DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

CATARINA DE ALMEIDA FERNANDES E ANDRADE BENTO

BSc in Computer Science and Engineering

CONDITIONAL DEEP GENERATIVE MODELS FOR MEMS DEVICES DESIGN

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CONDITIONAL DEEP GENERATIVE MODELS FOR MEMS DEVICES DESIGN

CATARINA DE ALMEIDA FERNANDES E ANDRADE BENTO

BSc in Computer Science and Engineering

Adviser: David Semedo
Assistant Professor, NOVA School of Science and Technology

Co-adviser: Chen Wang
Postdoctoral Researcher, KU Leuven University

Examination Committee

Chair: Vítor Duarte
Assistant Professor, NOVA School of Science and Technology

Rapporteur: Rui Jesus
Coordinating Professor, Lisbon School of Engineering

Member: David Semedo
Assistant Professor, NOVA School of Science and Technology

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

Conditional Deep Generative Models for MEMS Devices Design

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.27) [32].

*T*o my family and boyfriend. I hope all the efforts and sacrifices that were necessary to complete this project, and this course can help my future self

Acknowledgements

First and foremost, I would like to express my gratitude to my advisor, Professor David Semedo, whose guidance, support, availability, and motivation were fundamental for developing and completing this thesis. I would also like to thank my co-advisor, Researcher Chen Wang, and the entire KU Leuven team for their collaboration and support throughout this work.

I extend my sincere thanks to Professor João Magalhães and the NOVA Vision & Language Research Group for providing the critical infrastructure and technical assistance necessary for this research.

A special acknowledgement goes to my colleague and dear friend, Luís Tripa, for his companionship, support, and motivation during this shared journey. I am also immensely thankful to all my friends and colleagues from NOVA, particularly my fellow *Merendeiros*, for the memorable moments we've experienced together over these five years. This journey wouldn't have been the same without you.

A heartfelt thank you also goes to my boyfriend for being my rock these last few years. Your love, support, and understanding have helped me through the highs and lows of this journey. I couldn't have made it without you by my side.

Finally, I owe my deepest appreciation to my family - father, mother and brother - for their unconditional love, support, and encouragement throughout every stage of my life.

”

“Perseverance is not a long race; it is many short races one after the other.”

— Walter Elliot

”

“Design is not just what it looks like and feels like. Design is how it works.”

— Steve Jobs

Abstract

In recent decades, making electronic devices smaller has become essential to technological progress. While semiconductor advancements have facilitated the shrinking of electronic circuits, integrating large sensors and actuators has posed challenges, resulting in larger products and increased manufacturing time and cost.

The emergence of Micro Electro Mechanical Systems (MEMS) technology has offered a transformative solution. By combining electronic and mechanical structures on a microscale, MEMS devices enable greater efficiency and miniaturisation, revolutionising the most varied industries, such as the smartphone and healthcare industries. Despite the progress that has been made, the design process for MEMS devices remains intricate and time-consuming, often lacking optimal outcomes. Artificial intelligence can be a strong ally in this area, as it can quickly generate diverse designs that may ultimately surpass those crafted by humans in terms of efficiency and performance.

In this thesis, in collaboration with the Micro- and Nanosystems (MNS) Research Division from KU Leuven University, we present a novel approach to the MEMS design process using conditional deep generative models. We trained these models on datasets of MEMS devices' designs, with a focus on generating new designs that meet specific design constraints and are conditioned on the desired input frequencies. This approach speeds up the design process, encouraging exploration of different design paths that traditional techniques might overlook, potentially leading to more efficient and innovative designs.

Furthermore, we have developed an evaluation methodology crucial in objectively and reliably assessing the models' performance. It ensures that the designs generated by these models not only meet the MEMS design constraints but also have the desired frequencies. Our results demonstrate the effectiveness of generative models in producing valid MEMS designs, highlighting the potential of artificial intelligence in enabling significant advances in the field of microelectronics.

Keywords: MEMS Design, Generative Models, Deep Learning, Constraints

Resumo

Nas últimas décadas, miniaturizar dispositivos eletrónicos tornou-se essencial para o progresso tecnológico. Embora os avanços na área dos semicondutores tenham reduzido o tamanho dos circuitos elétricos, a integração de sensores e atuadores tem apresentado desafios, resultando em produtos maiores e no aumento do tempo e custo de fabricação.

O surgimento de sistemas microeletromecânicos (MEMS) ofereceu uma solução inovadora. Ao combinar estruturas eletrónicas e mecânicas numa escala microscópica, os dispositivos MEMS permitem maior eficiência e miniaturização, revolucionando diversas indústrias, como a dos smartphones e da saúde. Apesar do progresso alcançado, o design de dispositivos MEMS permanece um processo complicado e demorado, muitas vezes não levando a resultados ótimos. A inteligência artificial pode ser uma forte aliada nesta área, uma vez que consegue gerar rapidamente diversos designs que podem, eventualmente, superar os criados por humanos em termos de eficiência e desempenho.

Nesta tese, em colaboração com a Divisão de Investigação de Micro e Nanossistemas (MNS) da Universidade KU Leuven, apresentamos uma nova abordagem ao processo de design para dispositivos MEMS utilizando modelos generativos condicionais. Treinamos estes modelos em conjuntos de designs de dispositivos MEMS, focando-se na geração de novos designs que respeitem determinadas restrições e estejam condicionados com as frequências de input desejadas. Esta abordagem acelera o processo de design, incentivando a exploração de diferentes caminhos que técnicas tradicionais poderiam ignorar, conduzindo a designs mais eficientes e inovadores.

Desenvolvemos também uma metodologia de avaliação, crucial para avaliar de forma objetiva e fiável o desempenho dos modelos. Esta metodologia garante que os designs gerados por estes modelos não só cumprem as restrições de design dos MEMS, mas também têm as frequências desejadas. Os nossos resultados demonstram a eficácia dos modelos generativos em gerarem designs válidos para dispositivos MEMS, destacando o potencial da inteligência artificial em permitir avanços significativos na área da microelectrónica.

Palavras-chave: Design de Dispositivos MEMS, Modelos Generativos, Aprendizagem Profunda, Restrições

Contents

List of Figures

List of Tables

Acronyms

BCE Binary Cross-Entropy *(p. 31)*

CGAN Conditional Generative Adversarial Nets *(p. 10)*
CWGAN Conditional Wasserstein GAN *(pp. 16, 35, 36)*

DCGAN Deep Convolutional Generative Adversarial Nets *(pp. 9, 10)*

FCD Fully Connected Decoder *(p. 33)*
FEA Finite Element Analyses *(pp. 17, 18, 26)*
FID Frechet Inception Distance *(pp. 39, 40, 46)*

GAN Generative Adversarial Nets *(pp. 7, 8, 19)*

MAE Mean Absolute Error *(pp. 34, 35, 45, 54, 55, 60)*
MEMS Micro Electro Mechanical Systems *(p. 5)*
MRE Mean Relative Error *(pp. 43, 45, 54, 55, 60)*

WGAN Wasserstein Generative Adversarial Networks *(pp. 13, 14, 23, 36, 37)*
WGAN-GP Wasserstein GAN with Gradient Penalty *(p. 92)*

1 Introduction

1.1 Context and Motivation

Over the past decades, the evolution of electronic devices has seen a remarkable trend toward miniaturisation. While advancements in semiconductor technology have made it possible to create smaller electronic circuits, the use of large sensors and actuators has delayed this progress, resulting in larger products and increased manufacturing time and cost.

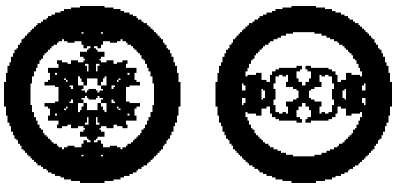

Figure 1.1: Example of MEMS designs.

Fortunately, the development of MEMS technology has enabled the production of smaller, more easily mass-produced components by combining electronic and mechanical structures, including micrometre-sized sensors and actuators. Figure 1.1 illustrates two different MEMS designs. This has resulted in greater product efficiency and miniaturisation, revolutionising a wide range of industries. MEMS devices, such as microphones and accelerometers for smartphones, as well as blood pressure sensors for healthcare, are now widely used across a broad spectrum of fields in our daily lives.

In the current landscape, the design of MEMS devices involves intricate and time-consuming methodologies that sometimes don't lead to optimal outcomes. Artificial intelligence can be a strong ally in this area, as it can generate diverse designs that we believe may ultimately surpass those crafted by humans in terms of efficiency and performance. AI's capacity to explore vast design spaces and iteratively refine solutions holds the potential to revolutionise the MEMS design process, eventually leading to more innovative and optimised outcomes.

1.2 Thesis Scope and Objective

This thesis is in collaboration with the micro and nanosystems division of KU Leuven University. It involves developing a conditional generative model to create new MEMS designs based on a desired frequency. The goal is to quickly generate new MEMS designs that adhere to the design constraints and have the desired frequency. The primary novelty of this thesis lies in the use of generative models, which enable us to create unique design paths that traditional techniques may not produce. This approach saves time in the creation process and allows the exploration of different design paths.

Six students are involved in this project: two MSc students from NOVA University (me and Luís Tripa) and four students from KU Leuven (three MSc students and one PhD student). The allocation of responsibilities among the students is illustrated in Figure 1.2. My specific role focused on developing and improving the generative model, ensuring that it generated functional MEMS designs with the desired frequencies. However, the success of my part was closely tied to the work of my peers, as I relied on their contributions to obtain a diverse dataset and effective evaluation techniques. Additionally, the impact of my work extended to my colleagues, particularly those involved in model optimisation, as they would directly use the designs generated by my improved model. The project's overall success depended on our collaborative teamwork, in which each team member's progress and enhancements significantly influenced the outcomes of others.

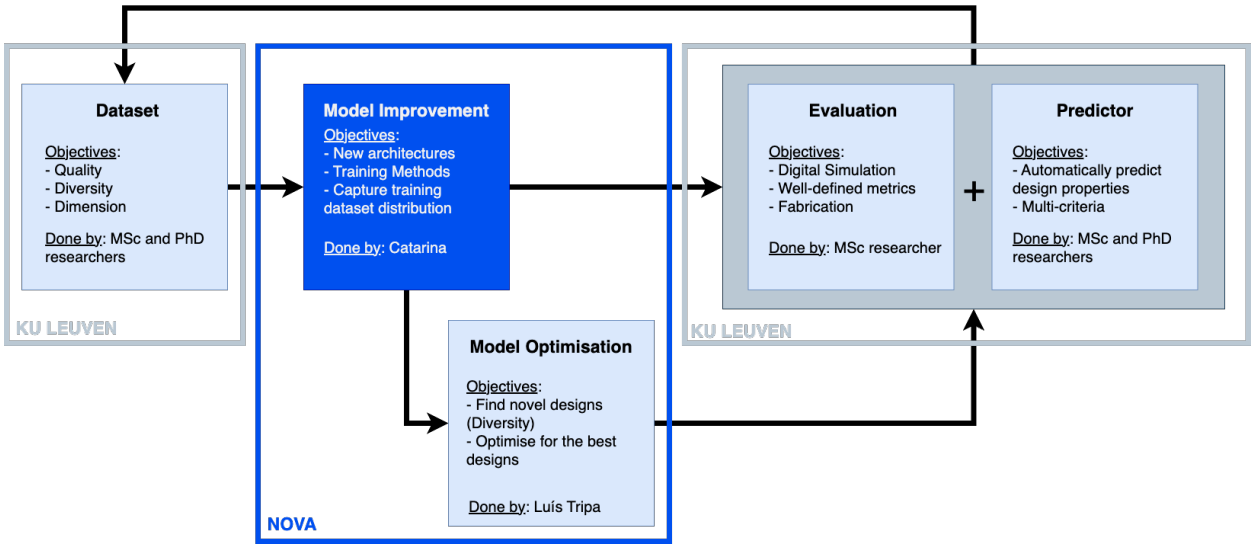

Figure 1.2: Project responsibilities between the two partner universities.

1.3 Challenges and Research Hypothesis

Although the potential impact of this thesis in the MEMS field is enormous, as it aims to significantly reduce the time required for designing MEMS with good performance, numerous challenges must be overcome to achieve these goals. These challenges include:

- Finding which architectures and techniques are the most adequate to generate images of such a specific and structured domain.
- Discovering how to effectively condition the models to create designs with a particular property.
- Finding ways to enforce the generation of designs that follow the MEMS device designs' constraints.
- Discovering ways to evaluate designs. It is very difficult to assess designs solely through visual inspection since visually distinct designs might perform similarly, while designs with significant performance differences can look alike. This makes it challenging to accurately evaluate a model's results based solely on visual inspection. We need to find more objective and reliable ways to evaluate the models' performance.

We propose to address these challenges by taking advantage of deep-learning generative models such as GANs (Goodfellow et al. [18]) and their variants. This model type has already demonstrated successful applications in numerous fields and quantities of data, including MEMS devices [41]. GANs are known to generate high-fidelity synthetic data that closely resembles real data. Once trained, they can produce data at a rapid pace, making GANs a suitable model for our project's objective.

This way, we hypothesise that, together with a diverse and sufficiently large MEMS dataset and a strong evaluation methodology, we can use these models as the basis of this thesis in order to create diverse and efficient MEMS designs while complying with device design constraints.

1.4 Contributions and Achievements

As a result of the work conducted in this thesis, we would like to emphasize the following contributions:

- **Novel dataset of MEMS designs, the KU Leuven dataset:** This dataset was iteratively created throughout this thesis in order to train our generative models. It contains a diverse and balanced set of MEMS designs and their respective frequencies.
- **A thorough study on CGANs aspects for MEMS generation:** We developed a set of generative models that generate MEMS designs following their design constraints and are conditioned on a specific property. In our case, we conditioned the generation of designs on a specific frequency mode, the flexural mode (Mode 4).
- **Novel Evaluation Methodology:** We developed a novel evaluation methodology to assess the models' performance more objectively and reliably. This methodology

is composed of a set of metrics that evaluate the models' performance in terms of the number of valid generated designs, i.e. that adhere to the design constraints, conditional relative error, diversity and quality.

- **Abstract submission under review for IEEE MEMS 2025:** *"Generative and Generalizable Optimization Framework For MEMS Devices Through Active Learning"*, with authors: Catarina Bento, Luís Tripa, David Semedo and Chen Wang (Available in Annex II).

1.5 Document Organisation

The remainder of this document is organised by the following chapters:

Chapter 2 - Background and Related Work: Presents an overview of MEMS technology and an analysis of architectures and techniques used by existing literature.

Chapter 3 - MEMS Datasets: Discusses the datasets used in this project and the image pre-processing performed on them.

Chapter 4 - Conditional Deep Generative Models for MEMS Devices Design: Describes the approach taken to develop the generative models, including the architectures and techniques used.

Chapter 5 - Evaluation: Presents the evaluation metrics used to assess the models' performance, the results obtained, and their analysis.

Chapter 6 - Conclusions: Defines the final conclusions of this thesis and outlines future work.

2

Background and Related Work

In the following chapter, we go through previous research works to gain insights into how these approaches can be beneficial when addressing challenges that resemble the ones presented in the context of our work. Firstly, we introduce the concept of MEMS devices and their properties. In the following sections, we present GANs and some of their variants, discussing some related problems and proposing solutions to those issues. After that, a MEMS-related generative model is explained in detail, including a presentation of results. Finally, we introduce Diffusion Models and their applications in the materials field, followed by a small discussion about some properties of synthetic images, namely the presence of artifacts and the influence that the training dataset has on the model's results.

2.1 MEMS

Micro Electro Mechanical Systems (MEMS) are machines at the micrometre scale composed of mechanical and electronic components. Although some people would consider MEMS to be electronic circuits in a broader and simpler definition, there is a characteristic that electronic circuits do not share with MEMS. While electronic circuits exhibit a solid and condensed design, MEMS incorporate features such as holes, cavities, channels, membranes, and more, emulating 'mechanical' components in some aspects [6].

The significant advantage of these devices is the miniaturisation. Miniaturisation reduces cost by decreasing material consumption and allowing batch fabrication. However, an important side benefit is also in the increase of applicability. Reducing their mass and size allows placing MEMS in places where a traditional system wouldn't have been able to fit [6]. Therefore, these devices are used in many fields, such as telecommunications, consumer electronics, transportation, building automation, and healthcare [1].

2.1.1 MEMS Resonators

There are a lot of different types of MEMS devices that are used in a variety of areas. However, in this project, we'll focus on a specific kind of MEMS, which are the resonators.

This type of MEMS devices is used for a large number of applications. These include sensing functionalities, where alterations in a resonant element are utilised to observe specific parameters, timing functionalities, where resonant elements are employed within electronic systems to produce precise and high-quality clock signals, or filtering functionalities, where resonant structures serve as integral components for filters used in radio-frequency wireless transceivers [1].

2.1.1.1 Properties

The MEMS resonators investigated in this thesis are composed of a ring and an anchor, which is the centre of the circle formed by the ring. These devices must have a path between the anchor and the ring, and this path has to be symmetric in the four quarters of the circle, as illustrated in Figure 2.1.

These devices can be represented computationally as an NxN binary matrix, where 0 represents a void element, and 1 represents a solid element, as demonstrated in Figure 2.1.

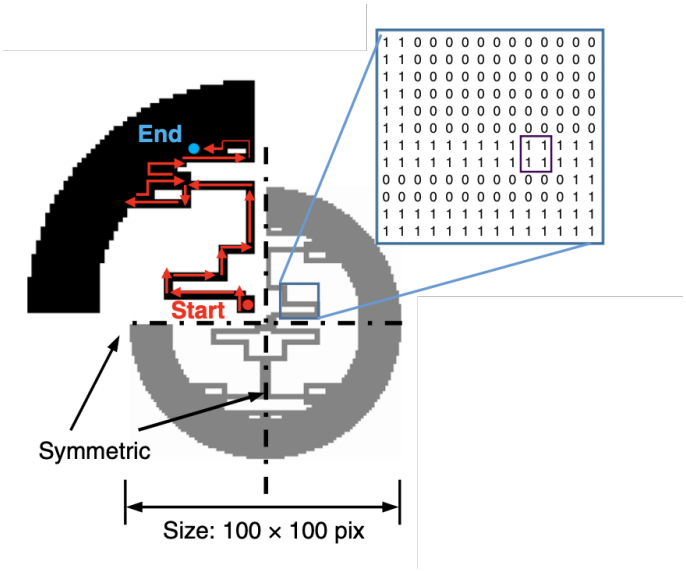

Figure 2.1: Illustration of a MEMS resonator. [20]

The two most important physical properties of a resonator are the resonant frequency modes and the quality factor, also known as the Q-factor. A resonator has infinite resonant modes, but in this project, we'll only focus on four: torsional about X, torsional about Y, rotational, and flexural modes. For simplification, the modes are given numerical order, just like the one shown in Figure 2.2. Mode 1 corresponds to the torsional about X frequency, mode 2 to the torsional about Y frequency, mode 3 to the rotational frequency and mode 4 to the flexural frequency.

The performance of a resonator depends on a large number of underlying resonant characteristics of the device, for example, its quality factor [23]. Therefore, in numerous applications, a key objective in the structural design of resonators is to minimise energy

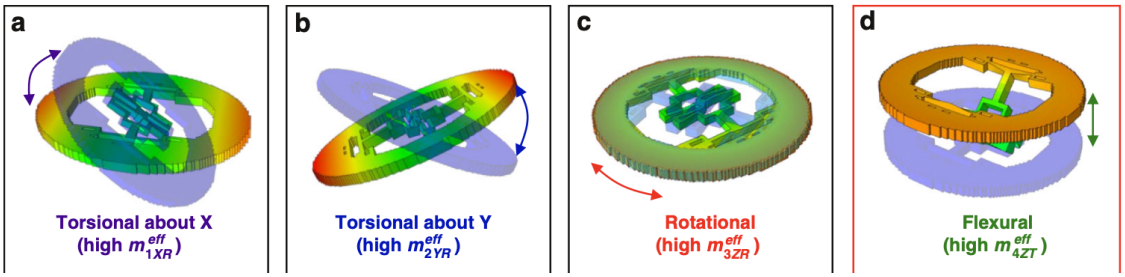

Figure 2.2: Illustration of the four frequency modes. [20]

dissipation in a selected mode, consequently augmenting the Q-factor. This improvement in the Q-factor is essential for enhancing the device's sensitivity, resolution, and overall accuracy. Identifying optimal geometric structures is critical in achieving the desired resonant frequency mode while ensuring a high Q-factor [20].

For the devices to function correctly, certain design constraints must be met. These include:

- Having a ring and an anchor
- Having a continuous path between the anchor and the ring
- Not having "circuit islands" where pixels are part of the circuit but not connected to the path or the ring
- The critical dimension of the design, i.e. the minimum circuit width, must exceed 2.2 micrometres

While we can address the last two constraints by discarding the "circuit islands" and assuming a minimum pixel width, the first two constraints are essential for the model to adhere to, as they directly impact the functionality of the designs.

2.2 Generative Adversarial Nets

For our work, we needed a model that generates MEMS designs according to the properties exposed in the previous section (2.1.1.1). In the remainder of this chapter, I will discuss several types of suitable models for our project.

Generative Adversarial Nets (GAN) are a type of generative model proposed originally by Goodfellow et al. [18] that simultaneously trains two models: a **generator** and a **discriminator**.

2.2.1 Generator

The task of the generator is to generate data that captures the dataset distribution. It can create images [18], videos [46], text [38], or even sound [30]. In our project, its purpose is to generate images.

The generator's input is random noise, typically from a normal distribution. It transforms this noise into an image, which is the model's output.

2.2.2 Discriminator

The discriminator is a binary classifier. Its job is to distinguish whether the samples are from the training dataset or from the generator. The images outputted by the generator are considered fake, i.e. have the label 0, and the images from the dataset are considered real, i.e. have the label 1. All images are handed to the discriminator to classify each image as real or fake. Its goal is to be as good as possible in discriminating the images as real or fake.

2.2.3 Training Procedure

The training procedure of the GAN corresponds to a minimax two-player game with the following objective function [18]:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

The objective of the discriminator is to maximise the probability of correctly classifying real and fake images ($\log(D(x))$), and of the generator is to minimise the probability of the discriminator predicting that its outputs are fake ($\log(1 - D(G(z)))$).

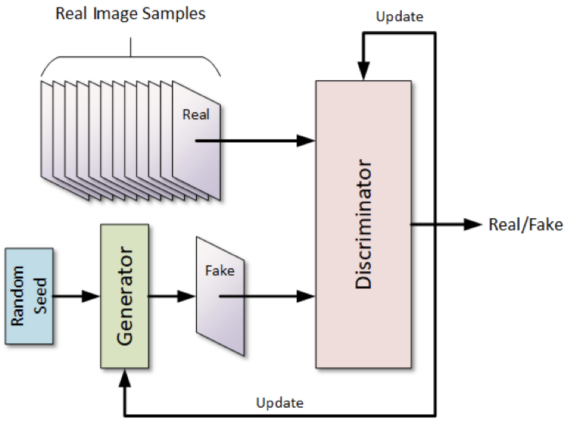

Figure 2.3: GAN training procedure. [17]

The two components are always in constant competition. While the generator tries to create images as similar as possible to the dataset to fool the discriminator, the discriminator always tries to expose the generator. The two components must converge to have the best possible GAN, which is often challenging.

We train the two components of the model alternatively. The discriminator is trained for one or more epochs, the same is done for the generator, and then we return to the discriminator and so on. While we train one component, the other remains constant, otherwise, we would be trying to hit a moving target and might never converge [16].

In the beginning, the generator will create images that don't have any similarities to the ones on the training dataset, and the same will happen with the discriminator since it will not be able to distinguish real and fake images correctly. However, as the training progresses, both components will start to learn with each other until they eventually converge, which is the ultimate goal.

The training procedure of the model is illustrated in Figure 2.3. A vector with random noise will be handed to the generator and transformed into an image. This image will be given to the discriminator along with the training dataset, and it will try to classify the image as real or fake. After that, the weights of the Discriminator and the Generator will be updated one at a time.

2.3 Deep Convolutional Generative Adversarial Nets

In the original GAN formulation introduced by Goodfellow et al. [18], both the generator and discriminator were composed of fully connected layers. Despite that, GAN architectures can vary widely, and researchers have explored different neural network architectures for both the generator and discriminator.

Convolutional layers are commonly used in GAN architectures, especially for image-related tasks. This type of layer is well-suited for capturing spatial hierarchies and local patterns in data, making them effective for tasks such as image generation.

Therefore, architectures similar to the one proposed by Radford et al. [36] are a good fit for our use case. In Deep Convolutional Generative Adversarial Nets (DCGAN), the discriminator is composed of mostly convolutional layers and the generator of transposed convolutional layers.

A convolutional layer is designed to identify specific patterns within the input data. It achieves this by using a set of filters known as kernels. These filters scan the input data and generate a feature map highlighting the patterns detected by each filter. The critical aspect of training involves learning the values of these filters so that the network learns to recognise and extract the relevant patterns from the input data. Figure 2.4 illustrates the operation performed behind convolutional layers, which is filter multiplication.

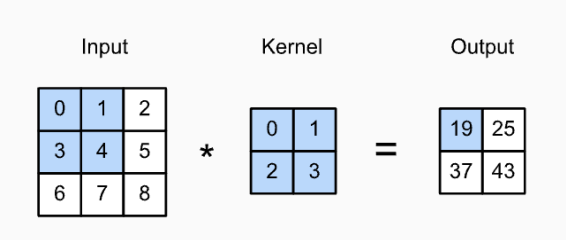

Figure 2.4: Schematic illustration of filter multiplication in convolutional layers. [12]

Transposed convolutional layers share the same goal as regular convolutional layers but focus on upsampling instead of downsampling. In a standard convolution, the kernel shrinks the input, leading to downsampling. Contrarily, in a transposed convolution,

the kernel expands or broadcasts the input, resulting in an output larger than the input. The idea behind transposed convolution is to carry out trainable upsampling. Figure 2.5 illustrates the operation behind transposed convolutional layers.

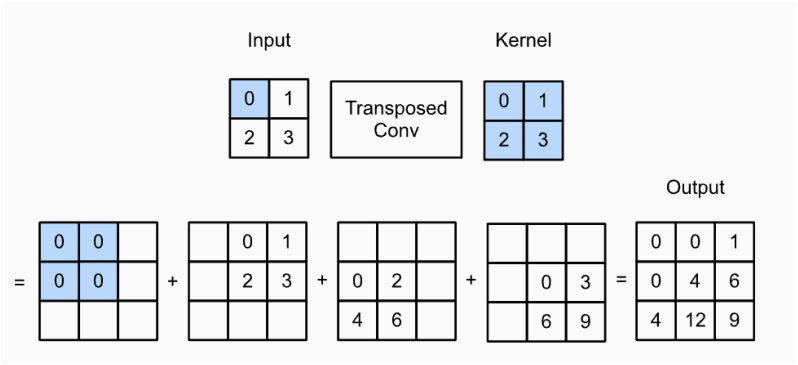

Figure 2.5: Schematic illustration of the operation performed behind transposed convolutional layers. [12]

Therefore, in a DCGAN, the generator uses transposed convolutional layers to transform the random noise into a larger dimension image according to the trainable parameters on the kernel, leading to the creation of a new image similar to the ones on the training dataset. In the discriminator, convolutional layers are used to generate a feature map, smaller than the input image, that highlights the patterns detected by each filter, allowing the classification of each image as real or fake.

2.4 Conditional Generative Adversarial Nets

Our project required a model capable of generating images based on specific properties, such as the frequency modes or the Q-factor. Conditional Generative Adversarial Nets (CGAN) is a GAN variant, proposed by Mirza et al. [33], that does precisely that. It's capable of conditioning the model into generating data according to some property we initially give it. In a standard GAN, the generator is trained to generate samples from random noise, and the discriminator is trained to distinguish between real and generated samples. In a CGAN, the model takes an additional input, typically referred to as a condition or label, which provides information about the desired properties of the generated data. The conditional input allows us to guide the generation process towards specific characteristics or properties we want the generated samples to have.

Its objective function is very similar to the one from the original GAN paper [18]. However, the probabilities are conditioned with the label y [33]:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(\mathbf{x}\,|\,\mathbf{y})] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(\mathbf{z}\,|\,\mathbf{y})))]$$

Figure 2.6 demonstrates precisely how the model is conditioned. Both components receive the label y. In the case of the generator, the condition is used to guide the generation process towards it, and in the discriminator, it's used to assist in the real/fake classification.

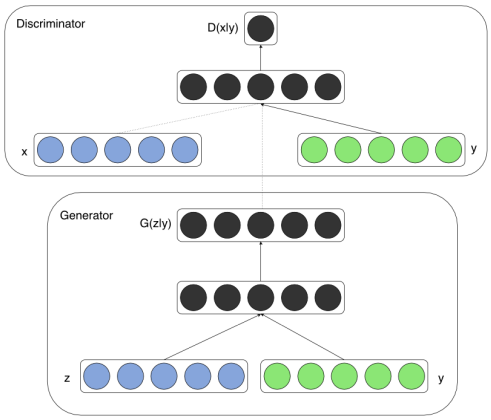

Figure 2.6: Schematic illustration of how the CGAN model is conditioned. [33]

By conditioning the model on relevant information, CGANs enable the creation of data that conforms to desired properties, making them a powerful tool in generative modelling and, thereby, beneficial for our project.

2.5 Mode Collapse

Unfortunately, GANs and their variants suffer from a problem called mode collapse. The generator's main job is to create diverse and realistic outputs, covering all the training dataset distribution. However, there's a risk that the generator becomes too focused on producing a specific output type, and the discriminator, whose job is to distinguish real from generated outputs, adapts by consistently rejecting that particular type.

If the discriminator gets stuck in this mindset and fails to learn more broadly, the generator can easily take advantage of this situation. In each iteration, the generator may discover another specific output that the current discriminator considers realistic, leading to a repetitive cycle. The generator then produces a limited set of similar outputs, and the discriminator cannot escape the trap, resulting in mode collapse [7].

CGANs typically provide higher-quality samples than GANs. Nevertheless, they require labelled data, which can be expensive and hard to find [5]. Usually, the availability of annotated data is quite restricted. Still, it's plausible to consider that even a small amount of annotated data can be more advantageous than having none. Surprisingly, using labels in specific scenarios with limited data leads to mode collapse, while unconditional learning leads to satisfactory generative ability [39].

2.5.1 Techniques to Overcome GAN's Issues

Over the years, much research has been done to get around these issues. Odena et al. [35] constructed a variant of GANs called auxiliary classifier GAN (AC-GAN). This variant employs label conditioning where the generator uses both the class label and the noise to generate images, and the discriminator gives both a probability distribution over sources,

i.e. if the images came from the dataset or the generator (real or fake), and a probability distribution over the class labels. The output of the AC-GAN results in 128 × 128 resolution image samples exhibiting high quality and more diversity.

Karras et al. [29] proposed a new training methodology for GANs to enable high-output resolutions. This technique involves starting with low-resolution images and progressively increasing the resolution by adding layers to both the generator and discriminator networks in synchrony, as illustrated in Figure 2.7. This step-by-step approach in training enables the model to initially grasp the main patterns within the image distribution before focusing on more complex details, which makes the training more stable. Instead of tackling all scales at once, the model gradually increases resolution, allowing it to address simpler questions sequentially rather than attempting to learn the entire complexity of the task from the beginning.

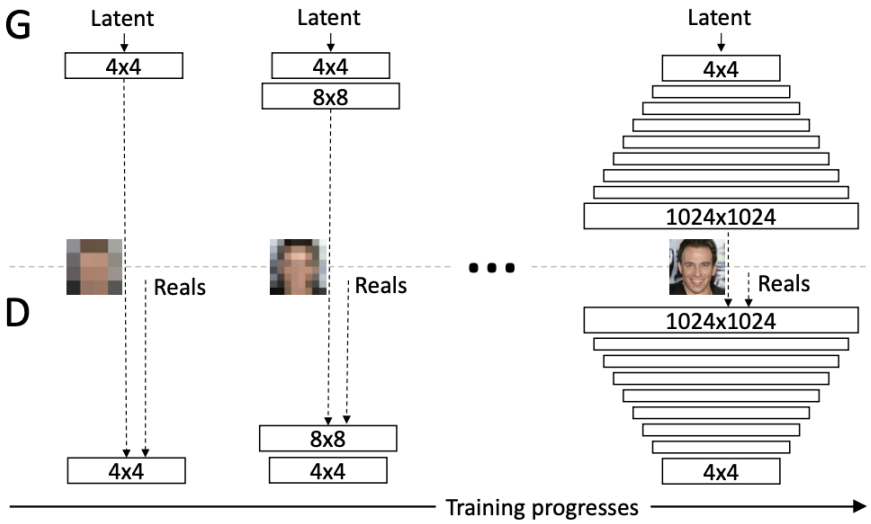

Figure 2.7: Progressive Growing GAN overview. [29]

However, reducing the size of mini-batches to stay within the available memory budget is necessary to enable high output resolutions, but that worsens results. Therefore, the authors also suggested increasing variation using mini-batch standard deviation. By adding a layer towards the later stages of training that forms an additional feature map using the standard deviation of each feature in each spatial location, they demonstrated that it improves the images in terms of resolution and variation.

