

Vom Fachbereich Mathematik der Rheinland-Pfälzischen Technischen Universität Kaiserslautern-Landau zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Doctor rerum naturalium, Dr. rer. nat.) genehmigte Dissertation

Generative Adversarial Networks for Time Series Simulation

Laurena Ramadani

Reviewed by:
Prof. Dr. Ralf Korn
Assoz. Univprof. Dr. Sascha Desmettre

Date of defense: December 20, 2024

DE-386

In loving memory of my mom, whom I lost at the beginning of this journey.

Acknowledgements

First and foremost, I would like to express my deep and sincere gratitude to my supervisor, Prof. Dr. Ralf Korn, for his patience, motivation, and careful supervision. His insightful comments and ideas, as well as his invaluable feedback on my thesis and various talks, have guided me greatly throughout my research. His immense expertise has been a cornerstone of my academic growth, and his support has been crucial in shaping this thesis. Beyond his expertise, working with Prof. Dr. Korn has been a privilege, both professionally and personally. His unwavering dedication, keen interest, and genuine willingness to help his students are truly inspiring. He has played a vital role in helping me grow, not only as a researcher but also as a professional. I am deeply grateful for the opportunity to learn from and work alongside him.

My sincere thanks also go to Assoz. Univprof. Dr. Sascha Desmettre from the University of Linz for taking the time to review my thesis and providing valuable evaluation.

I would also like to thank my colleagues at RPTU and ITWM for their valuable advice and support. In particular, I am grateful to Ajla, Andrea, and Christine for their friendship and encouragement throughout this journey. A special thanks goes to Ajla, with whom I had the pleasure of attending many conferences, making the experience all the more enjoyable. Additionally, I am grateful to the researchers who offered insightful discussions and suggestions during various conferences.

I also wish to express my heartfelt appreciation to my family, whose unwavering support, encouragement, and love have been an endless source of inspiration throughout this journey and my entire life. Their strength and belief in me have helped me face every challenge, and for that, I am forever grateful.

Contents

1 Introduction **4**
1.1 Outline 6

2 What are Generative Adversarial Networks? **8**
2.1 Introdcution and basics of Artificial Neural Networks and Deep learning 9

3 Architecture of GANs **31**
3.1 Loss (cost or objective) function 33
3.2 Optimization algorithm 35
3.3 Challenges and evolution of GANs 40
3.3.1 Mode Copllapse 41
3.3.2 Non-convergence and instability 41
3.3.3 A Taxonomy 42

4 ForGAN Applications: Simulating Time Series and Forecasting Financial Risks **44**
4.1 ForGAN for financial time series 71
4.2 Forecasting Value at Risk using ForGAN 81
4.2.1 VaR Estimation Using Forecasting Simulation 96

5 Portfolio Optimization with GANs **110**
5.1 Introduction to Portfolio Theory 110
5.2 Time-series Generative Adversarial Networks 120
5.2.1 Embedding and Recovery Functions 121
5.2.2 Sequence Generator and Discriminator 122
5.2.3 Reconstruction and Loss Functions 122
5.3 Principal Component Analysis 123
5.4 TimeGAN and ForGAN-PCA for Portfolio Optimization 127
5.5 Portfolio Optimization Using GANs and PCA for Stock Price Data 144
5.5.1 Portfolio Optimization Using GANs and PCA for Synthetic Data: Examples and Analysis 159
5.5.2 Portfolio Optimization and Expected Growth Rate Maximization 173

6 Conclusion **194**

Chapter 1

Introduction

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow in 2014 [41], represent a significant breakthrough in the field of deep learning. GANs consist of two neural networks: a *generator*, which produces new data instances, and a *discriminator*, which evaluates these instances as either real or generated. These networks engage in an adversarial process—where the generator creates data, and the discriminator classifies those data as either real or fake. As they improve, the generator becomes better at producing realistic data while the discriminator becomes more effective at distinguishing between real and generated data. When both networks perform optimally, the generator produces data that closely resembles the real data. This competitive (or cooperative) dynamic enables GANs to generate highly realistic data across various domains. Over the past decade, GANs have found applications in fields like image generation [41], [46], video generation [52], speech synthesis [76], and recently, in financial modeling [82].

The primary objective of this thesis is to leverage GANs for generating synthetic financial data that mirrors the characteristics of real-world data. Modeling financial data is challenging due to its inherent complexity, volatility, and uncertainty. Despite advancements in deep learning, applying these techniques in finance is often restricted by insufficient data to properly train the models. Here, GANs present a promising solution: they can generate synthetic data that follows the same distribution as actual financial data, allowing researchers to expand datasets and perform robust financial modeling and analysis. For instance, QuantGANs [82] employ temporal convolutional networks (TCNs) to approximate volatility and capture long-range dependencies, making them effective in modeling realistic time-series data.

In this work, we focus on two types of GANs: *ForGAN* [74] and *TimeGAN* [83]. *ForGAN* is a variant of conditional GANs (cGAN), where the condition 'c' refers to historical data used to condition the model. The goal of ForGAN is to model the probability distribution of the next time step, x_{t+1}, based on previous observations. Specifically, we use historical data $x_0, x_1, ..., x_t$ to estimate the conditional probability distribution $p(x_{t+1}|c)$, where again 'c' represents the historical input data. In this thesis, we apply ForGAN to simulate two types of time-series data: 15-minute

electricity consumption in Germany's intraday market taken from Smard[1], and financial time series data, the *Standard and Poor's 500* [2], which is a stock market index that tracks the stocks of 500 large-cap U.S. companies.

Through these simulations, we explore several key aspects of the model:

- The effectiveness of data transformation for improving accuracy by using a scaler as a hyper-parameter.
- Comparing the performance of Non-Saturating GAN loss against standard GAN loss functions.
- The impact of training iterations on the model's performance and strategies to avoid mode collapse.
- The importance of hyperparameter tuning in optimizing GAN performance.
- The role of the time window used for training, generating, and conditioning data in enhancing the accuracy of the model's forecasts and ensuring temporal consistency.

There are different types of GANs, used in various fields to generate realistic data, so, using GANs for data augmentation is an obvious suggestion (even from people who have never used them). With ForGAN enabling us to generate additional data for a specific time in the future, we came up with another objective for this research, to evaluate the use of GAN-generated data for financial applications, particularly for Value at Risk (VaR) estimation and portfolio optimization. Primarily, we consider the Markowitz Mean-Variance Approach (1952) [2]. We aim to answer the critical question: *Can GANs be effectively used for portfolio optimization?* A key challenge in portfolio optimization is estimating the parameters of the securities, specifically the means and covariances of their returns. In this thesis, we generate log-increments of stock prices for Amazon, Apple, and Netflix using GANs. Have we been able to tackle the problem of how to capture the correlation between the log increments of the stock prices when generating additional data? Does performing the principal component analysis (PCA) on the joint log-returns to capture correlations would help us to reduce the multi-asset generation problem of dependent data to three independent one-dimensional generation problems?

Additionally, we explore the reliability of GAN-generated data using the PCA method compared to historical data by conducting tests with synthetic data. This helps address the question of whether the assumptions in financial models are better satisfied by historical or GAN-generated data. Which data can be trusted more when we use the standard maximum likelihood estimations for mean-variance portfolio optimization?

TimeGAN is another generative model we utilize, specifically designed for time-series data, combining supervised and adversarial learning to capture both temporal dependencies and correlations. Unlike traditional GANs, which focus on

[1] https://www.smard.de/home

[2] https://www.spglobal.com/spdji/en/indices/equity/sp-500/overview

generating independent data points, TimeGAN introduces a supervised loss that ensures the generated sequences follow the correct temporal order, preserving the autocorrelation between time steps. By using a combination of reconstruction loss, supervised learning, and adversarial loss, TimeGAN maintains both the short-term and long-term relationships in time-series data, making it ideal for applications like financial data modeling.

Its architecture makes TimeGAN suitable for our setting here. In the course of this thesis, we highlight the strengths and limitations of both ForGAN with the PCA method and TimeGAN in the context of financial time-series modeling and portfolio optimization. While GANs offer significant potential for generating realistic financial data, each application presents unique challenges, particularly in capturing correlations. We are always focusing on portfolio optimization and estimating the relevant parameters.

We further explore the question which data can be trusted more, the train historical data or the ones generated from GANs. In the later sections of this thesis, we evaluate the reliability of GAN-generated data with the PCA method when applying it to a synthetic problem where the explicit solution is known. We focus on Expected Growth Rate Maximization, we first simulate artificial data from this known model, then train a GAN with it, and finally use the trained GAN to simulate artificial data. We then compare the portfolio performance in the sense that we check which optimal strategy is closer to the optimal one, the portfolio estimated on the training data or the one estimated on the generated data. To further validate the results, we compute Confidence Intervals (CIs) for key parameters, such as drift, volatility, and correlation, providing insight into the statistical robustness of our generated data and its application to financial decision-making.

1.1 Outline

The structure of this thesis is outlined below. In Chapter 2, we provide an introduction to key concepts in artificial neural networks (ANNs), machine learning, and deep learning. We distinguish these fields, explain their foundational principles, and highlight their significance in building GAN models. ANNs, inspired by the brain's structure, serve as the foundation for both machine learning and deep learning models. Machine learning encompasses a broad range of techniques for enabling computers to learn from data, while deep learning focuses on complex neural networks with multiple layers capable of processing vast amounts of data.

In Chapter 3, the architecture of Generative Adversarial Networks (GANs) is explained, focusing on the roles of the generator and discriminator and their adversarial relationship. Key optimization techniques are discussed, along with common challenges such as mode collapse and instability during training. The chapter also provides a brief overview of different GAN variants, highlighting the evolution and improvements made to the architecture over time. This lays the groundwork for understanding how GANs function before exploring their applications in later chapters.

In Chapter 4, we introduced and applied ForGAN, for time-series simulation of two distinct datasets: 15-minute intraday electricity consumption in Germany and the S&P 500 Index. These simulations were utilized to perform Value at Risk (VaR) forecasting, a critical measure in financial risk management. We conducted an in-depth analysis of the model's hyperparameters, including the effect of data scaling, different loss functions, and training iterations. Furthermore, we explored how various configurations impacted the stability and accuracy of ForGAN in capturing complex patterns and dependencies within the time-series data.

In Chapter 5, we focus on portfolio theory and its application through mean-variance analysis to optimize asset allocation. We then introduce TimeGAN and ForGAN, which, combined with Principal Component Analysis (PCA), are used to generate synthetic data for portfolio optimization. This chapter compares the performance of GAN-generated data versus historical data, raising the question of which data can be trusted more for portfolio optimization. To answer this, we conduct tests using synthetic data where the conditions are satisfied. To further explore this question, we introduce Expected Growth Rate Maximization, where we use synthetic data for which the exact solution is known. This allows us to test the reliability of the generated data in a controlled environment. Additionally, Confidence Intervals (CIs) are computed for key financial parameters, providing further insight into the robustness of the GAN-generated data for financial decision-making.

Finally, Chapter 6 contains the conclusion which summarizes our findings and points at possible future research.

Chapter 2

What are Generative Adversarial Networks?

Generative Adversarial Networks **GANs** are an exciting change in the architecture of deep neural networks. This architecture was proposed for the first time by Ian Goodfellow in 2014 [41], where instead of one, based on this technique, computers can use two separate neural networks pitted against each other to generate realistic data. This adversarial process consists of two primary components: the *generator*, which is trained to create new examples, and the *discriminator*, which attempts to classify these examples as either real (originating from the training data set) or fake (produced by the generator). Over time, the generator improves by attempting to fool the discriminator, while the discriminator becomes better at distinguishing real data from generated data.

GANs are an impressive and rapidly shifting topic of research today in the field of deep learning. They have achieved notable results in generating realistic examples in a variety of problem areas, such as producing real-looking images of human faces [72], translating one image to another, for example, photos of summer to winter [64], or day to night [61], 3D image generation [53], and many more applications related to image production. Along with images, other research areas have been analyzed, such as speech synthesis [69], and video generation [52].

Another field where the application of GANs has been explored is finance. Modeling in finance can be challenging due to the complex properties of the data and their unknown structure. Deep learning algorithms are improving in the field of data-driven models, but the insufficiency of existing data or the lack of access to sufficient data to train the models is impeding several new applications. Therefore the use of GANs for producing data that come from the same distribution as the data we have in the real world is beneficial, since then we can perform easy statics on the generated time series. For example, S. Hyland used Recurrent Conditional GANs in [60] to produce realistic *synthetic* medical data because of the limited access to medical reports due to privacy regulations. Moreover, QuantGANs [82] introduces utilizing temporal convolutional networks (TCNs) to approximate volatility and capture long-range dependencies between variables for generating time-series data.

2.1 Introdcution and basics of Artificial Neural Networks and Deep learning

Before moving on to define the architecture and how GANs work, we will give an introduction to the essential concepts of artificial neural networks (ANN), machine learning, and deep learning. For the foundational concepts of neural networks, we primarily reference [48], and for deep learning [67]. ANN is the foundational concept behind many machine learning algorithms, including deep learning. Those terms are often used interchangeably but they refer to different concepts within the field of artificial intelligence.

- **Machine Learning** is a broad field that covers all techniques that allow computers to learn from data.
- **Artificial Neural Networks** are a type of model within machine learning, inspired by the brain's structure, and used for both simple and complex tasks.
- **Deep Learning** is a specialized area of machine learning focusing on neural networks with many layers, capable of handling vast amounts of data and complex patterns.

An Artificial Neural Network (ANN) is a mathematical model that tries to simulate the structure and functionalities of biological neural networks. The fundamental unit of an ANN is the artificial neuron, which is essentially a simple mathematical function. This model operates based on three basic principles: multiplication, summation, and activation. At the input stage, each input value is multiplied by a corresponding weight. In the neuron's central part, a summation function aggregates all the weighted inputs along with a bias term. Finally, the resulting sum goes through an activation (or transfer) function, which determines the output of the artificial neuron, Figure 2.1.

ANNs are digitized models of a human brain, computer programs designed to simulate the way in which the human brain processes information. ANNs learn (or are trained) through experience with appropriate learning exemplars just as people do, not from programming. Neural networks gather their knowledge by detecting the patterns and relationships in data [19].

Definition 2.1.1. *(Neural Networks) Consider the input as a row vector* $\mathbf{x} \in \mathbb{R}^{n_1}$. *For each layer* $l = 1, \ldots, L$ *with* n_i *neurons per layer where* $i \in \{1, \ldots, L\}$, *define a scalar activation function* $\sigma : \mathbb{R} \to \mathbb{R}$. *Let* $\mathbf{w}_l \in \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l}$ *and* $\mathbf{b}_l \in \mathbb{R}^{n_l}$ *represent the weights and biases of layer* l. *The network is defined by the following feedforward equations:*

$$Z_1 = \mathbf{x}$$
$$A_l(\mathbf{x}) = \mathbf{w}_l \mathbf{x} + \mathbf{b}_l, \quad l \in \{2, \ldots, L\}$$
$$Z_l = \sigma(A_l(Z_{l-1})), \quad l \in \{2, \ldots, L-1\}$$

with the output layer:

$$Z_L = A_L(Z_{L-1}) = A_L \circ Z_{L-1} \circ \cdots \circ Z_1.$$

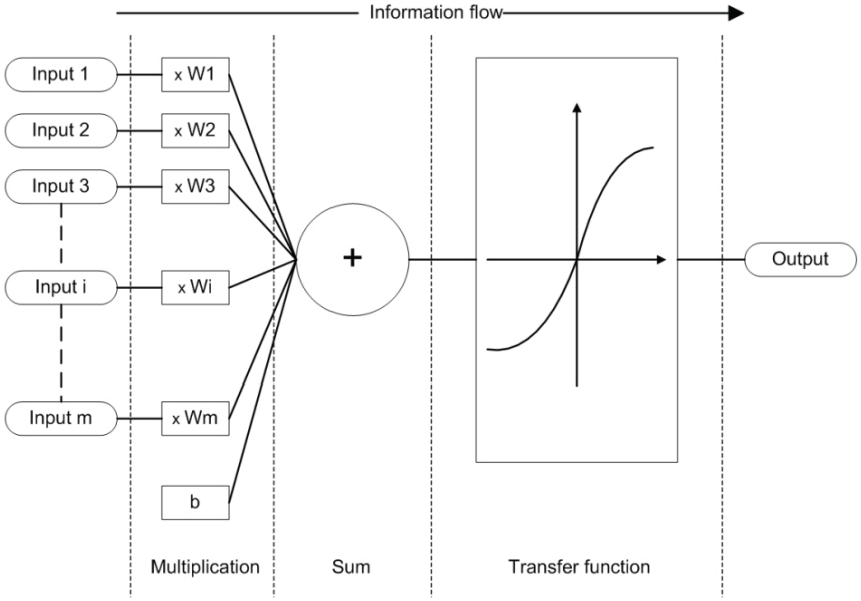


Figure 2.1: Working principle of an artificial neuron [36].

We denote the weights of the neural network to be estimated as $\theta := (\mathbf{w}_l, \mathbf{b}_l)_{l=2,\dots,L} \in \Theta$.

While the basic working principles and simple rules of artificial neurons may seem unremarkable, their true potential and computational power emerge when they are interconnected to form artificial neural networks (Figure 2.2). The first layer, i.e., the input layer consists of the input variables, which dictate the number of neurons (nodes) in this layer. The final layer produces the predicted values based on the neurons in the preceding layer, which apply the activation function to the weighted inputs. The hidden layer performs a non-linear transformation on the original input attributes and also contains neurons, with the edges between them representing the weights.

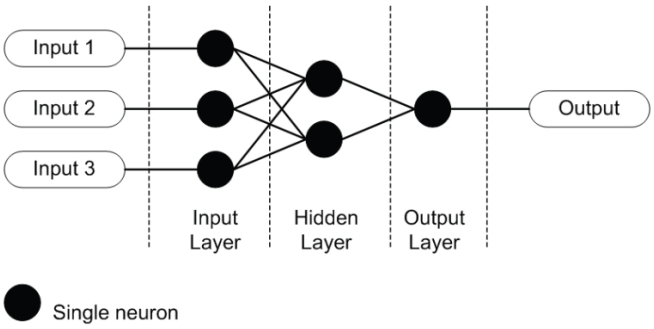


Figure 2.2: Example of simple artificial neural network.

Neurons

An artificial neuron, the fundamental unit of an Artificial Neural Network, is designed to mimic the functionality of a biological neuron, see Figure 2.3. The incoming signals, referred to as inputs, x_1, x_2, x_3, are multiplied by adjustable connection weights, w_1, w_2, w_3, and then combined through summation $\sum$. This combined input is then passed through a transfer function to generate the neuron's output. The activation function σ, which is the weighted sum of the neuron's inputs, is chosen on the basis of the problem that the artificial neuron (artificial neural network) needs to solve, and in most cases, we choose it from the following set of functions: *Step function, Linear function* and *Non-linear(Sigmoid) function.*

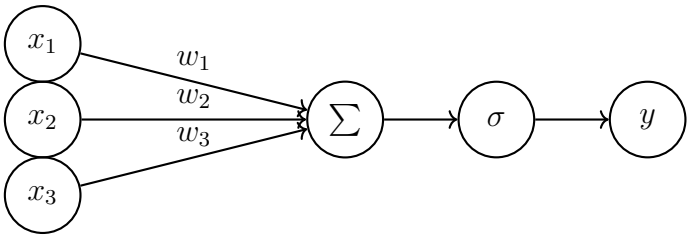


Figure 2.3: Artificial Neuron.

$$y = \sigma\left(\sum_{i=1}^{3} w_i x_i + b\right) \tag{2.1}$$

Equation 2.1 represents the neuron's output y as the activation function σ applied to the weighted sum of inputs plus a bias b.

Activation functions

The activation functions (AFs) play a very crucial role in neural networks [15] by learning the abstract features through non-linear transformations. Some common properties of the activation functions are as follows:

- It should add non-linear curvature in the optimization landscape to improve the training convergence of the network.
- It should not increase the computational complexity of the model extensively.
- It should not hamper the gradient flow during training.
- It should retain the distribution of data to facilitate the better training of the network.

Without activation functions, neural networks would only perform linear operations like matrix multiplication, resulting in purely linear transformations of the input. To enable the learning of complex patterns, neural networks use activation functions to introduce non-linearities. *Logistic Sigmoid* AF, squashes the output between $[0, 1]$ as shown in Figure 2.4. The sigmoid function is popular in the application because it can be easily differentiated and therefore reduces the computational cost during training. However, the output of the Logistic Sigmoid

function is saturated for higher and lower inputs, which leads to a vanishing gradient problem [88]. This issue occurs when the gradient of the objective function with respect to a parameter becomes nearly zero, resulting in minimal parameter updates during training with stochastic gradient descent, effectively halting the training process.

$$LogisticSigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

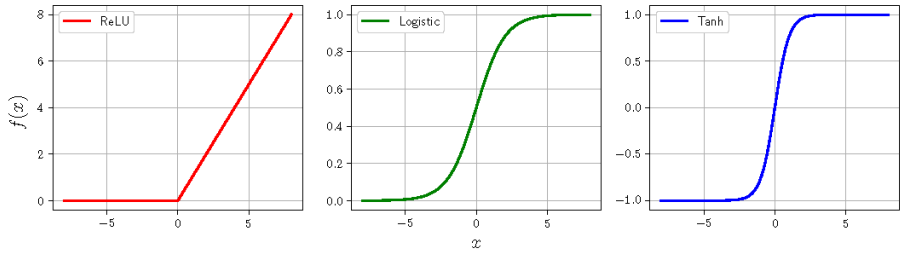


Figure 2.4: From left to right: the rectified linear unit (ReLU), the logistic, and the hyperbolic tangent activation functions [90].

The *Tanh* function shares similarities with the Logistic Sigmoid function but exhibits a zero-centered nature, as illustrated in Figure 2.4. It is defined as:

$$Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} - e^{-x}} \tag{2.3}$$

The Tanh function, similar to the Logistic Sigmoid, compresses its inputs within the range $[-1, 1]$ Due to its wider output range compared to sigmoid functions, it is occasionally more efficient for modeling complex nonlinear relationships. Nevertheless, it also inherits the drawbacks of the Logistic Sigmoid, including issues like the vanishing gradient problem and computational complexity. *Rectified Linear Unit Based Activation Functions (ReLU)*: the saturated output and increased complexity are the key liminatios of earlier Logistic Sigmoid and Tanh-based AFs. ReLU has emerged as a leading choice due to its simplicity and enhanced performance [34]. Researchers have explored several ReLU variants to mitigate its drawbacks, including the inability to use negative values, limited non-linearity, and unbounded output.

Learning Rule

There are various learning rules, but the *Delta* rule, also known as the *Backpropagation* rule, and *gradient descent* are the most commonly used. These two methods form a powerful combination in the learning process of neural networks [96]. As we described in the previous part, neural networks learn through forward propagation, utilizing weights, biases, and nonlinear activation functions to compute a predicted output, y', from the input x, aiming to closely match the true output y.

The cost (loss) function quantifies the difference between output y and prediction y'. There are various loss functions available, and the choice depends on the specific machine learning problem being addressed. Initially, when training a neural network, the inputs are assigned random weights since the optimal weights are unknown. Consequently, the network is likely to produce incorrect predictions until it is properly trained [94]. What we want is to reduce the output error. Since the weights affect the error, we will need to readjust the weights. We have to adjust the weights such that we have a combination of weights that minimizes the cost function.

Backpropagation allows us to readjust our weights to reduce output error. Rumelhart [6], describes backpropagation as a method to assign partial derivatives to individual weights and subsequently adapt the weights using gradient descent. These partial derivatives are computed with respect to the NN's cost function. The goal is to be closer to a **local minimum** after each step. The derivative is useful for minimizing a function because it indicates how to adjust x to achieve a slight improvement in y. For instance, we know that $f(x - \epsilon \cdot \text{sign}(f'(x)))$ is less than $f(x)$ for sufficiently small ϵ. Thus, we can reduce $f(x)$ by incrementally adjusting x in the direction opposite to the derivative. This technique is known as **gradient descent** (Cauchy, 1847 [1]). See Figure 2.5 for an illustration of this method.

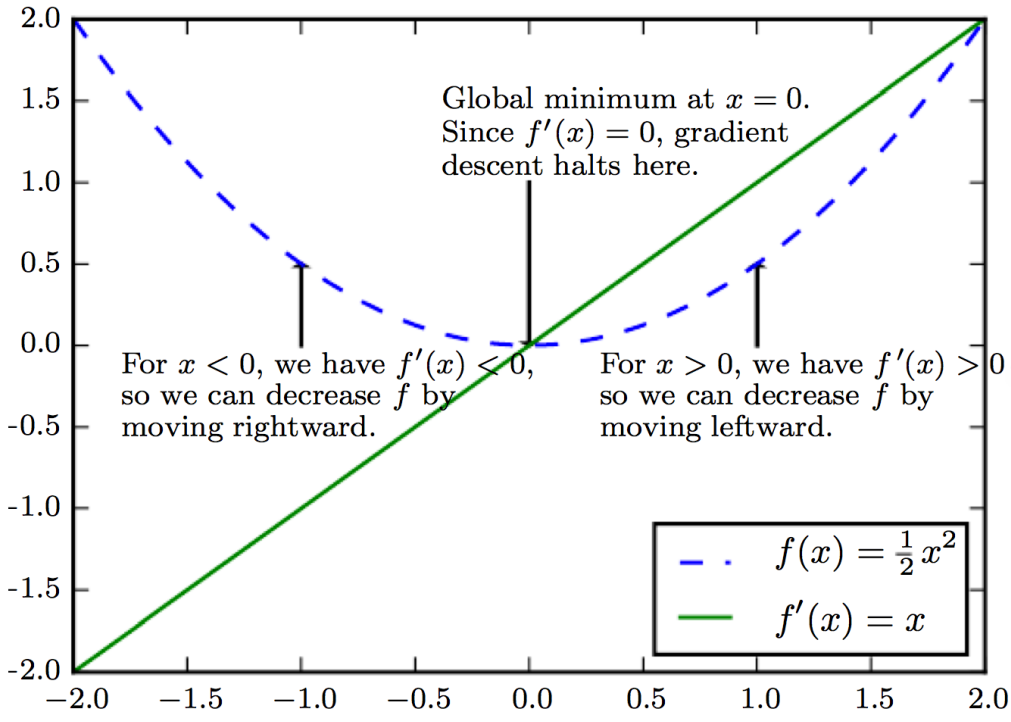


Figure 2.5: Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

Optimization algorithms can struggle to find a global minimum when faced with

multiple local minima or flat regions. In deep learning, these solutions are typically accepted if they yield significantly low values of the cost function, even if they are not the absolute minimum, see Figure 2.6. Gradient descent minimizes the

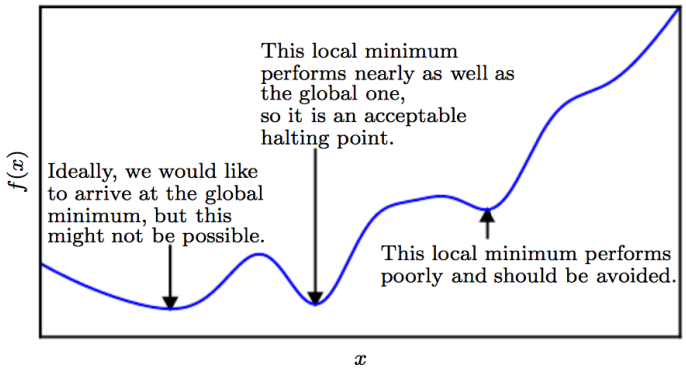


Figure 2.6: Approximate minimization.

loss function by iteratively moving in the direction of the negative gradient. The magnitude of each step, known as the **step size** or **learning rate** α can be adjusted. Let's consider the training data set $x^{(1:N)} := \{x^{(i)}\}_{i=0,\dots,N}$ and the cost function:

$$C(x^{(1:N)}, [y^{(1:N)}], \theta) : \mathbb{R}^{d_0 \cdot N}[\times \mathbb{R}^{d_1 \cdot N}] \times \Theta \to \mathbb{R}. \tag{2.4}$$

Then, the optimization algorithm, i.e., the parameter update of the gradient descent is represented as follows:

$$\theta_{n+1} = \theta_n - \alpha \nabla C(x, \theta_n), \tag{2.5}$$

where α is the learning rate.

There are different types of gradient descent: **batch**, **stochastic**, and **mini-batch**. In batch gradient descent, the gradient is computed for the entire dataset before taking a single step in the direction of the negative gradient. On the other hand, stochastic gradient descent (SGD) computes the gradient iteratively for small subsets of the dataset. Using the same notation as in 2.1.1, the following Lemma taken from [48], describes the process of Backpropagation and Gradient Descent.

Lemma 2.1.2. *(i)* $\delta_L = \frac{\partial C}{\partial (A^L)_j} = \frac{\partial C}{\partial (Z^L)_j} \frac{\partial (Z^L)_j}{\partial (A^L)_j}$

(ii) *For* $l \in \{2, \dots, L-1\}$: $\delta^l = \sigma'(A^l) \circ (w^{l+1})^T \delta^{l+1}$

(iii) *For* $l \in \{2, \dots, L\}$: $\frac{\partial C}{\partial b^l_j} = (\delta^l)_j$

(iv) *For* $l \in \{2, \dots, L\}$: $\frac{\partial C}{\partial w^l_{j,k}} = (\delta^l)_j \cdot (Z^{l-1})_k$

The above lemma enables us to develop an algorithm to calculate the derivatives iteratively. Starting with δ_L we can derive all $\delta_{L-1}, \ldots, \delta_1$ backwards. Hence, the whole algorithm is as follows:

(i) Derive $Z_1, \ldots, Z_L$ forwards.

(ii) Derive $\delta_L, \ldots, \delta_1$ backwards.

(iii) Update the parameters according to 2.5.

(iv) Repeat until satisfactory results are achieved.

Batch gradient descent

Batch gradient descent requires computing the gradient over the entire dataset for each update therefore it could be very slow and is impractical for datasets that don't fit into memory. After initializing the parameters with arbitrary values, we iteratively calculate the gradient of the cost function. This method results in a gradual and smooth update of the variables and a decrease in the cost function since every value in the training set is used simultaneously to update the parameters in each iteration.

Stochastic Gradient Descent

Since Batch Gradient Descent can be a slower algorithm to achieve faster computation, Stochastic Gradient Descent (SGD) is often preferred. Initially, the entire training set is randomized. In SGD, only one training example is used to compute the gradient of the cost function in each iteration, making it faster for larger datasets. The dataset is shuffled to introduce randomness, which is essential. Since SGD updates parameter values individually for each training sample, it creates significant noise in the cost function updates. To mitigate this noise, shuffling the dataset is crucial. The SGD calculates the gradient descent of the cost function C at a randomly chosen data point $x^{(i)}$, after calculating $C(x^{(i)}, \theta)$, the parameters will be updated as follows:

$$\theta_{n+1} = \theta_n - \alpha \nabla_\theta C(x^{(i)}, \theta). \tag{2.6}$$

SGD is significantly influenced by the learning rate. The noise in updates can be controlled by choosing an appropriate α. Moreover, SGD often reaches near the global minima in fewer iterations due to multiple parameter updates per iteration.

