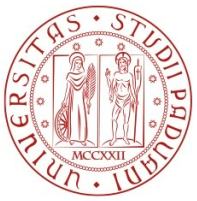

UNIVERSITÀ DEGLI STUDI DI PADOVA

Università degli Studi di Padova

Department of Information Engineering

Master Thesis in Electronic Engineering

”An explainable approach to single beat ECG generation using VAE”

Supervisor:
Prof. Giada Giorgi

Candidate:
Lorenzo Michelotti

Academic Year 2023-2024

Graduation Date 10/07/2024

Abstract

Electrocardiogram (ECG) is one of the most common noninvasive diagnostic tools used in clinical medicine. It is mostly used by cardiologists to assess heart function and electrophysiology in order to assist cardiovascular disease detection and development of treatment strategies.

In recent years, the rapid evolution of machine learning models has opened the possibility to automate a variety of tasks not feasible by traditional algorithms, such as the detection and classification of arrhythmia, biometric recognition, and even the diagnosis of other diseases.

Supervised machine learning models, especially deep neural networks, require a large amount of quality annotated data. The acquisition of ECG data, however, results difficult for several reasons, including privacy and other legal constraints.

A widely employed technique to overcome this kind of problem is data augmentation, which consists in increasing the volume, quality and diversity of a training dataset by generating additional data samples. It is especially useful in cases where data collection consumes a considerable amount of time and resources. One viable approach to data augmentation consists in estimating the underlying distribution of the data to be augmented. This enables the creation of new synthetic data, that exhibit a high degree of correlation with the existing ones, through a suitable sampling process. Recently, this approach reached state-of-art performances thanks to the embedding of neural networks into the probability estimators.

The aim of this thesis focuses on exploiting the potential of the VAE latent space by using its most visually interpretable representation, the two dimensional one. It is going to be shown how the model can be effectively used for two main purposes: the first one is generating new ECG data samples from the ones measured from a 'real' person, by also allowing an adjustable degree of variability. The second one is generating ECG data samples associated to a 'synthetic' person, which combines feature of the ECGs belonging to two or more 'real' people. The whole process can be directed by human intervention with a high degree of control and assisted by proper metrics that are used to evaluate the quality of the generated data. The proposed method allows the generation of single-beat ECGs having a fixed length. The temporal correlation among consecutive ECG beats and, in general, ECG trace variability, will be considered in a future work.

The first two chapters will be used to introduce the main topics of the thesis, such as the concepts of artificial intelligence, data augmentation, ECG signals, and explainable AI. The third one presents the theoretical background of the implemented variational autoencoder model

and its training process. The fourth one contains the specific of the chosen model architecture, while the fifth and sixth ones the application of the model for respectively a test sinusoid dataset and the assign ECG dataset. Finally, the last chapter is a sum up of the thesis work and results, along with some further work proposals.

Sommario

L'elettrocardiogramma (ECG) è uno degli strumenti diagnostici non invasivi più comunemente utilizzati in medicina clinica. Viene utilizzato principalmente dai cardiologi per valutare la funzione cardiaca e la sua elettrofisiologia al fine di assistere nel rilevamento di malattie cardiovascolari e nello sviluppo di strategie di trattamento.

Negli ultimi anni, la rapida evoluzione dei modelli di apprendimento automatico ha aperto la possibilità di automatizzare una serie di compiti non realizzabili dagli algoritmi tradizionali, come il rilevamento e la classificazione dell'aritmia, il riconoscimento biometrico e persino la diagnosi di altre malattie.

I modelli di machine learning supervisionati, in particolare le reti neurali profonde, richiedono una grande quantità di dati annotati di qualità. L'acquisizione dei dati ECG, tuttavia, risulta difficile per diversi motivi, tra cui la privacy e altri vincoli legali.

Una tecnica ampiamente utilizzata per alleviare questo tipo di problema è la "data augmentation", che consiste nell'aumentare il volume, la qualità e la diversità di un set di dati di addestramento generando ulteriori campioni di dati. È particolarmente utile nei casi in cui la raccolta di quest'ultimi consumi una notevole quantità di tempo e risorse. Un possibile approccio di "data augmentation" consiste nella stima della distribuzione di probabilità associata ai dati da aumentare di numero. Ciò consente la creazione di nuovi dati sintetici, che presentano un elevato grado di correlazione con quelli esistenti, attraverso un adeguato processo di campionamento. Recentemente questo approccio ha raggiunto prestazioni allo stato dell'arte grazie all'inclusione di reti neurali negli stimatori di probabilità.

Lo scopo di questa tesi si concentra sul massimo impiego del potenziale offerto dallo spazio latente del VAE, facendo uso della sua rappresentazione più visivamente interpretabile, quella bidimensionale. Verrà mostrato come il modello possa essere efficacemente utilizzato per due scopi principali: il primo è generare nuovi campioni di dati ECG basandosi su quelli misurati da una persona 'reale', consentendo anche di regolarne la variabilità. Il secondo consiste nel generare campioni di dati ECG di una persona 'sintetica', che combina le caratteristiche degli ECG appartenenti a due o più persone 'reali'. L'intero processo può essere diretto dall'intervento umano con un elevato grado di controllo e assistito da opportune metriche utilizzate per valutare la qualità dei dati generati. Il metodo proposto consente la generazione di ECG a singolo battito aventi lunghezza prefissata. La correlazione temporale tra battiti consecutivi di un ECG e, in generale, la variabilità del tracciato ECG stesso, verranno presi in considerazione in un lavoro

futuro.

I primi due capitoli sono utilizzati per introdurre i temi principali di questo lavoro di tesi, come ad esempio l'intelligenza artificiale, l'incremento artificiale dei dati, i segnali ECG e l'AI spiegabile. Il terzo, invece, presenta il background teorico relativo al modello di autoencoder variazionale implementato e la sua procedura di allenamento. Il quarto contiene le specifiche dell'architettura del modello, mentre il quinto e sesto l'applicazione di quest'ultimo per quanto riguarda il dataset di sinusoidi di prova e il dataset di elettrocardiogrammi assegnato. Infine, l'ultimo capitolo riassume il lavoro di tesi trattato e futuri potenziali miglioramenti e applicazioni del modello sviluppato.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 AI

Artificial Intelligence is a term coined in the 1950s, which encompasses the simulation of human intelligence processes by machines. Initially characterized by symbolic reasoning and rule-based systems, AI has evolved significantly with advancements in computational power and algorithm performances. Early AI systems focused on explicit programming to perform specific tasks, such as playing chess or simple decision making. However, the need for more adaptive and autonomous systems led to the emergence of Machine Learning.

Machine learning is the latest emerging branch of artificial intelligence: it is focused on the development of algorithms capable of enabling a machine to perform tasks without explicit instruction. Unlike conventional programming paradigms, the machine autonomously "learns" previous data in order to make, through statistical modelling and/or pattern matching, classification or prediction about future unknown data. A wide variety of definitions had been proposed to address this phenomenon, such as the one given in 1997 by Tom M. Mitchell

> *"A computer program is said to learn from experience E with respect to some class of tasks T and a performance measure P if its performance in tasks T, as measured by P, improves with experience E"*

Typically a machine learning algorithm involves processing multidimensional data, where each dimension is usually addressed as a feature, in order to perform a target task. Some of the most common ones are regression, classification, clustering, transcription, anomaly detection, synthesis and sampling. In order to train(teach) and finally evaluate the performance of a ML algorithm there is a need of a proper quantitative measure, which can vary based on the type of task and the type of data on which the model is trained on. Depending on the applied learning methodology, those algorithms can be classified into three main classes:

- *Supervised learning* : algorithms are trained using labeled dataset, wherein each data instance is associated to a target outcome or label, hence $\mathbf{x} \in D^m \longrightarrow y \in C$ from a training

dataset $S : \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)\}$. It is widely use in applications where a large number of labeled data is available, such as image classification, speech and text recognition, fraud detection and spam filters. Some of the models that come under supervised learning are Linear and Logistic regressions, Support Vector Machines, Gaussian Processes and Artificial Neural Networks.

- *Unsupervised learning* : the learning involves training algorithms on unlabeled data, where the objective is to discover inherent structures, patterns, or relationships without an explicit guidance. In this case the training set is made by only input data samples $S : \{\mathbf{x}_1, ..., \mathbf{x}_m\}$. Some of the most important algorithms are: Clustering, Principal Component Analysis and Anomaly Detection.

- *Self-supervised learning* : Self-supervised learning is a type of machine learning where the model learns from the data itself without requiring labeled input. The learning process makes use of the structure within the data itself to generate its own labels. For example, if it is wanted a model to encode a latent representation of the data $S : \{\mathbf{x}_1, ..., \mathbf{x}_m\}$, then the encoded data can be decoded back by a second model obtaining the reconstructed samples $\{\hat{\mathbf{x}}_1, ..., \hat{\mathbf{x}}_m\}$. Then the two networks can be trained jointly and implement an error function based on a distance metric between the original and decoded data samples, such as the MSE $l(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2$. Some deep learning algorithms that use self-supervised learning are Autoencoders, Variational Autoencoders and GANs.

- *Reinforcement learning* : it is a more generic learning framework which allows the machine decision making process to be driven by the state of the surrounding dynamic environment with the help of a reward based feedback loop. The machine makes a decision through an 'agent', which then has some repercussions on the dynamic 'environment' and makes its state change. Finally an 'interpreter' evaluates the state changes of the environment and, based on them, updates the 'agent' decision making process.

Artificial Neural Networks are one of the most used machine learning algorithm nowadays, they reached state of art performance in a large number of different applications. Those kind of algorithms constitute themself a subfield of machine learning, called **Deep Learning** (**DL**). The fast development of AI in the last decade is motivated by the fact that they are the only model which performs well in tasks where there is a need of interpreting really high dimensional data, such as images, videos, time series data and so on. The throwbacks, however, are that, respect to other algorithms, they require much more computational power, both for prediction and training, and they considerably lack of interpretability.

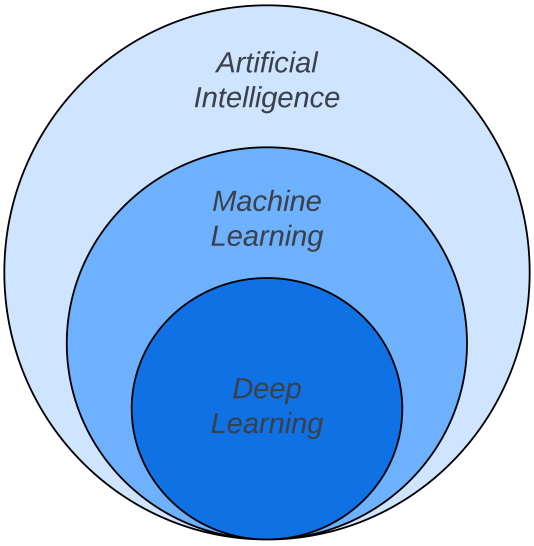

Figure 1.1: Artificial Intelligence, Machine Learning and Deep Learning

Nowadays, the development of hardware accelerators has made possible the training and deployment of very deep ANN architectures, even if with some intensive resource efforts. One of the biggest model released during 2023 is ChatGPT-4, that has been estimated to have more than 1 trillion parameters. The training seems to have required few tens of thousands of the best performing GPUs on the market and lasted for about three months. Nowadays, the most commonly explored and used hardware architecture for training and inference of neural networks are:

- CPU: the Central Processing Unit is a general purpose hardware responsible of handling the majority of tasks of a computing device. While being extremely versatile, low power and also low cost, it lacks of good parallel processing capabilities and is prone to memory bottlenecks. For this reasons its performances decrease proportionally to the model complexity and training dataset size, hence it is generally not suited for handling deep learning algorithms.

- GPU: the Graphic Processing Unit is an hardware accelerator originally designed to render graphics and video. The architecture is optimised for high throughput and parallel computing, which make them ideal for handling a large amount of data concurrently and performing operations that can be parallelised really fast. Moreover they are design to be easily scaled, making it possible to increase the performance even more. Some of the most commonly accelerated mathematical operations are matrix multiplication, convolution, element wise operations, gradient computation and much more. Those are the core of most of machine learning training and inference algorithm. The more complex those kind of calculus becomes(bigger matrices, larger convolutional filters, higher dimensional gradients), the more effective the parallelise computing carried on by the GPU becomes

effective. That said, this kind of hardware is the most common choice when working with deep learning algorithms, especially for the training stage. The main throwback is the cost per-unit and the low power efficiency, which bounds its use primarily within data centres.

- TPU: Tensor Processing Unit is an ASIC accelerator that can be thought as a close GPU parent, but specialised in processing ANN algorithms. They were among the first hardware accelerator focusing on deep learning tasks.

- FPGA: Field Programmable Gate Arrays is a semiconductor device that is based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. Most of modern models implement also hard blocks, such as SRAM, DSPs, eNVMs and so on. Its main feature is that it can be reprogrammed to meet the desired functionality requirements after manufacturing multiple times. Their flexibility allows for designing hardware targeting the specific algorithm to be implemented: by highly render in parallel operations, a low clock frequency can be selected, allowing for a much more efficient execution in terms of low latency, high throughput and low power consumption. Its flexibility makes it ideal for a wide range of algorithmic computational tasks, including deep learning ones.

Nowadays there is a huge interest in developing hardware aimed to speed up deep learning algorithms, such as TPUs, and given the threshold of fabric miniaturisation, it is up to the fantasy of the hardware designers to come up with new architectural principles.

1.2 XAI: Explainable Artificial Intelligence

AI systems such as machine learning (ML) or deep learning (DL) use algorithms learned by their own process of training, rather than by explicit human programming. During this process, they can discover new correlations between features of the input data and make decision(classification) or prediction(regression). If the models are highly complex and involve a large number of interacting parameters, it becomes more challenging to understand how the model behave, i.e. how the output is subsequently produced by the model. The reasons for which systems have made certain decisions may be unclear in these situations: this phenomenon is commonly referred to as the “black box” effect. The main problem related to this behaviour is that some of system deficiencies, such as biases, inaccuracies or ”hallucinations” can be hidden, having a direct impact on individuals, potentially being discriminatory or even harmful. For example, a not balanced training dataset could cause model biases that are hidden by the opacity of the model itself. An AI model used to select job applicants it may favour candidates from certain demographics or backgrounds, or applied for medical diagnosis purpose could disproportionately misdiagnose or miss certain conditions. These are some of the main reasons for which there is an urgency to open this ”black box” and make its processes more explainable.

Explainable Artificial Intelligence (XAI) is the ability of AI systems to provide clear and understandable explanations for their actions and decisions. Its central goal is to make the behaviour of these systems understandable to humans by making clearer the underlying mechanisms of their decision-making processes. A more complete definition is given by David Gunning [6]

> *"Ideally, XAI should include the ability to explain the system's competencies and understandings, explain its past actions, ongoing processes and upcoming steps, and disclose the relevant information on which its actions are based."*

This concept may be divided into three main components:

- *Transparency*: refers to the ability for a specific model to be understood. In the strictest sense, a model is transparent if a person can contemplate the entire model at once. A second and less strict notion of transparency might be that each part of the model (e.g., each input, parameter, and computation) admits an intuitive explanation [5]. It is fundamental to be able to design a model which aligns with ethical standards and legal requirements.

- *Interpretability*: refers to the degree of human comprehensibility of a given "black box" model or decision [4] [3]. Poorly interpretable models "are opaque in the sense that when presented with the resulting decision, rarely does one have any concrete sense of how or why a particular classification has been arrived at from inputs [2]. It allows the human designer or user to be able to estimate the behaviour of the model given a specific input and also if it makes a mistake.

- *Explainability*: it focuses on providing clear and coherent explanations for specific model predictions or decisions. It aims to offer human-understandable justifications for a specific outcome. Explainability requires interpretability as a building block but also looks to other fields and areas, such as human-computer interaction, law, and ethics [1]. It is fundamental in applications where human health or sensitive informations are at stake.

1.3 ECG

An electrocardiogram (ECG) is a recording of the electrical activity of the heart thought repeated cardiac cycles. It is an electogram displaying voltage versus time of the electrical activity of the heart using electrodes placed on the skin. These sensors are able to detect small voltage changes caused by the heart depolarization and repolarization phases. By ECG it is usually meant a 12-lead ECG with 10 electrodes placed on the patient's limb and chest. The overall magnitude of the heart's electrical potential is then measured from twelve different reference points ("leads") and is recorded over a certain period of time. A standard single beat ECG is composed by three main components: the P wave (depolarization of the atria), QRS complex (depolarization of the ventricles) and T wave(repolarization of the ventricles)

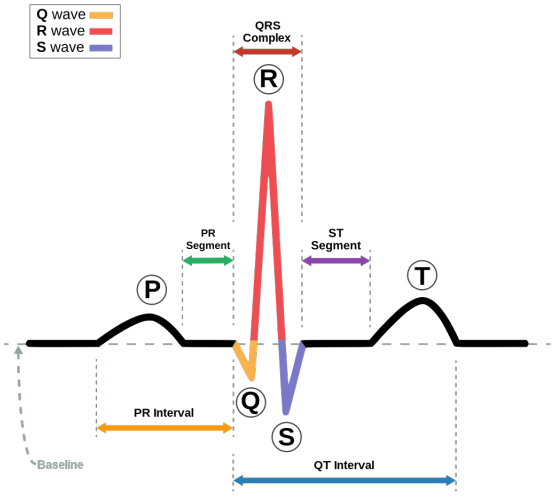

Figure 1.2: ECG cardiac beat model (Wikipedia)

Electrocardiogram (ECG) is one of the most common noninvasive diagnostic tools used in clinical medicine. It is mostly used by cardiologists to assess heart function and electrophysiology in order to assist in cardiovascular disease detection and development of treatment strategies. Due to a significant rise in the volume of ECG data and variations in measurements stemming from diverse medical devices and their placements, traditional diagnostic methods often prove inefficient. This is due to the necessity for intricate manual analysis and the expertise of highly trained medical professionals to attain satisfactory accuracy. During the past few decades, the computing power upgrades and the availability of larger datasets led to an increased use of machine data-driven models in many healthcare areas. In particular deep learning algorithms have been widely adopted since they achieved stat-of-art performances above all others machine learning approaches. These algorithms leverage deep neural networks to autonomously analyze ECG data, offering the potential to streamline diagnostic processes, enhance accuracy, and improve patient outcomes. This paradigm shift towards automated analysis, not only addresses the challenges posed by the expanding volume of ECG data, but also has the potential of facilitating early disease detection and personalised treatment strategies. For example Miao et al [7] proposed a combination of ResNet and LSTM in order to estimate blood pressure(BP) that achieved a mean average error of $-0.75\ mmHg$ for systolic BP and $2.228\ mmHg$ for diastolic BP prediction using a private database. Lu et al [8] used a 1D-CNN for arrhythmia classification, achieving an accuracy of 99.31% on the MIT-BIH Arrhythmia Database. Chiu et al [9] also used a 1D-CNN neural network to perform biometric recognition and achieved an identification rate of 99.10% by using single-lead ECG recordings that originated from the PTB Diagnostic ECG Database. Ozdemir et al [10] used a private database to diagnose COVID-19 through ECG classification (accuracy 93.00%). Moreover a study by Baghersalimi et al [11] evaluated the performance (sensitivity 90.24% and specificity 91.58%) of a fused 1D-CNN-ResNet network to detect epileptic seizure events from single-lead ECG signals originating from a private database.

There are however still some critical challenges in obtaining good quality annotated dataset due to several reasons. The first one regards the lack of accessibility and standardization of biometric data across healthcare systems. While some institutions may have robust biometric databases, others may lack the infrastructure or resources to collect and store such data efficiently. This creates discrepancies in data availability, hindering interoperability and the seamless exchange of medical information. Ethical considerations further complicate the acquisition of biometric data in healthcare. Privacy concerns, consent issues, and the risk of data breaches underscore the need for stringent regulations and ethical guidelines governing the collection, storage, and usage of biometric information. For what concerns ECG dataset, the vast majority of publicly available ones experience strong class imbalances, which, if not properly managed, could lead to model biases. For example, this can happen when trying to predict smaller classes that usually represent rare conditions, which are far less populated than larger, more common ones, representing healthy conditions.

1.4 Data augmentation

To ensure good performances, in terms of cost function and model generalisation, modern supervised machine learning models, especially deep learning ones, require a large amount of quality annotated data. Data collection and annotation processes are usually performed manually, and consume a lot of time and resources. Moreover, the quality and representativeness of curated data for a given task is usually dictated by the natural availability of clean data in the particular domain as well as the level of expertise of developers involved. In many real-world application settings it is often not feasible, for different reasons, to obtain a sufficient amount of training data for a proper train of the application to be developed. For those cases, data augmentation can solve or at least alleviate this issue.

Data augmentation is the process of increasing the volume, quality and diversity of a training dataset by generating additional training data samples. A wide range of basic deterministic transformations have been applied in the context of image data augmentation, such as cropping, scaling, mirroring, colour augmentation and so on. Those techniques, even if they are straightforward to apply, are not suited for all kinds of data. For examples time series data are one-dimensional, have an intrinsic sequential nature and often exhibit high temporal correlation between samples which varies greatly depending on the type of time series data (stock prices, ECG, temperature measures, ..). Therefore, in order to apply a traditional deterministic data augmentation transformations to those data, there must be some expertise knowledge about the data background in order to carefully adapt those augmentation algorithms, otherwise generating poor quality data could compromise the performance of the trained model.

Traditional techniques mostly fail in time domain manipulation of time series data, however those techniques could still be applied in general to manipulate the amplitude. For example using uniform scaling and/or added noise could be useful to simulate different conditions of the

measure environment.

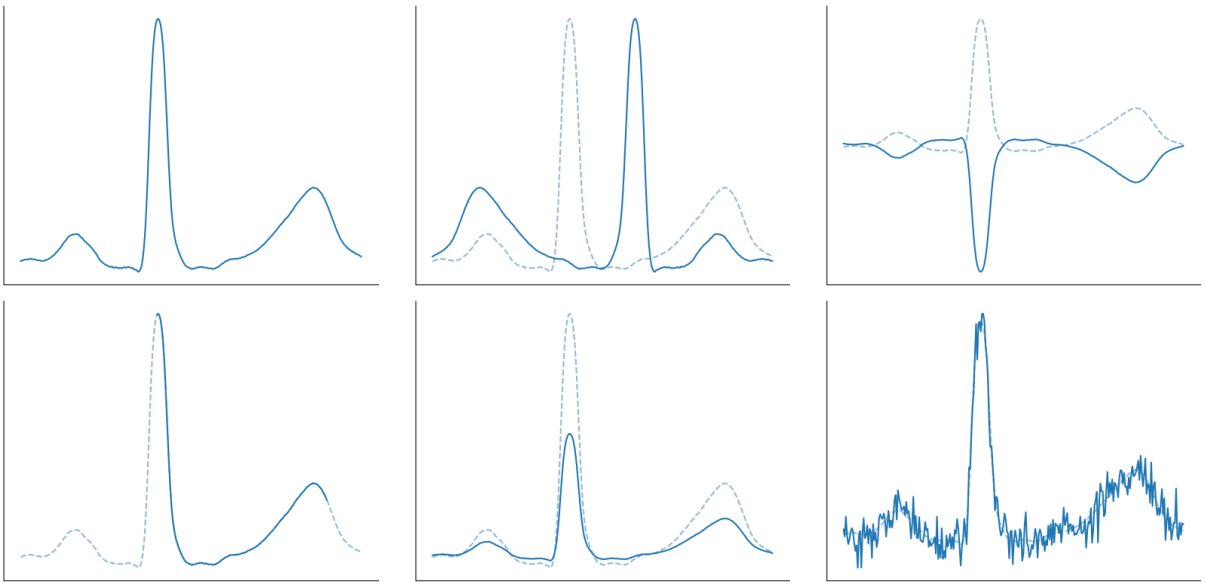

Figure 1.3: Common data augmentation transformation applied to a single beat ECG signal. (Top row) the original signal, mirroring and rotation. (Bottom row) cropping, amplitude variation and noise adding

Figure 1.4: Common data augmentation transformation applied to an image. (Top row) original image, mirroring and rotation. (Bottom row) cropping, brightness variation and noise adding

Another approach to data augmentation consist in trying to estimate the underlying distribution of the data to be augmented in order to sample synthetic datapoints that are highly correlated

with existing ones. Recently this approach has reached state-of-art performance thanks to the embedding of neural networks to the probability estimators. Through rigorous training on a limited dataset, the model can discern the underlying data features and approximate their probability distribution. This distribution can subsequently serves as a basis for sampling new data instances.

The most common generative model topologies used to perform data augmentation nowadays are *Generative Adversarial Networks* (GAN) and *Variational Autoencoders* (VAE). The first one is a deep learning architecture which trains two distinct neural networks to compete against each other to generate new, more authentic data from a given training dataset. A GAN is defined as "adversarial" because it trains two different networks and pits them against each other. One network generates new data by taking a sample of input data and modifying it as much as possible. The other network attempts to predict whether the generated data output belongs to the original data set. In other words, the prediction network determines whether the generated data is false or real. The system generates newer and improved versions of false data values until the prediction network can no longer distinguish the false values from the original ones. The second one is a probabilistic generative models composed by two neural networks, mainly referred to as the encoder and decoder. The encoder maps the input variable to a latent space that corresponds to the parameters of the estimated probability distribution of the input data. The decoder receives as inputs encoded samples from this probability distribution and decodes them back into the input dimensional space.

1.5 State of art

The work of this thesis specifically regards synthetic ECG generations. Previous work was already carried on in some studies: most of them make use of GAN, which turns out to be the best performing models, while some recent papers explored the use of VAE, which compared to GAN have less feature extraction capabilities, but have the advantage to be characterised by a much more explainable working principle.

In recent years, one of the most productive applications of deep learning in the domain of electrocardiogram (ECG) analysis has been discovered to lie in the generation of realistic synthetic ECG signals. Various studies, including [15],[16] and [18], have proposed combinations of 1D-CNN and Bidirectional Long Short-Term Memory (BiLSTM) GANs to generate single beat ECG signals, which have demonstrated state-of-the-art performances. Furthermore, [15] and [18] were able to respectively generate up to 2 and 8 beat-long synthetic ECG.

More recently, some other studies have advanced Variational Autoencoder (VAE) model-based solutions.[12] and [13] proposed a 1D-CNN VAE framework for the generation of single-lead, single-beat electrocardiogram (ECG) signals, while [14] introduced a 2D-CNN Conditional VAE (CVAE) approach capable of simultaneously generating synthetic ECG signals for 12 leads. Additionally, [17] advocated for the utilization of Gramian Angular Summation/Difference Fields

and Markov Transition Fields to convert single-beat ECG time series into image representations. Subsequently, they employed a Denoised Diffusion Probabilistic Model (DDPN) to generate new ECG image samples, yet achieving unsatisfactory performance outcomes.

Model	**Input**	**Dataset**	**Metric**	**Refs.**
1D-CNN VAE	ECG (1 beat)	LUDB	MMD(3.83×10^{-3})	[12]
VAE	ECG (1 beat)	UT-Heart	PSNR(64.84)	[13]
2D-CNN CVAE	ECG (1 beat)	UK-BB	MMD(3.05×10^{-3})	[14]
BiLSTM and 1D-CNN GAN	ECG (2 beats)	MIT-BIH	MMD(1.03×10^{-3}) DTW(11.664)	[15]
1D-CNN GAN	ECG (1 beat)	MIT-BIH	LSTM classifier acc. imprv. (4%)	[16]
DDPM	Images of ECG (1 beat)	MIT-BIH	MMD(35.9) DTW(6.36)	[17]
BiGridLSTM and 1D-CNN GAN	ECG (8 beats)	MIT-BIH	RMSE(0.126) PCC(0.991)	[18]

Table 1.1: Benchmark of already existing deep learning models for ECG signal generation

Chapter 2

Theoretical background

2.1 Early idea

The first attempt to develop a mathematical framework for a neural network took place in 1970's, by neurophysiologist Warren McCulloch and mathematician Walter Pitts. They modelled neural activity by means of propositional logic and also coined the widely used term "Artificial Neural Networks".
The basic computational unit of the brain is a neuron: each neuron can receive as inputs electrical stimulus from its dendrites, which are then processed in its body where their electrical potential gets summed up. If the sum reaches a certain threshold the neuron fires as output an electric signal along its single axon. The axon eventually branches out and connects via synapses to dendrites of other neurons(fig.2.1). The synaptic strengths can change thru time and experiences and control the strength of influence and its direction, excitatory or inhibitory of one neuron on another. This is also the principle of *feedforward* AANs.

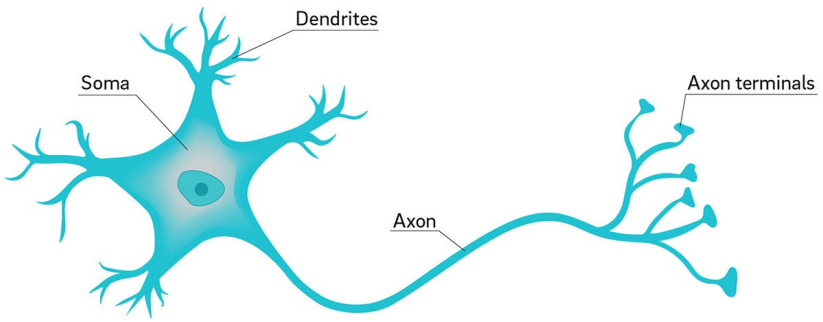

Figure 2.1: Main components of a biological neuron

2.2 Perceptron: first modern day neural network model

The most significant step forward in the development of an Artificial Neural Network was achieved by the psychologist Frank Rosenblatt in 1957. In the paper "The Perceptron: a perceiving and recognising automation", he proposed the Perceptron, a model capable of performing linear binary classification on a given labeled dataset. As hypothesis the dataset is composed in general by m training data of dimension d , hence $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$ labelled by $\mathbf{y} = \{y_1, .., y_m\}$ with $y_i = \{-1, 1\}$. The perceptron model performs the classification of a datapoint $\mathbf{x}'$ based on the sign of the value assumed by a "properly" chosen hyperplane $\mathbf{w}^T\mathbf{x}$ in that point

$$h_{\mathbf{w},\mathbf{b}}(x) = \text{sng}(\mathbf{w}^T\mathbf{x} + b) = \begin{cases} +1 & \mathbf{w}^T\mathbf{x} + b \geq 0 \\ -1 & \mathbf{w}^T\mathbf{x} + b \leq 0 \end{cases}$$

The model is trained by initialising the weights to zero $\mathbf{w}^{(0)} = 0$ and then by adjusting the weights in an iterative way as explained in algorithm 1. It can be demonstrated that the model converges to a solution if the datapoints are linearly separable, however if this condition is not satisfied there are no guarantees on the outcome. For example the model is not able to perform the binary *xor* operation. This problem makes the model not suited for most of practical cases. Moreover even if a proper solution is found, it is in general not optimal.

Algorithm 1 *Perceptron algorithm*

$\mathbf{w}, \mathbf{x}_i \in \mathbb{R}^d$
$y_i \in \mathbb{R}$
$\mathbf{w}^{(0)} \leftarrow 0$
while $\exists\, x_i : y_i[\mathbf{w}^{(k)}]^T\mathbf{x}_i \leq 0$ **do**
  select $x_i : y_i[\mathbf{w}^{(k)}]^T\mathbf{x}_i \leq 0$
  $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)s} + y_i\mathbf{x}_i$
end while

At the time, the initial enthusiasm around the Perceptron was damped in a short time due to the discovery of its inability to classy non linearly separable data. However this model contains, even if in a simplified way, the foundations of modern day neural network architecture and their optimisation algorithm. The entire model, in fact, has the same structure of the basic building block of a DNN ("Deep Neural Network"), the *neuron*. Neurons, however, depending on the application, can have various functions applied to the output of the linear combination of the inputs with the neuron weights, these are called $activation functions$. Moreover the Perceptron algorithm is just a special case of the SGD ("Stocastic Gradient Descent"), where both the size of the mini-batch B and learning rate γ are equal to one ($|B|, \gamma = 1$) and the loss function to

minimise is

$$l(\mathbf{w}, \mathbf{z}_i) = \begin{cases} -y_i \cdot \mathbf{w}^T \mathbf{x}_i & y \cdot \mathbf{w}^T \mathbf{x}_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbf{z}_i = \{\mathbf{x}_i, y_i\}$. Under those hypothesis the SGD and the Perceptron algorithm are equivalent given that the weights update step of SGD becomes the following

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \gamma \nabla_{\mathbf{w}} J_{B^{(k)}}(\mathbf{w}) \qquad \text{with } J_{B^{(k)}}(\mathbf{w}) = \frac{1}{|B|} \sum_{i \in B^{(k)}} l(\mathbf{w}, \mathbf{z}_i)$$

$$\Rightarrow w^{(k+1)} = w^{(k)} + \gamma \begin{cases} y_i \cdot \mathbf{x}_i & y_i \cdot (\mathbf{w}^{(k)})^T \mathbf{x}_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

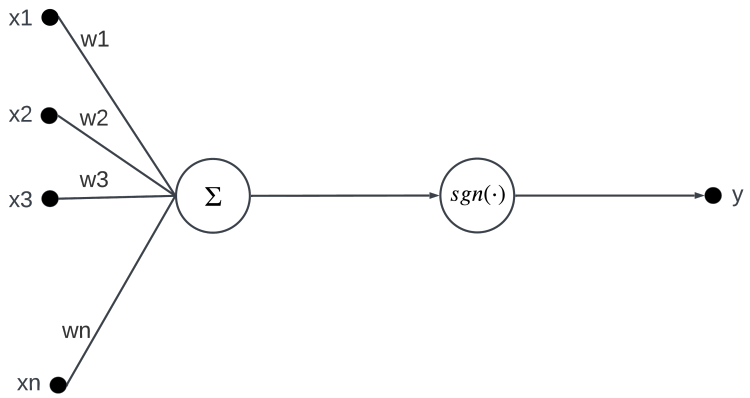

Figure 2.2: Block diagram of the Perceptron model

2.3 Fully connected neural networks

The basic building block of neural network is the *neuron*, a single computational unit. This unit takes a weighted sum of its inputs and adds to it a scalar *bias term*. Given a set of inputs $\mathbf{x} = \{x_1, x_2, ..., x_n\}$ the unit has associated a set of weights $\mathbf{w} = \{w_1, w_2, ..., w_n\}$ and a bias term b. Those quantities are in general real valued numbers. The resulted weighted sum is scalar

$$z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}^T \mathbf{x} + b$$

Instead of using just the linear combination of the inputs$z$ as the output, the neural units apply a non linear function f to z, called *activation function*

$$y = f(z) = f(\mathbf{w}^T\mathbf{x} + b)$$

Some of the most popular non-linear activation functions are the *sigmoid*, the *tanh* and the $ReLU$. The sigmoid maps the output into the range of $(0, 1)$ and it is differentiable within its domain.