Karras et al. [28] also introduced some techniques to improve the image quality of StyleGANs. The StyleGAN model was also proposed by Karras et al. [27], and its distinguishing feature compared with vanilla GAN is that it introduces a style vector that controls the style of the generated images, allowing for more control over the appearance of specific features. This style vector is incorporated into the network via normalisation layers, specifically Adaptive Instance Normalisation (AdaIN) layers. These layers are responsible for separating and controlling the style and content of the generated images.

However, this StyleGAN model displays some artifacts in generated images, putting

Karras et al. [28] searching for answers to these defects. The main changes were focused on the generator architecture, specifically the normalisation and progressive growth. Regarding normalisation, the paper suggests applying a "demodulation" operation to the feature maps before normalisation in the AdaIN layers. This operation divides the feature maps by their spatial standard deviation to scale the normalised features based on their variance across spatial dimensions. The purpose is to reduce the influence of feature values, helping to maintain a more stable and controlled normalisation.

To improve the progressive growth, the authors introduced a novel "skip-phase" training strategy, allowing the generator to skip specific resolutions during training and progressively add them later. This modification makes the training more stable and reduces potential artifacts. These combined adjustments contribute to the overall refinement of StyleGANs, leading to enhanced generation of high-quality images.

Liu et al. [31] proposed a Self-Conditioned GAN where a class-conditional GAN is trained without using manually annotated class labels. Instead, the model is conditioned on labels directly derived from clustering in the discriminator's feature space. The algorithm alternates between learning a feature representation for the clustering method and learning a better generative model that covers all the clusters. This way, we solve the problem of not having enough labelled data and, simultaneously, mode collapse by generating data from each cluster.

Casanova et al. [5] partitioned the data into overlapping neighbourhoods described by a datapoint and its nearest neighbours and introduced a model called instance-conditioned GAN that learns to model the distribution of the neighbourhood of a datapoint, also referred to as instance, by giving a representation of the instance as additional input to the generator and discriminatory and by using the neighbours of the instance as real samples for the discriminator.

Shahbazi et al. [39] suggested a training strategy for CGANs that gradually transitions from unconditional to conditional generative learning to prevent mode collapse. In the initial phase of the proposed approach, the unconditional objective is favoured to enhance stability. As the process advances, priority shifts towards the conditional objective, allowing better control over the output by incorporating conditioning factors.

Arjovsky et al. [3] proposed a new GAN variant called Wasserstein Generative Adversarial Networks (WGAN) that uses the Wasserstein distance instead of the Jensen-Shannon divergence. This change results in more stable training and better convergence, reducing the risk of mode collapse. In the next section, we'll explain the WGAN model and its improvements over the original GAN formulation in more detail.

All of these techniques can improve the quality of the generated images and prevent mode collapse.

2.6 Wasserstein Generative Adversarial Nets

WGAN is a GAN variant introduced by Arjovsky et al. [3] designed to enhance the training stability of GANs, preventing mode collapse. The key innovation in WGANs is replacing the Jensen-Shannon (JS) divergence, used in the original GAN formulation, with the Wasserstein distance.

The JS divergence measures the overlap or similarity between two probability distributions rather than their actual distance in the underlying space. Consequently, whether the real and generated distributions are very close or far apart, the JS divergence may not distinguish between these cases, leading to unstable training. In contrast, the Wasserstein distance offers a meaningful metric for measuring the actual distance between the real and generated distributions, resulting in more stable and reliable training.

The Wasserstein distance is defined as follows:

$$W(P_r, P_g) = \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$$

P_r represents the real data distribution, P_g the generated data distribution, and $\Pi(P_r, P_g)$ the set of all joint distributions $\gamma(x, y)$ whose marginals are P_r and P_g. Intuitively, $\gamma(x, y)$ represents the amount of "mass" that needs to be transported from x to y in order to convert the distribution P_r to the distribution P_g. The Wasserstein distance then quantifies the "cost" associated with the optimal transport plan.

If we apply one teorem to the equation above, we can obtain the following:

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim P_r}[f(x)] - \mathbb{E}_{x\sim P_g}[f(x)]$$

This equation shows that the Wasserstein distance can be expressed as the difference between the expected values of a function $f(x)$ over the real and generated data distributions, similar to how it works in a regular GAN. However, in the case of WGANs, the function $f(x)$ is constrained by a Lipschitz constant of at most 1, meaning that the function's slope is bounded by 1. This constraint ensures that the function remains smooth and doesn't become too steep, which is crucial for stable and reliable training.

We can also generalize this constraint to be K-Lipschitz for some constant K by replacing the 1 in the equation with K. This results in the expression $K \cdot W(P_r, P_g)$.

Constraining the function $f(x)$ to be K-Lipschitz offers the advantage that the output of discriminator, also called the critic in WGANs, is a continuous real number, unlike the binary 0 or 1 outputs in traditional GANs. This real-valued output represents the critic's confidence in the authenticity of the input image: higher values indicate a higher confidence that the image is real, while lower values suggest a higher confidence that the image is fake. This continuous output facilitates more stable training and improved gradient flow, leading to better convergence and overall performance.

The WGAN training process involves updating the critic to **maximise** the Wasserstein distance between the real and generated data distributions while updating the generator

to **minimise** the critic's output. The critic's goal is to separate the two distributions as much as possible, making it easier to distinguish real from fake images. On the other hand, the generator aims to bring them closer together, making the fake images closely resemble the real ones, thus making it difficult for the critic to tell the difference. However, it's important to note that in the original WGAN paper [3], the authors proposed training the critic multiple times per generator update. This strategy ensures that the critic has enough opportunities to accurately approximate the Wasserstein distance, providing more reliable feedback for the generator and stabilizing the training process.

Nevertheless, how can we enforce the Lipschitz constraint on the critic? There are two main approaches to achieve this: weight clipping and gradient penalty. In the following subsections, we'll explain both methods in detail.

2.6.1 Weight Clipping

Weight clipping, as proposed by Arjovsky et al. in the original WGAN paper [3], involves constraining the weights of the critic to a specific range, $[-C, C]$, where C is a hyperparameter. This constraint ensures that the critic's weights remain within a bounded range, preventing them from becoming too large and violating the Lipschitz constraint.

Therefore, the training process entails updating the critic's weights, clipping them to the range $[-C, C]$, and then updating the generator.

While weight clipping is a simple and effective method for enforcing the Lipschitz constraint, the authors themselves acknowledged in the original WGAN paper [3] that it is not the best way to achieve this goal. If the clipping parameter C is set too high, the weights may take a long time to reach their limits, making it challenging to train the critic optimally. On the other hand, if the clipping parameter is too low, it can lead to vanishing gradients, especially when dealing with deep networks or when batch normalisation is not applied.

2.6.2 Gradient Penalty

Gradient penalty is an alternative method for enforcing the Lipschitz constraint in WGANs, proposed by Gulrajani et al. [19]. This approach involves adding a regularization term to the critic's loss function, which penalizes the norm of the gradient of the critic's output. The gradient penalty specifically encourages the norm of these gradients to remain close to 1 for randomly selected inputs, thereby imposing a smoothness requirement on the critic's function. The new loss function for the critic with the gradient penalty term is given by:

$$L = \underbrace{\mathbb{E}_{\tilde{x}\sim P_g}[D(\tilde{x})] - \mathbb{E}_{x\sim P_r}[D(x)]}_{\text{Original critic loss}} + \underbrace{\lambda \mathbb{E}_{\hat{x}\sim P_{\hat{x}}}\left[\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1\right]^2}_{\text{Gradient penalty term}}$$

$D(x)$ represents the critic function, $P_{\hat{x}}$ the distribution of $\hat{x}$, which are random samples, and λ the hyperparameter that controls the strength of the penalty.

This method does not require clipping the critic's weights and defining a C value beforehand. Instead, the gradient penalty term enforces the Lipschitz constraint by penalizing the gradients directly, which is said to produce better results compared to weight clipping.

2.7 Conditional Wasserstein Generative Adversarial Nets

Conditional Wasserstein GAN (CWGAN) is a variant of the Wasserstein GAN designed to incorporate conditional generation, extending the capabilities of traditional WGAN by conditioning the model on specific attributes. Similar to CGANs, the CWGAN integrates auxiliary information, guiding the generation process to produce outputs that satisfy given conditions. However, instead of using the JS divergence, as seen in CGANs, CWGAN utilises the Wasserstein distance, which is known for providing more stable training dynamics and preventing issues such as mode collapse.

The CWGAN offers several advantages over both WGANs and CGANs. By combining the conditioning mechanism of CGANs with the stability of WGANs, CWGAN not only generates high-quality samples but also ensures that the samples conform to the desired conditional attributes. This hybrid approach leads to more reliable performance, especially in tasks where precise control over generated outputs is critical.

Several applications of the CWGAN have been proposed in recent literature. Fabbri et al. [15] applied CWGAN to tackle the challenge of generating condition-specific images, demonstrating its effectiveness in producing high-quality images while adhering to specific attributes. Zheng et al. [48] applied CWGAN in the context of imbalanced data classification, where the model was used to generate synthetic samples for the minority class, effectively improving classification performance. These studies showcase CWGANs' adaptability across diverse domains, highlighting its potential for various applications.

2.8 MEMS-related Generative Model

The use of deep learning for the generation of circuit designs has been recently investigated, proving to be a direction with immense potential.

Guo et al. [21] used GANs and some of their variants to create a framework for circuit synthesis design. A similar idea was then explored by Sui et al. [41] but applied to the MEMS field. The authors suggested using a conditional DCGAN to generate MEMS resonator designs based on a physical property. Guo et al. [20] also proposed using a neural network to quickly and accurately predict the physical properties of numerous MEMS design candidates without numerical simulation, which is very time- and resource-consuming. I'll explain the two MEMS-related papers in detail in the following subsections.

2.8.1 Deep Learning for Predicting MEMS Design Properties

In the Guo et al. paper [20], the authors used Brownian-like motion to create a trajectory from the centre anchor to the inner annulus of the resonator, generating a quarter of the design space. By applying symmetry operations along both axes, they generated full resonator designs.

However, they needed to obtain a ground truth to these generated designs, so they performed Finite Element Analyses (FEA) using a software program to calculate the frequency and Q-factor of each design. With these labels and corresponding images, they trained a neural network, which they called Predictor, that tries to predict the original properties defined by the FEA for each image.

The architecture used in this paper was based on the ResNet50 network [22], which is known for its use of residual learning blocks. These blocks help address the vanishing gradient problem, making training very deep neural networks easier.

The Predictor identified the frequency and Q-factor of these designs with approximately 98% and 96%, respectively, saving up to 96% of the total computation time when using conventional numerical tools.

2.8.2 Proposed CGAN

In the Sui et al. paper [41], they used similar processes to the previous work [20] to create the training dataset for the CGAN. This dataset has almost 15000 images and five different labels, which are the Q-factor and the four frequency modes that I presented in the MEMS properties section (2.1.1.1).

The architecture proposed is demonstrated in Figure 2.8. It comprises four parts: the Fully Connected Decoder, the Generator, the Discriminator, and the Predictor.

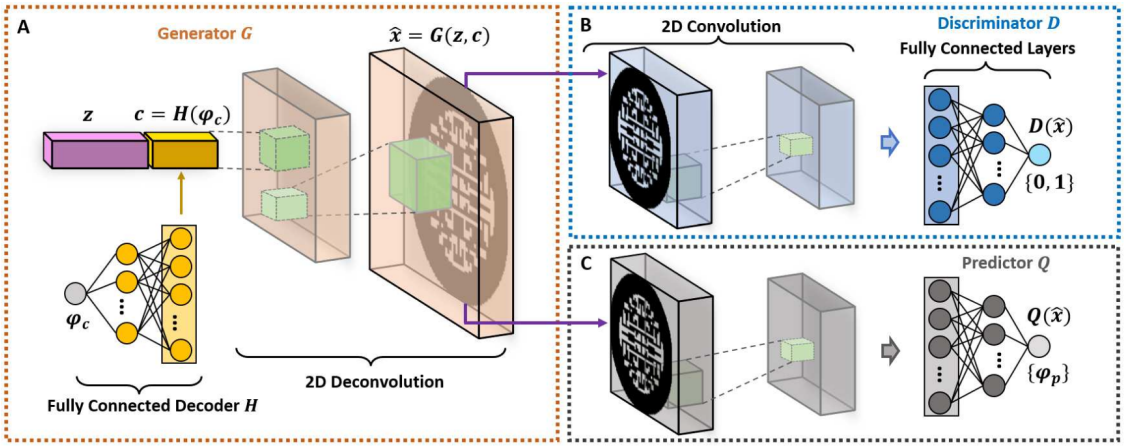

Figure 2.8: CGAN architecture overview. [41]

The model's input is a random noise vector, like in a standard GAN, plus a property handed to the Fully Connected Decoder. This property can be a frequency of one of the

four modes or a Q-factor, for example. The objective of the CGAN is to generate MEMS resonator designs with that target property given as input.

The Fully Connected Decoder is a multiple-layer perception, i.e. a group of fully connected linear layers, that is responsible for taking that desired property and transforming it into a higher dimensional vector. This output vector is then concatenated with the random noise vector to be given to the generator component as input.

The Generator receives that input and mainly through transposed convolutional layers, generates images that ideally translate the target property. These images are then given to the Discriminator and the Predictor.

The Discriminator takes a mixture of dataset images and generated images and, through convolutional layers followed by fully connected layers, classifies each image as real (1), i.e. came from the training dataset, or fake (0), i.e. was generated by the Generator, just like in a standard GAN.

The Predictor also receives the images and, through a similar architecture to the Discriminator, tries to retrieve the desired property from each image. This component's role is to attempt to guide the model in creating images that display the target properties given to it as input. The ultimate goal is for the Predictor's output to be the same as the original given properties. This is the only component that is trained separately. The other components are trained together.

The results given are illustrated in Figure 2.9. The generation accuracy for the four frequency modes is calculated by $|(\varphi_c - \varphi_p)|/\varphi_c$, where φ_c is the targeted physical property and φ_p is the corresponding outcome of the generated design calculated by FEA. The red dashed lines represent the entirely correct matching results in each case.

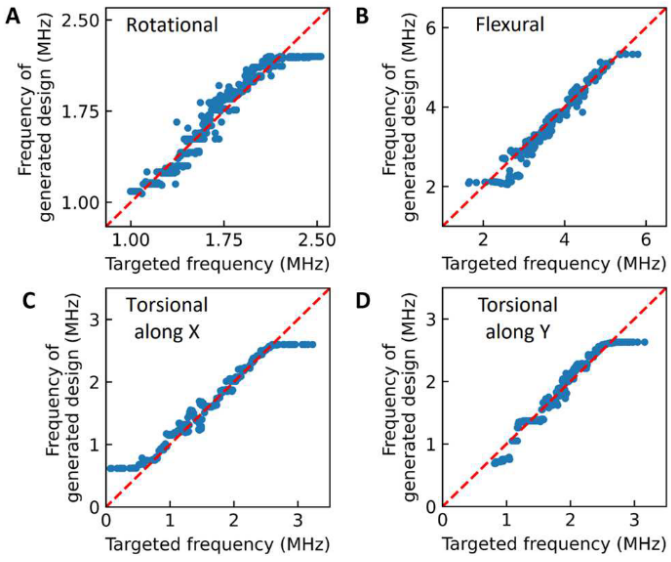

Figure 2.9: Generation accuracy for the four frequency modes. [41]

As we can observe, most of the generated datapoints are near the red dashed lines, with a generation accuracy for the four modes of approximately 96%, 95%, 88%, and 94%,

respectively.

The authors state that the model's performance is significantly influenced by the frequency range targeted, as it is reflected from the boundary values in the figures. They tried eliminating the outliers on the torsional mode along the X-direction, and the accuracy reached approximately 95%. Similar observations can be found for other vibrational modes. The rationale behind this improvement lies in the rarity of geometries associated with such extreme frequencies, leading to a lack of relevant training data during the random generation processes.

In my opinion, these designs with property values closer to the targeted boundaries are just as important as the other designs. All the datapoints are essential, and the model should be able to create all kinds of designs with properties across all the targeted ranges. The model from this paper shows potential. Nevertheless, there is still much room for improvement, namely in these boundary designs. A diverse and well-balanced dataset, along with techniques to address GANs issues, such as those described in Section 2.5.1, were essential in improving these results.

2.9 Diffusion Models

Diffusion Models are a new class of state-of-the-art generative models capable of generating diverse high-resolution images. They were initially proposed by Sohl-Dickstein et al. [40] and then by Ho et al. [25]. Since then, significant attention has been gathered following the successful training of large-scale models by OpenAI, Nvidia, and Google. GLIDE [4], DALLE-3 [10], and Imagen 2 [26] are examples of some popular architectures utilising diffusion models.

While a few years ago GANs were the state-of-the-art models on most image generation tasks, nowadays diffusion models have proven to be better [11]. As mentioned in a previous section (2.5), GANs are often difficult to train and prone to producing low-resolution samples with little diversity. Although GANs present more visual fidelity and quicker sampling, diffusion models are capable of generating diverse high-resolution images, typically being easier to scale and train, making them the new state-of-the-art models in the image generation field.

Diffusion models comprise two processes: **forward diffusion** and **reserve diffusion**. In the forward process, it receives an image and gradually adds Gaussian noise through a series of steps. Afterwards, in the reverse process, a neural network is trained to recover the original data by reversing the noising process. By modelling the reverse process, we can generate a new image. In the following subsections, I'll explain both processes and the model's architecture in more detail.

2.9.1 Forward Diffusion Process

The forward diffusion process comprises a Markov chain with T steps, meaning each step only depends on the previous one. At each step, we add Gaussian noise to the output of the previous step x_{t-1}, producing a new latent variable x_t with distribution $q(x_t|x_{t-1})$, as its formulated in the equation below:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I)$$

This way, we can define the forward process from the input x_0 to the last step x_T in the following manner:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1})$$

During this process, the input image is converted step by step into pure noise. In Figure 2.10, we can schematically see how the forward diffusion process works.

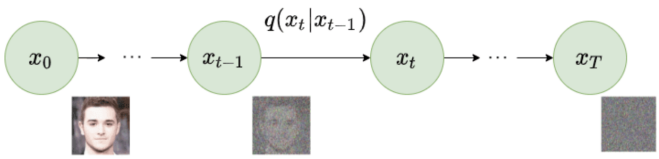

Figure 2.10: Forward diffusion process. [2]

2.9.2 Reparameterization Trick

However, if our model uses a lot of steps, applying the forward diffusion process that many times can be very computationally expensive. Hence, instead of applying the forward process repeatedly to get a desired datapoint x_t, we can use the reparameterization trick.

This trick consists of sampling noise and precomputing the variances and specific parameters to obtain a datapoint from any timestep one wishes. The variance parameter can be fixed to a constant or chosen as a schedule across the timesteps. This schedule can be linear, quadratic, cosine, etc. Ho et al. [25] used a linear schedule, but Nichol et al. [34] demonstrated that utilising a cosine schedule works better, as it can be observed in Figure 2.11. While the last quarter of the linear schedule is almost purely noise, with the cosine schedule, the noise is added more slowly.

2.9.3 Reverse Diffusion Process

The reverse diffusion process trains a neural network to recover the original image by reversing the noising process applied in the forward pass. We learn how to approximate $q(x_{t-1}|x_t)$ with a neural network p_θ. Since $q(x_{t-1}|x_t)$ will also be a Gaussian, we can choose p_θ to be a Gaussian and only parameterize the mean and variance as follows:

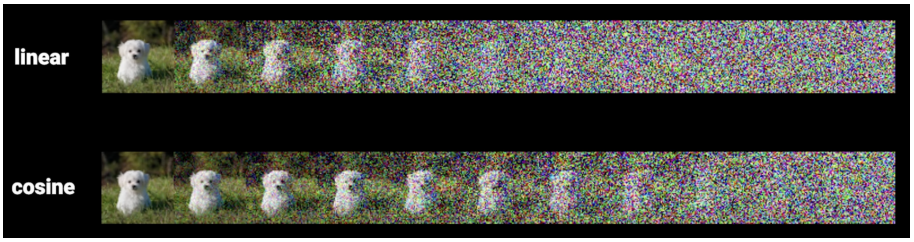

Figure 2.11: Difference between the linear and cosine schedules. (adapted from [34])

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \boldsymbol{\mu}_\theta(\mathbf{x}_t, t), \boldsymbol{\Sigma}_\theta(\mathbf{x}_t, t))$$

Applying the reverse process for all timesteps, we obtain the following data distribution:

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

During this process, the noise is reverted step by step, and the original image reappears. In Figure 2.12, we can schematically see how the reverse diffusion process works.

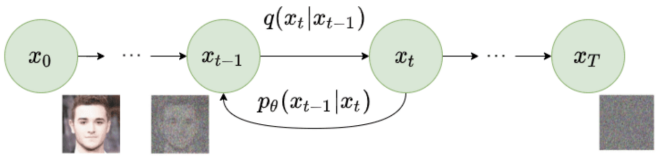

Figure 2.12: Reverse diffusion process. [2]

2.9.4 Architecture

The architecture proposed by Ho et al. [25] is very similar to a U-Net, a symmetrical architecture with input and output of the same dimensions, just like required for a diffusion model.

In Figure 2.13, we can see the original U-Net architecture presented by Ronneberger et al. [37]. It's composed of two parts: an **encoder** and a **decoder**.

The encoder consists of sequences of convolution layers followed by a max-pooling layer. Pooling layers condense the information from a group of neurons in one layer into a single neuron in the next layer. Max-pooling involves selecting the maximum value from a set of values within a specific region, as illustrated in Figure 2.14. This process reduces the data size, simplifying the representation while retaining the most critical features, which is the main objective of the encoder.

The decoder does the inverse of the encoder. It consists of sequences of convolution layers followed by a transposed convolutional layer responsible for the upsampling.

The U-Net is also composed of skip connections between symmetrical stages of the encoder/decoder, where the encoder's features are concatenated with the decoder's

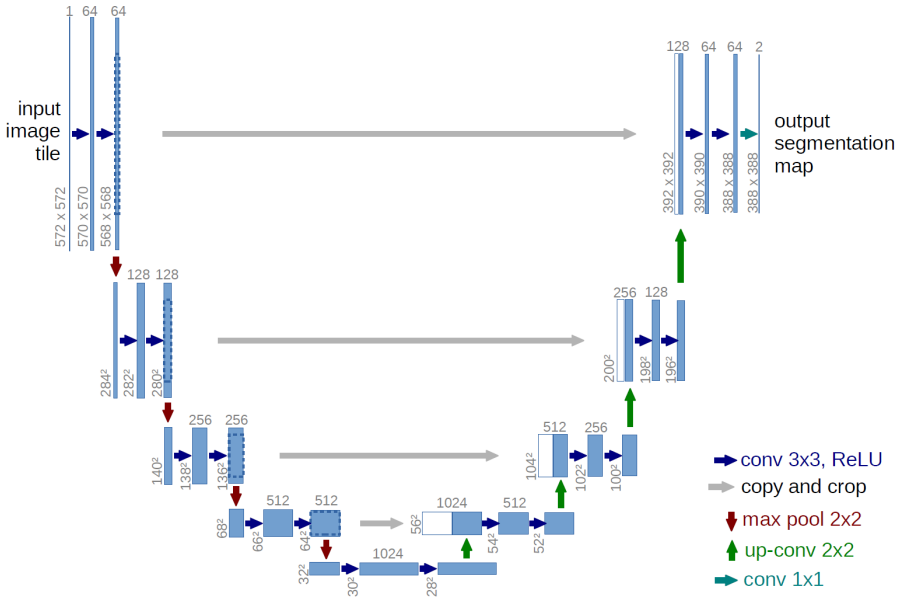

Figure 2.13: Original U-Net architecture. [37]

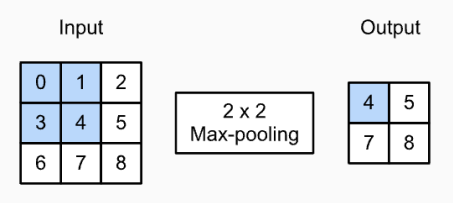

Figure 2.14: Schematic illustration of max pooling layers. [12]

features. This gives the model fine-grained details learned in the encoder part to construct an image in the decoder part.

The architecture proposed by Ho et al. [25] is similar to the original U-Net but is composed of ResNet blocks, self-attention blocks, and group normalisation, which is a normalisation layer that divides channels into groups and normalises the features within each group. ResNet blocks also use skip connections, also known as residual connections, to enable a direct flow of information from one layer to another. Each block consists of a set of convolutional layers followed by a shortcut connection that adds the original input to the output of the convolutional layers, just like illustrated in Figure 2.15. This residual learning approach allows the model to learn the residual information, making it easier to train very deep networks by reducing degradation issues and facilitating the flow of gradients during backpropagation.

Self-attention blocks enable a model to focus on different parts of its input sequence. Unlike convolutional layers, self-attention allows each element in the input sequence to attend to all other elements, capturing long-range dependencies [45].

The proposed architecture also specifies the diffusion timesteps by adding a position embedding into each residual block. This way, the model knows precisely which timestep we are in and can add and remove noise accordingly.

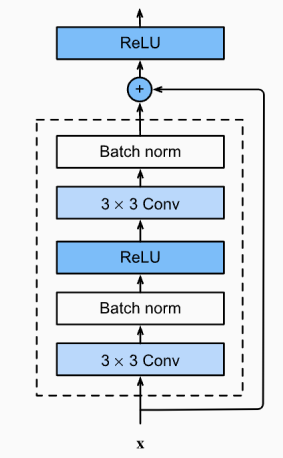

Figure 2.15: Schematic illustration of the architecture of a ResNet block. [12]

2.9.5 Applications in the Materials Field

Diffusion models have already been used for several different applications in the materials field. Herron et al. [24] compared the performance of diffusion models with WGAN in generating microstructures of photovoltaic cells. The paper authors [24] demonstrated that diffusion models outperform WGANs in the task of generating microstructures for organic solar cells, stating that diffusion models are thus far the highest-performing and a reliable option to generate high-fidelity microstructures since they can cover several modes of the target distribution.

Yang et al. [47] have presented the first diffusion model for materials generation that can scale to train on datasets with millions of materials by developing a novel representation, UniMat, based on the periodic table, which enables any crystal structure to be effectively represented. They state that UniMat allows the training of diffusion models, resulting in better generation quality than previous state-of-the-art learned materials generators.

Düreth et al. [14] used diffusion models to reconstruct real-world microstructure data. They constructed two datasets: a highly diverse and morphologically complex one and a smaller one. This way, they not only prove the applicability of diffusion models to small dataset sizes that a single lab can realistically create but also the ability of these models to capture diverse and complex features in this field.

Regarding the use of diffusion models with chemical formulas, Dong et al. [13] presented a deep-learning pipeline for material composition and structure design. This framework is composed of diffusion language models as the generator of compositions based on existing chemical formulas, a template-based crystal structure prediction algorithm to predict their corresponding structure and a structure relaxation using a graph neural network. To validate new structures generated through this pipeline, they used formation energy calculations. They state that through these calculations, they have identified six newly discovered material structures and that their diffusion-based composition generator outperforms previous GAN-based methodologies.

All of these approaches employ diffusion models in areas requiring following some geometric properties or physical criteria, similar to MEMS resonators, and have proven to be effective in producing high-quality images.

2.10 Properties of Synthetic Images

Images generated by deep-learning generative models have some intriguing characteristics that Corvi et al. [9] discussed in their paper.

Many GAN generators leave clear traces of their processing pipeline in the images. Similarly, images generated by diffusion models seem to display similar artifacts, suggesting a potentially shared origin or nature. Even the most sophisticated architectures create artifacts that can be used for detection.

Additionally, it has been observed that when the training dataset is heavily biased, there is a risk of the model learning and transferring these biases into the generated images. This poses a potential issue for both the generators and detectors involved in the process. It has also been noted that these models face challenges in replicating real images' mid to high-level characteristics.

These findings hold significant relevance in my thesis, given the necessity for the precise generation of MEMS designs. Any presence of artifacts in the images makes them unusable for our purposes. However, our images are binarised, contrary to the ones leading to these findings, which are natural images.

The challenges highlighted, such as the models struggling to accurately replicate certain features of original images and the risk of generating biased designs with a strongly biased dataset, were carefully considered during our project. It's important to note that our model's performance was heavily dependent on the quality of the dataset. The more diverse and well-balanced the dataset, the better the model learned and, consequently, the better the generated designs.

Ensuring the design constraints was also a big challenge. None of the models covered in this chapter guarantees that these constraints are being taken into account, which requires additional measures to ensure that these constraints are being followed in the generated designs.

All aspects mentioned were taken into account during the development process of this project in order to produce the best possible results. Ensuring our model's capability to generate diverse designs was crucial, and it required proactive measures to address and prevent these issues from arising in our work.

3

MEMS Datasets

In this chapter, we will discuss the two datasets used in this project. We will begin by explaining how the datasets were used and evolved throughout the project. Then, we will present each dataset individually. After that, we will compare the two datasets to highlight their differences. Finally, we will explain the image processing that was performed on the datasets and the images that were generated.

3.1 Dataset Usage and Evolution

In this project, we used two different datasets: the Berkeley dataset [41] and the KU Leuven dataset, developed collaboratively for our project. Both datasets contain MEMS device designs that adhere to the properties and constraints explained in Section 2.1.1.1.

We initially used the Berkeley dataset, which was provided by the Berkeley University team and served as a solid starting point for our work. While conducting our initial experiments with the Berkeley dataset, the KU Leuven team began creating our own dataset. We created the new dataset to have more detailed designs and to have control over the simulation. This approach allowed us to ensure the consistency of the designs and maintain a balanced dataset, a key factor in achieving good results with our models. Once we had accumulated a sufficient number of designs, we transitioned to using the KU Leuven dataset instead of the Berkeley dataset, as it was specifically tailored to our project needs. As stated in section 1.2 of the introduction chapter, this project was a collaboration between us, NOVA University, and KU Leuven. Therefore, throughout the project, the dataset underwent ongoing creation, modification, and evolution. Consequently, we utilised multiple versions of the dataset before obtaining the final version to train our models and draw our final conclusions.

3.2 Berkeley University Dataset

To kickstart our project, we leveraged the work of Sui et al. [41]. The associated dataset was generously provided by the Berkeley University team for our team to use as a starting

point.

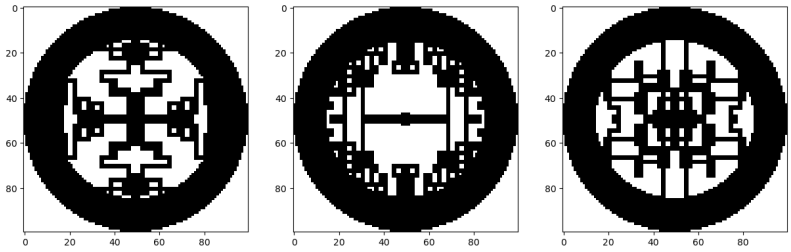

Figure 3.1: Examples of designs taken from the Berkeley dataset.

This dataset comprises almost 15000 images, like the ones shown in Figure 3.1, with a resolution of 100x100. Each image has five different labels: the Q-factor and the four frequency modes.

As explained in section 2.8, to obtain the ground truth of generated designs, Guo et al. [20] and Sui et al. [41] used a software program to perform finite element analyses (FEA). This software can simulate designs and estimate their frequencies and Q-factor. COMSOL [8] is the FEA tool the KU Leuven team has access to. However, the Berkeley team used a different program with different parameterization to simulate the designs. To ensure consistency in the datasets used in this project, we chose to take the designs from the Berkeley dataset and simulate them using COMSOL. However, we only obtained the frequency labels, as the Q-factor information was unnecessary for my work.

In Figure 3.2, we analyse the frequency distributions of the four different modes obtained by COMSOL for the Berkeley dataset. The dataset covers mode 1 and 2 frequencies, ranging approximately from 0.2 to 0.9 MHz, with a predominant cluster of designs concentrated between 0.45 to 0.5 MHz. In the case of mode 3, the frequencies span from 0.3 to 1.7 MHz, with the largest amount of datapoints centered around 0.6 MHz. Finally, in mode 4, the frequencies range from 0.4 to 1.1 MHz, with most designs around 0.55 MHz.