Mini-Batch Gradient Descent

The Mini-Batch algorithm is the most favorable and widely used, producing precise and faster results using a batch of k training examples. The whole training set is shuffled and divided into batches of k examples, which are then used for updating the gradient of the cost function[86]. Therefore, mini-batch descent calculates the gradient using a small subset of the training set:

$$\theta_{n+1} = \theta_n - \alpha \nabla_\theta C(x^{(i:i+m)}, \theta_n), \tag{2.7}$$

where m is the size of the mini-batches. Parameter updates occur in multiple batches of fixed training examples, leading to some noise in the cost function

gradient. However, compared to the Stochastic Gradient approach, this noise is significantly reduced due to the batched training method. Mini-batch gradient descent strikes a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

Machine Learning

A machine learning algorithm is an algorithm that is able to learn from data. Mitchell gives an excellent definition in his book [14] of what 'learning' means in this case: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." Machine learning, therefore, is the set of techniques concerned with getting a program to perform a task better with respect to some metric as the program gains more experience [57].

The tasks are considered as how the machine learning system should process an **example**, which is a collection of **features** taken from an event we want the machine learning algorithm to process. We denote an example as a vector $X \in \mathbb{R}^n$ where each entry x_i is another feature. There exists different tasks that can be solved by machine learning algorithms such as: *classification*, *classification with missing inputs*, *regression*, *transcription*, *anomaly detection*, *synthesis and sampling*, *density estimation* or *probability mass function estimation* and many more.

A significant step in each machine learning algorithm is to evaluate its performance. Mainly we should design a performance measure P which is specific to the task T the algorithm is solving. For example, for classification tasks, we measure the **accuracy**, i.e., the proportion of the correct outputs produced from the model, or the **error rate** the proportion of the incorrect outcomes.

The experience E of the machine learning algorithms classifies them into **supervised** and **unsupervised** learning algorithms.

Supervised learning algorithm uses labeled data sets to train or 'supervise' algorithms into classifying data or predicting outcomes accurately. Labeled data means that each example in the training data set is tagged with the answer algorithm should come up on its own. Thus, a labeled data set with flower images would tell the model which photos were of roses, daisies, and daffodils. When shown a new image, the model compares it to the training examples to predict the correct label.

Unsupervised learning algorithm uses the algorithm to analyze and cluster unlabeled data sets, which means that the data set is a collection of examples without a specific desired outcome. Hence, these algorithms detect hidden patterns in data without the need for human intervention.

We should note here that many machine learning algorithms can be used to perform both tasks, supervised and unsupervised learning since the line between them is often uncertain. For example, the general product rule in probability states that

for a vector $X \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(X) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1}). \tag{2.8}$$

This decomposition shows that we can solve the unsupervised problem of modeling the joint distribution $p(X)$ by breaking it into n supervised learning problems. On the other hand, we can solve the supervised learning problem of modeling the conditional distribution $p(y|X)$ by using unsupervised learning techniques to determine the joint distribution $p(X, y)$, then noting

$$p(y|X) = \frac{p(X, y)}{\sum_{y'} p(X, y)}. \tag{2.9}$$

Even though those two concepts are not wholly definite, there are problems such as regression, classification, and structured output that we refer to as supervised learning, and whenever we need density estimation as unsupervised learning.

A straightforward machine learning algorithm is linear regression. We build a system that can take a vector $\mathbf{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. We can denote the predicted outcome from the algorithm as

$$\hat{y} = \omega^\intercal \mathbf{x}, \tag{2.10}$$

where $\omega \in \mathbb{R}^n$ is a vector of parameters, i.e., a set of weights that determine the effect of each feature on the prediction, which is also the task T of the algorithm. To measure its performance P, we can split the data into a *training* set which is used to train the algorithm, and *test* set, and then compute for example the *mean squared error (MSE)* on the test set. The machine learning model should improve the weights ω such that this error is reduced, and we can obtain that by minimizing the MSE of the training data set.

Deep learning

Deep learning is a subset of machine learning which uses neural networks with many layers. A neural network can be thought of as a network of 'neurons' organized in layers-the first layer containing the predictors or inputs, and the top layer the forecasts or outputs. The layers in between comprise the 'hidden neurons,' making it a deep neural network. The neural networks with no hidden layers are equivalent to linear regression.

We focus on deep learning here due to the fact that GANs are a model architecture for training a generative model, and it is most common to use deep learning models in this architecture. Furthermore, note that there exists another critical classification of machine learning and deep learning algorithms: *generative* and *discriminative* modeling.

Generative models are models where the focus is on the distribution of the individual classes on a data set. The algorithm aims to model the distribution of the data points and then generate new data points based on the concept of joint

probability. On the other hand, *discriminative* models focus on the boundary between classes (labels) in a data set by modeling the conditional probability. Those algorithms are not capable of generating new data instances, and their utmost goal is to separate one class from another. For example, if we have input data x and we want to classify the data into labels y. That being the case, a generative model learns the joint probability distribution $p(x, y)$, and a discriminative model learns the conditional probability distribution $p(y|x)$. Note here that the distribution $p(y|x)$ is the natural distribution for classifying a given example x into a class y, which is why algorithms that model this directly are called discriminative algorithms. Generative algorithms model $p(x, y)$, which can be transformed into $p(y|x)$ by applying the Bayes rule and then used for classification. However, the distribution $p(x, y)$ can also be used for other purposes. For example, we could use $p(x, y)$ to generate (x, y) pairs.

In recent years, discriminative modeling has shown high success. In comparison, generative models have had a lower impact due to the gradient vanishing problem during training for generative models based on the Markov chain Monte Carlo (MCMC) algorithm, and the issue of difficult approximation of many intractable probabilistic computations appearing in maximum likelihood strategies. That is, until the proposal of GANs architecture.

Goodfellow in his paper in 2014 [41] summed up the issues regarding generative modeling presented on the following table

From the table, we can say that fundamentally, generative models were hard to train or build. Also, GANs can have their difficulties in training and design, but they have shown an essential improvement in outputs flexibility due to their convenience in setup.

Now let's review the concept of a *deep neural network* (DNN). A DNN is a collection of neurons organized in a sequence of multiple layers, where neurons receive as an input the neuron activations from the previous layer, and perform a simple computation (e.g. a weighted sum of the input followed by a nonlinear activation) [62]. The neurons of the network jointly implement a complex nonlinear mapping from the input to the output.

A DNN with L layers can be illustrated as a composition of L functions $f_i : E_i \times H_i \to E_{i+1}$, where E_i, H_i and E_{i+1} are inner product spaces for all $i \in [L]$. We will refer to the variables $x_i \in E_i$ as *state variables* and the variables $\theta_i \in H_i$ as *parameters*.

For compositions simplicity we will usually conceal the dependence of the function f_i on the parameter θ_i, i.e., f_i is taken as a function from E_i to E_{i+1} depending on $\theta_i \in H_i$. Then we can write the output of a neural network for a generic input $x \in E_1$ using the restraint function $F : E_1 \times (H_1 \times ... \times H_L) \to E_{L+1}$ based on

$$F(x; \theta) = (f_L \circ ... \circ f_1)(x), \tag{2.11}$$

where every function f_i is dependent on the parameter $\theta_i \in H_i$, and θ is the set of parameters $\theta_1, ..., \theta_L$, and for simplicity we consider that each θ_i is independent

	Deep directed graphical models	Deep undirected graphical models	Generative autoencoders	Adversarial models
Training	Inference needed during training.	Inference needed during training. MCMC needed to approximate partition function gradient.	Enforced trade-off between mixing and power of reconstruction generation.	Synchronizing the discriminator with the generator. Helvetica.
Inference	Learned approximate inference	Variational inference	MCMC-based inference	Learned approximate inference
Sampling	No difficulties	Requires Markov chain	Requires Markov chain	No difficulties
Evaluating $p(x)$	Intractable, may be approximated with AIS	Intractable, may be approximated with AIS	Not explicitly represented, may be approximated with Parzen density estimation	Not explicitly represented, may be approximated with Parzen density estimation
Model design	Nearly all models incur extreme difficulty	Careful design needed to ensure multiple properties	Any differentiable function is theoretically permitted	Any differentiable function is theoretically permitted

Table 2.1: A summary of the problems faced by different strategies to deep generative modeling for all of the main model operations.

of other parameters $\{\theta_j\}_{j\neq i}$. Moreover, for calculating the derivatives we will define the *head* and *tail* map. The *head* map at layer i, which we denote with $\alpha_i : E_1 \to E_i + 1$ is given by:

$$\alpha_i = f_i \circ ... \circ f_1 \tag{2.12}$$

for every $i \in [L]$. Moreover, α_i implicitly depends on the parameters $\theta_1, ..., \theta_i$, and for convenience we fix $\alpha_0 = I_d$, which is the identity map on E_1. Likewise, we can define the *tail* map at layer i, denoted with $\omega_i : E_i \to E_{L+1}$, as

$$\omega_i = f_L \circ ... \circ f_i \tag{2.13}$$

for each $i \in [L]$. Note that the map ω_i also depends implicitly on the parameters $\theta_1, ..., \theta_L$. Once more for convenience, we set ω_{L+1} as being the identity map on E_{L+1}. Obviously from the above equations we can see that the following holds true for all $i \in [L]$:

$$F = \omega_{i+1} \circ \alpha_i, \quad \omega_i = \omega_{i+1} \circ f_i, \quad \alpha_i = f_i \circ \alpha_{i-1}. \tag{2.14}$$

From the equations in (2.14) we can see that the output we get from a neural network can be decomposed as follows

$$F = \omega_{i+1} \circ f_i \circ \alpha_{i-1} \tag{2.15}$$

for every $i \in [L]$, where both ω_{i+1} and α_{i-1} does not depend on the parameter θ_i.

A very good mathematical background of deep neural networks including the most important concepts such as loss function, gradient descent, back-propagation, recurrent neural networks, and convolutions neural networks is given by Caterini and Chang in their book [66].

The following diagram represents a deep neural network with three hidden layers between the input and output. The higher the number of layers is the higher the number of neurons will be, which implies a much larger capacity for the neural network to learn.

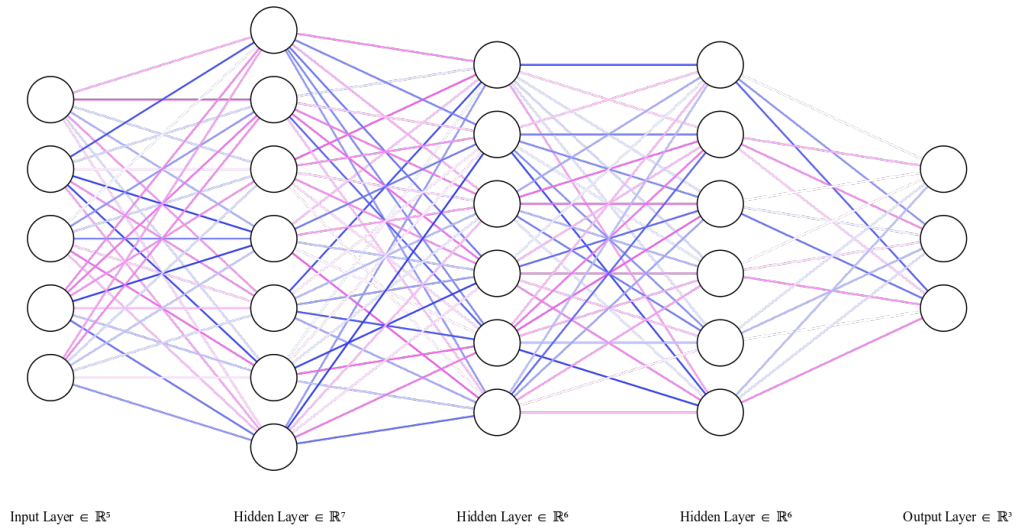


Figure 2.7: A deep neural network with three hidden layers.

This only illustrates the fundamental architecture for how a deep neural network is constructed. Many improvements and basic transformations of this architecture has been done, but still this graph gives the right pieces for applying a deep neural network. We will see in the following sections that deep neural networks are an important part of the GAN architecture.

Recurent Neural Networks - LSTM and GRU

When combining two or more artificial neurons, we create an artificial neural network. The way individual neurons are interconnected is referred to as the topology, architecture, or graph of the network. Topologies are divided into two basic classes, as shown in Figure 2.8. The left side shows a simple recurrent topology (semi-cyclic graph) where information can flow in both directions. The right side illustrates a simple feedforward topology (acyclic graph) where information flows in one direction from inputs to outputs. We will focus on RNNs here, as they are used in the architecture of GANs.

Recurrent Neural Networks (RNNs) are designed to detect patterns in sequential

data, such as handwriting, genomes, text, or numerical time series from industries like stock markets or sensors[44, 95]. They can also be applied to images when decomposed into sequences of patches

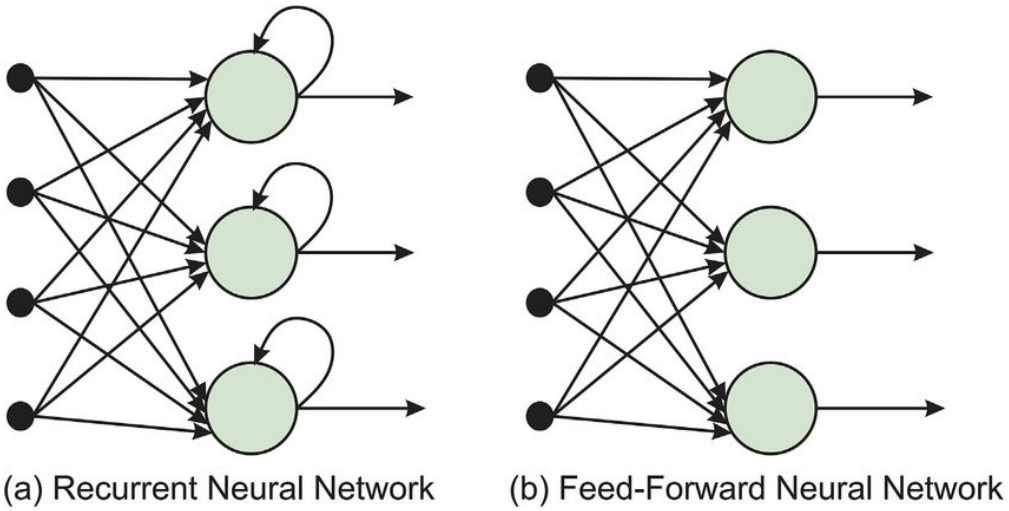


Figure 2.8: Recurrent (RNN) and Feed-forward (FNN) topology of an artificial neural network.

As we can see the key difference between Recurrent Neural Networks (RNNs) and Feedforward Neural Networks, lies in the way information is processed through the network. While Feedforward Networks pass information through the network without cycles, RNNs have cycles and transmit information back into themselves. This allows RNNs to consider previous inputs $X_{0:t-1}$ in addition to the current input X_t. This distinction is illustrated at a high level in Figure 2.9, where multiple hidden layers are represented as a single Hidden Layer block H, though this block can be extended to include multiple hidden layers.

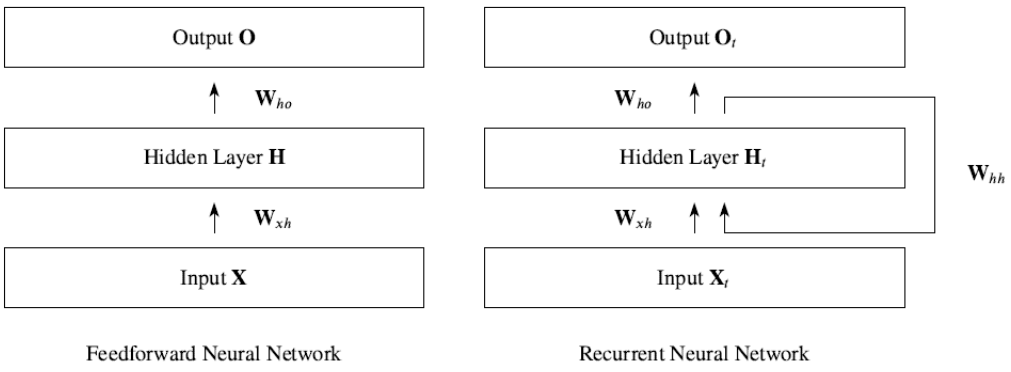


Figure 2.9: Visualisation of differences between Feedforward NNs and Recurrent NNs [80].

We describe the process of passing information from the previous iteration to the hidden layer using the notation from [84]. Let $H_t \in \mathbb{R}^{n \times h}$ and $X_t \in \mathbb{R}^{n \times d}$ represent

the hidden state and input at time step t, where n is the number of samples, d is the number of inputs per sample, and h is the number of hidden units. The weight matrix $W_{xh} \in \mathbb{R}^{d \times h}$, hidden-state-to-hidden-state matrix $W_{hh} \in \mathbb{R}^{h \times h}$, and bias parameter $b_h \in \mathbb{R}^{1 \times h}$ are used. These are passed through an activation function ϕ, usually a logistic sigmoid or tanh, to prepare the gradients for backpropagation. This yields 2.16 for the hidden variable and 2.17 for the output variable:

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h), \tag{2.16}$$

$$O_t = \phi(H_t W_{ho} + b_o). \tag{2.17}$$

Since H_t recursively includes H_{t-1}, the RNN retains traces of all previous hidden states.

Unfolding Computational Graphs

A computational graph is a method to illustrate the structure of a series of computations, including those required for mapping inputs and parameters to outputs and loss. Many recurrent neural networks employ 2.18 or a similar formula to determine the values of their hidden units.

$$h(t) = f(h(t-1), x(t); \theta) \tag{2.18}$$

The current hidden state $h(t)$ is determined by the function f, which takes as inputs the previous hidden state $h(t-1)$ and the current input $x(t)$. The parameters of the function f are denoted by θ, which is illustrated in Figure 2.10.

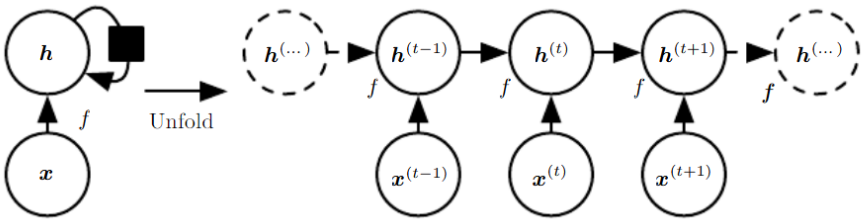


Figure 2.10: A recurrent network with no outputs. This recurrent network just processes information from the input x by incorporating it into the state that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of a single timestep. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance [48].

Typical RNNs often include additional architectural elements, such as output layers, that extract information from the state h to make predictions.

Some examples of key design patterns for recurrent neural networks include:

- Recurrent networks that generate an output at each time step and have recurrent connections between hidden units, as shown in Figure 2.11.
- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step.
- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.

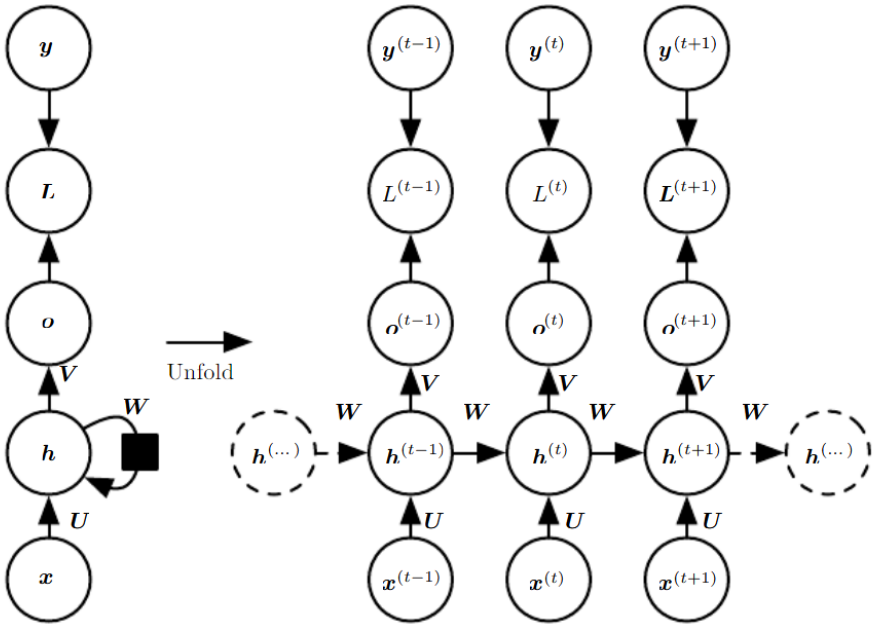


Figure 2.11: The computational graph for computing the training loss of a recurrent network that maps an input sequence x to a corresponding output sequence o [48].

We will now derive the forward propagation equations for the RNN shown in Figure 2.11. Suppose the hyperbolic tangent activation function is used for the hidden units, resulting in a discrete output, as if the RNN is utilized for predicting words or characters. The forward propagation for the illustrated figure begins with the initial state $h^{(0)}$. For each time step from $t = 1$ to $t = T$, the following update equations are applied:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}, \tag{2.19}$$

$$h^{(t)} = \tanh(a^{(t)}), \tag{2.20}$$

$$o^{(t)} = c + Vh^{(t)}, \tag{2.21}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)}). \tag{2.22}$$

The parameters include the bias vectors **b** and **c**, and the weight matrices **U**, **V**, and **W** for input-to-hidden, hidden-to-output, and hidden-to-hidden connections,

respectively. This recurrent network maps an input sequence to an output sequence of the same length. The softmax is the activation function calculated as:

$$\text{softmax}(o^{(t)}) = \frac{\exp(o^{(t)})}{\sum_{t=1}^{T} \exp(o^{(t)})}. \tag{2.23}$$

The total loss for a given sequence of $\mathbf{x}$ values paired with a sequence of $\mathbf{y}$ values is the sum of the losses over all time steps. For instance, if $L(t)$ represents the negative log-likelihood of $y(t)$ given $x(1), \ldots, x(t)$, then the total loss $L(\{x(1), \ldots, x(\tau)\}, \{y(1), \ldots, y(\tau)\})$ is given by:

$$L(\{x(1), \ldots, x(\tau)\}, \{y(1), \ldots, y(\tau)\}) = \sum_{t} L(t) \tag{2.24}$$

$$= -\sum_{t} \log p_{\text{model}}(y(t)|\{x(1), \ldots, x(t)\}) \tag{2.25}$$

where $p_{\text{model}}(y(t)|\{x(1), \ldots, x(t)\})$ is obtained by reading the entry for $y(t)$ from the model's output vector $\hat{y}(t)$.

Calculating the gradient in a recurrent neural network is straightforward. We just use the generalized back-propagation algorithm we described before on the unrolled computational graph. Gradients derived from back-propagation can then be utilized with any general-purpose gradient-based methods to train an RNN.

Problems of RNNs: Vanishing & Exploding Gradient

As in most neural networks, vanishing or exploding gradients are key problems of RNNs, which can hinder their ability to capture long-term dependencies. Exploding gradients occur when the algorithm assigns excessively high importance to the weights without a clear reason. In contrast, vanishing gradients happen when the gradient values are too small, causing the model to either stop learning or take a very long time to learn. Even if we assume the parameters are set so that the recurrent network remains stable (i.e., capable of storing memories without gradients exploding), long-term dependencies will still be present. This is due to the exponentially decreasing weights given to long-term interactions, which involve the multiplication of numerous Jacobians, compared to short-term interactions, for more details look at [8, 9].

Recurrent networks involve the composition of the same function multiple times, once per time step. These compositions can result in extremely nonlinear behavior. To simplify and illustrate long-term dependency, let's consider an RNN with all non-linearities removed. In this case, the relationship between $h(t)$ and $h(t-1)$ is reduced to simple matrix multiplication:

$$h^{(t)} = W^T h^{(t-1)}. \tag{2.26}$$

If this relationship is applied recurrently back to the initial state $h^{(0)}$, we get:

$$h^{(t)} = (W^t)^T h^{(0)}. \tag{2.27}$$

Assuming W can be decomposed into its eigenvalues and eigenvectors, we have:

$$W = Q\Lambda Q^T. \tag{2.28}$$

This simplifies the recurrence to:

$$h^{(t)} = Q^T \Lambda^t Q h^{(0)}. \tag{2.29}$$

The eigenvalues are raised to the power of t, which causes those eigenvalues with a magnitude less than one to decay to zero and those with a magnitude greater than one to explode. Consequently, any component of $h^{(0)}$ that is not aligned with the largest eigenvector will eventually be discarded. This problem motivated the introduction of *Gated recurrent units* (GRUs) and *Long short-term memory units* (LTMs), to particularly handle the vanishing gradient problem. LSTM approach was able to outperform traditional RNNs on a variety of tasks [47]. In the following section, we will look into the structure of LSTM and GRU units.

Long Short-Term Memory Units (LSTMs)

Long Short-Term Memory Units (LSTMs) were designed to overcome the problem of long-term dependencies in sequence prediction tasks. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn [45]. They were introduced by Hochreiter & Schmidhuber in 1997 [13], and were further developed and widely adopted by numerous researchers in the following studies [16, 32, 38, 43]. LSTMs use a more constant error (gradient error that is propagated back through time during the training process), enabling RNNs to learn across significantly more time steps (exceeding 1000). They achieve this by storing additional information in structures called gated cells, which are separate from the traditional neural network flow.

Let's see how do we get the constant error. There are two factors that affect the magnitude of gradients: the weights and the activation functions (or more precisely, their derivatives) that the gradient passes through.

If either of these factors is smaller than 1, then the gradients may vanish over time; if larger than 1, then exploding might happen. For example, the derivative of the `tanh` function is less than 1 for all inputs except 0; the derivative of `sigmoid` function is even worse and is always ≤ 0.25.

In the recurrence of the LSTM, the activation function is the identity function with a derivative of 1.0. Thus, the backpropagated gradient neither vanishes nor explodes when passing through, but remains **constant**.

The effective weight of the recurrence is equal to the forget gate activation. If the forget gate is on (activation close to 1.0), then the gradient does not vanish. Since

the forget gate activation is never greater than 1.0, the gradient can't explode either.

To make things work in an LSTM we use an output gate o_t to read entries of the cell, an input gate i_t to read data into the cell, and a forget gate f_t to reset the content of the cell.

Figure 2.12 shows the architecture of LSTM, which consists of a cell (which serves as the memory component), an input gate, an output gate, and a forget gate. Each component plays a unique role in the LSTM's operation:

- **Cell**: The cell maintains the state of a sequence, enabling it to retain or forget specific information.
- **Input Gate**: This gate controls the amount of information stored in the cell.
- **Output Gate**: This gate determines the value of the next hidden state.
- **Forget Gate**: This gate decides which information should be kept or get rid of.

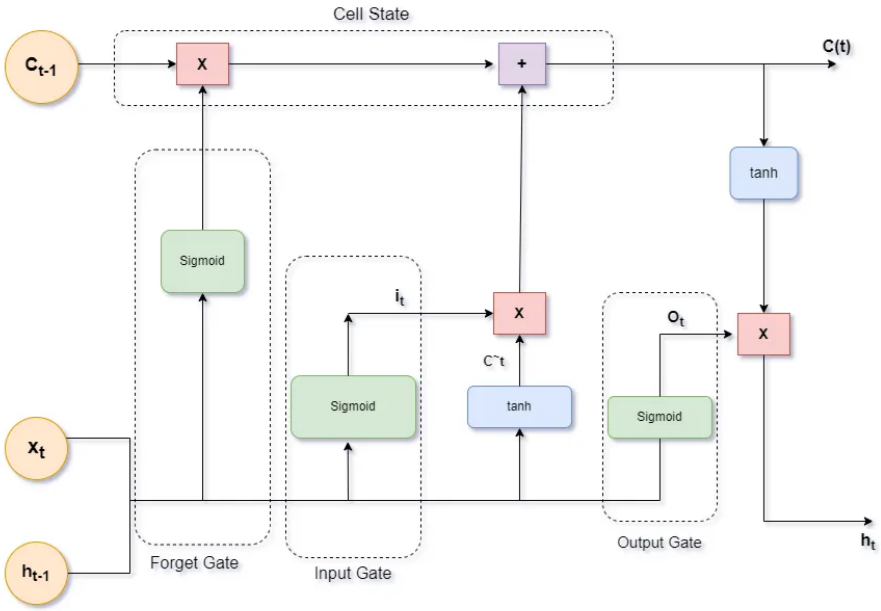


Figure 2.12: LSTM Architecture [73].

The issue of vanishing gradients is solved using LSTM, because it keeps the gradients steep enough, which ensures shorter training times and higher accuracy. The long-term dependencies and relations are encoded in the cell state vectors and it's the cell state derivative that can prevent the LSTM gradients from vanishing [68].

Step-by-step Working of LSTM

Assume we have a sequence of words $(W_1, W_2, W_3, \ldots, W_n)$ and we process each word in the sequence one at a time. Let (h_t, C_t) denote the state of the LSTM at time step t, where h_t represents the hidden state and C_t represents the cell state.

The **initial step** in an LSTM is determining which information to discard from the cell state. This is done by a sigmoid layer known as the "forget gate layer". The LSTM receives the input vector x_t and the previous state (h_{t-1}, C_{t-1}), then using that information the forget gate outputs a number between 0 and 1 for each number in the cell state C_{t-1}. The number 1 represents that the information is kept on the other hand 0 means that the information is thrown away. Figure 2.13, illustrates the computation of the first step. Referring to this figure, the equation for the forget gate can be expressed as follows:

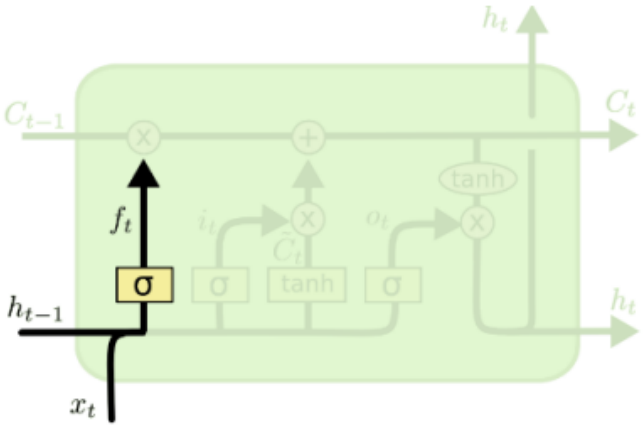


Figure 2.13: The LSTM forget gate operation.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f), \tag{2.30}$$

where W_f and b_f are model parameters. In the **next step**, the input gate i_t decides what new information to store in the cell state, which has two parts. The first part is a *sigmoid* layer called the *"input gate layer"* that decides which values we will update. The second one is a *tanh* layer which creates a vector of the new candidate values $\tilde{C}_t$ that could be added to the state. Therefore, this gate determines the relevant information to add from the current step. Figure 2.14 illustrates these calculations using the following equations:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.31}$$

$$\hat{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C). \tag{2.32}$$

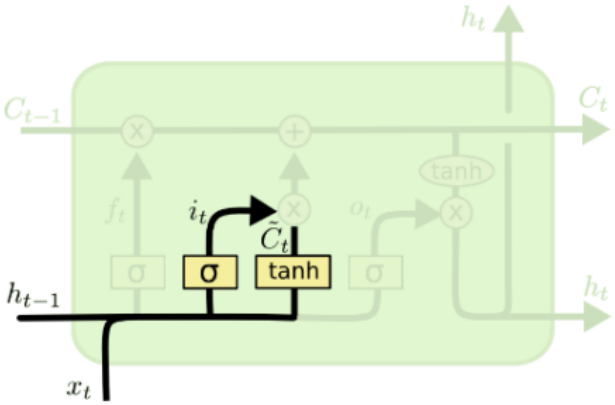


Figure 2.14: The LSTM input gate operation.