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

The hyperbolic tangent (tanh) is a variant of the sigmoid which ranges from $(-1, 1)$. It is still differentiable.

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The rectified linear unit (ReLU) function is among the most used activation functions. It is a piecewise linear function non differentiable in the origin.

$$y = ReLU(z) = \max(z, 0)$$

Each activation function has different proprieties that makes them more suited for specific kind of neural networks architectures. For example the hyperbolic tangent has the advantage of being smoothly differentiable and maps outlier values towards the mean. Sigmoid and tanh functions, however, have saturated outputs, which makes the derivative very close to zero. Zero derivatives causes problem during the training of the neural network.

The network is trained using the *backpropagation method*, which consists in propagating an error signal, from a properly chosen loss function, backwards, by multiplying gradients from each layer of the network. If some gradients are close to zero, the propagation error gets smaller and smaller until it cannot be used to properly update the network parameters. This phenomenon is called *vanishing gradient*. Rectifiers linear unit, on the other hand, does not have this problem, since the derivative for $z > 0$ is always equal to 1. Moreover the piecewise linear structure makes it computationally efficient.

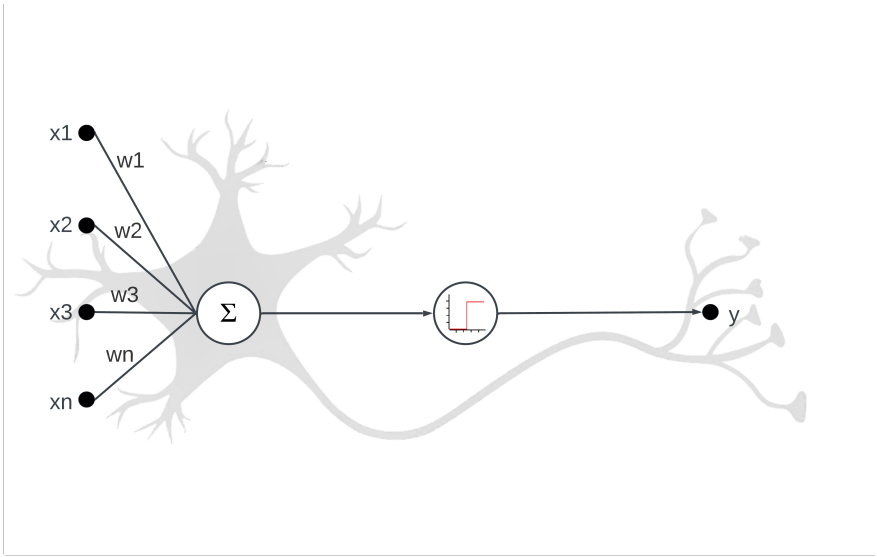

Figure 2.3: Feed forward neural network basic component: the neuron. It applies to the inputs a weighted linear combination, then the result is passed to a non linear activation function, such as the ReLU

2.4 Feedforward neural networks

A *feedfarward neural network* is a multilayer network in which the neurons are connected with no cycles. The output of the neurons of one layer are connected only to the neurons of the subsequent layer. Those kind of networks are also known by the name of *multi-layer perceptrons*, which is technically a misnomer since those network are characterised in general by non linear activation functions. There are three main kind of layers: input, hidden and output. The core of a fully connected feed forward neural networks is the hidden layer, composed by a specific number of neurons, each of which takes as inputs the weighted sum of all the outputs of the previous layer plus a bias and applies to it a non linear function. The output of the i-th neuron of the l-th layer is mathematically formulated as follows $y_i^{(l)} = (\sigma^{(l)}(\mathbf{w}_i^{(l)})^T \mathbf{y}^{(l-1)} + b_i^{(l)})$, where $\mathbf{y}^{(l-1)} = \{y_1^{(l-1)}, ..., y_m^{(l-1)}\}$ are the outputs of the previous layer composed by m layers, $\mathbf{w}_i^{(l)} = \{w_{i,1}^{(l)}, .., w_{i,m}^{(l)}, \}$ are the weights of the weighted sum, $b_i^{(l)}$ is the bias term and $\sigma^{(l)}$ is the activation function used by the neurons of the l-th layer. The output of such a network is then equal to

$$\mathbf{y}^{(l)} = \sigma^{(l)}\left(\mathbf{W}^{(l)}\mathbf{y}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

for deep neural network it is essential to choose a non linear $\sigma(\cdot)$ activation function, otherwise the resulting network is exactly equivalent to a single layer one. Consider the case of a two layer

neural network with an identity $\sigma(\mathbf{x}) = x$ activation function and input $\mathbf{x}$

$$\begin{aligned}\mathbf{y}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{y}^{(2)} &= \mathbf{W}^{(2)}\mathbf{y}^{(1)} + \mathbf{b}^{(2)} \\ &= \mathbf{W}^{(1)}\mathbf{W}^{(2)}\mathbf{x} + \mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)} \\ &= \mathbf{W}'\mathbf{x} + \mathbf{b}'\end{aligned}$$

which easily generalise to any number of layers.

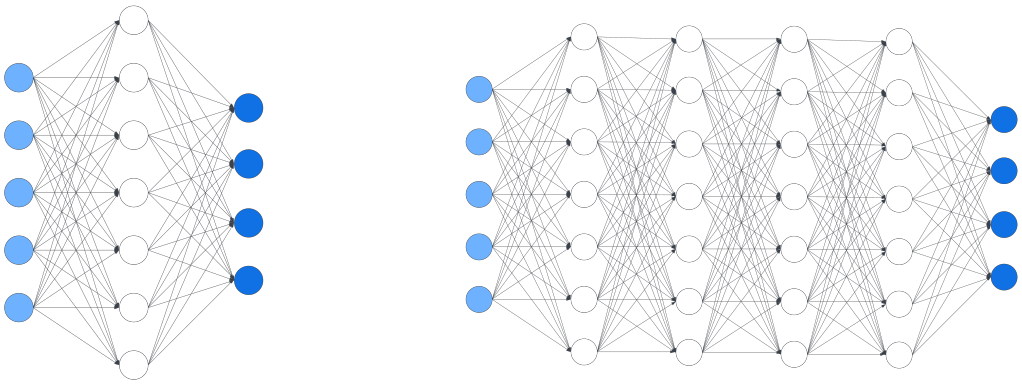

Figure 2.4: A single layer(left) and multi-layer(right) feed forward fully connected neural networks. The first one is characterised by only a single hidden layer, while the second one by multiple hidden layers. For this reason it is also called *deep* neural network. The light blue neurons form the input layer, while the dark blue ones form the output layer

2.5 Convolutional neural networks

Convolutional neural networks(CNN) are a kind of neural networks specialized in processing grid-like data, such as time series data (1D-grid of samples taken at regular time intervals) or images (2D-grid of pixels). A network can be defined convolutional if it implements the mathematical convolution operation in one of its layers. Since the work of the thesis deals with time signals only, the focus of this chapter will be on 1D convolution.

Given two Lebesgue integrable functions $f, g \in \mathbb{R}$ of variable $t \in \mathbb{R}$, the convolution between f and g is defined as

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

The first and second convolution argument functions are respectively referred to *input* and *kernel*, while the output is called *feature map*. When data are discretized, hence $t \longrightarrow nT_s$, with T_s the

sampling period, the discrete convolution is applied

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

In machine learning applications, usually the input is a multidimensional array of samples and the kernel is a multidimensional array of parameters tuned by the training algorithm.

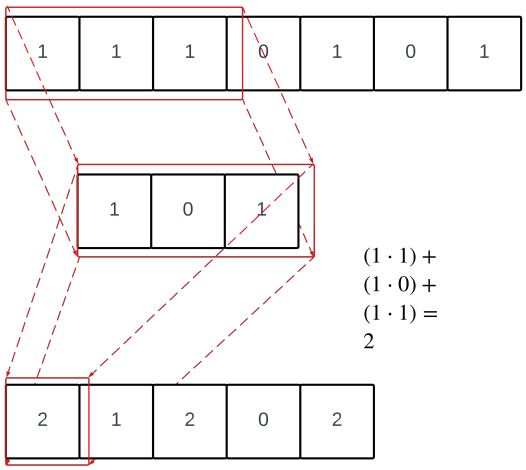

Figure 2.5: Example of the convolution operation carried on for discrete time signals. The upper sequence of numbers is the input discrete time series, the middle one is the convolutional kernel and the lower one is the output of the convolutional operation

Fully connected(dense) layers apply matrix multiplication between a specific matrix of trainable parameters associated to each neuron of the layers and all the inputs of the layer. This means that every output unit interacts with every input ones. On the other hand, convolutional layers have *sparse interactions*, since the kernel is usually smaller than the input. For example, if there is a time signal of specified length, a much smaller filter can be used to detect most relevant high frequency signal features. This has the advantages of reducing the number of stored model weights and reducing the number of operations used to obtain the output. Moreover, in a deep convolutional network, units in the deepest layer indirectly interact with larger portions of the input(fig. 2.8), allowing the network to also capture complicated interactions between many variables with simpler building blocks based on sparse interactions. Convolutional layers are also characterized by *parameters sharing*(or *tied weights*). In such a layer, each weight of the kernel is used at every position of the input(except for the boundary), so instead of learning a different set of parameters for each input position, the layer learns only one, the kernel. For example, if there are m inputs and n outputs the computational complexity of a fully connected layer is $O(mn)$ and the number of weights to be stored are mn. On the other hand, a convolutional layer with just one kernel of size k, which is usually much smaller than m, has the computational complexity of $O(mk)$ and the weigths to be stored are just k. An important property of the

convolutional layer is the *equivariance to translation*. When working with time-series data, convolution between the input data and a specific kernel generates a new temporal sequence that evaluates some features of the input data. Shifting in time the input will result in the identical representation appearing in the output, but shifted.

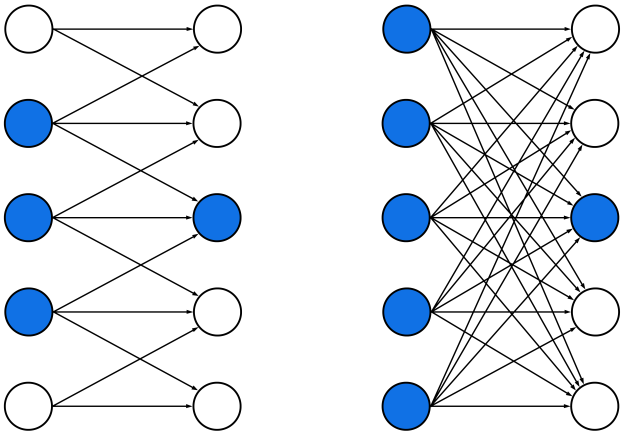

Figure 2.6: Sparse connectivity feature of the convolutional layers. It is highlight in blue one output neuron, and all the input neurons affecting the output one. When the layer is convolutional(left), with in this case kernel of width three, only three inputs affect the highlighted output neuron. On the other hand, if the layer is fully connected(right), all the inputs affects the highlighted output, hence the connectivity is no more *sparse*.

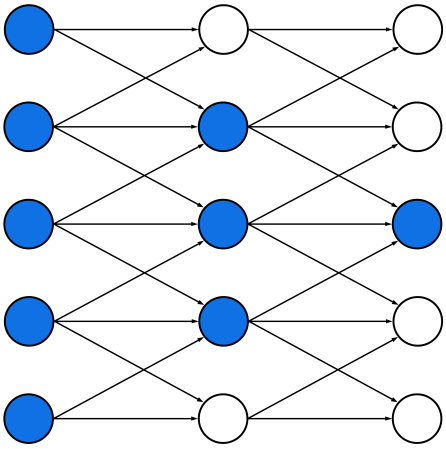

Figure 2.7: The receptive field of the neurons in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. Even if the *direct* connection in a convolutional layer are very sparse, the neurons in the deeper layers are can be *indirectly* connected to all or most of the neurons of the networks. The effect increases if strided convolution or max pooling are applied

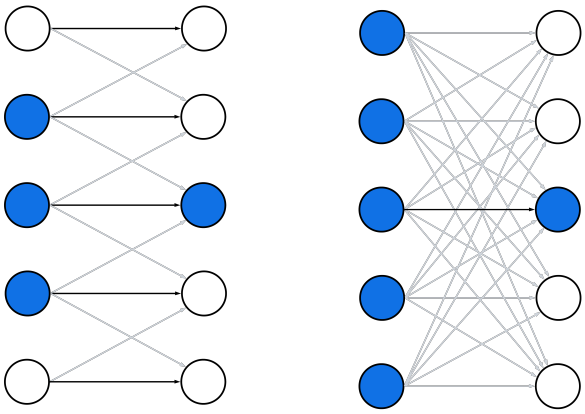

Figure 2.8: Parameters sharing feature. Black arrows indicate the connections that make use of a particular parameter. (Left) The black arrows indicate uses of the central element of a 3-element kernel in a convolutional layer. Because of parameter sharing, this single parameter is used at all input locations. (Right) The single black arrow indicates the use of the central element of the weight matrix in a fully connected layer. The parameter is used only once, hence there is no *parameter sharing*.

Convolutional neural networks have become a powerful tool for extracting meaningful features from sequential data, such as time series. Their effectiveness, however, relies on a set of hyper-parameters that define the architecture and learning behaviour of the convolutional layer. The tuning of these hyper-parameters enables the layer to capture the relevant essence of the input data, which is then properly elaborated to carry out the specific task of the deep neural network model(classification, regression, encoding,..) .

KERNEL SIZE: the kernel size determines the receptive field of the convolutional layer. This choice is related to the frequency components of the feature to be extracted from the input signal, such as global trends(low frequency) or spikes(high frequency). Larger kernel sizes tend capture broader patterns in the input sequence, while smaller kernel sizes focus on finer details. For example, in the context of financial time series analysis, a small kernel can pinpoint specific spikes or dips in the data, potentially revealing short-term fluctuations or sudden events. However, for longer sequences, these small kernels might miss out on broader trends, which however can be detected by larger kernels. Intuitively, smaller kernels acts as high pass filters and extract high frequency input signal components, while longer kernels are more suited for low pass filtering the input signal, hence extracting low frequency components.

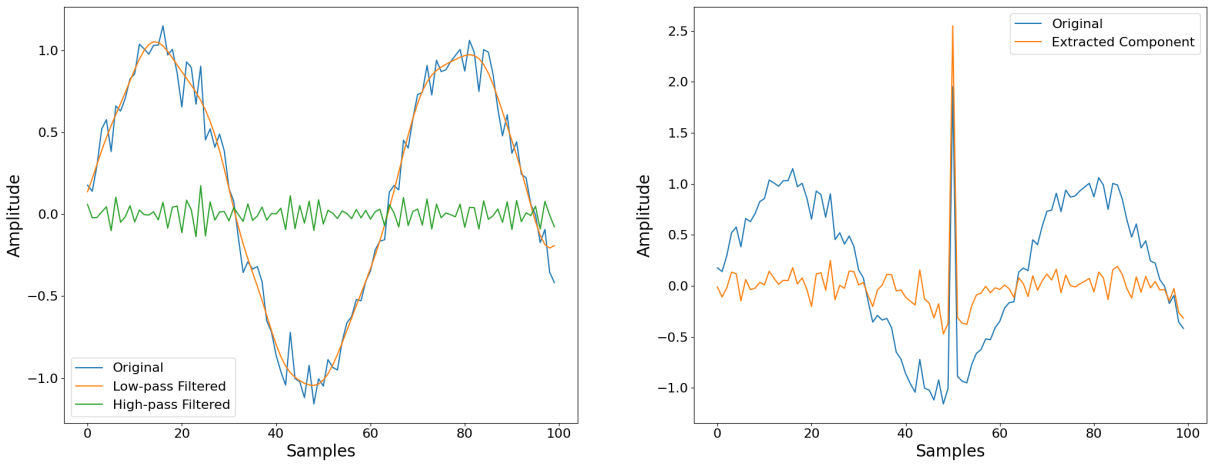

Figure 2.9: Convolutional operation filtering capabilities. (Left) Two convolutional operations are implemented for filtering respectively the low(orange) and high(green) frequency component of an input signal(blue). Those filters can also be applied to capture specific input signal feature, such as peaks. (Right) the convolutional filter is used to extract the peak feature(orange) from the input signal(blue).

PADDING: padding refers to the technique of adding additional values around the input data. Padding is commonly used for two main purposes: allowing to perform the convolution operation also at the edges of the signal and maintaining the original input size at the output. When convolving an input signal of M samples with a convolutional filter(kernel) of size N, the filter does not slide along the first and last $\lfloor N/2 \rfloor$ input signal edge samples and the output of the convolution operation has length of $M - 2\lfloor N/2 \rfloor$. Most commonly used padding strategies are zero-padding(adding zeros), replication padding(repeat the edges of the input data) and reflection padding(mirror the repeated edges of the input data).

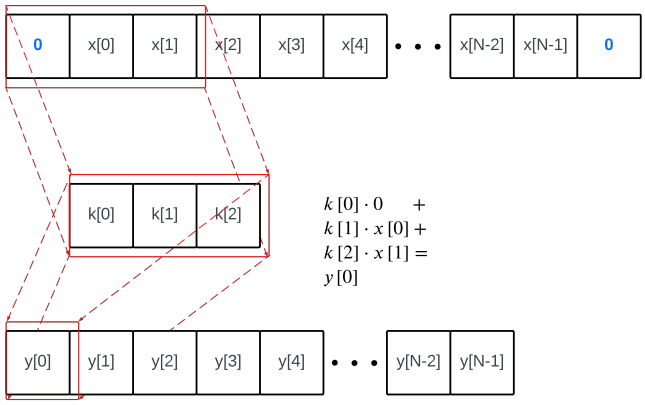

Figure 2.10: A zero padding is applied to the input signal to keep the dimensionality of the convolution output equal to the the input one

STRIDES: strides determine the step size of the moving kernel along the input sequence. A stride equal to one makes the kernel sliding one sample at a time across the entire input sequence,

which leads to capturing all the possible overlapping features and preserving the original spatial resolution of the data. A stride grater then one makes the kernel taking bigger leaps, jumping by a certain number of positions after each convolution operation. This reduces the size of the output, which can be useful for dimensionality reduction tasks. For example, in the context of an autoencoder neural network, a stride grater then one can be an alternative to max pooling for decoding the input signal into a lower dimensional space.

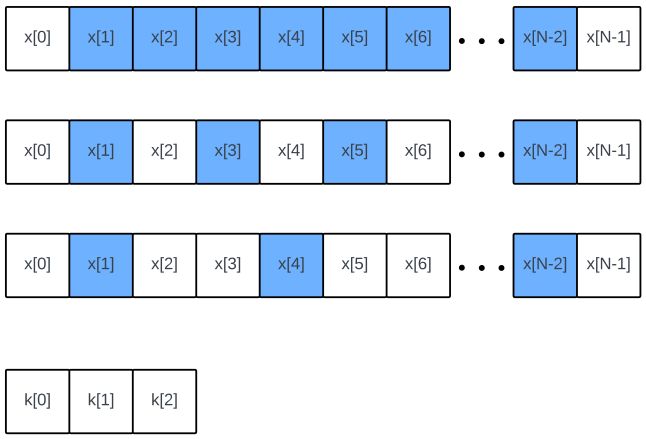

Figure 2.11: Strides. The figure shows three input signals, where a convolutional kernel of width equal to three is centred in the samples highlighted in blue colour. Normally the convolutional kernel slides thru the input signal one sample at the time(top). By applying strides larger than one, the convolution is applied every two(middle), three(bottom) or more samples. This results in reducing the dimensionality of the convolution output by half, one third and so on.

FILTER NUMBER: a convolutional layer usually applies multiple filters(kernels) to its input. The *number of filters* in a convolutional layer defines the number of output channels produced by each convolutional operation, also known as the layer *depth*. Each filter(kernel) within a convolutional layer acts as a feature extractor, learning to identify specific features from the input data. Increasing the number of filters allows the network to capture more diverse and complex patterns. However, this also increases the number of parameters to be learned, which leads to an higher computational cost and a risk of overfitting. The filter number should be chosen based on the complexity of the data, the amount of data and the available computational resources. As an example, a large amount of complex data would benefit of a larger number of filters, if proper computational resources can be used. In a 1D convolutional layer, however, the learned filters can also be two dimensional. Consider the case of two 1D convolutional layers connected to each other with respectively M and N filters. Suppose that for the second layer a kernel of size K is selected. For each one of the N filters of the second layer, a convolution between a kernel of size K and each output of filters of the previous layer is applied simultaneously. This is equivalent of vertically aligning the previous layer filter outputs and convolve the resulting matrix with a filter of size $M \cdot K$.

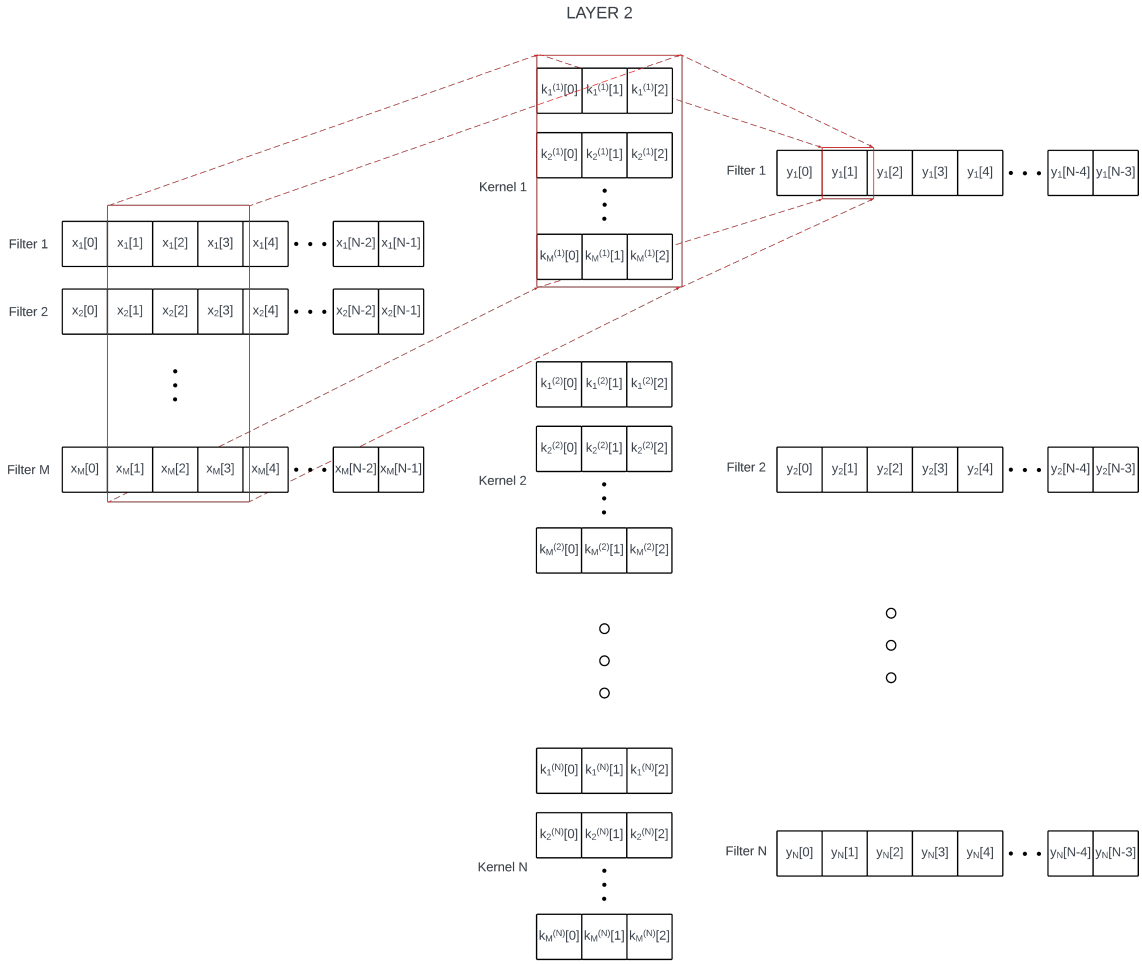

Figure 2.12: Filter number. Layer 2 in figure is a convolutional layer which receives as input the output of a previous convolutional layer with filter size equal to M. Layer 2 carries out a convolutional operation with a kernel of size 3 among all the input filters, hence the convolutional operation becomes bi-dimensional and the actual kernel size is $3 \times M$. Since layer 2 has N filters, during the network training, it learns the parameters of N kernels of size $3 \times M$, and returns N convolution output.

TRANSPOSED CONVOLUTIONAL LAYERS: transposed convolutional layers, also called deconvolutional layers, are a slight variation of the convolutional layer. The operation of deconvolution is mathematically considered as the inverse of the convolution transformation. Since the kernel parameters are learned during the training phase, the transpose convolutional layers do not apply directly a mathematical deconvolution, since there is not in general a specified convolution for which to find the inverse. The main difference between convolutional and transpose convolutional layers is in how the stride operation is applied. For this case, it consists in adding one or multiple zeros between the samples of the layer input. In this way, when applying the convolution with a specified kernel, the output will increase in size with respect to the input. This turns out to be useful in a variety of applications, such as image super resolution, semantic segmentation, GANs, autoencoders and variational autoencoders. For example, if a

stride of 2 and a padding of 1 is chosen for a transposed convolutional layer of kernel size 3, then the output signal is going to have double the samples of the input one(fig. 2.13)

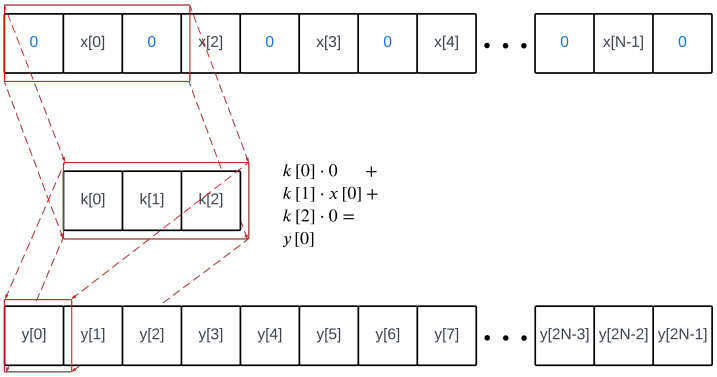

Figure 2.13: The transposed convolutional layer has the same working principle of the convolutional one, with just a change in the application of strides. A stride equal to two, applied to the input signal, adds zero values between the input signal samples, making the output signal have double the samples of the input one

2.6 Variational Autoencoder

Variational Autoencoders (VAEs) are deep learning generative models explicitly designed to capture the underlying probability distribution of a given dataset and generate novel samples. They leverage on an architecture that comprises an encoder-decoder structure, both implemented by artificial neural networks. The encoder transforms input data into a latent form, and the decoder aims to reconstruct the original data based on the latent representation. The VAE is programmed to minimise the dissimilarity between the original and reconstructed data, in a manner that enables it to comprehend the underlying data distribution and generate new samples that are conform to it. The main advantage is their ability to generate new data samples resembling the training data. In a vanilla VAE the latent space is continuous, so the decoder can generate new data points that seamlessly interpolate among the training data. The main theoretical building blocks that compose a VAE model are now introduced.

KL divergence

The Kullback-Leiber (KL) divergence is a type of statistical distance that measures the difference between two probability distributions p_θ and q_ϕ, even though it is not a metric in the proper sense, since it is not symmetric. In probabilistic terms it is the expected value of the difference between the two probability distributions with respect to one of those two probabilities. The

forward and *reverse* KL divergence are respectively defined as follows

$$\begin{aligned} D_{KL}(p_\theta \,||\, q_\phi) &= \mathbb{E}_{x\sim p_\theta}[\log p_\theta(x) - \log q_\phi(x)] \\ &= \mathbb{E}_{x\sim p_\theta}\left[\log \frac{p_\theta(x)}{q_\phi(x)}\right] \end{aligned}$$

$$D^{(r)}_{KL}(p_\theta \,||\, q_\phi) = \mathbb{E}_{x\sim q_\phi}\left[\log \frac{p_\theta(x)}{q_\phi(x)}\right]$$

Useful properties:

- Non symmetric: the forward and reverse KL divergence are in general not equal

$$D^{(f)}_{KL}(p_\theta \,||\, q_\phi) \neq D^{(r)}_{KL}(p_\theta, q_\phi)$$

- Positive semidefinite: the KL divergence is positive semidefinite

$$D_{KL}(p_\theta \,||\, q_\phi) \geq 0$$

Dim.

$$\begin{aligned} D_{KL}(p_\theta \,||\, q_\phi) &= \mathbb{E}_{x\sim p_\theta}\left[\log \frac{p_\theta(x)}{q_\phi(x)}\right] \\ &= -\,\mathbb{E}_{x\sim p_\theta}\left[\log \frac{q_\phi(x)}{p_\theta(x)}\right] \\ &\geq -\log \mathbb{E}_{x\sim p_\theta}\left[\frac{q_\phi(x)}{p_\theta(x)}\right] \\ &= -\log \int_{-\infty}^{+\infty} \cancel{p_\theta(x)}\, \frac{q_\phi(x)}{\cancel{p_\theta(x)}}\, dx \\ &= -log(1) \\ &= 0 \end{aligned}$$

Evidence Lower Bound (ELBO)

It is considered the case where $p_\theta(z|x)$ is unknown and $q_\phi(z|x)$ is known and chosen between known probability distributions(e.g. gaussian). The aim is to estimate the parameters $\hat{\phi}$ of the chosen distribution q_ϕ that better approximate $p_\theta(z|x)$. The KL divergence can be used as the loss function of such optimisation task

$$D_{KL}(q_\phi \,||\, p_\theta) = \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right]$$

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}\ D_{KL}(q_\phi \,||\, p_\theta)$$

The problem with this approach is that $p_\theta(z|x)$ has an unknown analytic expression, so the minimisation is intractable. By reshaping the KL divergence term it is however possible to make the problem feasible indirectly.

$$\begin{aligned}
D_{KL}(q_\phi \,||\, p_\theta) &= \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right] \\
&= \mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(z|x)\right] \\
&= \mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{p_\theta(z,x)}{p_\theta(x)}\right] \\
&= \mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(z,x)\right] + \log p_\theta(x)
\end{aligned}$$

Rearranging the terms it is obtained

$$\log p_\theta(x) = -\mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] + \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(z,x)\right] + D_{KL}(q_\phi \,||\, p_\theta)$$

Since the KL divergence term, as previously demonstrated, is positive semidefinite $D_{KL}(q_\phi \,||\, p_\theta) \geq 0$ it can be stated that

$$\log p_\theta(x) \geq -\mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] + \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(z,x)\right]$$

The right hand side is called *Evidence Lower Bound* (ELBO) and, in order for the previous equality to hold true, maximising it leads to indirectly minimise the KL divergence term

$$\begin{aligned}
ELBO &= -\mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] + \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(z,x)\right] \\
&= -\mathbb{E}_{z|x\sim q_\phi}\left[\log q_\phi(z|x)\right] + \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x|z)\right] + \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(z)\right] \\
&= \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x|z)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right]
\end{aligned}$$

The first term is the *expected reconstruction loss* while the second one is the KL divergence between the *approximated posterior* and the chosen *prior* probabilities.

Reparameterization Trick

In the variational autoencoders, neural networks are used as probabilistic encoders $q_\phi(z|x)$ and decoders $p_\theta(z|x)$. There are many possible choices of encoders and decoders, depending on the type of data and model. The neural networks weighs represent the parameters θ and ϕ of the respective probability distributions. The neural network encoder/decoder structure uses ELBO

as *loss function* and is trained with gradient back-propagation techniques, such as SGD, AGRAD or ADAM.