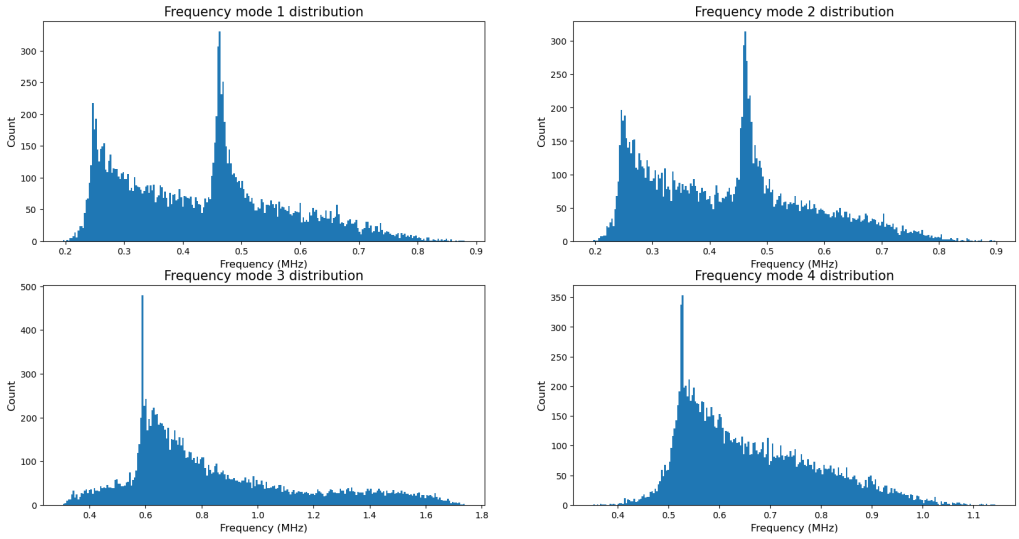

Figure 3.2: Berkeley dataset frequency distributions of the four modes.

3.3 KU Leuven Dataset

This dataset served as the main dataset used in the project for training the models that led us to our final conclusions. The designs were obtained using a similar approach to Guo et al. [20], which involved employing a Brownian-like motion, as explained in section 2.8.1. The labels were obtained using COMSOL simulation.

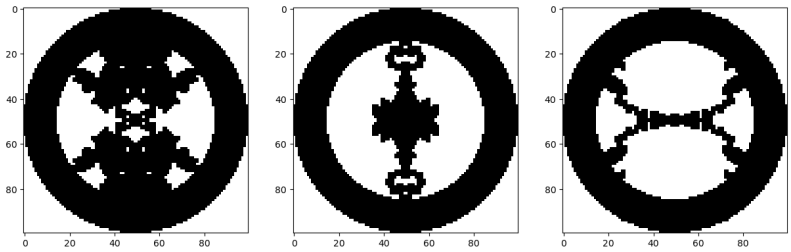

Figure 3.3: Examples of designs taken from the KU Leuven dataset.

The dataset contains 8349 high-quality images, like the ones shown in Figure 3.3, with a resolution of 100x100. Each image has four different labels, which are the four frequency modes.

Figure 3.4 displays the frequency distributions for the four different modes. For modes 1 and 2, frequencies range approximately from 0.2 to 1.1 MHz, with the highest concentration of datapoints between 0.4 and 0.6 MHz. Mode 3 frequencies span from 0.3 to 1.9 MHz, with datapoints almost evenly distributed across this range. In mode 4, frequencies range from 0.5 to 1.5 MHz, with most designs centered around 0.7 MHz.

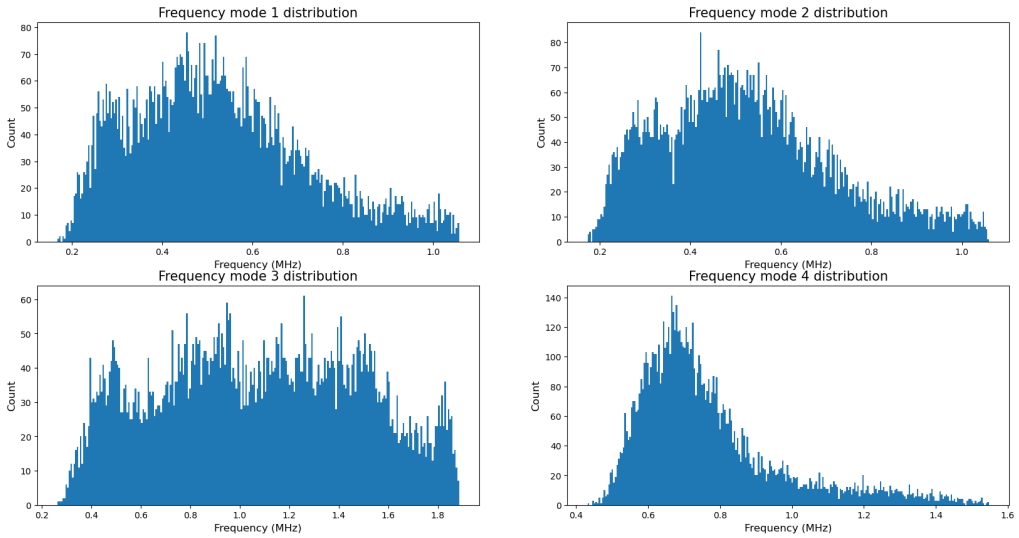

Figure 3.4: KU Leuven dataset frequency distributions of the four modes.

3.4 Dataset Comparison

Although both datasets contain designs that follow the properties described in Section 2.1.1.1 and their respective four frequency modes, they differ in some aspects:

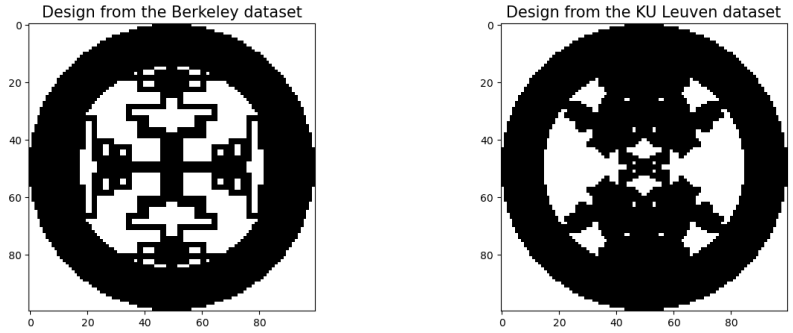

Figure 3.5: Comparison between the designs in the Berkeley and KU Leuven datasets.

- The designs in the KU Leuven dataset are more complex and have more fine-grained details than those in the Berkeley dataset. As we can observe in Figure 3.5, the designs in the KU Leuven dataset have more intricate patterns and shapes, while the designs in the Berkeley dataset are more orthogonal.
- Regarding the frequency distributions of the four modes in both datasets, we can see in Figure 3.6 that the designs in the KU Leuven dataset are spread more evenly across the frequency range for each mode, which is essential to obtain models that can cover all frequencies. On the other hand, the Berkeley dataset has a more concentrated distribution of designs around a specific frequency range for each mode. We can also observe in Figure 3.6 that the frequency ranges in both datasets are similar for the first three modes. In contrast, for Mode 4, the frequency range of the KU Leuven dataset is larger, reaching almost 1.6MHz, while the Berkeley dataset only goes up to around 1.2MHz.
- The KU Leuven dataset contains a substantial number of designs, but fewer than the Berkeley dataset.

3.5 Design Image Processing

Both datasets consist of 100x100 images, each with a single channel containing only zeros and ones. As explained in Section 2.1.1.1, the designs are symmetrical in both axes. This means that we only needed to generate a quarter of the design and then replicate it to obtain the full design. Consequently, we trained our models using 50x50 images to simplify the learning and generation process.

As a result, the dataset images required some pre-processing before being fed to the models, as well as some post-processing for the images generated by the models.

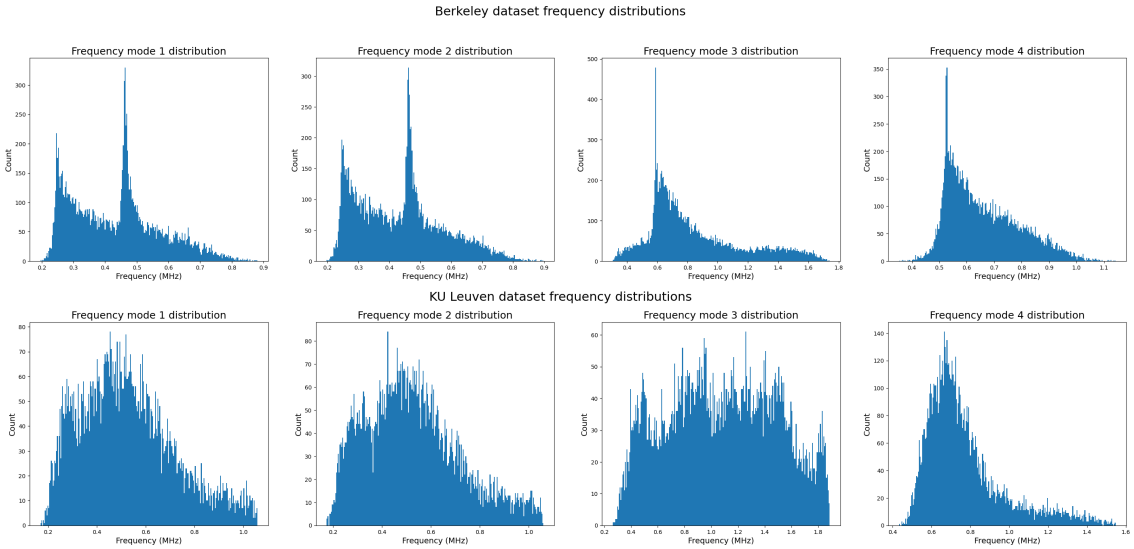

Figure 3.6: Comparison between the frequency distributions in both datasets.

The pre-processing involved transforming the 100x100 images into 50x50, retaining only the upper left quarter of the designs. We did not normalise the images. Instead, we directly provided the models with the binarised images as input.

The post-processing involved the reverse transformation, turning the 50x50 generated designs into 100x100 by replicating the quarters to obtain the full designs. Following this, we removed the pixels outside the ring and then added the ring and anchor pixels to ensure the designs had all the pixels from these two fundamental structures. Finally, we applied a threshold of 0.5 to all the pixels in the designs to ensure they only have values of 0 or 1.

Additionally, we performed some extra transformations to the generated designs before calculating certain metrics. Further details can be found in Section 5.3.

4

Conditional Deep Generative Models for MEMS Devices Design

In this chapter, we will discuss the different models we used to tackle our problem of generating MEMS devices designs conditioned on a specific attribute. We will explain why we chose each model, describe our approach, and emphasize the key aspects of the architectures that were crucial for achieving good results.

4.1 GAN

The first type of model we considered for this problem was GANs. There were already some papers that used GANs for similar problems and achieved good results, such as the Sui et al. paper [41] described in Section 2.8.2, which we used as a reference for this project. Despite our intention to condition the generation of designs based on a specific property, we chose to begin with non-conditional GAN models. This allowed us to address a simpler problem first in order to understand how the model learns before integrating additional components.

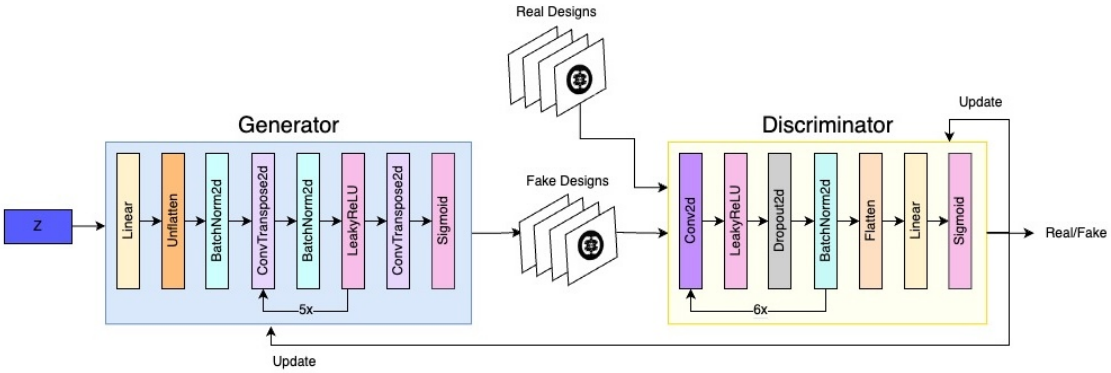

Figure 4.1: GAN Architecture.

The GAN architecture used is illustrated in Figure 4.1. The Generator receives a

random noise vector as input and generates designs. These designs are then given to the Discriminator, along with the real designs from the dataset, so that it can learn to distinguish between real and fake designs. Both components are then updated based on the Binary Cross-Entropy (BCE) losses calculated. The closer the BCE loss is to 0, the better the model is at distinguishing between the real and fake designs.

After training some different configurations of GANs and analysing the results, we identified some critical aspects of the architecture that were important to achieve good results. Using these techniques can help prevent mode collapse, a common issue in GAN training, as elaborated in Section 2.5. By balancing the learning dynamics between the generator and discriminator, we can stabilise the training process and reduce the risk of mode collapse. The specific aspects are outlined below:

Batch Size

We found that the most effective approach to the batch size, a critical parameter in the architecture, differed between the two datasets. For the Berkeley dataset, we found it advantageous to provide the discriminator with fewer fake examples, as it tended to learn faster than the generator. Specifically, we used a batch size of fake examples for the discriminator that was half the size of the real examples and the examples given to the generator. This adjustment slowed the discriminator's learning, helping to balance the training dynamics between the two GAN components.

In contrast, with the KU Leuven dataset, we observed that using the same batch size for both the generator and discriminator (real and fake) was more effective. The likely explanation is that this dataset features more complex designs, causing the discriminator to require more time to learn. Consequently, there was no need to adjust the batch sizes as we did with the Berkeley dataset.

The results of the experiments on both datasets that led to these conclusions are detailed in Section 5.4.9.1.

Learning Rate

We used a cosine learning rate scheduler with warm restarts for both the generator and discriminator. This means that the learning rate gradually decreased following a cosine curve and then reset periodically. This procedure helped prevent the model from becoming trapped in local minima and encouraged better exploration of the loss landscape. In our model, we set the restarts to occur at one-eighth of the total training epochs, providing regular opportunities for the model to refresh its learning rate and potentially discover new and improved designs. Additionally, we assigned a higher learning rate for the generator and a slightly lower one for the discriminator, acknowledging that the discriminator learns faster than the generator.

In this case, the same approach worked well for both datasets. The results of the experiments that led to these conclusions are described in Section 5.4.9.2.

Kernel Size in the Generator's Conv2d Layers

We discovered that the kernel size in the generator's Conv2d layers significantly impacted the model's performance. However, the optimal kernel size depended on the dataset used. For the Berkeley dataset, we found that a larger kernel size was most effective, while for the KU Leuven dataset, a smaller one was more suitable. This difference can be attributed to the complexity of the designs in each dataset. As explained in Section 3.4, the KU Leuven dataset contains more detailed and complex designs, requiring a smaller kernel size to capture the intricate patterns. On the other hand, the Berkeley dataset has simpler designs with fewer details, making a larger kernel size more appropriate for learning the general structure of the designs. The results of the experiments on both datasets that led to these conclusions are detailed in Section 5.4.9.3.

Critical Analysis

While our work with GANs has given us valuable insights, they are non-conditional models and do not allow us to generate designs based on specific attributes such as frequency, which was our goal. This has led us to explore conditional GANs (CGANs) as the next step in our approach, maintaining the techniques mentioned above for this new model type.

4.2 CGAN

CGANs are conditional models that take an input label into account to generate the output. A typical CGAN architecture consists of a generator and a discriminator, similar to a GAN, but each component also receives the label we want to condition on, as described in Section 2.4. However, we have chosen to follow the approach described in the Sui et al. paper [41], which involves only giving the property to condition to the generator.

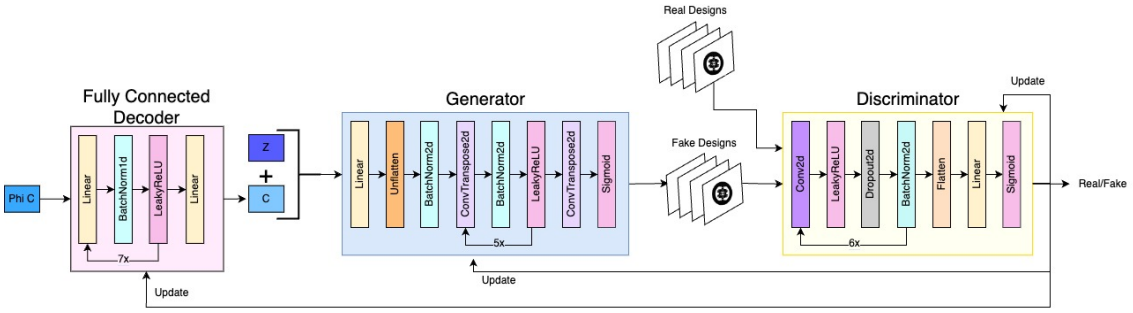

Figure 4.2: CGAN Architecture.

In our case, the property we want to condition the model on is the design's flexural frequency (Mode 4). We take random values from a normal distribution whose mean and standard deviation are calculated from the dataset designs' Mode 4 frequencies (Phi

C). Each random frequency value is then passed to a neural network called the Fully Connected Decoder (FCD), which transforms each property into higher dimensions (C), as explained in Section 2.8.2. The network's output is then concatenated with the random noise vector (Z) and passed to the generator to condition the designs. The architecture illustrated in Figure 4.2 shows the CGAN model I just described. As we can see, the architecture is similar to a GAN, but with the addition of the FCD and the conditioning property.

After training and analysing several CGAN models using this approach, we pinpointed some key architectural aspects that impacted the results:

Hyperbolic Tangent Activation Function in FCD

Using the hyperbolic tangent (Tanh) activation function at the end of the FCD architecture, as illustrated in Figure 4.3, improved our results in some experiments by ensuring that the output values were constrained between -1 and 1. While the normal distribution from which we take the input frequencies isn't limited to this range, many of its values do fall within it. By aligning the output range with this narrower span, we achieved a more balanced concatenation of the random noise vector with the conditioning vector provided by the FCD. This adjustment allows the generator to process the concatenated vector more effectively, leading to improved performance in generating conditioned designs.

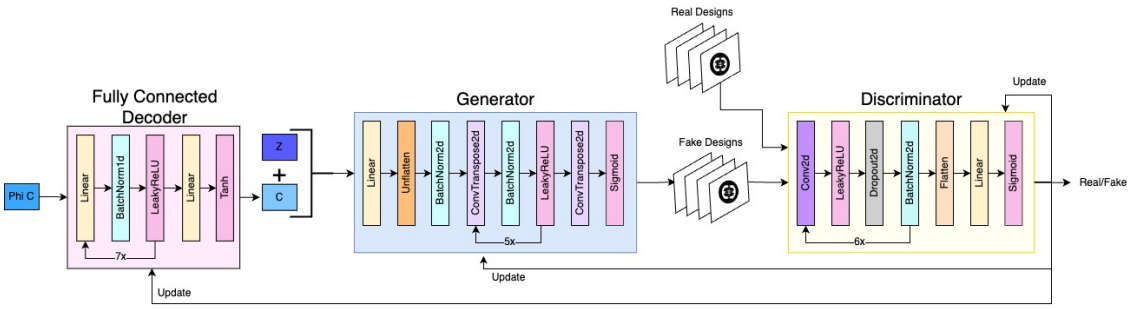

Figure 4.3: CGAN architecture with the Tanh activation function in the FCD.

Number of Layers in FCD

We discovered that using fewer layers in the FCD architecture, as illustrated in Figure 4.4, was more effective in some experiments. This approach preserved the core information of the input label while still transforming it into a higher-dimensional space, enabling the generator to better produce designs based on the desired attribute.

Critical Analysis

Our work with CGANs has provided valuable insights, but it doesn't allow us to directly evaluate whether the generated designs align with the desired frequency. The model

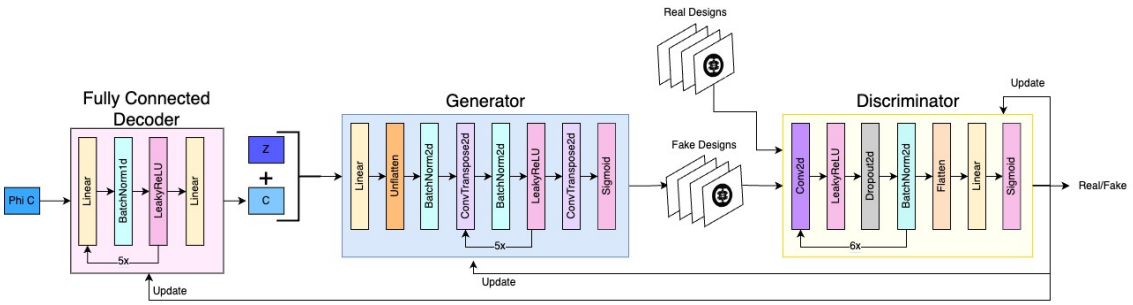

Figure 4.4: CGAN architecture with fewer layers in the FCD.

receives frequencies to condition the generation of designs. However, the model doesn't have a component that forces the generator to produce designs with the specified frequencies. That is why, as a next step, we attempted to incorporate a Predictor into the model to guide the learning process.

4.3 CGAN with Predictor

As detailed in Section 2.8.1, a Predictor is a neural network designed to predict the frequencies of generated designs. In the Sui et al. paper [41], the Predictor is used to compute a loss function that helps to guide the model towards generating designs with the desired frequency, as described in Section 2.8.2. We decided to follow their approach and incorporated a Predictor into our CGAN model to achieve the same goal.

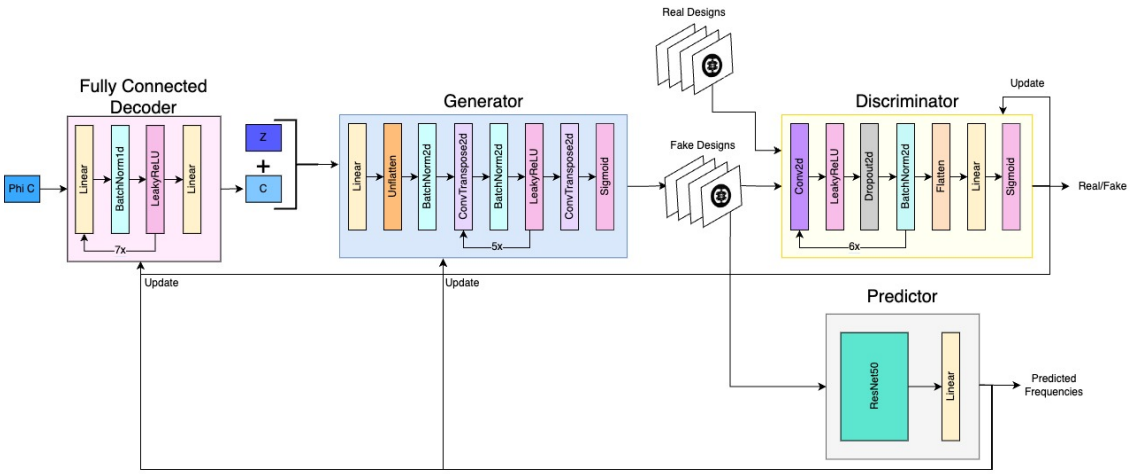

Figure 4.5: CGAN with Predictor architecture.

A member of the KU Leuven team trained a Predictor for each dataset and provided us with the trained models to use in our experiments. This component receives the generated designs and predicts their frequencies. The predicted frequencies are then compared to the desired frequencies to calculate an additional loss, which will also update the Generator and the FCD. The loss function used was the L1 Loss, also known as Mean

Absolute Error (MAE), which measures the difference between the predicted and input values of the frequency mode. The lower the MAE, the better the model supposedly is at conditioning the generation of designs, i.e. in generating designs that actually have the specified frequency. However, it's important to note that the Predictor is not perfect. Being a neural network, it is prone to errors. The architecture of the CGAN with Predictor model that I just described is illustrated in Figure 4.5.

After training and evaluating several CGAN models with the Predictor, we identified certain architectural aspects that notably improved the results:

Freezing the Predictor

We found that freezing the Predictor during CGAN training was beneficial. If the Predictor had been trained simultaneously with the rest of the CGAN model, it would have likely learned from incorrect examples, especially during the initial training phase when the generator was still producing poorly conditioned or potentially invalid designs. This would have resulted in the Predictor providing inaccurate feedback, which could have misled the model. By keeping the Predictor fixed, we ensured that it provided feedback based solely on its prior knowledge from the dataset, which only contained valid designs with accurate frequencies.

Starting from a Pre-trained GAN Checkpoint

We discovered that starting the CGAN training from a pre-trained GAN checkpoint was advantageous. This strategy allowed our model to leverage the general design structure it had already learned from the dataset, enabling it to concentrate more effectively on the conditioning aspects. By starting with a pre-trained GAN, we gave our model a head start, which enhanced its overall performance.

Freezing the GAN Checkpoint

We observed that freezing the GAN checkpoint and focusing solely on learning the conditioned part of the model led to better results. This approach avoided the instability that can arise from training everything simultaneously. By keeping the GAN checkpoint fixed, the model could concentrate exclusively on refining the conditioning aspects, which enhanced its performance.

Critical Analysis

Despite our efforts with the use of these techniques, the architecture remained quite unstable, as evident in Sections 5.4.3 and 5.4.6. Most models experienced mode collapse, and even those that didn't were still inconsistent, with results varying a bit between training attempts. To address these stability issues, we next explored CWGAN as a potential solution to stabilise the training process and enhance the model's reliability.

4.4 CWGAN with Predictor

A WGAN is a type of GAN that aims to improve training stability. By using the Wasserstein distance instead of the JS divergence like in regular GANs, the model accounts for the difference between the real and fake distributions, resulting in more stable and reliable training, as explained in Section 2.6.

As we can see in the equation below, the Wasserstein distance can be expressed as the difference between the expected values of a function $f(x)$ over the real and generated data distributions. The function $f(x)$ is constrained by a Lipschitz constant, meaning that the function's slope is bounded. This constraint ensures that the function remains smooth and doesn't become too steep, which is crucial for stable and reliable training. Constraining the function $f(x)$ to be K-Lipschitz offers the advantage that the output of the discriminator, also called the critic in WGANs, is a continuous real number, unlike the binary 0 or 1 outputs in traditional GANs. This continuous output facilitates more stable training and improved gradient flow, leading to better convergence and overall performance. This way, the model can learn more effectively and possibly avoid mode collapse. That's why we tried using this type of GAN to see if it improved our results and helped to stabilise the training process and prevent mode collapse.

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_g}[f(x)]$$

For this type of model, we employed two different loss functions: one for the critic and another for the generator. The critic's loss is the difference between the average critic score for fake images and real images. This helps the critic estimate the Wasserstein distance between the real and generated data distributions. The generator's loss, on the other hand, is the negative mean of the critic's score for fake images, pushing the generator to reduce the Wasserstein distance between the distributions. In these models, the aim is not to drive the losses to 0. Instead, the goal is to ensure that the critic's loss stabilises, which indicates effective differentiation between real and fake designs, and that the generator's loss decreases, reflecting more realistic generated designs.

So, we approached the problem in the same way we did with regular GANs. Initially, we attempted to train a WGAN model to tackle a simpler problem. Then, we included the conditional part, transforming it into a CWGAN, and finally, we added the Predictor. The architecture of the CWGAN with Predictor model is illustrated in Figure 4.6.

Regarding the WGAN experiments, we identified some critical aspects of the architecture that were important to achieve good results, which are outlined below. Regarding the CWGAN and CWGAN with Predictor experiments, the critical aspects of the architecture were the same as the CGAN and CGAN with Predictor.

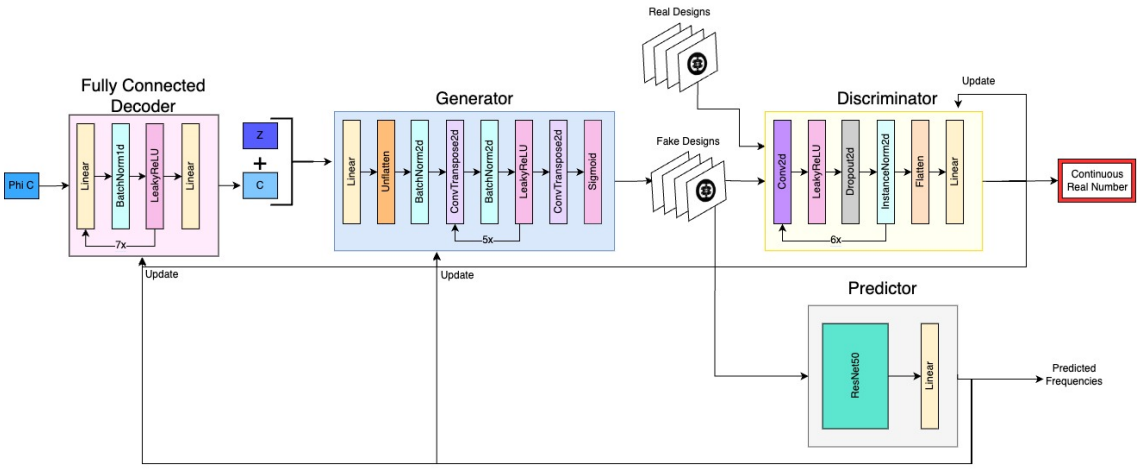

Figure 4.6: CWGAN with Predictor architecture.

Weight Clipping vs. Gradient Penalty

Weight clipping and gradient penalty are two different techniques used to enforce the Lipschitz constraint in WGAN models. The weight clipping technique involves clipping the weights of the critic to a certain range to ensure that the critic remains K-Lipschitz. The gradient penalty technique, on the other hand, involves adding a penalty term to the critic's loss function that constrains the gradient norm to be close to the Lipschitz constraint value, like illustrated in the equation below.

$$L = \underbrace{\mathbb{E}_{\tilde{x}\sim P_g}[D(\tilde{x})] - \mathbb{E}_{x\sim P_r}[D(x)]}_{\text{Original critic loss}} + \underbrace{\lambda \mathbb{E}_{\hat{x}\sim P_{\hat{x}}}\left[\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1\right]^2}_{\text{Gradient penalty term}}$$

We observed that the WGAN with gradient penalty gave better results compared to the original WGAN with weight clipping. The gradient penalty technique eliminates the need to define a clipping value, which is beneficial since choosing an appropriate clipping threshold can be challenging and may lead to poor performance if set incorrectly, as mentioned in Section 2.6.1. Instead, gradient penalty regularizes the model by ensuring that the gradient norm remains close to the Lipschitz constraint value, offering a more flexible and adaptive approach. This penalty results in greater stability during training, as it dynamically adjusts, leading to more reliable convergence and better overall performance in generating high-quality designs. The results that led to these conclusions are detailed in Section 5.4.10.1.

Number of Critic Iterations

Arjovsky et al. and Gulrajani et al. [3, 19] emphasize the importance of training the critic more frequently than the generator. This approach helps the critic more accurately approximate the Wasserstein distance between the real and generated data distributions before updating the generator, as discussed in Section 2.6. Both papers recommend training

the critic five times for each generator iteration. However, in our case, we observed that using three critic iterations per generator iteration produced better results. This could be because our critic trains quicker compared to the generator, making five iterations excessive in our specific context. The results that led to these conclusions are detailed in Section 5.4.10.2.

5

Evaluation

In this chapter, we'll start by explaining the metrics we used to evaluate the models we trained. Then, we'll show the Predictor model's results and explain the processing we performed on the designs before calculating some metrics. Finally, we'll show the results of the experiments that we conducted to evaluate the models and the conclusions we reached with them.

5.1 Metrics

In this project, we used several metrics to evaluate the models we trained. We can divide them into three groups:

Model-based Metrics and Indicators: These metrics evaluate the performance of the models as a whole. This group has the Frechet Inception Distance (FID) and the Losses.

Generated Designs' Metrics: These metrics evaluate the quality of the designs that the models generate. In this category, we have the Validity, the Conditional Relative Error, and the COMSOL Simulation.

Diversity Metrics: These metrics evaluate the diversity of the generated designs, both compared to the training dataset and themselves. This group has the Coverage, the Balance, and the Batch Diversity.

In the following sections, we'll explain each metric in more detail.

5.1.1 Model-based Metrics and Indicators

5.1.1.1 Frechet Inception Distance

The FID is a metric used for evaluating the quality of images created by a generative model. It measures the distance between the feature distributions of real and generated images extracted using a pre-trained Inception network ([42]). It provides a score indicating

how similar the two distributions are. FID scores range from 0 to $+\infty$, with lower values indicating better performance. The lower the score, the better the model is at generating images that resemble the real ones.

To calculate the FID scores of our models, we used the implementation provided by the *TorchMetrics* library. We generated 500 designs using each model and computed the FID scores by comparing these generated designs to a reference dataset of 1000 designs, specifically created by the KU Leuven team for this purpose. This reference dataset was constructed in the same manner as the training dataset. It consists of designs similar to those in the training dataset that the models did not see during training. This allowed for an unbiased evaluation of their performance in terms of FID scores.