As a **further step** we update the old cell state C_{t-1} to the new cell state C_t.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t, \tag{2.33}$$

where the $*$ operator represents the Hadamard product, which is an element-wise multiplication.

The previous cell state is multiplied by f_t to remove the information that we previously decided to forget. Next, we add the new candidate values, adjusted by the extent to which we have chosen to update each state value, which is shown in Figure 2.15.

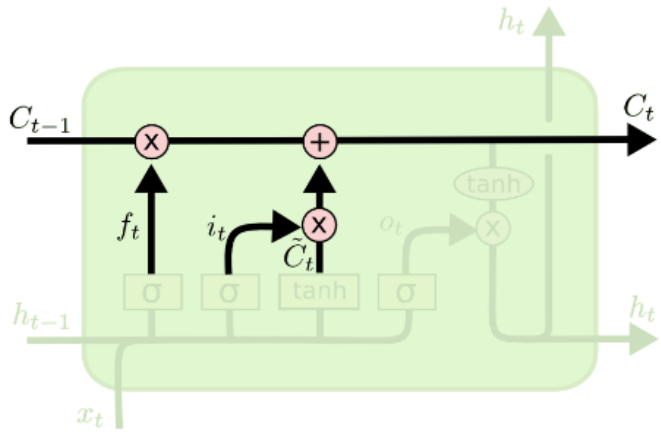


Figure 2.15: The LSTM update cell operation.

Finally we need to decide the output. The output is derived from the cell state but is filtered accordingly. Initially, a sigmoid layer determines which parts of the cell state to output. Next, the cell state is passed through a tanh function (to constrain the values between -1 and 1) and then multiplied by the output of the

sigmoid gate, making sure that we only output the parts we decided to. Therefore, the "output gate" dictates the content of the next hidden state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.34}$$

$$h_t = o_t * \tanh(C_t) \tag{2.35}$$

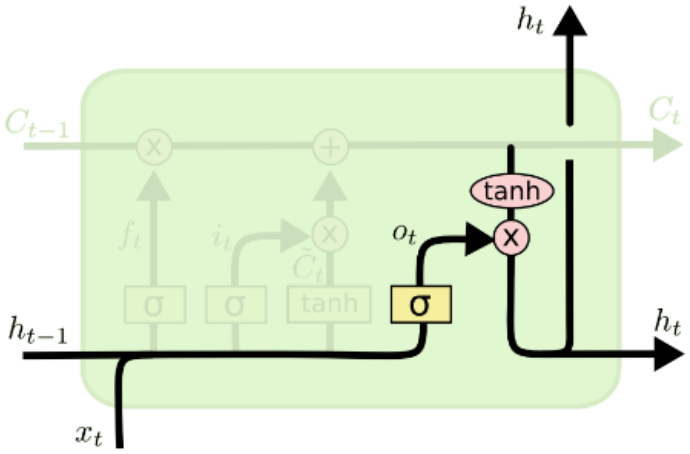


Figure 2.16: The LSTM output gate operation.

Gated recurrent units (GRUs)

A gated recurrent unit (GRU) was proposed by Cho et al. [39] as a simpler alternative to the LSTM. Similarly to the LSTM unit, the GRU has gating units that modulate the flow of information inside the unit, however, without having separate memory cells [40]. So, the **GRU** networks use only two gates to solve the vanishing gradient problem, a reset gate and an update gate.

It is not straightforward to choose between GRU and LSTM for a specific use case. The best approach is to perform trial and error to evaluate their performance. However, due to its simpler architecture, GRU requires less training time and is generally more efficient compared to LSTM.

The **first** step in a GRU is calculating the **update gate** z_t for time step t using the equation:

$$z_t = \sigma(W_z x_t + U_z h_{t-1}). \tag{2.36}$$

When x_t enters the network, it is multiplied by its own weight W_z. Similarly, h_{t-1} is multiplied by its own weight U_z. The results are summed and passed through a sigmoid activation to squash the output between 0 and 1. The update gate allows

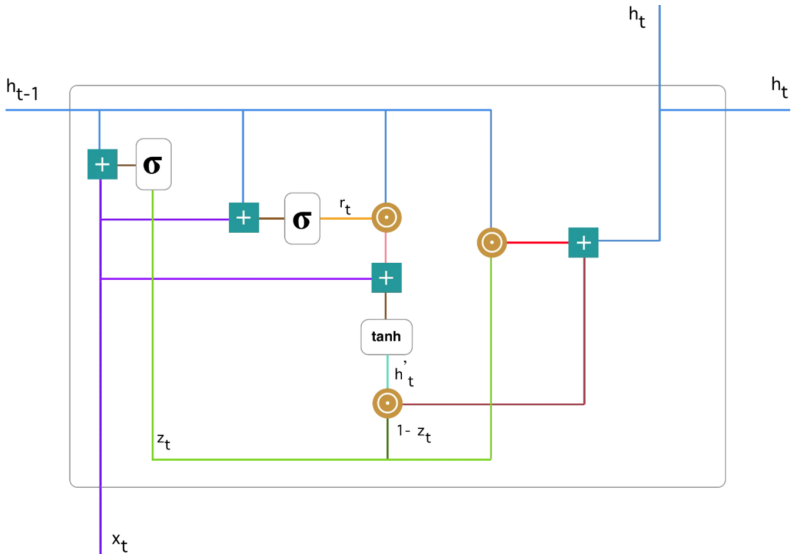


Figure 2.17: Gated Recurrent Unit [79].

the model to decide how much information from previous time steps should be carried forward.

Next step includes calculating the **reset gate** r_t, which is used from the model to decide how much of the past information to forget. It is calculated as follows:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}). \tag{2.37}$$

Further, we introduce a new **memory content** which will use the reset gate to store the relevant information from the past. It is calculated as follows:

$$h'_t = \tanh(W x_t + r_t * U h_{t-1}). \tag{2.38}$$

As the **final** step, the network computes h_t, a vector containing information for the current unit that is passed through the network. To achieve this, the update gate determines what to retain from the current memory content h'_t and what to keep from previous steps h_{t-1}. This is performed as follows:

$$h_t = z_t * h_{t-1} + (1 - z_t) * h'_t \tag{2.39}$$

1. Perform element-wise multiplication with the update gate z_t and h_{t-1}.
2. Perform element-wise multiplication with $(1 - z_t)$ and h'_t.
3. Sum the results from the first and second step.

Hence, this is how GRUs are able to store and filter the information using their update and reset gates. That eliminates the vanishing gradient problem since the model does not wash out the new input every single time but keeps the relevant information and passes it down to the next time steps of the network.

Chapter 3

Architecture of GANs

To describe GANs better, Kalin mentions a real-world analogy in his book [70]. This metaphor which Ian Goodfellow himself also uses, considers the classical analogy of GANs as the counterfeiter, who is the generator, and FBI agent, who is the discriminator. The counterfeiter is constantly looking for new ways to produce fake documents that can pass the FBI agent's tests. The more authentic-looking the counterfeit documents become, the better the detective must be at detecting them, and vice versa. Let's simplify this into a set of goals:

1. The *counterfeiter's (generator's)* goal is to create products such that the agent won't be able to distinguish between the real and fake ones.
2. The *cop's (discriminator's)* goal is to notice anomalous products by means of his previous experience to categorize the real and fake products.

A very good analogy to explain GANs was presented by Korn during CONVENTION A (2024). Imagine a Swiss chocolate expert (our discriminator) who has tasted the finest Swiss chocolates but doesn't know the exact recipe. A foreign chocolate company (our generator) wants to produce its own version of Swiss chocolate, as illustrated in Figure 3.1. They begin by making batches of chocolate and presenting samples to the Swiss expert for evaluation.

Each time the company produces a batch, the expert tastes it and provides honest, constructive feedback:

- "This batch is close but has too much sugar."
- "This one is almost there, but it contains too much milk."
- "The flavor profile is off—try reducing the bitterness."

The expert's role is to **discriminate** between the real Swiss chocolate and the company's attempt at replication. But he must provide helpful comments for the company to improve.

On the other side, the chocolate company takes these notes and refines its process, adjusting the recipe and producing new batches that gradually get closer to the

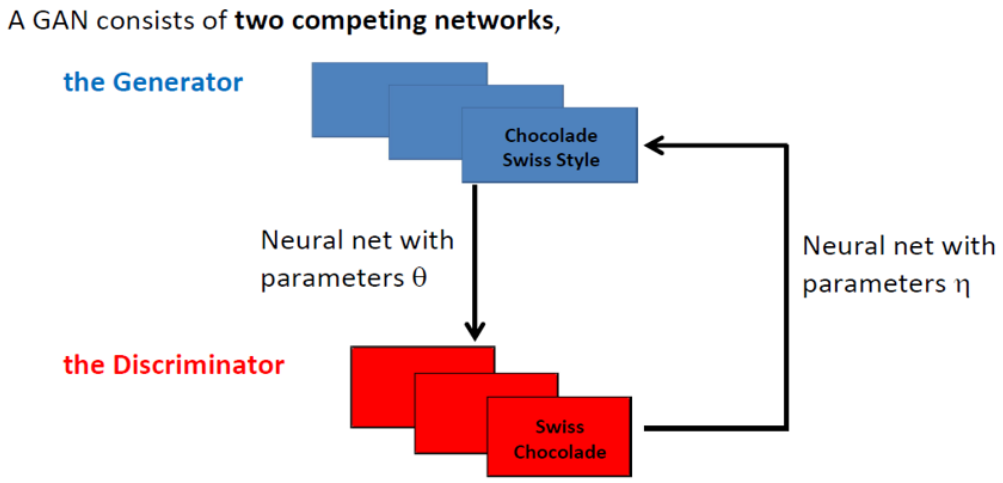


Figure 3.1: Swiss chocolate GAN analogy: Discriminator (chocolate expert) and Generator (chocolate company) cooperate to create Swiss-style chocolate [93].

authentic Swiss chocolate style.

This process continues over many iterations: the chocolate expert (**discriminator**) keeps giving more accurate feedback as he distinguishes the real Swiss chocolate from the company's, and the company (**generator**) continues improving the quality of its product. Eventually, the company produces chocolate that is not exactly the same as Swiss chocolate but is so close in style and quality that it becomes difficult to tell the difference.

Cooperation between the two parties is key: if the chocolate expert doesn't provide honest, useful feedback, or if the company doesn't take the feedback seriously, the process breaks down. But if both parties do their jobs well, they create a product that successfully mimics Swiss chocolate in quality and flavor.

In more technical terms, the generator aims to generate examples that capture the features of the training data set so well that the samples it produces look almost identical to the training data. The input into the generator is usually a vector of random numbers, which means that fundamentally generator learns to create patterns from scratch.

The generator learns by the feedback from the discriminator's classification whether the data is real or fake. The discriminator aims to detect if the particular example is coming from the data training data set and classifies it as accurate or is created by the generator and classifies it as fake. Correspondingly, each time the discriminator is fooled into classifying a fake example as real, the generator knows it did something well. Conversely, each time the discriminator correctly rejects a generator-produced example as fake, the generator receives the feedback that it needs to improve.

The discriminator keeps improving as well. It learns from how far its predictions are from the actual labels (real or fake). Thus, as the generator improves at generating realistic-looking data, the discriminator gets better at distinguishing

simulated data from the real ones, and both networks keep getting better accordingly.

The Figure 3.2 below represents the basic architecture of GANs. As implied from the previous explanation GAN learns to map the simple latent distribution to the more complex data distribution which we will denote with p_{data}. Moreover, the complete framework of GANs corresponds to a two-player minmax game between the generator and discrminator. Formally, the generator and discriminator are described by differentiable functions, for instance neural networks, where both have their own cost (loss or objective) function. Each network is trained using backpropagation by utilizing the discriminator's loss. The discriminator attempts to minimize the loss for both real and fake examples, while the generator tries to maximize the discriminator's loss for the fake examples that it produces.

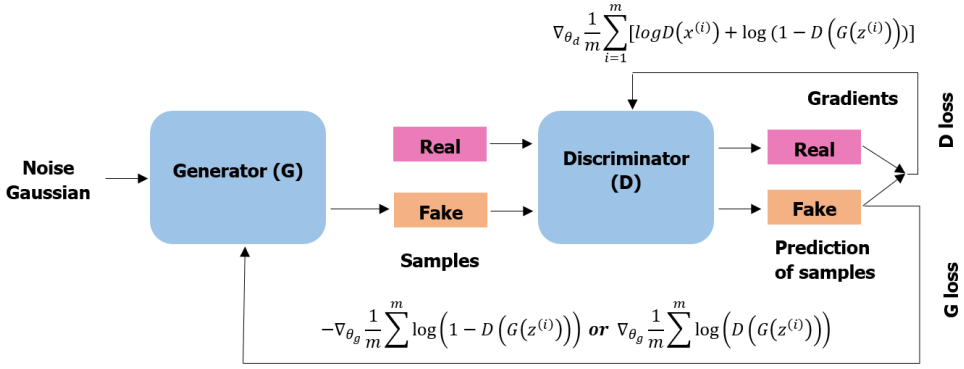


Figure 3.2: Basic GANs architecture.

Here, we have x which is the data taken from the real data distribution, p_{data}, a noise vector z extracted from a Gaussian prior distribution with zero-mean and unit variance p_z, and with p_g we denote the generator's distribution over data x. We give the latent vector z to the generator as an input and then the generator produces an example $G(z)$. The generator aims that the discriminator cannot distinguish between $G(z)$ and $D(x)$ data samples, i.e., $G(z)$ is very similar to $D(x)$. On the other hand, the discriminator is always trying to not get tricked by the generator. In GANs the discriminator is a classifier such that,

$$D(x) = \begin{cases} 0 & \text{if} \quad x \sim p_{data} \\ 1 & \text{if} \quad x \sim p_g \end{cases} \tag{3.1}$$

i.e., x comes either from p_{data} or p_g.

3.1 Loss (cost or objective) function

The loss function tries to match the distribution of the real data p_{data} with the generator's distribution p_g. Both the generator and the discriminator have their

own loss functions, for which we use the notation J_G and J_D, respectively. Moreover, we denote the trainable parameters (weights and biases) of the two networks with: θ_D for the discriminator which are trained to maximize the loss to differentiate between the real and fake samples, and with θ_G the parameters for the generator which are trained to minimize the loss that θ_D tries to maximize. We should note here that GANs differ from conventional neural networks in two key aspects as clarified in [78]. First, the cost function of a traditional neural network is determined explicitly in terms of its own trainable parameters, θ. While, in GANs the cost functions of both networks depend on their trainable parametrs, i.e., θ_D and θ_G. The second difference in that a traditional neural network can tune all its parameters during the training process, yet in GANs each network can tune only its own weights and biases. Hence, from what is said until know it implies we have a zero-sum game where the players compete with each other. The following minimax objective applied for training G and D models jointly via solving [85]:

$$\min_{\theta_G} \max_{\theta_D} V(G, D) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))], \tag{3.2}$$

where $V(D, G)$ is a binary cross entropy function, commonly used in binary classification problems [56]. In order to update the parameters of Equation 3.2, we train G and D by backpropagating the loss through their respective models. Actually the above equation is solved by interchanging the succeeding gradient updates:

$$\theta_D^{t+1} = \theta_D^t + \lambda^t \bigtriangledown_{\theta_D} V(D^t, G^t) \quad \text{and} \quad \theta_G^{t+1} = \theta_G^t + \lambda^t \bigtriangledown_{\theta_G} V(D^{t+1}, G^t), \tag{3.3}$$

where θ_G is the parameter of G (generator), θ_D the parameter of D (discriminator), λ is the learning rate, and t is the number of iterations. Moreover, note here that the generator learns through a differentiable function $G(z; \theta_G)$ represented by a deep neural network (multiple layer perceptron) with parameter θ_G. The discriminator is given by $D(x, \theta_D)$. D is trained to maximize $\log D(x)$, where $D(x)$ outputs the probability that $'x'$ is from the original data set, as G is trained to minimize $\log(1 - D(G(z)))$.

Goodfellow et al. [41] proposed an iterative approach of optimization to avoid over-fitting the discriminator D. They alternate between k steps of optimizing D and one step of optimizing G. Additionally, it was found that the second term of the Equation 3.2, $\log(1 - D(G(z))$, saturates and makes insufficient gradient flow through G, which means that the gradient value gets smaller and stops learning. Therefore, the objective function in 3.2 can be modified such that in can be redefined into two different objectives as

$$\begin{aligned} &\max_{\theta_D} \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z))], \\ &\text{and} \quad \max_{\theta_G} \mathbb{E}_{z \sim p_z}[\log(D(G(z)))]. \end{aligned} \tag{3.4}$$

Furthermore, after we compute the cost in the previous equations, we can use back-propagation to update the parameters of the model. Since we have two different objective functions, the update rule is given as

$$\theta_D^{t+1}, \theta_G^{t+1} \leftarrow \begin{cases} \text{Update if} \quad D(x) \quad \text{predicts wrong,} \\ \text{Update if} \quad D(G(z)) \quad \text{predicts wrong,} \\ \text{Update if} \quad D((G(z)) \quad \text{predicts correct.} \end{cases} \tag{3.5}$$

If G and D have enough capacity with adequate training iterations, G can transform a simple latent distribution p_g to more complex distributions, i.e., p_g converges to p_{data}, such as $p_g = p_{data}$.

3.2 Optimization algorithm

In GANs at the optimization step, we aim to find the global equilibrium of the previously explained minimax game, i.e., the saddle point of the minimax objective function. For solving minimization and saddle point problems, gradient-based optimization algorithms are usually used. In GANs any gradient-based algorithm can be used to minimise the loss of each player accordingly which leads to Nash equilibrium. For the basic GANs model we use the *Simultaneous Gradient Descent* to find the equilibrium and we update the parameters for the generator and discriminator simultaneously by using gradient descent on their utility functions. For finding a Nash equilibrium (θ_D, θ_G) both players aim to minimize their own cost function, J_D, J_D, respectively, where J_D is at minimum with respect to θ_D and J_G is at minimum with respect to θ_G.

Now let's show the *minimax optimization for Nash equilibrium*. We should note here that in a zero-sum game, the loss of all the players is always zero. Moreover, since in the zero-sum game, the solution holds minimization and maximization in an outer and inner loop, respectively, it is known as a minimax game. Thus,

$$J_g + J_D = 0. \tag{3.6}$$

All types of GANs modeled until now use the same cost function for the discriminator, J_D, but they differ in the generator's loss function, J_G. D's cost function w.r.t. θ_D is defined as

$$J_D(\theta_D, \theta_G) = -\frac{1}{2}\mathbb{E}_{x\sim P_{data}}[\log D(x)] - \frac{1}{2}\mathbb{E}_z[\log(1 - D(G(z)))], \tag{3.7}$$

and it presents the standard cross-entropy cost, which is minimized during training a typical binary classifier with a sigmoid output. In the minimax game, the generator aims to maximize the log probability of D being mistaken and minimize that of being correct. In GANs, to train a classifier we usually use two mini-batches

of data, a real data mini-batch where the examples have the label 1, and a mini-batch from G with the example's label 0, i.e., the density ratio between true and generated data is

$$D(x) = \frac{p_{data}(x)}{p_g(x)} = \frac{p(x|y=1)}{p(x \mid y=0)} = \frac{p(y=1 \mid x)}{p(y=0 \mid x)} = \frac{D^*(x)}{1 - D^*(x)}. \tag{3.8}$$

Here, $y = 0$ means that the data are generated, $y = 1$ means real data, and $p(y = 1) = p(y = 0)$ is assumed.

During the training of D we obtain the estimate of the ratio $p_{data}(x)/p_g(x)$ for every data point x which factors in the calculation of divergences and their gradients. As mentioned, the discriminator learns to detect the real data from the generated ones for any G.

To find the optimal value of $D(x)$, we follow these steps: first, compute the derivative of Equation 3.8 with respect to $D(x)$. Then, we set this derivative equal to zero to identify the optimal point.

$$J_D(\theta_D, \theta_G) = -\frac{1}{2}\int p_{data}(x)[log(D(x))dx] - \frac{1}{2}\int p_g(x)[log(1 - D(x))dx] \tag{3.9}$$

$$\frac{J_D(\theta_D, \theta_G)}{dD(x)} = -\frac{1}{2}\int p_{data}(x)\frac{1}{D(x)}dx - \frac{1}{2}\int p_g(x)\frac{1}{(1 - D(x))}dx \tag{3.10}$$

$$\frac{1}{2}\int p_{data}(x)\frac{1}{D(x)}dx = \frac{1}{2}\int p_g(x)(1 - D(x))dx \tag{3.11}$$

If integrands are equal for all x, i.e.,

$$p_{data}(x)\frac{1}{D(x)} = p_g(x)(1 - D(x)) \tag{3.12}$$

i.e.,

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}. \tag{3.13}$$

When we insert the optimal discriminator into the objective function we get

$$\begin{aligned} &C(G) = \max_{\theta_D}(V(G, D)) = \\ &\mathbb{E}_{p_{data}}\left[\log \frac{p_{data}(x)}{p_{data(x)} + p_g(x)}\right] + \mathbb{E}_{p_g}\left[\log \frac{p_g(x)}{p_{data(x)} + p_g(x)}\right]. \end{aligned} \tag{3.14}$$

Before we continue to the global optimum we need to introduce the definition of Kullback–Leibler divergence and the Jensen–Shannon divergence.

Definition 3.2.1 (Kullback–Leiblern (KL) divergence)**.** *The KL divergence it's a measure of how much a given distribution differs from the second distribution. Let P and Q be two probability distributions of a continuous random variable x. The definition of KL divergence is,*

$$D_{KL}(P||Q) = \int p(x) \log \frac{p(x)}{q(x)}. \tag{3.15}$$

Definition 3.2.2 (Jenson- Shanon (JS) divergence)**.** *The JS divergence measures the similarity between two probability distributions. For two probability distributions P and Q of a random varaible x, the JS divergence is,*

$$JSD(P||Q) = \frac{1}{2} D_{KL}\left(P \middle|\middle| \frac{P+Q}{2}\right) + \frac{1}{2} D_{KL}\left(Q \middle|\middle| \frac{P+Q}{2}\right). \tag{3.16}$$

In [25] it was proven that the square root of the JS-divergence indeed defines a probability metric. In particular, we can make use of the positive-definiteness to further simplify the GAN objective function (3.14).

Theorem 3.2.3. *The virtual training criterion C(G) achieves its global minimum if and only if $p_g = p_{data}$. In this case it holds $V(G^*, {D_G}^*) = -\log(4)$ and $D_G^* = \frac{1}{2}$.*

The following proof, adapted from Goodfellow et al. (2014) [41], is included here for clarity and completeness.

Proof. For $p_g = p_{data}$, $D_G^* = \frac{1}{2}$ follows directly from equation (3.13). Then, by inquiring Eq. (3.14) at $D_G^* = \frac{1}{2}$, we get $C(G) = \log(\frac{1}{2}) + \log(\frac{1}{2}) = -\log 4$. To see that this is also the global minimum point of $C(G)$ we subtract,

$$\mathbb{E}_{x\ p_{data}}[-\log 2] + \mathbb{E}_{x\ p_g}[-\log 2] = -\log 4$$

from $C(G)$ and get,

$$C(G) = -\log 4 + D_{KL}\left(p_{data} \middle|\middle| \frac{(p_{data}+p_g)}{2}\right) + D_{KL}\left(p_g \middle|\middle| \frac{(p_{data}+p_g)}{2}\right). \tag{3.17}$$

Now, we use the definition of JS-divergence and get,

$$C(G) = -\log 4 + 2 \cdot JSD(p_{data}||p_g). \tag{3.18}$$

The positive definiteness of the JS-divergence then implies the desired result. □

After appropriate training steps, G and D with sufficient capacity will converge, implying that the discriminator won't be able to differentiate between the two distributions, i.e., $p_g = p_{data}$. For the optimal D, we can reformulate the minimax game in Equation 3.2 as follows

$$J_d = -2log2 + 2(D_{JS}(p_{data}\|p_g)) \tag{3.19}$$

where D_{JS} is the Jensen-Shannon[1] divergence. In the case of an optimal D, G minimizes the Jensen-Shannon divergence (JDS), which is bounded by $[0, 1]$. In the basic GAN architecture, it is possible to estimate neural samplers over approximate minimization of the symmetric JDS

$$D_{JS}(p_{data}\|p_g) = \frac{1}{2}D_{KL}(p_{data}\|\frac{1}{2}(p_{data} + p_g)) + \frac{1}{2}D_{KL}(p_g\|\frac{1}{2}(p_{data} + p_g)), \quad (3.20)$$

where D_{KL} denotes the Kullback–Leibler[2] divergence.

Since $D_{JS}(p_{data}\|p_g)$ is a suitable measure of divergence between two distributions, i.e., we can correctly approximate the distribution of the real data p_{data} considering that adequate training samples exist and model class p_g can represent p_{data}. It is important to note here that the JSD between two distributions is a non negative value when two distributions are equivalent, therefore we get that $J_D* = -2log2$ is the global minimum of J_D which implies that $p_g = p_{data}$. So, at the point where $p_g = p_{data}$, both maximizing player and minimizing player will achieve their goals, i.e., it is our equilibrium point.

Goodfellow et al. came up with the following algorithm proving that it will get us to the optimal point.

So, at the point of $p_g = p_{data}$, both maximizing player and minimizing player will achieve their goals. In other words, that is our equilibrium point.

The global optimum i.e., $p_g = p_{data}$ in GANs occurs at a saddle point θ_G, θ_D. The saddle point of a loss function arises at a point which is the minimal point w.r.t. one pair of parameters and the maximal one w.r.t. another pair of parameters. The GAN training procedure includes swapping between maximization and minimization steps. If one step is more powerful than the other one, then the equilibrium path "slips off" the loss surface as shown in Figure 3.3, therefore, training becomes unstable and the network collapses [63]. For simplicity, since p_g depends completely on G, we will use p_g instead of G in Figure 3.4 in order to give a detailed explanation. For a better understanding, we will draw a plot of p_g versus $V(G, D)$, by making $D(x) = D^*(x)$ the optimal D at each point of p_g in Figure 3.3. We get this point when we apply the gradient descent algorithm for the discriminator D, for each fixed point p_g as in the inner loop of Convergence Algorithm, i.e., when we take at each p_g the maximum value of $V(G, D)$ by varying $D(x)$ and plot p_g versus $V(G, D)$ we obtain the following Figure 3.4. From the above statement, we note here that if we have a set of derivatives of $V(D^*, G)$ w.r.t. p_g, it will include all the partial derivatives of $V(D, G)$ w.r.t. p_g, at the locations where $V(G, D)$ is maximum for a given p_g (see the Graph 3.3). This means that the derivative at the red dot in Figure 3.3 is somewhere in the set of derivatives of the function in Figure 3.4. When applying the gradient descent on the parameters of $G(z)$, i.e.,

[1]Jensen–Shannon divergence measures the similarity between two probability distributions based on the Kullback–Leibler divergence, with some significant changes, in particular that it is symmetric and it always has a finite value.

[2]The Kullback–Leibler divergence is a measure of how one probability distribution is different from a second, reference probability distribution.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of raining iterations **do**

for k steps **do**

- Sample minibatch of m samples $z^{(1)}, ..., z^{(m)}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $x^{(1)}, ..., x^{(m)}$ from data generating distribution $p_{data}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\bigtriangledown\theta_d \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log(1 - D(G(z^{(i)})))].$$

end for

- Sample minibatch of m samples $z^{(1)}, ..., z^{(m)}$ from noise prior $p_g(z)$.
- Update the generator by ascending its stochastic gradient:

$$\bigtriangledown\theta_d \frac{1}{m} \sum_{i=1}^{m} \log(1 - D(G(z^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

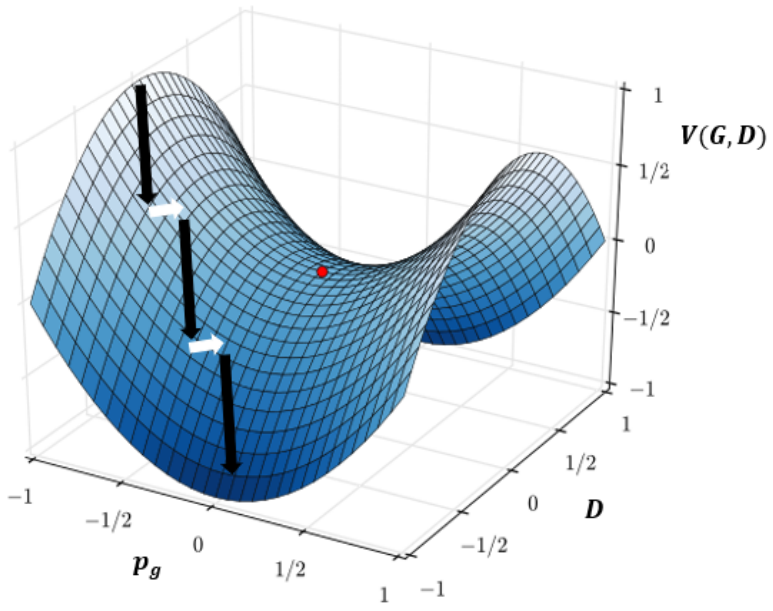


Figure 3.3: An illustration of the gradient method of Adversarial Net.

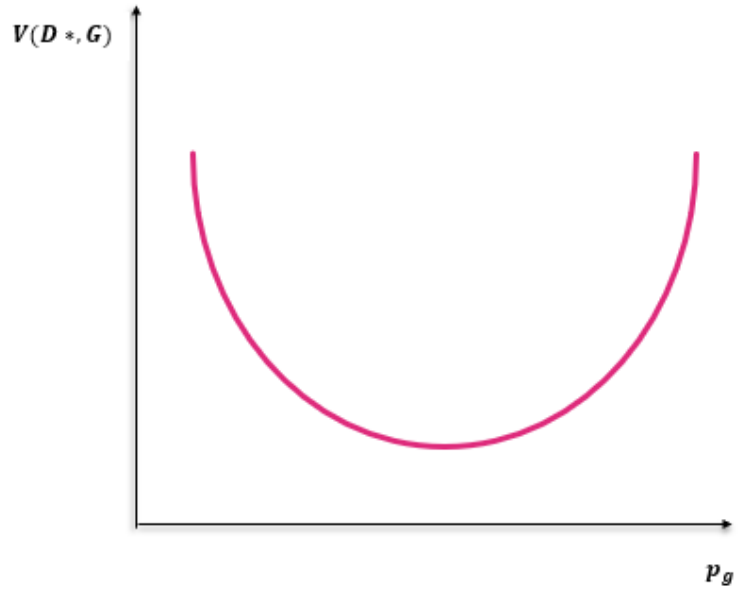


Figure 3.4: Minimax objective versus the generated data distribution.

partial differentiating $V(D^*, G)$ w.r.t. p_g, we update the generated data p_g and attain the red dot.

