or not each
#_#observation is  real or synthetic.
#_#Reviewer Notes\
#_#
#_#
    def Make_Critic(self, number_of_layers):
        """
        This makes a critic network with 'number_of_layers' layers and 'net_dim' of nodes per layer.
        It takes in a vector of data that is 'data_dim' long and
        outputs a probability of the data being real or synthetic.
        """
        #_#Steps/
        #_#Create empty neural network object
        self.Critic = nn.Sequential()
        #_# add the input layers
        self.Critic.add_module(
            str(number_of_layers) + "Clayer", nn.Linear(self.data_dim, self.net_dim)
        )
        #_# Add an activation function
        self.Critic.add_module(str(number_of_layers) + "active", nn.Tanh())
        #_# Reduce the number of layer count by 1
        number_of_layers -= 1
        #_# Loop until number layer count = 0
        while number_of_layers > 1:
        #_# Create a hidden layer
            self.Critic.add_module(
                str(number_of_layers) + "Clayer",
                nn.Linear(self.net_dim, self.net_dim),
            )
            #_# Add an activation function
            self.Critic.add_module(str(number_of_layers) + "active", nn.Tanh())
            number_of_layers -= 1
        #_# Create the ouput layer
        self.Critic.add_module(
            str(number_of_layers) + "Clayer", nn.Linear(self.net_dim, 1)
        )
#-----------------------------------------------------------------------------------
#_#
#__#5. Create synthetic data
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function creates a batch of synthetic data to use outside training
#_#Reviewer Notes\
#_#
#_#

    def create_synthetic(self, batch_size):
        """
        This creates a batch of synthetic data
        """
        #_#Steps\
        #_#Create a random matrix with dimensions batch_size * data_dim
        z = torch.randn(batch_size, self.data_dim)
        #_# Feed the random matrix into the Generator
        synthetic_data = self.Generator(z)
        #_# Output the synthetic dataset without the gradient metadata
        return synthetic_data.detach().numpy()
#-----------------------------------------------------------------------------------
#_#
#__#6. Training the GAN
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function trains the GAN by alternating between training the Critic
#_#a number of times (critic_round) and training the Generator once in each epoch on
#_#a given dataset (data). If the given dataset has a 'hasmissing flag' set to TRUE,
#_#the MAWGAN code is run, otherwise the WGAN-GP code is run.
#_#This function will record the Loss of the Generator and Critic on a given schedule
#_#Reviewer Notes\
#_#
#_#
```

```
def train(
    self,
    data,
    batch_size,
    epochs,
    hasmissing=False,
    record_every_n_batches=10,
    n_critic=5,
    usegpu=False
):
    """
    This trains the GAN by alternating between training the critic 'critic_round' times
    and training the generator once in each epoch
    """
    #_#Steps\
    #_#Save usegpu flag to the class object
    self.usegpu = usegpu
    #_#If the usegpu flag is true then move the gan to the gpu
    if self.usegpu:
        self.Critic = self.Critic.cuda()
        self.Generator = self.Generator.cuda()
    #_#Save the batch size to the class object
    self.batch_size = batch_size
    #_# Convert data format so it can be processed (from numpy.array to torch.tensor)
    data_tensor = torch.Tensor(data)
    #_# Create a matrix of ones
    one = torch.tensor(1, dtype=torch.float)
    #_# Create a matrix of minus ones
    mone = one * -1
    #_# mask the data
    mask, binary_mask = make_mask(data_tensor)
    #_# Apply the mask
    data_tensor[binary_mask] = 0
    #_# tranfer the original data to gpu if needed
    if self.usegpu:
        data_tensor = data_tensor.cuda()
        #_# tranfer the mask to gpu if needed
        mask =  mask.cuda()
    #_#Main traing loop
    for g_iter in range(epochs):
        #_# Allow the critic to be trained
        for p in self.Critic.parameters():
            p.requires_grad = True
        for d_iter in range(n_critic):
            #_# reset Critic calculate_gradient_penaltyent