5.1.1.2 Loss Indicators

The model's losses are also crucial indicators to consider when evaluating its performance. Although they are more like indicators than traditional metrics, they provide valuable insights into how well the model is learning and where it may be falling short. Losses are calculated continuously during the training process and are then plotted on a graph, known as a loss curve, to visualize their progress. By observing the trends in these curves, we can identify important aspects of the model's performance.

Therefore, losses serve as dynamic indicators of the model's learning progress and areas that need improvement, guiding decisions such as adjusting hyperparameters or modifying the model architecture. While they may not represent final performance outcomes, they provide continuous feedback throughout training, helping to monitor and refine the models.

5.1.2 Generated Designs' Metrics

5.1.2.1 Validity

To be considered valid, a device must meet specific design constraints outlined in Section 2.1.1.1. If one of these constraints is not met, the design is considered invalid. As a result, we developed an algorithm to verify the validity of each generated design. This algorithm is capable of identifying which criteria are not being fulfilled.

There are four reasons for a design to be invalid in our algorithm:

- **Outside Ring (OR):** The design has pixels that are outside the ring.
- **No Ring/Anchor (AR):** The design doesn't have all the pixels that compose the ring and the anchor.
- **No Path (P):** There is no continuous path between the anchor and the ring.
- **Islands (I):** There are islands in the design, i.e., there are pixels that are not connected to the path or the ring.

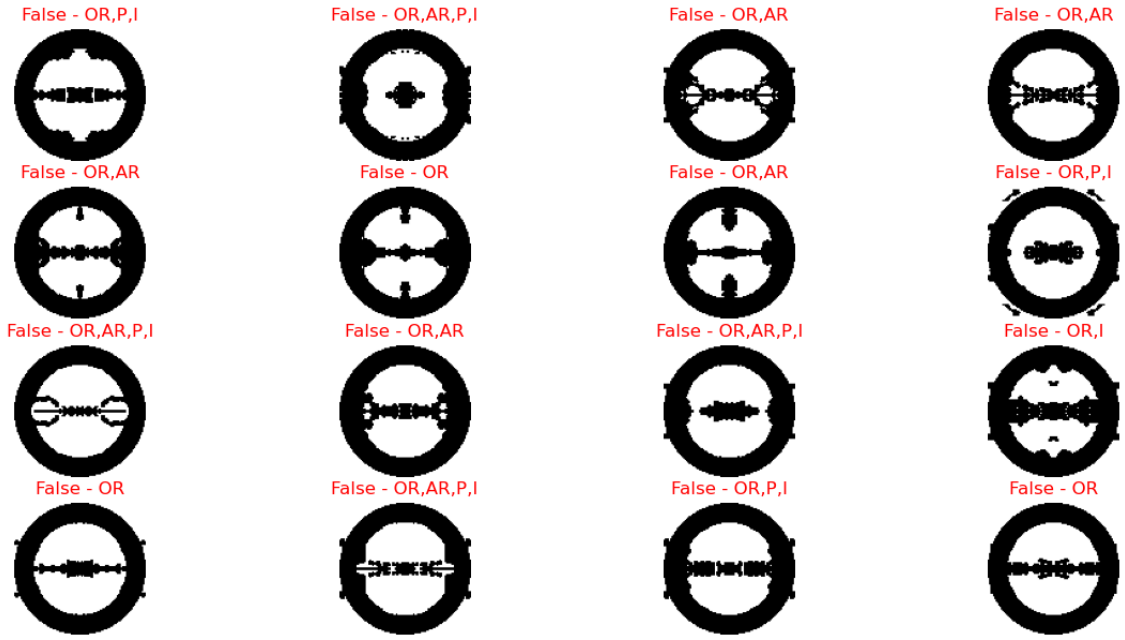

Figure 5.1: Examples of generated designs with their respective reasons for invalidity.

The acronyms in parentheses are used to identify which reasons are being broken in the generated designs, if that's the case, as shown in Figure 5.1.

The last constraint mentioned in Section 2.1.1.1 is not being taken into account in our algorithm since we are assuming that each pixel of the designs exceeds 2.2 micrometres.

In Figure 5.2, we can observe examples of designs that illustrate each of the four reasons for invalidity as defined in our algorithm. For the first design, we notice that there are pixels located outside the ring, which aligns with the "outside ring" reason. Moving on to the second design, we can observe that there are missing pixels in the interior border of the ring, leading to the "no ring/anchor" reason. In the third design, we can see that the middle section of the design is not connected to the rest of the structure that is linked to the ring, thereby falling under the "no path" reason. Lastly, in the fourth design, we can see two isolated groups of pixels that are not connected to the path or the ring, resulting in the "islands" reason.

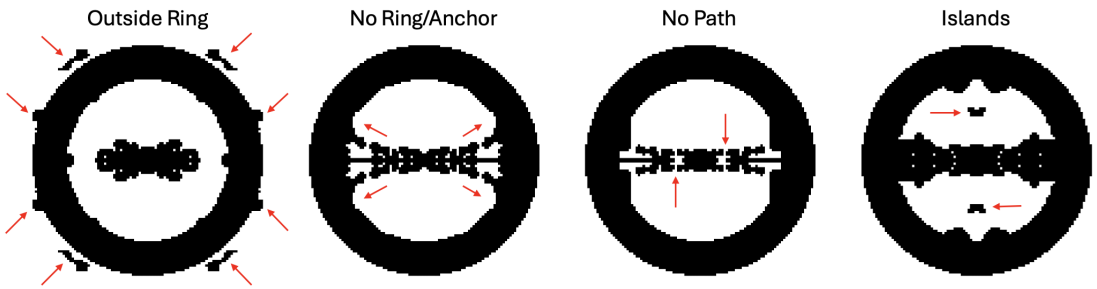

Figure 5.2: Examples of designs with each one of the four reasons for invalidity.

Despite each of the designs in the figure having each one of the reasons to be an invalid

design, each design can have more than one reason to be invalid. In the case of the second and fourth designs, they only have one reason to be invalid, but the first and third designs have more than one reason. The first design has the "outside ring" reason, but it also has the "no path" reason since there is no path connecting the anchor to the ring, and the "islands" reason too since it has an isolated group of pixels. The third design has the "no path" reason, but it also has the "islands" reason since the middle section of the design is isolated from the rest of the structure. Our algorithm is not only implemented to check the validity of a design but also to identify all the reasons why each design is invalid, if applicable.

The validity procedure that we develop is based on the flood fill algorithm. We start at the anchor and put all the pixels that are connected to it in a queue. We pop each pixel from the queue and check if it's a black pixel, i.e. part of the design, and if it has not already been visited before. If not, we add the pixels connected to it to the queue, mark this pixel as visited and add it to the reachable array. We repeat this process until the queue is empty. In the end, we have in the reachable array all the pixels that were reached by the algorithm. This array will be used later to check some reasons for invalidity.

Also, to help us find the invalidity reasons of the designs, we used a mask that contains all the pixels that can be part of a design and a template of a design with the anchor and the ring. Figure 5.3 shows the mask and the template used in our algorithm.

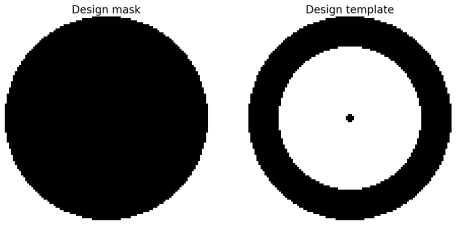

Figure 5.3: Design mask and template.

The procedure to check each one of the invalidity reasons is as follows:

- To determine if there are pixels outside the ring, we compare the generated design with the mask. If there are pixels in the design that are not in the mask, then the design has the "outside ring" issue.

- To check for missing pixels in the ring and the anchor, we compare the generated design with the template. If there are pixels in the template that are not in the design, then the design has the "no ring/anchor" issue.

- To verify if there is a continuous path between the anchor and the ring, we use the reachable array from the flood fill algorithm. We specifically consider the pixels in the reachable array that correspond to the area of the ring. If there are no black pixels in this selected array, it indicates that the ring wasn't reached, resulting in the "no path" issue for the design.

- To identify islands in the design, we also use the reachable array. If the algorithm doesn't reach all the pixels in the design, the design is flagged with the "islands" issue.

If a design passes all of these checks, it's considered valid. If it fails any of them, it's considered invalid.

When evaluating our models, we calculate the validity metric as the percentage of valid designs in a batch of 500 generated designs. After training each model, we analyse 500 designs and calculate the percentage of valid designs, as well as the percentage of designs without each one of the reasons for invalidity. Therefore, the higher the percentage of valid designs, the better the model.

However, it's important to note that there is a relationship between the "no path" and "islands" reasons. If a design lacks a path between the anchor and the ring, it will also have islands in the design. This connection arises from the fact that the flood fill algorithm may not reach all the pixels in the design, leading to the presence of islands if it doesn't reach the ring. Table 5.1 shows the relationship between these two reasons for invalidity.

	Don't have path (0)	**Have path (1)**
Have islands (0)	**Invalid Design**	**Invalid Design**
Don't have islands (1)	**Impossible**	**Valid Design**

Table 5.1: Relationship between the "no path" and "islands" reasons for invalidity.

Thus, when analysing the validity metric results, it's crucial to consider the relationship between the "no path" and "islands" reasons.

5.1.2.2 Conditional Relative Error

When evaluating the conditional aspect of models, one of the metrics used is the relative error. This is calculated after the models have been trained by generating 500 designs and then computing the Mean Relative Error (MRE) between the specified frequencies used to condition the design generation, and the frequencies predicted by the Predictor. The lower the percentage of relative error, the better the model is at generating designs with the desired frequency. However, it's important to keep in mind that the Predictor is also a neural network, and it has errors, as will be demonstrated in Section 5.2. Therefore, this metric serves as an indicator and not a perfect measure. It provides a quick way to evaluate if the models are learning to generate designs that have the desired frequency without having to simulate the designs with COMSOL, which is a time-consuming process.

5.1.2.3 COMSOL Simulation

The COMSOL simulation provides a more accurate metric to assess the conditional aspect of the models. This software is capable of simulating MEMS devices and providing the real frequencies of the designs. Therefore, this metric serves as the ground truth for the generated designs' frequencies. By leveraging its simulation results, we can evaluate the models' performance in generating designs with the desired frequency and assess the Predictor's accuracy in predicting the frequencies. These results enable us to calculate various metrics and create graphs to analyse and compare the performance of different models.

5.1.3 Diversity Metrics

We evaluated our models in terms of diversity in two different ways: by comparing the generated designs with the training dataset and by comparing the generated designs with themselves [1].

To calculate the diversity of the generated designs compared to the training dataset, we used two metrics calculated based on the results of a clustering algorithm (KMeans). The first metric is the **coverage**, which measures the percentage of clusters from the training dataset covered by the generated designs, i.e. that have at least one design. This metric ranges from 0 to 1, with higher values indicating better diversity. The second metric is the **balance**, which measures how the designs are distributed among the clusters. It is divided into two parts: the *balance mean* and the *balance std*. The *balance mean* measures the average number of valid designs per cluster, while the *balance std* measures the standard deviation of the number of valid designs per cluster. One way to evaluate these balance metrics is to see if the mean is higher than the standard deviation. This indicates that the designs are well distributed among the clusters.

To assess the diversity of the generated designs compared to themselves, we used the **batch diversity** metric. This metric is determined by calculating the cosine similarity between the designs' embeddings, which are provided by an Autoencoder trained explicitly for this purpose, within a batch of 500 designs. This metric ranges from -1 to 1, where 1 indicates that all the designs in the batch are identical and -1 indicates that they are all different. The lower the value, the better the diversity of the designs in the batch.

[1] Both metrics were implemented by Luís Tripa, my colleague from NOVA University, who was also working on this project.

5.2 Predictor's Results

The Predictor component was trained by a member of the KU Leuven team to predict the four frequency modes of the designs using the ResNet50 network as a base, similar to the approach described in the Guo et al. paper [20]. Two different versions of the model were trained: one with the Berkeley dataset and another with the KU Leuven dataset. The models were given 50x50 images along with their respective labels for the four frequency modes.

After training, the models were tested with 20% of the datasets. The test loss (L1 Loss) was approximately 0.035MHz for the Berkeley model and 0.052 MHz for the KU Leuven model. The L1 Loss measures the MAE between the predicted and true values of the frequency modes. In this case, it means that the Predictor can predict the frequency modes with an average error of 0.035MHz and 0.052 MHz respectively.

Table 5.2 shows the MRE and MAE values of the Berkeley model for each of the four frequency modes obtained with the test set. Table 5.3 presents the same results, but for the KU Leuven model.

Mode	MRE (%)	MAE (MHz)
1	5.80	0.0235
2	5.86	0.0240
3	8.25	0.0649
4	4.36	0.0282

Table 5.2: MRE and MAE values of the Berkeley model for each one of the frequency modes obtained with the test set.

Mode	MRE (%)	MAE (MHz)
1	8.44	0.0390
2	8.14	0.0380
3	10.57	0.0953
4	4.73	0.0366

Table 5.3: MRE and MAE values of the KU Leuven model for each one of the frequency modes obtained with the test set.

In our models, we only used the Mode 4 frequency as the label for the conditioning part. Therefore, we are only interested in the MRE and MAE values for this mode. As shown in Table 5.2, the MRE and MAE values of the Berkeley model for Mode 4 are 4.36% and 0.0282 MHz, respectively. The values for the KU Leuven model are 4.73% and 0.0366 MHz, as displayed in Table 5.3.

5.3 Image Processing Before Metrics

As detailed in Section 3.5, the models generate 50x50 images, i.e. only one-quarter of the designs. Therefore, we needed to do some post-processing, which involved transforming the generated designs to 100x100 by replicating the quarters to obtain the complete designs, removing the pixels outside the ring using the design mask displayed in Figure 5.3, adding the ring and anchor pixels using the design template also shown in Figure 5.3, and applying a threshold of 0.5 to all the pixels in the designs to ensure they only have values of 0 or 1.

However, before calculating specific metrics, we needed to make some additional transformations to the designs.

To evaluate validity, we performed the transformations described above, enabling our validity algorithm to operate accurately. We removed the pixels outside the ring and added the ring and anchor pixels to the designs to ensure that the generated designs followed the two constraints outlined in Section 5.1.2.1, which we could fix by hand. This allowed us to focus only on the constraints that the model must have learned, as we couldn't manually resolve them for the designs to become valid. We did these transformations before each time we calculated the validity metric, which was at the end of each epoch and at the end of training.

Before calculating the conditional relative error, we did one transformation to the generated designs. As mentioned in Section 5.2, the Predictor is based on the ResNet50 and was trained with 50x50 images. This network expects 224x224 images as input, so we resized the 50x50 generated designs to 224x224 to match the dimensions the ResNet was expecting. This enabled us to pass the designs to the Predictor, so it could predict the frequencies of the generated designs, enabling us to accurately calculate the conditional relative error with those frequencies.

For the FID metric, not only did we do transformations to the generated designs, but we also had to do transformations to the reference dataset. As mentioned in Section 5.1.1.1, we used the Inception network to extract the feature distributions of the real and generated designs to calculate the scores. This network expects 299x299 images with 3 channels as input, so for the reference dataset, we resized the 100x100 designs to 299x299 and replicated the channels to have 3 channels. For the generated designs, we applied the same transformations we applied before calculating the validity metric and then resized them to 299x299.

Before calculating the diversity metrics, we also transformed the generated designs because we used an Autoencoder to extract the embeddings, as explained in Section 5.1.3. This model was trained on 100x100 images with 1 channel, excluding the designs' ring and anchor. Therefore, we resized the generated

designs to 100x100, removed the ring and anchor pixels, and selected one of the 3 channels, as they are all the same due to the images being black and white.

5.4 Experiment Results

5.4.1 Experiments Plan

In the following sections, we'll show the results of the experiments that we conducted to evaluate the models we've trained.

We'll start by showing the results of the best models that we obtained for our problem, which were the models that we used to generate the designs for the COMSOL simulation. With these results and the results from the COMSOL simulation, we'll explain the reasons that led us to conclude which model is the best for our problem.

After that, we'll show the results of the groups of experiments that we conducted to evaluate which parameters and techniques were the best for each type of model (CGAN and CWGAN) trained in each dataset (Berkeley and KU Leuven).

Then, we'll present the configuration and results of the best GAN and WGAN models that we've trained with each dataset. We'll also show the results of some experiments that we did with GAN and WGAN models to compare different configurations and parameters.

Finally, we'll summarise the conclusions we've reached with all the results of the experiments.

Here is a small index of the following sections:

5.4.2 Best Model for Our Problem

To assess which model is the best for our problem, we chose two models from each type (CGAN and CWGAN) that performed the best in terms of validity and conditional relative error in the experiments that we'll show in the next sections. It's important to note that the selected models were trained on the KU Leuven dataset, as it is the main dataset of our project, and we want to take the final conclusions with it.

The chosen CGAN models were the CGAN KU Leuven V8 and CGAN KU Leuven V9, while the selected CWGAN models were the CWGAN KU Leuven V6 and CWGAN KU Leuven V9.

The **CGAN KU Leuven V8** model has the following configuration:

- Predictor: Frozen and receives all the designs generated by the Generator (valid and invalid)
- Pre-trained GAN checkpoint: Yes, it's a **frozen** checkpoint of the best GAN obtained with the KU Leuven dataset (Section 5.4.9)
- Tanh in FCD: Yes
- Reduced Architecture of the FCD: No

Figure 5.4 shows the plots of the Predictor's loss and the validity with the reasons, for the second training done with the model.

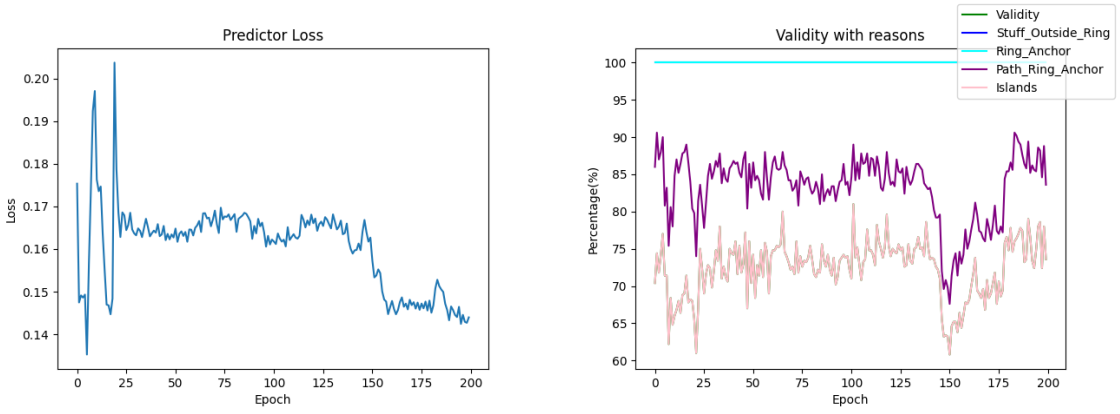

Figure 5.4: Predictor's Loss and Validity plots of the second training done with the CGAN KU Leuven V8 model.

The configuration of the **CGAN KU Leuven V9** model is as follows:

- Predictor: Frozen and receives all the designs generated by the Generator (valid and invalid)

- Pre-trained GAN checkpoint: Yes, it's a **frozen** checkpoint of the best GAN obtained with the KU Leuven dataset (Section 5.4.9)
- Tanh in FCD: Yes
- Reduced Architecture of the FCD: Yes

Figure 5.5 shows the plots of the Predictor's loss and the validity with the reasons, for the second training done with the model.

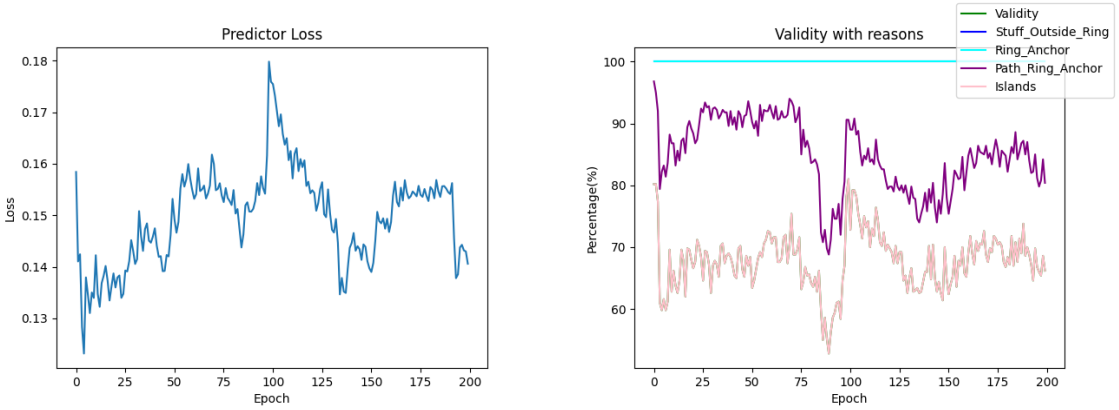

Figure 5.5: Predictor's Loss and Validity plots of the second training done with the CGAN KU Leuven V9 model.

The **CWGAN KU Leuven V6** model has the following configuration:

- Predictor: Frozen and receives all the designs generated by the Generator (valid and invalid)
- Pre-trained WGAN checkpoint: Yes, it's a **frozen** checkpoint of the best WGAN obtained with the KU Leuven dataset (Section 5.4.10)
- Tanh in FCD: No
- Reduced Architecture of the FCD: No

Figure 5.6 shows the plots of the Predictor's loss and the validity with the reasons, for the third training done with the model.

The configuration of the **CWGAN KU Leuven V9** model is as follows:

- Predictor: Frozen and receives all the designs generated by the Generator (valid and invalid)
- Pre-trained WGAN checkpoint: Yes, it's a **frozen** checkpoint of the best WGAN obtained with the KU Leuven dataset (Section 5.4.10)

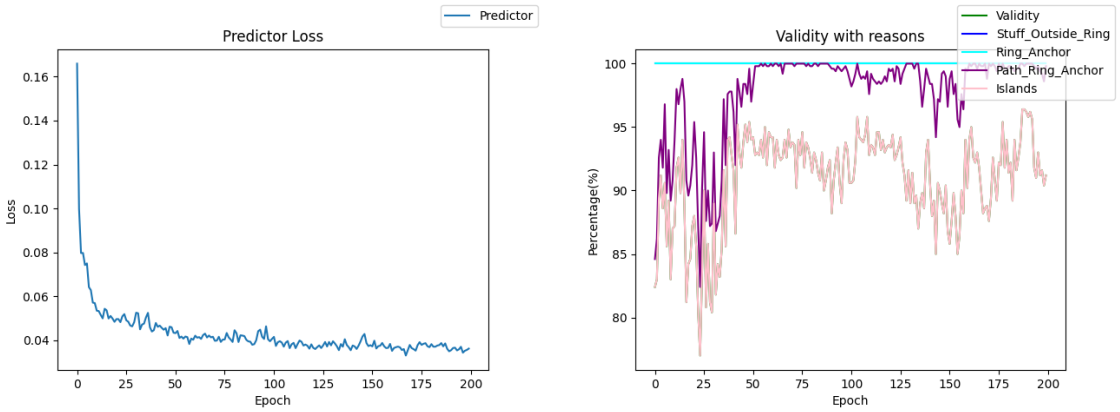

Figure 5.6: Predictor's Loss and Validity plots of the third training done with the CWGAN KU Leuven V6 model.

- Tanh in FCD: No
- Reduced Architecture of the FCD: Yes

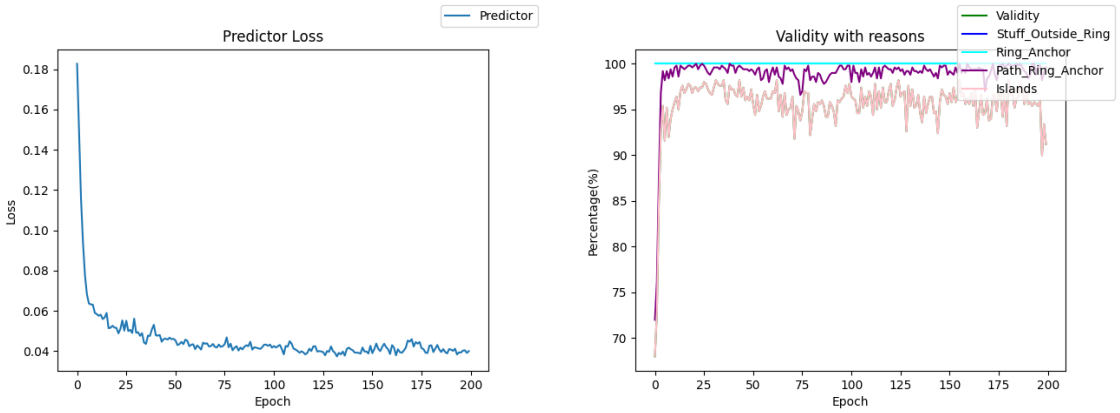

Figure 5.7: Predictor's Loss and Validity plots of the third training done with the CWGAN KU Leuven V9 model.

Figure 5.7 shows the plots of the Predictor's loss and the validity with the reasons, for the third training done with the model.

It's important to note that the parameters that are not mentioned in the descriptions are the same as the best GAN/WGAN models obtained with the KU Leuven dataset (Sections 5.4.9 and 5.4.10).

Discussion

As we can see in all the validity plots, the "outside ring" and "no ring/anchor" reasons are always at 100% because we fixed these issues in the designs before calculating the validity metric, like explained in Section 5.3.

We also can observe that the "no path" reason has always a higher percentage than the "islands" reason, which is expected due to the relationship between these two reasons for invalidity, as explained in Section 5.1.2.1. If there is no path between the anchor and the ring, there will always be islands in the design. However, if a design has islands, it doesn't necessary mean that there is no path between the anchor and the ring. Therefore, the "no path" reason has always a higher percentage than the "islands" reason, like we can see in the plots.

Another important aspect to notice is that the "islands" reason line is overlapping the validity line in all the plots. If we think about it, it makes sense because if the "islands" line was above the validity line, it would mean that there are valid designs that have islands, which is not possible. If it was below the validity line, it would mean that there are invalid designs that don't have islands, meaning that there is other reason for invalidity. The only other reason for invalidity that we have is the "no path" reason, but if a design has the "no path" reason, it will always have the "islands" reason too, which makes the option of having the line below the validity line impossible. Therefore, the "islands" line will always be overlapping the validity line, as we can see in the plots.

Model	**Mean Validity (%)**	**Mean Conditional Relative Error (%)**	**FID score**	**Batch Diversity**	**Diversity compared with the training dataset**		
					Coverage	Balance Mean	Balance Std
CGAN KU Leuven V8	60	22.2	186	0.43	0.70	16.65	34.84
CGAN KU Leuven V9	71	22.6	163	0.36	0.78	15.04	31.45
CWGAN KU Leuven V6	88	6.2	199	0.62	0.30	19.83	45.00
CWGAN KU Leuven V9	90	5.5	180	0.28	0.43	20.39	33.64

Table 5.4: Metrics results of the best CGAN and CWGAN models for our problem.

The results of the other metrics for the four best models are shown in Table 5.4. The table shows that the CWGAN models have better validity and conditional relative error results than the CGAN models. This fact can also be observed in the plots of the validity and the Predictor's loss. The validity line of the CWGAN models is always higher than the line of the CGAN models, indicating that the CWGAN models generate more valid designs, and the Predictor's loss is always

lower for the CWGAN models, indicating that they learn better to condition the generation of designs (the difference between the predicted frequencies and the input frequencies is lower). We can also see that the FID scores for all models are relatively similar, with the CWGAN KU Leuven V6 model having the highest score and the CGAN KU Leuven V9 model having the lowest score. This means that the model that generates designs that are closer to the real designs is the CGAN KU Leuven V9 model.

Regarding the **batch diversity**, the CWGAN models have the highest and lowest values, with the CWGAN KU Leuven V6 model having the highest value and the CWGAN KU Leuven V9 model having the lowest value. This means that the CWGAN KU Leuven V6 model generates designs that are less diverse within a batch of 500 designs, while the CWGAN KU Leuven V9 model generates designs that are more diverse.

We can also notice that the CWGAN models have a lower coverage than the CGAN models, which means that the CWGAN models cover fewer clusters from the training dataset than the CGAN models. Therefore, we can conclude that the CWGAN models generate less diverse designs compared with the training dataset than the CGAN models.

Finally, the *balance mean* and *balance std* values show that the number of designs in each cluster is not that balanced in all the models since the *balance std* is higher than the *balance mean* in all of them.

Based on all this results, we can conclude that the best type of model for our problem is the CWGAN since it gives higher validity percentages and lower conditional relative errors, which are the two most important metrics to achieve our goal. Among the CWGAN models, the best model is the CWGAN KU Leuven V9 since it has the highest validity percentage and the lowest conditional relative error. It also has the lowest FID score, which means that it generates designs that are closer to the real designs, the lowest batch diversity, which means that it generates more diverse designs within a batch of 500 designs, and also the highest coverage, meaning that it covers more clusters than the other CWGAN model. However, we cannot say that the CWGAN KU Leuven V9 model is the best model until we analyse the COMSOL simulation results, which we'll do in the next section.

5.4.2.1 COMSOL Simulation Results

In Section 5.1.2.3, we explained that by simulating the designs generated in the COMSOL software, we can assess how well the models perform in generating designs with the desired frequency and also evaluate the accuracy of the Predictor

in predicting the frequencies. Therefore, we used it to simulate the designs generated by the best CGAN and CWGAN models, obtaining their real frequencies to draw conclusions about their performance based on the results. After analysing the results, we have all the tools to finally conclude which model is the best for our problem.

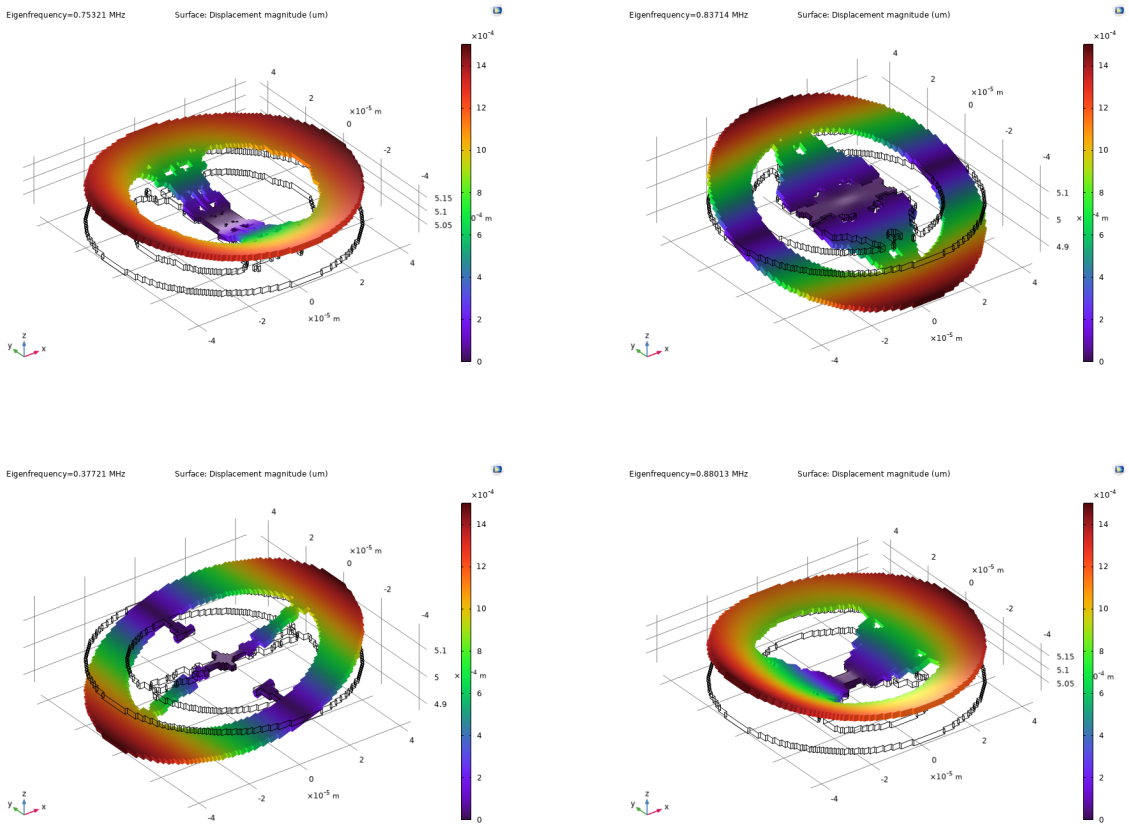

Figure 5.8: Examples of simulations performed by COMSOL for designs generated by the best CGAN and CWGAN models.

Figure 5.8 shows some examples of simulations performed by COMSOL for designs generated by the best CGAN and CWGAN models.