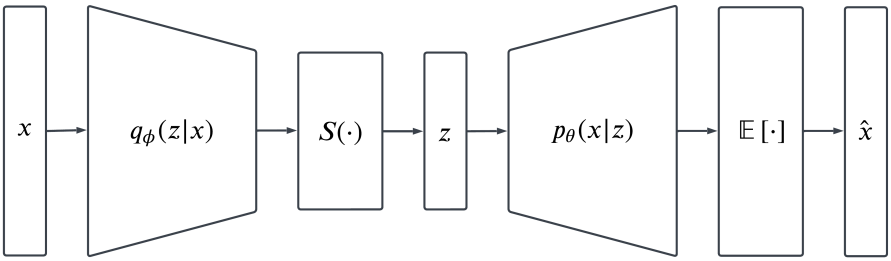

Figure 2.14: VAE theoretical model

First it is reshaped the loss ELBO function

$$\begin{aligned}\mathcal{L}_{\theta,\phi}(x) &= \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x|z)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right] \\ &= \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right]\end{aligned}$$

calculating the gradient for the discrete probability distribution case presents no issues, thanks to the linearity rule of the derivative. However, the continuous case is not that simple, so it is now treated more in detail. Thanks to the Leibniz integral rule the gradient with respect to θ can be easily obtained as

$$\begin{aligned}\nabla_\theta \mathcal{L}_{\theta,\phi}(x) &= \nabla_\theta\, \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right] \\ &= \mathbb{E}_{z|x\sim q_\phi}\left[\nabla_\theta(\log p_\theta(x,z) - \log q_\phi(z|x))\right] \\ &= \mathbb{E}_{z|x\sim q_\phi}\left[\nabla_\theta \log p_\theta(x,z)\right]\end{aligned}$$

it is not possible to apply the same rule when deriving with respect to ϕ since the support of the continuous expectation integrals is a function of it. The reparameterization trick consists in applying a change of variable to the continuous random variable $z|x \sim q_\phi(z|x)$, in order to express this probability distribution as a deterministic function g of the parameters ϕ, a random variable that follows a properly chosen *base distribution* $\epsilon \sim p(\epsilon)$ and x. This allows to indirectly apply the Leibniz integral rule. The "trick" can be applied to various known continuous probability distributions(e.g. gaussian).

$$z = g_\phi(\epsilon, x) \qquad where \quad \epsilon \sim p(\epsilon)$$

Finally, thanks to the Law of the Unconscious Statistician, the unbiased gradient expression can be obtained

$$\begin{aligned}\nabla_\phi \mathcal{L}_{\theta,\phi}(x) &= \nabla_\phi \, \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right] \\ &= \nabla_\phi \, \mathbb{E}_{\epsilon\sim p(\epsilon)}\left[\log p_\theta(x, g_\phi(\epsilon,x)) - \log q_\phi(g_\phi(\epsilon,x)|x)\right] \\ &= \mathbb{E}_{\epsilon\sim p(\epsilon)}\left[\nabla_\phi(\ \log p_\theta(x, g_\phi(\epsilon,x)) - \log q_\phi(g_\phi(\epsilon,x)|x)\ )\right]\end{aligned}$$

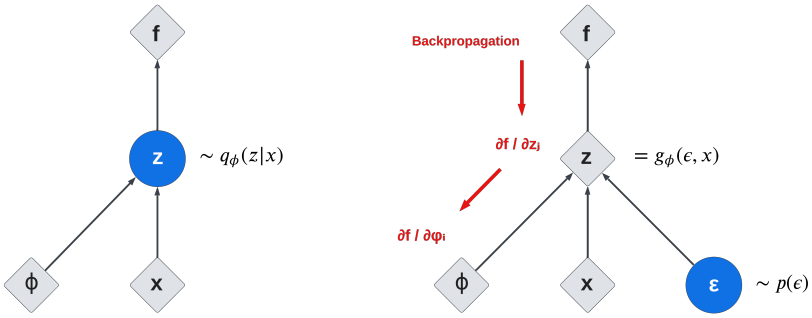

Figure 2.15: Reparemeterization trick

Gaussian case

The gaussian case considers the prior $p_\theta(z)$ as a standard normal distribution, while $p_\theta(x|z)$ and $q_\phi(z|x)$ are gaussian distributions whose parameters are mean and diagonal covariance matrix, which are function of respectively θ and ϕ, so given the VAE loss function

$$\mathcal{L}_{\theta,\phi}(x) = \mathbb{E}_{z|x\sim q_\phi}\left[\log p_\theta(x|z)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right]$$

the probability distributions for the gaussian case are defined as

$$\begin{aligned}p_\theta(z) &= \mathcal{N}(\mu = 0,\ \Sigma = 1) \\ p_\theta(x|z) &= \mathcal{N}(\mu_\theta = f_\theta^{(\mu)}(z),\ \sigma_\theta^2 I = f_\theta^{(\sigma)}(z)) \\ q_\phi(z|x) &= \mathcal{N}(\mu_\phi = h_\phi^{(\mu)}(x),\ \sigma_\phi^2 I = h_\phi^{(\sigma)}(x))\end{aligned}$$

For this particular case the KL divergence term of the loss function has the following closed form

$$
\begin{aligned}
D_{KL}(q_\phi(z|x) \,||\, p_\theta(z)) &= \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right] \\
&= \int_{-\infty}^{\infty} \log\left[\frac{q_\phi(z|x)}{p_\theta(z)}\right] q_\phi(z|x)dz \\
&= \int_{-\infty}^{\infty} \log\left[q_\phi(z|x)\right] q_\phi(z|x)dz - \int_{-\infty}^{\infty} \log\left[p_\theta(z))\right] q_\phi(z|x)dz \\
&= \left[-\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(\mu_{\phi,j}^2 + \sigma_{\phi,j}^2)\right] - \\
&- \left[-\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(1 + \log\sigma_{\phi,j}^2)\right] \\
&= \frac{1}{2}\sum_{j=1}^{J}(1 + \log(\sigma_{\phi,j}^2) - \mu_{\phi,j}^2 - \sigma_{\phi,j}^2)
\end{aligned}
$$

where $J = C(z)$ is the dimension of the latent space. The full proof is found in the Appendix. On the other hand, the reconstruction loss term has no analytical closed form and it must be estimated

$$
\mathbb{E}_{z|x\sim q_\phi}[\log p_\theta(x|z)] \simeq \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(x|z^{(l)})
$$

This is a simple Montecarlo estimator, where L is the number of datapoints drawn from the latent space given an input x. This estimator is not feasible in practice during training, so the MSE ("Mean Squared Error") is used instead

$$
\begin{aligned}
\mathbb{E}_{z|x\sim q_\phi}[\log p_\theta(x|z)] &\simeq \frac{1}{L}\sum_{l=1}^{L}\log p_\theta(x|z^{(l)}) \\
&\simeq \log p_\theta(x|z) && (L = 1) \\
&= -\frac{1}{2}\sum_{k=1}^{K}(2\pi\sigma_{\theta,k}^2) - \frac{1}{2}\sum_{k=1}^{K}\frac{(x_k - \mu_{\theta,k})^2}{\sigma_{\theta,k}^2} \\
&\propto -\sum_{k=1}^{K}(x_k - \mu_{\theta,k})^2 && (\hat{x}_k = \mu_{\theta,k}) \\
&= -||x - \hat{x}||_2^2
\end{aligned}
$$

where $K = C(x)$ is the dimension of input x. Finally, given M datapoints, the loss function is estimated as

$$\hat{\mathcal{L}}_{\theta,\phi}(X) = \sum_{i=1}^{M} \left(||x^{(i)} - \hat{x}^{(i)}||_2^2 - \frac{1}{2} \sum_{j=1}^{J} (1 + \log(\sigma_{\phi,j}^{(i)})^2 - (\mu_{\phi,j}^{(i)})^2 - (\sigma_{\phi,j}^{(i)})^2) \right)$$

where σ_ϕ and μ_ϕ are the outputs of the encoder neural network

$$\begin{bmatrix} \log \sigma_{\phi,1}^2 \\ \vdots \\ \log \sigma_{\phi,J}^2 \\ \mu_{\phi,1} \\ \vdots \\ \mu_{\phi,J} \end{bmatrix} = f_\phi(x)$$

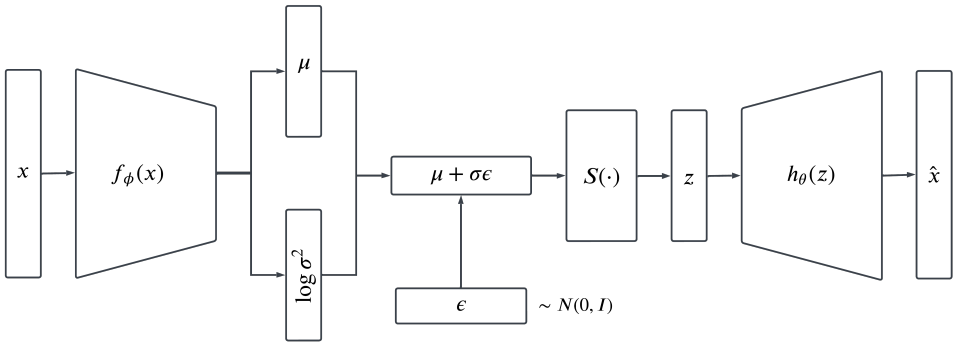

Figure 2.16: VAE model with encoder and decoder implemented by neural networks, gaussian prior, and the sampling procedure carried on using the reparameterization trick

2.7 Numerical optimization algorithm

A mathematical optimization problem, or just optimization problem, has the form of

$$\begin{aligned} &\text{minimize} && f_0(\mathbf{x}) \\ &\text{subject to} && f_i(\mathbf{x}) \leq a_i, \quad i = 1, \ldots, n. \\ &&& h_j(\mathbf{x}) = b_j, \quad j = 1, \ldots, m. \end{aligned}$$

the vector $\mathbf{x} = (x_1, .., x_d)$ is the *optimization variable* of the problem, the function $f_0 : \mathbb{R}^d \to \mathbb{R}$ is the *objective function*, the functions $f_i : \mathbb{R}^d \leftarrow \mathbb{R},\ i = 1, ..., n$ and $h_j : \mathbb{R}^d \leftarrow \mathbb{R},\ j = 1, ..., m$ are respectively the inequality and equality constraint functions, and the constants $a_i, b_j\ \forall i, j$ are the *limits* or *bounds* of the constraints. A vector x^* is called an *optimal solution* of the optimization problem, if it has the smallest objective function value among all the vectors that satisfies the constraints, hence $\forall \mathbf{x} \in \mathbb{R}^d : f_i(\mathbf{x}) \leq a_i \vee h_j(\mathbf{x}) = b_j \forall i = 1, ..., n;\ j = 1, ..., m \Rightarrow f_0(\mathbf{x}) \geq f_0(\mathbf{x}^*)$

There are several families of optimization problems, characterized by particular forms of the objective and constraint functions. As an important example, the optimization problem discussed at the beginnning is called a linear program if the objective and constraint functions $f_0, f_i, h_j\ \forall i, j$ are linear, i.e. satisfy

$$f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y}) \quad \forall\ \mathbf{x}, \mathbf{y} \in \mathbb{R}^d \vee \alpha, \beta \in \mathbb{R}$$

If the optimization problem is not linear, it is called a *nonlinear program.*. Another important family is called *convex optimization problems*, in which the objective and constraint functions are convex, hence they satisfy the following inequality

$$f(\alpha\mathbf{x} + \beta\mathbf{y}) \leq \alpha f(\mathbf{x}) + \beta f(\mathbf{y}) \quad \forall\ \mathbf{x}, \mathbf{y} \in \mathbb{R}^d \vee \alpha, \beta \in \mathbb{R}$$

the convex optimization problem can be considered a generalization of linear programming. A *solution method* refers to an algorithmic approach designed to find a solution to a specific optimization problem within a certain degree of precision. Since the late 1940s, considerable attention has been devoted into crafting algorithms capable of handling different optimization problems. This involves not only creating precise algorithms but also developing suitable software implementations in order to make it computationally affordable. The effectiveness of a solution method relies on numerous factors, including the distinct forms of the objective and constraint functions involved, the quantity of variables, the constraints within the problem and the unique structural attributes it possesses, such as sparsity. A specific class of optimization problems is called *unconstrained minimization problem*, and it is based on finding the optimal domain value for which the objective function has its minimum value regardless any constraints

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\, f(\mathbf{x})$$

The optimal value is denoted as $\inf_{\mathbf{x}} f(\mathbf{x}) = f(\mathbf{x}^*) = p^*$.

From now on, for theoretical background purposes, the problem is assumed to be convex, i.e. there is only one global minimum of the objective function. However, all the following proposed techniques can be applied also to non convex problems. In those cases, however, the proposed algorithm will in general not converge to the global optimum of the objective function, but one of its local ones if it is assumed the objective function to be differentiable and convex, a

necessary and sufficient condition for $\hat{x}$ to be optimal is

$$\mathbf{x}^* \quad \text{s.t.} \quad \nabla_{\mathbf{x}} f(\mathbf{x}^*) = 0$$

In few special cases, such as linear and quadratic programs, there is a closed form analytical solution, but usually the problem must be solved by means of an iterative algorithm. The desired algorithm behaviour is such that it computes a sequence of points called *minimizing sequence* $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots \in \mathbf{dom} f$ with $f(\mathbf{x}^{(k)}) \rightarrow p^*$ as $k \rightarrow \infty$. In practice the algorithm is terminated with an *early stopping*, i.e. when $f(\mathbf{x}^{(k)}) - p^* \leq \epsilon$, where $\epsilon > 0$ is some accepted tolerance value. This approach leads to an estimate of the optimal value $\hat{\mathbf{x}} \simeq \mathbf{x}^*$.
Such iterative class of algorithms require a suitable intialization value $\mathbf{x}^{(0)}$. The initialization point must lie in the domain of the objective function $\mathbf{dom} f$ and the following subset level

$$S = \{\mathbf{x} \in \mathbf{dom} f \ \text{ s.t. } \ f(\mathbf{x}) \leq \mathbf{x}^{(\mathbf{0})}\}$$

must be closed. This condition is satisfied for all $\mathbf{x}^{(0)} \in \mathbf{dom} f$ if the function f is closed itself, i.e. all its sublevel sets are closed. As an example, all the functions with real domain $\mathbf{dom} f \in \mathbf{R}^n$ are closed, hence they satisfy the condition. This family of algorithms produce a minimizing sequence $\mathbf{x}^{(k)}$, $k = 1, 2, \ldots$, such that

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + t^{(k)} \triangle \mathbf{x}^{(k)}$$

with $t^{(k)} > 0$ ($t^{(k)} = 0$ when $\mathbf{x}^{(k)} = \mathbf{x}^*$). Here $\triangle \mathbf{x}$ is a vector in $\mathbf{R}^n$ called *search direction*, $k = 0, 1, \ldots$ denotes the iteration number and the scalar $t^{(k)}$ is called the *step size* or , for machine learning applications, *learning rate*. Ideally, for all the *descent methods* applied to differentiable convex functions, the following condition must be satisfied:

$$f(\mathbf{x}^{(k+1)}) < f(\mathbf{x}^{(k)})$$

except when $\mathbf{x}^{(k)} = \mathbf{x}^*$. This implies that $\forall k \ \ \exists \ \ \mathbf{x}^{(k)} \in S \Rightarrow \mathbf{x}^{(k)} \in \mathbf{dom} f$. Moreover if the cost function is convex $\nabla f(\mathbf{x}^{(k)})^T (\mathbf{y} - \mathbf{x}^{(k)}) \geq 0 \Rightarrow f(\mathbf{y}) \geq f(\mathbf{x}^{(k)})$, so the search direction in a descent method must satisfy the condition

$$\nabla f(x^{(k)})^T \triangle \mathbf{x}^{(k)} < 0$$

a search direction satisfying the condition above is called a *descent direction*. The outline of a general descent algorithm is described here below

Algorithm 2 *General descent method*

Init. $\mathbf{x}^{(0)} \leftarrow 0$
while stopping criterion is satisfied **do**
  Determine $\triangle\mathbf{x}^{(k)}$
  Select $t^{(k)} > 0$ (*line search*)
  $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + t^{(k)}\triangle\mathbf{x}^{(k)}$
end while

A practical stopping criterion is often implemented in the form of —— $\nabla f(\mathbf{x})||_2 = \eta$, where η is positive and 'small'.

2.7.1 Gradient descent

If the search direction is set to be the negative gradient of the objective function, then the descent method is called **Gradient Descent**. The intuitive explanation of choosing the opposite

Algorithm 3 *Gradient descent*

Init. $\mathbf{x}^{(0)} \leftarrow 0$
while stopping criterion is satisfied **do**
  $\triangle\mathbf{x}^{(k)} = -\nabla f(\mathbf{x}^{(k)})$
  Select $t^{(k)} > 0$ (*line search*)
  $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + t^{(k)}\triangle\mathbf{x}^{(k)}$
end while

of the objective function $f(\mathbf{x})$ gradient as the search direction traces back to the first order Taylor approximation centred in $f(\mathbf{x}^{(k)})$. Assuming f is differentiable in $\mathbf{x}^{(k)}$, for any small perturbation $\triangle\mathbf{x}^{(k)}$

$$f(\mathbf{x}^{(k)} + \triangle\mathbf{x}^{(k)}) \simeq f(\mathbf{x}^{(k)}) + \triangle(\mathbf{x}^{(k)})^T\nabla f(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}^{(k)}}$$

now it is explicit that in order to get maximum leverage out, $\triangle\mathbf{x}$ must be align along $-\nabla f(\mathbf{x}^{(k)})$. By multiplying with a positive value $t^{(k)}$ the following results still holds

$$f(\mathbf{x}^{(k+1)}) = f(\mathbf{x}^{(k)}) - t^{(k)}((\nabla f(\mathbf{x}^{(k)})^T\nabla f(\mathbf{x}^{(k)}) < f(\mathbf{x}^{(k)})$$

The algorithm parameter $t^{(k)}$ plays an important role in the optimization process. Intuitively, since the first-order approximation is good only for small $\triangle\mathbf{x}$, it is preferred to choose it in order to make the search direction small. Additionally, a high learning rate leads to "overshooting" past the local minima point and may even lead to the algorithm diverging. On the other hand, a small learning rate increases the time the algorithm takes to converge. Choosing a right balance between the two is essential in order to get a stable and efficient algorithm.

2.7.2 Momentum-based GD

Polyak GD

Heavy-ball method, which is also referenced as *momentum* in deep learning, was first proposed by Polyak [23] and is a modification of vanilla gradient descent. The main idea is to move in a direction given by a linear combination of past gradients in each step of the algorithm. The main purpose was to make the algorithm converge faster and still in a stable way, this was shown to be more effective the GD for quadratics problems, but was not able to generalize well to other kinds of objective function. The algorithm leverages on the current position in the function space x^k, and the "momentum" u^k, which is a linear combination of the past gradients. The update rule is the following:

Algorithm 4 *Polyak GD*

Init. $\mathbf{x}^{(0)} \leftarrow 0$
while stopping criterion is satisfied **do**
  Select $\beta^{(k)}, t^{(k)} > 0$
  $\mathbf{u}^{(k+1)} \leftarrow \mathbf{u}^{(k)} - \beta^{(k)} \nabla f(\mathbf{x}^{(k)})$
  $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + t^{(k)} u^{(k)}$
end while

The scalar hyperparameters $\beta^{(k)}$ and $t^{(k)}$ must be properly tuned, usually based on the bounds of the condition number of the objective function Hessian matrix.

Nesterov GD

Leveraging the idea of momentum introduced by Polyak, Nesterov [24] introduced a sligh modification to the update rule that has been shown to converge well not only for quadratic functions, but for general convex functions. While Polyak GD makes the gradient step first and then adds the momentum, Nesterov GD can be thought of as adding the momentum first and evaluating the gradient step from the new point after.

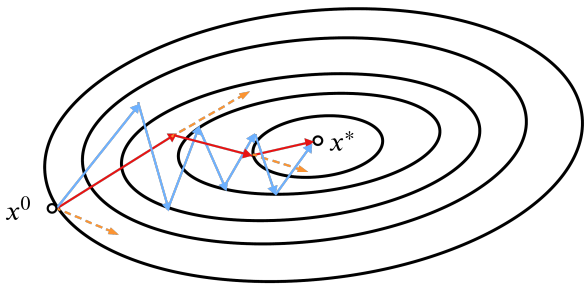

Figure 2.17: Paths followed by GD(blue) and Momentum GD(red) for reaching the function minimum. In orange the momentum component at each iteration

Algorithm 5 *Nesterov GD*

Init. $\mathbf{x}^{(0)} \leftarrow 0$
while stopping criterion is satisfied **do**
  Select $\beta^{(k)}, t^{(k)} > 0$
  $\mathbf{u}^{(k+1)} \leftarrow \mathbf{x}^{(k)} - \beta^{(k)} \nabla f(\mathbf{x}^{(k)})$
  $\mathbf{x}^{(k+1)} \leftarrow \mathbf{u}^{(k)} + t^{(k)}(u^{(k+1)} - u^{(k)})$
end while

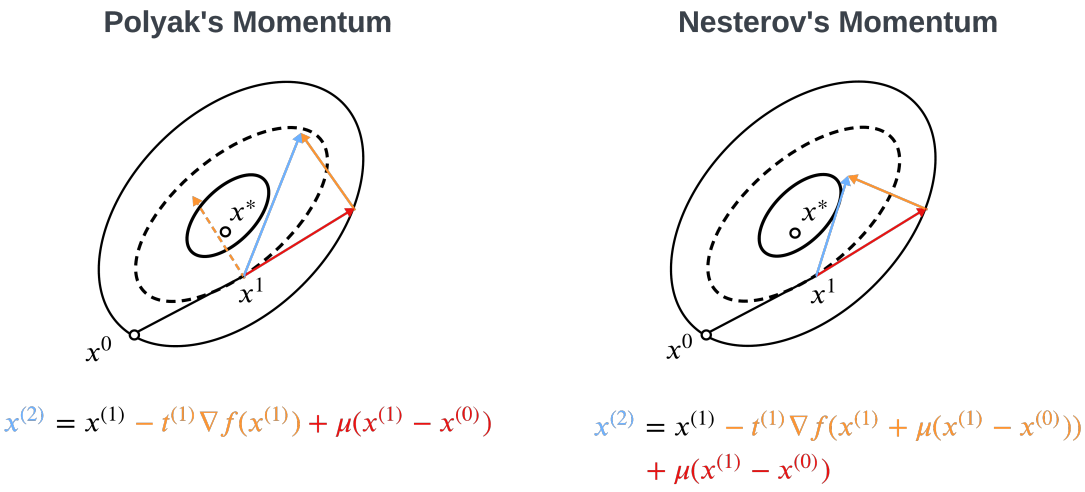

Figure 2.18: Polyak versus Nesterov momentum update step

2.7.3 SGD and minibatch

Gradient Descent algorithms are not subjected to training variance since at every step it is computed the average gradient using the whole dataset. The downside is that every step is very computationally expensive. The complexity of each iteration is $O(nd)$, where n is the number of samples in our dataset and d is the number of dimensions of $\mathbf{x}$. This becomes impractical when dealing with very large dataset. The stochastic gradient descent algorithm (SGD) does not update the direction based on all the dataset points, but at each iteration it randomly picks just one of them. The updated direction is not the exact gradient, like in GD, but a random vector $\mathbf{v}_k$ which is required to have expected value equal to the gradient. In this way, the computation complexity is reduced to $O(d)$ for each iteration. Let $f(\boldsymbol{\theta}) : \mathbb{R}^n \rightarrow \mathbb{R}$ be a stochastic scalar. The aim of SGD is to minimize $\mathbb{E}[f(\boldsymbol{\theta})]$ w.r.t. $\boldsymbol{\theta}$. The functions $f^{(1)}(\boldsymbol{\theta}), f^{(2)}(\boldsymbol{\theta}), ..., f^{(K)}(\boldsymbol{\theta})$ denote the realization of the stochastic function at each iteration. The minibatch variation considers, at each iteration, a subset $S^{(k)} = [1, m]$ of the m available data samples. Note that if $S^{(k)} = 1$ then the minibatch SGD is equal to the SGD and if $S^{(k)} = m$ then it is equal to the GD.

Algorithm 6 *Stochastic gradient descent*

Init. $\boldsymbol{\theta}^{(0)} \leftarrow 0$
while stopping criterion is satisfied **do**
  Select $t^{(k)} > 0$
  Choose $\mathbf{v}_k$ at random from a distribution s.t. $\mathbb{E}[\mathbf{v}_k|\boldsymbol{\theta}^{(k)}] = \nabla f(\boldsymbol{\theta}^{(k)})$
  $\boldsymbol{\theta}^{(k+1)} \leftarrow \boldsymbol{\theta}^{(k)} - t^{(k)}\mathbf{v}_k$
  $\hat{\boldsymbol{\theta}} \leftarrow \frac{1}{k+1}\sum_{i=1}^{k+1}\boldsymbol{\theta}^{(i)}$
end while

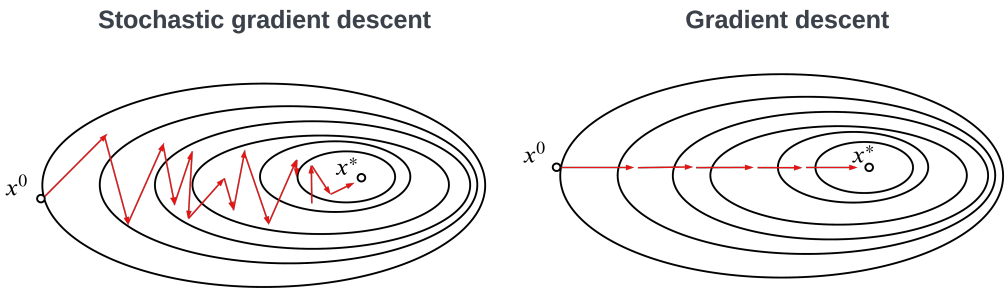

Figure 2.19: Path towards the function minimum followed by GD(left) and SGD(right)

2.7.4 ADAM

ADAM("Adaptive Moment Estimation") is an efficient stochastic optimization method that only requires first order gradients with little memory requirements. It combines ideas from both AdaGrad and Momentum GD, indeed its key features include momentum-based updating of gradients and adaptive scaling of learning rates for each parameter. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Let $f(\boldsymbol{\theta})$ be a stochastic scalar function with parameters $\boldsymbol{\theta}$. The aim is to minimize $\mathbb{E}[f(\boldsymbol{\theta})]$ w.r.t. $\boldsymbol{\theta}$ and $f^{(1)}(\boldsymbol{\theta}), f^{(2)}(\boldsymbol{\theta}), ..., f^{(K)}(\boldsymbol{\theta})$ denote the realization of the stochastic function at each iteration. The stochasticity might come from the evaluation at random subsamples (mini-batches) of datapoints, or arise from inherent function noise. The algorithm updates exponential moving averages of the gradient $m^{(k)}$ and the squared gradient $v^{(k)}$ where the hyper-parameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages. Since the moving averages are initialized to zero this leads to moment estimates biased towards zero. Anyway this initialization bias can be easily counteracted, resulting in bias-corrected estimates $\hat{m}^{(k)}$ and $\hat{v}^{(k)}$. The gradient of the objective function is denoted by $\mathbf{g}^{(k)} = \nabla_{\boldsymbol{\theta}} f^{(k)}(\boldsymbol{\theta})$, while $(g^{(k)})^2 = \mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$ and β_1^k and β_2^k denote β values to the power of k. All vector operations are element-wise. Good default settings of the hyperparameters proposed by the paper [27] are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$

Algorithm 7 *ADAM*

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\boldsymbol{\theta})$: Stochastic objective function with parameters $\boldsymbol{\theta}$
Require: $\boldsymbol{\theta}_0$: Initial parameter vector
$\mathbf{m}^{(0)} \leftarrow 0$ (Initialize 1st moment vector)
$\mathbf{v}^{(0)} \leftarrow 0$ (Initialize 2nd moment vector)
$k \leftarrow 0$ (Initialize timestep)
while stopping criterion is satisfied **do**
$k \leftarrow k + 1$
$\mathbf{g}^{(k)} \leftarrow \nabla_{\boldsymbol{\theta}} f^{(k)}(\boldsymbol{\theta}^{(k-1)})$ (Get gradients w.r.t. stochastic objective at timestep k)
$\mathbf{m}^{(k)} \leftarrow \beta_1 \cdot \mathbf{m}^{(k-1)} + (1 - \beta_1) \cdot \mathbf{g}^{(k)}$ (Update biased first moment estimate)
$\mathbf{v}^{(k)} \leftarrow \beta_2 \cdot \mathbf{v}^{(k-1)} + (1-\beta_2) \cdot (\mathbf{g}^{(k)})^2$ (Update biased second raw moment estimate)
$\hat{\mathbf{m}}^{(k)} \leftarrow \mathbf{m}^{(k)} / (1 - \beta_1^k)$ (Compute bias-corrected first moment estimate)
$\hat{\mathbf{v}}^{(k)} \leftarrow \mathbf{v}^{(k)} / (1 - \beta_2^k)$ (Compute bias-corrected second raw moment estimate)
$\boldsymbol{\theta}^{(k)} \leftarrow \boldsymbol{\theta}^{(k-1)} - \alpha \cdot \hat{\mathbf{m}}^{(k)} / (\sqrt{\hat{\mathbf{v}}^{(k)}} + \epsilon)$ (Update parameters)
end while
return $\boldsymbol{\theta}^{(k)}$ (Resulting parameters)

2.8 Backpropagation algorithm

Deep neural networks are trained using a particular kind of algorithm called *back-propagation*, which is an efficient algorithm based on the *chain rule*, with the purpose of computing the gradient update step in a computationally more efficient way for specifically the DNN model topology. The chain rules states that given a differentiable function $g(x)$ at some point x^* and f a differentiable function at $g(x^*)$, then the composite function $f \circ g$ is differentiable in x^* and its derivative is equal to

$$(f \circ g)(x^*) = f'(g(x^*))g'(x^*)$$

The concept is generalized to multiple composite functions using the Leibniz notation

$$\frac{df_1}{dx}(x^*) = \frac{df_1}{df_2}(f_2(x^*)) \cdot \frac{df_2}{df_3}(f_3(x^*)) \cdot \ldots \cdot \frac{df_n}{dx}(x^*)$$

Next, it is going to be considered a generic example of a multilayer feedforward fully connected neural networks composed by L layers, multiple inputs and outputs, to be trained using the MSE loss function and M training data samples. The l-th layer is supposed to have J outputs(neurons) and I inputs. The output of a dense layer, without considering the activation function, is the

following

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} = \begin{cases} y_1 = w_{11}x_1 + w_{12}x_2 + \cdots + w_{1I}x_I + b_1 \\ y_2 = w_{21}x_1 + w_{22}x_2 + \cdots + w_{2I}x_I + b_2 \\ \quad \vdots \\ y_J = w_{J1}x_1 + w_{J2}x_2 + \cdots + w_{JI}x_I + b_J \end{cases}$$

Given E the error back propagated to the l-th dense layer we are interested on calculating the gradient of this with respect to the trainable parameters $\mathbf{W}$ and $\mathbf{b}$ and the input $\mathbf{x}$, to be passed to the previous dense layer. In order to do so, the gradient of the error is first calculated with respect to the layer outputs $\mathbf{y}$ and then it is applied the chain rule

$$\nabla_{\mathbf{y}} E = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_J} \end{bmatrix} \quad \nabla_{\mathbf{W}} E = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{12}} & \cdots & \frac{\partial E}{\partial w_{1I}} \\ \frac{\partial E}{\partial w_{21}} & \frac{\partial E}{\partial w_{22}} & \cdots & \frac{\partial E}{\partial w_{2I}} \\ & & \vdots & \\ \frac{\partial E}{\partial w_{J1}} & \frac{\partial E}{\partial w_{J2}} & \cdots & \frac{\partial E}{\partial w_{JI}} \end{bmatrix} \quad \nabla_{\mathbf{b}} E = \begin{bmatrix} \frac{\partial E}{\partial b_1} \\ \frac{\partial E}{\partial b_2} \\ \vdots \\ \frac{\partial E}{\partial b_J} \end{bmatrix} \quad \nabla_{\mathbf{x}} E = \begin{bmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial x_2} \\ \vdots \\ \frac{\partial E}{\partial x_J} \end{bmatrix}$$

The expression $\nabla_{\mathbf{W}} E$ is an abuse of notation, it represent the gradient of E over all the weights w_{ij} associated to the layer. Using the chain rule the gradients can be easily evaluated

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i} x_i \quad \forall i, j \qquad \rightarrow \qquad \nabla_{\mathbf{W}} E = \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \frac{\partial E}{\partial y_1} x_2 & \cdots & \frac{\partial E}{\partial y_1} x_I \\ \frac{\partial E}{\partial y_2} x_1 & \frac{\partial E}{\partial y_2} x_2 & \cdots & \frac{\partial E}{\partial y_2} x_I \\ & \vdots & & \\ \frac{\partial E}{\partial y_J} x_1 & \frac{\partial E}{\partial y_J} x_2 & \cdots & \frac{\partial E}{\partial y_J} x_I \end{bmatrix} = \nabla_{\mathbf{y}} E \cdot \mathbf{x}^T$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j} \quad \forall j \qquad \rightarrow \qquad \nabla_{\mathbf{b}} E = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_J} \end{bmatrix} = \nabla_{\mathbf{y}} E$$

$$\frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial y_1} w_{1i} + \cdots + \frac{\partial E}{\partial y_J} w_{Ji} \quad \forall i \qquad \rightarrow \qquad \nabla_{\mathbf{W}} E = \begin{bmatrix} \frac{\partial E}{\partial y_1} w_{11} & \frac{\partial E}{\partial y_2} w_{21} & \cdots & \frac{\partial E}{\partial y_J} w_{J1} \\ \frac{\partial E}{\partial y_1} w_{12} & \frac{\partial E}{\partial y_2} w_{22} & \cdots & \frac{\partial E}{\partial y_J} w_{J2} \\ & \vdots & & \\ \frac{\partial E}{\partial y_1} w_{1I} & \frac{\partial E}{\partial y_2} w_{2I} & \cdots & \frac{\partial E}{\partial y_J} w_{JI} \end{bmatrix} = W^T \cdot \nabla_{\mathbf{y}} E$$

The activation function can be considered a layer itself, with no trainable parameter

$$\mathbf{y} = f(\mathbf{x}) = \begin{cases} y_1 = f(x_1) \\ y_2 = f(x_2) \\ \quad \vdots \\ y_J = f(x_J) \end{cases}$$

Since it has no trainable parameters, the only quantity of interest is the gradient of the received error with respect to the layer inputs.