            self.Critic.zero_grad()
            #_# sample dataset
            sample_data, sample_mask = self.pick_sample(data_tensor,mask)
            org_data = Variable(sample_data)
            #_#create a random matrix with dimensions batch_size * data_dim
            z = Variable(torch.randn(self.batch_size, self.data_dim))
            #_# feed the random matrix into the generator, tranfer to gpu if needed
            if self.usegpu:
                synthetic_data = self.Generator(z.cuda())
            else:
                synthetic_data = self.Generator(z)
            #_# Applies the mask to the generated data
            synthetic_data = synthetic_data * sample_mask
            #_# feed the original data to the Critic
            d_loss_real = self.Critic(org_data)
            #_# calculate the mean of the outputs of the Critic
            d_loss_real = d_loss_real.mean()
            #_# calculate negative gradient
            d_loss_real.backward(mone)
            #_# feed the synthetic data to the Critic
            d_loss_synthetic = self.Critic(synthetic_data)
            #_# calculate the mean of the outputs of the Critic
            d_loss_synthetic = d_loss_synthetic.mean()
            #_# calculate positive gradient
            d_loss_synthetic.backward(one)
            #_#calculate the gradient penalty
            gradient_penalty = self.calculate_gradient_penalty(
                org_data.data, synthetic_data.data
            )
            #_# calculate the gradient of the gradient penalty
            gradient_penalty.backward()
            #_# add all the loss
            d_loss = d_loss_synthetic - d_loss_real + gradient_penalty
            #_#adjust the weight of the Critic
            self.d_optimizer.step()
        #_# Forbid the critic to be trained
        for p in self.Critic.parameters():
            p.requires_grad = False  # to avoid computation
        self.Generator.zero_grad()
        #_#create a random matrix with dimensions batch_size * data_dim
        z = Variable(torch.randn(self.batch_size, self.data_dim))
        #_# feed the random matrix into the generator, tranfer to gpu if needed
        if self.usegpu:
            synthetic_data = self.Generator(z.cuda())
        else:
            synthetic_data = self.Generator(z)
        #_# feed the synthetic data to the Critic
        g_loss = self.Critic(synthetic_data)
        #_# calculate the mean of the outputs of the Critic
        g_loss = g_loss.mean()
        #_# calculate negative gradient
        g_loss.backward(mone)
        #_#adjust the weight of the generator
        self.g_optimizer.step()
        #_# send progress to the log file
        if g_iter % record_every_n_batches == 0:
            logging.info(
                f"iteration: {g_iter}/{epochs},
                  g_loss: {g_loss:.2f},
                  loss_synthetic: {d_loss_synthetic:.2f},
                  loss_real: {d_loss_real:.2f}"
            )
    if usegpu:
        #_# move the networks back to the cpu unless they were alredy there
        self.Critic = self.Critic.cpu()
        self.Generator = self.Generator.cpu()
```

```
#------------------------------------------------------------------------------------
#_#
#__#7. Calculate the gradient penalty
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#Computes a gradient penalty based on the randomly weighted mean of the real and synthetic sample
#_#Reviewer Notes\
#_#
    def calculate_gradient_penalty(self, real_data, synthetic_data):
        """
        This function computes a gradient penalty based on the
        randomly weighted mean of the real and synthetic sample
        """
        #_#Steps\
        #_#Create eta: a random value beteween 0 and 1
        eta = torch.FloatTensor(self.batch_size, 1).uniform_(0, 1)
        #_# Move eta to gpu if the gpu flag is True
        if self.usegpu:
            eta = eta.cuda()
        #_# Match the eta shape to the original and synthetic shape
        eta = eta.expand(self.batch_size, real_data.size(1))
        #_# Calculate eta*real_data + (1-eta)*synthetic_data term
        interpolated = eta * real_data + ((1 - eta) * synthetic_data)