After simulating all designs generated by our best models, the KU Leuven team provided the simulation results, i.e. the real frequencies of the designs. It's important to note we were only interested in the Mode 4 frequency because it was the only frequency we conditioned the generation of designs with. With these results, we calculated the MAE and MRE between the input frequencies and the simulated frequencies by COMSOL, as well as the MAE and MRE between the predicted frequencies by the Predictor and the simulated frequencies. Tables 5.5 and 5.6 show the results of these calculations.

To evaluate the performance of our models, we must look at Table 5.5. The lower

Model	Number of designs	MAE (MHz)	MRE (%)
CGAN KU Leuven V8	149	0.169	24.604
CGAN KU Leuven V9	150	0.138	16.761
CWGAN KU Leuven V6	150	0.089	12.228
CWGAN KU Leuven V9	149	0.077	10.365

Table 5.5: MAE and MRE between Input and Simulated Frequencies for the best CGAN and CWGAN models.

Model	Number of designs	MAE (MHz)	MRE (%)
CGAN KU Leuven V8	149	0.055	8.135
CGAN KU Leuven V9	150	0.056	6.819
CWGAN KU Leuven V6	150	0.068	8.622
CWGAN KU Leuven V9	149	0.061	7.529

Table 5.6: MAE and MRE between Predicted and Simulated Frequencies for the best CGAN and CWGAN models.

the values of the MAE and MRE between the input and simulated frequencies, the better the model is at generating designs with the desired frequency. The values in the table show that the CWGAN models have lower MAE and MRE values than the CGAN models, which means that the CWGAN models are better at conditioning the generation of designs. This fact aligns with the results of the Predictor's loss and the mean conditional relative error, as mentioned in the previous section. However, these two metrics are based on the Predictor, a neural network that can make errors, so they are not perfect measures.

To evaluate the Predictor's performance, we must look at Table 5.6. The lower the values of the MAE and MRE between the predicted and simulated frequencies, the better the Predictor is at predicting the frequencies of the designs. As we can see by the values in the table, all four models have similar MAE and MRE values, and all of them are relatively low, which means that the Predictor is good at predicting the frequencies of the designs and the errors that it makes are not that significant. Therefore, we can conclude that the CWGAN models are the best at conditioning the generation of designs.

Input VS Simulated Frequencies Analyses

To further analyse the results, we created boxplots comparing the input frequencies with the simulated frequencies (Figure 5.9) and the predicted frequencies with the simulated frequencies (Figure 5.10) for the four models. The latter boxplot was created just to analyse the performance of the Predictor in all the four models.

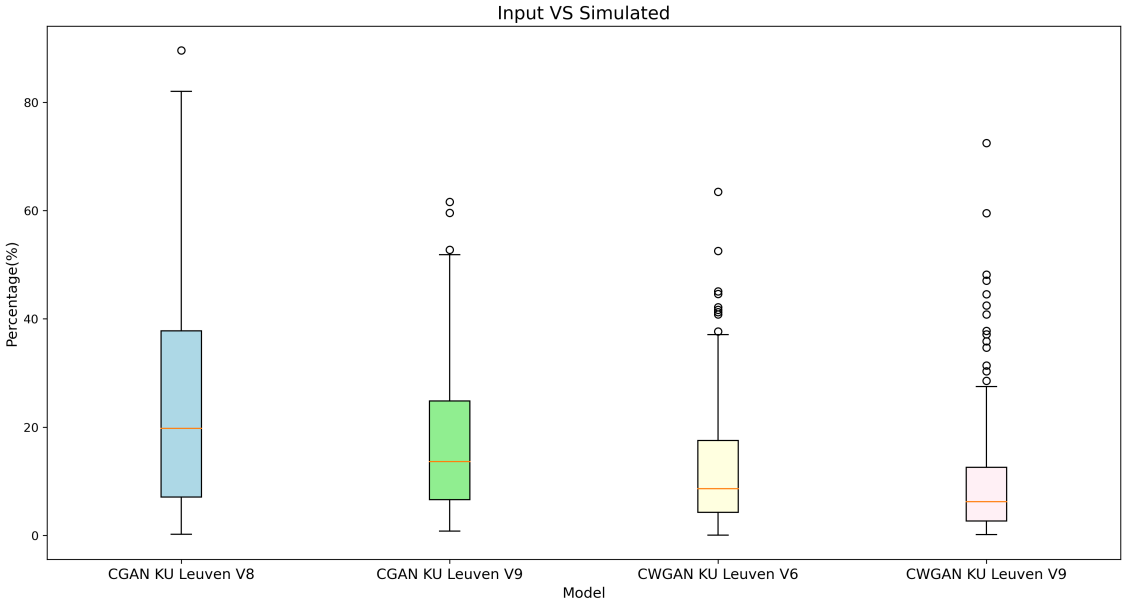

Figure 5.9: Boxplot comparing the input frequencies with the simulated frequencies for the best CGAN and CWGAN models.

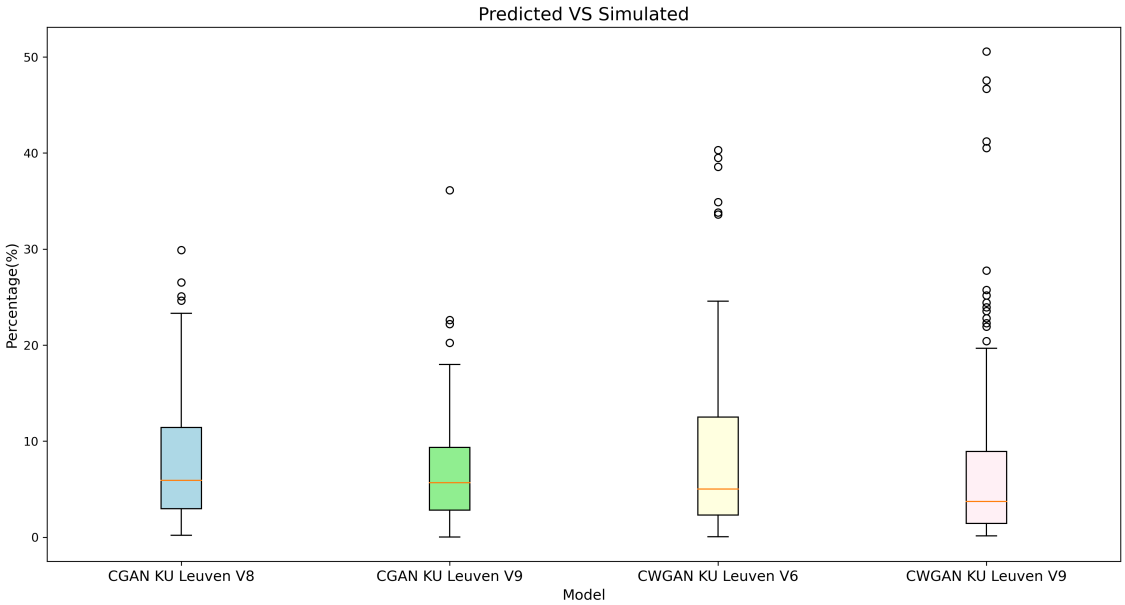

Figure 5.10: Boxplot comparing the predicted frequencies with the simulated frequencies for the best CGAN and CWGAN models.

Regarding the difference between the input frequencies and the simulated frequencies, Figure 5.9 shows that the CWGAN KU Leuven V9 model has the lowest median error and the narrowest spread, indicating more consistent performance with fewer large deviations between the input and simulated frequencies. However, it does exhibit a significant number of outliers, suggesting occasional high errors. The CWGAN KU Leuven V6 model has a higher median error and a wider spread, as well as numerous outliers. However, it shows a more consistent performance than the CGAN models, which have the highest median errors and the widest spreads, indicating poorer performance, especially the CGAN KU Leuven V8 model.

Regarding the difference between the predicted frequencies and the simulated frequencies, Figure 5.10 shows that in the CWGAN KU Leuven V9 model, the Predictor had the lowest median error and also a narrow spread. However, it has a lot of outliers, indicating occasional more significant errors despite a generally low central tendency. The Predictor had similar median errors in the other three models, but the CWGAN KU Leuven V6 model had the highest spread, suggesting higher variability and less reliable predictions.

Input Frequency Distribution Analyses

Other aspect to consider is the frequency distributions of the input frequencies given to the models and the COMSOL simulated frequencies of the designs generated by our best four models.

In Figure 5.11 are displayed the frequency distributions of the input frequencies. We can observe that all the distributions are relatively similar, ranging approximately from 0.2 to 1.2 MHz, except for the CWGAN KU Leuven V9 model, which has a slightly wider range (from 0.1 to 1.4 MHz). The majority of the input frequencies are around 0.5 to 1.0 MHz.

When we analyse Figure 5.12, which shows the frequency distributions of the simulated frequencies, we can see that the distributions are different from each other and from the input frequencies. The ranges are narrower, being approximately from 0.3 to 0.9 MHz, except for the CGAN KU Leuven V9 model, which goes till almost 1.3 MHz. The range within which most of the simulated frequencies fall varies from model to model, but it aligns with the range within which the majority of the input frequencies are (0.5 to 1.0 MHz). This suggests that the models are producing designs with frequencies close to the desired values, as the majority of the simulated frequencies align with the range of the input frequencies. This observation is consistent with our previous analyses.

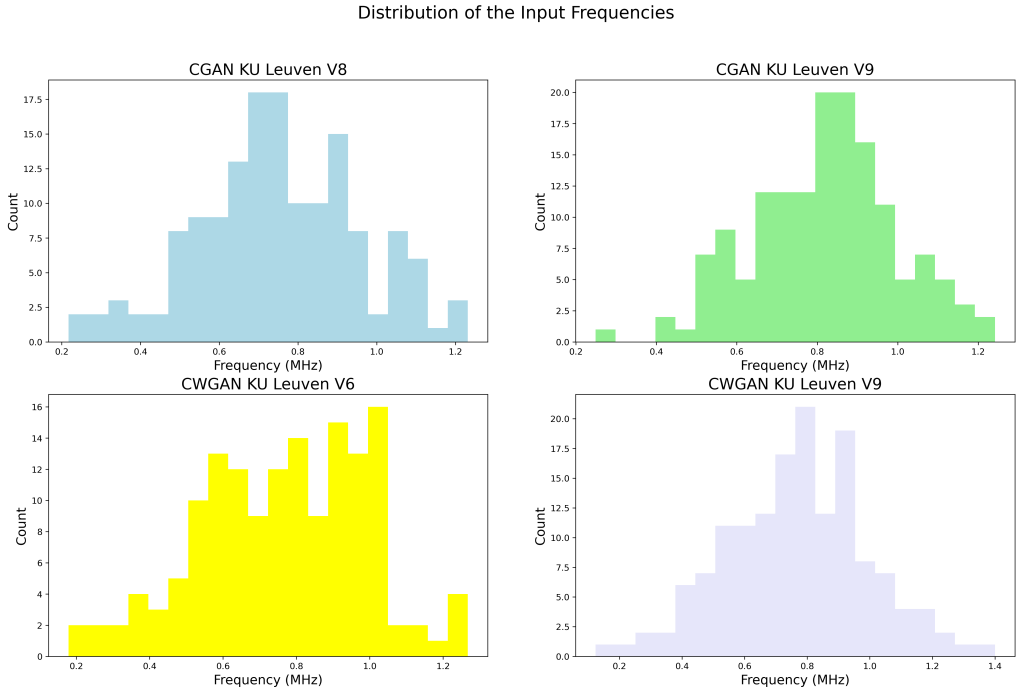

Figure 5.11: Frequency distributions of the input frequencies given to the best CGAN and CWGAN models to generate the designs that were then simulated by COMSOL.

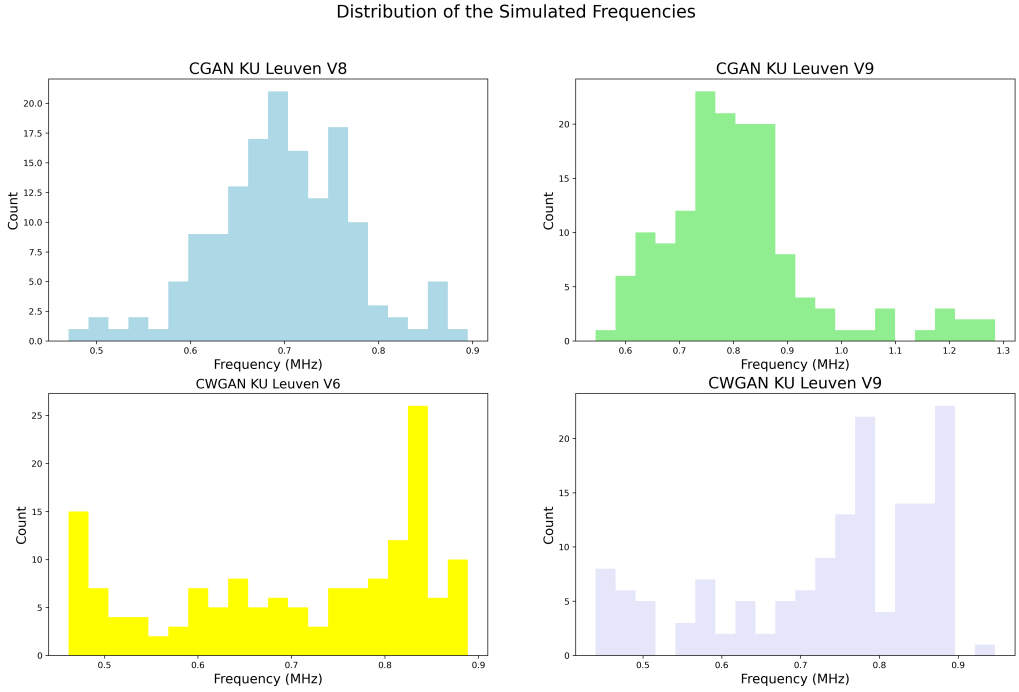

Figure 5.12: Frequency distributions of the simulated frequencies by COMSOL of the designs generated by the best CGAN and CWGAN models.

Designs Analyses

Finally, we analysed the designs with the smallest and largest relative errors between the input and the simulated frequencies (Figure 5.13) and between the

predicted and the simulated frequencies (Figure 5.14) for our best four models.

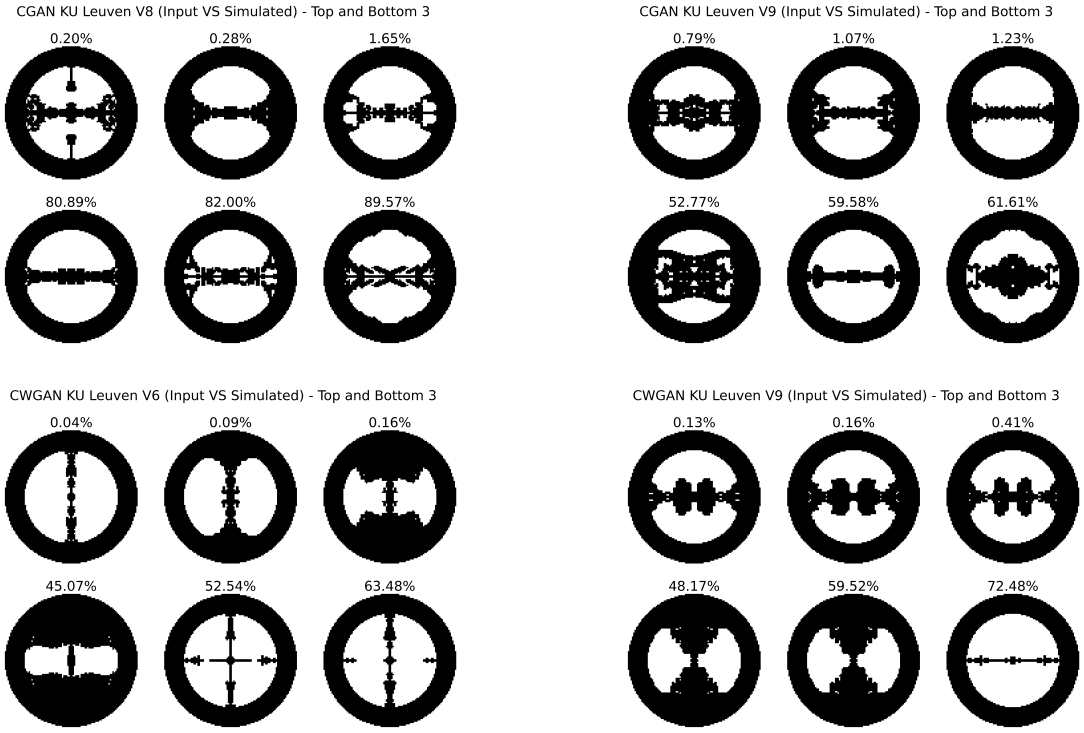

Figure 5.13: Designs with the smallest and largest relative error between the input and the simulated frequencies for the best CGAN and CWGAN models.

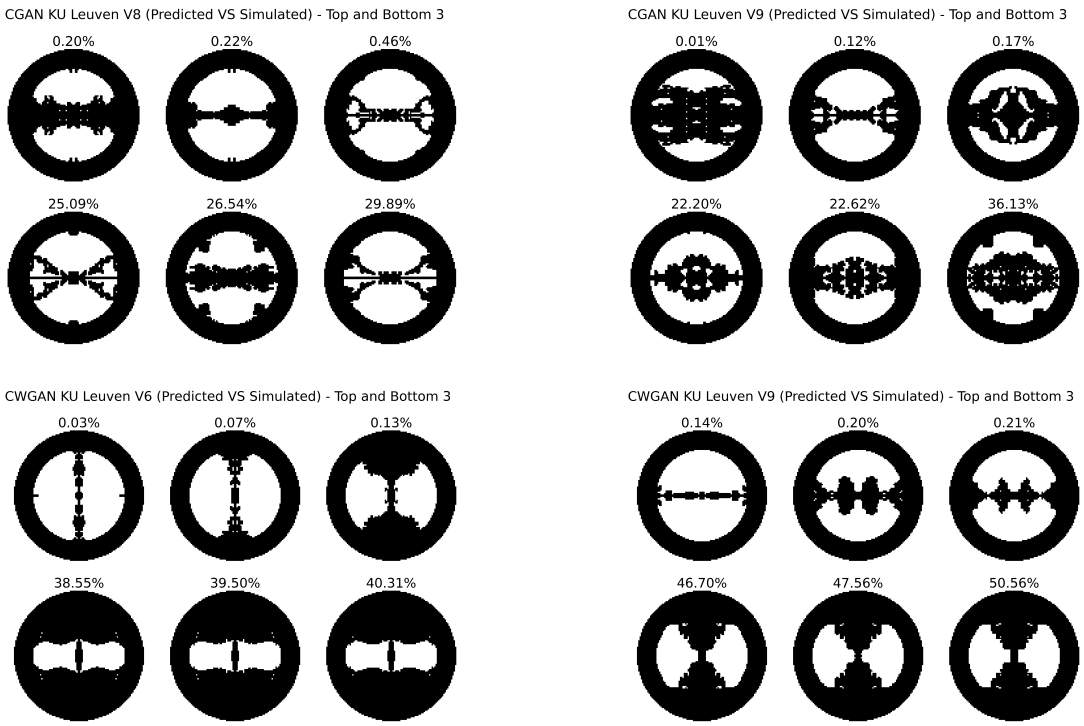

Figure 5.14: Designs with the smallest and largest relative error between the predicted and the simulated frequencies for the best CGAN and CWGAN models.

We can observe that some designs with the smallest and largest relative errors between the input and the simulated frequencies are very similar to those with the smallest and largest relative errors between the predicted and the simulated frequencies by model. For example, the three designs from the CWGAN KU Leuven V9 model with the smallest relative errors between the input and the simulated frequencies are very similar to two of the designs from the same model with the smallest relative errors between the predicted and the simulated frequencies. The same happens with the designs with the largest relative errors from the same model. This suggests that the Predictor is not making significant errors in predicting the frequencies of the designs, as the designs with the smallest and largest relative errors between the input and the simulated frequencies are also the designs with the smallest and largest relative errors between the predicted and the simulated frequencies.

Overall Discussion

Based on all the analyses we made in this section and in the previous one, we can conclude that the best type of model for our problem is the CWGAN, as they generated more valid designs and gave lower conditional relative errors. They also gave lower MAE and MRE values between the input and the simulated frequencies, which means that they are better at conditioning the generation of designs. This fact was also observed in the boxplots comparing the input and the simulated frequencies.

Among the CWGAN models, the best model is the CWGAN KU Leuven V9, as it gave the highest validity percentage, the lowest conditional relative error and the lowest MAE and MRE values between the input and the simulated frequencies. In the boxplot comparing the input and the simulated frequencies, it also showed the lowest median error and the narrowest spread, indicating more consistent performance with fewer large deviations between the input and the simulated frequencies, despite having some outliers. Therefore, we can conclude that the CWGAN KU Leuven V9 model is the best model for our problem. Its architecture is described in detail in Appendix A.

In the following sections, we will present the results of the groups of experiments that we did, which led us to choose the four models we analysed in this section. We will analyse the results of each group of experiments and decide which configuration is the best before continuing to the next group of experiments. These experiments were carried out on both datasets and with both model types, enabling us to identify the best configuration for each model type trained on each dataset.

5.4.3 CGAN with Predictor Experiments on the Berkeley Dataset

5.4.3.1 With or Without Predictor Experiments

In our initial set of experiments, we compared the outcomes of a standard CGAN (without the Predictor) with a CGAN that includes the Predictor. The aim was to determine whether the Predictor could assist the model in generating designs with the desired frequency.

We trained the CGAN with Predictor model in two different ways: by providing all the generated designs (both valid and invalid) to the Predictor, and by only providing the valid designs to the Predictor. Ideally, the model should only be trained using the valid designs, as the invalid designs do not have frequencies. Therefore, if the Predictor is given invalid designs, it would be predicting frequencies that do not actually exist. However, we still wanted to explore the effects of providing all designs to the Predictor, so we trained the model in both ways.

The Predictor that we used in all of these experiments was the one trained by the KU Leuven team on the Berkeley dataset with the resimulated labels given by the COMSOL simulation. The results of this Predictor can be seen in Section 5.2.

It's important to note that during the training of the CGAN models with the Predictor, we made sure to keep the weights of the Predictor frozen. This means that the Predictor was not being trained alongside the CGAN. As a result, if the Predictor was given invalid designs, it would not be updated with the predicted frequencies.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CGAN **without** Predictor	1st	0	0	0	0	16.6	19.7	1.33
	2nd	0		0	0	25.6		
	3rd	0		0	0	16.8		
CGAN **with** Predictor (**all** generated designs)	1st	0	8	0	0	16.4	14.5	2.41
	2nd	23		100	22.6	4.2		
	3rd	0		100	0	22.9		
CGAN **with** Predictor (**only valid** designs)	1st	0	0	0	0	nan	nan	7.30
	2nd	0		0	0	nan		
	3rd	0		100	0	nan		

Table 5.7: Results of the CGAN with and without Predictor experiments on the Berkeley dataset.

In Table 5.7, we can observe the results of these experiments. Note that *V* stands for Validity, *MV* for Mean Validity, *P* for No Path, *I* for Islands, *R* for Results of the Conditional Relative Error, and *MR* for Mean Conditional Relative Error. These acronyms will be used in the following tables as well.

It is evident that the outcomes of all experiments were unsatisfactory. In terms of validity, nearly all of them gave 0% valid designs, and the conditional relative error was also high.

I also noticed by the generated designs of all experiments that all gave mode collapse. Figure 5.15 displays the designs generated by the CGAN with Predictor (all generated designs) model (second training). It's clear that all the designs are nearly identical, indicating mode collapse.

Figure 5.15: Designs generated by the CGAN with Predictor (all generated designs) model (second training).

In addition, the experiment using the CGAN with Predictor (only valid designs) model only produced "nan" relative errors because all the designs were invalid, resulting in no designs being given to the Predictor. This experiment also required a lengthy training time of 7.3 hours and did not give any valid designs. This way, we decided to continue our experiments with the **CGAN with Predictor (all generated designs)** model. This model not only delivered the best results in terms of validity and conditional relative error but also trained in one third of the time it took to train the CGAN with Predictor (only valid designs) model.

5.4.3.2 Predictor Frozen VS Not Frozen Experiments

In this set of experiments, we compared the results of the CGAN with the Predictor when the Predictor's weights were frozen and when they were not. Although the correct way would be to freeze the Predictor's weights, we still wanted to explore the effects of not freezing them.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN with Predictor **Frozen** (all generated designs)	1st	0	8	0	0	16.4	14.5	2.41
	2nd	23		100	22.6	4.2		
	3rd	0		100	0	22.9		
CGAN with Predictor **Not Frozen** (all generated designs)	1st	0	0	0	0	16.7	14.5	2.70
	2nd	0		0	0	4.4		
	3rd	0		0	0	22.3		

Table 5.8: Results of the experiments on the Berkeley dataset with the CGAN with the Predictor Frozen and Not Frozen.

In Table 5.8, we can see the results of these experiments. Both experiments had unsatisfactory outcomes, with nearly all designs being invalid and showing high conditional relative errors. I also observed by the generated designs of both experiments that both gave mode collapse, similar to the previous experiments.

That being said, we found that **the best way to train the CGAN with the Predictor is by freezing the Predictor's weights**. This approach not only gave the best results in terms of validity but also prevented training the Predictor incorrectly. As explained in Section 4.3, this was especially important in the early stages of training, where the generated designs certainly didn't match the input frequency, thus providing misleading information to the Predictor. By freezing the Predictor's weights, it only learns from the real designs in the dataset it was previously trained on, rather than from the generated designs.

5.4.3.3 Experiments Leveraging a Pre-trained GAN Checkpoint

In the next set of experiments, we compared the performance of the CGAN with the Predictor when using a pre-trained GAN checkpoint both as a base and with the checkpoint frozen, meaning only the conditional part of the model was trained. The GAN checkpoint used is the one described in Section 5.4.9. It was the GAN

model trained on the Berkeley dataset that gave the best results. The results of these experiments are presented in Table 5.9.

In the experiment with the frozen pre-trained GAN checkpoint, although the percentage of valid designs was lower, the conditional relative error was improved compared to the model with the unfrozen checkpoint, and the training time was nearly halved. Both scenarios, however, showed signs of mode collapse in the generated designs once again. Despite this, the conditional relative error was lower than in earlier experiments (Tables 5.8 and 5.7), demonstrating that **using a pre-trained GAN checkpoint was beneficial** in our case, as discussed in Section 4.3. By leveraging a GAN checkpoint pre-trained on the Berkeley dataset, the model was able to focus more on conditioning the designs with the desired frequency, resulting in a significant reduction of the conditional relative error.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CGAN with Predictor frozen (all generated designs) and **pre-trained GAN checkpoint**	1st	23	8	62.4	23.4	4.2	9.4	2.51
	2nd	1.8		79.4	1.8	2.9		
	3rd	0		100	0	21.2		
CGAN with Predictor frozen (all generated designs) and **frozen pre-trained GAN checkpoint**	1st	0	1.5	100	0	5.0	6.2	1.35
	2nd	4.6		98	4.6	5.9		
	3rd	0		97.8	0	7.7		

Table 5.9: Results of the experiments on the Berkeley dataset with the CGAN with the Predictor and a pre-trained GAN checkpoint.

5.4.3.4 Non-linear Function in FCD Experiments

In these experiments, we compared the results of the CGAN with the Predictor and a pre-trained GAN checkpoint (with both frozen and unfrozen settings) when using the Tanh activation function in the FCD. The results are summarised in Table 5.10.

For the first time, we observed that the CGAN model did not suffer from mode collapse when using a frozen pre-trained GAN checkpoint and the Tanh activation function in the FCD. Figure 5.16 showcases several generated designs by this model's first training. This configuration also achieved the highest validity across all experiments, with an average validity of 76%. Although the conditional relative error was high, the model's ability to avoid mode collapse and consistently generate a high percentage of valid designs marks a significant improvement.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN with Predictor frozen (all generated designs), pre-trained GAN checkpoint and **Tanh in FCD**	1st	0	26	100	0	2.7	2.6	2.62
	2nd	79		99.8	79	2.5		
	3rd	0		100	0	2.5		
CGAN with Predictor frozen (all generated designs), **frozen** pre-trained GAN checkpoint and **Tanh in FCD**	1st	84	76	99.4	83.8	15.4	16.9	1.33
	2nd	71		98.4	71	18.1		
	3rd	72		91.8	71.8	17.2		

Table 5.10: Results of the experiments on the Berkeley dataset with the CGAN with the Predictor, a pre-trained GAN checkpoint and the Tanh activation function in the FCD.

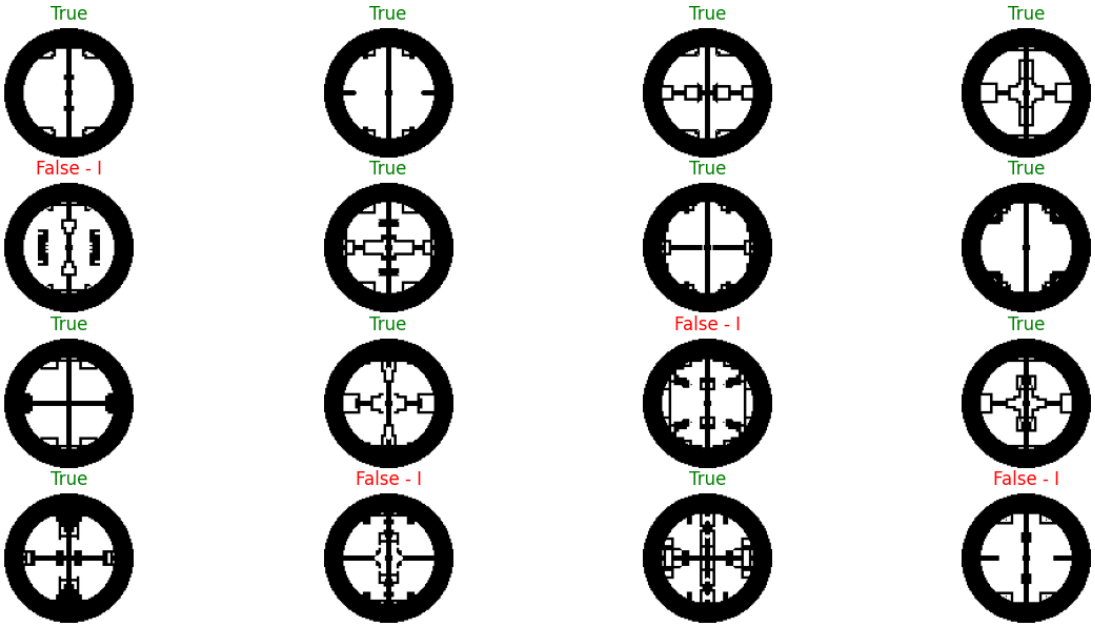

Figure 5.16: Designs generated by the CGAN with Predictor (all generated designs), frozen pre-trained GAN checkpoint and Tanh in FCD model (first training).

The model with the unfrozen pre-trained GAN checkpoint and the Tanh activation function in the FCD also performed well during the second training, achieving 79% validity and a conditional relative error of only 2.5%. However, the model wasn't consistent across subsequent trainings, giving mode collapse.

Despite this, we can conclude that **incorporating the Tanh activation function in the FCD was beneficial** in our case, as mentioned in Section 4.2. It helped prevent mode collapse in certain trainings and delivered the highest validity results to date.

5.4.3.5 Reduced Architecture of FCD Experiments

In the final set of CGAN experiments using the Berkeley dataset, we evaluated the performance of the CGAN with the Predictor by modifying the architecture of the FCD. We retained the best model settings from the previous section, with the only change being a reduction in the FCD architecture from 8 to 6 linear layers. The results of these experiments are summarised in Table 5.11.

Although the model with the reduced FCD architecture achieved a 2% higher mean validity compared to the non-reduced architecture, it also exhibited a significantly higher mean conditional relative error. Consequently, for the Berkeley dataset, the model with the **non-reduced FCD architecture proved to be the better option**, as it produced the lowest conditional relative error while still maintaining a high mean validity.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint, and Tanh in FCD (8 linear layers)	1st	84	76	99.4	83.8	15.4	16.9	1.33
	2nd	71		98.4	71.0	18.1		
	3rd	72		91.8	71.8	17.2		
CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint, Tanh and **reduced architecture of FCD** (6 linear layers)	1st	77	78	99.8	76.6	32.5	24	1.30
	2nd	85		97.0	85.0	19.4		
	3rd	71		98.4	71.0	19.1		

Table 5.11: Results of the experiments on the Berkeley dataset using the CGAN with the Predictor, a frozen pre-trained GAN checkpoint, the Tanh activation function, and varying FCD architectures (reduced and non-reduced).