With other words this means, when applying gradient descent to D, for a given distribution p_g, we obtain the optimal D for the particular p_g, which represents the inner loop of the convergence algorithm. Then keeping D(X) fixed, we apply gradient descent to the distribution p_g and we get closer to the global optimum, i.e., the outer loop of the convergence algorithm.

Therefore, since the partial derivatives of p_g at the points where the discriminator D is optimal include the global optimum (red point), when giving enough capacity to D and G, we will eventually reach the global optimum using the convergence algorithm.

3.3 Challenges and evolution of GANs

In the previous sections of this Chapter, we have seen that GANs are based on the min-max game. In this game, the players compete (cooperate) with each other, where one player maximizes its actions (discriminator), the other player tries to minimize them (generator) and the model converges when together the discriminator and the generator arrive at the Nash equilibrium stage. However, finding Nash equilibrium in GANs is a very challenging problem as objective functions are non-convex, parameters are continuous, and parameter space is high-dimensional [51], we saw this when we described Figure 3.3, e.g., an update to θ_D that decreases

J_D can increase J_G, and an update to θ_G that decreases J_G can increase J_D, i.e., training fails to converge. Thus, GANs are hard to train, and training experiences issues such as convergence difficulties, instability, and mode collapse.

3.3.1 Mode Copllapse

Mode collapse happens when the generator focuses on producing a limited set of data patterns that deceive the discriminator. It becomes fixated on a few dominant modes in the training data and fails to capture the full diversity of the data distribution [92].

Figure 3.5 shows GANs training on a toy dataset in which G produces only a single mode instead of having multi-model training dataset and keeps cycling between different modes as D continuously rejects samples generated by G. This shows GANs oscillates and have difficulty in achieving Nash equilibrium. Mode collapse is one of the key reasons for the cause of unstable GANs training – another big challenge in GANs.

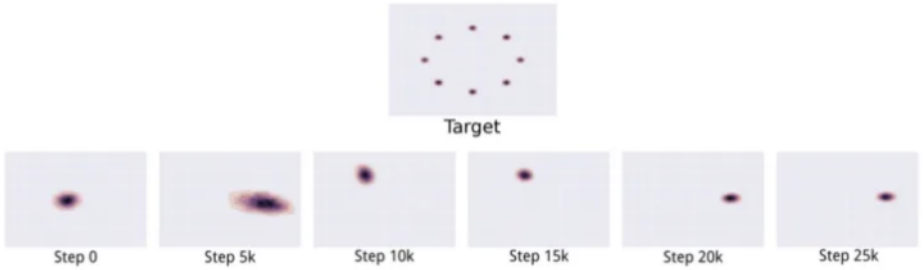


Figure 3.5: An illustration of the mode collapse problem on a two-dimensional toy dataset [49] .

An inappropriate objective function can significantly contribute to mode collapse. To address this, various GAN variants have been developed, including modifications to the discriminator's objective function [65][58] and changes to the generator's objective function [81]. In these variants, the generator theoretically can learn the entire data distribution at equilibrium, but achieving convergence in practice remains challenging.

3.3.2 Non-convergence and instability

In traditional GANs, the generator (G) uses two loss functions as we already described: $\mathbb{E}[\log D(G(z))]$ and $\mathbb{E}[\log(1 - D(G(z)))]$. However, these can lead to

issues in training. The first loss function can cause the gradient vanishing problem when the discriminator (D) can easily distinguish real from fake samples. When D is optimal, minimizing G's loss is like minimizing the Jensen-Shannon Divergence (JSD) between the real and generated distributions, as already discussed, resulting in a JSD of $\log 4$. This means D assigns a probability of 0 to fake samples and 1 to real ones, causing G's gradients to vanish, as shown in Figure 3.7.

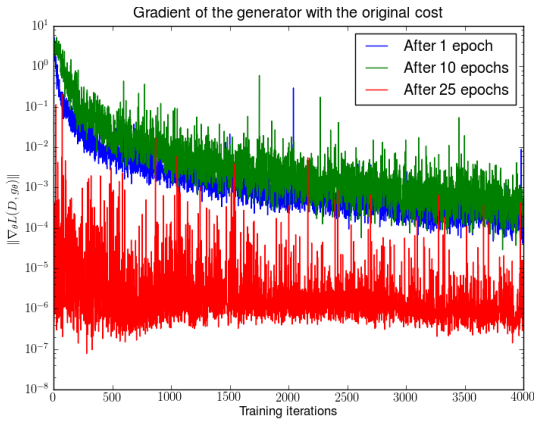


Figure 3.6: DCGANs for 1, 10 and 25 epochs. G is fixed while D is trained from scratch and using original cost function to compute the gradients [55].

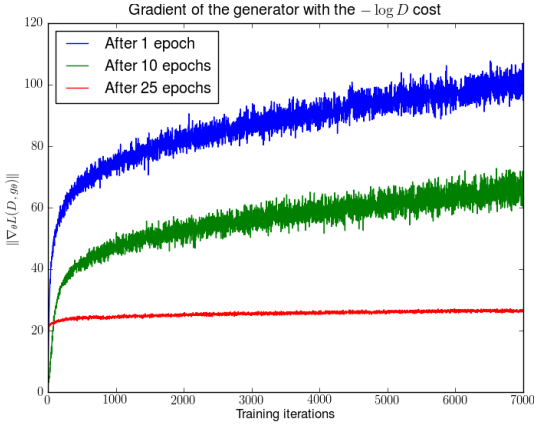


Figure 3.7: DCGANs for 1, 10 and 25 epochs. G is fixed, D is trained from scratch and using -log D cost function to compute the gradients [55].

To reduce cross-entropy between a target class and a classifier's predicted distribution, the classifier needs to select the correct class [91]. In GANs, D tries to minimize cross-entropy while G tries to maximize it. When D is highly confident, it rejects G's samples, causing G's gradient to vanish. One solution is to reverse the target for constructing the cross-entropy cost, which changes G's cost to:

$$J_G = -\frac{1}{2}\mathbb{E}_z[\log D(G(z))], \quad \text{or} \tag{3.21}$$

$$J_G = -\frac{1}{2}\mathbb{E}_z[\log(1 - D(G(z)))]. \tag{3.22}$$

The second loss function is known as the $-\log D$ trick [41] [54]. Minimizing G's loss function, $\mathbb{E}_z[\log(1 - D(G(z)))]$, is equivalent to minimizing $D_{KL}(p_g \| p_{data}) - 2(D_{JS}(p_{data} \| p_g))$. This leads to unstable gradients because it minimizes the KL divergence while maximizing the JSD, causing gradient instability. Figure 3.7 shows that G's gradients increase rapidly, and their variance grows, resulting in the generation of low-quality samples.

3.3.3 A Taxonomy

Saxena and Cao in [91] proposed a taxonomy of GAN design and optimization solutions to address the two major training challenges: non-convergence and instability.

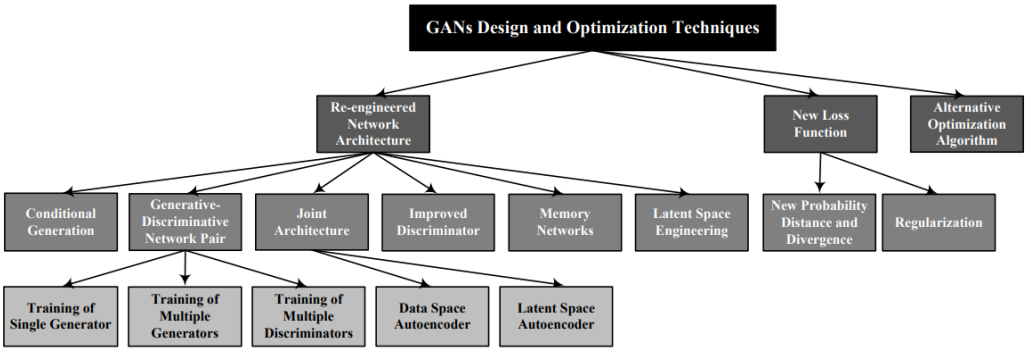


Figure 3.8: Proposed Taxonomy.

Recently, various solutions have been proposed to improve the design and optimization of basic GANs using three main techniques: reengineering network architecture, new loss functions, and employing alternative optimization algorithms. Reengineered network architectures focus on modifying the GANs' structure [46] [59] [71], new loss functions involve modified or new objective functions [49] [75], and alternative optimization algorithms include using modified or regularized methods for training GANs [49].

Chapter 4

ForGAN Applications: Simulating Time Series and Forecasting Financial Risks

The goal of the first GAN implementation for forecasting is to model the probability distribution of the next value in a time series, x_{t+1}, using a conditional GAN (cGAN). In this case, historical data until time t serves as the condition, $c = (x_0, x_1, \ldots, x_t)$, and we aim to model $p(x_{t+1}|c)$.

Both the generator and the discriminator take the historical data as input (the condition c). The generator, in addition, takes a random noise vector-sampled from a Gaussian distribution with mean zero and standard deviation one-and returns a forecast for x_{t+1}, based on the condition-historical data c.

The discriminator then evaluates whether this generated value is plausible as the next time step in the series. It does so by distinguishing between the real data and the generated data, assessing whether x_{t+1} follows logically from the historical sequence c.

Therefore the value function in this case takes the following form:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x_{t+1} \sim p_{data}(x_{t+1})}[\log D(x_{t+1}|c)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z|c)))]. \tag{4.1}$$

By training this model, the optimal generator models the full probability distribution of x_{t+1} for a given condition window. With having the full probability distribution in hand, we can extract information regarding any possible outcome and the probability of their occurrence by sampling.

Regarding the architecture, we use RNN for both discriminator and generator. We have described the RNNs, their architecture, and their forms in Section 2.1. The generator takes the condition window and passes it through an RNN layer to construct its representation. Then it connects the representation with the noise

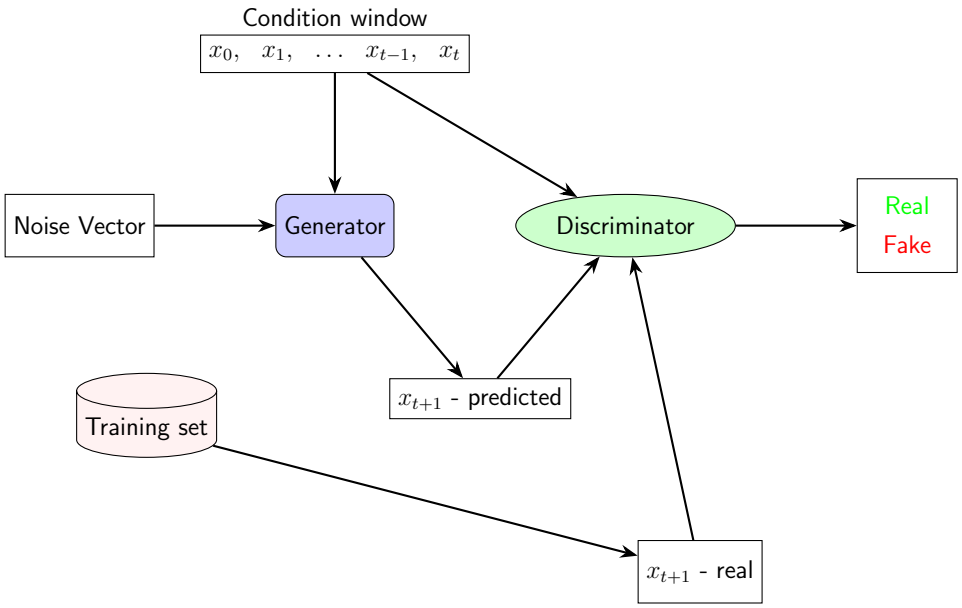


Figure 4.1: ForGan architecture where the condition c is handed to the generator G and discriminator D.

and passes them through two dense layers which gives us the prediction x_{t+1}. Then the discriminator takes the value x_{t+1} which to his view can either originate from the generator (fake data) or from the real data together with the condition window and chains the x_{t+1} at the end of the window to get $x_0, ..., x_t, x_{t+1}$, the next part of the network evaluates this time window as real or fake. Therefore, it passes the window through an RNN layer followed by a dense layer in order to obtain the evaluation value. The main components of ForGAN are:

The Generator has two key components:

- *Condition-to-Latent Layer (LSTM or GRU)*: This component uses either a Long Short-Term Memory (LSTM) neural network or a Gated Recurrent Unit (GRU) to process the conditional input. Both LSTM and GRU layers are capable of capturing sequential dependencies and patterns in the input data.
- *Model (Sequential Neural Network)* It processes the conditional input, and the model further expresses it into a synthetic data sample. It consists of three layers:

  Linear Layer: A fully connected layer that combines the processed conditional input with noise vectors, learning to map them to a representation.

  ReLU Activation: This activation function introduces non-linearity to the network, allowing it to capture complex patterns in the data.

  Linear Layer: Another fully connected layer that transforms the model into the output space, generating the synthetic data sample.

The Discriminator has two key components:

- *Input-to-Latent Layer (LSTM or GRU)*: Similar to the generator, the discriminator uses either an LSTM or GRU layer to process input data samples and find relevant features.
- *Model (Sequential Neural Network)*: This model evaluates the authenticity of the input data sample by converting the latent representation into a single probability score. It consists of two layers:

  Linear Layer: A fully connected layer that maps the latent representation to a single output, indicating the probability that the input sample is from the actual data.

  Sigmoid Activation: This is applied to the output of the linear layer; this activation function compresses the score between 0 and 1, representing the likelihood of the input being real.

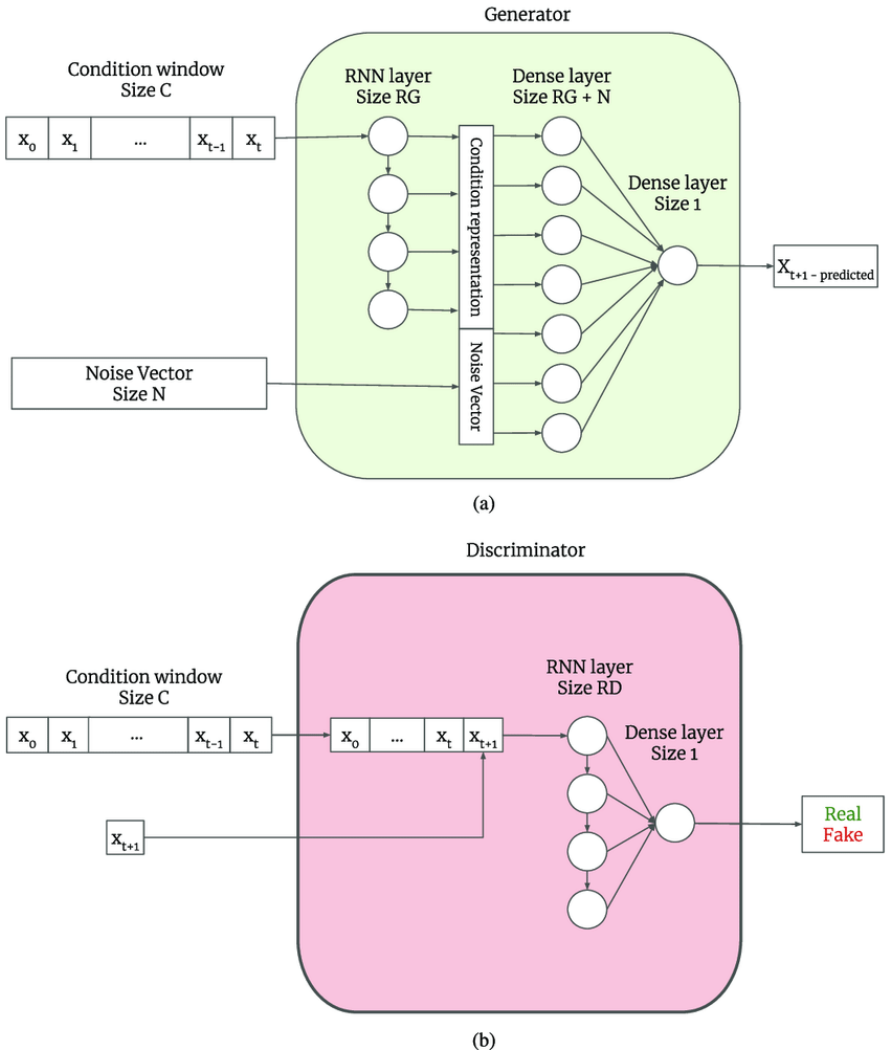


Figure 4.2: (a): The architecture of the generator. (b): The architecture of the discriminator.

As a data set we are using intraday electricity consumption data for every 15-minute intervals throughout January 2015 in Germany, shown in Figure 4.3. We can see the seasonality in the electricity consumption, we have regular daily patterns with lower consumption over the weekends. The data is fairly consistent, with repeating peaks and valleys, suggesting predictable energy use. Daily fluctuations show higher consumption during the day and lower at night. All the electric consumption data sets in the following analysis are taken from Smard[1].

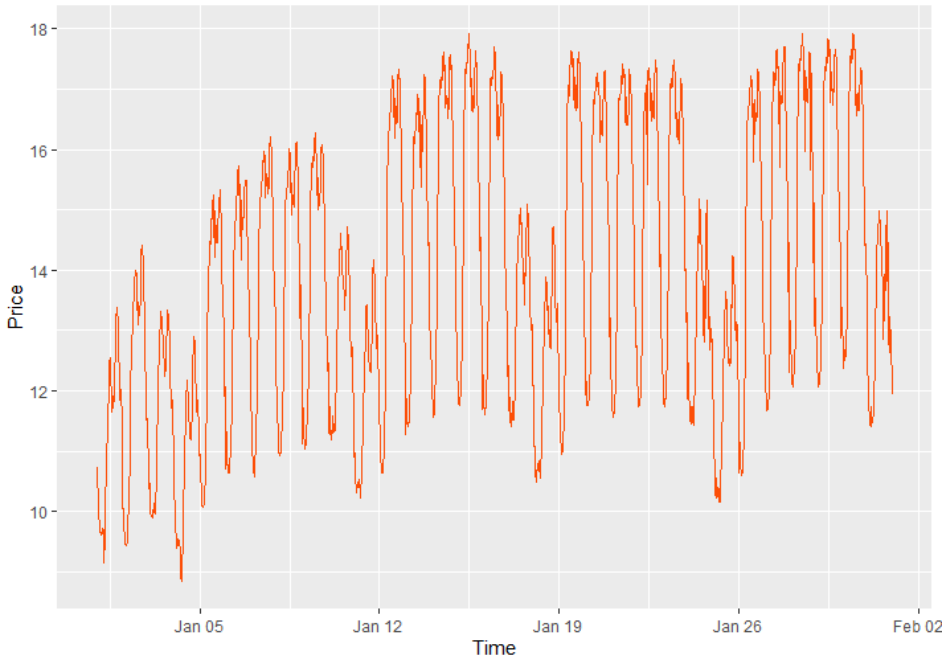


Figure 4.3: One-month electricity consumption for 15-minute periods.

Since we are interested in measuring how accurately we predicted the data distribution of x_{t+1} as an evaluation metric we use the Kullback-Leibler divergence (KLD). The KLD divergence measures the divergence between two distributions. Due to the fact that our data samples are finite, we use the following version of KLD:

$$KL(P|Q) = \sum_i P_i log \frac{P_i}{Q_i}. \tag{4.2}$$

P is the distribution of our data and Q is the predicted probability distribution. It's important to note that since in equation 4.2 Q is present in the denominator, in cases when the predicted distribution does not cover the data distribution accurately, then KLD is not defined. To compute the KLD, we sample 100 forecasts of x_{t+1} for any condition in the test set. Then we create ForGAN's predicted probability distribution for the entire test set and compute the KLD between this predicted distribution and the true test set data distribution.

[1]https://www.smard.de/home

In addition to KLD, to have complete information regarding the performance of our model we calculate the point-wise error metrics, by running the model 100 times over the test producing 100 generated paths for the test data. This helps us evaluate the point-wise performance of our model. The KLD value shows how accurately our method has learned the data distribution, while the point-wise error metrics specify how well it has considered the condition to forecast x_{t+1}.

The data is split with 50% for training, 40% for testing, and 10% for validation. Since our data is recorded every 15 minutes, we have 2976 data points for a month. We have tuned the following hyperparameters in our model:

- We used GRU cells for the RNN.
- The size of the noise vector is set to 32 and the condition size to 96.
- The number of cells in the discriminator and generator are 64 and 8, respectively.
- The number of training iterations for the discriminator is 2.
- We used 100 training steps.

It's important to note that increasing the size of the condition window may lead to memory issues. Regarding the loss function, we employed the non-saturating GAN loss. We do not perform any pre-processing of the data, as the data are quite regular, as seen from Figure 4.3. And as we can see from Figure 4.6, we managed to get good forecasting results.

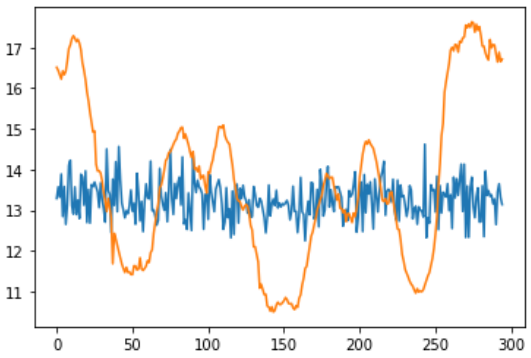


Figure 4.4: The validation set and generated set on the first step.

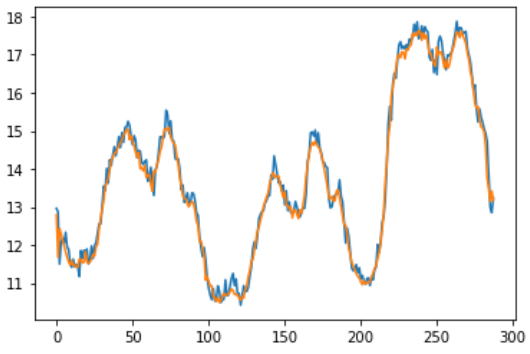


Figure 4.5: The validation set and generated set on the last step.

Table 4.1 shows us the performance of the model for generating the forecasts for the test set. The performance metrics, including RMSE, MAE, and MAPE, were calculated based on the error between the predicted and actual values. This was done by running the model 100 times over the test set, averaging the values for each run, and then computing the mean and standard deviation of these values (as we can see two different values in the table). Additionally, the Continuous Ranked Probability Score (CRPS) is calculated as the mean absolute difference between the predicted cumulative distribution and the test data cumulative distribution,

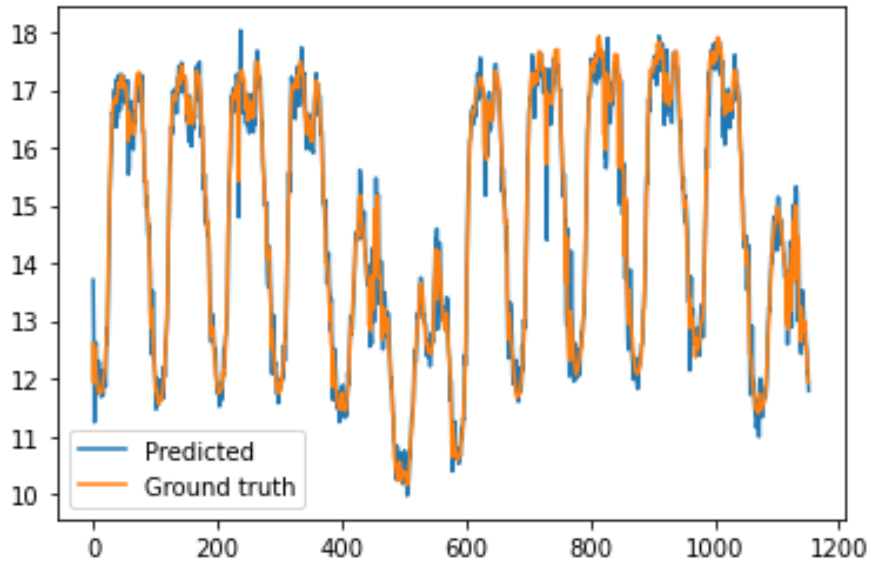


Figure 4.6: Probability distribution of one step ahead forecast learned from ForGAN, one generated path.

Metric	Mean	Standard Deviation
RMSE	0.38545	0.00498
MAE	0.26130	0.00381
MAPE	1.81169	0.02698
CRPS	0.16240	-
KLD	0.14105	-

Table 4.1: Performance of the ForGAN model across different metrics for the test set, showing the mean and standard deviation (where applicable) based on 100 runs.

adjusted by a factor to account for the spread of the predictions. The Kullback-Leibler Divergence (KLD) provides a measure of how well the model has captured the distribution of the actual data. Since both CRPS and KLD measure the overall quality of the predicted distributions, no standard deviation is reported for these metrics.

From the results we can see that ForGAN is able to model the full probability distribution of x_{t+1} for a given time window condition to a certain extent. However, it does not do so perfectly, indicating that there is still room for improvement in its ability to represent the true distribution accurately. We can change the tuned hyperparameters and see how they affect the results. But before making adjustments, we will examine the loss function, as described in detail in Section 3.1.

Figures 4.7 and 4.8, illustrate the loss during the training process of ForGAN. With

Discriminator loss, we refer to the loss that is obtained when the discriminator classifies fake (generated) data as accurate, while *Generator loss* is obtained when the discriminator classifies the generated data as fake. We train both networks, generator, and discriminator, using the same loss function - Binary Cross-Entropy (BCE).

In this implementation, we use the non-saturating GAN loss, which is a subtle variation of the standard loss function. In this approach generator aims to increase the probability that its generated data are classified as real, rather than decreasing the probability that they are classified as fake. What happens here is that the discriminator seeks to maximize the average of the log probability of actual data and the log of the inverse probability for fake data, i.e., $\log D(x) + log(1 – D(G(z)))$. While the generator seeks to maximize the probability of data being predicted as accurate, i.e., maximize $log(D(G(z)))$. To obtain the min-max game, we multiply the generator's BCE loss with -1. Therefore, from the Figure we can say that the losses in both cases are erratic early in the run, and they start stabilizing after the 20 steps (one step means one gradient update). However, the losses are not sufficiently stable and still exhibit fluctuations.

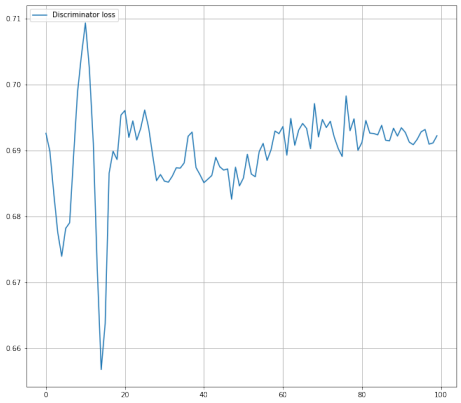


Figure 4.7: Discriminator's loss during the the training process.

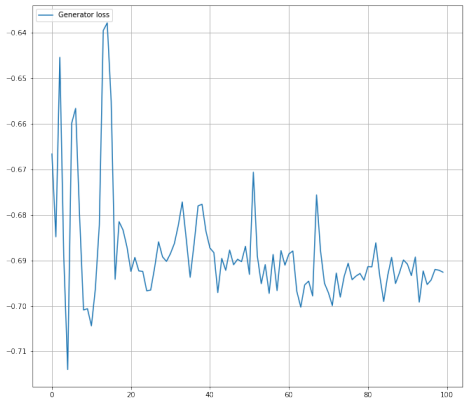


Figure 4.8: Generator's loss during the training process.

When we increase the number of iterations from 100 to 1000 this improves the model's performance, and we get a more stable loss. We can see from Table 4.2 that the error values drop when increasing the number of iterations compared to the previous results in Table 4.1. Moreover, during the validation process the last generated path follows the actual one very closely, see Figures 4.9 and 4.10.

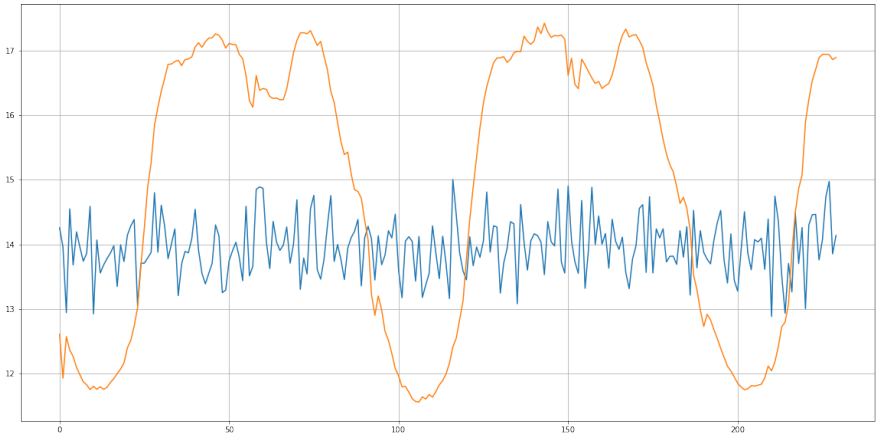


Figure 4.9: The validation set and generated set on the first step.

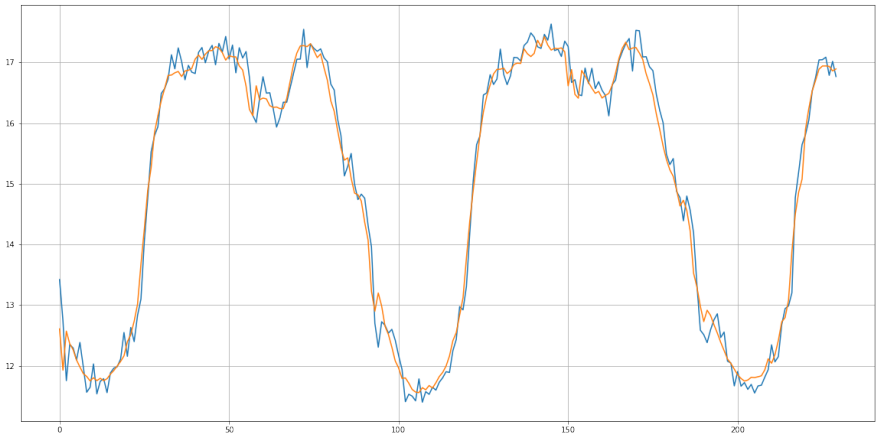


Figure 4.10: The validation set and generated set on the last step.