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j}\frac{\partial f}{\partial x_j} \;\; \forall j \qquad \rightarrow \qquad \nabla_{\mathbf{x}} E = \nabla_{\mathbf{y}} E \odot \nabla_{\mathbf{x}} f$$

The blockchain can be summarised by the block diagram 2.20. At the very end of the network, the error propagating is the loss function used to train the model. In this example, it is considered the MSE loss with M training data samples as the initial back-propagated error

$$\nabla_{\mathbf{y}} E = \nabla_{\mathbf{y}} \frac{1}{M} \sum_{i=1}^{M} ||\mathbf{y}_i - \mathbf{y}_i^*||^2$$

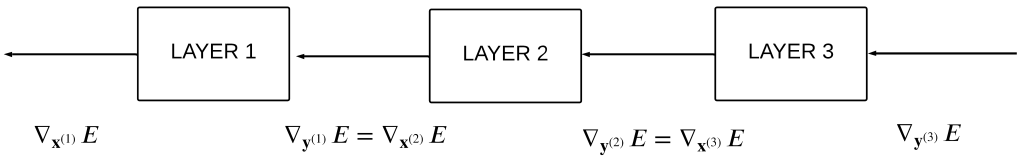

Figure 2.20: Backpropagation of the error through a multi-layer fully connected neural network

2.9 Evaluation metrics

2.9.1 Dynamic Time Warping

Dynamic time warping (DTW) is a dynamic programming technique used to find the optimal alignment between two given time-dependent sequences of possibly different length. Intuitively, the sequences are warped in a non-linear fashion to match each other, which allows to compare and measure the similarity between the two, that may vary in time or speed. The objective of DTW is to compare two (time-dependent) sequences $X = \{x_1, x_2, ..., x_N\}$ and $Y = \{y_1, y_2, ..., y_M\}$ of respectively length of $N, M \in \mathbb{N}$. To compare two different features $x, y \in \mathbb{R}$, one needs a local cost measure, sometimes also referred to as local distance measure inversely proportional to the difference. This can be defined to be a function $c : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+$. Evaluating the local cost measure foreach pair of elements of the sequences X and Y leads to a cost matrix $C \in \mathbb{R}^{N \times M}$ such that $C(n, m) = c(x_n, y_m)$. The goal is to find an alignment between

X and Y having minimal overall cost. An (N, M) *warping path* is a sequence $p = (p_1, .., P_L)$ with $p_l = (n_l, m_l) \in [1 : N] \times [1 : M]$ for $l \in [1 : L]$ satisfying the following three conditions:

- Boundary condition: $p_1 = (1, 1)$ and $p_L = (N, M)$
- Monotonicity condition: $n_1 \leq n_2 \leq ... \leq n_L$ and $m_1 \leq m_2 \leq ... \leq m_L$
- Step size condition: $p_{l+1} - p_l \in \{(1, 0), (0, 1), (1, 1)\}$ for $l \in [1 : L - 1]$

The *total cost* $c_p(X, Y)$ of a warping path p between X and Y with respect to the local cost measure c (such as the squared l_2 norm $c = || \cdot ||_2^2$) is defined as

$$c_p(X, Y) = \sum_{l=1}^{L} c(x_{n_l}, y_{m_l})$$

An *optimal warping path* p^* between X and Y is a warping path having minimal total cost among all possible ones. Finally, the *DTW distance* is defined as the total cost of p^*

$$DTW(X, Y) = c_{p^*}(X, Y) = min\{c_p(X, Y)\}$$

To determine an optimal path p^*, one could test every possible warping path between X and Y. Such a procedure, however, would lead to a computational complexity that is exponential in the lengths N and M. There exists, however, a dynamic programming algorithm that can reduce this complexity to $O(NM)$. To this end, the *prefix sequences* $X(1 : n) = \{x_1, ..., x_n\}$ and $Y(1 : m) = \{y_1, ..., y_m\}$ are defined for $n = [1 : N]$ and $m = [1 : M]$ and set

$$D(n, m) = DTW(X(1 : n), Y(1 : m))$$

The values $D(n, m)$ defines a $N \times M$ matrix defined as *accumulated cost matrix*. The accumulated cost matrix defines the following identities

$$\begin{aligned} D(n, 1) &= \sum_{k=1}^{n} c(x_k, y_1) \\ D(1, m) &= \sum_{k=1}^{m} c(x_1, y_k) \\ D(n, m) &= \min\{D(n-1, m-1), D(n-1, m), D(n, m-1)\} + c(x_n, y_m) \end{aligned}$$

for $n \in [1, N]$ and $m \in [1, M]$. It can be demonstrated that, by using the accumulated cost matrix, $DTW(X, Y) = D(N, M)$ can computed with a complexity of $O(NM)$ by the algorithm 8. In order for the agorithm to work the first row and column of the accumulated cost matrix must be initialized to infinity except for the corner element which is initialized to zero.

Algorithm 8 *Accumulated cost matrix and DTW*

$D(0,0) \leftarrow 0$
$D(n,0) \leftarrow \infty, \quad n \in [1:N]$
$D(0,m) \leftarrow \infty, \quad m \in [1:M]$
for n = 1:N **do**
  for m = 1:M **do**
    $c(n,m) \leftarrow (X(n) - Y(m))^2$
    $D(n,m) \leftarrow c(n,m) + \min\{D(n-1,m), D(n,m-1), D(n-1,m-1)\}$
  end for
end for
$DTW(X,Y) \leftarrow D(N,M)$
return $D, DTW(X,Y)$

Algorithm 9 *Optimal path*

Eval. D (*accumulated cost matrix*)
Init. $p_L^* \leftarrow [N,M]$
Init. $l \leftarrow L$
while $p_l^* \neq [1,1]$ **do**
  $p_{l-1}^* \leftarrow \text{argmin}\{D(n-1,m-1), D(n-1,m), D(n,m-1)\}$
  $l \leftarrow l-1$
end while
$p_1^* = [1,1]$
return p^*

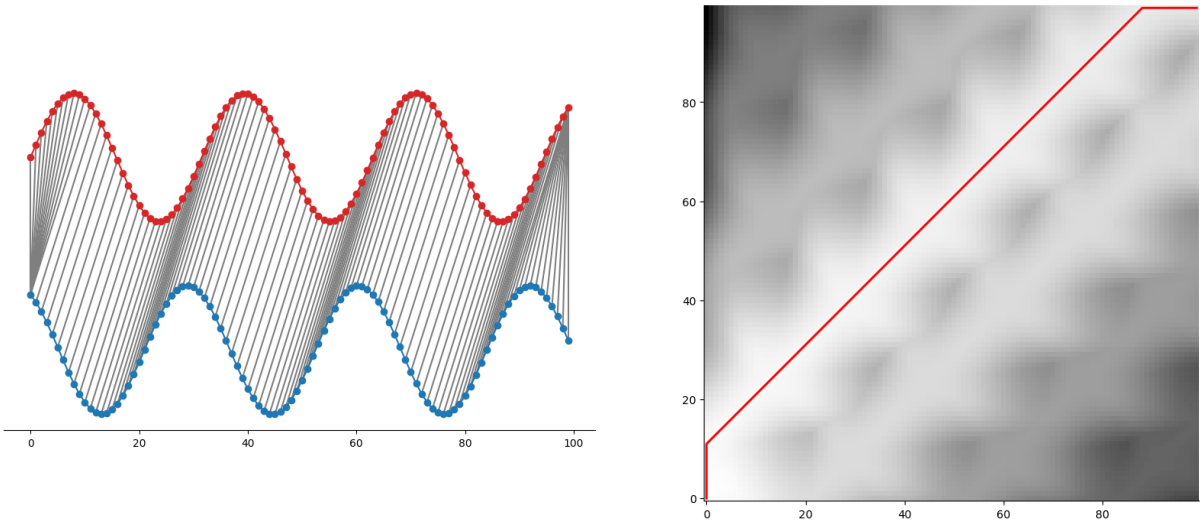

Figure 2.21: Time warping of two shifted sinusoidal time series(left) and the associated cost matrix(right) with the optimal path displayed in red.

2.9.2 Squared Maximum Mean Discrepancy

Maximum mean discrepancy (MMD) is a kernel based statistical test used to measure the level of discrepancy between two probability distributions, which is defined as the distance between their embeddings in a Reproducing Kernel Hilbert Space (RKHS). Given $x \sim p$ and $y \sim q$

$$MMD(p,q) = \sup_{\|\phi\| \in \mathcal{H}} (\mathbb{E}_{x\sim p}[\phi(x)] - \mathbb{E}_{y\sim q}[\phi(y)])$$

By considering $\mu_p = [\mathbb{E}_{x\sim p}\, \phi(x)]$ and $\mu_q = [\mathbb{E}_{y\sim q}\, \phi(y)]$ the MMD square root can be demonstrated to be

$$\begin{aligned} MMD^2(p,q) =& \left\|\mu_p - \mu_q\right\|^2_{\mathcal{H}} \\ &= \langle \mu_p, \mu_p \rangle_{\mathcal{H}} + \langle \mu_q, \mu_q \rangle_{\mathcal{H}} - 2\langle \mu_p, \mu_q \rangle_{\mathcal{H}} \\ &= \mathbb{E}_{x,x'\sim p}\langle \phi(x), \phi(x') \rangle_{\mathcal{H}} + \mathbb{E}_{y,y'\sim q}\langle \phi(y), \phi(y') \rangle_{\mathcal{H}} - 2\,\mathbb{E}_{x\sim p, y\sim q}\langle \phi(x), \phi(y) \rangle_{\mathcal{H}} \end{aligned}$$

By defining the Kernel as $K(x,y) = \langle \phi(x), \phi(y) \rangle$, it leads to

$$MMD^2(p,q) = \mathbb{E}_{x,x'\sim p}\, K(x,x') + \mathbb{E}_{y,y'\sim q}\, K(y,y') + \mathbb{E}_{x\sim p, y\sim q}\, K(x,y)$$

The empirical estimate is obtained by replacing the population expectations with their corresponding U or V-statistics and sample averages, which respectively leads to unbiased and biased estimators

$$MMD_u^2(x,y) = \frac{1}{m(m-1)} \sum_{i=1}^{m} \sum_{j\neq i}^{m} K(x_i, x_j) + \frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j\neq i}^{n} K(y_i, y_j) - \frac{2}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} K(x_i, y_j)$$

$$MMD_b^2(x,y) = \frac{1}{m^2} \sum_{i,j=1}^{m} K(x_i, x_j) + \frac{1}{n^2} \sum_{i,j=1}^{n} K(y_i, y_j) - \frac{2}{mn} \sum_{i,j=1}^{m,n} K(x_i, y_j)$$

The characteristics of the two estimators are the following: the U-statistic is unbiased, close to minimum variance and **not always** ≥ 0. On the other hand, the V-statistic is biased, with minimum variance and **always** ≥ 0. It is used the gaussian Radial Basis Function (RBF) kernel

$$K(x,y) = e^{-\gamma \|x-y\|^2}$$

the bandwidth γ hyper parameter is set to the median pairwise norm-1 distances among the joint kernel input data as suggested in [21]. The dimension of x and y is assumed to be the same ($\#x_i = \#y_i = c$)

$$\gamma = Med(\{d(x_i, y_i) \mid x_i \in x, y_i \in y\})$$

$$d(x_i, y_i) = \sum_{i=1}^{c} |x_i - y_i|$$

Chapter 3

Proposed model

3.1 Tensorflow and Keras

TensorFlow is an open-source machine learning library developed by the Google Brain team. It is above the most popular frameworks used in the field of artificial intelligence and machine learning. It is designed to simplify the creation, training, and deployment of machine learning models, ranging from simple linear regressions to complex neural networks. One of its main features is its versatility. It supports a wide range of machine learning algorithms and can be run on different types of hardware, including CPUs, GPUs, and even TPUs. This flexibility makes TensorFlow suitable for both academic research and large-scale production environments. TensorFlow relies on data flow graphs. In this architectural paradigm, nodes in the graph represent mathematical operations, while the edges represent the data arrays (tensors) that flow between them. TensorFlow provides a large built-in support for deep learning, by implementing a rich set of tools for constructing and training neural networks. This includes pre-built datasets (MNIST, CIFAR-10-100, COCO, ...), models (ResNet, YOLO, GPT, ...), layers (Dense, CNN, RNN, LSTM, ...) and optimizers (SGD, Adagrad, Adam, ...).

Moreover, TensorFlow has an already developed and varied ecosystem. TensorFlow Extended (TFX) is a production-ready machine learning platform that includes tools for model training, serving, and validation. TensorFlow Lite is a lightweight version of the library designed for mobile and embedded devices, which allows developers to run machine learning models on smartphones, IoT devices and other resource-constrained platforms.

One of its strengths is the community support and extensive documentation. The community continuously contributes to the library's development and the documentation is comprehensive, providing detailed explanations of its APIs, modules and functions.

Another significant aspect of TensorFlow is the integration with other Google services and tools. TensorFlow can seamlessly integrate with Google Cloud, allowing users to leverage the power of cloud computing for training and deploying models. Google Cloud AI Platform provides managed services for TensorFlow, enabling users to train models on powerful

cloud-based hardware without worrying about the underlying infrastructure. While Python is the primary language used for TensorFlow, the library also provides APIs for other languages such as C++, Java, and JavaScript. This cross-language support ensures that TensorFlow can adapt to different environments. TensorFlow.js, for example, allows for the development and deployment of machine learning models in the browser, enabling web developers to incorporate AI into their applications without needing server-side processing. Another important example is Tensorflow Lite for Microcontrollers, which is a C++ library which allows to implement Tensorflow into an embedded system. The Tensorflow APIs hierarchy is rapresented in figure 3.1. The high-level API is object-oriented and is called Keras. It is integrated into TensorFlow to offer a straightforward way to define and train models, making it accessible for beginners while still powerful enough for advanced users. It abstracts much of the complexity involved in developing software for machine learning applications, allowing users to focus on the architecture of their models rather than the underlying implementation details. When developing a custom neural network architecture and/or training procedure and there are no high level Keras building blocks to fulfill the task, then some lower level API must be used. Tensorflow provides reusable libraries for common model components which can be used to customize the model and/or training with a higher degree of freedom. Moreover, the core low-level API, allows the user to directly work with the fundamental data structures(Tensor, TensorArray, Variable, ...), primitive API(Shape, Concat, Slicing ...), Numerical(Math, Linalg, Random, ...) and functional components(GradientTape, Function, ...).

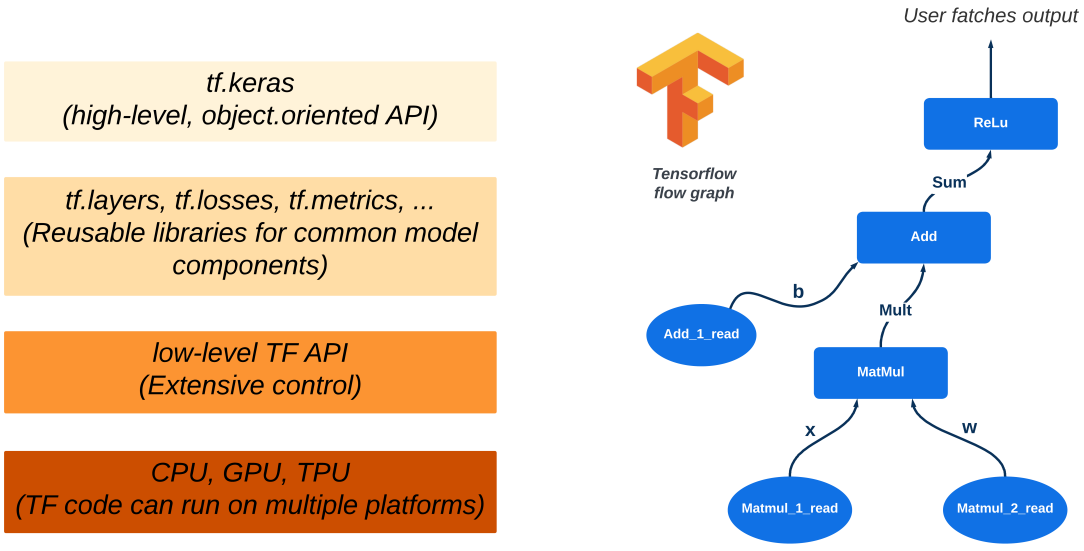

Figure 3.1: TensorFlow APIs hierarchy(left) and the TensorFlow data flow graph of a fully connected layer(right)

3.2 Model architecture

The proposed VAE model is implemented and trained using Python and Tensorflow version 2.16.1, the structure is illustrated in figure 3.2.

The time series signal is provided as input to two one-dimensional convolutional Layers (Conv1D) working in parallel, with respectively kernel size of 11 and 5 and 32 filters for both of them. The convolution is performed by applying strides of length 2, which reduces the dimension of the layers' output feature maps, allowing to gradually encoding a compressed representation of the input. The outputs of the two Conv1D layers are successively concatenated along the filter axis, thus obtaining a total of 64 filters. A cascade of other two Conv1D layers leads to an output of 18 samples with 32 filters. This output is reshaped into a one-dimensional array of 576 samples by a Flatten layer. A Dense layer, composed by 32 neurons, is finally added as penultimate layer to reduce the data size that are finally split into 2-neuron Dense layers. These two final layers represent, respectively, the means and logarithmic variances of the bivariate normal distribution associated to the two-dimensional latent space.

The decoder is symmetric to the encoder. The compressed vector is provided in input to a cascade of two Dense layers of size 32 and 576, respectively. Then the data is reshaped into a structure having size $(18, 32)$. Two one-dimensional transposed convolutional layers (Conv1DT) with reversed strides and 32 filters are used to gradually augment the data size up to $(144, 32)$. The signal is again split into two vectors to feed two different Conv1DT layers with 32 filters. The outputs of these filters are than concatenated and a final Conv1DT with only one filter is applied to obtain the original input size. All layers implement a Rectified Linear Unit (ReLU) activation function, except for the decoder last layer, where a hyperbolic tangent (Tanh) is employed. Tables 3.1 and 3.2 summarize the layers of the two deep-learning networks. The sampling layer, the custom loss function and training procedure are implemented using core low level Tensorflow API.

Table 3.1: Encoder architecture

Layer type	Output shape	Kernel size	Strides	Act. function
Input	(288,1)	N/A	N/A	N/A
Conv1D	(144,32)	11	2	ReLU
Conv1D	(144,32)	5	2	ReLU
"Concat."	(144,64)	N/A	N/A	N/A
Conv1D	(72,32)	11	2	ReLU
Conv1D	(18,32)	9	4	ReLU
Flatten	(576)	N/A	N/A	N/A
Dense	(32)	N/A	N/A	ReLU
Dense	(2)	N/A	N/A	ReLU
Dense	(2)	N/A	N/A	ReLU
"Sampling"	(2)	N/A	N/A	ReLU

Table 3.2: Decoder architecture

Layer type	Output shape	Kernel size	Strides	Act. function
Input	(2)	N/A	N/A	N/A
Dense	(32)	N/A	N/A	ReLU
Dense	(576)	N/A	N/A	ReLU
”Reshape”	(18,32)	N/A	N/A	N/A
Conv1DT	(72,32)	9	4	ReLU
Conv1DT	(144,32)	11	2	ReLU
Conv1DT	(288,32)	13	2	ReLU
Conv1DT	(288,32)	5	2	ReLU
”Concat.”	(288,64)	N/A	N/A	N/A
Conv1DT	(288,1)	13	1	tanh

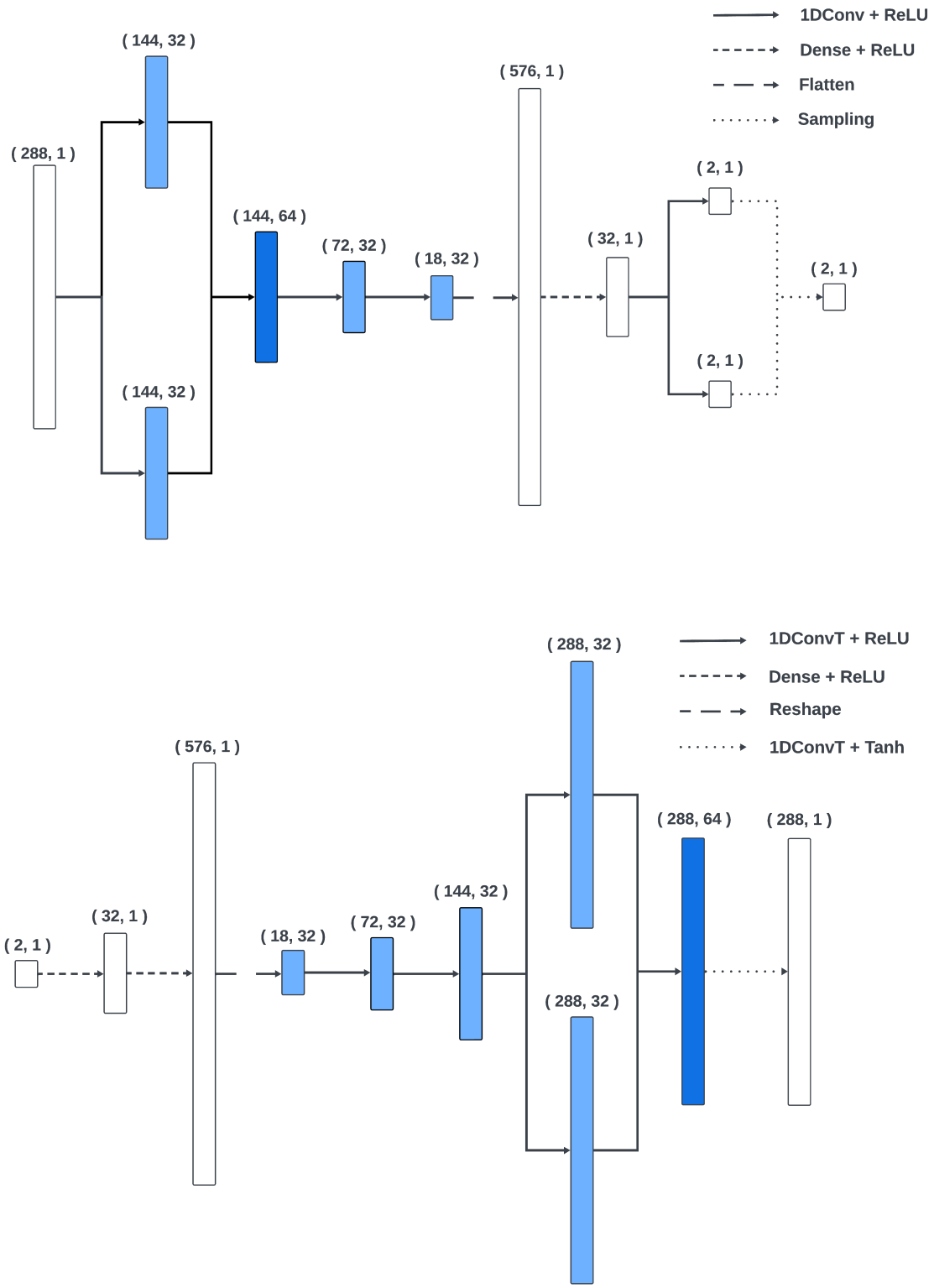

Figure 3.2: Encoder (up) and decoder (bottom) architectures

3.3 Model training

Encoder and decoder are trained jointly: a sampling layer samples from the normal distribution parameterized by the encoder outputs and obtains a two-dimensional vector to be fed into the decoder network, which tries to reconstruct the original signal. The available dataset has very few samples and the aim is to augment their number and diversity. DNN based architectures, given their large amount of parameters, require quite a large amount of data in order not to be underfitted. The required amount of data, however, is unavailable, so the only possible solution is a very "intensive training", which leads to overfitting the model to the small amount of available data, but does no compromise its generative capabilities of augmenting the training data itself. Network parameters are initialized using the Gloroth uniform initializer. The ADAM optimizer is chosen. The training is concluded with an early stopping when there is no relevant improvement of the reconstruction loss after a certain number of epochs. At the end of training phase, the quality of the model is evaluated by considering the Reconstruction Loss $RL = \sum_{i=1}^{B} \|x^{(i)} - \hat{x}^{(i)}\|_2^2$, which corresponds to the average mean squared error between original and reconstructed single-beat ECG signals, along with the KL-Divergence $KL = -\frac{1}{2} \sum_{i=1}^{B} \sum_{j=1}^{J} (1 + \log(\sigma_{\phi,j}^{(i)})^2 - (\mu_{\phi,j}^{(i)})^2 - (\sigma_{\phi,j}^{(i)})^2)$, with B the number of samples in the mini-batch.

The best trained model values are: $RL = 1.09$ and $KL = 1.43$. The reconstruction error might seem strange at first glance, since it is quite high for training values which are normalized in the amplitude range of $[-1, 1]$. This is indeed due to the fact that the reconstructed signal is derived from a sampled latent space variable. In fact, a training datapoint is first fed as input of the encoder neural network, which returns as output the values of the bivariate gaussian distribution $\boldsymbol{\mu} = [\mu_1, \mu_2]$ and $\log \boldsymbol{\sigma}^2 = [\log \sigma_1, \log \sigma_2]$. A latent variable is then sampled from that distribution and reconstructed by the decoder neural network. In order to evaluate a proper MSE measurements it is indeed necessary to fed to the decoder input the mean values estimated by the encoder network, bypassing the sampling procedure. Given $\mathbf{x}_i$ for $i = 1, 2, ..m$ the original m training data samples and $\hat{\mathbf{x}}_i$ for $i = 1, 2, ..m$ the reconstructed ones bypassing the sampling procedure, the MSE is evaluated as

$$MSE = \sum_{i=1}^{m} ||\hat{\mathbf{x}}_i - \mathbf{x}_i||^2$$

Chapter 4

Application 1: test with a sinusoid dataset

4.1 Introduction

In the time domain, a sinewave signal is represented by 3 main features: amplitude, phase and frequency. Sinusoidal signals represent mathematically simple and well understood signals, for which those proprieties are well-defined and predictable.

$$\begin{aligned} x(t) &= A\sin(2\pi f_0 t + \phi) \quad t \in \mathbb{R} \\ x[n] &= A\sin(2\pi f_0 n + \phi) \quad n = 0, 1, ..., N-1 \end{aligned}$$

This allows to more easily assess the model interpolation and extrapolation capabilities of those essential characteristics. A sinewave is one of the most employed test signals. Amplitude can be varied in order to evaluate the dynamic range of a given device, while by varying the frequency we can easily test the frequency response of the device under test.

Any periodic signal can be decomposed into a sum of sinusoidal components using the Fourier series, this makes sinusoidal signals suited for the analysis and understandings of more complex periodic signals. The continuous and discrete sinusoid signal Fourier transforms are

$$\begin{aligned} X(f) &= \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt = \frac{A}{2}\Big(e^{j\phi}\delta(f-f_0) + e^{-j\phi}\delta(f+f_0)\Big) \\ X[m] &= \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}mn} \\ &= \frac{e^{j\phi}}{2}\sum_{n=0}^{N-1} e^{-j2\pi\frac{m-Nf_0}{N}n} + \frac{e^{-j\phi}}{2}\sum_{n=0}^{N-1} e^{-j2\pi\frac{m+Nf_0}{N}n} \quad m = 1, 2, ..., N-1 \end{aligned}$$

Those can be used to evaluate the frequency behaviour of the model generative process. Since this thesis work deals only with real amplitude time signals, the Fourier transform is always symmetric with respect to the origin, hence only positive values $f \geq 0$ are going to be considered.

4.2 Sinusoid dataset

The sinusoid training dataset was created with 6 different classes, each composed by 100 sinusoidal signals belonging to a specific set of frequency values $\{1, 2, 3, 4, 5, 6\}Hz$. Each signal was generated over an observation window of 1 second and the sampling frequency was set to $288Hz$, obtaining a total of 288 samples. The amplitude of each sinusoid is sampled from the uniform distribution $A \sim U[0.1, 1]$, in this way the signals are already normalised and ready to be used for training the VAE. Initially, the sinusoids are going to be set with a null phase component. Later, the phase component is going to be implemented by sampling it from the probability distribution $\phi \sim U[0, 1]$. Finally, a white gaussian noise is added to the sampled sinusoid amplitudes in order to assess the model rejection to noise. The noise is set to be proportional to the sampled amplitude $A \sim U[0.1, 1]$ and scaled by a proper factor $K = 0.1$. Given $n_z \sim \mathcal{N}(0, 1)$, the added noise is equal to $n = K \cdot A \cdot n_z$. Three dataset are then synthetized

1. Dataset 1: 6 fixed frequencies(classes), randomly varying amplitude
2. Dataset 2: 6 fixed frequencies(classes), randomly varying amplitude, randomly varying phase
3. Dataset 3: 6 fixed frequencies(classes), randomly varying amplitude, added gaussian noise to the amplitude

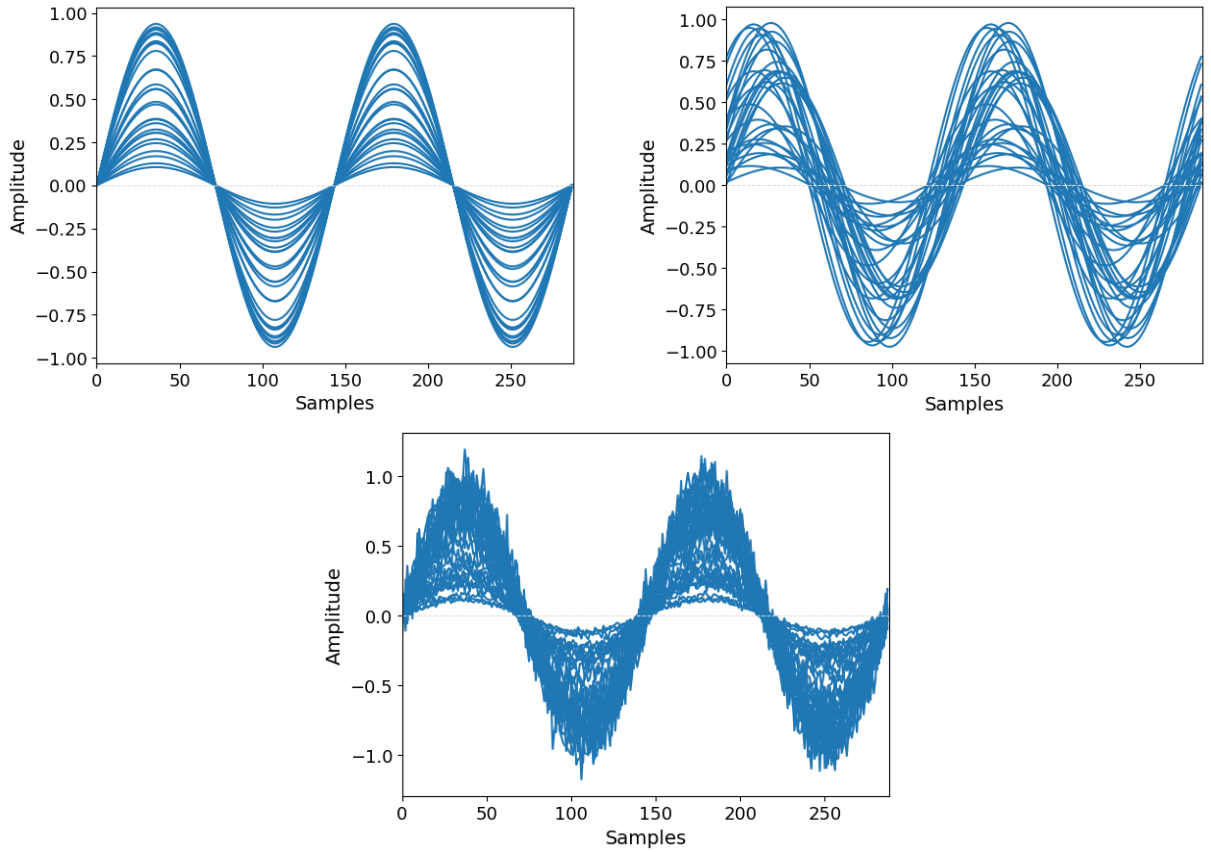

Figure 4.1: Some signals of the dataset 1 (top-left), 2 (top-right) and 3 (bottom)

4.3 Training results

The model is trained using mini-batch Adam optimiser, with a batch size of both 32 and 64 samples and a total of 72 epochs. It took approximately 4 minutes to complete the training process by using the standard CPU available in Google Colab, an Intel Xeon CPU with 2 vCPUs and 13GB of RAM.

The evolution of the latent space mapping during the training phase is reported in figure 4.2; in this case the first dataset was employed. During the first epoch, the latent space is *collapsed*. The model provides a unique representation for all the training data, hence it is still not able to capture the data features and distinguish the different classes. After the third epoch, the training datapoints are encoded along a single diirection. The model is starting to slowly capture some of the data feature and mapping the latent space accordingly. Since the classes are still not clearly separated, this means that the model is first learning the amplitude feature, but it is still not able to recognise the frequency of the sinusoids. It is expected for the reconstructed signals to not preserve the morphology typical of a sinusoidal signal, but still have different amplitudes along the mapped line. From epoch 5 to 14 the model learns also to better recognize the frequency feature. This leads to a mapping of the latent space where each class of clusters occupies a well defined region of space, leading to a final star shaped formation. The remaining epochs still mapped the class clusters with a common origin, but more equally spaced and in a more compressed region of space.

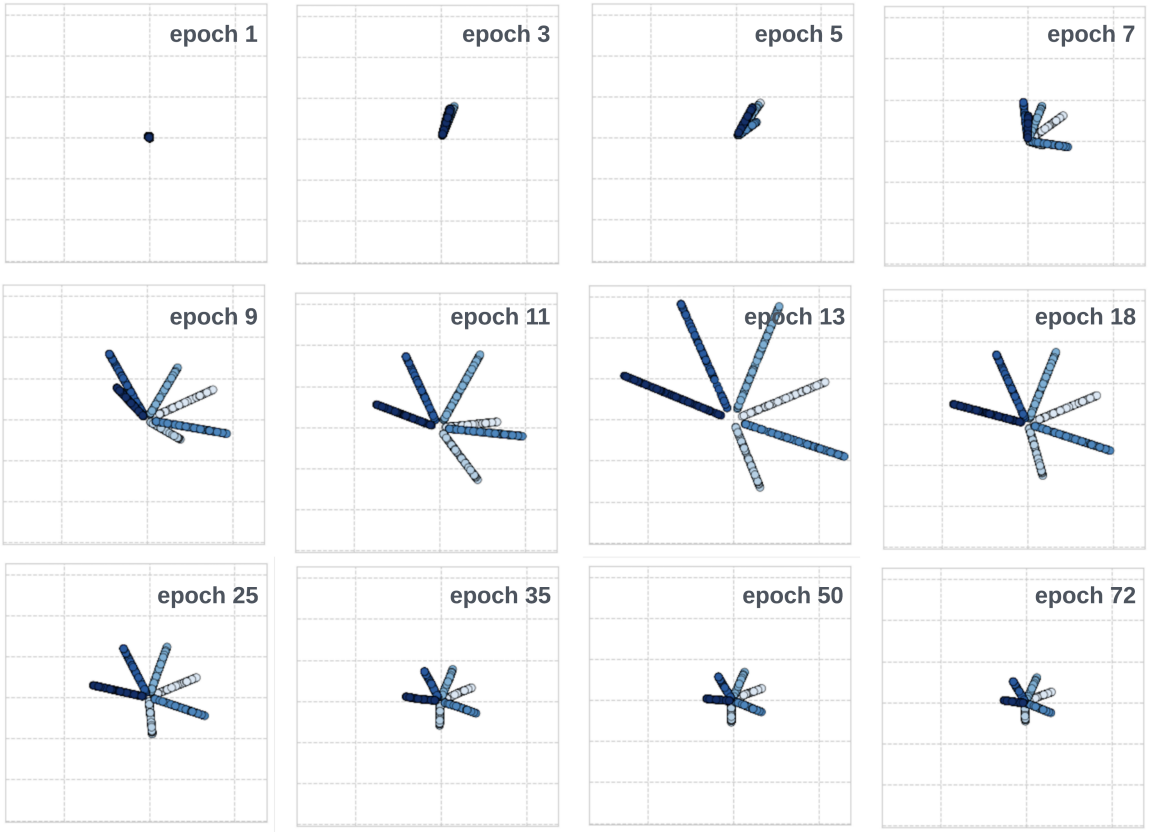

Figure 4.2: Mapping of the training datapoints in the latent space at different training epochs

The original and reconstructed sinusoidal waveforms are plotted in figure 4.3, 4.4 and 4.5 for respectively the dataset 1, 2 and 3. The morphological shapes seem to be preserved, moreover figure 4.5 shows also the noise rejection capabilities of the model. The reconstructed waveforms, compared with the original one, are clearly distinguishable.