5.4.4 CWGAN with Predictor Experiments on the Berkeley Dataset

5.4.4.1 With or Without Predictor Experiments

In the first set of CWGAN experiments on the Berkeley dataset, we compared the performance of a standard CWGAN (without the Predictor) with a CWGAN that incorporates the Predictor. As in the CGAN experiments, we trained the CWGAN with the Predictor in two different ways: one by providing all generated designs (both valid and invalid) to the Predictor, and the other by providing only valid designs. In both cases, the Predictor's weights were frozen during training. The results are presented in Table 5.12.

All models demonstrated satisfactory outcomes, with high mean validity percentages and no mode collapse. However, the mean conditional relative error remained high across all experiments:

- The model without the Predictor had the highest mean conditional relative error (28.4%), indicating that the Predictor helps the model better condition the generated designs on specific frequencies.
- The model with the Predictor (only valid designs) produced strong results in terms of mean validity (70%). However, it required significantly longer training time (13.6 hours), more than twice the time of the other two models, and still exhibited a high mean conditional relative error (24.2%).
- The model with the Predictor (all generated designs), while having the lowest mean validity (66%), achieved the lowest mean conditional relative error (19.8%). Given the relatively small difference in validity and the substantially lower conditional relative error, we chose to proceed with the **CWGAN with the Predictor (all generated designs)** model for the next experiments.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CWGAN **without** Predictor	1st	72	70	91.6	72.2	27.2	28.4	5.26
	2nd	70		86.8	69.6	24.3		
	3rd	68		86.6	67.6	33.6		
CWGAN **with** Predictor (**all** generated designs)	1st	72	66	90.2	72.2	19.0	19.8	6.70
	2nd	58		83.2	57.6	20.2		
	3rd	68		83.6	68.4	20.1		
CWGAN **with** Predictor (**only valid** designs)	1st	66	70	80.0	66.0	22.6	24.2	13.6
	2nd	68		84.8	68.4	25.1		
	3rd	76		97.0	76.4	25.0		

Table 5.12: Results of the CWGAN with and without Predictor experiments on the Berkeley dataset.

5.4.4.2 Predictor Frozen VS Not Frozen Experiments

In the next set of experiments, we compared the performance of the CWGAN with the Predictor when the Predictor's weights were frozen and when they were not. The results are presented in Table 5.13.

As we can observe, both models had similar outcomes in terms of validity, but in terms of the conditional relative error the model with the unfrozen Predictor gave better results. One possible reason for this is that the Predictor learned from both correct and incorrect examples, i.e. designs that do not match the target frequency provided as input. In the early stages of training, the generated designs likely do not follow the input frequency, yet the Predictor is being trained as if they do. This allows the model to learn from mistakes, ultimately reducing the relative error. Therefore, despite the higher conditional relative error, we will proceed with the **model with the Predictor frozen** for the next experiments.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CWGAN with Predictor **Frozen** (all generated designs)	1st	72	66	90.2	72.2	19.0	19.8	6.7
	2nd	58		83.2	57. 6	20.2		
	3rd	68		83.6	68.4	20.1		
CWGAN with Predictor **Not Frozen** (all generated designs)	1st	67	66	83.4	67.2	15.0	16.3	6.9
	2nd	62		78.4	62.0	16.4		
	3rd	69		87.2	69.2	17.6		

Table 5.13: Results of the experiments on the Berkeley dataset with the CWGAN with the Predictor Frozen and Not Frozen.

5.4.4.3 Experiments Leveraging a Pre-trained WGAN Checkpoint

In this set of experiments using the Berkeley dataset, we compared the performance of the CWGAN with the Predictor when using a pre-trained WGAN checkpoint as a base, and when freezing the checkpoint so that only the conditional part of the model was trained. The WGAN checkpoint referenced is the one described in Section 5.4.10. This was the WGAN model trained on the Berkeley dataset that achieved the best results. The outcomes of these experiments are summarised in Table 5.14.

The model with the frozen pre-trained WGAN checkpoint achieved the highest mean validity (79%) and the lowest mean conditional relative error (5.1%) so far. Additionally, it required significantly less training time (1.45 hours), as only the conditional part of the model was trained. These findings suggest that, in our case, **not only using a pre-trained WGAN checkpoint was beneficial, but also freezing it**, as discussed in Section 4.3. Freezing the checkpoint allowed the model to focus exclusively on learning the conditional component, which led to better

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CWGAN with Predictor frozen (all generated designs) and **pre-trained WGAN checkpoint**	1st	75	70	85.4	74.8	22.9	19.3	6.53
	2nd	72		86.6	72.4	19.9		
	3rd	64		77.4	64.2	15.0		
CWGAN with Predictor frozen (all generated designs) and **frozen pre-trained WGAN checkpoint**	1st	79	79	84.0	79.2	5.6	5.1	1.45
	2nd	81		86.6	81.0	3.7		
	3rd	76		79.8	76.4	5.9		

Table 5.14: Results of the experiments on the Berkeley dataset with the CWGAN with the Predictor and a pre-trained WGAN checkpoint.

results in terms of conditional relative error while maintaining a high percentage of valid generated designs.

The only drawback of this model is that the generated designs tend to be relatively similar, as we can observe by the designs displayed in Figure 5.17.

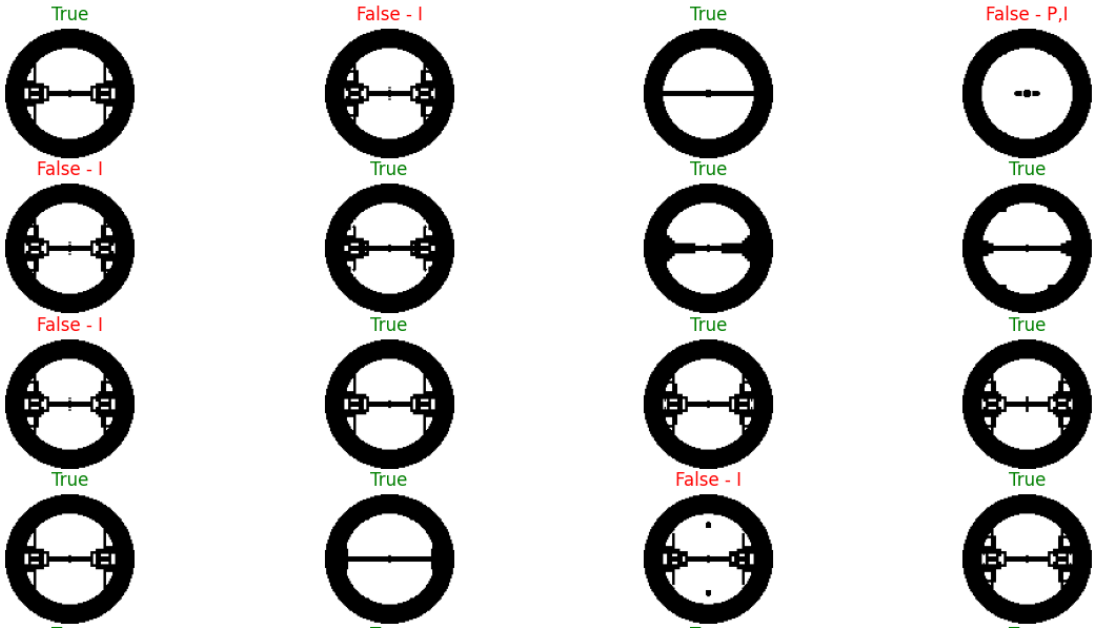

Figure 5.17: Designs generated by the CWGAN with Predictor (all generated designs) with frozen pre-trained WGAN checkpoint (second training).

5.4.4.4 Non-linear Function in FCD Experiments

In the following experiments, we compared the results of the CWGAN with the Predictor and a pre-trained WGAN checkpoint (with both frozen and unfrozen

settings) when using the Tanh activation function in the FCD. The results are presented in Table 5.15.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CWGAN with Predictor frozen (all generated designs), pre-trained WGAN checkpoint and **Tanh in FCD**	1st	66	71	79.8	66.4	17.9	17.1	6.64
	2nd	72		86.4	72.0	15.0		
	3rd	74		86.0	74.0	18.4		
CWGAN with Predictor frozen (all generated designs), **frozen** pre-trained WGAN checkpoint and **Tanh in FCD**	1st	81	82	87.4	81.2	8.1	7.5	1.46
	2nd	84		89.4	83.8	7.6		
	3rd	80		84.2	80.2	6.7		

Table 5.15: Results of the experiments on the Berkeley dataset with the CWGAN with the Predictor, a pre-trained WGAN checkpoint and the Tanh activation function in the FCD.

Compared with the table from the previous section (Table 5.14), we can see that both models achieved higher mean validity and lower mean conditional relative error, proving that using the Tanh activation function in the FCD was advantageous, as discussed in Section 4.2.

Similar to the last round of experiments, the model with the frozen pre-trained WGAN checkpoint gave better results in terms of validity and conditional relative error than the model with the unfrozen checkpoint. As expected, the training time was also significantly reduced, as only the conditional part of the model was trained.

In comparison with the last experiments, the model with the frozen pre-trained WGAN checkpoint and the Tanh activation function in the FCD generated a higher percentage of valid designs, with an average validity of 82% compared to the previous 79% obtained by the frozen version without the Tanh. However, the conditional relative error was slightly higher, with an average of 7.5% compared to the previous 5.1%. Since both mean validities are high, we have decided to proceed with the model showing the lower conditional relative error for the next experiments, the **model with the frozen pre-trained WGAN checkpoint and without the Tanh activation function in the FCD**.

5.4.4.5 Reduced Architecture of FCD Experiments

In this final set of experiments using the Berkeley dataset, we compared the performance of the best CWGAN with the Predictor model so far with the same model but with a reduced architecture of the FCD. Similar to the CGAN experiments,

the FCD architecture was reduced from 8 to 6 linear layers. The results can be found in Table 5.16.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint, and FCD (8 linear layers)	1st	79	79	84.0	79.2	5.6	5.1	1.45
	2nd	81		86.6	81.0	3.7		
	3rd	76		79.8	76.4	5.9		
CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint, and **reduced architecture of FCD** (6 linear layers)	1st	79	82	81.2	78.6	3.9	3.8	1.48
	2nd	76		82.4	76.2	4.6		
	3rd	92		98.8	92.2	2.8		

Table 5.16: Results of the experiments on the Berkeley dataset using the CWGAN with the Predictor, a frozen pre-trained WGAN checkpoint and varying FCD architectures (reduced and non-reduced).

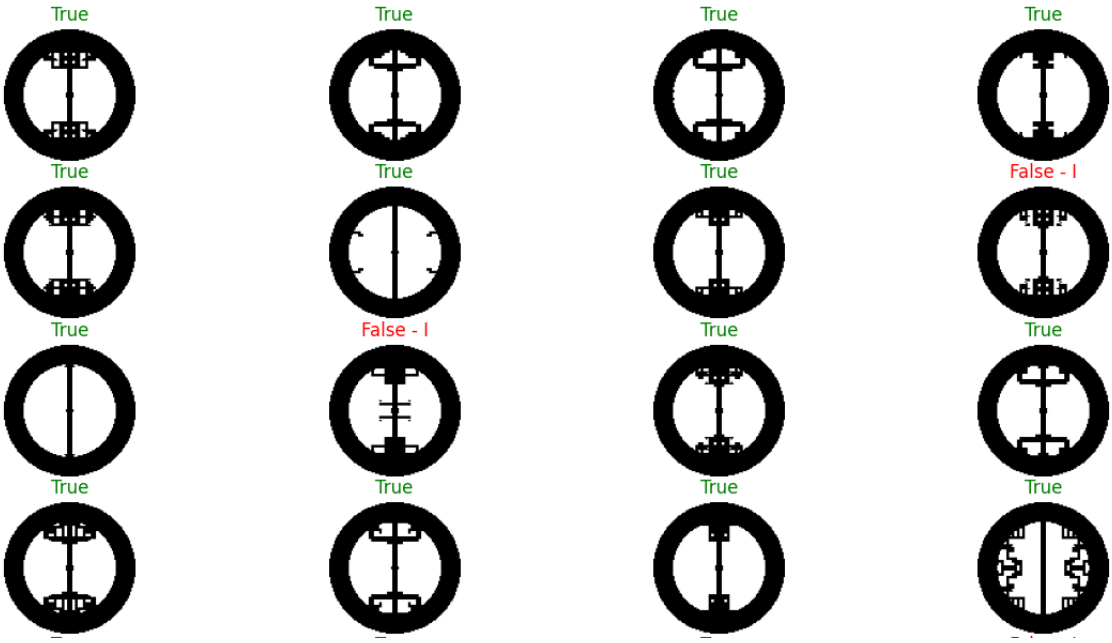

Figure 5.18: Designs generated by the CWGAN with Predictor (all generated designs) with frozen pre-trained WGAN checkpoint and reduced FCD architecture (third training).

The model with the reduced FCD architecture achieved a higher mean validity (82%) compared to the model with the non-reduced architecture (79%), and also a lower conditional relative error (3.8% compared to 5.1%). These results demonstrate that **slightly reducing the architecture of the FCD was beneficial** in this case, as discussed in Section 4.2. By reducing the architecture, the model

preserved the essential information of the label while still transforming it into a higher-dimensional space. This allowed the generator to better produce designs based on the desired frequency.

The only downside of this model is that, like the model with the non-reduced FCD architecture, the generated designs tend to be relatively similar, as we can observe from the designs displayed in Figure 5.18.

5.4.5 Overall Conclusions on the Berkeley Dataset Experiments

After analysing the results of all experiments trained on the Berkeley dataset (Sections 5.4.3 and 5.4.4), we can draw the following conclusions:

- The configuration that worked best for the CGAN models was: CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint and Tanh activation function in the FCD (Section 5.4.3.4).
- The configuration that worked best for the CWGAN models was: CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint and reduced architecture of the FCD (Section 5.4.4.5).
- On the Berkeley dataset, the CWGAN was the best model type for our problem. It achieved higher mean validity, lower mean conditional relative error, and more stability (no mode collapse) compared to the CGAN models.
- The designs generated by the CWGAN models tended to be less diverse than those generated by the CGAN models, as we can observe in the designs displayed in Figures 5.16 and 5.18.

5.4.6 CGAN with Predictor Experiments on the KU Leuven Dataset

5.4.6.1 With or Without Predictor Experiments

Similar to the Berkeley dataset experiments, we started by comparing the performance of a standard CGAN (without the Predictor) with a CGAN that incorporates the Predictor. The Predictor that we used in all of these experiments was the one trained by the KU Leuven team on the KU Leuven dataset. The results of this Predictor are displayed in Section 5.2.

We trained the CGAN with the Predictor in two different ways: one by providing all generated designs (both valid and invalid) to the Predictor, and the other by providing only valid designs. In both cases, the Predictor's weights were frozen during training. The results of these experiments are summarised in Table 5.17.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN **without** Predictor	1st	50	17	56.0	50.2	16.0	25.4	0.89
	2nd	0		0.0	0.0	25.7		
	3rd	0		51.8	0.0	34.4		
CGAN **with** Predictor (**all** generated designs)	1st	57	19	57.6	56.7	34.7	29.9	1.46
	2nd	0		0.0	0.0	28.6		
	3rd	0		0.0	0.0	26.5		
CGAN **with** Predictor (**only valid** designs)	1st	2	32	2.0	1.6	26.2	40.6	4.91
	2nd	57		100	56.8	46.0		
	3rd	36		100	35.8	49.5		

Table 5.17: Results of the CGAN with and without Predictor experiments on the KU Leuven dataset.

Unfortunately, similar to the experiments with the Berkeley dataset, all three models experienced mode collapse. The average validity was low in all models, and the average conditional relative error was high.

Regarding validity, the model that gave better results was the one where we only passed valid designs to the Predictor. However, this model also had the worst results in terms of conditional relative error and training time. Consequently, we decided to proceed with the model that showed the second-best results in terms of validity, along with lower conditional relative error and training time: the **model with the Predictor (all generated designs)**.

5.4.6.2 Predictor Frozen VS Not Frozen Experiments

In this set of experiments, we compared the performance of the CGAN with the Predictor when the Predictor's weights were frozen and when they were not. The results are displayed in Table 5.18.

It's evident that the model with the unfrozen Predictor showed slightly better results in terms of validity and conditional relative error. However, both models experienced mode collapse once again.

Despite the slightly better results of the model with the unfrozen Predictor, we have decided to proceed with the **model using the frozen Predictor** for the next experiments. This decision is based on the understanding that training the Predictor with incorrect examples, as discussed in Section 5.4.4.2, would not be meaningful.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN with Predictor **Frozen** (all generated designs)	1st	57	19	57.6	56.7	34.7	29.9	1.46
	2nd	0		0.0	0.0	28.6		
	3rd	0		0.0	0.0	26.5		
CGAN with Predictor **Not Frozen** (all generated designs)	1st	37	27	38.0	36.8	18.4	27.7	1.68
	2nd	43		44.2	43.2	36.3		
	3rd	0		0.0	0.0	28.3		

Table 5.18: Results of the experiments on the KU Leuven dataset with the CGAN with the Predictor Frozen and Not Frozen.

5.4.6.3 Experiments Leveraging a Pre-trained GAN Checkpoint

In the following experiments, we compared the performance of the CGAN with the Predictor when using a pre-trained GAN checkpoint as a base, and when freezing the checkpoint so that only the conditional part of the model was trained. The GAN checkpoint used in these experiments was the one described in Section 5.4.9. This was the GAN model trained on the KU Leuven dataset that achieved the best results. The outcomes of these experiments are summarised in Table 5.19.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN with Predictor frozen (all generated designs) and **pre-trained GAN checkpoint**	1st	9	69	36.4	9.2	17.9	28.8	1.60
	2nd	100		100	100	50.2		
	3rd	97		100	97.0	18.4		
CGAN with Predictor frozen (all generated designs) and **frozen pre-trained GAN checkpoint**	1st	0	2	0.0	0.0	5.8	8.6	0.72
	2nd	5		31.2	4.6	7.3		
	3rd	0		0.0	0.0	12.6		

Table 5.19: Results of the experiments on the KU Leuven dataset with the CGAN with the Predictor and a pre-trained GAN checkpoint.

Unfortunately, both models experienced mode collapse once again. However, in the model with the unfrozen checkpoint, almost all the few different designs generated were valid in the second and third training. In the second training, the model consistently generated the same valid design, resulting in a 100% validity. The design generated is presented in Figure 5.19. This led to a significantly higher

mean validity (69%) compared to the model with the frozen checkpoint (2%), where almost all the few different generated designs were invalid.

Figure 5.19: Designs generated by the CGAN with Predictor (all generated designs) with a pre-trained GAN checkpoint (second training).

In terms of conditional relative error, the model with the frozen checkpoint gave better results, with a mean of 8.6% compared to the 28.8% of the model with the unfrozen checkpoint. This outcome may be attributed to the fact that the model with the frozen checkpoint was focused exclusively on learning the conditional part of the model, resulting in a lower conditional relative error.

5.4.6.4 Non-linear Function in FCD Experiments

In this next set of experiments on the KU Leuven dataset, we compared the results of the CGAN with the Predictor and a pre-trained GAN checkpoint (with both frozen and unfrozen settings) when using the Tanh activation function in the FCD. The results are presented in Table 5.20.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CGAN with Predictor frozen (all generated designs), pre-trained GAN checkpoint and **Tanh in FCD**	1st	6	69	47.6	6.0	27.2	36.9	1.56
	2nd	99.8		100	99.8	44.7		
	3rd	99.8		100	99.8	38.7		
CGAN with Predictor frozen (all generated designs), **frozen** pre-trained GAN checkpoint and **Tanh in FCD**	1st	66	60	75.2	65.6	22.3	22.2	1.33
	2nd	79		90.0	78.6	20.2		
	3rd	34		38.8	34.0	24.2		

Table 5.20: Results of the experiments on the KU Leuven dataset with the CGAN with the Predictor, a pre-trained GAN checkpoint and the Tanh activation function in the FCD.

As in the same experiments on the Berkeley dataset, a CGAN model didn't experience mode collapse for the first time, which was the model with the frozen pre-trained GAN checkpoint. This model gave the best results in terms of mean conditional relative error (22.2%) compared with the model with the unfrozen checkpoint (36.9%). However, the model with the unfrozen checkpoint achieved a higher mean validity (69%) than the model with the frozen checkpoint (60%). This

is because the model with the unfrozen checkpoint gave mode collapse, generating almost the same valid designs in the second and third trainings, leading to 99.8% validity in these cases. Although in the first training, the model generated almost all invalid designs, leading to 6% validity, the mean validity increased a lot because of the second and third training.

That being said, we decided to proceed with the model that didn't give mode collapse, the **model with the frozen pre-trained GAN checkpoint and the Tanh activation function in the FCD**.

5.4.6.5 Reduced Architecture of FCD Experiments

In this last set of experiments, we compared the performance of the best CGAN with the Predictor model so far with the same model but with a reduced architecture of the FCD. Similar to our previous experiments, the FCD architecture was reduced from 8 to 6 linear layers. The results can be found in Table 5.21.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint, and Tanh in FCD (8 linear layers)	1st	66	60	75.2	65.6	22.3	22.2	1.33
	2nd	79		90.0	78.6	20.2		
	3rd	34		38.8	34.0	24.2		
CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint, Tanh and **reduced architecture of FCD** (6 linear layers)	1st	74	71	85.4	74.0	23.1	22.6	0.84
	2nd	71		83.4	70.8	21.6		
	3rd	68		86.8	68.0	23.1		

Table 5.21: Results of the experiments on the KU Leuven dataset using the CGAN with the Predictor, a frozen pre-trained GAN checkpoint, the Tanh activation function, and varying FCD architectures (reduced and non-reduced).

This time, neither of the models encountered mode collapse, which was a positive outcome. The model with the reduced FCD architecture achieved a higher mean validity (71%) compared to the model with the non-reduced architecture (60%), and also had a shorter training time (0.84h compared to 1.33h). However, it had a slightly higher mean conditional relative error (22.6% compared to 22.2%).

This indicates that **reducing the architecture of the FCD was advantageous** in this case, as the model with the reduced architecture achieved better results, despite the slightly higher mean conditional relative error. Figure 5.20 shows some designs generated by the model with the reduced FCD architecture (second training).

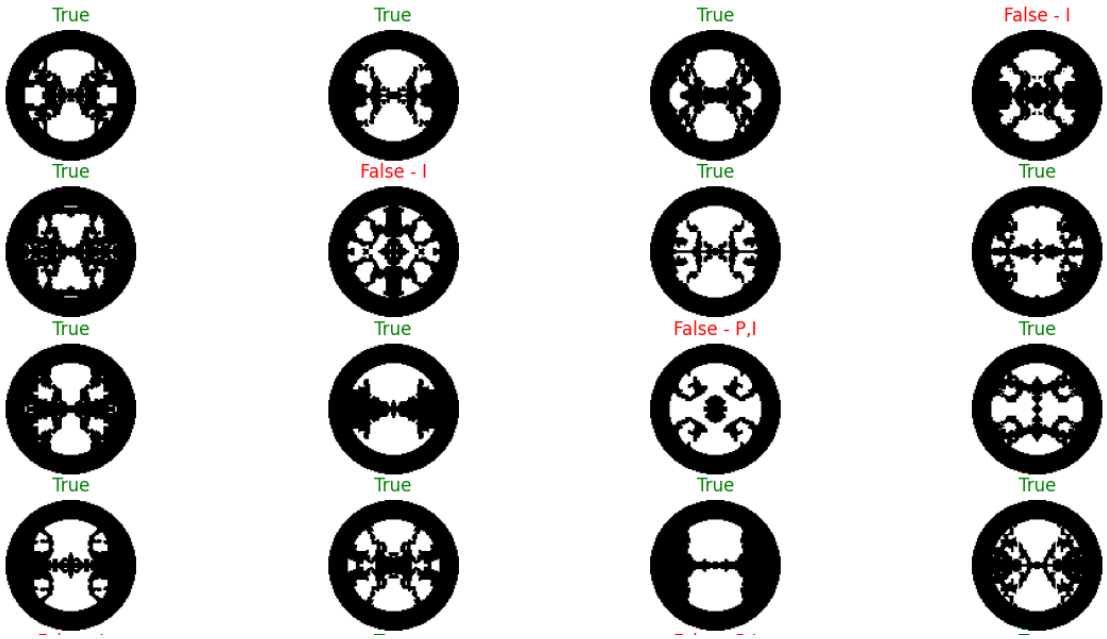

Figure 5.20: Designs generated by the CGAN with Predictor (all generated designs), frozen pre-trained GAN checkpoint, Tanh and reduced architecture of the FCD (second training).

5.4.7 CWGAN with Predictor Experiments on the KU Leuven Dataset

5.4.7.1 With or Without Predictor Experiments

In the first set of CWGAN experiments on the KU Leuven dataset, we compared the performance of a standard CWGAN (without the Predictor) with a CWGAN incorporating the Predictor. The Predictor used in all of these experiments was the one trained by the KU Leuven team on the KU Leuven dataset. The results of this Predictor are displayed in Section 5.2.

Once again, we trained the CWGAN with the Predictor in two different ways: one by providing all generated designs (both valid and invalid) to the Predictor and the other by providing only valid designs. In both cases, the Predictor's weights were frozen during training. The results of these experiments are summarised in Table 5.22.

All three models did not experience mode collapse, which was a positive outcome compared with the CGAN models.

The model without the Predictor achieved the worst results regarding mean validity (62%) and conditional relative error (28.4%).

The model with the Predictor (all generated designs) achieved the best results in terms of mean validity (68%). However, the model with the Predictor (only valid designs) had a lower mean conditional relative error (20.8% compared to 24.1%). Regarding training time, the model with the Predictor (only valid designs) took more than twice as long to train compared to the model with the Predictor

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CWGAN **without** Predictor	1st	63	62	73.8	62.6	35.0	28.4	3.34
	2nd	58		71.9	58.2	25.0		
	3rd	66		76.8	76.8	25.2		
CWGAN **with** Predictor (**all** generated designs)	1st	73	68	82.6	73.4	31.7	24.1	4.10
	2nd	65		81.8	65.4	24.4		
	3rd	65		74.8	64.8	16.1		
CWGAN **with** Predictor (**only valid** designs)	1st	61	63	73.2	61.4	26.9	20.8	8.23
	2nd	63		78.4	63.4	17.7		
	3rd	66		75.8	66.0	17.9		

Table 5.22: Results of the CWGAN with and without Predictor experiments on the KU Leuven dataset.

(all generated designs).

Therefore, we decided to proceed with the model that showed the best validity results: the **model with the Predictor (all generated designs)**.

5.4.7.2 Predictor Frozen VS Not Frozen Experiments

In the next set of experiments, we compared the performance of the CWGAN with the Predictor when the Predictor's weights were frozen and when they were not. The results are presented in Table 5.23.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CWGAN with Predictor **Frozen** (all generated designs)	1st	73	68	82.6	73.4	31.7	24.1	4.10
	2nd	65		81.8	65.4	24.4		
	3rd	65		74.8	64.8	16.1		
CWGAN with Predictor **Not Frozen** (all generated designs)	1st	61	66	69.4	60.8	22.4	18.7	4.35
	2nd	73		83.8	73.4	19.3		
	3rd	64		76.0	63.6	14.5		

Table 5.23: Results of the experiments on the KU Leuven dataset with the CWGAN with the Predictor Frozen and Not Frozen.

The model with the frozen Predictor achieved slightly better mean validity

(68%) than the model with the unfrozen Predictor (66%). However, the model with the unfrozen Predictor had a lower mean conditional relative error (18.7% compared to 24.1%). As explained in previous sections, this fact can be attributed to the predictor's training with possibly incorrect examples, especially at the beginning of training, which is not meaningful. Therefore, we proceeded with the **model using the frozen Predictor** for the following experiments.

5.4.7.3 Experiments Leveraging a Pre-trained WGAN Checkpoint

In this set of experiments, we compared the performance of the CWGAN with the Predictor when using a pre-trained WGAN checkpoint as a base and when freezing the checkpoint so that only the conditional part of the model was trained. The WGAN checkpoint used in these experiments was the one described in Section 5.4.10. This was the WGAN model trained on the KU Leuven dataset that achieved the best results. The outcomes of these experiments are summarised in Table 5.24.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CWGAN with Predictor frozen (all generated designs) and **pre-trained WGAN checkpoint**	1st	52	60	65.8	52.0	37.5	30.1	4.12
	2nd	79		88.2	78.8	34.6		
	3rd	50		60.4	50.4	18.3		
CWGAN with Predictor frozen (all generated designs) and **frozen pre-trained WGAN checkpoint**	1st	89	88	98.0	89.0	6.3	6.2	1.05
	2nd	87		100	87.4	7.0		
	3rd	89		99.8	89.0	5.3		

Table 5.24: Results of the experiments on the KU Leuven dataset with the CWGAN with the Predictor and a pre-trained WGAN checkpoint.

The **model with the frozen checkpoint** achieved a higher mean validity (88%) than the model with the unfrozen checkpoint (60%) and also had a lower mean conditional relative error (6.2% compared to 30.1%). Regarding training time, it also had a much shorter training time (1.05h compared to 4.12h), as expected, since only the conditional part of the model was trained. This was the model that gave the best results so far, so we decided to proceed with it for the next experiments.

Once again, the only downside of this model is that the designs generated are similar to each other and not very diverse, as shown in Figure 5.21.

5.4.7.4 Non-linear Function in FCD Experiments

In the following experiments, we compared the results of the CWGAN with the Predictor and a pre-trained WGAN checkpoint (with both frozen and unfrozen

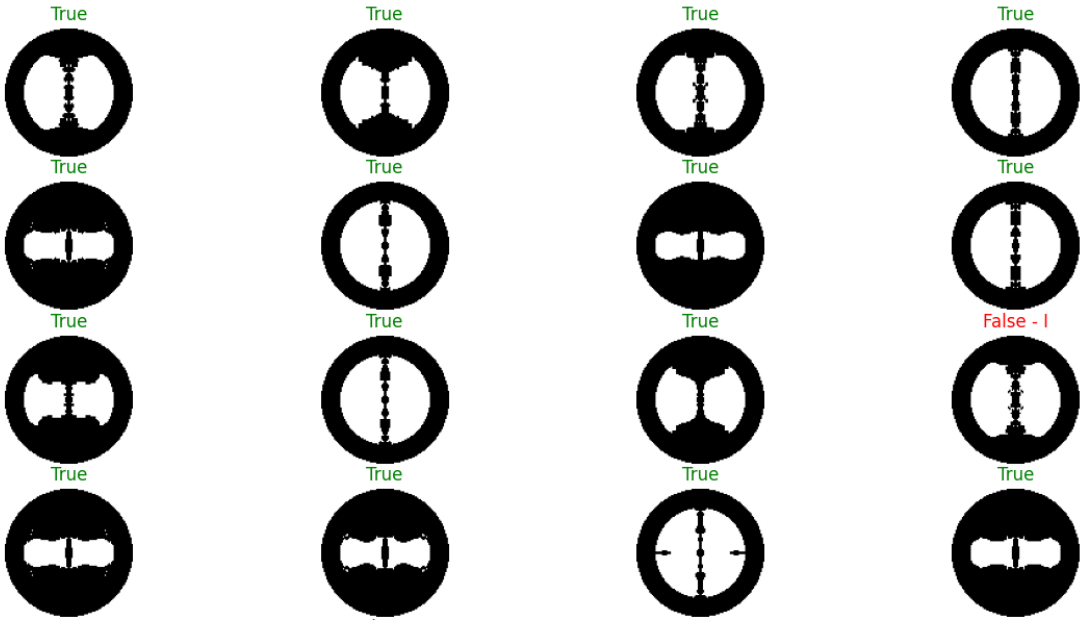

Figure 5.21: Designs generated by the CWGAN with Predictor (all generated designs) with a frozen pre-trained WGAN checkpoint (third training).

settings) when using the Tanh activation function in the FCD. The results are presented in Table 5.25.

Model	**Training Round**	**Validity (%)**				**Conditional Relative Error (%)**		**Training Time (hours)**
		V	MV	P	I	R	MR	
CWGAN with Predictor frozen (all generated designs), pre-trained WGAN checkpoint and **Tanh in FCD**	1st	59	62	64.0	58.8	19.2	24.2	4.05
	2nd	62		72.8	61.8	23.0		
	3rd	65		74.4	64.8	30.5		
CWGAN with Predictor frozen (all generated designs), **frozen** pre-trained WGAN checkpoint and **Tanh in FCD**	1st	95	90	99.2	95.0	8.3	9.2	1.00
	2nd	88		91.4	88.0	8.5		
	3rd	88		94.6	88.2	10.8		

Table 5.25: Results of the experiments on the KU Leuven dataset with the CWGAN with the Predictor, a pre-trained WGAN checkpoint and the Tanh activation function in the FCD.

The model with the frozen checkpoint achieved significantly better results in terms of mean validity (90%) and mean conditional relative error (9.2%) compared to the model with the unfrozen checkpoint (62% mean validity and 24.2% mean conditional relative error). This model also had a much shorter training time (1.00h compared to 4.05h), as expected, since only the conditional part was trained.