Metric	Mean	Standard Deviation
RMSE	0.35727	0.00496
MAE	0.24050	0.00396
MAPE	1.67336	0.02830
CRPS	0.15007	-
KLD	0.28126	-

Table 4.2: Performance metrics for ForGAN after 1000 training iterations.

The performance of ForGAN improves noticeably when increasing the training iterations to 1000 steps. As illustrated in Figures 4.11 and 4.12, the results show improved accuracy. The distribution in Figure 4.11 indicates that ForGAN successfully captures the underlying distribution of the one-step-ahead forecasts, while Figure 4.12 demonstrates how closely the forecasted paths align with the actual data. These validation plots confirm that the model performs better with extended training.

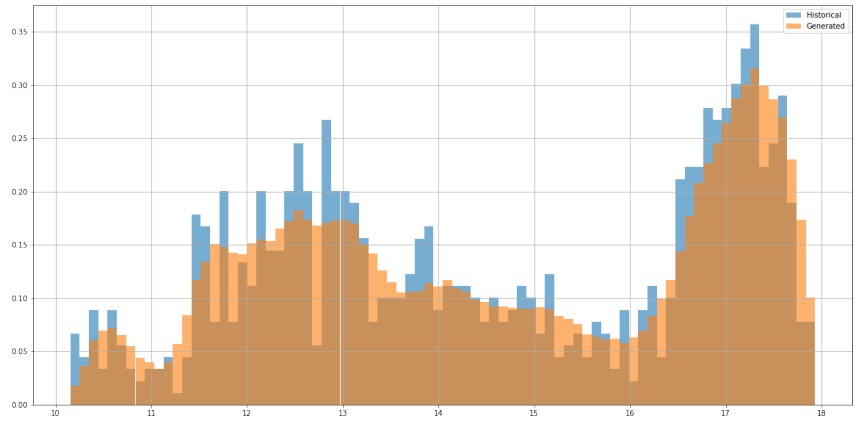


Figure 4.11: Probability distribution of one step ahead forecast learned from ForGAN with 1000 training iterations.

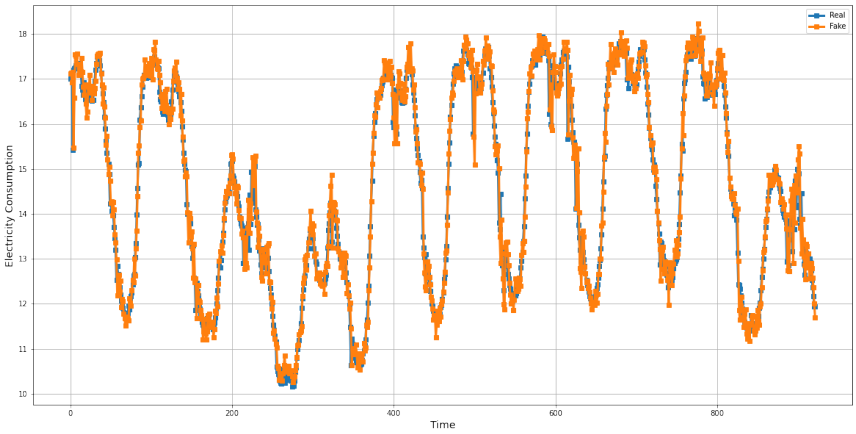


Figure 4.12: Mean of five forecasted paths versus actual one using ForGAN with 1000 training iterations.

Next, we plot the discriminator's and generator's loss during the training process. We have generated good results, but Figure 4.14 shows the Generator's loss during

the training process it is not quite stable. Both losses stabilize after initial fluctuations, indicating the GAN is training correctly with the generator improving over time despite occasional spikes.

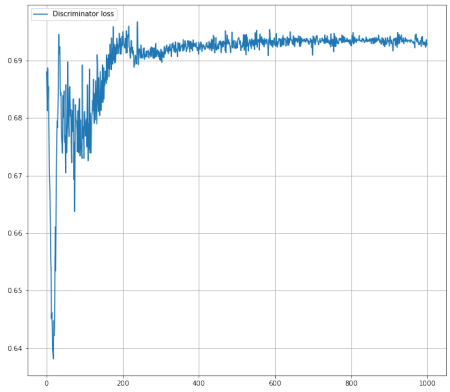


Figure 4.13: Discriminator's loss during the training process.

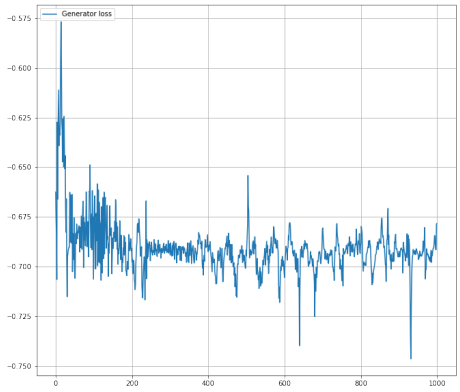


Figure 4.14: Generator's loss during the training process.

In the previous examples we have used the non-saturating GAN loss which focuses on maximizing the probability that the generated samples are classified as real. Now, we switch to the standard GAN loss. In this setup, the discriminator aims to maximize the probability of correctly classifying real samples as real and generated samples as fake. On the other hand, the generator aims to minimize the probability that the discriminator correctly classifies generated samples as fake.

However, despite using the standard GAN loss, we observe that this does not fully stabilize the generator's loss. As shown in Figure 4.15, the discriminator's loss remains relatively stable, but the generator's loss is more erratic, indicating that the training process is not fully optimized.

This suggests that the current 1000 training iterations may not be sufficient for full convergence. A logical next step is to increase the number of training iterations to 5000 to assess whether this improves stability and the overall performance of the model.

Metric	Mean	Standard Deviation
RMSE	0.32965	0.00421
MAE	0.03485	0.00418
MAPE	0.16335	0.02917
CRPS	0.03439	-
KLD	0.12932	-

Table 4.3: Performance metrics for ForGAN after 5000 training iterations.

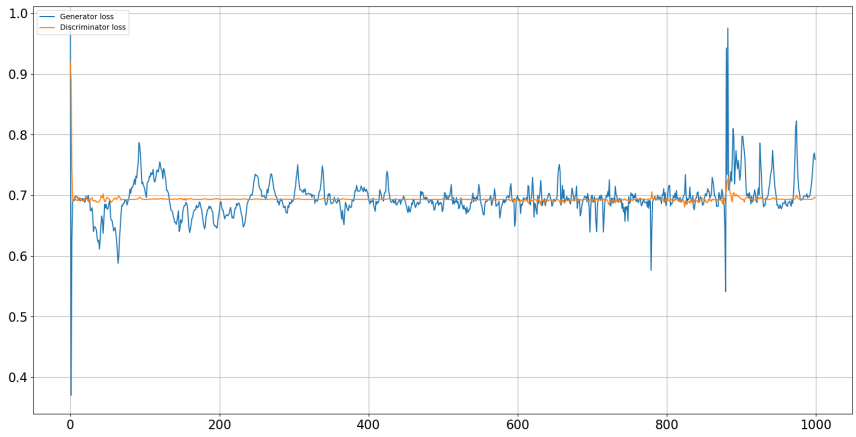


Figure 4.15: Disciminator's and Generator's loss during the training process with 1000 training iterations.

Figure 4.16 shows that with additional training steps, we achieve stable loss for both Discriminator and Generator. Moreover, when we plot the histogram of the probability distribution for the actual and forecasted test data, using 5000 training steps we see that the generated data distribution does follow the historical one quite closely, Figure 4.17.

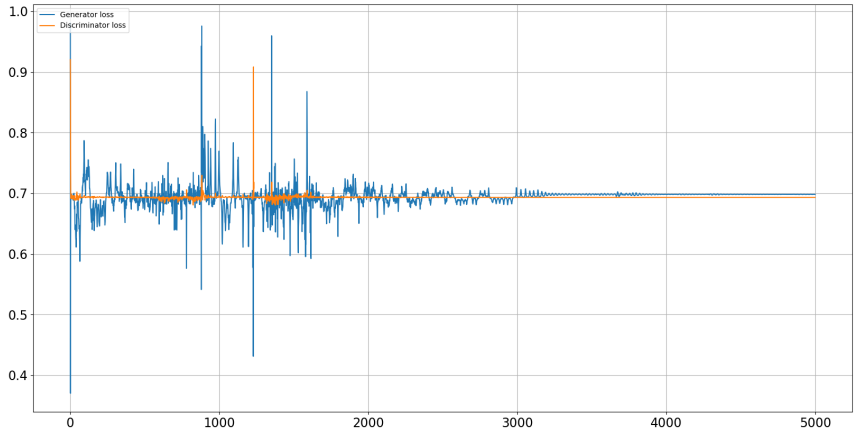


Figure 4.16: Disciminator's and Generator's loss during the training process with 5000 training iterations.

Also, Tables 4.2 and 4.3 confirm that the performance metrics show an improvement in the model's accuracy after increasing the number of training iterations from 1000 to 5000. Specifically, the values for RMSE, MAE, MAPE, and KLD are lower in the second table, indicating that with 5000 training iterations, For-GAN produces more accurate forecasts and better captures the underlying data

distribution. This is further reflected in the reduced KLD value, showing that the predicted probability distribution aligns more closely with the true data distribution after 5000 iterations compared to 1000 iterations.

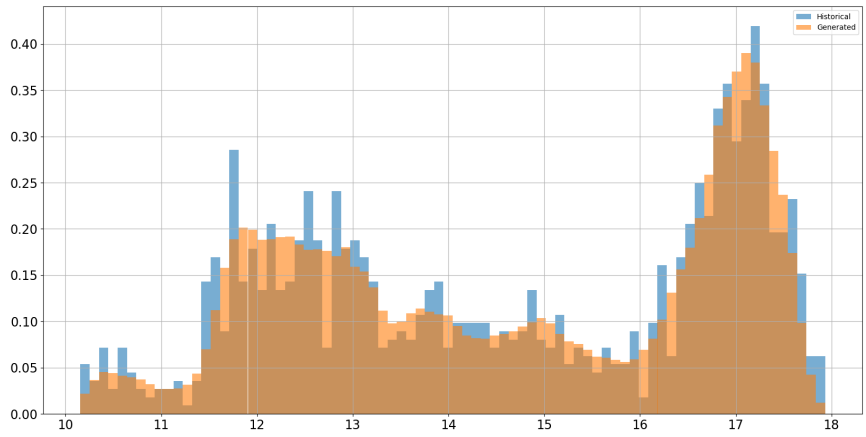


Figure 4.17: Probability distribution of one step ahead forecast learned from For-GAN 5000 training iterations.

Next, we conduct additional experiments on the same data set. First, we increase the condition size to **672** and check the model's performance. The first plots 4.18, 4.19 show the generated paths versus the actual data paths during the validation process. On the first iteration, the generated one does not follow the real data pattern at all. Then after 100 iterations on the validation set, the generated path aligns well with the real data.

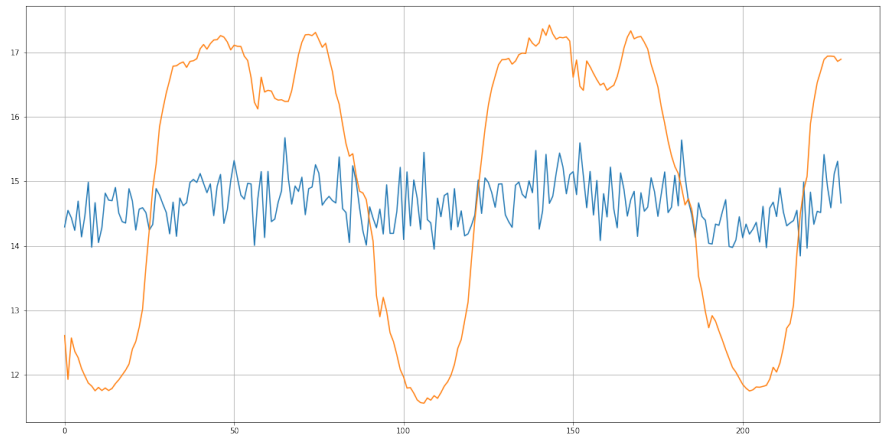


Figure 4.18: The validation set and generated set on the first step.

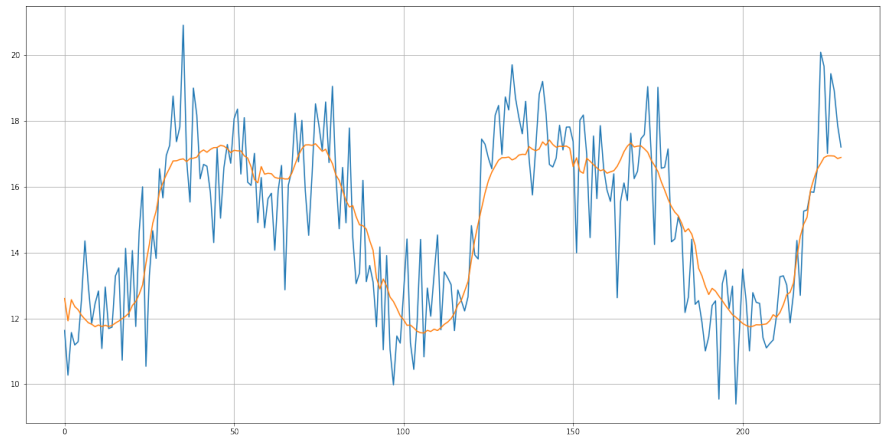


Figure 4.19: The validation set and generated set on the last step.

Moreover, to calculate the KLD, we sampled 200 forecasts of x_{t+1}. Subsequently, we derived the predicted probability distribution for the test set and calculated performance metrics, including the KLD, to quantify the disparity between the generated and actual data distributions, as described previously. The results are summarized in Table 4.4.

In addition, we plot the histogram of the probability distribution for the actual and forecasted test data shown in Figure 4.20. We can see that the generated data distribution does not closely match the historical one, i.e., it is not able to capture the distribution pattern. This tells us that increasing the condition size that we feed to the generator and discriminator does not necessarily mean that we get better results.

Metric	Mean	Standard Deviation
RMSE	1.35597	0.03045
MAE	1.07908	0.02566
MAPE (%)	7.61108	0.18164
CRPS	0.40454	-
KLD	2.41200	-

Table 4.4: Performance metrics for ForGAN with increased condition size to **672** during training.

Another evaluation plot of our model is presented below in Figure 4.21. We calculate the mean of five generated paths and plot them against the actual historical data. In this case, the generated paths follow the pattern of the real one quite well but with some deviations.

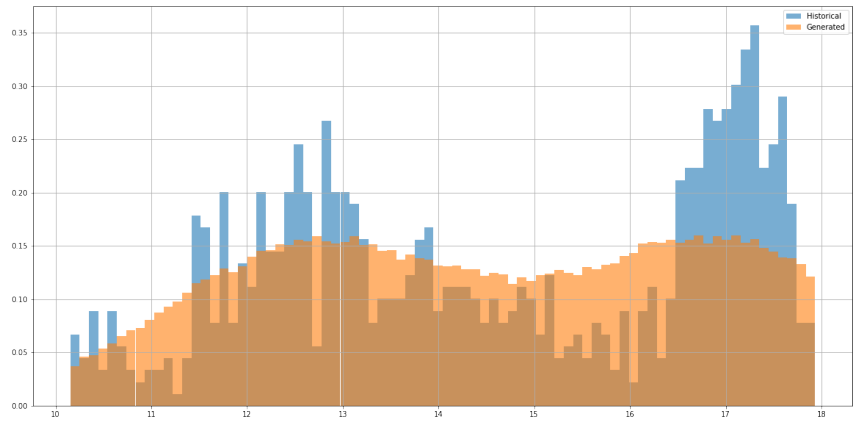


Figure 4.20: Probability distribution of one step ahead forecast learned from ForGAN with increased condition size to **672** during training.

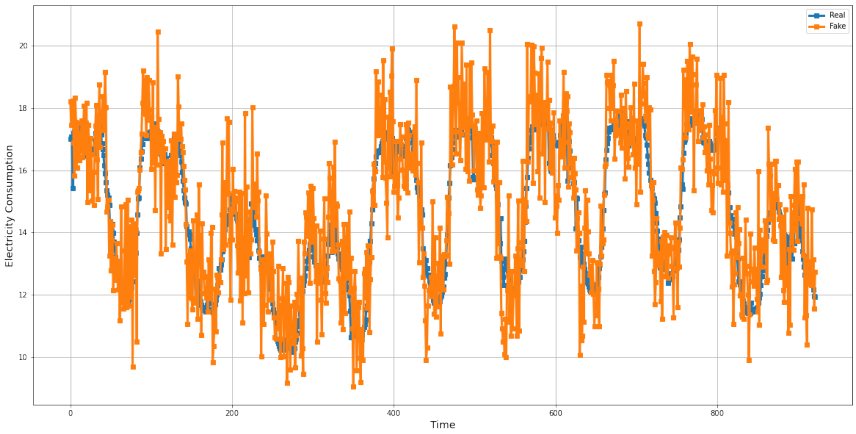


Figure 4.21: Mean of five forecasted paths versus actual one using ForGAN with increased condition size to **672** during training.

When increasing the condition size very large, the model's performance drops, and memory issues may arise. Since there is no general rule for selecting hyperparameters, trial and error is one of the possible approaches, along with systematic methods like grid search, and random search.

Further, we use the three years of 15-minute intraday electricity consumption in Germany, spanning from 01.01.2015 – 31.12.2018. Our experiments demonstrate that increasing the number of data points improves the results, even though at the cost of additional time. The dataset is quite regular, as shown in Figure 4.22. Specifically, the data displays clear *seasonal patterns*, with consistent daily cycles of electricity usage. For instance, consumption peaks during daytime hours and

declines during nighttime, reflecting typical human activity and business operations. Additionally, there is noticeable *weekly seasonality*, characterized by lower consumption levels on weekends compared to weekdays. These regular patterns are crucial for time-series modeling as they allow the model to capture underlying trends and make more accurate forecasts. The regularity and seasonality in the data provide a structured framework that ForGAN can leverage to learn and predict future consumption effectively.

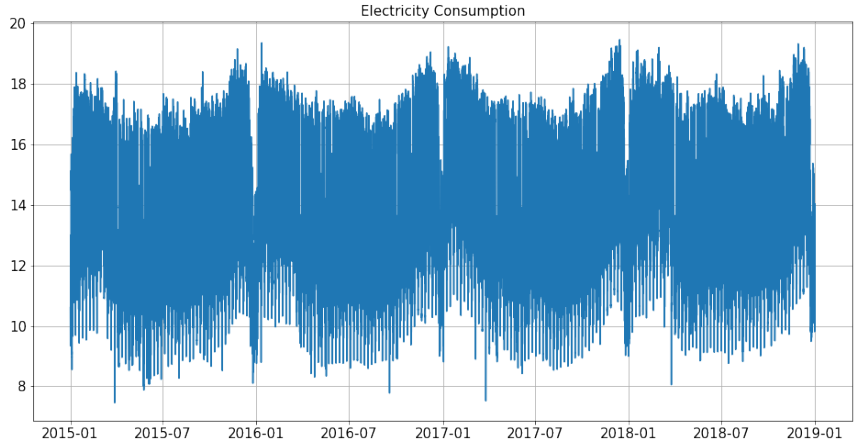


Figure 4.22: Three years of electricity consumption for 15-minute periods.

Moreover, to see the correlation and possible outliers for our data set, we draw the following box plots and the so-called *heat-map*. Each box in the heat map illustrates the correlation value between the variables on both axes, with the color intensity corresponding to the strength of the correlation, as indicated by the color bar on the right side of the graph.

The box plots, shown in Figure 4.23, highlight the variation in electricity consumption across different time intervals. It is evident from these plots that electricity consumption decreases significantly during weekends, and gradually increases over the three-year period. Additionally, there is a slight rise in consumption during the winter months, suggesting higher energy usage for heating purposes.

Another concern regarding our data is removing the outliers. Based on the previous graphs, it looks like our data do not contain outliers. To confirm this, we use the *Inter quartile range (IQR)* method, which indicates that there are indeed no outliers present. Additionally, the box plot shown in Figure 4.24 provides further confirmation that the dataset does not contain any significant outliers.

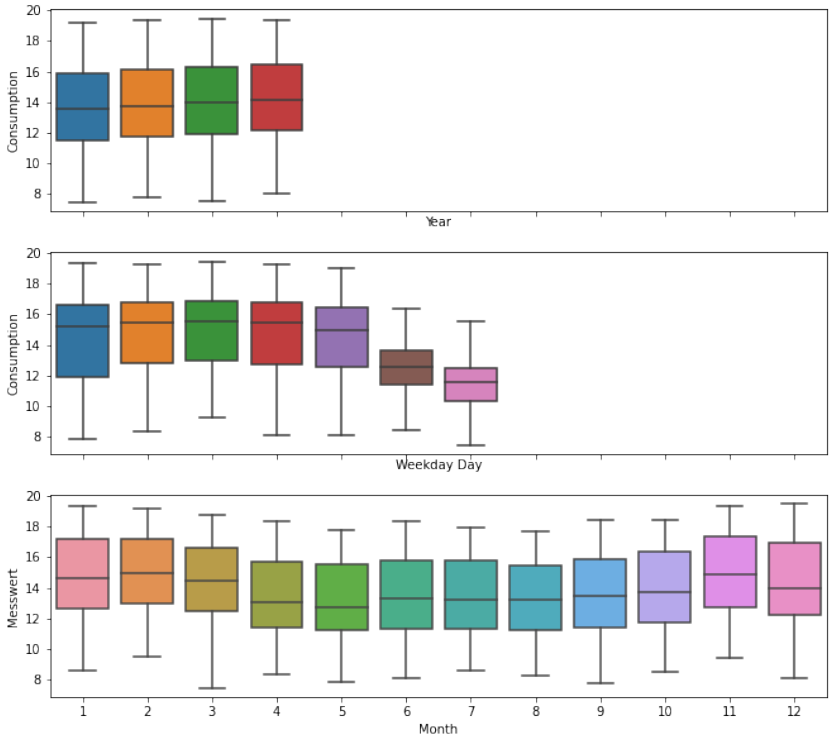


Figure 4.23: Box plots regarding weeks, months, and years for 15-minute electricity consumption.

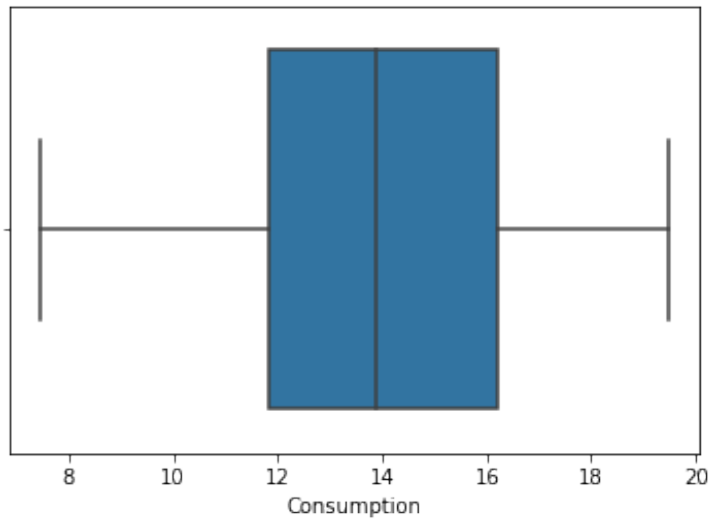


Figure 4.24: Box plot of the complete dataset, confirming no significant outliers.

Furthermore, to explore the correlation between the months of the year and the days of the week, as mentioned above we use the heat map, see Figure 4.25. The heatmap reveals strong correlations in electricity consumption based on seasonal changes and weekly cycles. These correlations are consistent across the three-year period, indicating predictable and regular usage patterns. The absence of significant outliers in the heatmap further suggests stable and reliable data trends. We can see that in general there is a lower correlation between the weekends compared to weekdays across different months. However, this pattern is not uniform, as some months (particularly those with extreme weather conditions) show higher overall consumption that slightly weaken the drop typically seen on weekends. This indicates that while weekends generally have lower electricity usage, the amount of this reduction can vary based on the month.

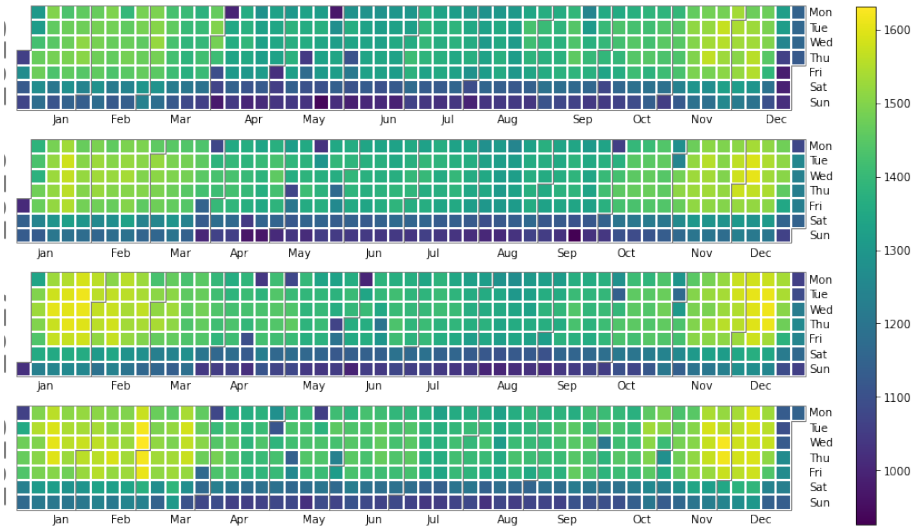


Figure 4.25: Heatmap showing correlations between days of the week and months of the year for electricity consumption.

A **data transformation** does not improve our model in this case. We plugged the *scaler* as a hyper-parameter in our model, and it turns out that scaling the data does not improve the prediction accuracy.

The condition window (look-back window) should be smaller than 200 because it causes memory problems. However, the results are pretty good with a condition size of 64. We get a value of Kullback Leibler divergence equal to 0.007154, which tells us that the two distributions are quite close to each other. As mentioned, to calculate the KLD value, we generate 200 paths of x_{t+1} to obtain the probability distribution of the generated data and compare it with the real ones.

In total, we have 140257 data points. We use 50% of them for training the model, which means the first 730 days of electricity consumption, the following 10% of the data are used for validation, i.e., around 146 days, and last the 40% of the data are used for testing.

Figure 4.26 shows us the first generated path during the validation process, and Figure 4.27 shows us the last one, which follows the actual path very accurately.

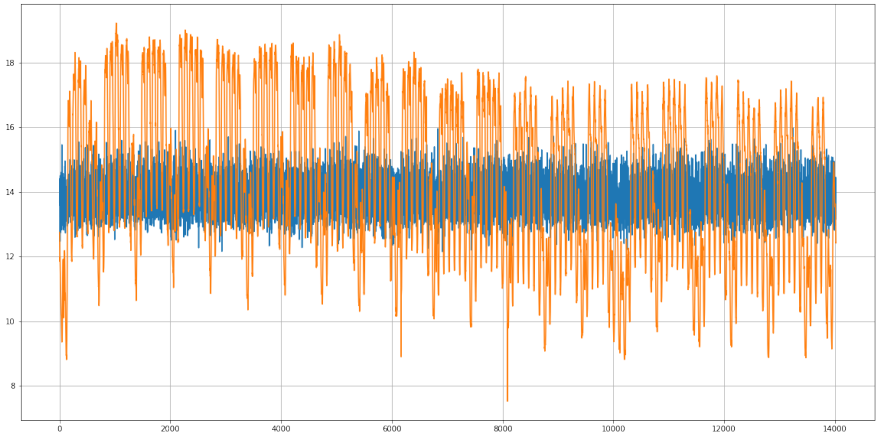


Figure 4.26: The validation set and generated set on the first step.

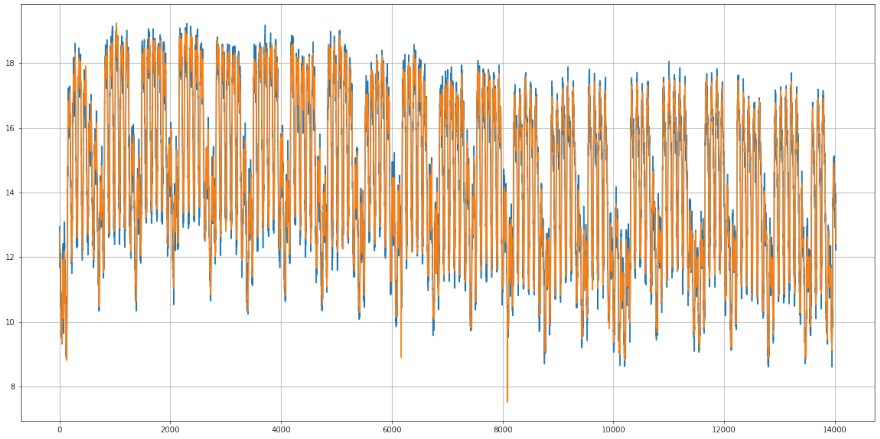


Figure 4.27: The validation set and generated set on the last step.

The following evaluation plots presented in Figure 4.28 and Figure 4.29, show a very good performance of our model. We can see that the generated data's probability distribution captures the original ones' distribution pattern.

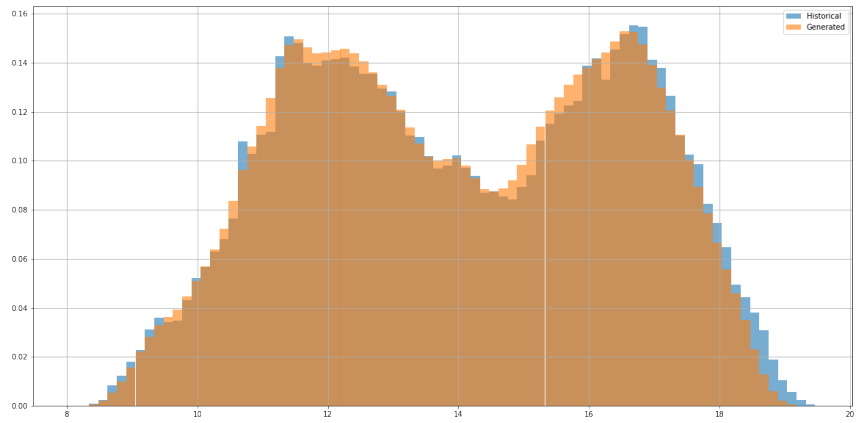


Figure 4.28: Probability distribution of one step ahead forecast learned from ForGAN.