        #_# Convert the interpolated variable to a format required by the library
        interpolated = Variable(interpolated, requires_grad=True)
        #_# Feed the interpolated data to the Critic
        prob_interpolated = self.Critic(interpolated)
        #_# Calculate gradients of the above output
        if self.usegpu:
            gradients = autograd.grad(
                outputs=prob_interpolated,
                inputs=interpolated,
                grad_outputs=torch.ones(prob_interpolated.size()).cuda(),
                create_graph=True,
                retain_graph=True,
            )[0]
        else:
            gradients = autograd.grad(
                outputs=prob_interpolated,
                inputs=interpolated,
                grad_outputs=torch.ones(prob_interpolated.size()),
                create_graph=True,
                retain_graph=True,
            )[0]
        #_# Calculate gradient penalty
        grad_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * self.lambda_term
        #_# outputs gradient penalty
        return grad_penalty

#------------------------------------------------------------------------------------
#_#
#__#8. Summary
#_#
#__#Review Decision:
```

```
#_#
#_#Author Notes\
#_#Prints the composition of the GAN
#_#Reviewer Notes\
#_#
    def summary(self):

        """
        prints the composition of the GAN
        """
        #_# Steps\
        #_# Prints the composition of the Critic
        print(self.Critic)
        #_# Prints the composition of the Generator
        print(self.Generator)
#-----------------------------------------------------------------------------
#_#
#__#9. Pick a sample
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function picks a sample of the data the size of a batch
#_#Reviewer Notes\
#_#
    def pick_sample(self, data, mask):
        """
        This function picks a sample of the data the size of a batch
        """
        #_# Steps\
        #_# Reorder the index randomly
        perm = torch.randperm(len(data))
        #_# Crop the index to the lentgh of the batch size
        index = perm[: self.batch_size]
        #_# Output the data selected by the random index
        return data[index], mask[index]
#-----------------------------------------------------------------------------
#_#
#__#10. Save the model
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function saves the weights of the two networks that are used in the GAN on the 'filepath'.
#_#Reviewer Notes\
#_#
    def save_model(self, filepath):
        """
        This saves the weights of the two networks that are used in the GAN on the 'filepath'.
        """
        #_#Steps\
        #_# Save Generator weights
        torch.save(self.Generator.state_dict(), filepath + "_generator.pkl")
        #_# Save Critic weights
        torch.save(self.Critic.state_dict(), filepath + "_critic.pkl")
        #_# Print that the process is complete
        print("Models saved ")
#--------------------------------------------------------------------------------
#_#
#__#10. Model loading
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function loads the weights of the two networks that are used in the GAN on the 'filepath'.
#_#Reviewer Notes\
#_#
    def load_model(self, filepath):
        """
        This loads the weights of the two networks that are used in the GAN on the 'filepath'.
        """
        #_#Steps\
        #_# Load Critic weights
        self.Critic.load_state_dict(torch.load(filepath + "_critic.pkl"))
        #_# Print that loading Critic weights is complete
        print("Critic model loaded from {}-".format(filepath + "_critic.pkl"))
        #_# Load generator weights
        self.Generator.load_state_dict(torch.load(filepath + "_generator.pkl"))
        #_# Print that loading generator weights is complete
        print("Generator model loaded from {}.".format(filepath + "_generator.pkl"))
```

A.3 MaWGAN's mask

```
from torch import tensor


#-------------------------------------------------------------------------------
#_#
#__#1. Make mask
#_#
#__#Review Decision:
#_#
#_#Author Notes\
#_#This function makes a mask that indicates where the missing data is, it
#_# outputs a numeral and binary mask.