In order to make some more consistent evaluation of the reconstructing capabilities of the model and its noise rejection, proper quantitative evaluation metrics are used. The reconstruction loss (RL) and the Kullback-Leibler divergence (KLD) are the losses used during the training process, while the mean squared error(MSE) is only used for evaluation purposes. The first two metrics are just the output of the final epoch of the training process while the MSE has been calculated has explained in section 3.3.

The dataset 1 and 2 present a very low MSE, proving very good performances of the model in reconstructing the training data samples. The metric is one order of magnitude higher for the dataset 2, which contains sinusoids with also a varying phase shift. This is an expected behaviour, since the complexity of the feature in this case is higher, while the training procedure is kept the same. For what concern dataset 3, the RL metric is much higher compared to dataset 1 and 2. The model reconstructs the input signals by rejecting the high frequency gaussian noise, hence the RL is calculated between the original noisy signals and the reconstructed de-noised ones, making the amount of error higher by a factor proportional to the added noise. In order to

properly evaluate the reconstruction and denoising capability of the model, without interfering with the generative oriented training process, the MSE is used. It is considered $\mathbf{x}_i$ the original data samples without added noise, $\mathbf{x}_i^{(n)}$ the original data samples with added noise and $\hat{\mathbf{x}}_i^{(n)}$ the reconstructed data samples obtained by giving as input to the VAE the original ones with added noise.

$$MSE = \sum_{i=1}^{m} ||\hat{\mathbf{x}}_i^{(n)} - \mathbf{x}_i||^2$$

In this way it is possible to provide a first evaluation about the denoising capability of the developed VAE. In this example, a $MSE = 2 \times 10^{-4}$ is obtained, thus suggesting that the proposed VAE could be considered, in future applications, also for signal denoising.

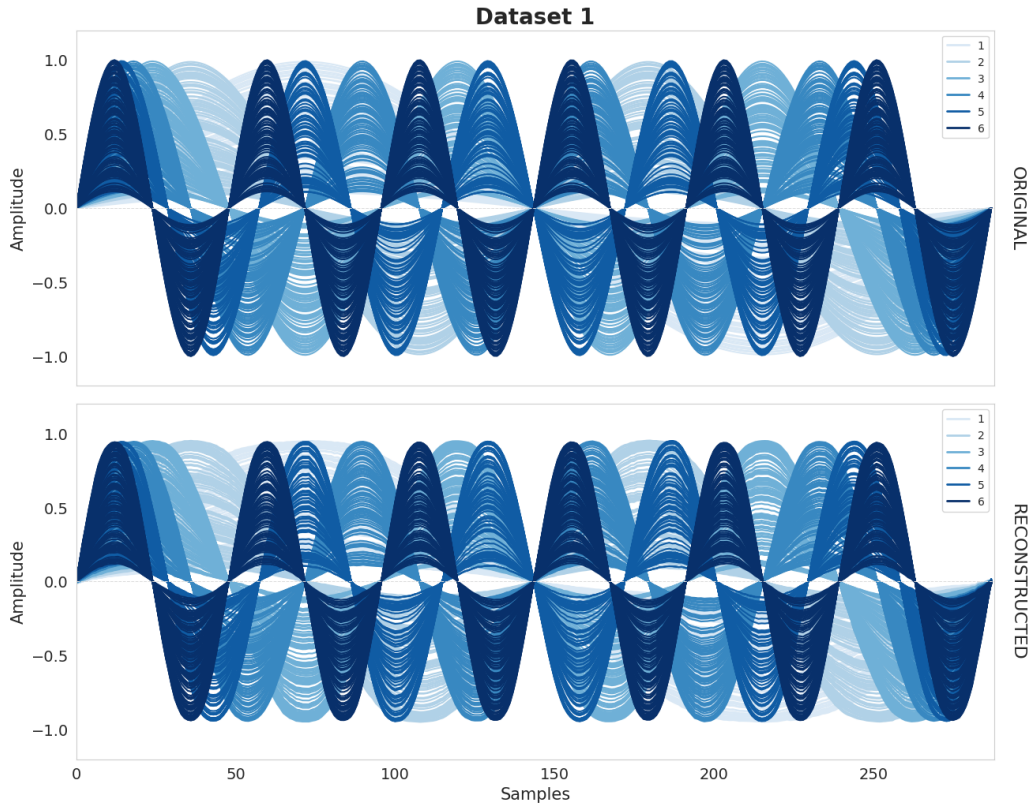

Figure 4.3: Dataset 1: original training signals(top) and reconstructed training signals(bottom)

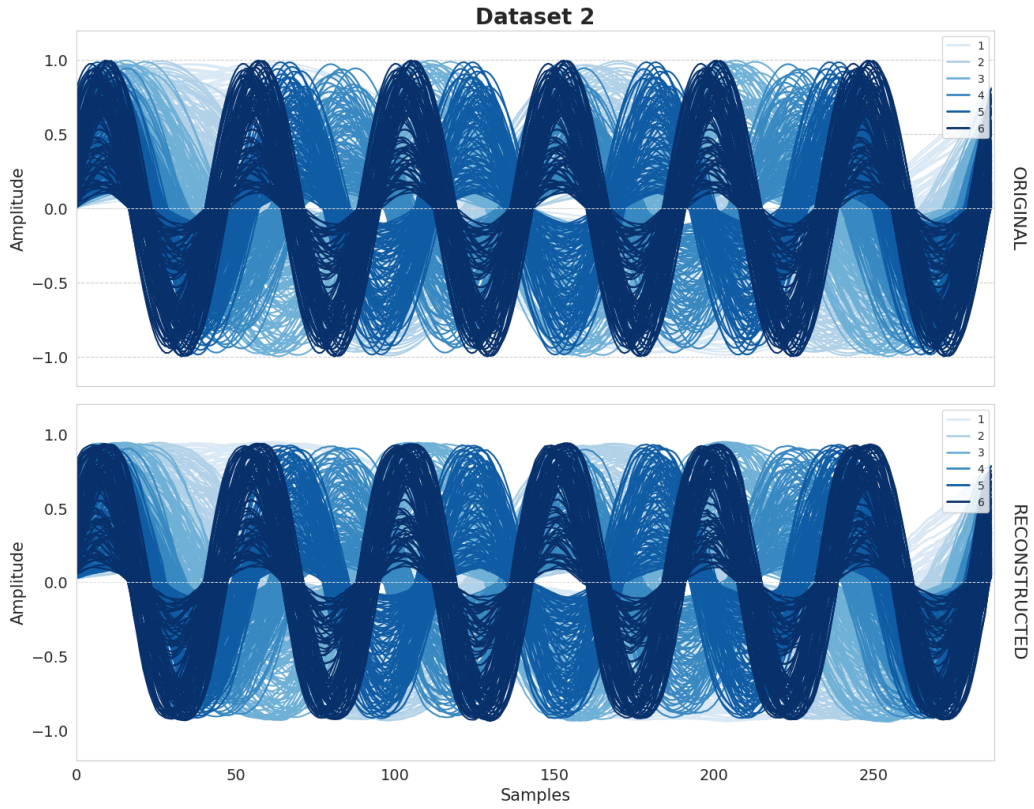

Figure 4.4: Dataset 2: original training signals(top) and reconstructed training signals(bottom)

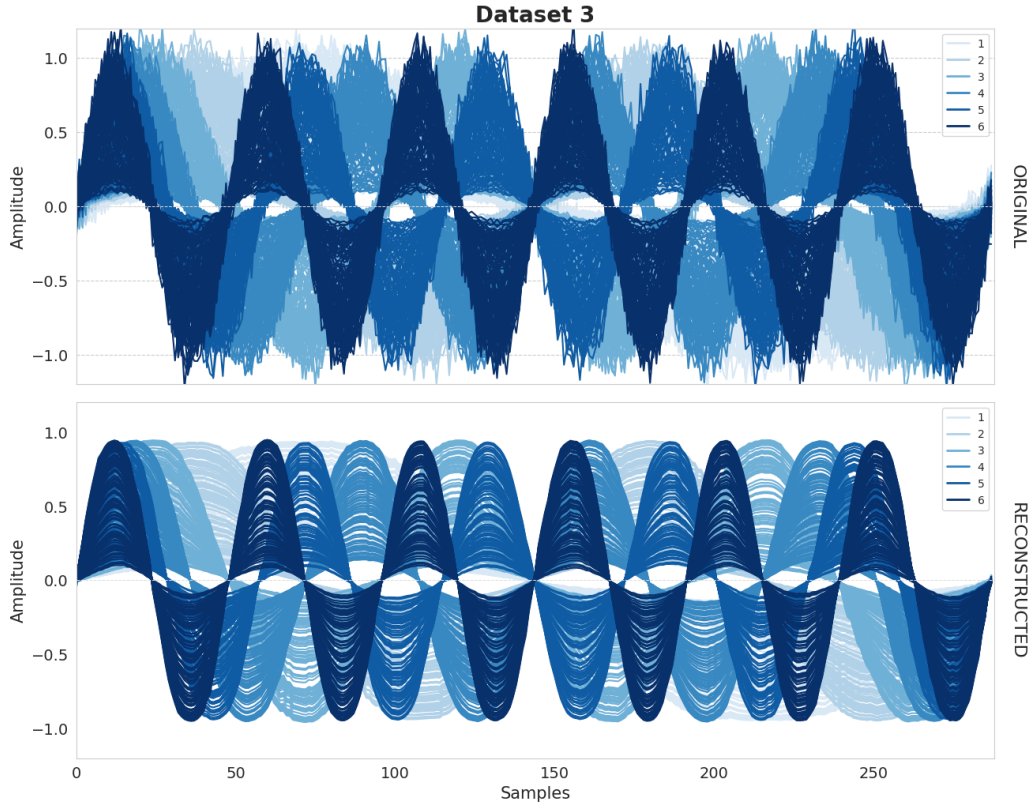

Figure 4.5: Dataset 3: original training signals(top) and reconstructed training signals(bottom)

Dataset	RL	KLD	MSE
1	1.8	6.8	6×10^{-05}
2	4.3	8.4	4.5×10^{-04}
3	35	8.3	1.9×10^{-04}

Table 4.1: Reconstruction Loss (RL), Kullback-Leibler Divergence (KLD) and Mean Squares Error (MSE) results of the sinusoid dataset

4.4 Latent space

The developed VAE latent space is a two-dimensional space in which the encoder networks maps the input data. More precisely, the encoder network, given an input data, returns the parameters of the bivariate gaussian distribution(means and logarithmic covariances) from which an encoded data point is then sampled. The latent space is mapped continuously, enabling smooth interpolation between the encoded training data points. Different regions of the latent space are associated to encoded data having different features. This interpolation capability is a direct result of the continuity and smoothness of the latent space and is particularly useful for morphing between different signals shapes and structures. It allows to generate new data samples characterised by a mix of feature of the training data. Once the proposed VAE model is trained, the encoder and decoder components can be used as independent neural network models.

The encoder neural network receive as input a 288-th dimensional variable, in this case a 288 samples long signal, and encodes it into a lower dimensional latent space, with just two dimensional components $f_e : \mathbb{R}^{288} \rightarrow \mathbb{R}^2$. The decoder on the other hand, receives a two dimensional variable as input and then outputs a 288-th dimensional one $f_d : \mathbb{R}^2 \rightarrow \mathbb{R}^{288}$. The last layer of the encoder component is a custom sampling layer based on a bivariate gaussian distribution. The encoder inner architecture converts a 288 samples signal input into a 4 parameters of the bivariate gaussian distribution, the mean values $\boldsymbol{\mu} = [\mu_1, \mu_2]$ and the logarithm variances $\log \boldsymbol{\sigma^2} = [\log \sigma_1, \log \sigma_2]$. Next, the sampling layer samples from that probability distribution. The randomness added by the sampling layer is essential during the training phase for preserving the model generative performances. In order to map the latent space, however, the sampling procedure is bypassed and it is plotted directly the gaussian mean values $\mathbf{z} = \boldsymbol{\mu} = [\mu_1, \mu_2]$ of each training data.

The figure 4.6 represents the latent space encoded mean values of the training data. The generated latent space representation has mapped the sinusoidal training signals approximately along 6 lines, one for each class. This exhibits the model capability of properly interpreting the specific classes features, hence, distinguishing the data samples of each individual class by grouping them in well defined clusters. Those lines are quite equally distanced and arranged in a radial pattern with a common origin.

The model, even if the same training data are used, learns a different latent representation each time it is trained. This is due to the random nature of the mini-batch training process,

which uniformly samples the training signals to be used at a specified training epoch. However, if properly trained, even if the position of the class clusters changes, the main mapping structure remains the same.

The dataset 1 contains sinusoids with a specific frequency and varying amplitude. By just looking at the latent space, it can be assessed a general interpretation on how it is mapped. Each line represent a specific class, i.e. sinusoid signals at different frequencies, while each point along that line represent sinusoids with different amplitude values. Sinusoidal signals with different frequencies tend to be harder to distinguish as the amplitude decreases, this suggests that the points nearby the star center are the ones with lower amplitude. The points (encoded signal) seems to be quite uniformly distributed along the lines. This again comes with a meaningful interpretation, since the sinusoids of each class had their amplitude uniformly sampled in the range $[0.1, 1]$. The encoded signals of dataset 2 have a similar interpretation of dataset 1, with the main difference that also the changing phase component is mapped into the latent space, making the lines spread radially, but still clearly distinguishable. This, along with the low value of the reconstruction loss 4.1 , is an indicator of the model ability to correctly capture both the changing amplitude and phase features. Finally, the latent space associated to dataset 3 has a shape similar to the one associated to dataset 1.

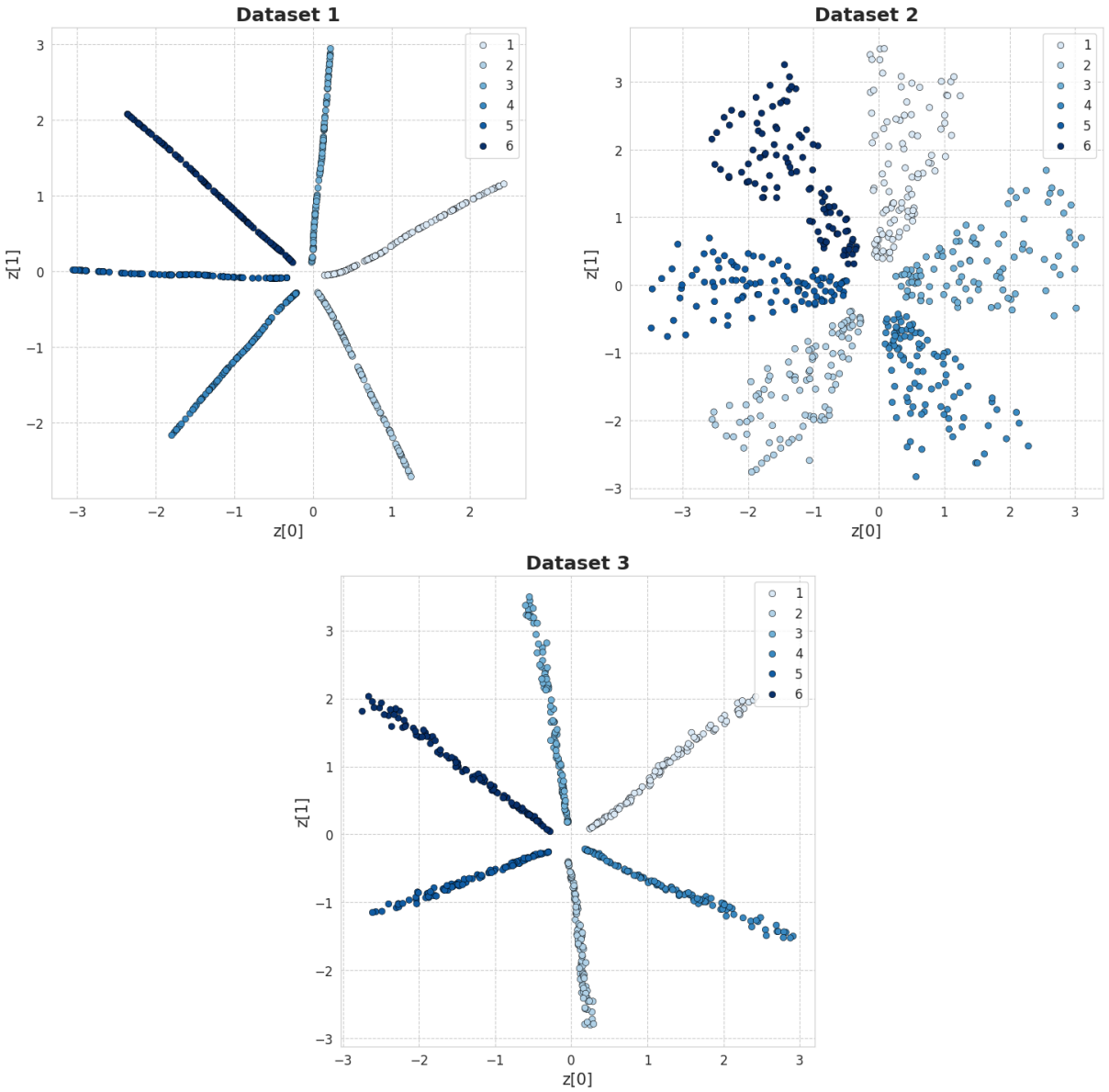

Figure 4.6: Latent space mapped with the training datapoints belonging to dataset 1(top-left), 2 (top-right) and 3 (bottom)

4.4.1 Cluster inspection

In the following sections a specific cluster of points, the coloured ones in the presented figures, is properly chosen. The encoded points are decoded to obtain the associated reconstructed signals and their magnitude Fourier transform with the purpose to more clearly assess the mapping of the latent space. Figures 4.7, 4.8 and 4.9 plots respectively a cluster of points along one of the six training data clusters. The reconstructed signals confirm the hypothesis carried on in the previous section. Along one of the six lines there are encoded sinusoidal signals with the same frequency and different amplitudes, which increases by moving from the center of the star to the external points. The FFT plot shows that the decoded signals are sinusoids with perfectly the same frequency of the considered class.

Figure 4.8 shows that, at a similar radial distance from the center, the class points capture the phase shifting feature of the training signals. The reconstructed signals have indeed the same frequency, approximately the same amplitude and different phase values.

Figure 4.9 shows that the reconstructed signals are almost identical to the ones without noise of dataset 1, with just a small amount of high frequency noise. From the reconstructed signal in the time domain, it can be noticed that the high frequency noisy components are more relevant for the sinusoids with larger amplitudes. This is an expected behaviour since, in the dataset 3, the amount of added noise is proportional to the sinusoid amplitude.

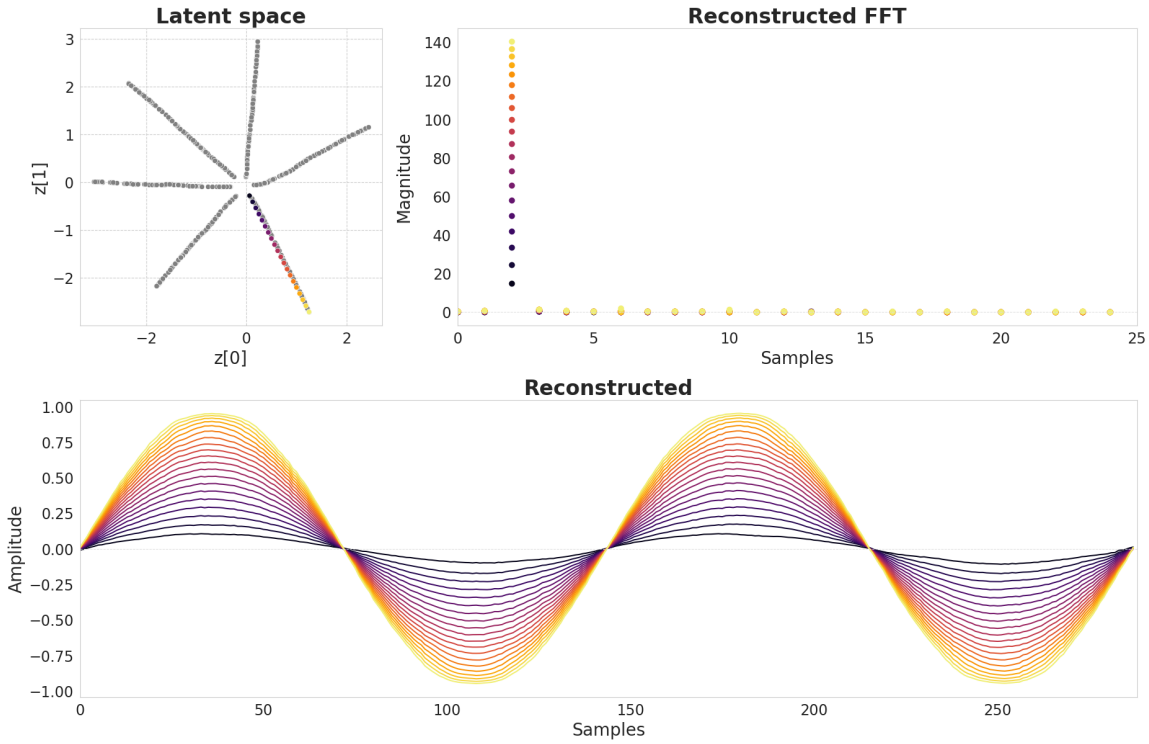

Figure 4.7: Dataset 1: cluster inspection

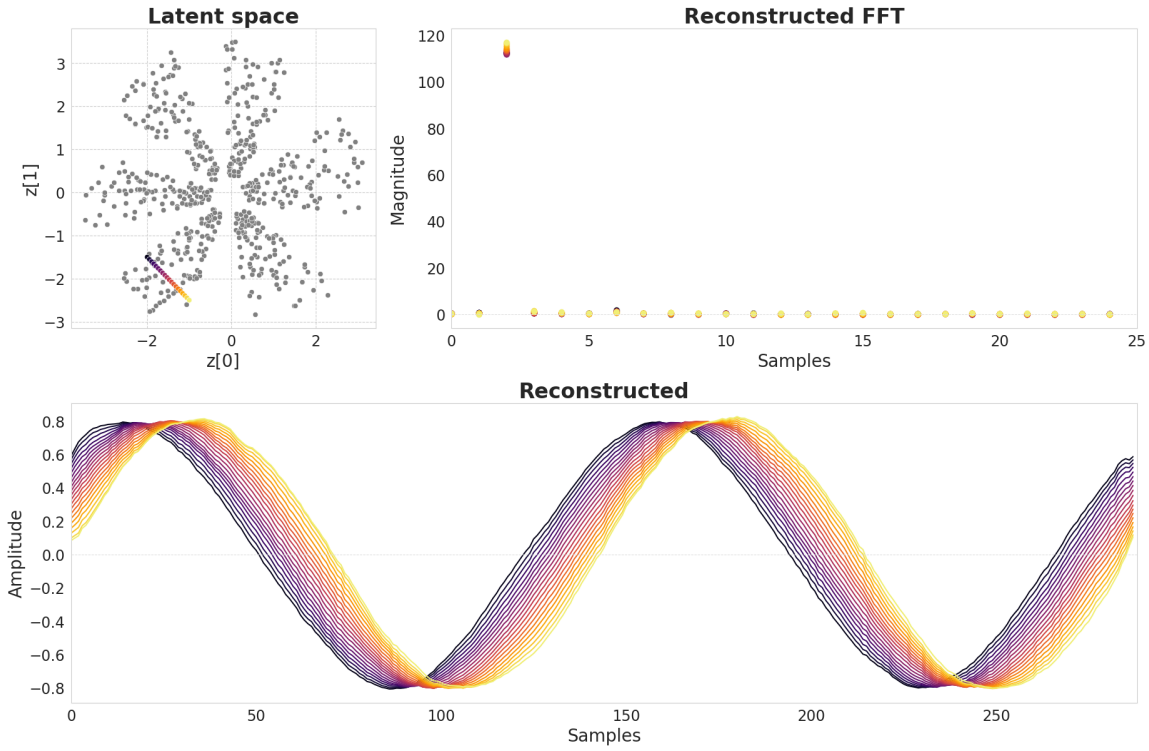

Figure 4.8: Dataset 2: cluster inspection

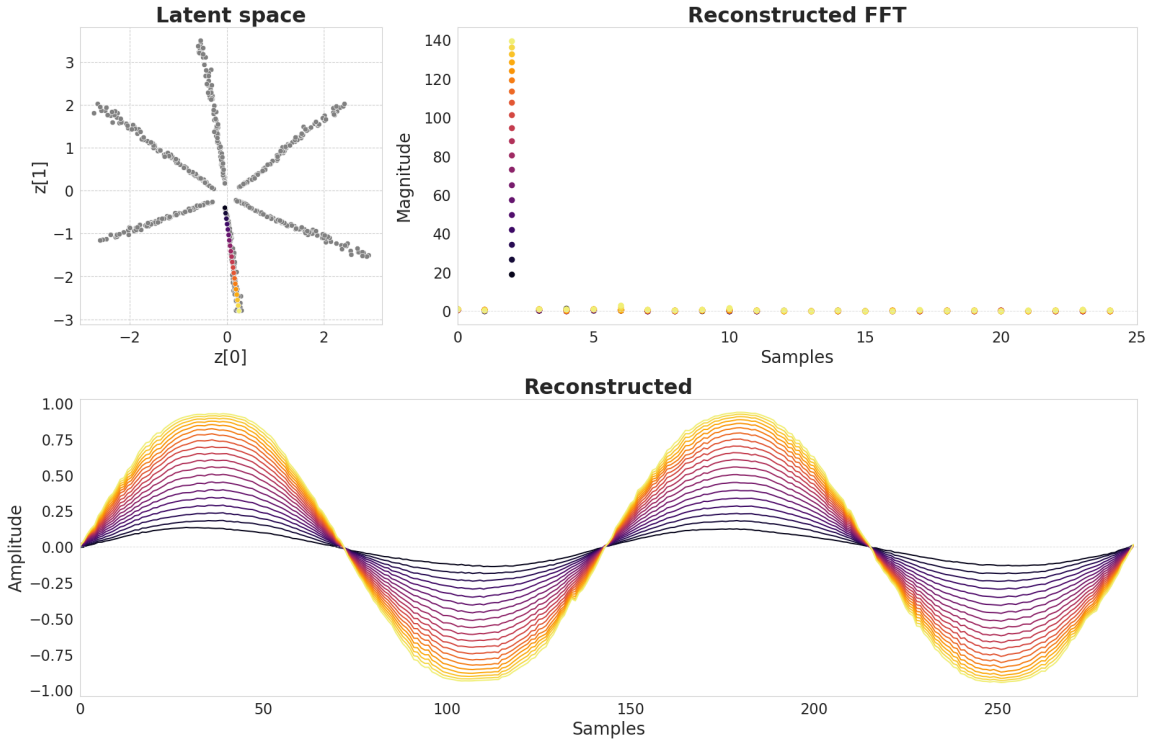

Figure 4.9: Dataset 3: cluster inspection

4.4.2 Interpolation

Next, the interpolation capability of the model is evaluated. It is chosen a set of points in the latent space, which are then decoded using the decoder network. The reconstructed signals will be more affected by the features of the closer clusters. The closer the cluster is and the higher its effect will be. Figure 4.10, 4.11 and 4.12 represent the interpolating points along a line among two classes clusters for respectively dataset 1, 2 and 3. Figure 4.10 shows an interpolation between the features of the two surrounding clusters. By moving from one cluster towards another, the frequency component gradually changes.

Once that the training signal features are correctly identified, that, in this case, correspond to the frequency of the two classes, the VAE maps the intermediate latent space region in such a way that both the features are combined with different weights. The amount of specific features that characterise the reconstructed signal is inversely proportional to the geometric distance between the encoded signal and the features of the specified cluster. The closer the encoded signal to one cluster, the more the signal is characterised by the features of that cluster.

The sinusoids of dataset 1 are composed by 2 two features, the amplitude and the frequency. The behaviour becomes much clearer by looking at the frequency domain of the reconstructed signal. There are two main frequency components, which are the frequencies associated to the two surrounding clusters, along with some other noisy components in the nearby spectrum. The interpolation seems to be approximately just the sum of the signals associated to the clusters, weighted proportionally to their geometric distances.

Figure 4.11 shows that the coloured line edges point (darker and lighter ones) are associated to sinusoidal signals with different amplitudes. This is expected since their encoded representations have different radial distances from the star center. By looking at the reconstructed signals, the sinusoidal signals with frequency equal to $4Hz$ start shifting in phase and keep almost the same amplitude. This is due to the fact that the line moves along the width of the class cluster. Next, in the intermediate region, the features of the two sinusoidal components are mixed. Finally, when reaching the second cluster, the reconstructed sinusoidal signals have the same frequency of the associated cluster, but still shift in phase.

Figure 4.12 shows that the model, trained with dataset 3, behaves almost identically to the one trained with dataset 1. The model is able to maintain its generative properties while still applying its denoising feature.

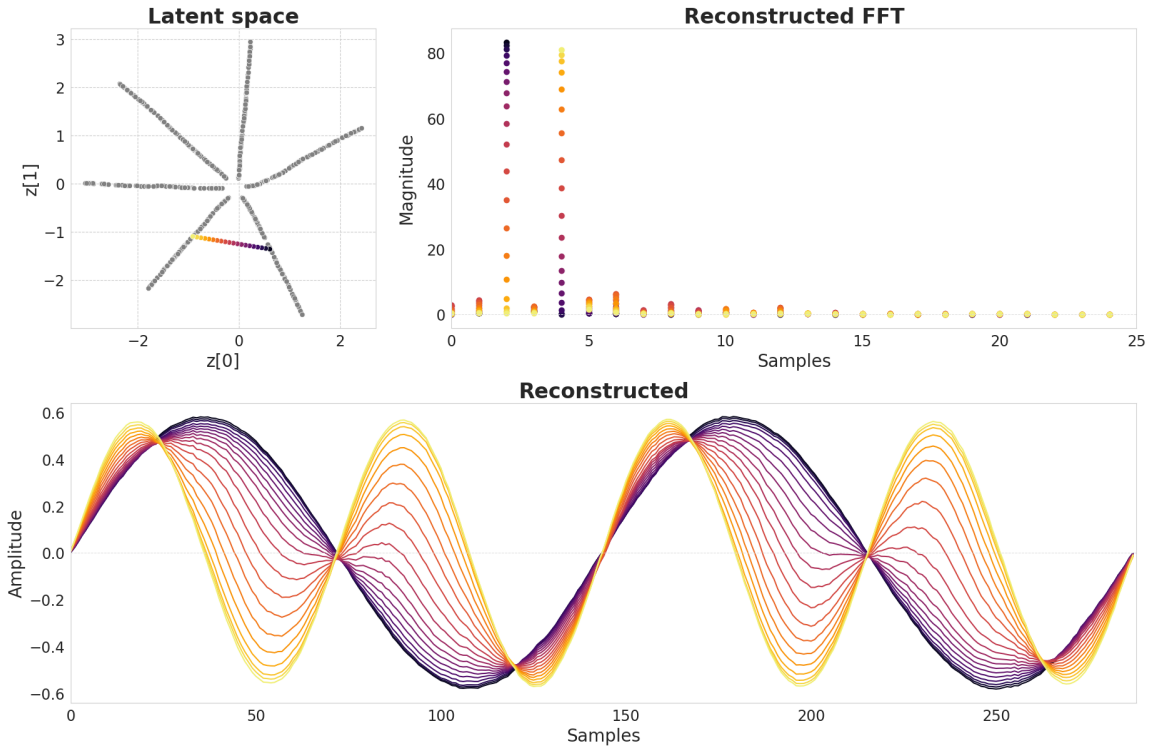

Figure 4.10: Dataset 1: interpolation capabilities

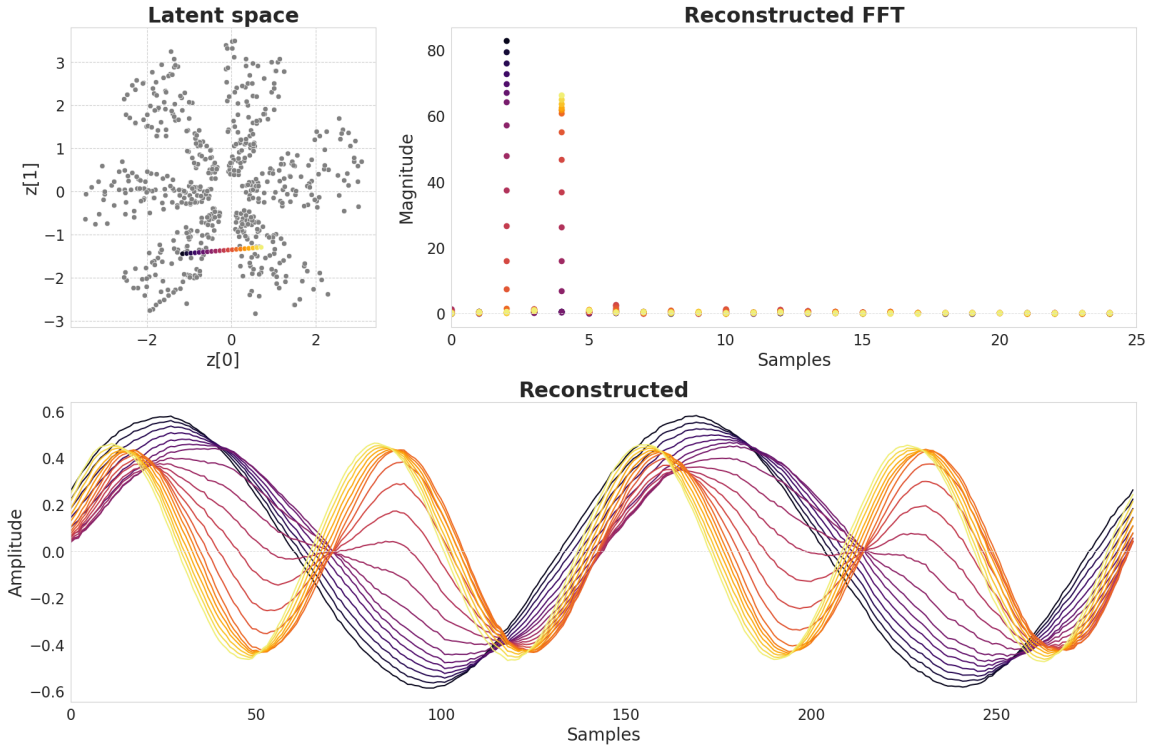

Figure 4.11: Dataset 2: interpolation capabilities

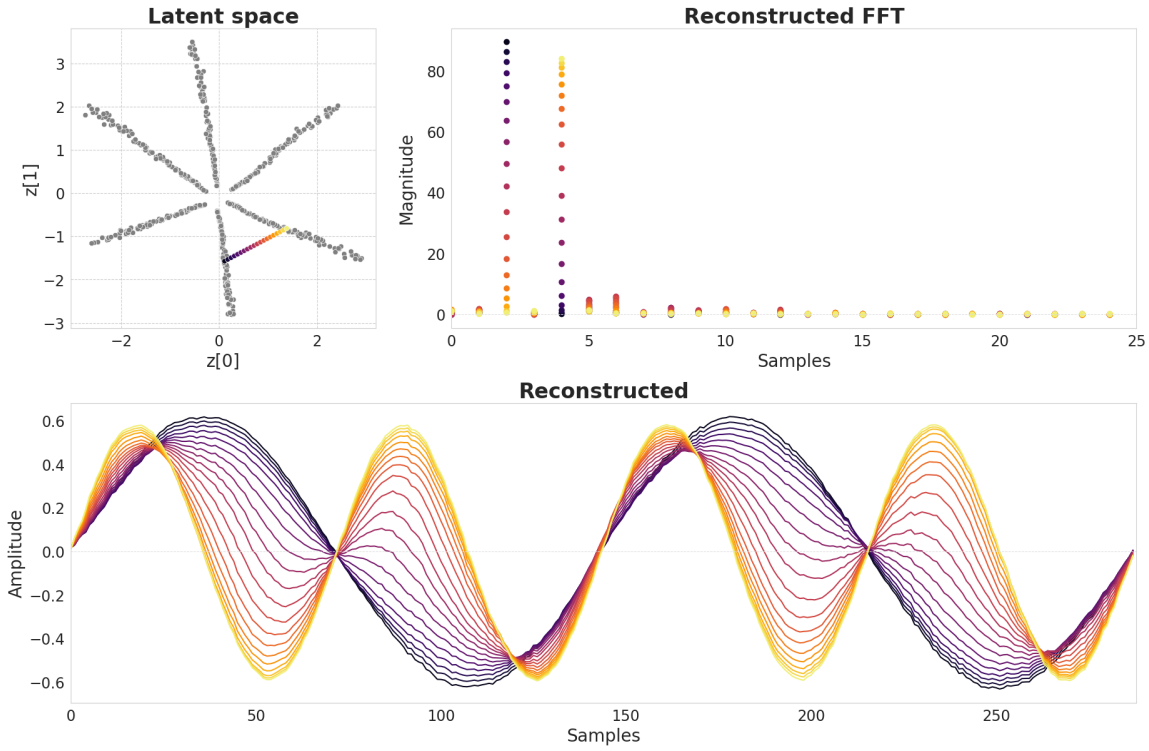

Figure 4.12: Dataset 3: interpolation capabilities

4.4.3 Extrapolation

In order to assess the extrapolation capability of the proposed model, it is needed to choose some encoded signals in the latent space outside the region mapped with the training ones. The dataset 1 is used for this purpose, since it has the clearer and more interpretable latent space mapping.