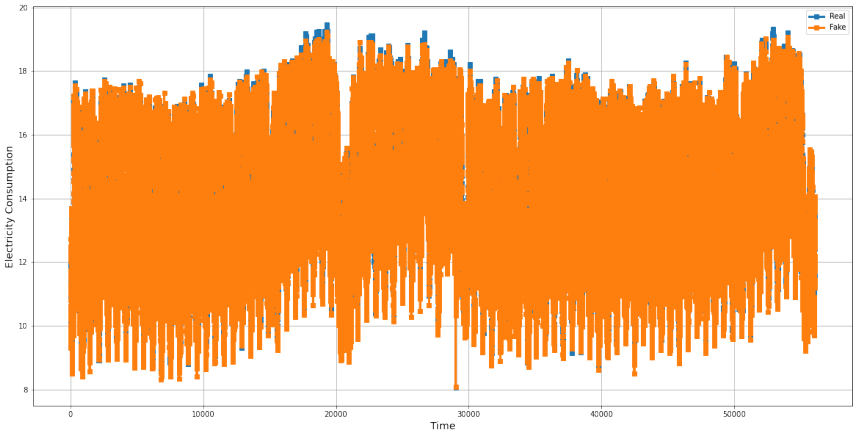


Figure 4.29: Mean of five forecasted paths versus actual one using ForGAN.

Next, we check the generator's and discriminator's loss during the training process, shown in Figures 4.30 and 4.31. We have used the non-saturating loss with 1000 training steps. After 200 iterations the discriminator's loss gets quite stable with values varying mostly between $0.69 - 0.70$. The generator's loss on the other hand is more unstable with larger fluctuations, but it stabilizes after 300 training steps.

Another good evaluation of our model is the auto-correlation plots of the generated and actual data. We use different ways to plot the auto-correlation in Python. One of them is to use the *Pandas* software library, which standardizes the data before computing the auto-correlation. This library subtracts the mean and divides it by the standard deviation of the data. To generate the following plot, 4.32, we choose random samples of 400 data points from our test and generated data since if we

plot the complete test data set, we have too many data points, 56096 in total, and this gives a very fuzzy plot.

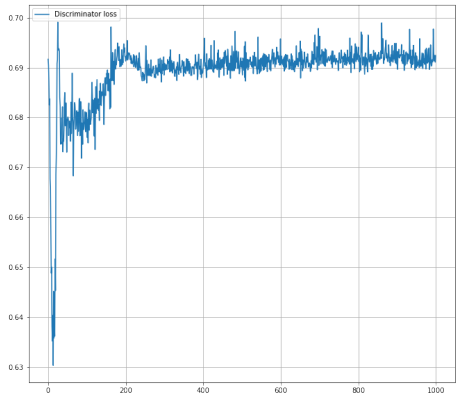


Figure 4.30: Discriminator's loss during the the training process.

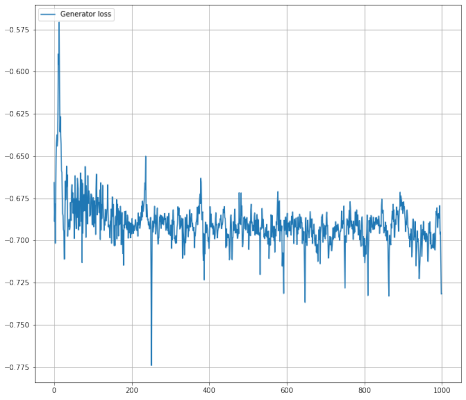


Figure 4.31: Generator's loss during the training process.

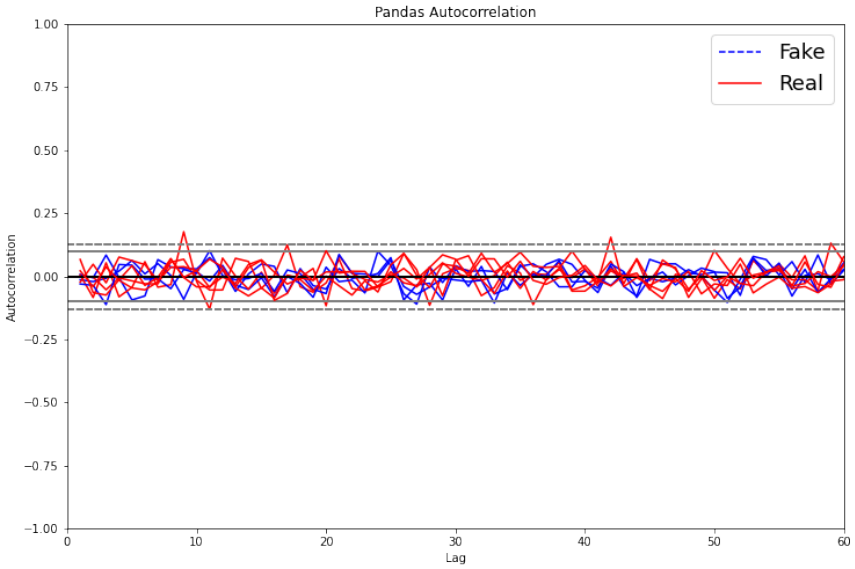


Figure 4.32: Auto-correlation plot of the real and generated data of five random paths.

We repeat this for 5 paths and get the pandas auto-correlation graph, which shows us that there are no significant lags on the data, only lag 8 and 41 may be slightly significant for the test data set.

A different module uses almost the same way as previously described for creating the ACF plot in Python, named *statsmodels*. We sampled a path of the generated and actual data of 400 data points and plotted their auto-correlations as presented below, Figure 4.33 and 4.34 . Also, this graph shows us that there are no compelling lags in our data set, meaning that the previous consumption does not influence the current one extensively, reflecting the stable and regular nature of the dataset.

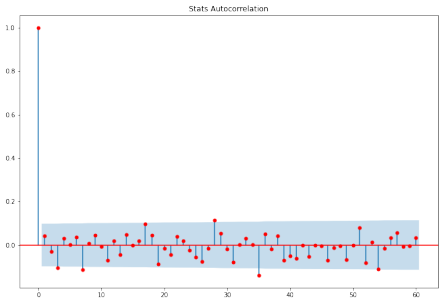


Figure 4.33: Auto-correlation plot of the real data.

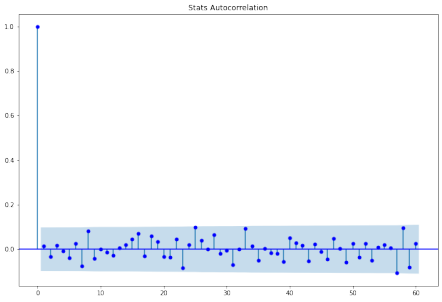


Figure 4.34: Auto-correlation plot of the generated data.

Another informative analysis is the cross-correlation plot shown in Figure 4.35. We sample 400 data points from the true and generated data, repeat it for 10 random paths for both data sets, and check the cross-correlation between those paths after subtracting their mean from each data point.

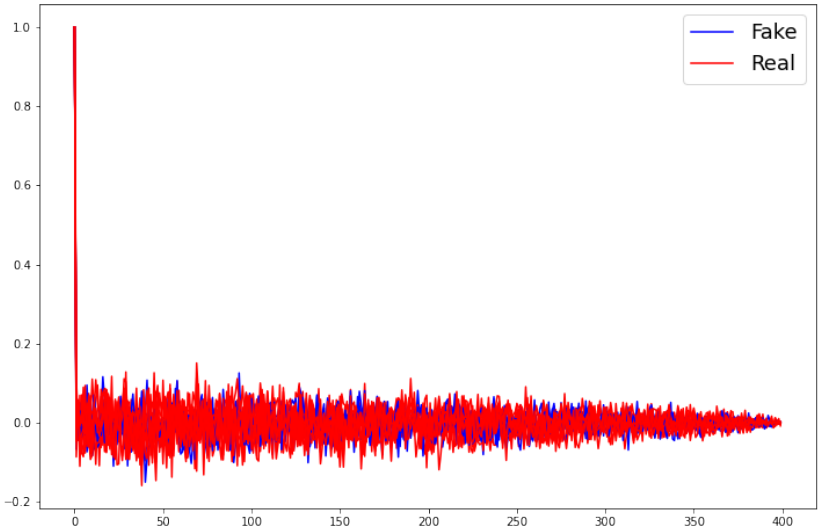


Figure 4.35: Cross-correlation plot of the real and generated data for ten random paths.

We can see that the generated data closely follows the mean and variance of the real data throughout the time steps, as well as that the real and generated data are highly correlated at the start, which slightly diminishes over time.

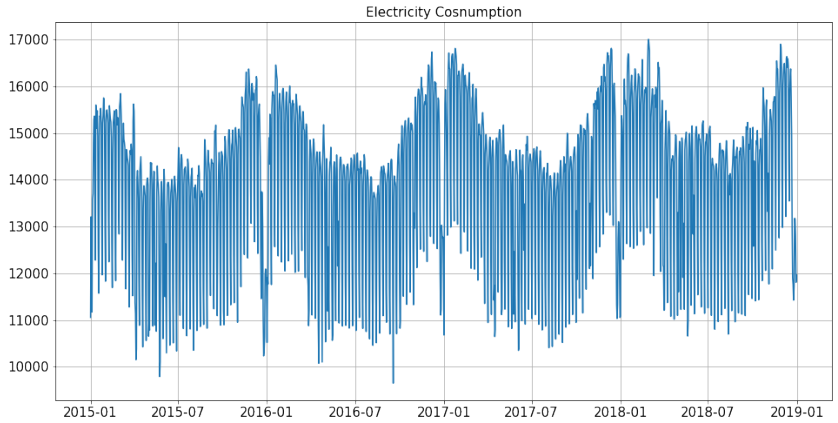


Figure 4.36: Three years of daily electricity consumption.

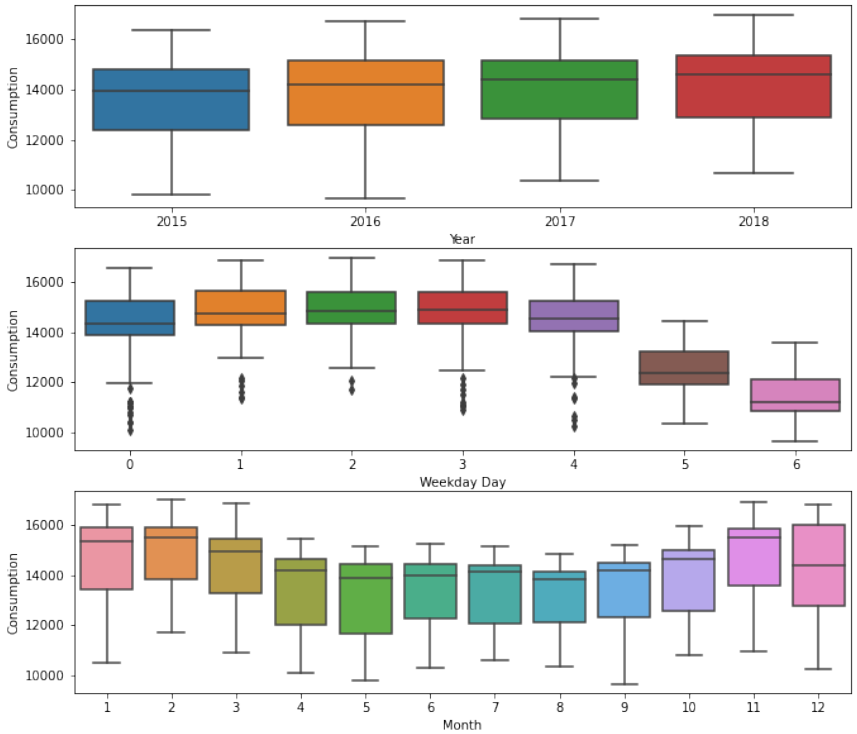


Figure 4.37: Box plots illustrating the distribution of daily electricity consumption across different weeks, months, and years.

Next, we aggregate the dataset into daily electricity consumption. From 96 points a day, we obtain one data point per day by calculating their mean value. Our model performs better with more data points, as the Discriminator has a larger data set to learn from. Although we still achieve a low KL-divergence value, other plots, such as the probability distribution and mean of generated paths (see Figures 4.40, 4.41), show that the forecasting is not as good as the one from the not aggregated data set, even though we use more training steps (iterations). The loss of finer details due to aggregation makes it harder for the model to capture shorter-term variations, which were more evident in the original data set. However, the overall seasonality and trend patterns are still preserved, and we observe good performance in terms of the model's ability to capture the general daily consumption trend. The ForGAN model is still able to model broad seasonal patterns and weekday-weekend variations effectively, indicating that the aggregation does not completely obscure important trends in the data. Nevertheless, we are still obtaining solid results.

Figure 4.36 shows the daily aggregated electricity consumption in Germany for the period from January 1, 2015, to December 31, 2018. We observe that the seasonality patterns in the daily aggregated electricity consumption data remain consistent with those observed in the un-aggregated data. Despite the aggregation, we can still see clear cycles of higher consumption during the winter months and lower consumption during the summer. Additionally, the weekday versus weekend consumption differences, which were prominent in the un-aggregated data, persist in this aggregated view, indicating stable and predictable trends across the three-year period.

When examining the box-plot of our data for weeks, months, and years, (see Figure 4.37), it is evident that electricity consumption drops during weekends and slightly increases over the years.

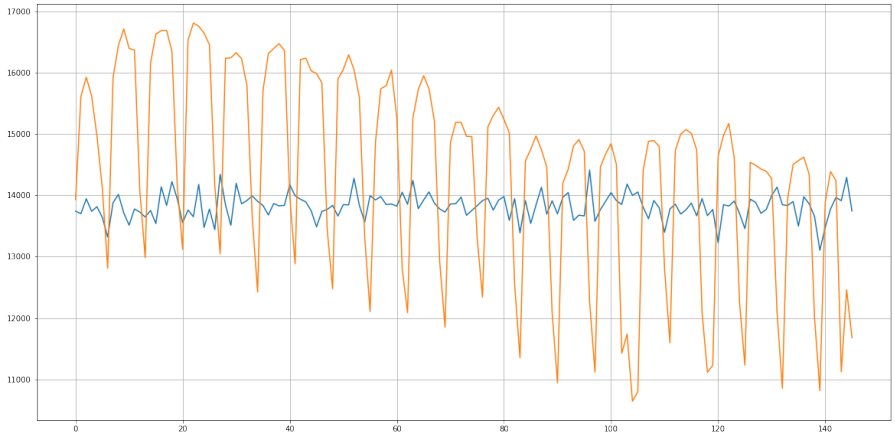


Figure 4.38: The validation set and generated set on the first step.

While there are no significant outliers on a yearly and monthly basis, some out-

liers can be observed for the days of the week. Moreover, monthly variations indicate higher consumption during colder months (January and December) and lower during summer months (July and August).

In the same way, as previously we do not do any pre-processing of the data. We use 50% of them for training the model, the following 10% of the data are used for validation, and last the 40% of the data are used for testing. Figure 4.38 shows us the first generated path during the validation process, and Figure 4.39 shows us the last one, which follows the actual data relatively closely.

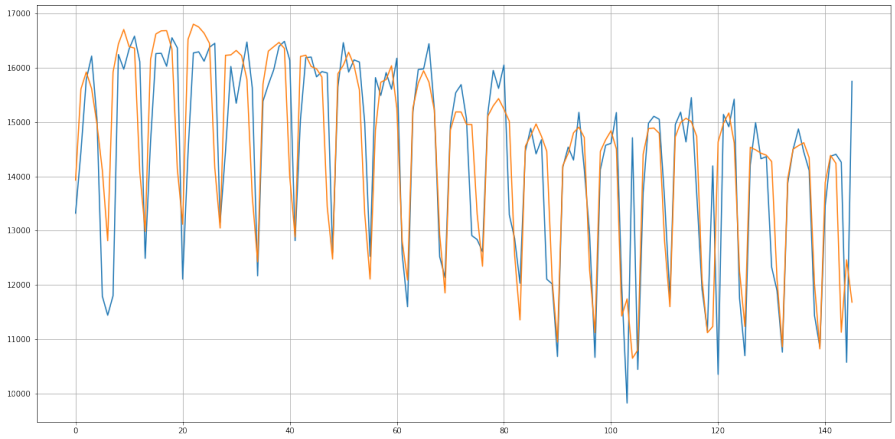


Figure 4.39: The validation set and generated set on the last step.

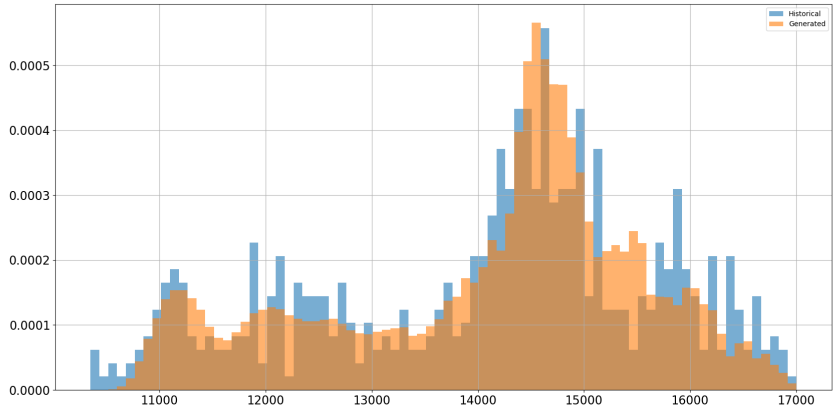


Figure 4.40: Probability distribution of one-step-ahead forecast generated by For-GAN for daily aggregated electricity consumption data.

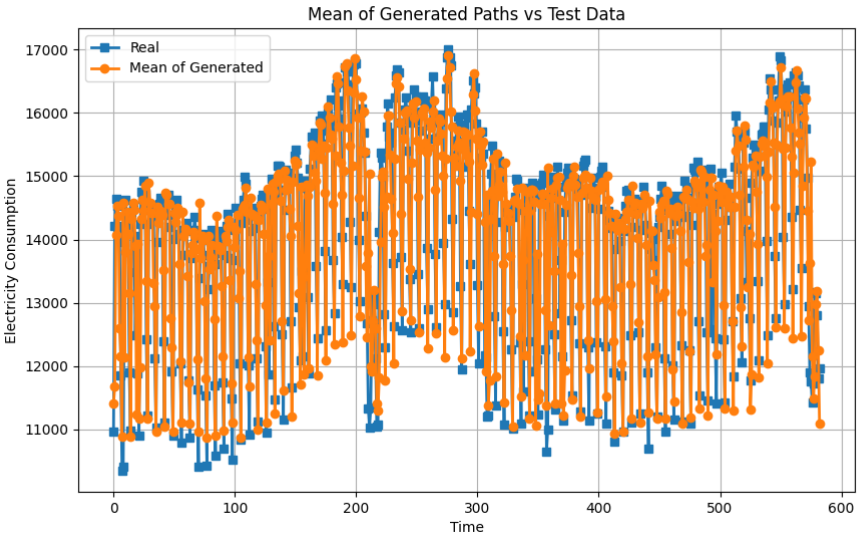


Figure 4.41: Mean of fifty forecasted paths compared to the actual path using ForGAN for daily aggregated electricity consumption data.

Figure 4.42 illustrates that both the Discriminator's and Generator's loss functions exhibit initial fluctuations. However, after approximately 2000 iterations, these fluctuations stabilize, indicating that both models are learning effectively and have reached a more stable state of training.

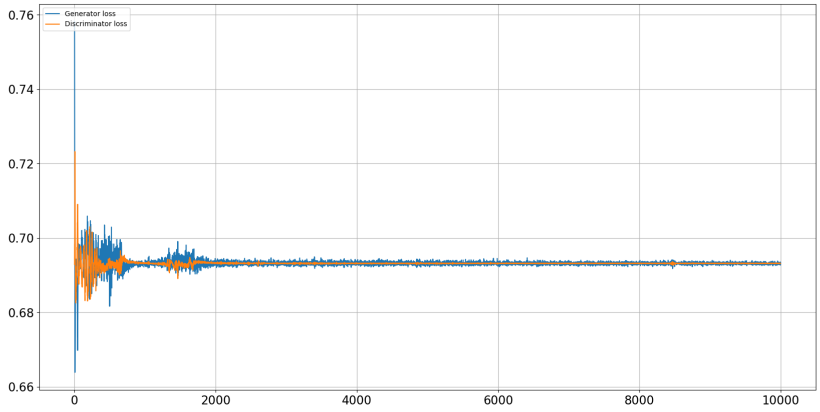


Figure 4.42: Generator's and Discriminator's loss during the training process.

In the same manner as before we are going to look at the auto-correlation plots for true and generated data. We can spot a stronger correlation and more significant lags than for unaggregated data. Moreover, both the real and generated data exhibit similar autocorrelation patterns, suggesting that the model has successfully captured the autocorrelation structure of the real data. The fluctuations around

zero for both datasets indicate that there is no strong periodicity in the data (see Figure 4.43), however, most values fall within the confidence intervals. For Figures 4.44 and 4.45 we sampled a path of the generated and actual data of 400 data points and plotted their auto-correlations, which confirms that the model has successfully learned and replicated the autocorrelation structure of the real data.

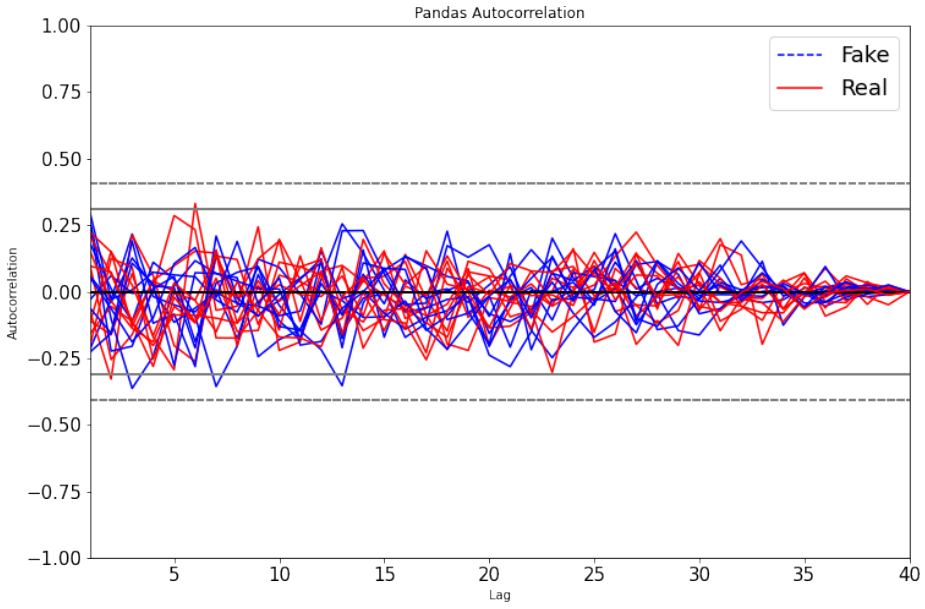


Figure 4.43: Auto-correlation plot of the real and generated data of five random paths.

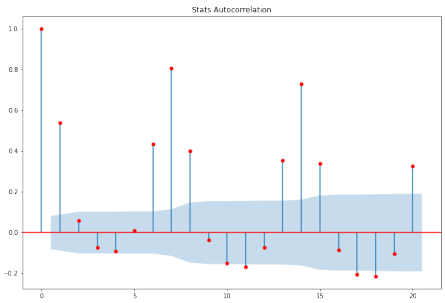


Figure 4.44: Auto-correlation plot of the real data.

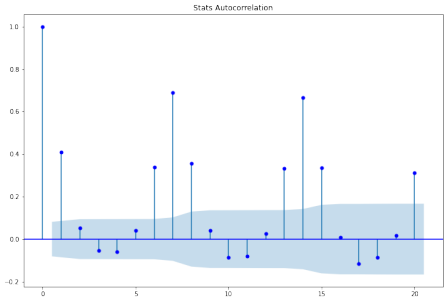


Figure 4.45: Auto-correlation plot of the generated data.

Another plot that gives us more information is the following one, see 4.46. We check the cross-correlation between 10 random paths for true and generated data after we subtract their mean from each data point. The cross-correlation plot indicates that the generated data has a strong initial correlation with the real

data, but this correlation diminishes as the lag increases. The model captures some temporal dependencies but introduces more variability and noise over longer periods.

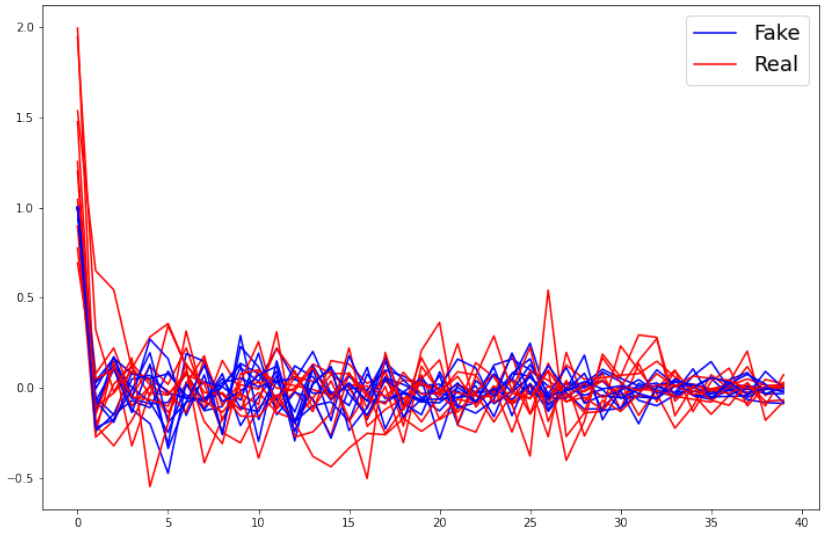


Figure 4.46: Cross-correlation plot of the real and generated data for ten random paths.

Aggregating the data into daily values speeds up the computations and simplifies the modeling process. However, our analysis shows that the model yields better forecasting accuracy and captures the underlying patterns more effectively when using the non-aggregated, higher-frequency data. Despite the computational efficiency gained through aggregation, the detailed temporal information in the unaggregated data provides richer insights and improves model performance

Summary and Implications

Our analysis of electricity consumption data using ForGAN shows that due to the regularity and seasonality of the data, data transformations are unnecessary. The results underscore the importance of careful hyperparameter tuning, particularly the number of iterations, which improves accuracy. However, huge condition windows lead to memory issues without improving the results. Aggregating data into daily values speeds up the process but reduces forecast accuracy slightly, as higher-frequency data captures more detailed patterns. Therefore, balancing computational efficiency with data resolution is crucial for achieving optimal results.

4.1 ForGAN for financial time series

Financial data, such as the S&P 500 Index, are notoriously more difficult to model compared to other types of data due to their volatility, complexity, and the influence of various unpredictable external factors like macroeconomic events. Unlike the relatively regular patterns seen in energy consumption data in the previous Section, financial data exhibit sharp fluctuations and long-term trends, making it essential to capture these dynamics accurately when generating forecasts or simulating future paths.

Further, we are going to use a different data set, the (S & P) 500 Index Data for the period 01.01.1871 − 04.01.2018. The *Standard and Poor's 500* [2] is a stock market index that tracks the stocks of 500 large-cap U.S. companies. It represents the stock market's performance by reporting the risks and returns of the biggest companies. The index is calculated by taking the sum of the adjusted market capitalization of all S & P 500 stocks and then dividing it with an index divisor, which is a proprietary figure developed by Standard & Poor's.

$$S\&P\ 500\ Index = \frac{Market\ Cap\ for\ All\ S\&P\ 500\ Stocks}{Index\ Divisor} \tag{4.3}$$

When we plot the data (see Figure 4.47), we observe a strictly increasing trend over the years. Initially, the index was relatively low, but it expanded rapidly over time, reflecting the overall growth of the U.S. economy and corporate earnings. However, this makes it more challenging to generate accurate paths and forecasts.

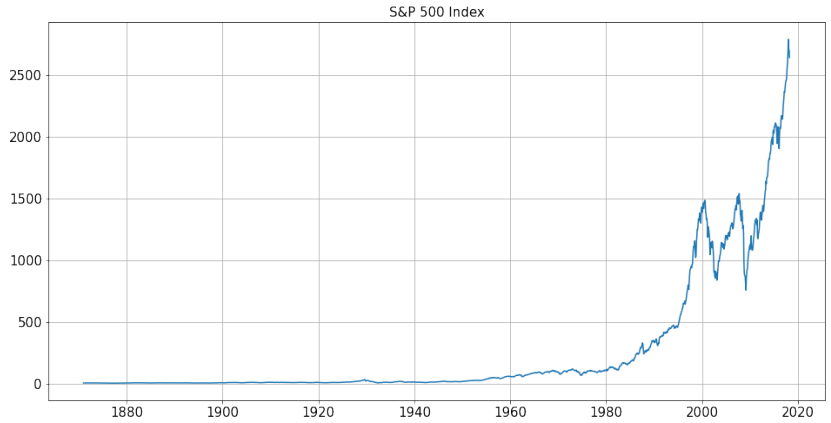


Figure 4.47: Standard and Poor's 500 Index

Consequently, the box plots in Figure 4.48 reveal numerous outliers. In this scenario, removing these outliers would not be advisable. The outliers represent significant market movements and events that are crucial for accurate forecasting and

[2] https://www.spglobal.com/spdji/en/indices/equity/sp-500/overview

path generation. By excluding these outliers, we risk oversimplifying the model and losing valuable information that reflects real market behavior and volatility.

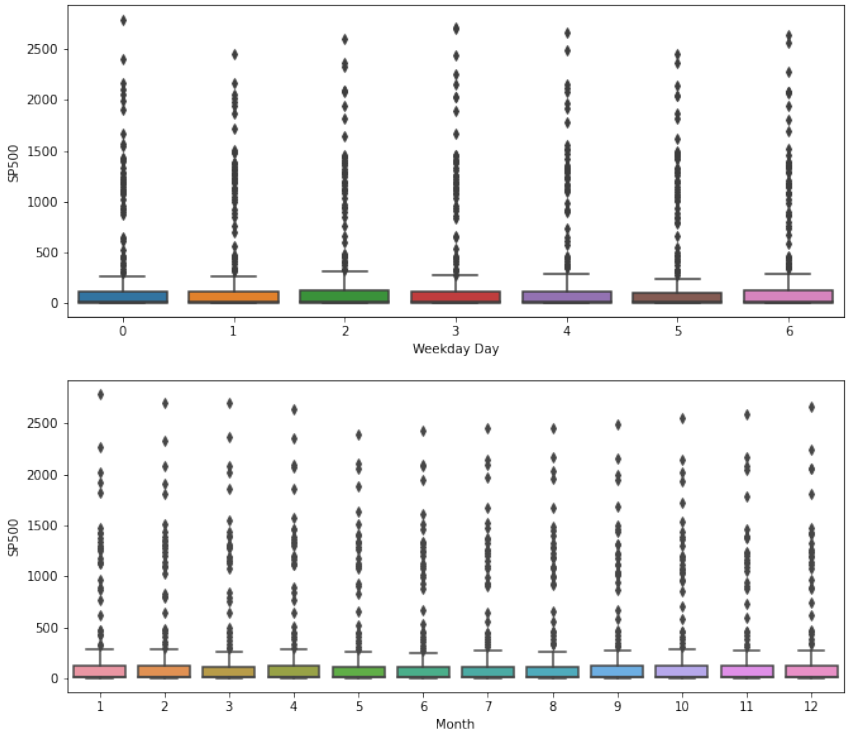


Figure 4.48: Box plots showing weekly and monthly variations in the S&P 500 Index.