#_#Reviewer Notes\

def make_mask(data):
    """
    This function makes a mask that indicates where the missing data is
    """
    #_#Steps\
    #_#The first step is to create a binary matrix which flags True
    #_#if there is data missing in this location
    binary_mask = data.isnan()
    #_# Convert the binary mask to a numeral mask
    inverse_mask = binary_mask.to(dtype=int)
    #_# Inverse the 0s and 1s so 0 is where the data is missing
    mask = 1 - inverse_mask
    #_# Outputs the numeral and binary mask
    return mask, binary_mask
```

A.4 Force-GAN

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
from torch import autograd
import time as t
from utlility import copy_format, make_mask
import os
import logging
import numpy


class FGAN(object):
    def __init__(
        self,
        optimiser,
        input_dim,
        noise_size,
        number_of_layers,
        lambdas,
        learning_rate,
        network,
        k_size,
    ):
        if network == "wgangp":
            self.network = "linear"
            print("old model")
            # backward comblilty with old models
        else:
            self.network = network.lower()
        self.net_dim = noise_size
        self.input_dim = input_dim
        self.data_dim = input_dim * k_size
        print("building", number_of_layers, self.net_dim, self.data_dim)
        self.Make_Generator(number_of_layers)
        print("Generator")
        self.Make_Critic(number_of_layers)
        print("Critic")
        self.Make_forcast(number_of_layers, input_dim)
        print("forcaster")
        # WGAN values from paper
        self.learning_rate = learning_rate

        # WGAN_gradient penalty uses ADAM
        self.d_optimizer = optim.Adam(
            self.Critic.parameters(), lr=self.learning_rate
        )
        self.g_optimizer = optim.Adam(
            self.Generator.parameters(), lr=self.learning_rate
        )
        self.f_optimizer = optim.Adam(
            self.forcaster.parameters(), lr=self.learning_rate
        )
        self.lambda_term = lambdas
```

```
def Make_Generator(self, number_of_layers):
    """
    This makes a generator network with 'number_of_layers' layers and
    'net_dim' of nodes per layer.It takes in a vector of 'batch_size'
    length and outputs a vector of data that is 'data_dim' long.
    """
    self.Generator = nn.Sequential()
    self.Generator.add_module(
        str(number_of_layers) + "Glayer", nn.Linear(self.data_dim, self.net_dim)
    )
    self.Generator.add_module(str(number_of_layers) + "active", nn.Tanh())
    number_of_layers -= 1
    while number_of_layers > 1:
        self.Generator.add_module(
            str(number_of_layers) + "Glayer", nn.Linear(self.net_dim, self.net_dim),
        )
        self.Generator.add_module(str(number_of_layers) + "active", nn.Tanh())
        number_of_layers -= 1
        print(number_of_layers)
    self.Generator.add_module(
        str(number_of_layers) + "Glayer", nn.Linear(self.net_dim, self.data_dim)
    )

def Make_Critic(self, number_of_layers):
    """
    This makes a critic network with 'number_of_layers' layers and 'net_dim' of nodes per layer.
    It takes in a vector of data that is 'data_dim' long and
    outputs a probability of the data being real or fake.
    """
    self.Critic = nn.Sequential()
    self.Critic.add_module(
        str(number_of_layers) + "Clayer", nn.Linear(self.data_dim, self.net_dim)
    )
    self.Critic.add_module(str(number_of_layers) + "active", nn.Tanh())
    number_of_layers -= 1
    while number_of_layers > 1:
        self.Critic.add_module(
            str(number_of_layers) + "Clayer", nn.Linear(self.net_dim, self.net_dim),
        )
        self.Critic.add_module(str(number_of_layers) + "active", nn.Tanh())
        number_of_layers -= 1
    self.Critic.add_module(
        str(number_of_layers) + "Clayer", nn.Linear(self.net_dim, 1)
    )

def Make_forcast(self, number_of_layers, input_dim):
    """
    input sinthetic
    """
    self.forcaster = nn.Sequential()
    self.forcaster.add_module(
        str(number_of_layers) + "Flayer", nn.Linear(self.data_dim, self.net_dim)
    )
    self.forcaster.add_module(str(number_of_layers) + "active", nn.Tanh())
    number_of_layers -= 1
    while number_of_layers > 1:
        self.forcaster.add_module(
            str(number_of_layers) + "Flayer", nn.Linear(self.net_dim, self.net_dim),
        )
        self.forcaster.add_module(str(number_of_layers) + "active", nn.Tanh())
        number_of_layers -= 1
    self.forcaster.add_module(