The latent space is sampled along a line starting from the center of the star and with length about two times the length of the radius of the star, as reported in Fig. 41. The reconstructed signal saturates if its encoded representation stands outside the mapped region. Figure 4.13 shows that the reconstructed signals are not more characterized by a sinusoidal shape and tends toward a square wave. This can also be evaluated in the frequency domain, where there are also harmonic components in correspondence to the odd multiples of the fundamental frequency.

This behaviour has a clear explanation: the last layer of the decoder neural network has an hyperbolic tangent as its activation function. This means that all the output of the decoder, hence the reconstructed amplitudes, are compressed in the range $[-1, 1]$. The VAE clearly captures the amplitude feature, but it is not able to extrapolate it further with respect to the training signals due to the limitation imposed by the tanh activation function. Without entering in details, the most straightforward solution to the problem is to change the activation function from the hyperbolic tangent to the linear ones $f(\mathbf{x}) = \mathbf{x}$. In this way, both negative and positive signal values are passed through without any compression. This choice has the main drawback of reducing the numerical stability of the training procedure. In fact, without any constraints,

the output values can become very large or very small. If the network is trained on the same dataset 1, but the last decoder layer activation function is set to ”linear”, the problem is solved (fig. 4.14). Moreover, it seems that this approach leads to a similarly good qualitative results compared with the initial model.

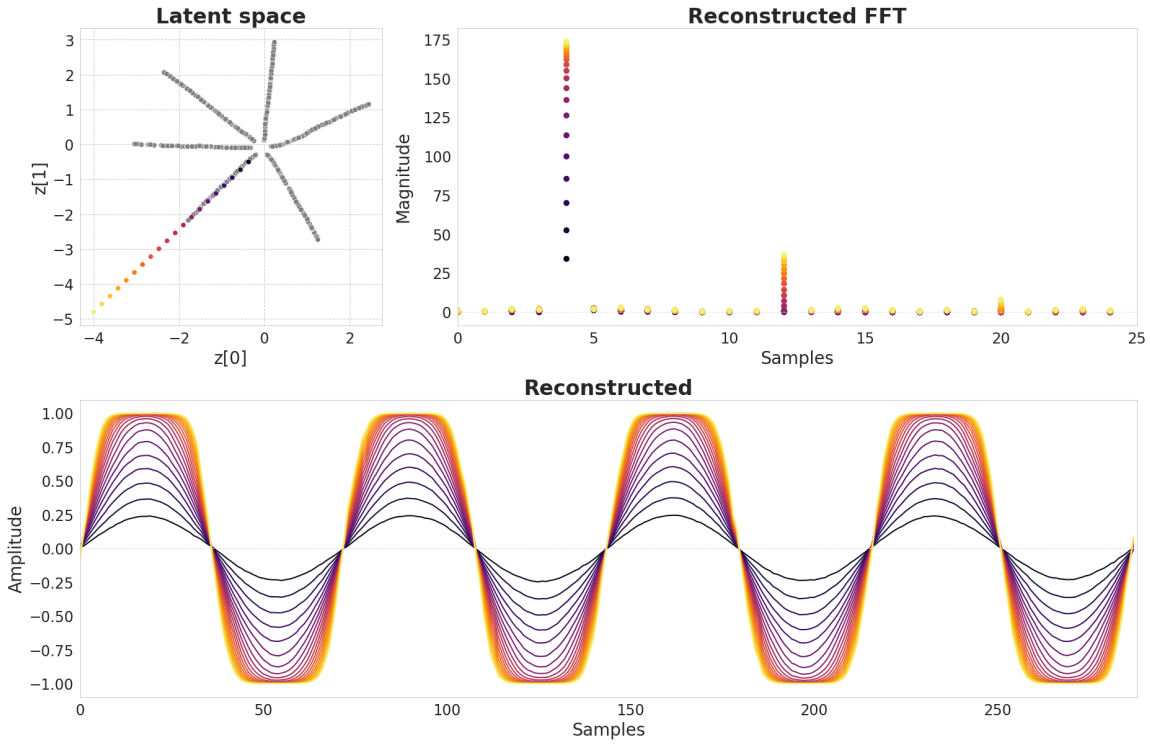

Figure 4.13: Amplitude compression of the reconstructed sinusoids, due to the tanh activation function of the last decoder layer

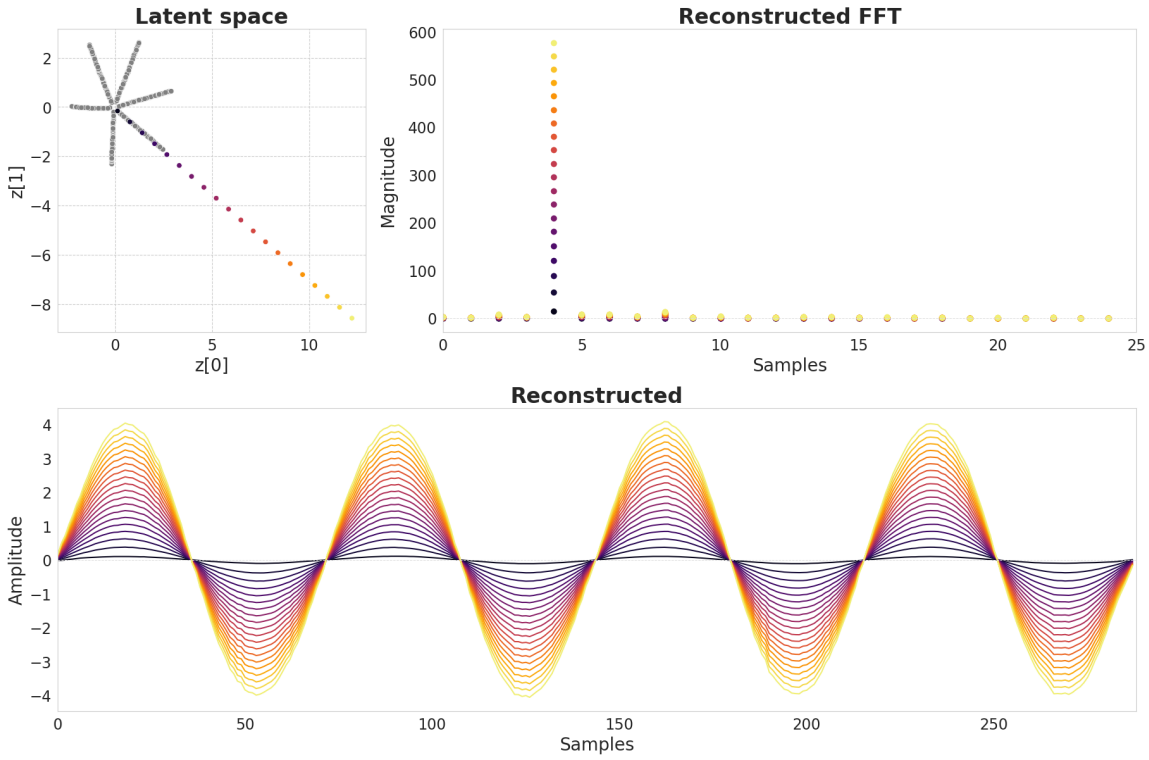

Figure 4.14: The amplitude is no more compressed if it is applied a linear activation function to the last decoder layer. The VAE is now capable of extrapolating the amplitude features of the sinusoidal signals.

Chapter 5

Application 2: ECG dataset

5.1 Dataset and pre-processing

The dataset under test is composed of 12 lead ECG signals up to one minute long and of 20 different healthy people. It is derived from the public database MIT-BIH Arrhythmia Database. This dataset is scarcely populated, hence synthetically augmenting it could be greatly beneficial for its potential use in data-driven applications. The main features are listed below:

- Only healthy ECG
- Number of people = 20
- Sampling frequency = 500 Hz
- Number of ECG leads = 12
- ECG traces length = up to 1 minute
- The network is trained on lead no. 1 of 6 people

In machine learning applications, data preprocessing stands as an essential foundation for the efficacy of any data driven model. It acts as a critical intermediary step, transforming raw data into a much more usable format that aims to improve the learning process of a model. It ensure that the data conforms to specific application requirements, allowing the model to achieve proper feature extraction capabilities. Data preprocessing deals with both the quality and the quantity of the data, along with the most suited data representation for the model to learn. Good quality data has a minimum presence of noise, no inconsistencies, no duplicates and its outliers are carefully handled. In general, the more complex the data structure is, the greater the amount of data samples is required for the model to correctly interpret it. Moreover, when dealing with labeled data(supervised learning), it is essential to have approximately the same amount of samples for each class. Finally, proper transformations applied to the specific data type can

improve the numerical stability of the model during training and evaluation, leading to a faster training and better performances.

The quality of the measured ECG is quite high, there is very little presence of high frequency noise, since the data was already cleaned beforehand. No duplicate ECG traces were found to be present in the dataset.

During a visual inspection of the ECG traces, two of the classes(15 and 19) clearly appeared to be compromised. The amplitude readings of the electrical hearth activity of each leads were completely off scale and the cardiac beat morphology was abnormal. These classes have been discarded.

Since the model aims to process single beat ECGs, the next task is to slice the ECG traces and extract each cardiac beat. For this purposes, the Pan-Tompkins algorithm is used to detect the R-peaks and a suited observation window is used to extract the samples of each ECG beat. The cardiac frequency, in general, changes quite frequently in time. If the measured subject is supposed to be in a state of rest during the ECG measurements, it is highly probable that the cardiac frequency remains almost the same for smaller interval durations, such as the ones from the measurements(up to 1 minute). However from one healthy subject to another it could vary consistently. This leads to ECG traces that presents slight or more consistent changes in the cardiac frequency. Therefore, the extracted ECG beat signals have different samples length. The implemented deep learning model, however, requires a fixed-length input, so also the length of the single beat ECG signals must be fixed. Ideally, this would be achieved by cutting the number of edges samples of all the beats down to the smallest amount of signal samples among the dataset signals. However, for longer heart beat signals, this could cause cutting some of the relevant information of the signal itself(one of the main peaks). A possible solution is to choose a proper amount of samples to keep, then removing all the signals with a lower amount of samples and cutting the edges of the remaining ones. The used dataset is scarcely populated and the overall cardiac frequency of the stored ECG leads does not change excessively, so the proposed solution has excluded only a really small amount of signals.

Data *normalization* involves scaling the input data to a specific range or distribution. This process ensures that each feature contributes more equally to the learning process, preventing certain features from dominating the others due to their scale. Moreover, normalizing data helps in accelerating the training process and improves the overall performances of deep learning models. For example, since neural networks are typically optimized using gradient-based methods, if the input features vary widely in scale, the gradients can also vary significantly, leading to inefficient updates and slow convergence.

Normalizing the data ensures that all input features have a similar scale, leading to more uniform gradient updates and faster convergence during training. Neural networks involve also numerous mathematical operations, including multiplication, addition and activation functions. When the input data has large variations in scale, it can lead to numerical instability, causing computational issues such as overflow or underflow. Normalizing the data helps in maintaining

numerical stability, ensuring that the training process remains robust and efficient. Finally, normalized data helps in regularizing the model, reducing the risk of overfitting to the training data. By ensuring that all features contribute equally, normalization promotes learning representations that generalize better to unseen data.

Commonly used normalization techniques are Min-Max, Z-score, Robust, Log scaling, Robust scaling and Max Abs scaling. Since ECG signals are essentially a measure of the difference of electrical potential between two points(ECG lead), they can have both positive and negative amplitude values. The zero corresponds to the absence of electrical activity. During the generative process it is essential to preserve this representation in order for the signal to be properly interpreted. Applying techniques such as min-max or log scaling, would not allow to reconstruct back this representation. It is, indeed, needed a zero-centred transformation, which normalize the amplitude values in the range of both negative and positive values. For this purpose, it is used the Max Abs scaling, which constraints the data in the range [-1,1], it is zero centered and easily reversible

$$\mathbf{x}_n = \frac{\mathbf{x}}{\max(|\mathbf{x}|)}$$

The single beat ECG signal is composed of 5 main peak components (see section 1.3). In an healthy subject, the R peak has usually the highest amplitude among all. The difference of this peak compared to the others, especially Q and S can be very large even after the normalisation of the signal. The VAE model is based also on the reconstruction loss metric which in our case has been implemented with a MSE. This means that the model would tend to give priority to the reconstruction to the higher amplitudes peaks, especially the R ones. In order to reduce this effect, a *dynamic compression* has been applied to the data. This must compress efficiently a signal with both positive and negative values, be zero centered and reversible. For this purpose, a simple transformation based on the hyperbolic tangent was used

$$x_c = \tanh(k \cdot x)$$

with $k = 1.5$ being the applied compression ratio.

Class balancing refers to the process of ensuring that each class in a dataset is represented equally. When classes are imbalanced, a model might become biased towards the majority class, leading to poor performance on minority classes. Class balancing can be achieved with two main approaches: oversampling and undersampling. Class oversampling is one of the main goal of the developed model. Synthetically augmenting the data used to train it would ruin the genuineness of the model results, hence the only left approach is undersampling. The dataset being used present 18 classes(excluding the two abnormal ones), with a wide range of single beat ECG data samples(fig. 5.1). In order to have a better insights on the model working principle, only 6 classes are chosen $\{1, 2, 3, 4, 14, 18\}$. The smallest populated classes among them are 3 and 14, with 15 samples, while the others have been undersampled down to that number, by randomly removing excess data samples.

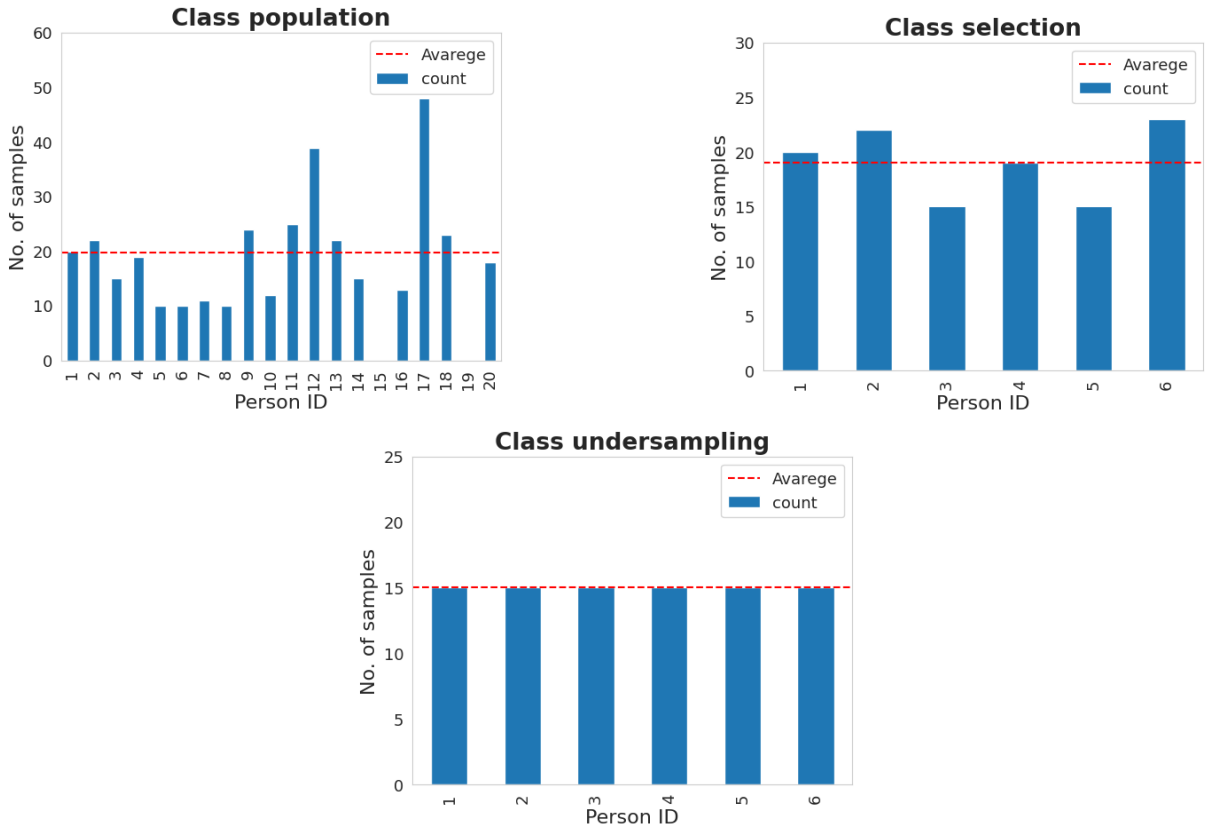

Figure 5.1: The class population of the provided single beat ECG dataset. The number of samples represents the number of single beat signals associated to a specific person(class). (Top-left) The population of all the dataset classes excluded the abnormal ones. (Top-right) The population of the selected classes to be augmented. (Bottom) Undersampled selected classes, which achieved classes balance

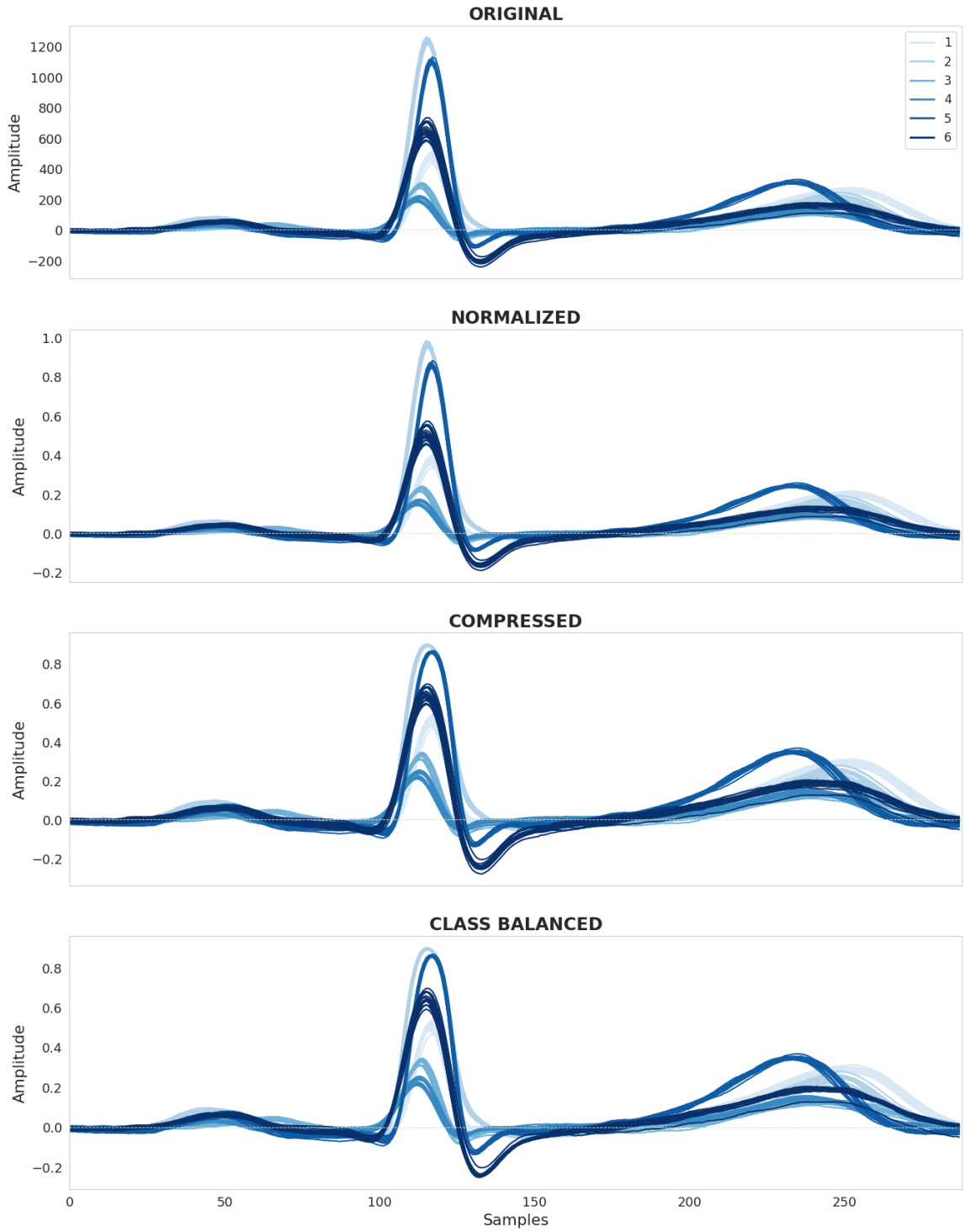

Figure 5.2: Preprocessing steps of the ECG training dataset. From top to bottom the visualisation of the training signals once each transformation is applied: normalization, compression and class balancing.

5.2 Training results

The model is trained using mini-batch Adam optimiser, with a batch size of both 32 and 64 samples and a total of 216 epochs. It took approximately 1 minute and 29 seconds to complete the training process by just using the standard CPU available in Google Colab, an Intel Xeon CPU with 2 vCPUs and 13GB of RAM. During the first epoch, the latent space representation is *collapsed*. The model finds a unique reconstructed representation for all the training data, hence it is still not able to capture the data features and distinguish the different data classes, just like the sinusoid test dataset. After the 18th epoch, the training datapoints of each classes start to be mapped into clusters in different region of the latent space. Up to the 72th epoch those class clusters becomes more and more defined, even if the model seems to be mapping them along a linear subregion of the latent space. From epoch 90th to 126th the model learns a more fruitful mapping of the training signals by spreading them thru a two dimensional region of space.

This concept may be explained with a simple example. The trained VAE maps the latent space such as the different regions are associated to different features of the training signal. Regions next to each other have one or more similar features. On the other hand, regions located really far away from each other are expected to have different features. If three clusters of points share a common feature with a fourth one, they should be placed around the fourth one, however if the latent space has dimension of just 1, this cannot happen and the training forces the model to find a less suited latent space mapping. The higher the latent space dimensionality, the more placement possibility there are for the feature regions. If the latent space dimension is too low to correctly capture all the training data diversity, the model fails to accurately reconstruct the encoded representation. This is also called as the *bottleneck effect*.

The remaining epochs don't seem to be changing the latent space mapping in some relevant way. They indeed have been used to achieve a really small increase in the overall training error. This particular case, shows that the chosen training process is stable.

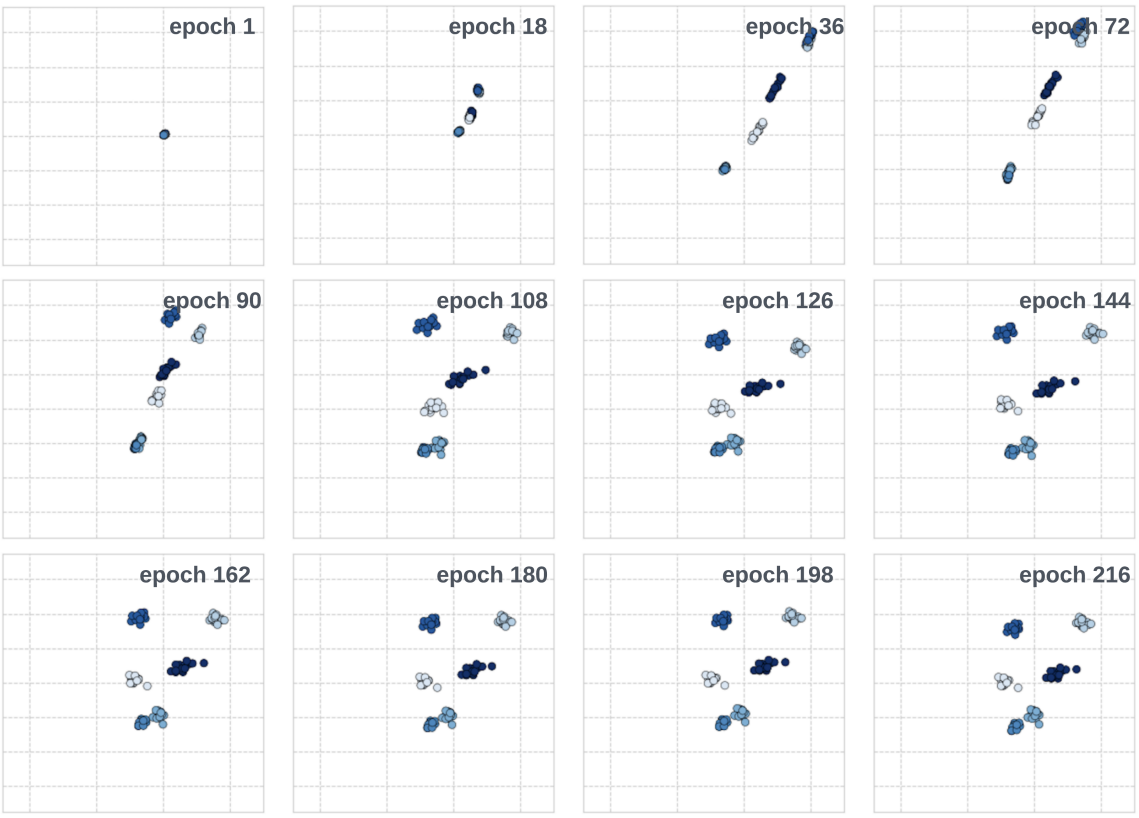

Figure 5.3: Mapping of the ECG training datapoints in the latent space at different training epochs

After training the model, the reconstruction loss (RL), KL divergence (KLD) and mean squared error (MSE) values are

- $RL = 1.0895$
- $KLD = 1.4276$
- $MSE = 1.0912 \times 10^{-4}$

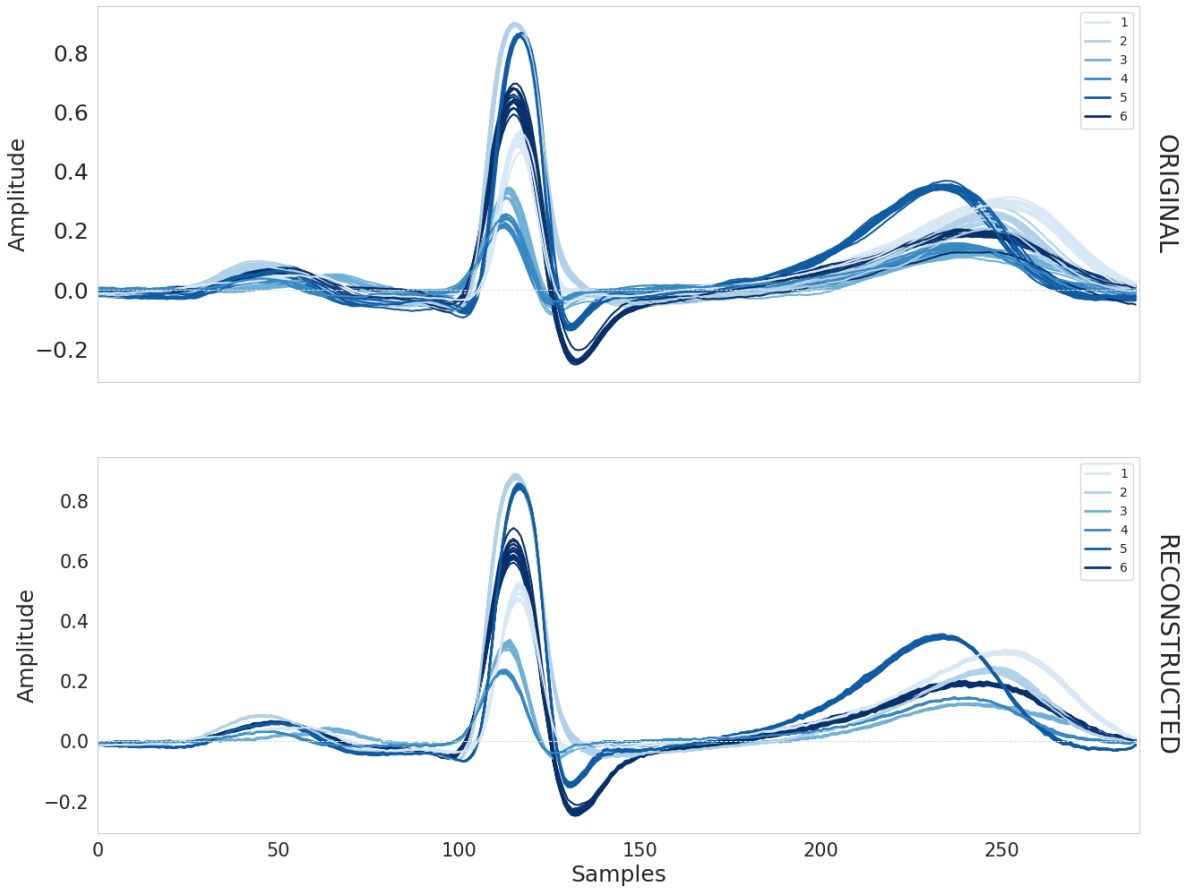

Figure 5.4: Original vs reconstructed ECG training signals. The reconstructed signals are the output of the VAE model given as input the original ones

5.3 Latent space

The figure 5.5 is a plot of the mean values encoded in the latent space for the training data. This representation has mapped the single beat ECG signals of each class into 6 well defined clusters. This exhibits the model capability of properly interpreting the specific class features. Clusters of encoded signals which are closer to each other, represents signals with more similar morphological features. For example, signals from classes 3 and 4 have a much more similar structure with respect to the other classes. From fig 5.4, it can be seen that they present really similar QRS complex and T wave, but some differences in the P wave. The small amount of differences is correctly captured by the model. This can be assessed in two ways: by looking at the reconstructed training signals or, much more clearly, by looking at the latent space, which visibly separates the encoded signals of the two classes in well defined clusters.