When comparing this model with the frozen WGAN checkpoint and the Tanh

in the FCD with the best model so far (frozen WGAN checkpoint and no Tanh in the FCD), the model with the Tanh achieved a higher mean validity (90% compared to 88%), but a higher mean conditional relative error (9.2% compared to 6.2%). Since the difference in mean conditional relative error was more significant than the difference in mean validity, we chose to proceed with the **model without the Tanh in the FCD** for the following experiments.

5.4.7.5 Reduced Architecture of FCD Experiments

In this last set of experiments, we compared the performance of the best CWGAN with the Predictor model so far with the same model but with a reduced architecture of the FCD. Similar to our previous experiments, the FCD architecture was reduced from 8 to 6 linear layers. The results can be found in Table 5.26.

Model	Training Round	Validity (%)				Conditional Relative Error (%)		Training Time (hours)
		V	MV	P	I	R	MR	
CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint, and FCD (8 linear layers)	1st	89	88	98.0	89.0	6.3	6.2	1.05
	2nd	87		100	87.4	7.0		
	3rd	89		99.8	89.0	5.3		
CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint, and **reduced architecture of FCD** (6 linear layers)	1st	97	90	99.0	96.8	5.7	5.5	1.00
	2nd	80		100	79.8	5.5		
	3rd	93		99.2	93.4	5.4		

Table 5.26: Results of the experiments on the KU Leuven dataset using the CWGAN with the Predictor, a frozen pre-trained WGAN checkpoint and varying FCD architectures (reduced and non-reduced).

The model with the reduced FCD architecture achieved a higher mean validity (90%) compared to the model with the non-reduced architecture (88%), and also had a lower mean conditional relative error (5.5% compared to 6.2%). This proves that, in this case, **reducing the architecture of the FCD was advantageous**, as the model with the reduced architecture achieved better results.

However, this model also shares the same downside as the previous best model: the designs generated are similar to each other and not very diverse, as shown in Figure 5.22.

5.4.8 Overall Conclusions on the KU Leuven Dataset Experiments

After analysing the results of all experiments trained on the KU Leuven dataset (Sections 5.4.6 and 5.4.7), we can draw the following conclusions:

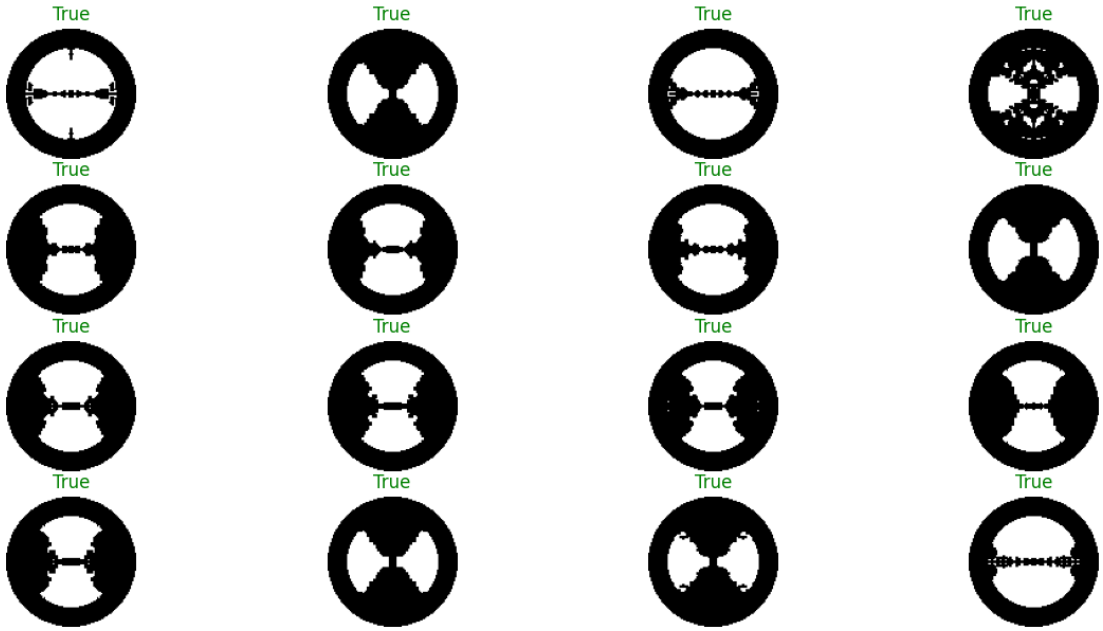

Figure 5.22: Designs generated by the CWGAN with Predictor (all generated designs) with a frozen pre-trained WGAN checkpoint and reduced FCD architecture (third training).

- The configuration that worked best for the CGAN models was: CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint, Tanh activation function and reduced architecture of the FCD (Section 5.4.6.5).

- The configuration that worked best for the CWGAN models was: CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint and reduced architecture of the FCD (Section 5.4.7.5).

- On the KU Leuven dataset, the CWGAN was the best model type for our problem. It achieved higher mean validity, lower mean conditional relative error, and more stability (no mode collapse) compared to the CGAN models.

- The designs generated by the CWGAN models tended to be less diverse than those generated by the CGAN models, as we can observe in the designs displayed in Figures 5.20 and 5.22.

Now that we have analysed the results of all CGAN and CWGAN experiments conducted on both datasets, we'll present in the following sections the results of some GAN and WGAN experiments that led us choose the best configurations for each model type trained on each dataset. The best configurations identified for the GAN and WGAN models were then implemented in the CGAN and CWGAN models that we have just shown, respectively, to achieve the best possible results.

5.4.9 GAN Experiments

During our experiments with the GAN model, we came to the conclusion that the best model configuration varied depending on the dataset.

In the case of the model trained on the Berkeley dataset, the best configuration was the one displayed in Table 5.27. Note that *G* stands for Generator and *D* for Discriminator. These acronyms we'll also be used in the following tables.

The model was trained three times and the mean validity was 76%. Figure 5.23 shows the progress of the losses and the validity during the second training, and Figure 5.24 shows some designs generated by the model.

Number of Epochs		200
Batch Size	G and D Reals	256
	D Fakes	128
Learning Rate	G	*CosineAnnealingWarmRestarts(opt_g, 25, 1, 0.0050)*
	D	*CosineAnnealingWarmRestarts(opt_d, 25, 1, 0.00025)*
Beta 1		0.5
Beta 2		0.9999
Kernel Size of the Conv2d Layers	G	5
	D	3

Table 5.27: Best GAN configuration for the Berkeley Dataset.

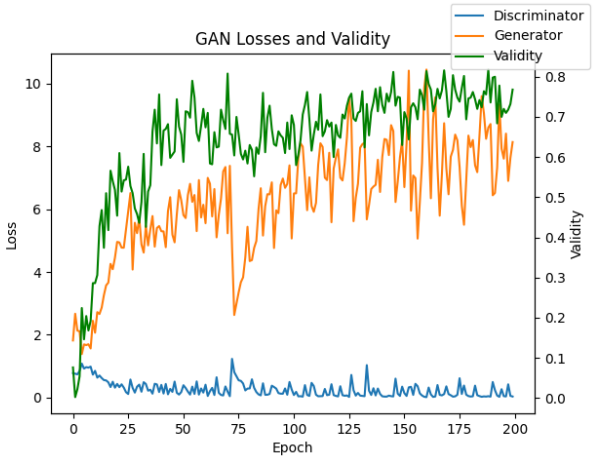

Figure 5.23: Losses and Validity plot of the second training done with the best GAN configuration for the Berkeley dataset.

In the case of the model trained on the KU Leuven dataset, the best configuration was the one described in Table 5.28. This model was also trained three times and the mean validity was 61%. Figure 5.25 shows the progress of the losses and the validity during the second training, and Figure 5.26 shows some designs generated by the model.

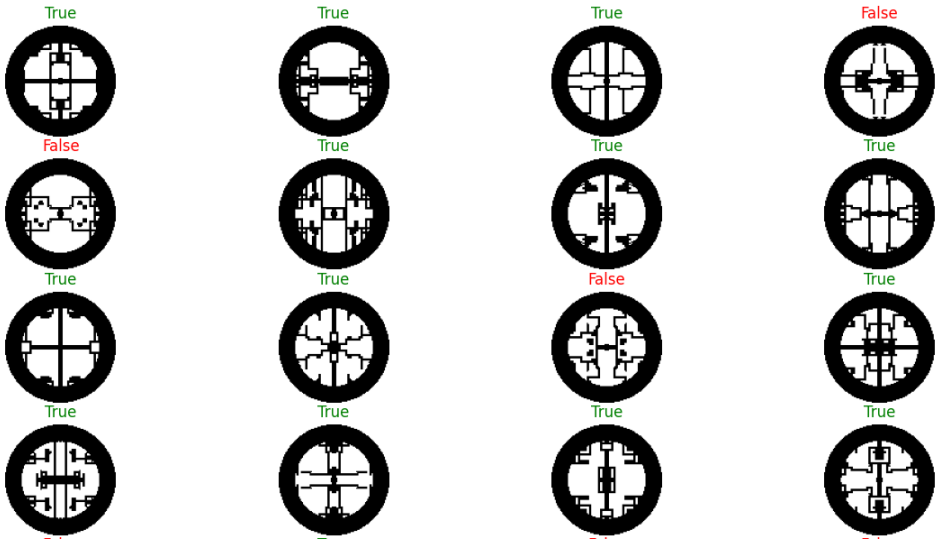

Figure 5.24: Designs generated by the best GAN model for the Berkeley dataset (second training).

Number of Epochs		200
Batch Size		256
Learning Rate	G	*CosineAnnealingWarmRestarts(opt_g, 25, 1, 0.0050)*
	D	*CosineAnnealingWarmRestarts(opt_d, 25, 1, 0.00025)*
Beta 1		0.5
Beta 2		0.9999
Kernel Size of the Conv2d Layers	G	3
	D	3

Table 5.28: Best GAN configuration for the KU Leuven Dataset.

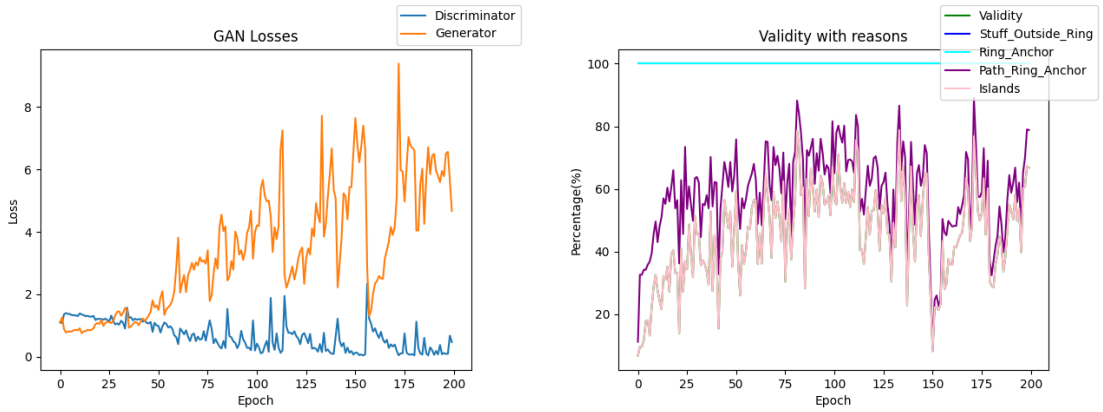

Figure 5.25: Losses and Validity plots of the second training done with the best GAN configuration for the KU Leuven dataset.

In the following sections, we will present the results of some experiments conducted with the GAN models, comparing different configurations and parameters

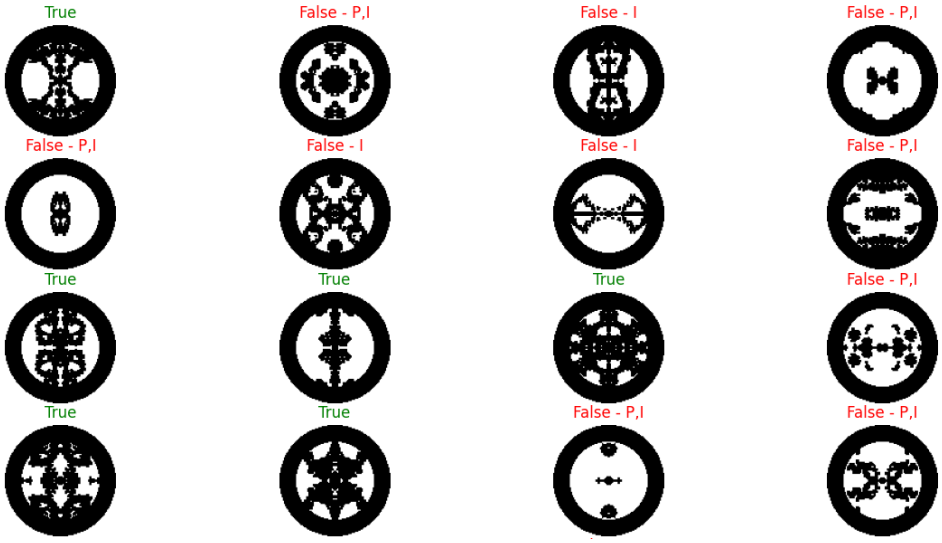

Figure 5.26: Designs generated by the best GAN model for the KU Leuven dataset (second training).

we mentioned in Section 4.1 as essential to achieving good results. It's important to note that if a parameter is not mentioned in the tables in the next sections, it is because it's the same as the best model, depending on the dataset.

5.4.9.1 Batch Size Tuning for GAN Models

As mentioned in Section 4.1, we conducted experiments with different batch sizes for the GAN models and discovered that the optimal value varied depending on the dataset. Table 5.29 presents the results on the Berkeley dataset using a batch size of 128 fake designs and 256 real designs for the discriminator, and 256 for the generator, as well as with a batch size of 256 for all components. All other parameters remained constant for both models.

Our experiments showed that the model with a batch size of 256 for all components achieved a mean validity of 45%, while the model with half the fake designs achieved a mean validity of 76%. This indicates that the model with fewer fake designs outperformed in terms of validity, generating a higher number of valid designs, which aligns with our primary goal in the GAN models. This implies that, for the Berkeley dataset, the most effective approach is to provide fewer fake examples to the discriminator.

However, for the KU Leuven dataset, we can't say the same. Table 5.30 shows the results for the same batch size experiments conducted on the Berkeley dataset, but this time on the KU Leuven dataset. As we can see, the model with a batch size of 256 for all components achieved a mean validity of 61%, while the model with half the fake designs achieved a mean validity of 58%. This indicates that, for

Batch Size	Learning Rate	Kernel Size (G)	Validity		
			Training Round	V (%)	MV (%)
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	5	1st	76	76
			2nd	78	
			3rd	75	
256	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	5	1st	50	45
			2nd	66	
			3rd	18	

Table 5.29: Comparison of the results obtained on the Berkeley dataset when providing fewer fake designs to the discriminator and when using the same batch size for all components.

the KU Leuven dataset, the most effective approach is to provide the same batch size for all components.

Batch Size	Learning Rate	Kernel Size (G)	Validity				
			Training Round	V (%)	P (%)	I (%)	MV (%)
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	3	1st	49	56.8	48.8	58
			2nd	59	65.6	58.8	
			3rd	66	72.0	66.0	
256	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	3	1st	65	74.6	64.6	61
			2nd	66	75.2	65.6	
			3rd	63	60.8	53.2	

Table 5.30: Comparison of the results obtained on the KU Leuven dataset when providing fewer fake designs to the discriminator and when using the same batch size for all components.

5.4.9.2 Learning Rate Tuning for GAN Models

In our experiments, we observed that using a cosine learning rate scheduler and setting a higher learning rate for the generator compared to the discriminator had a positive impact, as discussed in Section 4.1.

Table 5.31 presents the results from experiments on the KU Leuven dataset when using the cosine learning rate scheduler and when not using it. All other parameters remained constant for both models. The model with the cosine learning rate scheduler achieved a mean validity of 58%, while the model without it achieved a mean validity of 51%. These results indicate that using this type of scheduler was advantageous in our scenario.

Batch Size	Learning Rate	Kernel Size (G)	Validity				
			Training Round	V (%)	P (%)	I (%)	MV (%)
256 (G and D reals) 128 (D fakes)	0.0050 (G) 0.00025 (D)	3	1st	51	63.2	50.8	51
			2nd	57	66.0	56.6	
			3rd	46	60.0	46.4	
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	3	1st	49	56.8	48.8	58
			2nd	59	65.6	58.8	
			3rd	66	72.0	66.0	

Table 5.31: Comparison of the results achieved on the KU Leuven dataset with and without using the cosine learning rate scheduler.

In Table 5.32, we present the results from experiments on the KU Leuven dataset when using the same learning rate for both the generator and the discriminator and when using a higher learning rate for the generator compared to the discriminator. As usual, all other parameters remained constant for both models. The model with a higher learning rate for the generator achieved a mean validity of 58%, while the model with the same learning rate for both components achieved a mean validity of 51%. This demonstrates that setting a higher learning rate for the generator was more effective in our case.

Batch Size	Learning Rate	Kernel Size (G)	Validity				
			Training Round	V (%)	P (%)	I (%)	MV (%)
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, **0.00025**) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, **0.00025**) (D)	3	1st	45	58.6	45.4	51
			2nd	57	65.8	57.2	
			3rd	51	72.8	51.2	
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, **0.0050**) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, **0.00025**) (D)	3	1st	49	56.8	48.8	58
			2nd	59	65.6	58.8	
			3rd	66	72.0	66.0	

Table 5.32: Comparison of the results obtained on the KU Leuven dataset when using the same learning rate for both GAN components or using a higher learning rate in the generator than in the discriminator.

5.4.9.3 Tuning of the Kernel Size of Generator's Conv2d Layers for GAN Models

During our GAN experiments, we observed that the kernel size used in the generator's Conv2d layers had a significant impact on the results, and the optimal value depended on the dataset.

In the case of the Berkeley dataset, as we can see in Table 5.33, the model with a kernel size of 5 achieved a validity of 80%, while the models with kernel sizes of

3 and 7 achieved validities of 73% and 67%, respectively. This indicates that the model with a kernel size of 5 was the most effective for the Berkeley dataset.

Batch Size	Learning Rate	Kernel Size (G)	Validity (%)
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	**3**	73
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	**5**	80
256 (G and D reals) 128 (D fakes)	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	**7**	67

Table 5.33: Comparison of results obtained on the Berkeley dataset when using different kernel sizes in the generator's Conv2d layers.

For the KU Leuven dataset, the results were different, as shown in Table 5.34. The model with a kernel size of 3 achieved a mean validity of 61%, while the models with kernel sizes of 5 and 7 achieved mean validities of 47% and 40%, respectively. This demonstrates that the model with a kernel size of 3 was the most beneficial for the KU Leuven dataset.

Batch Size	Learning Rate	Kernel Size (G)	Mean Validity (%)
256	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	**3**	61
256	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	**5**	47
256	CosineAnnealingWarmRestarts (opt_g, 25, 1, 0.0050) (G) CosineAnnealingWarmRestarts (opt_d, 25, 1, 0.00025) (D)	**7**	40

Table 5.34: Comparison of results obtained on the KU Leuven dataset when using different kernel sizes in the generator's Conv2d layers.

5.4.10 WGAN Experiments

Contrary to the GAN models, we found that the best WGAN configuration was the same for both datasets as the results were similar.

The best WGAN configuration is displayed in Table 5.35. The model was trained three times per dataset and the mean validity was 69% for the Berkeley dataset and 65% for the KU Leuven dataset. Figure 5.27 shows the progress of the losses and the validity during the first training on the Berkeley dataset, and Figure 5.28 shows some designs generated by that model. Figure 5.29 shows the progress of the losses and the validity during the first training on the KU Leuven dataset, and Figure 5.30 shows some designs generated by that model.

Number of Epochs		200
Batch Size		64
Learning Rate		0.0001
Beta 1		0
Beta 2		0.9
Kernel Size of the Conv2d Layers	G	5
	D	3
Weight Clipping VS Gradient Penalty		Gradient Penalty
Number of Critic Iterations		3
Lambda Value		10

Table 5.35: Best WGAN configuration for both datasets.

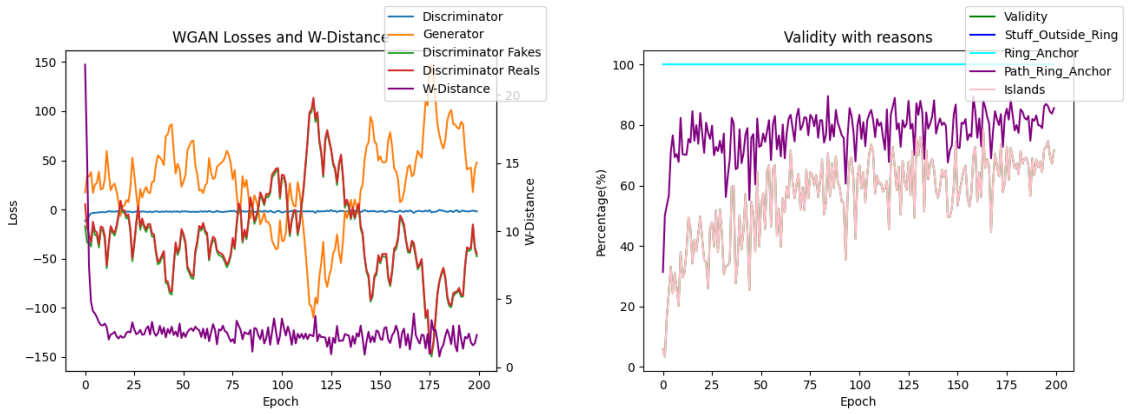

Figure 5.27: Losses and Validity plots of the first training done with the best WGAN configuration on the Berkeley dataset.

In the following sections we will present the results of some experiments conducted with the WGAN models, comparing different configurations of parameters we mentioned in Section 4.4 as essential to achieving good results. Once again, if

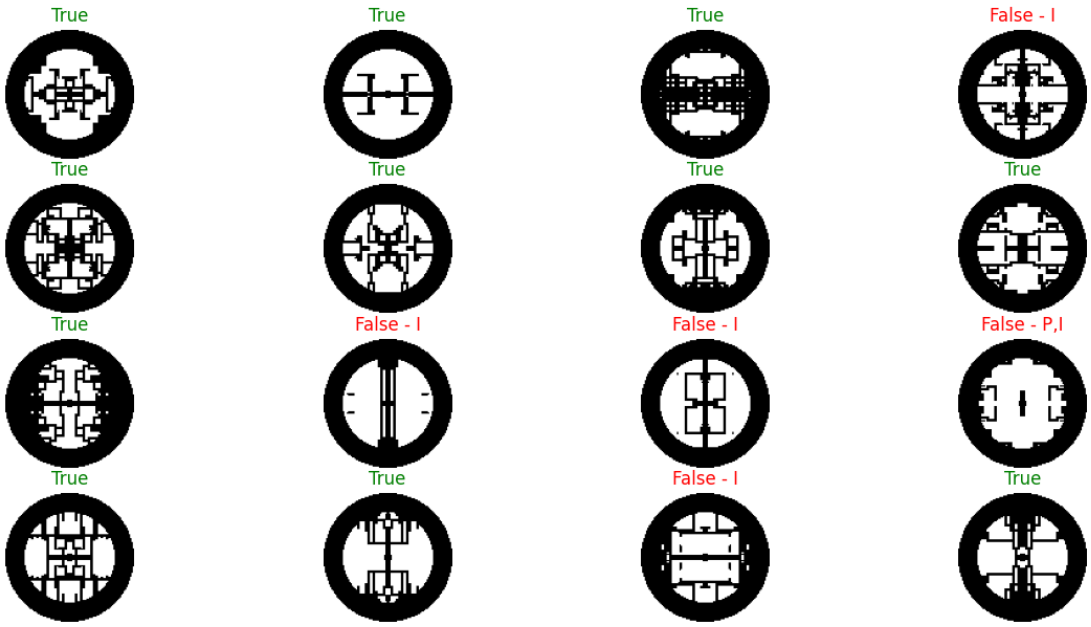

Figure 5.28: Designs generated by the best WGAN model on the Berkeley dataset (first training).

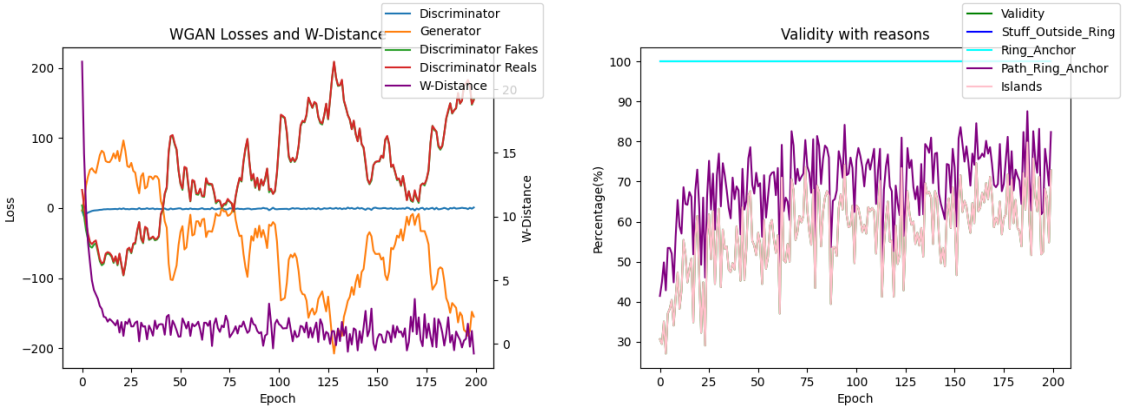

Figure 5.29: Losses and Validity plots of the first training done with the best WGAN configuration on the KU Leuven dataset.

a parameter is not mentioned in the tables in the next sections, it is because it is the same as the best model.

5.4.10.1 Weight Clipping VS Gradient Penalty

As mentioned in Section 4.4, we found that the method used to enforce the Lipschitz constraint had a significant impact on the results of our WGAN experiments.

Table 5.36 shows the results obtained on the Berkeley dataset when using the weight clipping and the gradient penalty techniques in the WGAN. It's important to note that we used the original parameters from the papers for both models:

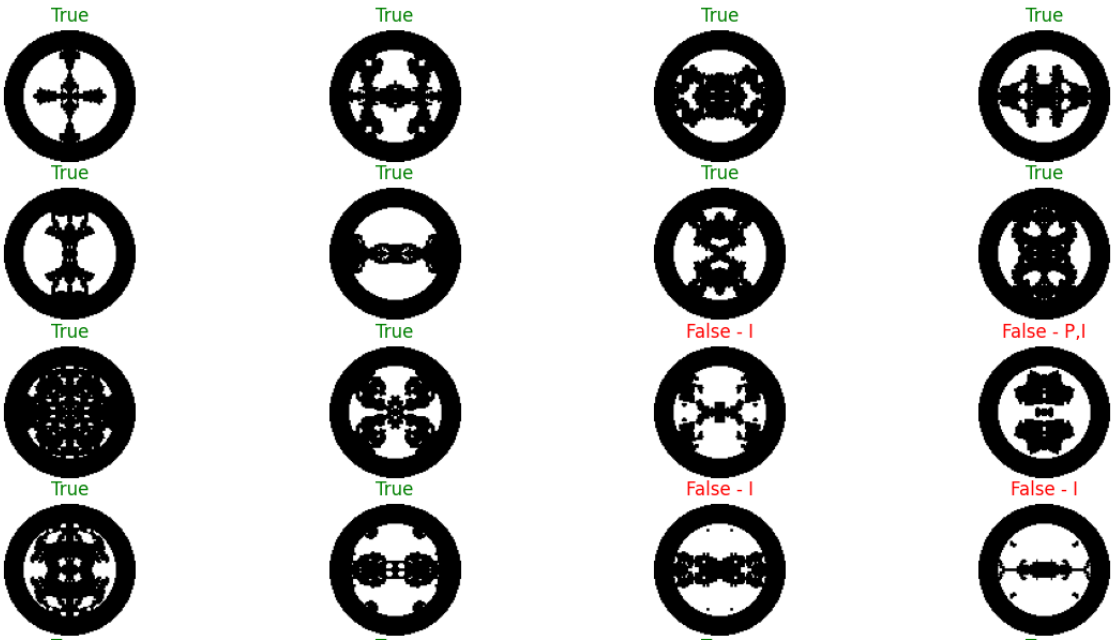

Figure 5.30: Designs generated by the best WGAN model on the KU Leuven dataset (first training).

- For the model with gradient penalty, we set beta 1 and beta 2 as 0 and 0.9, respectively, the lambda value as 10, and the number of critic iterations as 5.

- Regarding the model with weight clipping, we defined the clipping value as 0.01 and the number of critic iterations as 5.

As shown in the table, the model with gradient penalty achieved a mean validity of 65%, while the model with weight clipping achieved a mean validity of 22%. This proves that the gradient penalty technique produces better results.

Model	**Batch Size**	**Learning Rate**	**Validity (%)**		
			V	P	I
WGAN with **weight clipping**	64	0.00005	22	24.0	22.4
WGAN with **gradient penalty**	64	0.0001	65	81.4	65.0

Table 5.36: Comparison of the results obtained on the Berkeley dataset when using the weight clipping and the gradient penalty techniques in the WGAN.

5.4.10.2 Number of Critic Iterations

In our WGAN experiments, we also noticed that the number of critic iterations significantly impacted the results. Table 5.37 presents the results obtained on the Berkeley dataset when using various numbers of critic iterations in the Wasserstein GAN with Gradient Penalty (WGAN-GP). All other parameters remained constant for all models. The table indicates that the model with 3 critic iterations achieved a 73% mean validity, while the models with 5 and 2 critic iterations achieved mean validities of 58% and 65% respectively. This demonstrates that the model with 3 critic iterations was the most effective in our scenario.

Batch Size	Learning Rate	Critic Iterations	Validity (%)		
			V	P	I
64	0.0001	**5**	58	74.2	57.8
64	0.0001	**3**	73	85.0	73.2
64	0.0001	**2**	65	78.4	65.4

Table 5.37: Comparison of the results obtained with the Berkeley dataset when using various numbers of critic iterations in the WGAN-GP.

5.4.11 Results Conclusions

After analysing all the results displayed in the previous sections, we can take some conclusions:

- The CWGAN with Predictor is the best model for both datasets, as it achieved the highest mean validity and lowest mean conditional relative error (Sections 5.4.4.5 and 5.4.7.5).
- The CWGAN models are more stable than the CGAN models. The CGAN models resulted in mode collapse in almost every experiment with both datasets, while the CWGAN models did not experience mode collapse in any experiment with both datasets.
- The designs generated by the CWGAN models tend to be less diverse than those generated by the CGAN models (Figures 5.16 and 5.20 VS Figures 5.18 and 5.22).
- The configuration that worked best for the CGAN models trained on the Berkeley dataset was: CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint and Tanh activation function in the FCD (Section 5.4.3.4).

- The configuration that worked best for the CGAN models trained on the KU Leuven dataset was: CGAN with Predictor frozen (all generated designs), frozen pre-trained GAN checkpoint and Tanh activation function and reduced architecture of the FCD (Section 5.4.6.5).
- The configuration that worked best for the CWGAN models was the same for both datasets: CWGAN with Predictor frozen (all generated designs), frozen pre-trained WGAN checkpoint and reduced architecture of the FCD (Sections 5.4.4.5 and 5.4.7.5).
- Ultimately, the best model for our problem is the CWGAN KU Leuven V9 model (Section 5.4.2).

6

Conclusions

In this thesis, we explored the usage of conditional deep generative models for the design of MEMS devices. The traditional design process for the design of MEMS devices is very time-consuming. It requires a lot of human expertise, which can be a bottleneck when developing new devices. Therefore, using generative models can be a game changer in this field, as it can save time in the creation process and allow the exploration of different design paths. Our objective was to develop conditional generative models that, given specific frequencies, could generate designs that not only adhere to the MEMS design constraints but also have those frequencies. To support our work, we created the KU Leuven dataset, a specifically tailored dataset of MEMS devices with their respective frequencies to train our models.

After an extensive analysis of current MEMS design literature and existing generative models, we decided to explore GANs, as they were already used in the MEMS design field. We experimented with two GAN variants: Conditional GANs (CGANs) and Conditional Wasserstein GANs (CWGANs).

To assess the performance and reliability of these models, we developed a novel evaluation methodology consisting of several metrics. The most important ones were the mean number of valid designs generated and the mean error between the input and the predicted frequencies of the generated designs. We conducted multiple experiments with different parameters and techniques, and then assessed them using our new evaluation methodology to identify the best configuration for each model type and, consequently, the best model for our task.

Ultimately, we discovered that CWGANs outperformed CGANs, as they were much more stable and produced a greater number of valid and better-conditioned designs. However, the designs generated by CWGANs were found to be less diverse compared to those generated by CGANs, which was a limitation of this model type. Despite this limitation, we were able to generate designs that adhered

to the design constraints and had the desired frequencies, which was our primary goal.