        str(number_of_layers) + "Flayer", nn.Linear(self.net_dim, input_dim)
    )

def create_fake(self, batch_size):
    """
    this creates a batch of fake data
    """
    z = torch.randn(batch_size, self.data_dim)
    if self.usegpu:
        z = z.cuda()
    fake_images = self.Generator(z)
    return fake_images  # .detach().numpy()

def create_series(self,length_of_series):
    batch_size = 1
    w = self.create_fake(batch_size)[0]
    end = self.data_dim - self.input_dim
    wfac = w[:end]
    series = [w[:self.input_dim].detach().numpy()]
    for i in range(length_of_series):
        b = torch.randn(self.input_dim)
        to_forcat = torch.cat((wfac, b))
        predic = self.forcaster(to_forcat)
        middle = wfac[self.input_dim:]
        wfac = torch.cat((middle, predic))
        series.append(middle[:self.input_dim].detach().numpy())
    series.append(predic.detach().numpy())
    return series


def create_series3(self,length_of_series):
    batch_size = 1
    w = self.create_fake(batch_size)[0]
    for i in range(length_of_series):
        b = torch.randn(self.input_dim)
        to_forcat = torch.cat((w, b))
        predic = self.forcaster(to_forcat)
        w = torch.cat(w, predic)
    return w.detach().numpy()

def linear_sample(self, data):
    "select samples that are linearly dependent"
    sizes = len(data) - self.batch_size
    start_loc = torch.randint(0, sizes, (1,))
    index = range(start_loc, start_loc + self.batch_size)
    return data[index]

def sample_type(self, data):
    if self.network == "linear":
        sample = self.pick_sample(data)
    else:
        sample = self.linear_sample(data)
    return sample

def fcast(self, w):
    b = torch.randn(self.batch_size, self.input_dim)
    end = self.data_dim - self.input_dim
    fac = torch.tensor([[1]*end + [0]*self.input_dim]*self.batch_size)
    invfac = torch.tensor([[0]*end]*self.batch_size)
    if self.usegpu:
        fac = fac.cuda()
        invfac = invfac.cuda()
        b = b.cuda()
    wfac = w*fac
    to_forcat = wfac + torch.cat((invfac, b), dim=1)
    predic = self.forcaster(to_forcat)
    together = wfac + torch.cat((invfac, predic), dim=1)
    return together

def psudo_training(self, data, tf, tsf, tc, mc, print_every_n_batches):
    """
    data_dim is the number
    """
    if self.usegpu:
        self.forcaster.cuda()
    # torch.autograd.set_detect_anomaly(True)
    d_loss_real, d_loss_fake = [0,0]
    data = torch.Tensor(data)
    nummask,bimask = make_mask(data)
    if self.usegpu:
        nummask = nummask.cuda()
    for s in range(tf):
        if s >= tsf:
            flag = True
            for p in self.Critic.parameters():
                p.requires_grad = True
            for t in range(tc):
                self.Critic.zero_grad()
                ubar = Variable(self.sample_type(data))
                if self.usegpu:
                    ubar = ubar.cuda()
                # for i in range(mc):
                ubar[bimask[self.index]] = 0
                initw = self.create_fake(self.batch_size)
                w = self.fcast(initw)
                wbar = w * nummask[self.index]
                d_loss_real, d_loss_fake = self.critic_update(
                    ubar, wbar
                )  # include calculate_gradient_penalty,loss calculation, critic weights opti
            # end for
        else:
            flag = False
        # end if
        # for i in range(mf):
        for p in self.Critic.parameters():
            p.requires_grad = False
        self.forcaster.zero_grad()
        intw = self.create_fake(self.batch_size)
        w = self.fcast(intw)
        floss2 = self.Critic(w)  # wk is not used
        # end = self.data_dim - self.input_dim
        # floss2 = floss[:,end:]
        floss2 = floss2.mean()
        if s % print_every_n_batches == 0:
            logging.info(
                f"iteration: {s}/{tf}({flag}),
                f_loss: {floss2:.2f},
                loss_fake: {d_loss_fake:.2f},
                loss_real: {d_loss_real:.2f}"
            )
        floss2.backward(
            self.mone
        )  # this funtions needs more info which i am going to add later
        self.f_optimizer.step()
    print("fc")

def train_critcgen(
    self, data, epochs, hasmissing=False, print_every_n_batches=10, n_critic=5,
):
    """
    This trains the GAN by alternating between training the critic 'critic_round' times
    and training the generator once in each epoch on
    the dataset x_train which has a length of batch_size.