Each encoded signal represents a single beat ECG of a specific ECG lead of a specific person. One of the first analysis, done on these data, aims at studying possible time variations of the ECG beats for a given subject during the observation time. This is an essential evaluation to properly interpret and make use of the generated single beat ECG. As an example, if an ECG

measurement is carried on a person which changes motion activity in time, such as walking and running, the ECG measurements would be quite different and the model is expected to identify ECG beats of each motion state and group them in different clusters in the latent space. In this case, it is needed to further label the training data samples, with the motion state at the time of their measurement. This can be achieved also through the VAE model. Both clusters, representing walking and running, can be inspected and then interpreted. Next, the user can decide to generate the person EGC in walking or running state. This proves the power of the VAE interpretable latent space, which allows for the user to generate data with really specific features. Figure 5.6, shows the single ECG beats for each subject, hence for each cluster. Each plot represents only the region of the latent space corresponding to a given cluster; each dot corresponds to an encoded ECG beat. The colours represent the time positioning of each single beat along the ECG trace, ranging from the more recent(lighter) to the latest(darker) ones.

There is no clear evidence of a time correlation between the individual ECG beats of the traces. This is also confirmed by the fact that the measurements were taken on patients with healthy conditions and during a rest state.

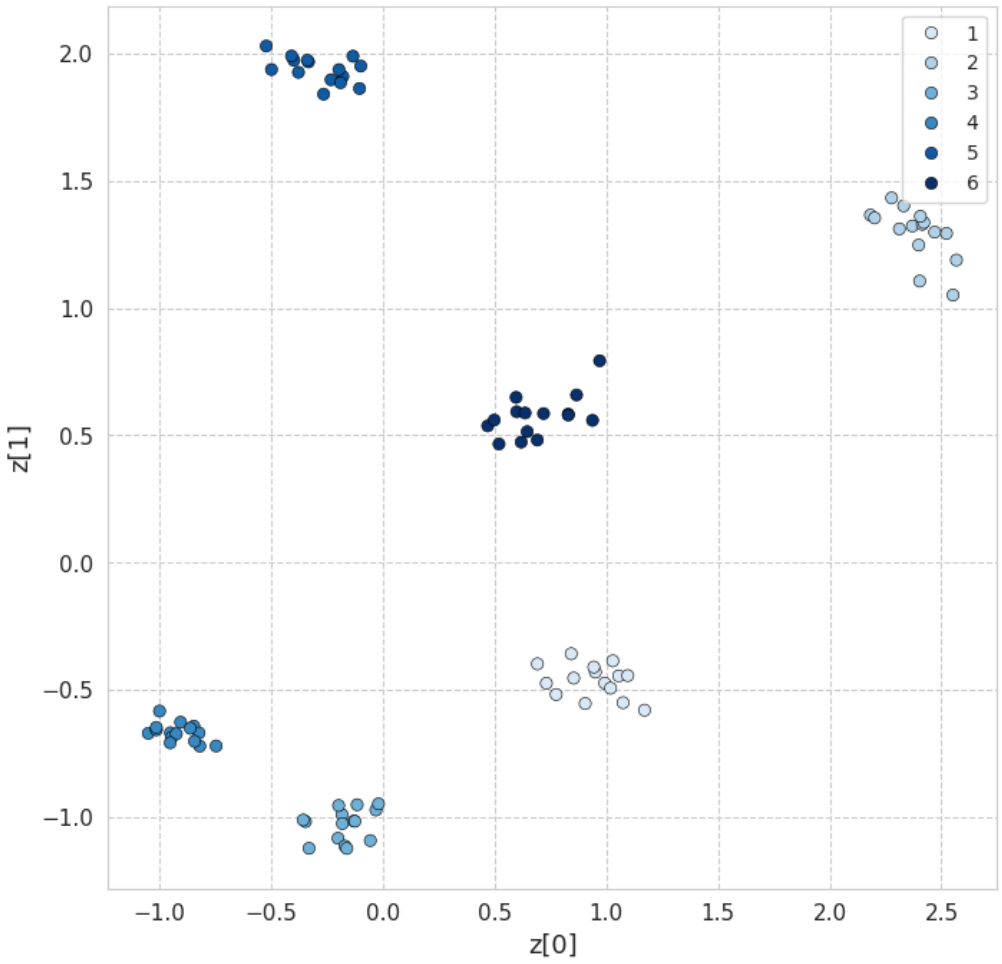

Figure 5.5: Latent space mapped with the single beat ECG encoded training datapoints

Figure 5.6: Each plot represent the latent space cluster of a specified class. The points, representing the single beat ECGs of those classes, are coloured based on the time positioning along the ECG trace, from more recent (lighter) ones to the latest latest (darker) ones

5.3.1 Interpolation

As already explained in section 4.4.2, the trained VAE maps the latent space in a continuous way, by interpolating between the data samples. In fig. 5.7 the points belonging to the colored line are reconstructed by the decoder. This analysis allows to evaluate the interpolation capability of the model in the subregion of space between the clusters of classes 3 and 6. The main differences between the ECG beats of the two classes are the R and S peaks. Class 3 contains signals characterised by a high R and a much smaller S peak, while the other class 6 has a significantly lower R peak and a much more pronounced S peak. The model clearly interpolates those features, reconstructing signals that have features that almost linearly range from the ones of the first class to the ones of the second one.

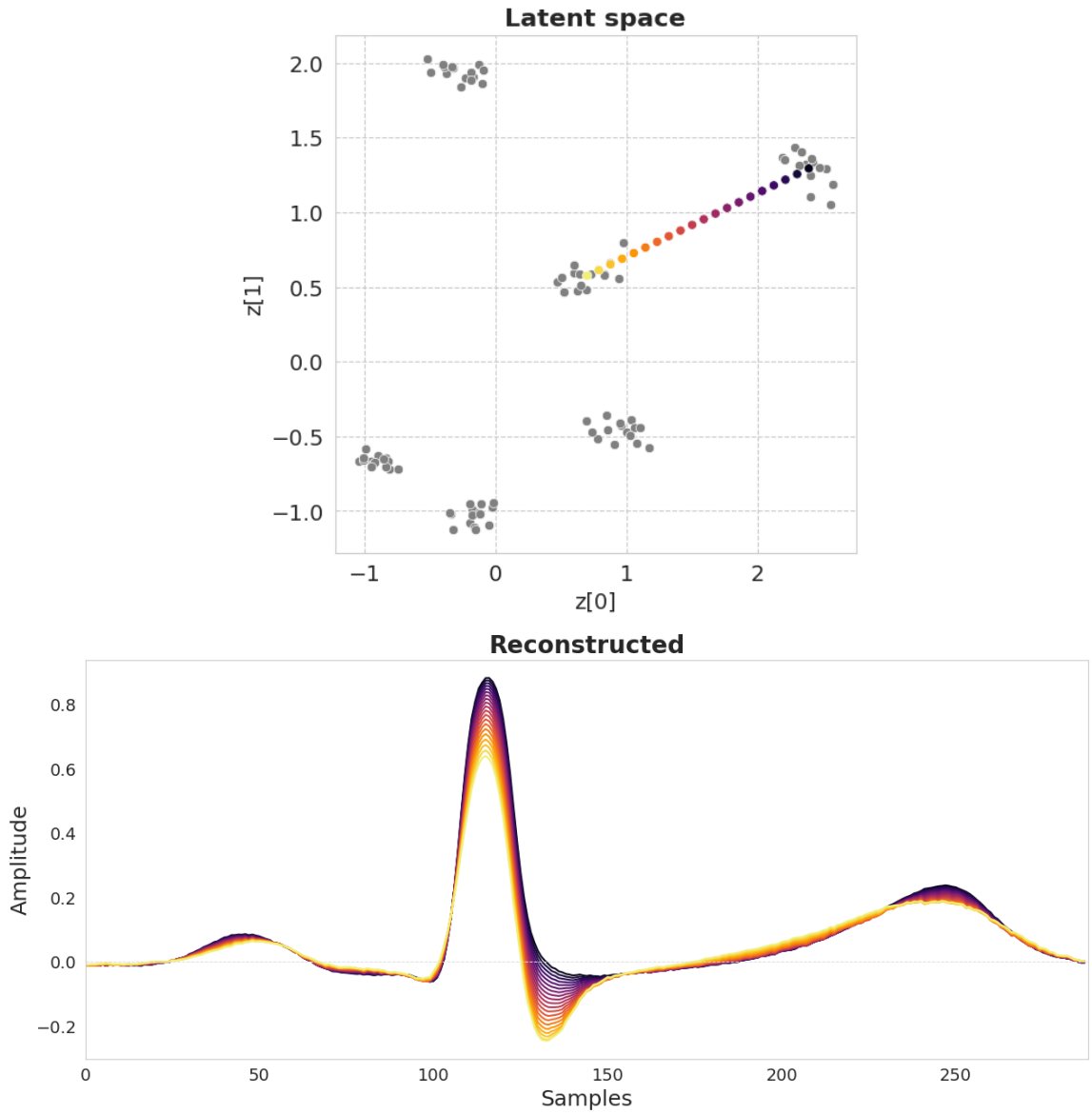

Figure 5.7: Interpolating capabilities of the model on the ECG dataset

5.3.2 Extrapolation

In order to visualise and give an initial assessment of the model extrapolation capabilities, some encoded signals are picked outside the mapped region of the latent space, in particular six points along two concentric circumferences, the inner and outer ones (fig. 5.8).

The reconstructed single beat ECG signals associated to the inner circumference still preserve a good morphological structure, even if the high frequency noise increases for lower amplitude peaks. The reconstructed ones associated to the outer circumference definitely do not maintain intact the morphological ECG structure of an healthy person. The R wave peak is compressed down to the threshold of 1. This is due to the activation function of the last layer of the decoder network, as already discussed in section 4.4.3. The hyperbolic tangent, indeed, has the codomain constrained in the range $[-1, 1]$, hence it does not allow the reconstructed output signal to have

values outside that range. This problem could be solved by training the model choosing a linear activation function. The S peak has increased a lot, while the P peak has widened towards the R peak. The T wave seems to have kept most of the morphological structures, but it is characterised by an high level of high frequency noise. The most probable cause of the scarse extrapolating capability of the proposed VAE model is the low dimensionality of the latent space. A single beat ECG signal is characterised by mainly 5 waves: P, Q, R, S, T. Each wave is characterised by approximately three mean features: the value of the highest peak, its position and the width of the wave itself, for a grant total of 15 features. The model, it was indeed able to encode those features, but only in the mapped region of the latent space, hence it is learned the range of values of those features typical of the data it is trained on. In conclusion, this shows that the developed model generative capabilities are constrained within the mapped region of the latent space. Ideally, a latent space of dimension 15 would be suited to allow the model to extrapolate all the main signal features. For example, an alternative version of the standard VAE, called β-VAE [28] is proven to make the model learn only one specific feature for each dimensions of the latent space. The main drawback is that the higher dimensional space becomes no more visually interpretable.

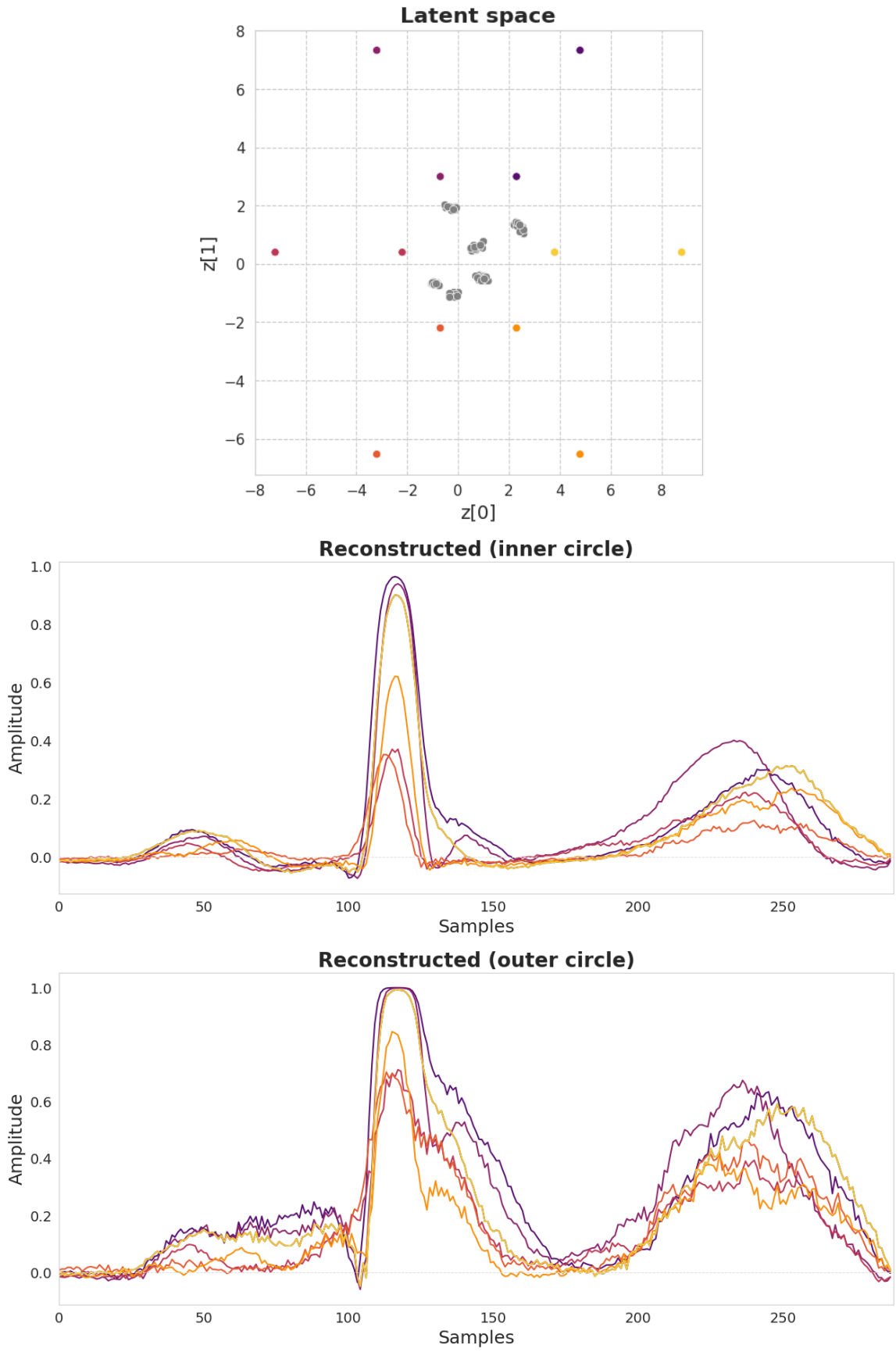

Figure 5.8: Extrapolation capabilities of the VAE on the ECG dataset. Six points are picked around two concentric circles and then decoded, in order to assess the reconstruction capabilities of the model outside the mapped region of latent space.

5.4 Use cases

In this section it is presented the generative use cases of the developed model. The main aim of this thesis project is to increase in size and diversity a really scarce time series dataset. One issue is the high level of class imbalance (fig. 5.1). If the dataset is intended to be used for machine learning applications, it is essential to have a proper level of class balancing, in order to avoid poor performances on minority classes. Balancing the dataset can be achieved in two ways: undersampling and oversampling. The first one consists in randomly removing samples from the majority classes, while the second one in adding samples to the minority classes through a data augmentation technique. The first use case of the developed VAE is oversampling minority classes and achieving dataset balancing.

Another issue is that it contains only few people ECG, which in some cases would not be suited for training a data-driven model since there is not enough data diversity. The second use case of the proposed model regards the possibility to generate data of *synthetic classes* which combines data features of two or more training classes.

5.4.1 Class oversampling

The first use case of the model regards oversampling the minority classes of the dataset or even augmenting all the classes data samples. The process is carried on by using the mapped two dimensional latent space. For simplicity, the datapoints of each cluster are assumed to be distributed as a bivariate gaussian distribution. Given $\mathbf{z} \in \mathbb{R}^2$ the latent space variable

$$f(\mathbf{z}) = \frac{\exp\Big(-\frac{1}{2}(\mathbf{z}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{z}-\boldsymbol{\mu})\Big)}{\sqrt{(2\pi)^2 \det(\boldsymbol{\Sigma})}}$$

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix}$$

It is first chosen the class to be augmented, then the parameters of a bi-variate gaussian distribution are estimated by a Montecarlo estimator. Given M the number of datapoints $\mathbf{z}_j^{(k)}$ associated to the k-th cluster, the mean vector $\boldsymbol{\mu}^{(k)}$ and covariance matrix $\boldsymbol{\Sigma}^{(k)}$ can be estimated as

$$\boldsymbol{\mu}^{(k)} = \frac{1}{M}\sum_{j=1}^{M} \mathbf{z}_j^{(k)}$$

$$\boldsymbol{\Sigma}^{(k)} = \frac{1}{M-1}\sum_{j=1}^{M} (\mathbf{z}_j^{(k)} - \boldsymbol{\mu}^{(k)})(\mathbf{z}_j^{(k)} - \boldsymbol{\mu}^{(k)})^T$$

New datapoints can be sampled from that distribution and reconstructed by the decoder, obtaining synthetic data samples highly correlated to the ones associated to the chosen class.

$$\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}^{(k)}, \boldsymbol{\Sigma}^{(k)})$$

Moreover, by multiplicating the gaussian variance matrix coefficients by a chosen factor α, it can be set the amount of features variability of the reconstructed signals

$$\boldsymbol{\Sigma}^{(k)} = \begin{pmatrix} \alpha \cdot \sigma_{11}^{(k)} & \alpha \cdot \sigma_{12}^{(k)} \\ \alpha \cdot \sigma_{21}^{(k)} & \alpha \cdot \sigma_{22}^{(k)} \end{pmatrix}$$

Figure 5.10 shows three cases in which the value of the multiplicative factor is respectively $\alpha < 1$(top), $\alpha = 1$(middle) and $\alpha > 1$(bottom). Each of the three plot contains the mapped latent space on the left and the reconstructed gaussian samples on the right. Depending on the chosen values of α the reconstructed signal features may vary from the ones of the associated class

- if $\alpha < 1$, then the reconstructed signals tend to have the ”average” class features. In fig. 5.10 , the top plot shows that the generated reconstructed signals are somewhat in between the original training ones.
- if $\alpha = 1$, then the reconstructed signals are expected to be closely like to the ones of the associated class
- if $\alpha > 1$, then the reconstructed signals present some variability from the original signals features of the chosen class. For example, in figure 5.9 the plot on the bottom shows that the R and T waves peak values change significantly. The higher the value of α the greater the changes of features.

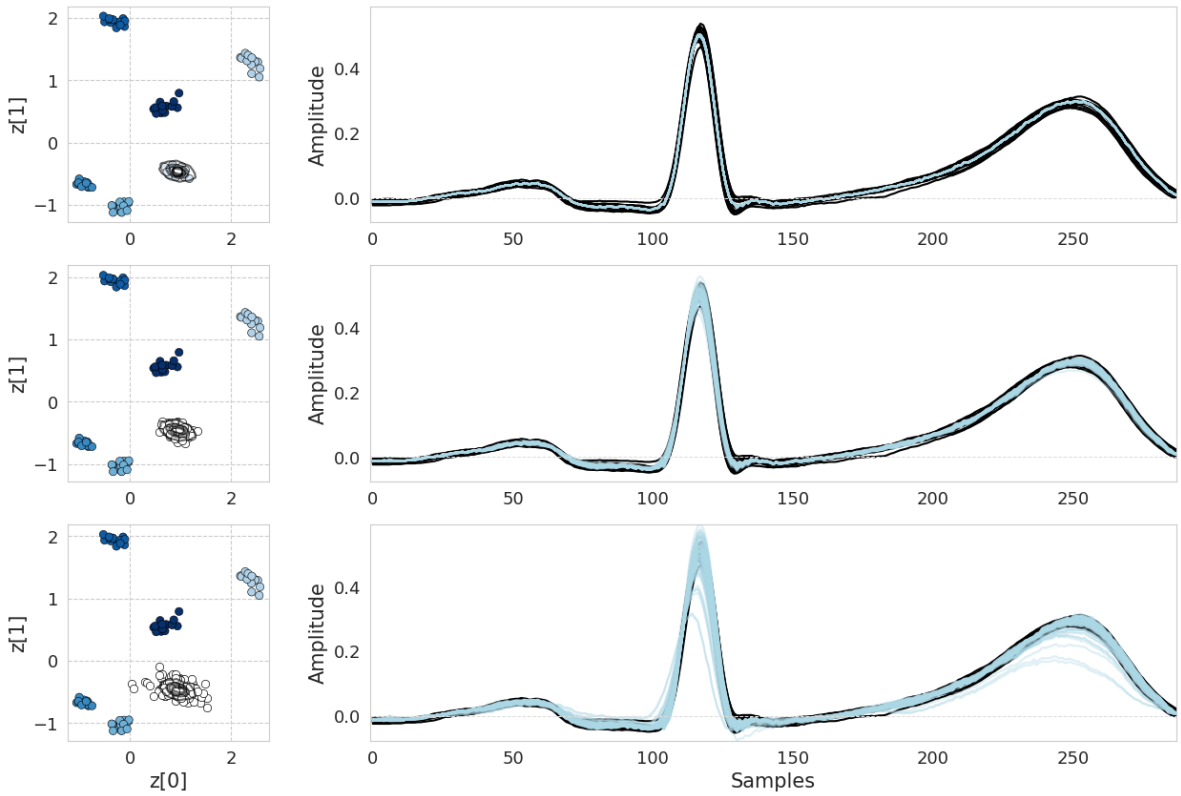

Figure 5.9: Class oversampling on the ECG dataset with varying covariance matrix coefficients. From top to bottom the covariance coefficients are multiplied respectively by $\alpha < 1$, $\alpha = 1$, $\alpha > 1$

In order to evaluate the quality of the generated single beat ECG signals, two metrics are used: the Maximum Mean Discrepancy(MMD) and the Dynamic Time Warping(DTW), both introduced in section 2.9. There are a total of $M = 15$ samples for each class and another 15 samples are generated with the Montecarlo sampling procedure introduced above. The estimated covariance matrix is not modified, hence $\alpha = 1$. Figure 5.10 shows the original training signals(black) and the sampled reconstructed ones(lightblue). For each k-th class the MMD is evaluated between the original training data $\mathbf{x}_o^{(k)}$ and generated $\mathbf{x}_g^{(k)}$ ones.

$$\mathbf{X}_o^{(k)} = \begin{pmatrix} \mathbf{x}_{o,1}^{(k)} \\ \vdots \\ \mathbf{x}_{o,M}^{(k)} \end{pmatrix} \qquad \mathbf{X}_g^{(k)} = \begin{pmatrix} \mathbf{x}_{g,1}^{(k)} \\ \vdots \\ \mathbf{x}_{g,M}^{(k)} \end{pmatrix}$$

$$MMD^{(k)} = MMD(\mathbf{X}_o^{(k)}, \mathbf{X}_g^{(k)})$$

The DTW, on the other hand, for each class, is evaluated with all the combinations of the original and generated signals. The resulting values are then averaged. Given M the total number of both original and generated samples (in this case $J = 15$), then for each k-th class the DTW is

evaluated as

$$DTW^{(k)} = \frac{1}{M^2} \sum_{i=1}^{M} \sum_{j=1}^{M} DTW(\mathbf{x}_{o,i}^{(k)}, \mathbf{x}_{g,j}^{(k)})$$

Finally, for both MMD and DTW, the average is calculated among all the $K = 6$ classes. This evaluation metric is used for keeping track of the quality of the oversampling generative process.

$$MMD_{tot} = \frac{1}{K} \sum_{k=1}^{K} MMD^{(k)}$$

$$DTW_{tot} = \frac{1}{K} \sum_{k=1}^{K} DTW^{(k)}$$

The proposed model led to results of $MMD_{tot} = 5.1 \times 10^{-3}$ and $DTW_{tot} = 0.69$. Those values reach the same order of magnitude of the state-of-art approaches listed in the benchmark table 1.1.

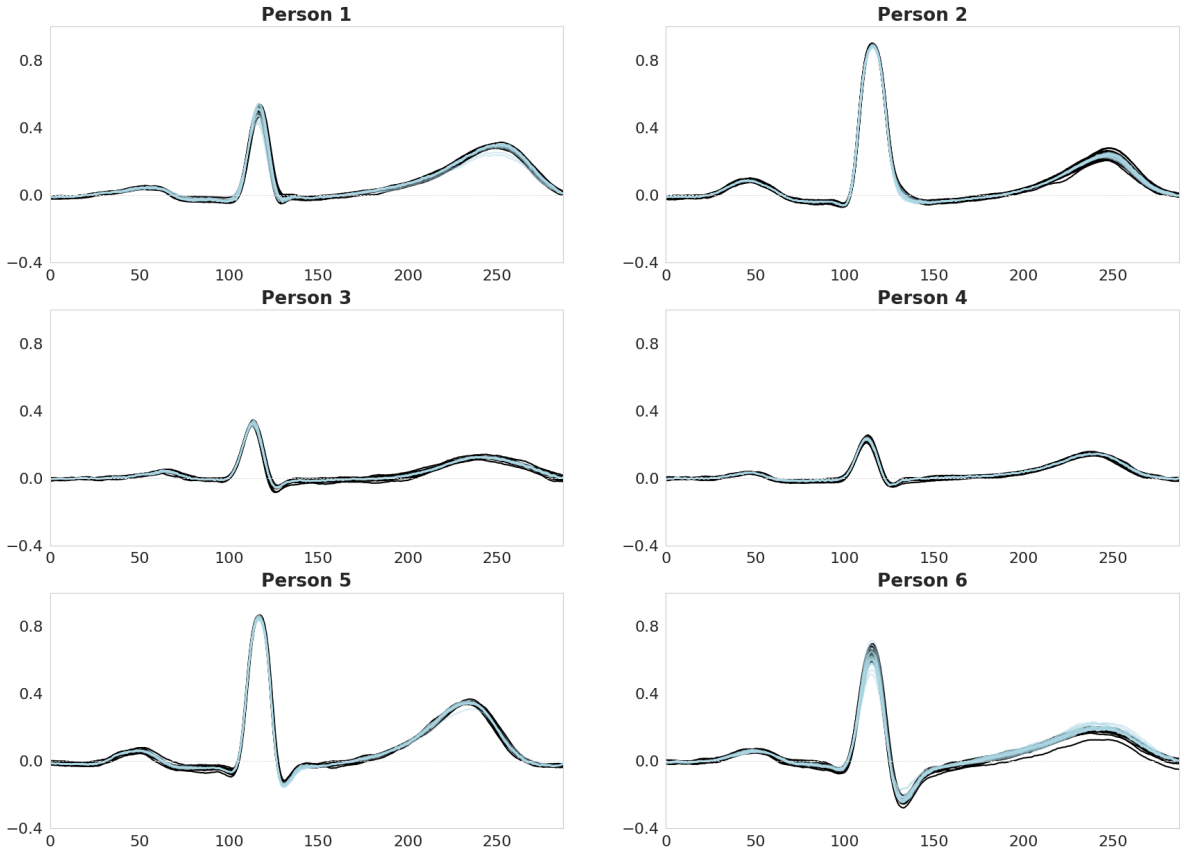

Figure 5.10: Class oversampling carried on for each class of data. For each class they are plotted the original training signals(black) and the generated ones(lightblue). The x-axis represent the sample indices of the signals

5.4.2 Generating a new synthetic class

In the previous sections, it has been shown the interpolating capabilities of the VAE model. Each mapped cluster contains encoded signal with specific features. The empty regions of latent space represent encoded signals which are characterised by a combination of the features associated to the surrounding clusters. Leveraging on the mapped latent space, the user can set a proper vector of means and covariances matrix, and start sampling from the gaussian probability distribution with the chosen parameters. As an example, in figure 5.11, there is a new single beat ECG class which has a mix of the features of classes 2 and 5. The vector means is chosen to be at the same geometric distance from the two surrounding cluster centers. The covariance matrix is chosen by averaging the values of the two surrounding cluster covariance matrices.

Given this setup, the reconstructed signals have a mix of features associated to the two surrounding classes. For this reason, the synthetic class can also be called *child* class, while the classes associated to the nearby clusters can be called *parent* classes. The results are indeed consistent to what was expected. It can be clearly seen how the synthetic signals present peak amplitudes of the R and S wave that stand approximately between the ones associated to the signals of the nearby clusters. The quantitative evaluation is carried on using the metrics MMD and DTW (section 2.9). It is generated an amount of samples for the synthetic class equal to the amount of the surrounding once, hence 15 samples. Next, the DTW and MMD are calculated between the reconstructed samples of the synthetic class and the original ones of respectively the two nearby clusters. It is given $\mathbf{x}_o^{(2)}$ and $\mathbf{x}_o^{(6)}$ the original training signals of respectively classes $k = 2$ and $k = 6$, $\mathbf{x}_s$ the reconstructed signals of the synthetic class and $M = 15$ the total amount of samples of each of the three classes.

$$MMD_s^{(k)} = MMD(\mathbf{X}_o^{(k)}, \mathbf{X}_s)$$

$$DTW_s^{(k)} = \frac{1}{M^2} \sum_{i=1}^{M} \sum_{j=1}^{M} DTW(\mathbf{x}_{o,i}^{(k)}, \mathbf{x}_{s,j})$$

The following results are obtained

$$\begin{aligned} MMD_s^{(2)} &= 1.74 \\ MMD_s^{(6)} &= 1.65 \\ DTW_s^{(2)} &= 1.70 \\ DTW_s^{(6)} &= 1.69 \end{aligned}$$

These values are significantly greater than the MMD obtained when samples of the same clusters are considered ($MMD = 5.1 \times 10^{-3}$ and $DTW = 0.69$), thus confirming that the new generated ECG signals are quite dissimilar to the closer clusters ones.

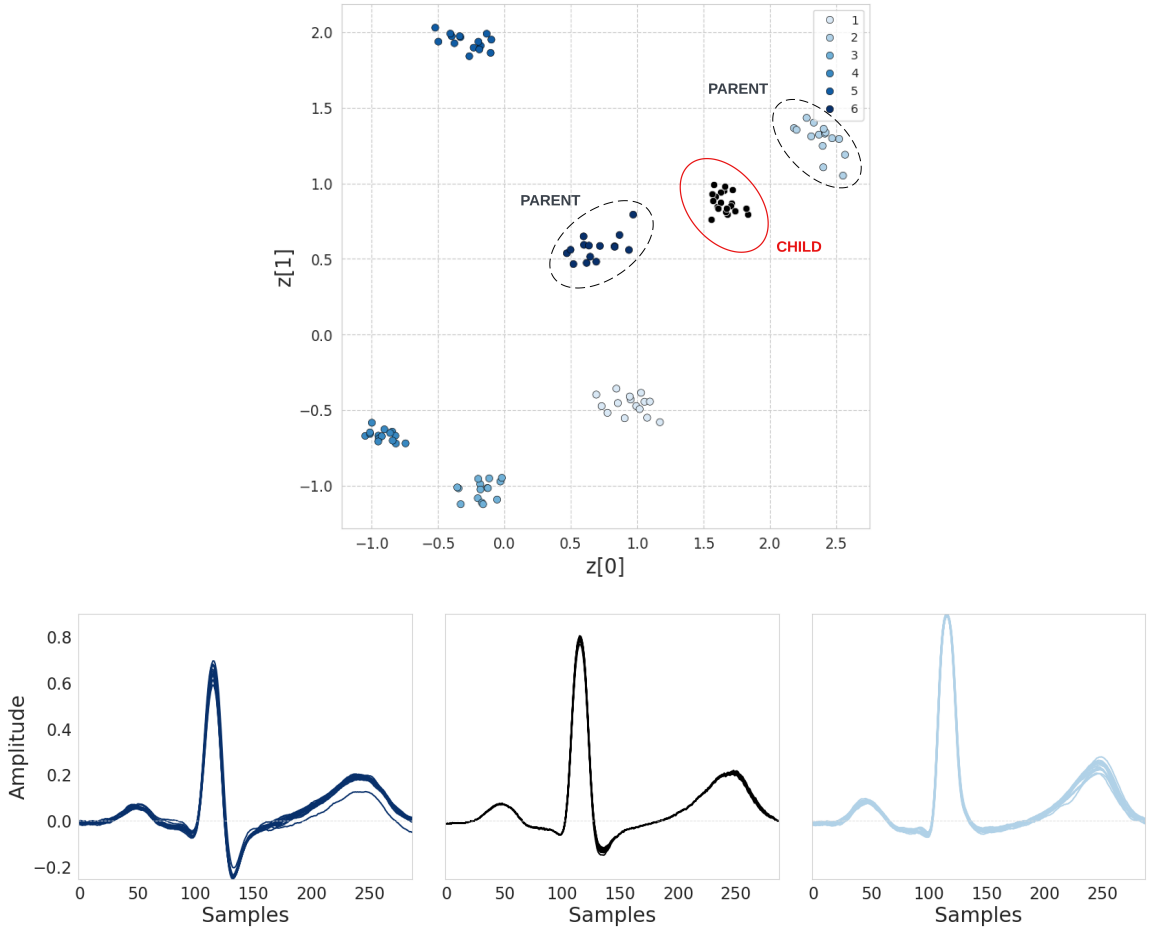

Figure 5.11: The black dots shows the encoded signals of the synthetic class. At the bottom, the plot shows the reconstructed signals of the synthetic class(center) and of the closer clusters(left and right)

Chapter 6

Conclusions and further works

6.1 Conclusions

The main objective of the thesis is to find a way to augment a proposed ECG dataset composed of multiple ECG traces of various healthy patients. The traces contained a varying amount of single beat recordings, making the dataset highly imbalanced. Moreover, the number of recorded ECG beats, as well as the number of patients, is overall really scarce. Existing generative solutions are investigated, leading to the choice of a deep learning based approach. Those kind of models reach stat-of-art performances in generative tasks, but are characterised by a poorly explainable functioning process, acting most of the times as grey-box models. A deep learning generative approach based on a 1D convolutional Variational Autoencoder(VAE) is proposed. This model topology is, indeed, characterised by an interpretable latent space, allowing for a more explainable and controllable generative process. A proper VAE model architecture capable of extracting the dataset signal features is then developed. The latent space dimensions are set to 2, in order to have the highest interpretable encoded data representation. It is first tested the developed model on a sinusoid dataset, where the main features and working principles are explained. Next, the model is applied to the ECG dataset. After pre-processing the data and training the network, the latent space is mapped with the training ECG signals. The mapping shows that the model is able to clearly identify the specific class(patient) ECG features, since each class datapoints are grouped in well defined clusters. The generative process is carried on by leveraging on the mapped latent space and the decoder network. The proposed solution allows for two main applications: oversampling an already existing class and generating samples of a new *synthetic* class, by making use of a generative process highly explainable and controllable. The results are qualitatively evaluate using the Maximum Mean Discrepancy(MMD) and Dynamic Time Warping(DTW) metrics (tab. 6.1), obtaining values close to the current state-of-art (tab. 1.1).