Therefore, to better handle these outliers and address the challenges of volatility and non-stationarity in financial data, we calculated the log-returns of the S&P 500 Index. This transformation helps to stabilize the data and eliminate non-stationary trends. Figures 4.47 and 4.49 visually demonstrate this difference. The S&P 500 index exhibits a clear upward trend, while the log returns present a more stable and stationary pattern, with only volatility spikes around significant market events.

When we use ForGAN for forecasting the log-returns of the S & P 500 index, we get good results. See the Figures below. However, when compared to direct forecasting of the S & P 500 index values (without using log returns), the log-return forecasts provide more accurate results. This can be attributed to the more stable and stationary properties of log returns, which make it easier for the model to capture underlying trends.

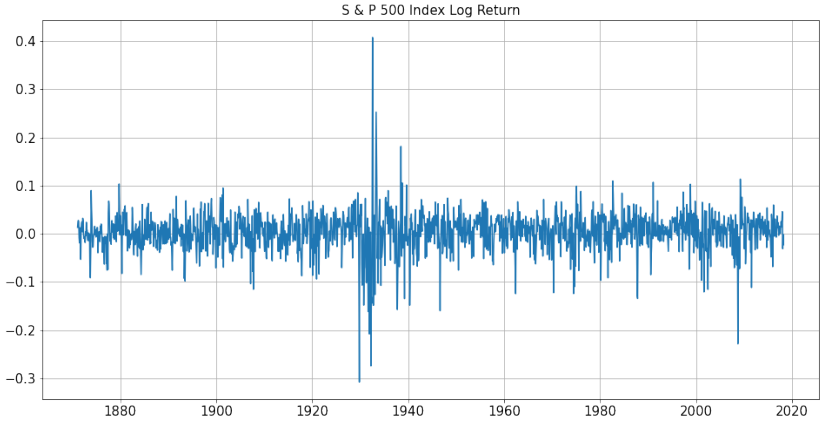


Figure 4.49: Log returns of the Standard and Poor's 500 Index from January 1871 to January 2018.

Figure 4.50 shows us the first generated path during the validation process, and Figure 4.51 shows us the last one, which follows the actual one relatively closely, but not as good as in the corresponding problem with electricity data.

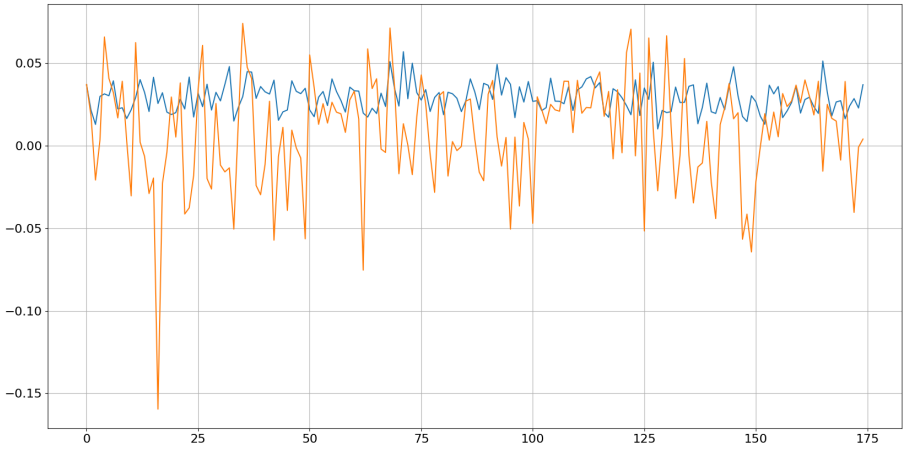


Figure 4.50: Comparison of the validation set and the first generated path for the log-returns of the S&P 500 index.

Next, we examine the Generator's and Discriminator's losses, as shown in Figure 4.52. Initially, both losses show significant fluctuations up to around 2000 iterations, reflecting the networks' early learning stages. Around 2000-3000 iterations, the losses begin to stabilize, indicating improved balance in the learning process. After 3000 iterations, both losses maintain a stable trajectory with minimal fluctuations, suggesting that the training has reached a steady state. The Generator

loss stabilizes around 0.69, with the Discriminator loss closely aligned, indicating a balanced and effective training process.

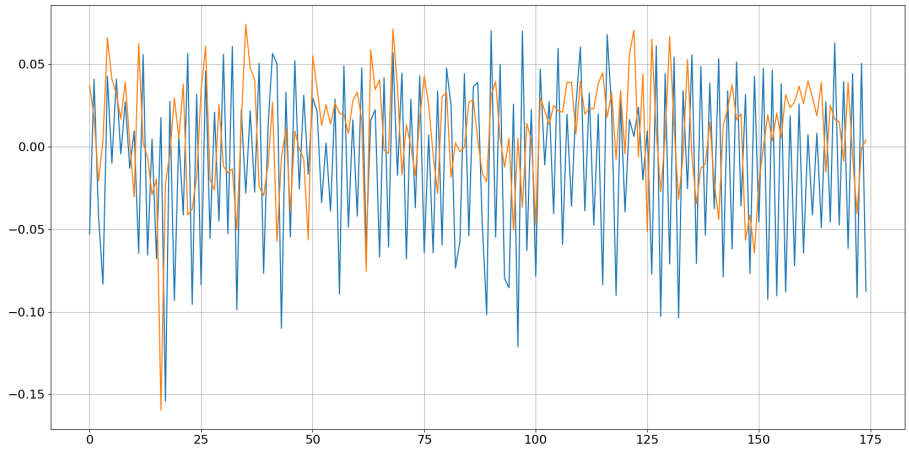


Figure 4.51: Comparison of the validation set and the last generated path for the log-returns of the S&P 500 index.

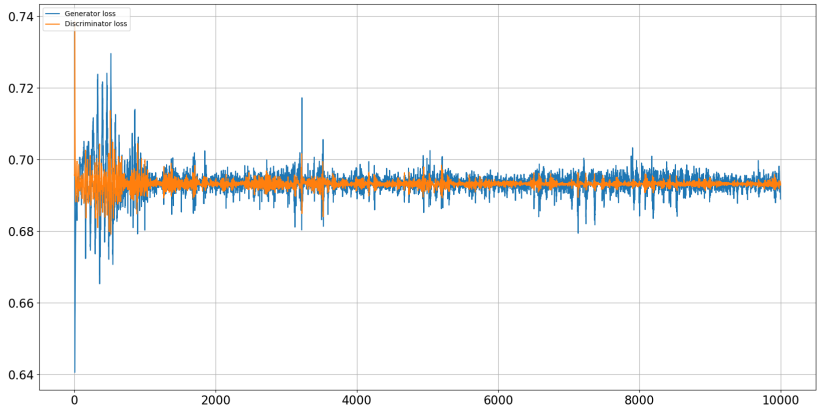


Figure 4.52: Generator's and Discriminator's loss during the training process.

The figures below show us a good performance of ForGANs for forecasting the distribution of the log-returns of the S & P 500 Index.

- Figure 4.53: This figure presents the probability distribution of the log-returns for the forecasted paths. The comparison between the real (historical) data and the generated forecasts demonstrates that ForGAN successfully captures the shape and spread of the log-returns distribution. The overlapping areas suggest that the model can replicate the true data distribution quite well.

- Figure 4.54: The plot shows the mean of fifty forecasted paths versus the actual data. Although the model generally captures the trend of the real data, there is some deviation, particularly in the extremes, where the forecasted paths exhibit more noise. Nevertheless, the model accurately captures the overall central tendency of the log-returns.
- Figure 4.55: This figure represents the cross-correlation plot of real and generated data over ten random paths. The high initial correlation values, which gradually decrease over time, suggest that ForGAN performs well in modeling short-term dependencies in the data, although the accuracy declines slightly over longer time horizons, as expected in financial data forecasting.

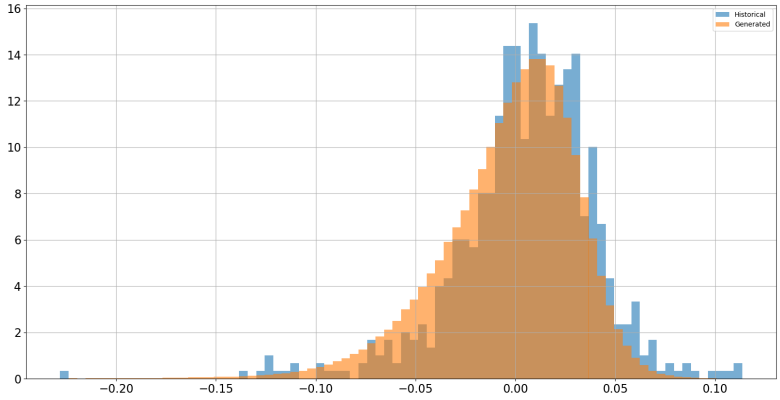


Figure 4.53: Probability distribution of one step ahead forecast learned from ForGAN for log-returns of S & P 500 index.

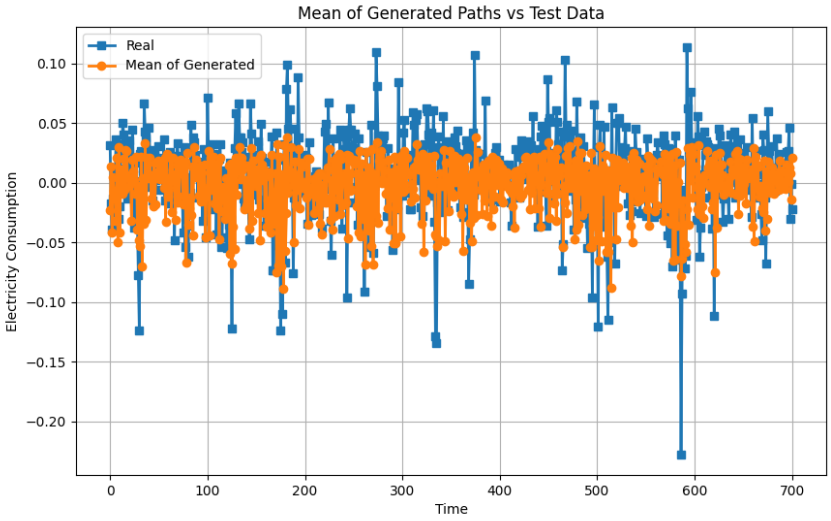


Figure 4.54: Mean of fifty forecasted paths versus actual one using ForGAN for log-returns of S & P 500 index.

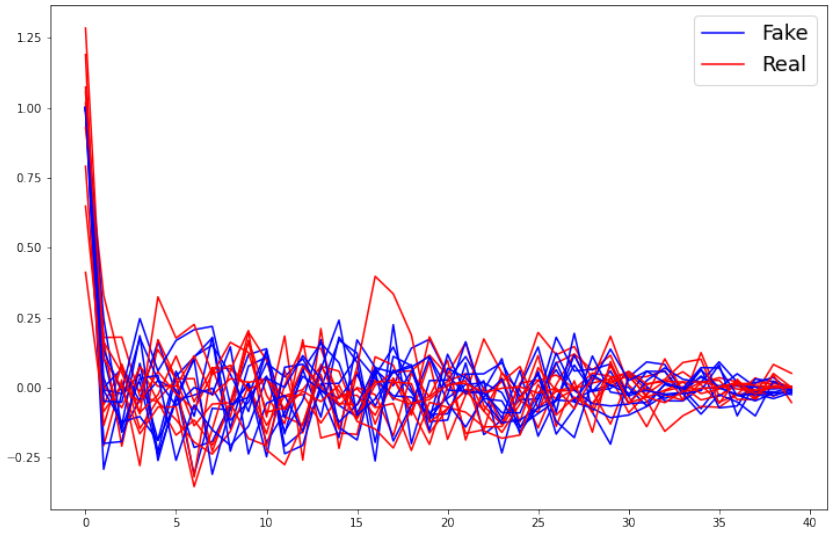


Figure 4.55: Cross-correlation plot of the real and generated data for ten random paths.

Figure 4.56, showing S&P 500 close prices for five years, from *March 22, 2017*, to *March 21, 2022*, is better for forecasting and analysis because it focuses on recent and relevant data, providing detailed views of short- to medium term trends. This recent data is more likely to be stationary and reflects current market conditions, leading to more accurate forecasts. Additionally, the smaller, more manageable dataset avoids the noise and complexity of long-term data, making analysis more efficient.

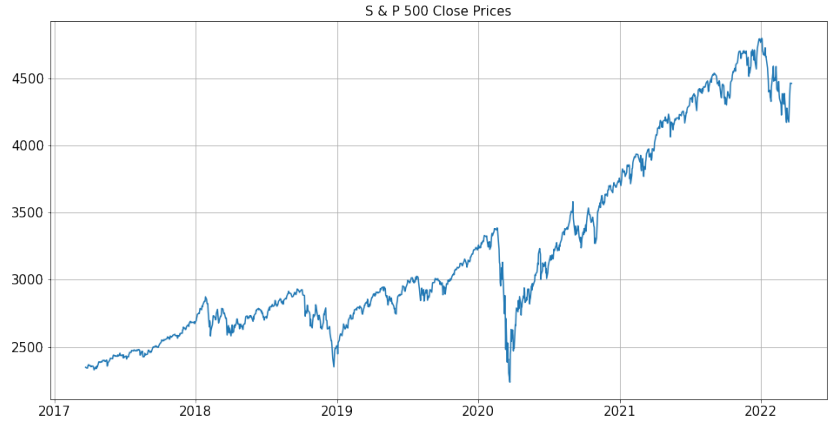


Figure 4.56: S & P 500 close prices for five years.

Table 4.5 shows the first five rows of this dataset, which includes the Close, Open,

High, and Low prices for each trading day. Notably, there are no missing values in this dataset, ensuring the completeness and reliability of our analysis.

For our study, we will focus exclusively on the Close prices. We plotted the Close prices over time, resulting in 1,259 observations. This plot is shown in Figure 4.56.

Date	**Close/Last**	**Open**	**High**	**Low**
2022-03-21	4461.18	4462.40	4481.75	4424.30
2022-03-18	4463.12	4407.34	4465.40	4390.57
2022-03-17	4411.67	4345.11	4412.67	4335.65
2022-03-16	4357.86	4288.14	4358.90	4251.99
2022-03-15	4262.45	4188.82	4271.05	4187.90

Table 4.5: S & P 500 Data Set.

Next, to address the issue of trends, we analyze returns instead of prices to obtain a more stationary time series. As previously mentioned, we also examine log returns because they help eliminate the non-stationary properties of the time series. When we plot the data, we observe some small, almost imperceptible differences between the returns and log-returns of the Close prices, as illustrated in Figure 4.57.



Figure 4.57: S & P 500 returns and log returns of close prices for five years.

Furthermore, we use our ForGAN model to forecast the log returns of the S&P 500 Index close prices. The results show a good level of accuracy, but as previously mentioned, ForGAN performs better when more data points are available. In this case, we observed that the standard min-max loss function yielded better results than the previously used non-saturating loss. The forecasted results are shown in Figures 4.63, 4.64 below.

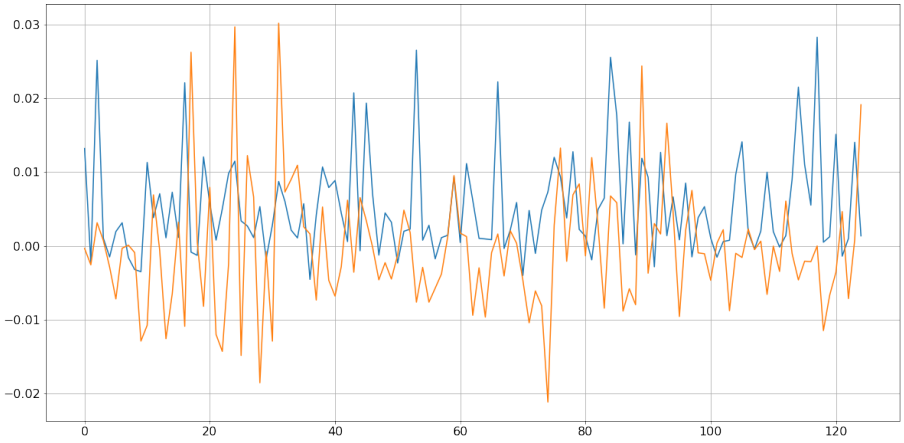


Figure 4.58: The validation set and generated set on the first step.

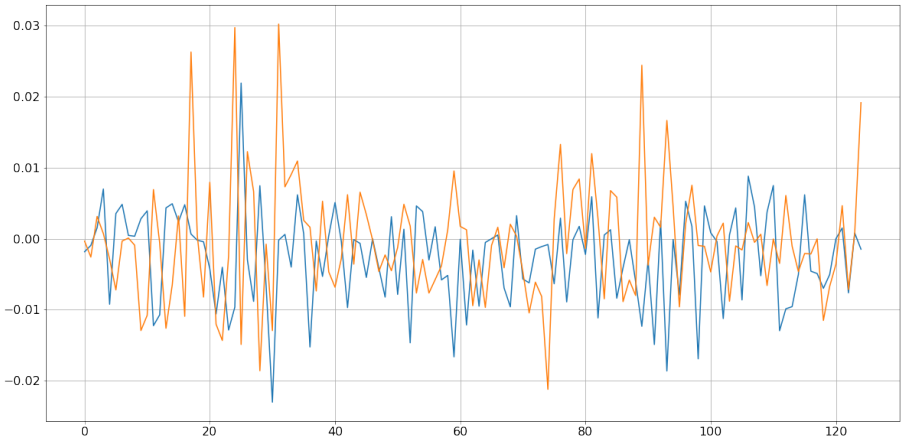


Figure 4.59: The validation set and generated set on the last step.

Next, we examine the Generator's and Discriminator's loss, as shown in Figures 4.60, 4.61 (note that the scales for the two plots differ), and 4.62. The losses suggest that while the generated distribution is relatively close to the actual data distribution, the GAN training process exhibits instability. This is evident from the figures where the Discriminator's loss displays greater stability compared to the Generator's loss.

The observed stability in the Discriminator's loss implies that the Discriminator is learning to differentiate between real and generated data quite well. On the other hand, the Generator's fluctuating loss suggests that it is struggling to keep up with the Discriminator's advancements. Therefore, we should consider increasing the

capacity of the Generator by adding more layers or neurons. Additionally, reducing the number of training iterations for the Discriminator might help prevent it from becoming very capable of distinguishing the data, thus allowing the Generator to improve its performance.

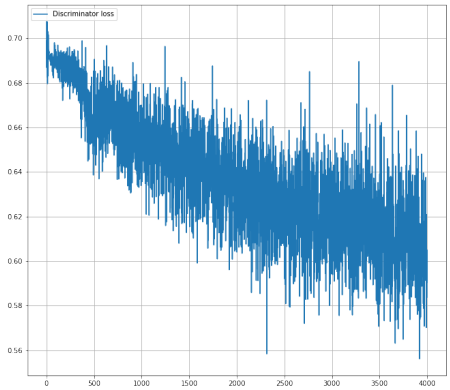


Figure 4.60: Discriminator's loss during the training process.

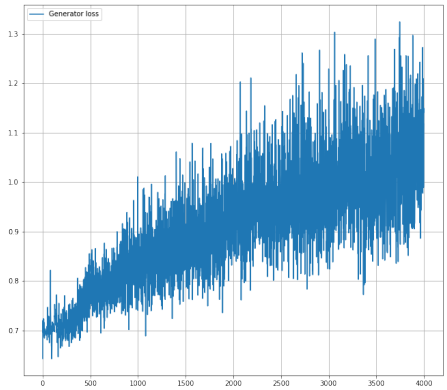


Figure 4.61: Generator's loss during the training process.

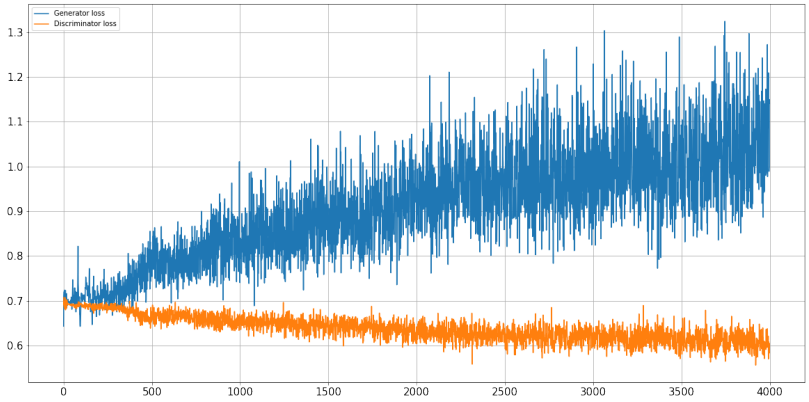


Figure 4.62: Generator's and Discriminator's loss during the training process.

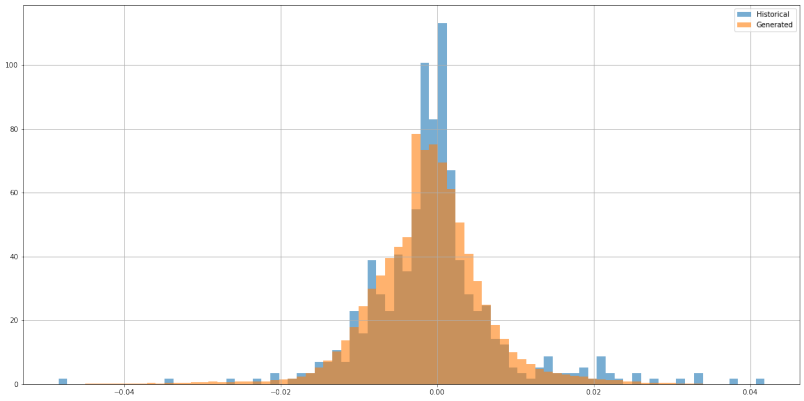


Figure 4.63: Probability distribution of one step ahead forecast learned from ForGAN for log-returns of S & P 500 index.

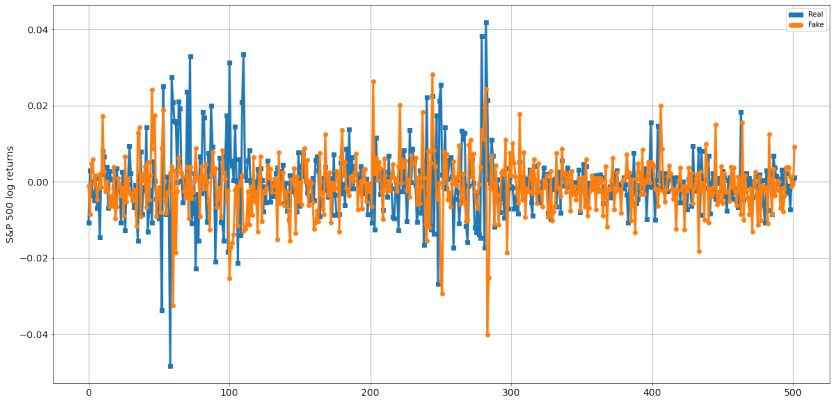


Figure 4.64: Mean of five forecasted paths versus actual one using ForGAN for log-returns of S & P 500 index.

After implementing the suggested adjustments—namely, increasing the capacity of the Generator and reducing the number of training iterations for the Discriminator—we observed notable improvements in the training process. Figure 4.65 illustrates the outcomes of these adjustments, including increasing the number of training iterations, after 3000 iterations we observe a stable loss for both the Generator and Discriminator.

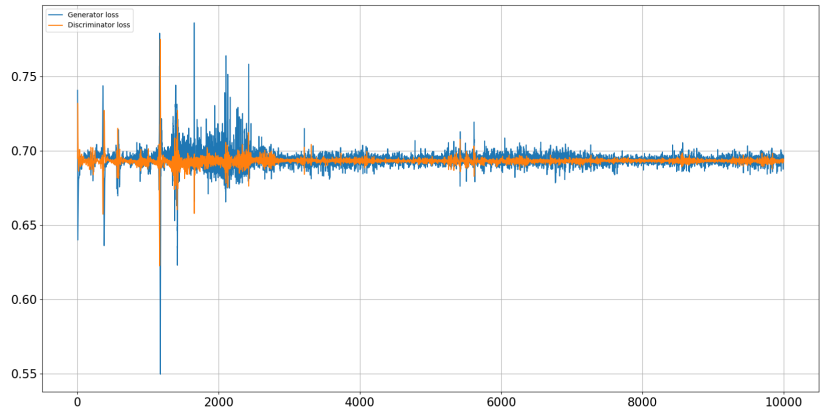


Figure 4.65: Improved Generator's and Discriminator's loss during the training process.

Summary and Implications

While ForGANs offer a promising approach to generating financial time series, their performance has been moderate when applied to complex financial time series like the S&P 500, as shown in Figure 4.47. The S&P 500 Index is marked by high volatility, non-stationarity, and complex patterns such as heavy tails, autocorrelation, and sudden market shifts. These characteristics make forecasting particularly challenging, as GANs struggle to capture the unpredictable and often chaotic nature of financial markets.

On the other hand, when applied to forecasting 15-minute electricity consumption data, ForGANs have demonstrated significantly better performance. Unlike the S&P 500, electricity consumption data is much more regular, exhibiting clear seasonal patterns and almost no outliers. These regular and stable patterns in electricity consumption make it easier for GANs to model and generate accurate forecasts, highlighting how the nature of the dataset—whether volatile and unpredictable like the S&P 500 or regular and consistent like electricity usage—significantly affects the effectiveness of GAN-based models.

4.2 Forecasting Value at Risk using ForGAN

Effective risk management has become an essential task for any major financial institution. The view to control financial risks more accurately has gained widespread acceptance, especially in light of the increasing number of financial crashes [11]. To this end, the Basel Committee on banking supervision [29] announced financial regulations that banks should adhere to impose a uniform measure of risk and stabilize the economy. The risk model that the financial regulators charged became known as the Value at Risk (VaR) measure.

Due to its importance for effective risk management, it is crucial that the reported

VaR is a reliable estimate of the corresponding true quantile of the loss distribution. The choice of the appropriate model for estimating this loss distribution is, therefore, of utmost importance. So many methods have been proposed over the years, and a very good survey of all those models is given in [50]. Most of these approaches are based on classical tools from statistics and probability theory such as maximum likelihood estimation, extreme value theory (EVT) or time series analysis.

With the advancement of machine learning, new approaches were introduced for solving the problem of distribution estimation, such as *generative models*. As mentioned, a crucial example of generative models are Generative Adversarial Networks (GANs). We have specified different applications of GANs, now we will see if we are able to sample from the much lower dimensional distribution of portfolio returns and hence might constitute a credible competitor of the classical methods for Value-at-Risk estimation. Until now very little work has been done in this direction, such as those two blog posts [70, 77], where the authors use BiGAN and WGAN-GP to estimate VaR.

As we have seen previously with GANs instead of estimating the parameters of a closed-form probability density function from the available data, we generate new samples approximating the true underlying distribution.

Value at Risk (VaR) is a statistical measure that estimates the potential maximum loss a portfolio could face over a specific time frame, given a certain level of confidence. It quantifies the monetary amount at risk, indicating the worst expected loss under normal market conditions (without unexpected disruptions like financial crises or extreme volatility). The time period depends on how long the portfolio is assumed to be held constant. The general formal definition of VaR is as follows:

Value at Risk is a given maximum amount of money that may be lost on a portfolio over a given period of time, with a given level of confidence [17].

The precise definition of *Value at Risk (VaR)* can be formulated as follows: Consider an asset with market values Y_t over n periods, and let the log-returns be $X_t = \log(Y_t/Y_{t-1})$. Assuming the sequence $\{X_t\}_{t=1}^n$ is strictly stationary with a marginal distribution function F, the VaR at confidence level $1-\lambda$ (where $\lambda \in (0,1)$ and closed to zero) is defined as:

$$v_\lambda = -\inf\{u : F(u) \geq \lambda\},$$

which specifies the maximum loss with a given confidence level $1-\lambda$. If we denote the λ quantile of $F(x)$ as $Q(\lambda = \inf\{t : F(t) \geq \lambda\})$ then $v_\lambda = -Q(\lambda)$.

Traditional VaR estimators often rely on parametric models, such as Gaussian or t-distributions, which are computationally convenient and widely used. However, these methods assume that returns follow a particular distribution, which may not accurately reflect real market conditions, especially during times of market stress or crisis. Parametric models, for example, tend to underestimate risk in the

presence of fat-tailed distributions, where extreme losses are more frequent than predicted by the model. This mismatch between assumptions and reality can lead to significant underestimation of risk, especially in volatile market conditions.

Since the true return distribution is difficult to ascertain in practice, recent approaches have employed extreme value theory, which is particularly effective for estimating VaR at very small tail probabilities.

Additionally, nonparametric estimators have gained attention for their ability to handle fat-tailed distributions without assuming a specific form for the distribution function F, and for requiring weaker assumptions regarding the return process dynamics. These advantages make nonparametric methods a valuable tool for more accurate and robust VaR estimation.

Dowd (2001) [21] explored the Value at Risk (VaR) estimation using nonparametric methods based on sample quantiles. The simulation results indicated that estimation accuracy improved as the sample size increased. However, Berkowitz and O'Brien (2002) [23] reported that VaR estimates often tend to be conservative. They attributed this tendency to a notable positive bias in sample quantiles for portfolios with fat-tailed distributions. Koji et al. (2005) [27] also found that this estimator frequently results in a significant positive bias. Consequently, there is a need to develop alternative methods for VaR estimation. Gourieroux et al. (2000) [18] introduced a nonparametric kernel estimator for VaR, and Chen and Tang (2005) [26] proposed a nonparametric VaR estimation method based on kernel quantile estimation.

In recent developments, Buch et al. (2023) [89] propose a temporal variational autoencoder (TempVAE) for estimating VaR. To address posterior collapse, a common issue in VAEs, they employ annealing regularization, improving the model's ability to capture relevant features in financial data.



Figure 4.66: S & P 500 close prices over twelve years. The prices show upward trends and increasing variance, indicating non-stationarity.

Their approach outperforms traditional methods like GARCH and historical simulations, yielding more accurate VaR estimates under real market conditions.

For a comprehensive exploration of Value at Risk (VaR) and its applications across various financial markets, P. Jorion provides a detailed examination in his book [30], which can be referred to for further insights.

We utilize the S&P 500 index dataset, covering the period from *January 4, 2010*, to *December 31, 2021*. This dataset includes Close, Open, High, and Low prices, with no missing values. Our analysis will focus exclusively on the Close prices over these 12 years.

For our study, we use log returns because they help reduce the non-stationary properties of the time series, making the data more suitable for analysis. Visually, we observe that while the original price series shows clear trends and increasing variance over time (Figure 4.66), the log returns exhibit a more stationary behavior, with relatively consistent variance (Figure 4.67). Although some degree of non-stationarity remains, the transformation makes the dataset more stable and suitable for the modeling techniques we apply, improving the reliability of our results.