    It will print and record the loss of the generator and critic every_n_batches.
    """
    print("critc train =", n_critic)
    if self.usegpu:
        self.Critic = self.Critic.cuda()
        self.Generator = self.Generator.cuda()
    if hasmissing:
        print("missing data mode on")
    data_tensor = torch.Tensor(data)
    for g_iter in range(epochs):
        # Requires grad, Generator requires_grad = False
        for p in self.Critic.parameters():
            p.requires_grad = True
        d_loss_real = 0
        d_loss_fake = 0
        Wasserstein_D = 0
        for d_iter in range(n_critic):
            self.Critic.zero_grad()
            sample = self.sample_type(data_tensor)
            images = Variable(sample)
            # Train discriminator
            z = Variable(torch.randn(self.batch_size, self.data_dim))
            if self.usegpu:
                fake_images = self.Generator(z.cuda())
            else:
                fake_images = self.Generator(z)
            if hasmissing:
                images, fake_images = copy_format(images, fake_images, self.usegpu)
            if self.usegpu:
```

```
                images = images.cuda()
            # Train with real images
            d_loss_real, d_loss_fake = self.critic_update(images, fake_images)
        # Generator update
        for p in self.Critic.parameters():
            p.requires_grad = False  # to avoid computation

        self.Generator.zero_grad()
        # train generator
        # compute loss with fake images
        z = Variable(torch.randn(self.batch_size, self.data_dim))
        if self.usegpu:
            fake_images = self.Generator(z.cuda())
        else:
            fake_images = self.Generator(z)
        g_loss = self.Critic(fake_images)
        g_loss = g_loss.mean()
        g_loss.backward(self.mone)
        g_cost = -g_loss
        self.g_optimizer.step()
        if g_iter % print_every_n_batches == 0:
            logging.info(
                f"iteration: {g_iter}/{epochs},
                g_loss: {g_loss:.2f},
                loss_fake: {d_loss_fake:.2f},
                loss_real: {d_loss_real:.2f}"
            )
        assert g_loss > 0 or g_loss < 0
    # Saving model and sampling images every 1000th generator iterations

def critic_update(self, images, fake_images):
    # Train with real images
    d_loss_real = self.Critic(images)
    d_loss_real = d_loss_real.mean()
    d_loss_real.backward(self.mone)

    # Train with fake images

    d_loss_fake = self.Critic(fake_images)
    d_loss_fake = d_loss_fake.mean()
    d_loss_fake.backward(self.one)
    # Train with gradient penalty
    gradient_penalty = self.calculate_gradient_penalty(
        images.data, fake_images.data
    )
    gradient_penalty.backward()

    d_loss = d_loss_fake - d_loss_real + gradient_penalty
    Wasserstein_D = d_loss_real - d_loss_fake
    self.d_optimizer.step()
    return d_loss_real, d_loss_fake

def calculate_gradient_penalty(self, real_images, fake_images):
    """
    Computes gradient penalty based on prediction and weighted real / fake samples
    """
```

```
        size = real_images.shape[0]
        eta = torch.FloatTensor(size, 1).uniform_(0, 1)
        if self.usegpu:
            eta = eta.cuda()
        eta = eta.expand(size, real_images.size(1))
        interpolated = eta * real_images + ((1 - eta) * fake_images)
        # define it to calculate gradient
        interpolated = Variable(interpolated, requires_grad=True)
        # calculate probability of interpolated examples
        prob_interpolated = self.Critic(interpolated)
        # calculate gradients of probabilities with respect to examples
        if self.usegpu:
            gradients = autograd.grad(
                outputs=prob_interpolated,
                inputs=interpolated,
                grad_outputs=torch.ones(prob_interpolated.size()).cuda(),
                create_graph=True,
                retain_graph=True,
            )[0]