Application	Metric	Value
class oversampling	MMD	5.12×10^{-3}
	DTW	0.698
synthetic class	$MMD_s^{(2)}$	1.74
	$MMD_s^{(6)}$	1.65
	$DTW_s^{(2)}$	1.70
	$DTW_s^{(6)}$	1.69

Table 6.1: Evaluation metrics results

In section 1.2, three main XAI concept have been introduced: *transparency*, *interpretability* and *explainability*. It is now shown how these concepts apply for the developed VAE model.

The first one, transparency, refers to the ability of a model and its inner working principles to be understood, if not formally and quantitatively , at least intuitively. The VAE encoder and decoder components are built up with neural network architectures, which are above the less explainable models. However, most of the neural networks layers are of type 1D convolutional, which are characterised by a well defined principle of functioning, the mathematical operation of convolution, and have a transparent output. By looking at the output of each convolutional filter it can be intuitively assessed the model ability to capture the signal features. Each output filter represents a feature map that contains specific features of the output signal. By reducing the dimensionality of the output at each 1D convolutional layers, the features start to be encoded into a lower dimensional space, hence only the most prominent features are sent to the subsequent layers, until reaching the fully connected layers. Those layers are above the less explainable ones, but they are necessary to map a function from the given input to the parameters of the bivariate gaussian distribution of the sampling layer.

To better understand the working process, it is chosen as test signal a single ECG beat from the training dataset (fig. 6.1). The outputs of the first two convolutional layer, working in parallel, respectively with a kernel size of 11 and 5, are plotted in figure 6.2 and 6.3. Each convolutional layer has 32 filter, meaning that there are 32 feature maps. It is clear how the model captures various features with each filter. For example it is clear how some filters remove the less relevant component and outputs only the main R and T waves, while some others do the opposite. This allows to have an insight of the intricate behaviour of the network feature extraction process and it can be useful both during the network architecture design and the evaluation phase. Figure 6.4, instead, plots the feature map of the last convolutional layer of the encoder. Due to the low dimensionality, the encoder creates a feature map of only the most relevant features, which usually are difficult to be understood by a human operator. In conclusion, the shallow layers are return the clearer outputs, while the deeper ones, due to their low dimensionality nature, compress the feature informations in such a way that is difficult to understand. During the network design and evaluation phase, the shallow convolutional layers

give the best insights about the network working mechanisms.

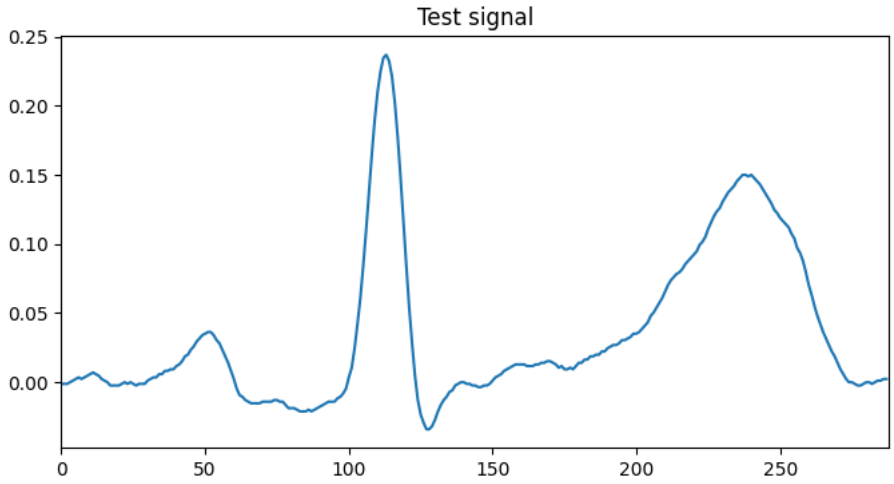

Figure 6.1: Test signal, randomly picked from the ECG training dataset

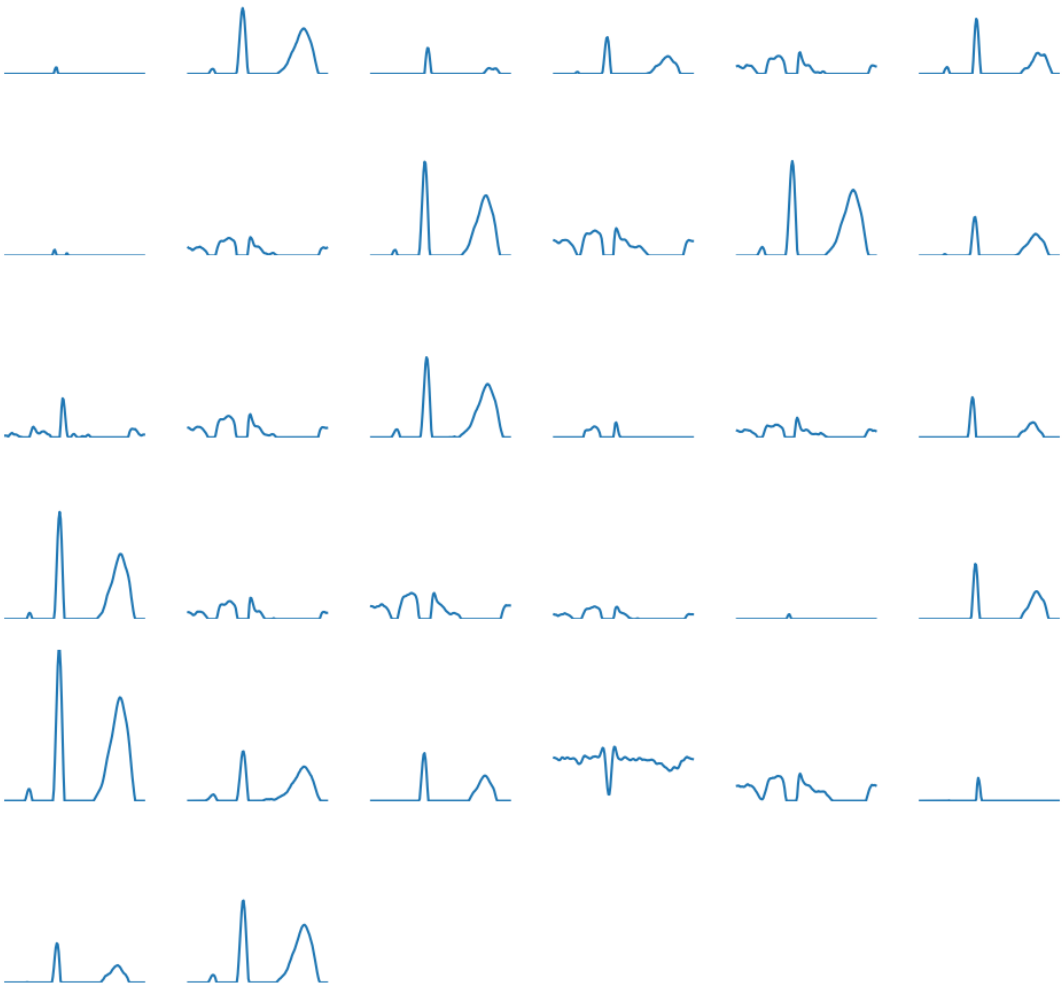

Figure 6.2: Feature maps (32) of the input convolutional layer with kernel size 11 of the encoder

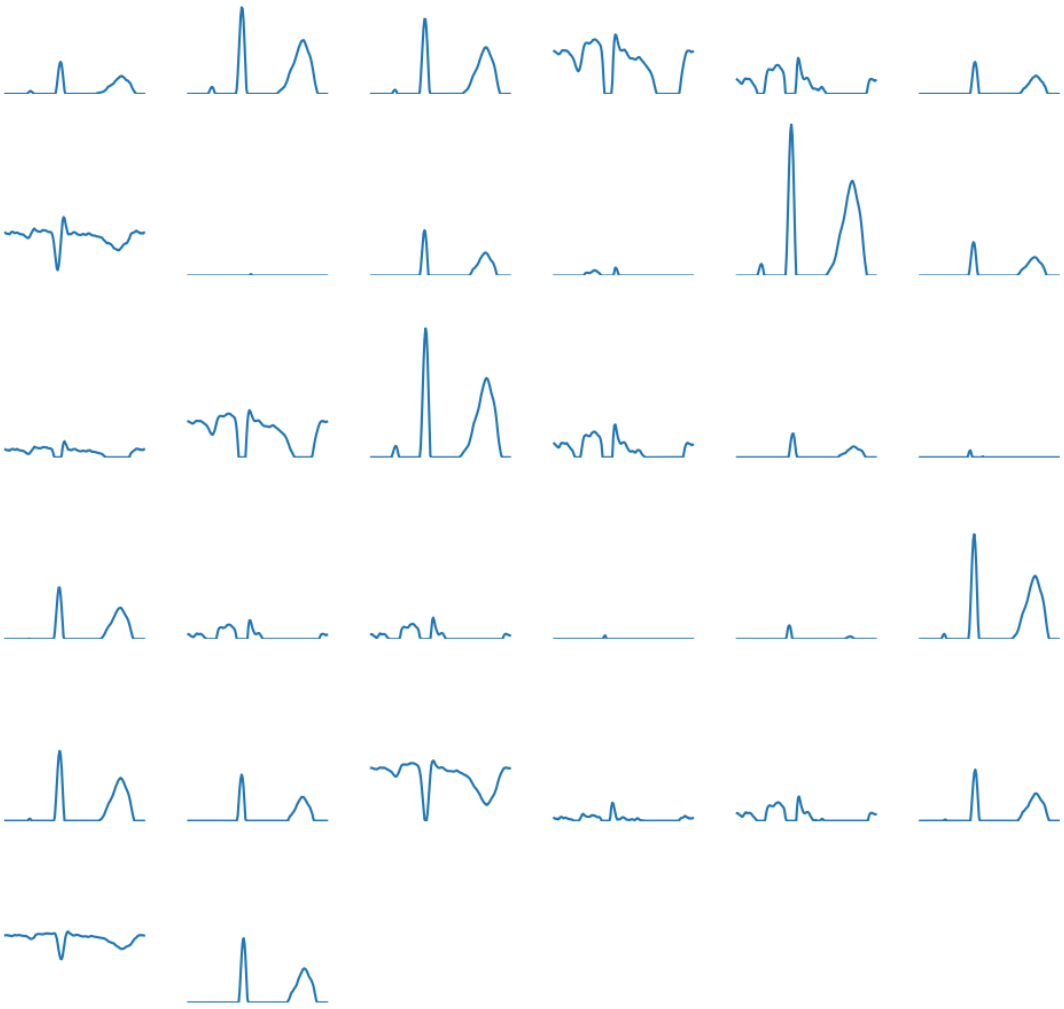

Figure 6.3: Feature maps (32) of the input convolutional layer with kernel size 5 of the encoder

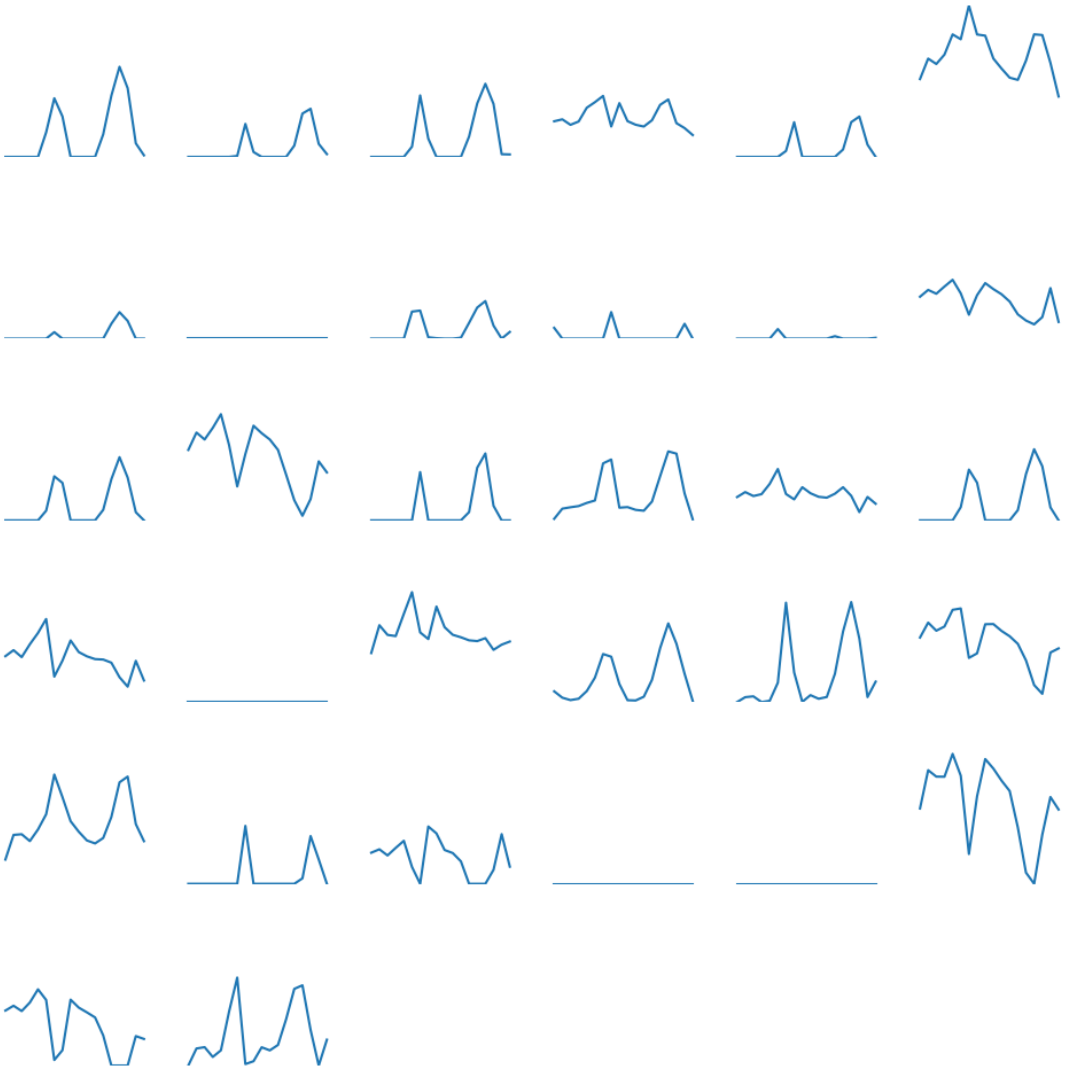

Figure 6.4: Feature maps (32) of the last convolutional layer of the encoder

Interpretability, on the other hand, refers to the ability of the model to make clear the reason why a specific input results in a specific output. In other words, it should allow the user to estimate the behaviour of the model given the specific input. Once the proposed VAE is trained, the generative part is carried on using only the decoder. The latent space allows for a clear understanding of what to expect at the decoder output given a specific input(a point in the latent space). After training the model, the training data samples can be encoded in the latent space through the encoder network. Since its dimensionality is equal just to 2, the encoded data are clearly visualised(fig. 5.5). Moreover, if the network architecture is well suited for the type of training data, it should capture the features of each training class and properly map those samples into clusters which occupy a well defined subregion of space. Due to the interpolating capabilities of the model, the empty regions of latent space, not mapped by the training data points, are also interpretable. As an example, given two clusters, by taking a value at the same geometric distance between their centers, the decoder reconstructs a signal having a mix of features typical of the datapoints of the two classes. By moving closer to one of the clusters, the

features of the associated class would become more prominent in the reconstructed signal(fig. 5.7). Once the model has been properly trained and the latent space mapped with the training data points, the generative process that relies on the decoder network becomes interpretable.

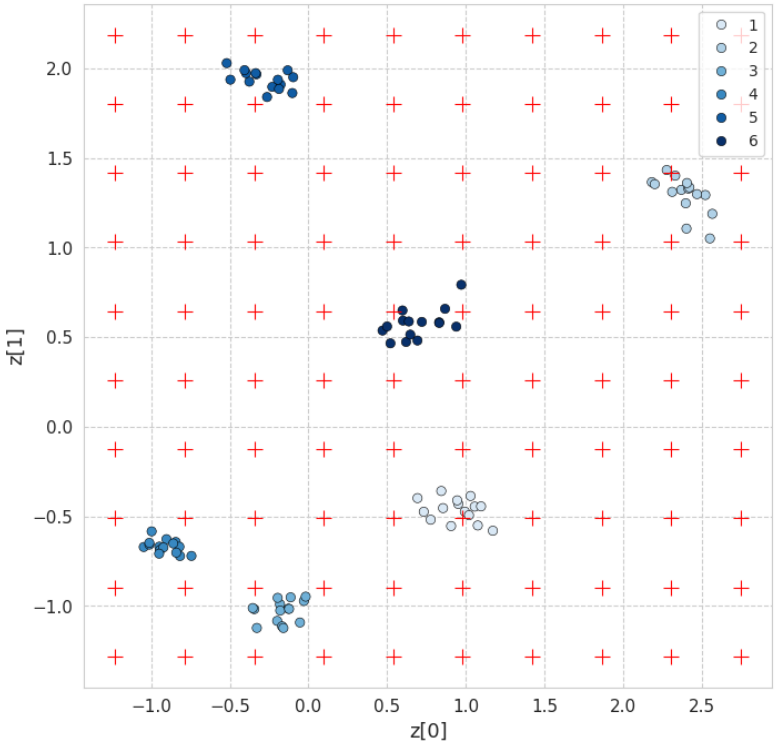

Figure 6.5: Latent space representation of the training ECG signals. The red cross indicates the encoded signals that are reconstructed by the decoder in the following figure.

Figure 6.6: Reconstructed signals. Each plot is a reconstructed signal associated to the encoded signal on the same position in the figure above. For example, the top-left plot is the reconstructed signal of the top-left red cross datapoint in the previous figure.

Finally, explainability, extends the interpretability concept encompassing also a well defined human-computer interaction. The developed model has automated only the training of the network, during which the weights of the decoder network are properly tuned based on the training data. Once the model is trained and the latent space mapped, the decoder becomes a toll that, with some approximation, allows the user to choose the characteristics of the single beat ECG signal to be generated. The generative process is carried on with an high presence of human intervention. As an example, the first model application explained in section 5.4.1, allows the user to set for which class generating new data samples, the amount of data samples to be generated and their feature variability by properly setting the factor parameter α. The human user control over the generative process happens also in the second application (section

5.4.2), when generating samples from a brand new *synthetic class*. The vector means and the covariance matrix are, in fact, set by the user.

6.2 Further works

The proposed model could be further improved by automating the process of sampling from a *synthetic* class. The user should be allowed to choose from which class signals mix the features into the *synthetic* class, then a proper gaussian distribution mean vector and covariance matrix should be automatically set. The user can then choose the number *synthetic* class signals to sample.

The sinusoid test application showed that the model is also able to capture the features typical of sinusoidal signals. In general, the model can be trained on the most various kind of time series, with the constraint of a fixed sample length(in this case 288 samples). The only limitation is the amount and complexity of the time series features to be capture. The more complex the dataset signals are, the more features the model needs to learn. Proportionally, the neural network architecture complexity must be increased in depth(higher number of layers) and/or in width(higher number of convolutional layer filters and/or fully connected layer neurons). This, however, is rarely the case when working with topic specific signals and with a low number of classes. This includes also other bio signals measurements, such as Electroencephalogram (EEG), Photoplethysmogram (PPG), Force and Pressure Measurements, Gait Analysis, and so on. Further work can be carried out to further test and validate this hypothesis.

the model can indirectly be used for other relevant tasks. As an example, the inner encoding working principle can be use to map the input time series into a lower dimensional representation for the purpose of data visualisation. The feature extraction capabilities of the model allows to use it also as a classifier. Figure 4.6 and 5.5 shows how the model clearly capture the specific features of each training class, since it maps the class datapoints in well defined region of the latent space.

Moreover, since the encoder is trained to map the input data into a lower dimensional space, it can also be used for data compression purposes. The compression would be lossy, but potentially with a very low loss of information (see evaluation metric results in section 4.3 and 5.2). Compared with a deterministic autoencoder model, the interpolation capabilities allow for a better model generalization. Moreover, by properly setting the latent space dimension, the VAE tend to disentangle the training data features along each dimension [28] . As an example, one dimension could encode the amplitude of the R wave peak of the single beat ECG signal. This behaviour leads to an improvement of the model extrapolation capabilities, therefore a further improved model generalization.

All this applications can be further investigated in future works.

Chapter 7

Appendix

7.1 Leibniz integral rule

Theorem – Let $f(x,t)$ be a function such that both $f(x,t)$ and its partial derivative $f_x(x,t)$ are continuous in t and x in some region of the xt-plane, including $a(x) \leq t \leq b(x)$, $x_0 \leq x \leq x_1$. Also suppose the functions $a(x)$ and $b(x)$ are both continuous and both have continuous derivatives for $x_0 \leq x \leq x_1$. Then , for $x_0 \leq x \leq x_1$,

$$\frac{d}{dx}\left(\int_{a(x)}^{b(x)} f(x,t)dt\right) = f(x,b(x)) \cdot \frac{d}{dx}b(x) - f(x,a(x)) \cdot \frac{d}{dx}a(x) + \int_{a(x)}^{b(x)} \frac{\partial}{\partial x}f(x,t)dt$$

If the integral extremes are independent from the derivation variable then

$$\frac{d}{dx}\left(\int_{a}^{b} f(x,t)dt\right) = \int_{a(x)}^{b(x)} \frac{\partial}{\partial x}f(x,t)dt$$

7.2 Law of the unconscious statistician

Theorem – Let x be a discrete (or continuous) random variable and let f_x be its probability density function. Given y a new random variable such that $y = h(x)$, where h is a deterministic function, then

$$\text{discrete:} \quad \mathbb{E}[y] = \mathbb{E}[h(x)] = \sum_x h(x) f_x(x)$$

$$\text{continuous:} \quad \mathbb{E}[y] = \mathbb{E}[h(x)] = \int_{-\infty}^{\infty} h(x) f_x(x) dx$$

7.3 Reparameterization trick: generic example

Given x, a continuous random variable following the parametrised distribution $x \sim p_\theta(x)$ and $f(x)$ a deterministic function of x, it is wanted to calculate the gradient with respect to θ of the

expected value of $f(x)$

$$\begin{aligned}
\nabla \mathbb{E}_{x\sim p_\theta}[f(x)] &= \nabla_\theta \int_{-\infty}^{\infty} f(x) p_\theta(x) dx \\
&= \nabla_\theta \int_{-\infty}^{\infty} f(g(\epsilon,\theta)) p(\epsilon) d\epsilon \\
&= \int_{-\infty}^{\infty} \nabla_\theta f(g(\epsilon,\theta)) p(\epsilon) d\epsilon \\
&\simeq \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta f(g(\epsilon^{(i)},\theta)) p(\epsilon)
\end{aligned}$$

In the first step it is applied the *law of the unconscious statistician* and in the second one the *reparameterization trick* such that $x = g(\epsilon, \theta)$ where $\epsilon \sim p(\epsilon)$ and g is a deterministic function. Finally, in the third step the *Leibniz integral rule* is applied. Since the integral is most of the time intractable, a basic Monte Carlo estimator can be used for its practical evaluation.

7.4 VAE KL divergence: gaussian case

It is considered both the prior $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ and the posterior $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ distributions as Gaussian. Let $J = C(\mathbf{z})$ be the dimensionality of $\mathbf{z}$. Let $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ denote the variational mean and s.d. evaluated at the i-th datapoint, and let μ_j and σ_j simply denote the j-th element of these vectors. Both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are functions of the encoder network weights $\boldsymbol{\phi}$ and the input data $\mathbf{x}$.

$$\begin{aligned}
D_{KL}(q_\phi(z|x) \,||\, p_\theta(z)) &= \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z)}\right] \\
&= \int_{-\infty}^{\infty} \log\left[\frac{q_\phi(z|x)}{p_\theta(z)}\right] q_\phi(z|x) dz \\
&= \underbrace{\int_{-\infty}^{\infty} \log\left[q_\phi(z|x)\right] q_\phi(z|x) dz}_{(1)} - \underbrace{\int_{-\infty}^{\infty} \log\left[p_\theta(z))\right] q_\phi(z|x) dz}_{(2)}
\end{aligned}$$

Solving (1)

$$\int_{-\infty}^{\infty} \log\left[q_\phi(z|x)\right] q_\phi(z|x)dz = (*)$$

$$(*) = \int_{-\infty}^{\infty} \left[-\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_j^2 - \frac{1}{2}\sum_{j=1}^{J}\frac{(z_j-\mu_j)^2}{\sigma_j^2}\right] q_\phi(z|x)dz$$

$$= -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_j^2 - \frac{1}{2}\sum_{j=1}^{J}\int_{-\infty}^{\infty}\frac{(z_j-\mu_j)^2}{\sigma_j^2}q_\phi(z|x)dz$$

$$= -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_j^2 - \frac{1}{2}\sum_{j=1}^{J}\int_{z_j}\frac{(z_j-\mu_j)^2}{\sigma_j^2}q_\phi(z_j|x)\cdot$$

$$\cdot \underbrace{\int_{z_i\neq z_j}\Pi_{i\neq j}q_\phi(z_i|x)dz_{i\neq j}}_{=1}\, dz_j$$

$$= -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_j^2 - \frac{1}{2}\sum_{j=1}^{J}\frac{1}{\sigma_j^2}\int_{-\infty}^{\infty}(z_j^2 - 2z_j\mu_j + \mu_j^2)^2 q_\phi(z_j|x)dz_j$$

$$= \frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_j^2 - \frac{1}{2}\sum_{j=1}^{J}\frac{1}{\sigma_j^2}(\mathbb{E}_\phi[z_j^2|x] - 2\mu_j\,\mathbb{E}_\phi[z_j|x] + \mu_j^2)$$

$$= \frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\log\sigma_j^2 - \frac{1}{2}\sum_{j=1}^{J}\underbrace{\frac{1}{\sigma_j^2}(\mu_j^2 + \sigma_j^2 - 2\mu_j^2 + \mu_j^2)}_{=1}$$

$$= \frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(1 + \log\sigma_j^2)$$

where in the second to last equation it has simply used the definition of variance $\sigma^2(x) = \mathbb{E}[x^2] - \mathbb{E}[x]^2$. Solving (2) requires the same tools already used in (1)

$$\begin{aligned}
\int_{-\infty}^{\infty} \log\left[p_\theta(z))\right] q_\phi(z|x)dz &= \int_{-\infty}^{\infty} \left[-\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J} z_j^2\right] q_\phi(z|x)dz \\
&= -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\int_{-\infty}^{\infty} z_j^2 q_\phi(z|x)dz \\
&= -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\int_{z_j} z_j^2 q_\phi(z_j|x) \underbrace{\int_{z_i \neq z_j} q_\phi(z_i|x)dz_{i\neq j}}_{=1} dz_j \\
&= \frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}\mathbb{E}_\phi[z_j^2|x] \\
&= \frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(\sigma_j^2 + \mu_j^2)
\end{aligned}$$

Combining term (1) and (2), the closed form of the KL divergence term becomes

$$D_{KL}(q_\phi(z|x)||p_\theta(z)) = -\frac{1}{2}\sum_{j=1}^{J}(1 + \log\sigma_j^2 - \mu_j^2 - \sigma_j^2)$$

Bibliography

[1] Thampi. (2022). Interpretable AI: Building explainable machine learning systems. Simon and Schuster, 14-15.

[2] Burrell, J. (2016). How the machine 'thinks': Understanding opacity in machine learning algorithms. Big data & society, 3(1)

[3] Miller, T. H. (2017). Explainable AI: Beware of inmates running the asylum or: How I learnt to stop worrying and love the social and behavioural sciences. arXiv preprint arXiv:1712.00547.

[4] Lisboa, P. J. (2013). Interpretability in machine learning–principles and practice. International Workshop on Fuzzy Logic and Applications. Cham: Springer International Publishing., 15-21.

[5] Lepri, B. O. (2018). Fair, transparent, and accountable algorithmic decision-making processes: The premise, the proposed solutions, and the open challenges. Philosophy & Technology, 31, 611-627

[6] Gunning, D. and Aha, D.W. (2019), DARPA's Explainable Artificial Intelligence Program. AI Magazine, 40: 44-58.

[7] Miao F, Wen B, Hu Z, Fortino G, Wang XP, Liu ZD, Tang M, Li Y. Continuous blood pressure measurement from one-channel electrocardiogram signal using deep-learning techniques. Artif Intell Med. 2020 June.

[8] Lu P, Gao Y, Xi H, Zhang Y, Gao C, Zhou B, et al. KecNet: a light neural network for arrhythmia classification based on knowledge reinforcement. J Healthc Eng 2021 April.

[9] Chiu JK, Chang CS, Wu SC. ECG-based biometric recognition without QRS segmentation: a deep learning-based approach.In: Proceedings of the 2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society(EMBC). 2021.

[10] Ozdemir MA, Ozdemir GD, Guren O. Classification of COVID-19 electrocardiograms by using hexaxial feature mapping and deep learning. BMC Med Inform Decis Mak. 2021 May.

[11] Baghersalimi S, Teijeiro T, Atienza D, Aminifar A. Personalized real-time federated learning for epileptic seizure detection. IEEE J Biomed Health Inform 2022 Feb

[12] Kuznetsov, V. V., Moskalenko, V. A., & Zolotykh, N. Yu. Electrocardiogram Generation and Feature Extraction Using a Variational Autoencoder. 2020.

[13] Ryo Nishikimi, Masahiro Nakano, Kunio Kashino, Shingo Tsukada, Variational autoencoder–based neural electrocardiogram synthesis trained by FEM-based heart simulator, Cardiovascular Digital Health Journal, Volume 5, Issue 1, 2024.

[14] Y. Sang, M. Beetz and V. Grau, "Generation of 12-Lead Electrocardiogram with Subject-Specific, Image-Derived Characteristics Using a Conditional Variational Autoencoder," 2022 IEEE 19th International Symposium on Biomedical Imaging (ISBI), Kolkata, India, 2022, pp. 1-5, doi: 10.1109/ISBI52829.2022.9761431.

[15] Delaney, A. M., Brophy, E., & Ward, T. E. . Synthesis of Realistic ECG using Generative Adversarial Networks. 2019.

[16] Golany, Tomer & Lavee, Gal & Yarden, Shai & Radinsky, Kira. Improving ECG Classification Using Generative Adversarial Networks. Proceedings of the AAAI Conference on Artificial Intelligence. 2020.

[17] E. Adib, A. S. Fernandez, F. Afghah and J. J. Prevost, "Synthetic ECG Signal Generation Using Probabilistic Diffusion Models," in IEEE Access, vol. 11, pp. 75818-75828, 2023.

[18] Hazra D, Byun YC. SynSigGAN: Generative Adversarial Networks for Synthetic Biomedical Signal Generation. Biology (Basel). 2020 December.

[19] Ian Goodfellow, Yoshua Bengio, & Aaron Courville (2016). Deep Learning. MIT Press.

[20] Müller, Meinard. Dynamic time warping. Information Retrieval for Music and Motion. 2. 69-84, 2007.

[21] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Scholkopf, & Alexander Smola. A Kernel Two-Sample Test. Journal of Machine Learning Research, 13(25), 723-773. (2012)

[22] Boyd Stephen, Vandenberghe Lieven, "Convex Optimization", Cambridge, Cambridge University Press, 2004.

[23] B.T. Polyak, "Some methods of speeding up the convergence of iteration methods", USSR Computational Mathematics and Mathematical Physics, Volume 4, Issue 5, 1964.

[24] Yurii Nesterov, A method for solving the convex programming problem with convergence rate O(1/k2), Dokl. Akad. Nauk SSSR, Vol. 269, 1983.

[25] Duchi, J., Hazan, E. & Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal Of Machine Learning Research*. **12**, 2121-2159 (2011)

[26] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, & Yoshua Bengio. (2014). Generative Adversarial Networks.

[27] Kingma, D. & Ba, J. Adam: A Method for Stochastic Optimization. (2017)

[28] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, & Alexander Lerchner (2017). beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework. In International Conference on Learning Representations.

[29] Diederik P Kingma, & Max Welling. (2022). Auto-Encoding Variational Bayes.

[30] Kingma, D., & Welling, M. (2019). An Introduction to Variational Autoencoders. Foundations and Trends in Machine Learning.

[31] Sebastian Ruder. (2017). An overview of gradient descent optimization algorithms.

[32] Iglesias, G., Talavera, E., Gonzalez-Prieto, Mozo, A., & Gomez-Canaval, S. (2023). Data Augmentation techniques in time series domain: a survey and taxonomy. Neural Computing and Applications.

[33] Shenda Hong, Yuxi Zhou, Junyuan Shang, Cao Xiao, & Jimeng Sun. (2020). Opportunities and Challenges of Deep Learning Methods for Electrocardiogram Data: A Systematic Review.

[34] Rahman MM, Rivolta MW, Badilini F, Sassi R. A Systematic Survey of Data Augmentation of ECG Signals for AI Applications. Sensors. 2023.

[35] Keiron O'Shea, & Ryan Nash. (2015). An Introduction to Convolutional Neural Networks.

[36] Albawi, S., Mohammed, T., & Al-Zawi, S. (2017). Understanding of a convolutional neural network. In 2017 International Conference on Engineering and Technology (ICET) (pp. 1-6).

[37] Zhao, B., Lu, H., Chen, S., Liu, J., & Wu, D. (2017). Convolutional neural networks for time series classification. Journal of Systems Engineering and Electronics, 28(1), 162-169.

[38] Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. Nature 323, 533–536 (1986).

[39] Hecht-Nielsen (1989). Theory of the backpropagation neural network. In International 1989 Joint Conference on Neural Networks.

[40] Obaid, H., Dheyab, S., & Sabry, S. (2019). The Impact of Data Pre-Processing Techniques and Dimensionality Reduction on the Accuracy of Machine Learning.

[41] Balderas, L., Lastra, M., & Benítez, J. (2024). Optimizing dense feed-forward neural networks.