In conclusion, we can say that our initial objective was achieved. Our best model (CWGAN KU Leuven V9) generated an average of 90% valid designs, with a 10% average error in the frequencies of the generated designs compared to the input. Although it does not produce highly diverse designs, it only takes around 11 seconds to generate a design based on a frequency, which is a very significant improvement compared to the traditional design process. This work will contribute to the MEMS design field, as it can significantly reduce the time required to design MEMS devices with good performance and ultimately produce different design paths that humans might not think of.

6.1 Future Work

In this project, we also tried training Diffusion Models, currently considered state-of-the-art in generative modelling. Unfortunately, the results we achieved were not satisfactory, as described in more detail in Annex I. We were unable to generate valid designs using this model, and we lacked the time to investigate the underlying reasons for this challenge. Consequently, a potential direction for future work would be to explore these models further to understand better and address the issues we encountered, as this type of model has the potential to outperform GANs in this task.

Bibliography

[1] R. Abdolvand et al. "Micromachined Resonators: A Review". In: *Micromachines* 7.9 (2016). ISSN: 2072-666X. DOI: 10.3390/mi7090160. URL: https://www.mdpi.com/2072-666X/7/9/160 (cit. on pp. 5, 6).

[2] S. K. Adaloglou Nikolas. *How diffusion models work: the math from scratch*. en. 2022-09. URL: https://theaisummer.com/diffusion-models/ (visited on 2024-01-15) (cit. on pp. 20, 21).

[3] M. Arjovsky, S. Chintala, and L. Bottou. "Wasserstein Generative Adversarial Networks". en. In: *Proceedings of the 34th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, 2017-07, pp. 214–223. URL: https://proceedings.mlr.press/v70/arjovsky17a.html (visited on 2024-01-17) (cit. on pp. 13–15, 37).

[4] *Build and Deploy Custom, AI-Powered Business Apps | Glide*. en. URL: https://www.glideapps.com/ai (visited on 2024-01-15) (cit. on p. 19).

[5] A. Casanova et al. "Instance-Conditioned GAN". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 27517–27529. URL: https://proceedings.neurips.cc/paper/2021/hash/e7ac288b0f2d41445904d071ba37aaff-Abstract.html (visited on 2024-01-02) (cit. on pp. 11, 13).

[6] F. Chollet and H. Liu. *A (not so) short introduction to MEMS*. 2018-08. ISBN: 978-2-9542015-0-4 (cit. on p. 5).

[7] *Common Problems | Machine Learning*. en. URL: https://developers.google.com/machine-learning/gan/problems (visited on 2024-01-09) (cit. on p. 11).

[8] *COMSOL: Multiphysics Software for Optimizing Designs*. en. URL: https://www.comsol.com/ (visited on 2024-01-21) (cit. on p. 26).

[9] R. Corvi et al. "Intriguing properties of synthetic images: from generative adversarial networks to diffusion models". In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. ISSN: 2160-7516. 2023-06, pp. 973–982. DOI: 10.1109/CVPRW59228.2023.00104. URL: https://ieeexplore.ieee.org/document/10209003 (visited on 2024-01-18) (cit. on p. 24).

[10] *DALL·E 3*. en-US. URL: https://openai.com/dall-e-3 (visited on 2024-01-15) (cit. on p. 19).

[11] P. Dhariwal and A. Nichol. "Diffusion Models Beat GANs on Image Synthesis". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 8780–8794. URL: https://proceedings.neurips.cc/paper_files/paper/2021/hash/49ad23d1ec9fa4bd8d77d02681df5cfa-Abstract.html (visited on 2024-01-15) (cit. on p. 19).

[12] *Dive into Deep Learning — Dive into Deep Learning 1.0.3 documentation*. URL: https://d2l.ai/ (visited on 2024-01-13) (cit. on pp. 9, 10, 22, 23).

[13] R. Dong et al. *Generative Design of inorganic compounds using deep diffusion language models*. arXiv:2310.00475 [cond-mat]. 2023-09. DOI: 10.48550/arXiv.2310.00475. URL: http://arxiv.org/abs/2310.00475 (visited on 2024-01-18) (cit. on p. 23).

[14] C. Düreth et al. "Conditional diffusion-based microstructure reconstruction". In: *Materials Today Communications* 35 (2023-06), p. 105608. ISSN: 2352-4928. DOI: 10.1016/j.mtcomm.2023.105608. URL: https://www.sciencedirect.com/science/article/pii/S2352492823002982 (visited on 2023-12-26) (cit. on p. 23).

[15] C. Fabbri. "Conditional Wasserstein Generative Adversarial Networks". In: () (cit. on p. 16).

[16] *GAN Training | Machine Learning*. en. URL: https://developers.google.com/machine-learning/gan/training (visited on 2024-01-09) (cit. on p. 8).

[17] *Generative Adversarial Network (GAN)*. en-US. URL: https://semiengineering.com/knowledge_centers/artificial-intelligence/neural-networks/generative-adversarial-network-gan/ (visited on 2024-01-05) (cit. on p. 8).

[18] I. Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper_files/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html (visited on 2023-12-21) (cit. on pp. 3, 7–10).

[19] I. Gulrajani et al. "Improved Training of Wasserstein GANs". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. (Visited on 2024-08-25) (cit. on pp. 15, 37).

[20] R. Guo et al. "Deep learning for non-parameterized MEMS structural design". en. In: *Microsystems & Nanoengineering* 8.1 (2022-08). Number: 1 Publisher: Nature Publishing Group, pp. 1–10. ISSN: 2055-7434. DOI: 10.1038/s41378-022-00432-9. URL: https://www.nature.com/articles/s41378-022-00432-9 (visited on 2023-12-26) (cit. on pp. 6, 7, 16, 17, 26, 27, 45).

[21] T. Guo, D. Herber, and J. T. Allison. "Circuit Synthesis Using Generative Adversarial Networks (GANs)". en. In: *AIAA Scitech 2019 Forum*. San Diego, California: American Institute of Aeronautics and Astronautics, 2019-01. ISBN: 978-1-62410-578-4. DOI: 10.2514/6.2019-2350. URL: https://arc.aiaa.org/doi/10.2514/6.2019-2350 (visited on 2024-01-18) (cit. on p. 16).

[22] K. He et al. "Deep Residual Learning for Image Recognition". en. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, 2016-06, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90. URL: http://ieeexplore.ieee.org/document/7780459/ (visited on 2024-02-08) (cit. on p. 17).

[23] S. M. Heinrich and I. Dufour. "Fundamental Theory of Resonant MEMS Devices". en. In: *Resonant MEMS*. John Wiley & Sons, Ltd, 2015, pp. 1–28. ISBN: 978-3-527-67633-0. DOI: 10.1002/9783527676330.ch1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9783527676330.ch1 (visited on 2024-01-07) (cit. on p. 6).

[24] E. Herron et al. "Generative Design of Material Microstructures for Organic Solar Cells using Diffusion Models". en. In: 2022-11. URL: https://openreview.net/forum?id=f9Lk1G9q-G- (visited on 2023-12-26) (cit. on p. 23).

[25] J. Ho, A. Jain, and P. Abbeel. "Denoising Diffusion Probabilistic Models". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 6840–6851. URL: https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html (visited on 2023-12-21) (cit. on pp. 19–22).

[26] *Imagen 2 - our most advanced text-to-image technology*. en. URL: https://deepmind.google/technologies/imagen-2/ (visited on 2024-01-15) (cit. on p. 19).

[27] T. Karras, S. Laine, and T. Aila. "A Style-Based Generator Architecture for Generative Adversarial Networks". en. In: () (cit. on p. 12).

[28] T. Karras et al. "Analyzing and Improving the Image Quality of StyleGAN". en. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, 2020-06, pp. 8107–8116. ISBN: 978-1-72817-168-5. DOI: 10.1109/CVPR42600.2020.00813. URL: https://ieeexplore.ieee.org/document/9156570/ (visited on 2024-01-18) (cit. on pp. 12, 13).

[29] T. Karras et al. "PROGRESSIVE GROWING OF GANS FOR IMPROVED QUALITY, STABILITY, AND VARIATION". en. In: (2018) (cit. on p. 12).

[30] K. Kochetov and A. Filchenkov. "Generative Adversarial Networks for Respiratory Sound Augmentation". In: *Proceedings of the 2020 1st International Conference on Control, Robotics and Intelligent System*. CCRIS '20. New York, NY, USA: Association for Computing Machinery, 2021-01, pp. 106–111. ISBN: 978-1-4503-8805-4. DOI: 10.1145/3437802.3437821. URL: https://dl.acm.org/doi/10.1145/3437802.3437821 (visited on 2024-01-07) (cit. on p. 7).

[31] S. Liu et al. "Diverse Image Generation via Self-Conditioned GANs". en. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, 2020-06, pp. 14274–14283. ISBN: 978-1-72817-168-5. DOI: 10.1109/CVPR42600.2020.01429. URL: https://ieeexplore.ieee.org/document/9157790/ (visited on 2024-01-02) (cit. on p. 13).

[32] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[33] M. Mirza and S. Osindero. *Conditional Generative Adversarial Nets*. arXiv:1411.1784 [cs, stat]. 2014-11. DOI: 10.48550/arXiv.1411.1784. URL: http://arxiv.org/abs/1411.1784 (visited on 2024-02-08) (cit. on pp. 10, 11).

[34] A. Q. Nichol and P. Dhariwal. "Improved Denoising Diffusion Probabilistic Models". en. In: *Proceedings of the 38th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, 2021-07, pp. 8162–8171. URL: https://proceedings.mlr.press/v139/nichol21a.html (visited on 2024-01-15) (cit. on pp. 20, 21).

[35] A. Odena, C. Olah, and J. Shlens. "Conditional Image Synthesis with Auxiliary Classifier GANs". en. In: () (cit. on p. 11).

[36] A. Radford, L. Metz, and S. Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. arXiv:1511.06434 [cs]. 2016-01. DOI: 10.48550/arXiv.1511.06434. URL: http://arxiv.org/abs/1511.06434 (visited on 2024-01-15) (cit. on p. 9).

[37] O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". en. In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by N. Navab et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 234–241. ISBN: 978-3-319-24574-4. DOI: 10.1007/978-3-319-24574-4_28 (cit. on pp. 21, 22).

[38] G. H. de Rosa and J. P. Papa. "A survey on text generation using generative adversarial networks". In: *Pattern Recognition* 119 (2021-11), p. 108098. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2021.108098. URL: https://www.sciencedirect.com/science/article/pii/S0031320321002855 (visited on 2024-01-07) (cit. on p. 7).

[39] M. Shahbazi et al. "Collapse by Conditioning: Training Class-conditional GANs with Limited Data". en. In: 2021-10. URL: https://openreview.net/forum?id=7TZeCsNOUB_ (visited on 2024-01-02) (cit. on pp. 11, 13).

[40] J. Sohl-Dickstein et al. "Deep Unsupervised Learning using Nonequilibrium Thermodynamics". en. In: *Proceedings of the 32nd International Conference on Machine Learning*. ISSN: 1938-7228. PMLR, 2015-06, pp. 2256–2265. URL: https://proceedings.mlr.press/v37/sohl-dickstein15.html (visited on 2024-01-15) (cit. on p. 19).

[41] F. Sui et al. "Customizing Mems Designs via Conditional Generative Adversarial Networks". en. In: *2022 IEEE 35th International Conference on Micro Electro Mechanical Systems Conference (MEMS)*. Tokyo, Japan: IEEE, 2022-01, pp. 450–453. ISBN: 978-1-66540-911-7. DOI: 10.1109/MEMS51670.2022.9699476. URL: https://ieeexplore.ieee.org/document/9699476/ (visited on 2023-12-20) (cit. on pp. 3, 16–18, 25, 26, 30, 32, 34).

[42] C. Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2818–2826. DOI: 10.1109/CVPR.2016.308 (cit. on p. 39).

[43] *UNet2DConditionModel*. https://huggingface.co/docs/diffusers/api/models/unet2d-cond. (Visited on 2024-09-17) (cit. on p. 105).

[44] *UNet2DModel*. https://huggingface.co/docs/diffusers/api/models/unet2d. (Visited on 2024-09-17) (cit. on p. 105).

[45] A. Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (visited on 2024-01-16) (cit. on p. 22).

[46] Y. Wang et al. "ImaGINator: Conditional Spatio-Temporal GAN for Video Generation". en. In: *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Snowmass Village, CO, USA: IEEE, 2020-03, pp. 1149–1158. ISBN: 978-1-72816-553-0. DOI: 10.1109/WACV45572.2020.9093492. URL: https://ieeexplore.ieee.org/document/9093492/ (visited on 2024-01-07) (cit. on p. 7).

[47] M. Yang et al. "Scalable Diffusion for Materials Generation". In: arXiv:2311.09235 [cs]. Neural Information Processing Systems, 2023-10. URL: http://arxiv.org/abs/2311.09235 (visited on 2023-12-20) (cit. on p. 23).

[48] M. Zheng et al. "Conditional Wasserstein generative adversarial network-gradient penalty-based approach to alleviating imbalanced data classification". In: *Information Sciences* 512 (2020), pp. 1009–1023. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2019.10.014. URL: https://www.sciencedirect.com/science/article/pii/S0020025519309715 (cit. on p. 16).

A

Best Model Architecture

This is the architecture of the CWGAN KU Leuven V9 model, which we found to be the best model for our problem. It's important to note that the Predictor's architecture is not shown in this figure to simplify since it is based on the ResNet50 network.

```
CWGAN(
    (wgan): WGAN_GP(
        (generator): Generator(
        (model): Sequential(
            (0): Linear(in_features=64, out_features
               =640000, bias=True)
            (1): Unflatten(dim=1, unflattened_size=(256,
               50, 50))
            (2): BatchNorm2d(256, eps=1e-05, momentum=0.1,
               affine=True, track_running_stats=True)
            (3): ConvTranspose2d(256, 256, kernel_size=(5,
               5), stride=(1, 1), padding=(2, 2))
            (4): BatchNorm2d(256, eps=0.8, momentum=0.1,
               affine=True, track_running_stats=True)
            (5): LeakyReLU(negative_slope=0.2, inplace=True
               )
            (6): ConvTranspose2d(256, 128, kernel_size=(5,
               5), stride=(1, 1), padding=(2, 2))
            (7): BatchNorm2d(128, eps=0.8, momentum=0.1,
               affine=True, track_running_stats=True)
            (8): LeakyReLU(negative_slope=0.2, inplace=True
               )
```

```
        (9): ConvTranspose2d(128, 64, kernel_size=(5,
           5), stride=(1, 1), padding=(2, 2))
        (10): BatchNorm2d(64, eps=0.8, momentum=0.1,
           affine=True, track_running_stats=True)
        (11): LeakyReLU(negative_slope=0.2, inplace=
           True)
        (12): ConvTranspose2d(64, 32, kernel_size=(5,
           5), stride=(1, 1), padding=(2, 2))
        (13): BatchNorm2d(32, eps=0.8, momentum=0.1,
           affine=True, track_running_stats=True)
        (14): LeakyReLU(negative_slope=0.2, inplace=
           True)
        (15): ConvTranspose2d(32, 16, kernel_size=(5,
           5), stride=(1, 1), padding=(2, 2))
        (16): BatchNorm2d(16, eps=0.8, momentum=0.1,
           affine=True, track_running_stats=True)
        (17): LeakyReLU(negative_slope=0.2, inplace=
           True)
        (18): ConvTranspose2d(16, 3, kernel_size=(5, 5)
           , stride=(1, 1), padding=(2, 2))
        (19): Sigmoid()
    )
    )
    (discriminator): Discriminator(
    (model): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride
           =(1, 1), padding=(1, 1))
        (1): LeakyReLU(negative_slope=0.2, inplace=True
           )
        (2): Dropout2d(p=0.25, inplace=False)
        (3): Conv2d(64, 64, kernel_size=(3, 3), stride
           =(1, 1), padding=(1, 1))
        (4): LeakyReLU(negative_slope=0.2, inplace=True
           )
        (5): Dropout2d(p=0.25, inplace=False)
        (6): InstanceNorm2d(64, eps=1e-05, momentum
           =0.1, affine=True, track_running_stats=False
           )
```

```
            (7): Conv2d(64, 64, kernel_size=(3, 3), stride
               =(1, 1), padding=(1, 1))
            (8): LeakyReLU(negative_slope=0.2, inplace=True
               )
            (9): Dropout2d(p=0.25, inplace=False)
            (10): InstanceNorm2d(64, eps=1e-05, momentum
               =0.1, affine=True, track_running_stats=False
               )
            (11): Conv2d(64, 64, kernel_size=(3, 3), stride
               =(1, 1), padding=(1, 1))
            (12): LeakyReLU(negative_slope=0.2, inplace=
               True)
            (13): Dropout2d(p=0.25, inplace=False)
            (14): InstanceNorm2d(64, eps=1e-05, momentum
               =0.1, affine=True, track_running_stats=False
               )
            (15): Conv2d(64, 64, kernel_size=(3, 3), stride
               =(1, 1), padding=(1, 1))
            (16): LeakyReLU(negative_slope=0.2, inplace=
               True)
            (17): Dropout2d(p=0.25, inplace=False)
            (18): InstanceNorm2d(64, eps=1e-05, momentum
               =0.1, affine=True, track_running_stats=False
               )
            (19): Conv2d(64, 128, kernel_size=(3, 3),
               stride=(1, 1), padding=(1, 1))
            (20): LeakyReLU(negative_slope=0.2, inplace=
               True)
            (21): Dropout2d(p=0.25, inplace=False)
            (22): InstanceNorm2d(128, eps=1e-05, momentum
               =0.1, affine=True, track_running_stats=False
               )
            (23): Flatten(start_dim=1, end_dim=-1)
            (24): Linear(in_features=320000, out_features
               =1, bias=True)
        )
        )
)
```

```
    (fcd): FullyConnectedDecoder(
        (model): Sequential(
        (0): Linear(in_features=1, out_features=128, bias=
           True)
        (1): LeakyReLU(negative_slope=0.2, inplace=True)
        (2): Linear(in_features=128, out_features=512, bias
           =True)
        (3): BatchNorm1d(512, eps=0.8, momentum=0.1, affine
           =True, track_running_stats=True)
        (4): LeakyReLU(negative_slope=0.2, inplace=True)
        (5): Linear(in_features=512, out_features=1024,
           bias=True)
        (6): BatchNorm1d(1024, eps=0.8, momentum=0.1,
           affine=True, track_running_stats=True)
        (7): LeakyReLU(negative_slope=0.2, inplace=True)
        (8): Linear(in_features=1024, out_features=1024,
           bias=True)
        (9): BatchNorm1d(1024, eps=0.8, momentum=0.1,
           affine=True, track_running_stats=True)
        (10): LeakyReLU(negative_slope=0.2, inplace=True)
        (11): Linear(in_features=1024, out_features=512,
           bias=True)
        (12): BatchNorm1d(512, eps=0.8, momentum=0.1,
           affine=True, track_running_stats=True)
        (13): LeakyReLU(negative_slope=0.2, inplace=True)
        (14): Linear(in_features=512, out_features=32, bias
           =True)
        )
    )
    (predictor): Predictor(
        (base_model): ResNet(...)
    )
)
```

I

Diffusion Models

During the development of this thesis, we also tried to use Diffusion Models to generate designs. Our objective was to see if this type of model could generate better designs than the ones generated by the CGANs and CWGANs.

We tried to use two different models from HuggingFace's Diffusers library, the UNet2DModel [44] and the UNet2DConditionModel [43]. We trained both models on the KU Leuven dataset, but the results were unsatisfactory. We did various experiments with different hyperparameters, but the results were always the same: all epochs generated all invalid designs, i.e., the validity metric was always zero. Figure I.1 shows the losses and validity plots of one of the experiments we did with the UNet2DConditionModel, and Figure I.2 shows the designs generated by the same model. As we can observe, the loss values were decreasing, but the validity metric was always zero, which means that all the designs generated were invalid. The designs generated were different from each other but very complex and not visually similar to the designs in the dataset or to the designs generated by the CGANs and CWGANs.

Unfortunately, we didn't manage to understand what the problem was with this type of model and why it was unable to generate valid designs. We didn't have enough time to investigate further.

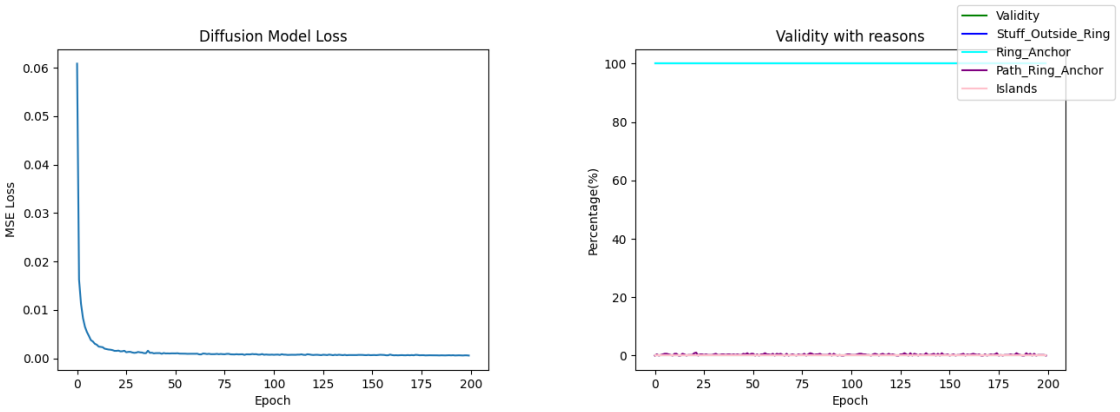

Figure I.1: Losses and Validity plots of a UNet2DConditionModel experiment on the KU Leuven dataset.

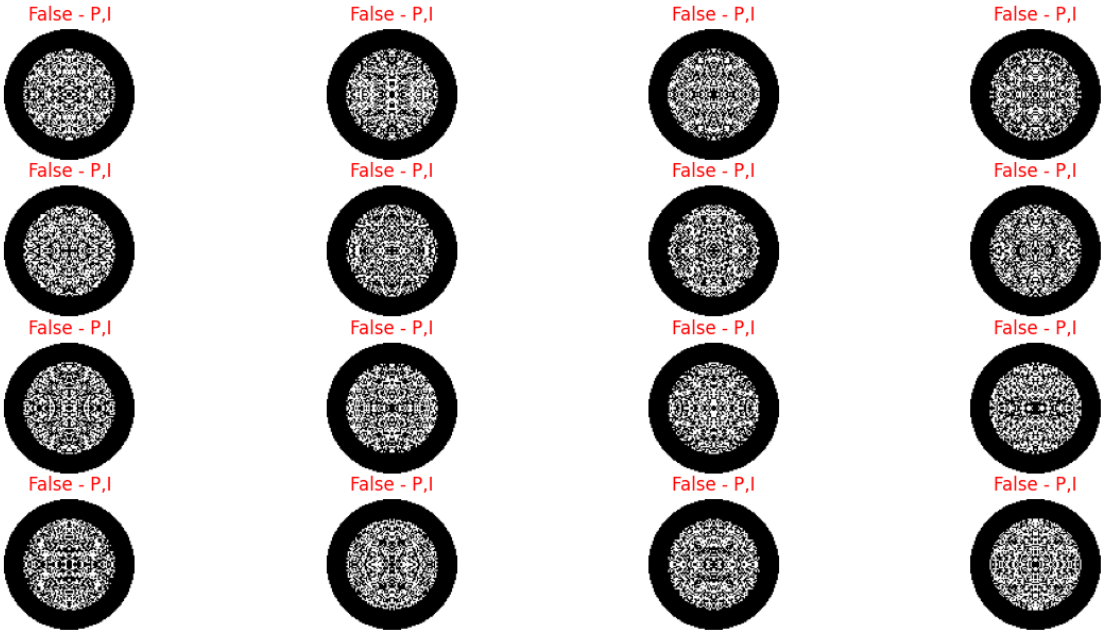

Figure I.2: Designs generated by a UNet2DConditionModel trained on the KU Leuven dataset.

II

Submitted Abstract

This Annex reveals our abstract submission to the IEEE MEMS 2025 conference.

8.05 Machine Learning (ML) & Artificial Intelligence (AI)-Enhanced MEMS/NEMS Design, Manufacturing, and Applications 0796

GENERATIVE AND GENERALIZABLE OPTIMIZATION FRAMEWORK FOR MEMS DEVICES THROUGH ACTIVE LEARNING

Novelty / Progress Claim(s)

This paper presents a novel and highly flexible active learning framework for the optimization of micro-electro-mechanical systems (MEMS) devices. At its core, it leverages a deep convolutional generative adversarial network (GAN), regularized by a MEMS performance predictor, to selectively and iteratively shift the model design geometry learned distribution towards optimizing a target criterion. Given its principled design, the framework trivially generalizes to distinct optimization criteria. To illustrate the efficacy of the framework, this study introduces a MEMS resonator featuring an unparameterized arbitrary suspension, where we explicitly optimize the quality factor (Q) while ensuring fabrication feasibility. We showcase a diverse array of fabrication-feasible MEMS resonators exhibiting improved Q-factors, achieving up to 10% Q relative improvement. To the best of our knowledge, this is the first successful implementation of a MEMS devices' design generative machine learning optimization process.

Background / State of the Art

Recently, machine learning algorithms have emerged as a novel methodology in the design of MEMS devices. For instance, Siu *et al.* conducted a study on the conditional generation of MEMS devices utilizing deep convolutional generative adversarial networks (GANs) [1]. Despite promising, it often produces designs that exhibit suboptimal Q values. That suggests its limitation in generating designs higher-performing designs than those in the training dataset, due to the lack of explicit optimization. Active learning optimization techniques [2,3], have been applied across various scientific domains and show considerable potential. This is further evidenced by a recent study on the design of freeform diffractive meta gratings based on generative adversarial networks [4]. Nevertheless, these optimization strategies have yet to be implemented in the design of MEMS devices, thereby presenting a significant opportunity to advance device performance to a new level.

Description of the New Method or System

Our proposed methodology is structured around a four-part pipeline that includes (Figure 1): (1) a generator pre-trained on a designated training dataset, (2) a Q-value evaluation process, (3) a buffer designed to temporarily store generated designs along with their evaluated properties, and (4) a buffer update strategy that iteratively incorporates new and improved designs based on a predefined criteria. The process initiates with the generation of a batch of sub-optimal designs utilizing the generator, which are subsequently assessed through the Q value evaluation process. Following this evaluation, the selected update strategy determines which designs will be integrated into the buffer, effectively replacing those with lower Q values. Ultimately, the GAN is trained in an interleaved manner, with designs from the original training dataset the optimization buffer. This biases the model's generation toward designs that optimize the target criteria. The principled active learning mechanism can be tuned to different target criteria of the operator, by simply defining a corresponding buffer update strategy.

Experimental Results

Our proposed methodology has been successful at improving the quality factor (Q) of generated designs by an average of 10% as observed in Figure 4. Our process involved optimizing the generative model for 50 epochs with 180 steps each and with 80% of designs in batches coming from the optimization buffer. Some flexural frequency ranges have achieved higher improvements, up to around 20%. Higher frequencies, on the other hand, noticed little improvement, sometimes even decreasing in Q value by a small margin. Additionally, the framework could optimize the designs with no penalty on feasibility or design diversity, as observed in Figure 3.

Word count: 536

References

[1] Sui, Fanping, et al. 2022 IEEE 35th MEMS, pp. 450-453.
[2] Rui Xin, et al. The Journal of Physical Chemistry C 2021 125 (29),
[3] Bailey, Michael, et al. bioRxiv (2023): 2023-07.
[4] Wen, Fufang, et al. Acs Photonics 7.8 (2020): 2098-2104.

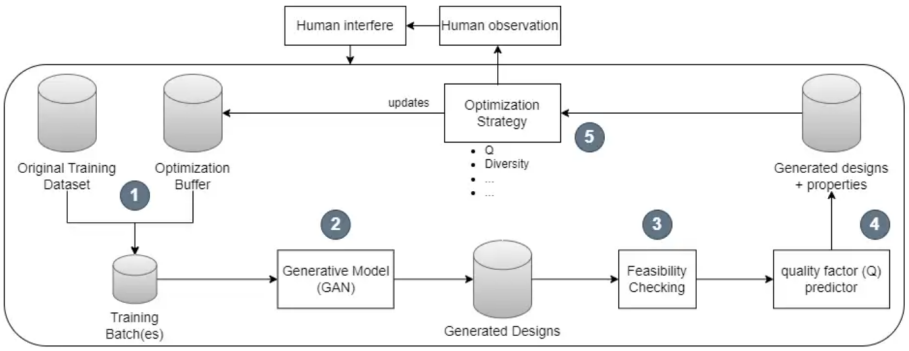

Figure 1 - Optimization framework pipeline architecture: (1) training mini-batches are constructed from both the original training dataset and the optimization buffer. These will be used to fine-tune a pre-trained generative adversarial network (2) to generate batches of new designs. These new designs will then pass through a feasibility checker (3), which validates the structure and filters out any non-feasible designs. Furthermore, the remaining designs' quality factor (Q) is computed using a predictor (4) and these are passed to the optimization strategy component (5), which contains the optimization buffer update logic. This process is repeated a configurable number of times.

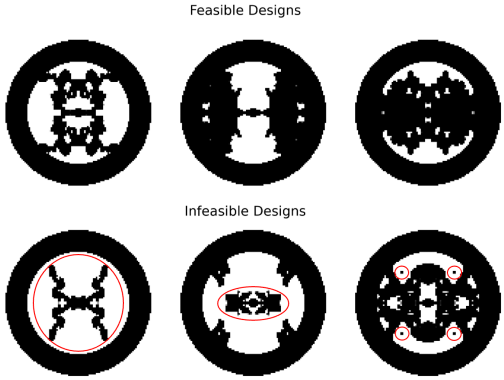

Figure 2 - Examples of feasible and infeasible designs, with reasons for infeasibility marked in red.

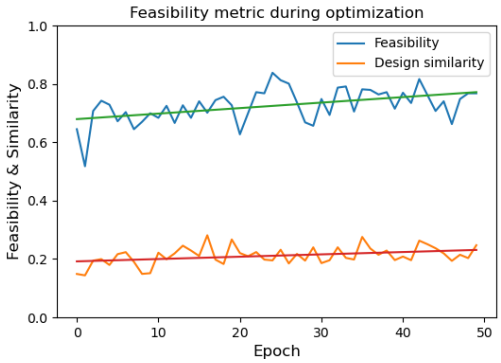

Figure 3 - Feasibility and similarity observed during the optimization epochs. Both the feasibility and the design similarity remain stable throughout the whole process meaning our optimization framework can optimize the generated designs with no feasibility or diversity penalty.

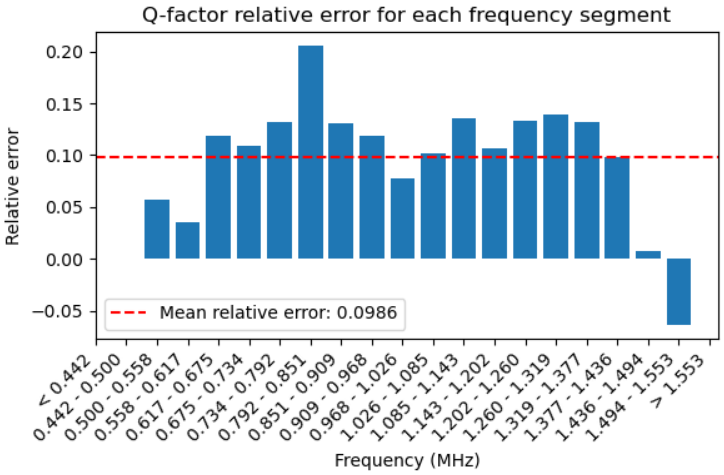

Figure 4 - Q-factor mean relative error when comparing with the non-optimized model. Most segments have improved Q factors suggesting our optimization framework improved the original model to the point of generating improved Q factor designs.

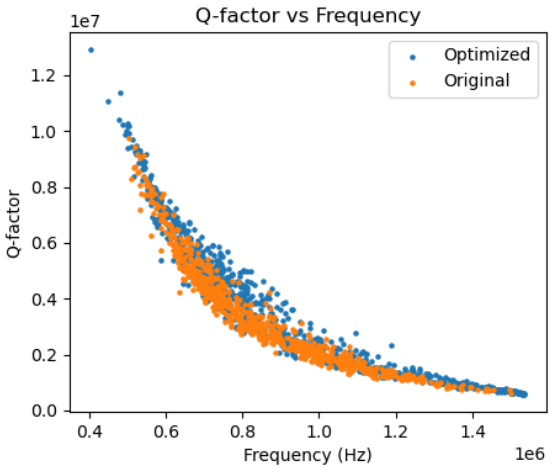

Figure 5 - Distribution of the frequencies and Q factors of designs generated by the original non-optimized and the optimized checkpoint.