        else:
            gradients = autograd.grad(
                outputs=prob_interpolated,
                inputs=interpolated,
                grad_outputs=torch.ones(prob_interpolated.size()),
                create_graph=True,
                retain_graph=True,
            )[0]
        grad_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * self.lambda_term
        return grad_penalty

    def full_train(
        self,
        data,
        batch_size,
        print_every_n_batches,
        n_gen,
        n_critic,
        n_forcast,
        tsf,
        usegpu,
        lag
    ):
        u_matrix = numpy.array(self.extend_data(lag, data, len(data)))
        self.one = torch.tensor(1, dtype=torch.float)
        self.mone = self.one * -1
        self.batch_size = batch_size
        print("gpu =", usegpu)
        self.usegpu = usegpu
        print("u", u_matrix.shape, "d", data.shape)
        self.train_critcgen(u_matrix, n_gen, True, print_every_n_batches, n_critic)
        torch.save(self.Generator.state_dict(), "quicksave_generator.pkl")
        torch.save(self.Critic.state_dict(), "quicksave_critic.pkl")
        print("gen training complet")
        self.psudo_training(
            u_matrix,
            n_forcast,
            tsf,
            n_critic,
            batch_size,
            print_every_n_batches,
        )
        if usegpu:
            self.Critic = self.Critic.cpu()
            self.Generator = self.Generator.cpu()
            self.forcaster = self.forcaster.cpu()

    def extend_data(self, k, x, n):
        U = []
        try:
            x[0,]
        except KeyError:
            x = x.to_numpy()
        for i in range(n - k + 1):
            temp = numpy.array([])
            for j in range(k):
                temp = numpy.append(temp, x[i + j,])
            U.append(temp)
        return U

    def summary(self):
        """
        prints the composition of the gan
        """
        print(self.Critic)
        print(self.Generator)
        print(self.forcaster)

    def pick_sample(self, data):
        """
        pick a smaple of the data of size of the batch
        """
        perm = torch.randperm(len(data))
        index = perm[: self.batch_size]
        self.index = index
        return data[index]

    def save_model(self, filepath):
        """
        This saves the weights of the two networks that are used in the GAN on the 'filepath'.
        """
        torch.save(self.Generator.state_dict(), filepath + "_generator.pkl")
        torch.save(self.Critic.state_dict(), filepath + "_critic.pkl")
        torch.save(self.forcaster.state_dict(), filepath + "_forcat.pkl")
        print("Models saved ")

    def load_model(self, filepath):
        """
        This loads the weights of the two networks that are used in the GAN on the 'filepath'.
        """
        G_model_filename = filepath + "_generator.pkl"
        D_model_filename = filepath + "_critic.pkl"
        F_model_filename = filepath + "_forcat.pkl"
```

```
D_model_path = os.path.join(os.getcwd(), D_model_filename)
G_model_path = os.path.join(os.getcwd(), G_model_filename)
F_model_path = os.path.join(os.getcwd(), F_model_filename)
self.Critic.load_state_dict(torch.load(D_model_path))
self.Generator.load_state_dict(torch.load(G_model_path))
self.forcaster.load_state_dict(torch.load(F_model_path))
print("Generator model loaded from {}.".format(G_model_path))
print("Critic model loaded from {}-".format(D_model_path))
print("forcaster model loaded from {}-".format(F_model_path))
```

A.5 Hankel Imputaion

A.5.1 Interface

```python
from .functions import *
import pandas as pd


def testmax_Lag(numpy_data,N,dim):
    """
    Test the Maximum number that lag can have a large lag gives a better results
    but too big have an error occurs
    """
    pnt = int(N / 2.8)
    for v in range(pnt):
        lag = pnt - v
        try:
            if dim > 1:
                construct(numpy_data, lag, dim)
            elif dim == 1:
                construct(numpy_data.transpose()[0], lag, dim)
        except ValueError as e:
            if lag % 5 == 0:
                print("less than", lag)
            if lag < int(N / 10):
                print(e)
                return
        else:
            print(lag)
            return lag


def batchfilling(numpy_data, mask, e, dim, batch_size, N, **kwags):
    """
    fill data in batches using hankel imputaion method
    """
    list_filled_dataframe = []
    lag = testmax_Lag(numpy_data[:batch_size],batch_size,dim)
    for batch in range(0, N, batch_size):
        endbatch = min(batch + batch_size,N)
        newbat = endbatch - batch
        if newbat > 1:
            print(batch, endbatch)
            if newbat < batch_size:
                    lag = testmax_Lag(numpy_data[:newbat],newbat,dim)
            filled_dataframe = filling(
                numpy_data[batch : endbatch],
                mask[batch : endbatch],
                lag,
                e,
                dim,
                **kwags,
            )
            list_filled_dataframe.append(filled_dataframe)
    return pd.concat(list_filled_dataframe)
```

```
def filling(numpy_data, mask, lag, e, dim, **kawgs):
    """
    fill data using hankel imputaion method
    """
    if dim > 1:
        filled = hankel_imputaion(numpy_data, lag, e, mask, dim, **kawgs)
    elif dim == 1:
        filled = hankel_imputaion(
            numpy_data.transpose()[0], lag, e, mask.to_numpy().transpose()[0], **kawgs
        )
    else:
        filled = hankel_imputaion(
            numpy_data, lag, e, mask, **kawgs
        )
    return pd.DataFrame(filled)


def processing(data, batch, e, **kawgs):
    """
    calulates the mask, the lag, the shape
    """
    print("epsilon:",e)
    numpy_data = data.to_numpy()
    mask = data.notna()
    predata = data.copy().interpolate().to_numpy()
    try:
        N, dim = data.shape
    except ValueError:
         N = len(data)
         dim = 0
    print("your data has",N,"timesteps and",dim,"varibles")
    if batch == 0:
        lag = testmax_Lag(numpy_data,N,dim)
        filled = filling(numpy_data, mask, lag, e, dim, predata=predata, **kawgs)
    else:
        filled = batchfilling(numpy_data, mask, e, dim, batch, N, predata=predata, **kawgs)
    return filled


def refillzero(dataframe):
    refill = dataframe.where(dataframe != 0.0)
    print("failed amount:", refill.isna().sum())
    return refill.interpolate()
```

A.5.2 Functions

```
import cvxpy as cp
import numpy as np

def cphmat(vec,hoz,dim,N):
    """
    hmat Hankel matrix from a vector "vec"

    """
    try:
        vert = 1 + N[0] - hoz
    except:
        vert = 1 + N - hoz
    col = []
    for i in range(vert):
        row = []
        for j in range(hoz):
            row.append(vec[i + j])
        col.append(row)
    return cp.bmat(col)


def hankel_imputaion(data, lag, e, mask, dim=1, double=True, predata=None,**kawgs):
    """
    by a convex optimation of minimation of the norm of hankle matrix of imputed variables
    with constraints = norm of mask of data and imputed variables less than a value e
    """
    if dim <= 1:
        print("monovar filling")
        N = len(data)
    else:
        print("Multivar filling")
        N = data.shape
    Yapp = cp.Variable(N)
    try:
        if dim <= 1:
            Yapp.value = predata.transpose()[0]
        else:
            Yapp.value = predata[:N[0]]
    except Exception as essa:
        print("no pre-data because of",essa,"\n",Yapp.shape,predata.shape)
    foward = cphmat(Yapp,lag,dim,N)
    objective = cp.Minimize(cp.normNuc(foward))
    constraints = [cp.norm((data[mask] - Yapp[mask])) <= e]
    prob = cp.Problem(objective, constraints)
    prob.solve(**kawgs)
    return Yapp.value



def construct(data, lag, dim):
    """
    sends a ValueError if lag is too big
    """
    N = data.shape
    Yapp = cp.Variable(N)
    cp.normNuc(cphmat(Yapp, lag, dim, N))
```