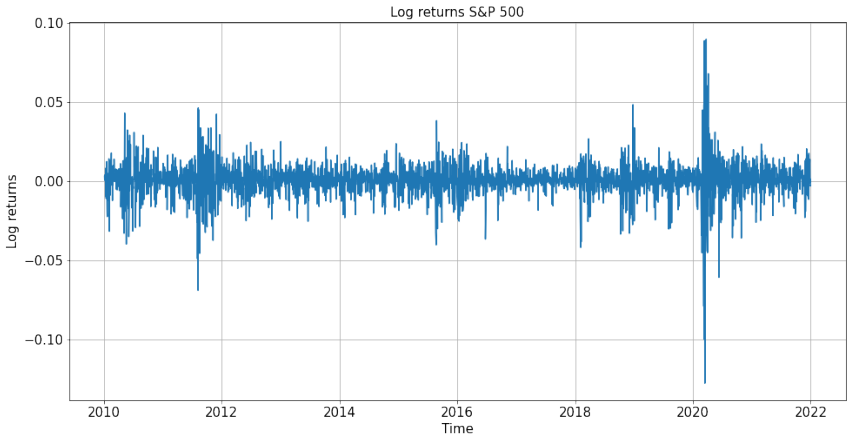


Figure 4.67: S & P 500 log returns of close prices over twelve years. The log returns display a more stationary behavior.

Table 4.6 provides a detailed overview of the descriptive statistics for the S&P 500 log returns dataset. The mean log return is 0.000476, and a standard deviation of 0.010855, highlighting the variability in daily returns. The minimum and maximum log returns are -0.127652 and 0.089683, respectively, which indicates significant fluctuations during certain days.

We train our model using 50% of the dataset (1510 data points), while the next 10% is used for validation and the remaining 40% for testing. For the test period, we generate synthetic data using GANs, aiming to replicate the original data's

distribution. We then compute the VaR separately for both the original test set and the GAN-generated data.

S & P 500 log returns	
Mean	0.000476
Stdev	0.010855
Minimum	-0.127652
25% Quantile	-0.003473
50% Quantile	0.000706
75% Quantile	0.005383
Maximum	0.089683

Table 4.6: Descriptive statistics of the S & P 500 log returns.

During the validation process we get the following paths, see Figures 4.68 and 4.69. We can see that the last generated path follows the actual path at a fair level.

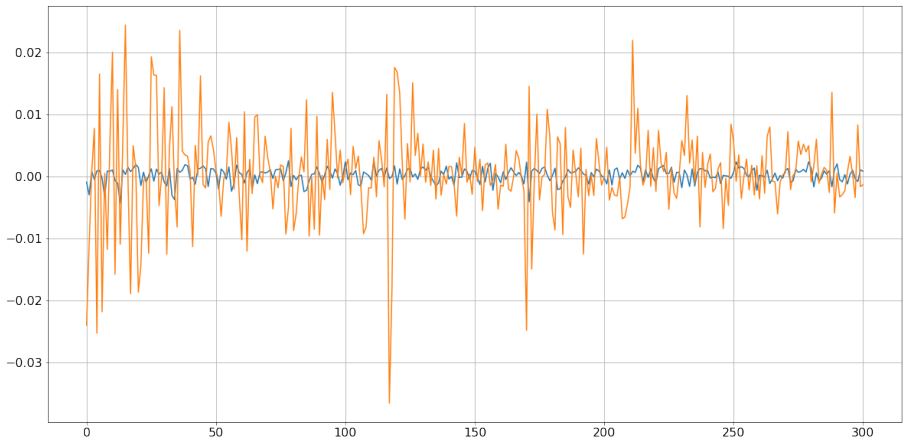


Figure 4.68: The validation set and generated set on the first step.

Figures 4.70, 4.71, and 4.72 show the training progress using the non-saturating loss for both the generator and discriminator over 3000 training iterations.

Both the generator and discriminator losses appear stable over time without significant oscillations or divergence. This stability is a positive sign, suggesting that the training process is not experiencing instability or mode collapse.

The generator loss remains in the range of approximately -0.6 to -0.4, while the discriminator loss is relatively stable around 0.6. These loss values indicate balanced training dynamics where the generator is continually challenged by the

discriminator, and the discriminator is effectively distinguishing between real and generated samples.

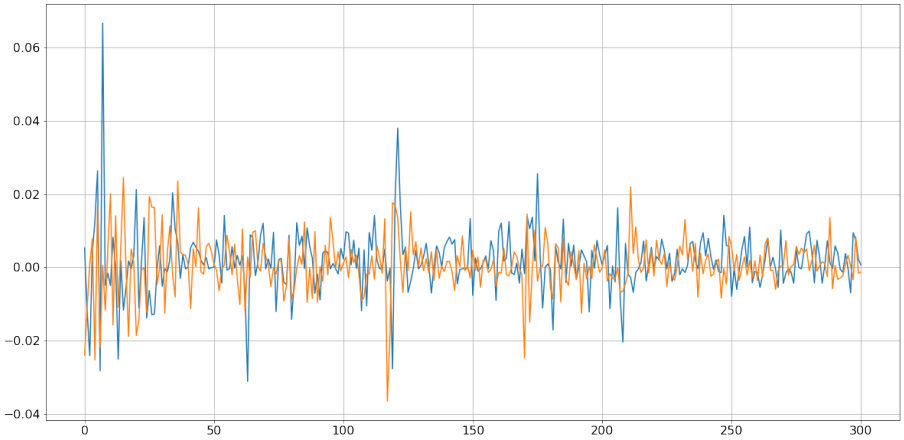


Figure 4.69: The validation set and generated set on the last step.

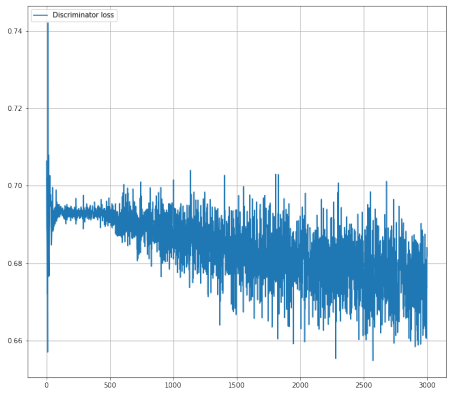


Figure 4.70: Discriminator loss over 3000 training iterations for log-returns of the S&P 500 index.

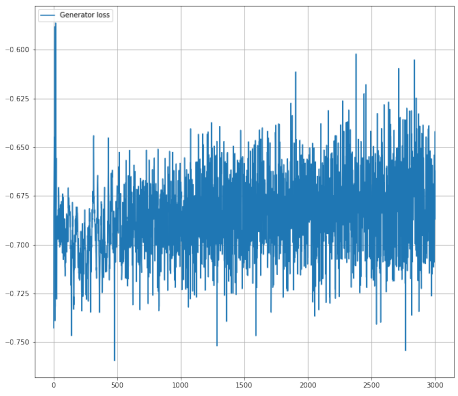


Figure 4.71: Generator loss over 3000 training iterations for log-returns of the S&P 500 index.

From the probability distribution in Figure 4.73, we can say that the generated and actual data follow a pretty similar distribution. The results in Table 4.7 indicate that the ForGAN model achieves a low Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE), suggesting good accuracy and precision in the generated paths.

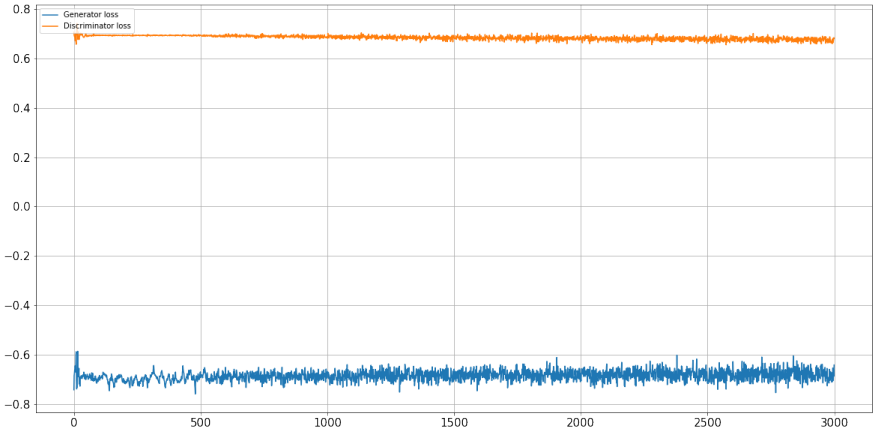


Figure 4.72: Generator and discriminator losses during training for log-returns of the S&P 500 index.

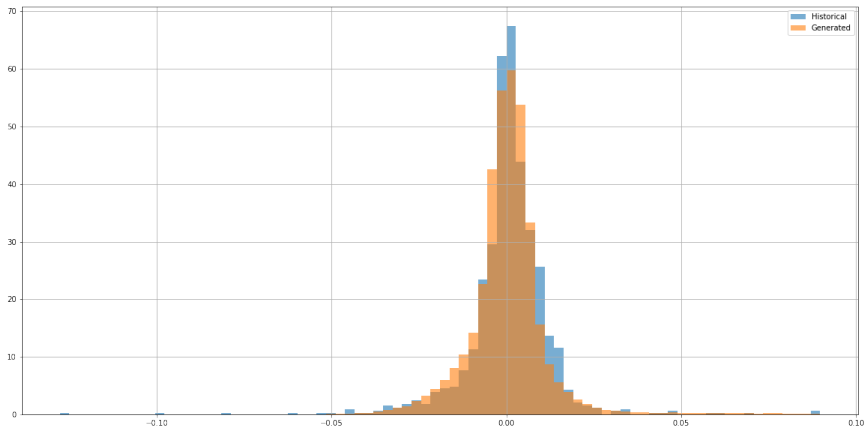


Figure 4.73: Probability distribution of one-step-ahead forecasts learned from For-GAN for the log-returns of the S&P 500 index.

The Mean Absolute Percentage Error (MAPE) is relatively high, which may indicate sensitivity to small true values or high variability in percentage terms. The Continuous Ranked Probability Score (CRPS) is very low, indicating that the probabilistic forecasts are well-calibrated, while the higher Kullback-Leibler Divergence (KLD) points to some differences in the distribution between generated and real data.

Metric	Mean	Standard Deviation
RMSE	0.019775	0.000760
MAE	0.000651	0.000313
MAPE	136.673	94.156
CRPS	0.000648	-
KLD	24.817	-

Table 4.7: Error results from the generated path using ForGANs.

For calculating Value at Risk we use three different methods namely: *historical method*, *variance-covariance method* and *monte carlo simulation method*. As previously described, each of these methods has its own advantages and disadvantages.

1. **Historical Method**:

- **Advantages**: Simple to implement, does not require assumptions about the return distribution, and reflects actual market conditions.
- **Disadvantages**: Assumes that historical patterns will repeat in the future, may not accurately capture future risks, and can be influenced by outliers or non-stationary data.

2. **Variance-Covariance Method**:

- **Advantages**: Computationally efficient, easy to understand, and widely used.
- **Disadvantages**: Assumes normally distributed returns, which may not be accurate for financial data with fat tails and skewness, potentially underestimating risk during turbulent market conditions.

3. **Monte Carlo Simulation Method**:

- **Advantages**: Flexible, capable of modeling complex behaviors and nonlinearities, and does not require the assumption of normality.
- **Disadvantages**: Computationally intensive, requires extensive data and assumptions about the return distribution and can be sensitive to model parameters.

Next, we assess which distribution best fits our dataset. Initially, we check the distribution of the real test data against the t-distribution, normal distribution, log-normal distribution, and beta distribution (Figure 4.74). As expected, the normal, log-normal, and beta distributions do not accurately model the underlying data, likely due to their lighter tails and the absence of a strong peak around the mean, which is critical for financial data. The t-distribution, however, provides a much better fit, as confirmed by the results in Table 4.8. Therefore, due to the flatness of the normal distribution, we use the t-distribution to simulate data with GANs.

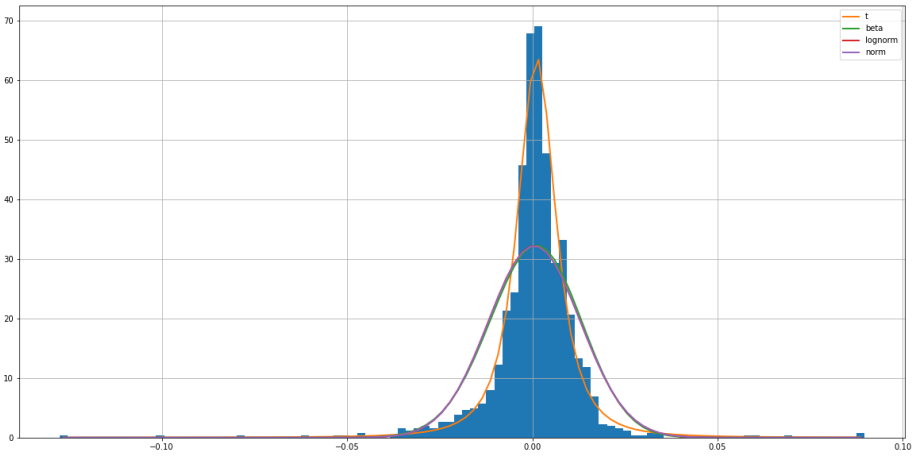


Figure 4.74: Comparison of the real data distribution with t, beta, log-normal, and normal distributions.

Distributions	**Sum-Square Error**	**AIC**	**BIC**
t	399.434	246.039	-1313.468
beta	3991.300	1906.484	1471.930
log normal	4049.749	2144.677	1482.382
normal	4055.388	2115.223	1476.965

Table 4.8: Comparison of different distributions with the true dataset.

We perform the same distributional analysis on the data generated by the GAN. Figure 4.75 and Table 4.9 demonstrate that the t-distribution provides the best fit for the generated data as well. Although the generated data display longer tails compared to the real data, their density closely matches that of the t-distribution.

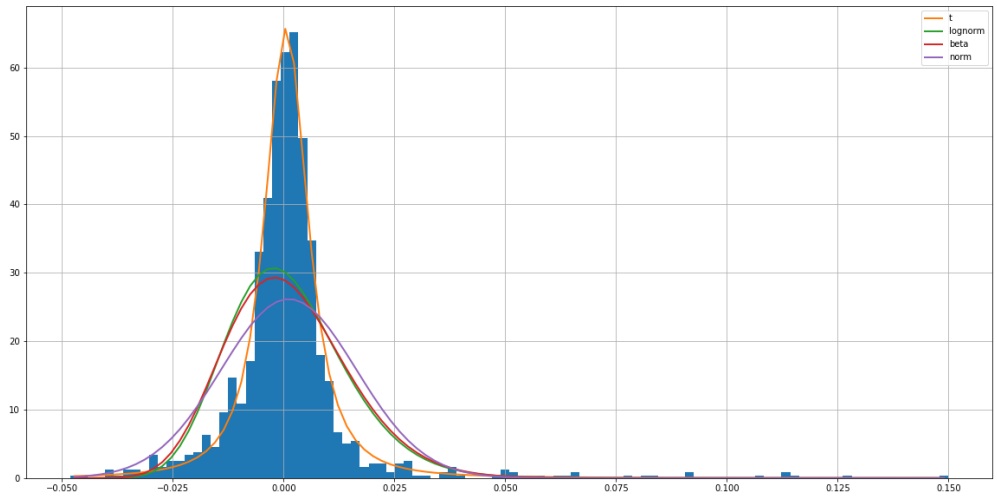


Figure 4.75: Comparing the distribution of the generated data with t, beta, log normal, and normal distribution.

Distributions	Sum-Square Error	AIC	BIC
t	185.977	233.601	-2236.127
beta	5193.851	877.573	1782.708
log normal	5527.104	989.981	1864.865
normal	6341.479	1824.105	2016.573

Table 4.9: Comparison of different distributions with the generated dataset.

The t-distribution's superior fit for both real and generated data highlights its suitability for financial data analysis, capturing key characteristics such as heavy tails and central peak more effectively than the other distributions considered. The fact that the t-distribution fits both the real and generated data well suggests that the GAN is successful in capturing the key statistical properties of the real data, such as heavy tails and central peak. However, the generated data exhibit longer tails compared to the real data.

We use the test data and calculate the Value at Risk using the *Historical*, *Variance-Covariance* and *Monte Carlo simulation* methods.

First, we estimate the VaR at 95% and 99% confidence interval for true test data using the **Historical Method**, as shown in Figure 4.76. We get the following values for historical VaR estimation:

$$1\% \, Var: -3.57\%$$
$$5\% \, Var: -1.78.\%$$

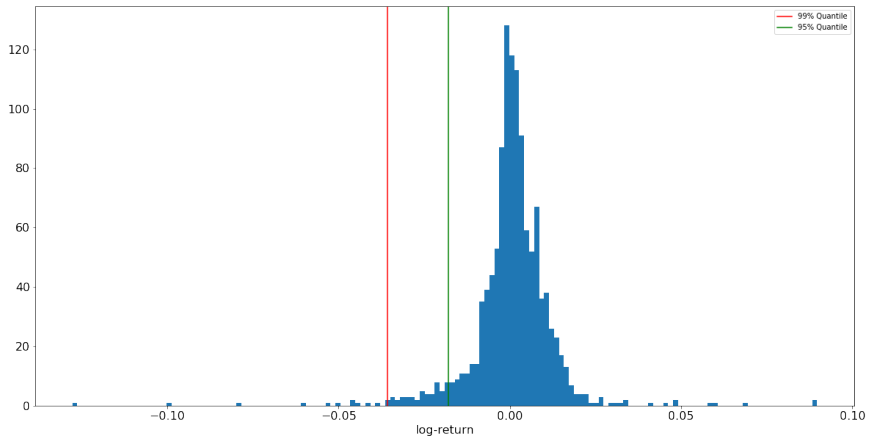


Figure 4.76: Historical value at risk for 95 % and 99 % confidence intervals for the true data.

These values indicate that there is a 1% chance to have a return worse than -3.57%, and a 5% chance to have a return worse than -1.78%.

Next, we perform the same estimation for the generated data using the Historical VaR method. Figure 4.77 shows the results, and the estimated VaR values are close to those obtained from the true data:

$$1\% \, Var: -2.65\%$$
$$5\% \, Var: -1.64.\%$$

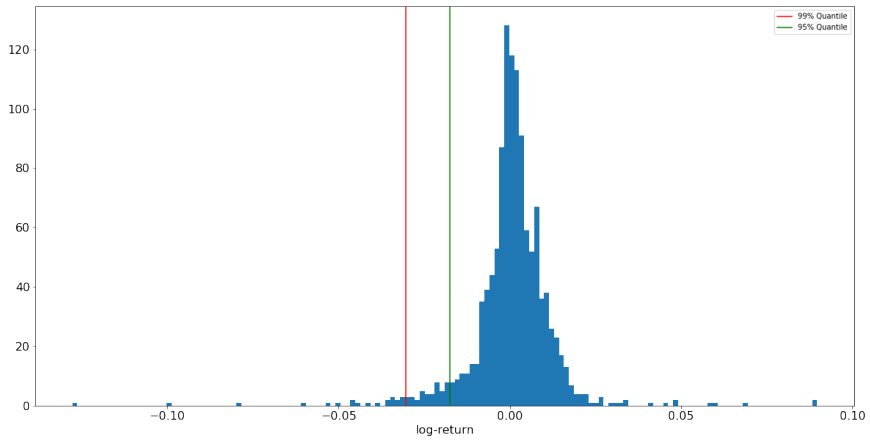


Figure 4.77: Historical value at risk for 95 % and 99 % confidence intervals for the generated data.

The Historical VaR for the real data shows a 99% confidence level loss of 3.57% and a 95% level loss of 1.78%. The generated data produces slightly lower values: 2.65% and 1.64%, respectively. This close alignment suggests that the GAN effectively captures the risk profile of the real data, with only minor differences.

Further, we estimate the VaR using the **parametric method**, specifically the variance-covariance method, which assumes normally distributed returns. For the true data, we obtain the following results:

$$1\% \, Var: -2.83\%$$
$$5\% \, Var: -1.98.\%$$

For the generated data using ForGAN we get the following value at risk estimations:

$$1\% \, Var: -3.74\%$$
$$5\% \, Var: -2.62.\%$$

The variance-covariance method, which relies on the assumption of normality, provides a slightly lower estimate of risk for the true data compared to the historical method. For the ForGAN-generated data, however, the VaR estimates are noticeably higher. This suggests that the GAN model might capture a greater range of extreme outcomes, leading to predictions of more severe potential losses. The

higher VaR values for generated data could indicate that the model anticipates higher risk or extreme market events that are not as prominent in the historical data.

For the **Monte Carlo simulation** of VaR, we performed 1000 simulations of the price movement over the next 31 days. The simulation approach provides us with a flexible way to model potential price paths by introducing stochasticity, allowing us to assess extreme losses that may not have occurred in the observed dataset. While the t-distribution fits well, using Monte Carlo simulations helps in exploring a wide range of scenarios beyond the limitations of parametric assumptions. Based on these simulations, we calculated the VaR. The results for the actual data are shown in Figure 4.78.

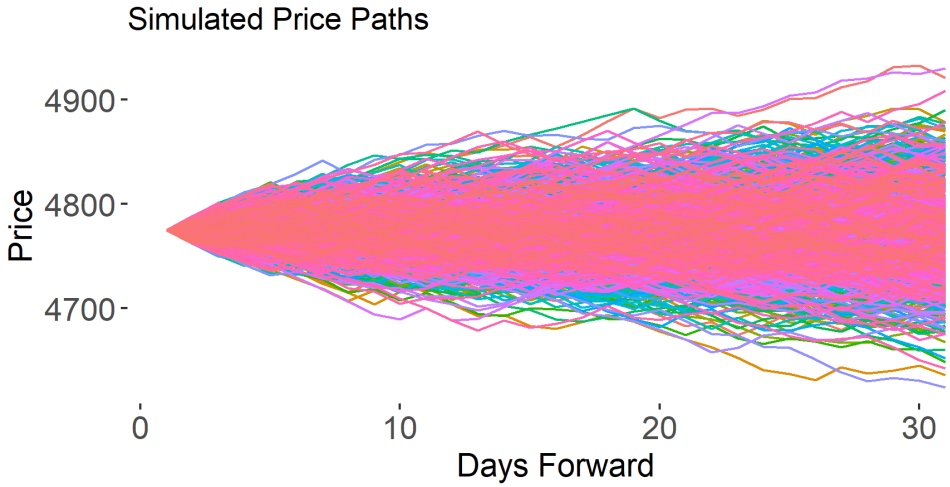


Figure 4.78: Monte-Carlo simulation of the prices for true data.

Run	DaysForward	Price
700	31	4624.410
84	31	4635.933
924	31	4642.474
307	31	4648.558
623	31	4652.021
376	31	4660.494
237	31	4667.733
273	31	4675.386
886	31	4675.423
898	31	4675.676

Table 4.10: Monte-Carlo simulated prices for actual data.

Next we create a data frame with the 1000 simulations of the price on the 31^{st} day.

By ranking these prices from lowest to highest (see Table 4.10), we can identify the 5^{th} and 1^{st} lowest prices to determine the VaR at the 95% and 99% confidence levels, respectively, since we have 1000 simulations.

Therefore, based on our Monte-Carlo simulations, we can calculate our estimated loss with 1% and 5% certainty. We have taken our starting price to be the last one in our test data set \$4775.21, and we subtract it from the 1^{st} and 5^{th} lowest prices from the simulation results table.

$$\$4775.21 - \$4624.410 = \$150.8$$
$$\$4775.21 - \$4652.021 = \$123.189$$

In other words, we could be 99% and 95% sure that daily loss for our portfolio of S & P 500 close prices will not exceed \$150.8 and \$123.189 respectively, in 31 days for actual data.

In the same way we estimate VaR by Monte-Carlo simulations for GAN-generated data. When we generate 1000 simulations of the prices for the next 31 days, we obtain the results shown in Figure 4.79. Further, we sort the obtained prices from the lowest to the highest and calculate the potential loss.

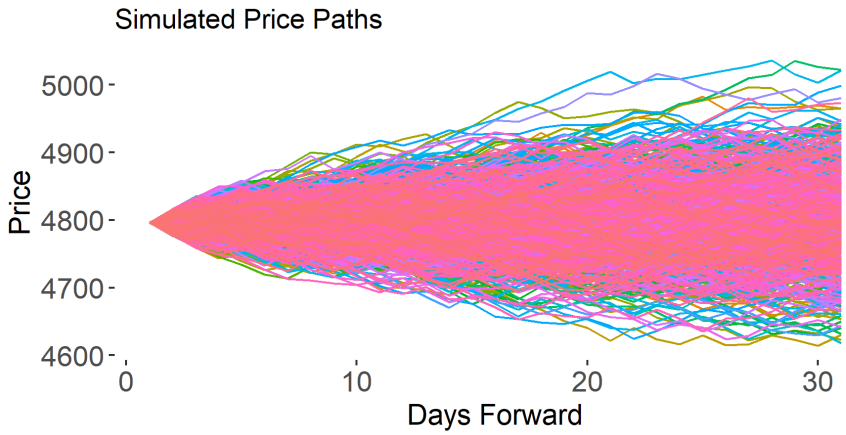


Figure 4.79: Monte-Carlo simulation of the prices for generated data.

Table 4.11 shows the prices starting from the worst one. Using these values, we calculate the loss for the 99% and 95% confidence intervals. In this case, the starting price is the last one from the generated data set, which is \$4796.32.

The potential losses are calculated as follows:

$$\$4796.32 - \$4617.813 = \$178.507$$
$$\$4796.32 - \$4640.184 = \$156.136$$

In other words, we could be 99% and 95% sure that daily loss for our portfolio of S & P 500 close prices will not exceed \$178.507 and \$156.136 respectively, over the next 31 days for generated data.

Run	DaysForward	Price
527	31	4617.813
217	31	4622.930
155	31	4629.984
353	31	4631.617
323	31	4640.184
814	31	4641.887
656	31	4643.329
338	31	4650.334
205	31	4652.759
84	31	4653.673

Table 4.11: Monte-Carlo simulated prices for generated data.

For the actual data, the 99% VaR was \$150.8, and the 95% VaR was \$123.189. For the generated data, the 99% VaR is \$178.507, and the 95% VaR is \$156.136. This comparison indicates that the generated data predicts higher potential losses at both the 99% and 95% confidence levels compared to the historical data. This suggests that the generated data reflects a higher degree of risk, potentially capturing more extreme price movements and fat-tailed risks, across different confidence levels.

Overall, the comparison between historical and GAN-generated data demonstrates that the GAN-based VaR estimation tends to predict higher risk. The GAN model seems to capture more extreme movements and heavier tails, which is particularly evident in the Monte Carlo simulations. This capability is useful for anticipating severe market events that may not have been fully reflected in historical data, providing a more robust risk assessment framework for extreme scenarios.

GARCH and Backtesting

Next, we are going to use *Generalised Autoregressive Conditional Heteroskedasticity (GARCH)* modeling to the log returns generated by our ForGAN model in order to forecast Value-at-Risk (VaR). GARCH, introduced by Bollerslev [5], extends the Autoregressive Conditional Heteroskedasticity (ARCH) model developed by Engle [4]. GARCH models are robust tools developed for the modeling of high-frequency time-series data. Past experiments show that these models efficiently capture the stylized feature of volatility clustering in financial data. GARCH models are, therefore, most often used to forecast volatility and subsequently Value-at-Risk (VaR) [22].

There are several variations of GARCH models, but here we apply the standard GARCH(1,1) model, which has proven effective for forecasting volatility. Using the generated log returns, the estimated coefficients for the model are as follows:

	Estimate	Std. Error	t value	Pr(> |t|)
mu	-0.00074	0.00024	-3.11593	0.00183
omega	0.00002	0.00000	7.46184	0.00000
alpha1	0.61470	0.07057	8.70993	0.00000
beta1	0.38430	0.04385	8.76340	0.00000

Table 4.12: sGARCH(1,1) model's coefficient for generated data.

Using those coefficients we can write the model's equation:

$$r_t = \mu + \epsilon_t = -0.000737 + \epsilon_t \tag{4.4}$$

where ϵ_t is the error term, which follows a normal distribution with mean zero and variance σ^2. The GARCH(1,1) model for the conditional variance is given by:

$$\begin{aligned} \sigma_t^2 &= \omega + \alpha_1 \epsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2 \\ &= 0.000025 + 0.614701 \epsilon_{t-1}^2 + 0.384298 \sigma_{t-1}^2 \end{aligned} \tag{4.5}$$

This model provides insights into how volatility evolves over time, allowing us to forecast the future distribution of returns and thus the VaR. Using this GARCH(1,1) model, we calculate the 95% and 99% VaR:

$$1\% \, VaR: -2.83\%$$
$$5\% \, VaR: -1.67.\%$$

We also perform a rolling 1-step ahead forecast of the conditional standard deviation, $\hat{\sigma}_{t+1|t}$, using the *ugarchroll* method. This method allows for dynamic re-estimation of the GARCH parameters, and we re-estimate the model every 50 observations, setting the last 500 observations as the test set.

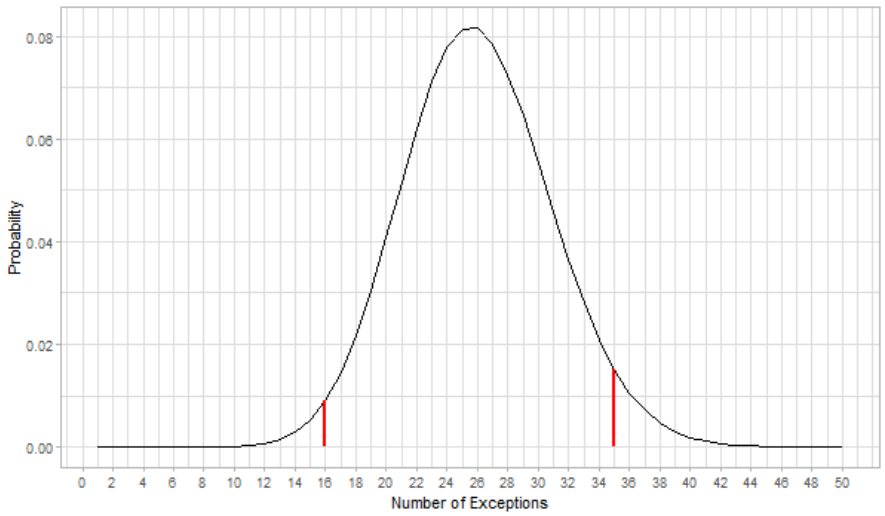


Figure 4.80: Distribution of probabilities for exceptions given by the binomial distribution.

Backtesting Let $N = \sum_{t=1}^{T} I_t$ be the number of days over a T period that stock returns were lower than the VaR estimate, where I_t is 1 if $y_{t+1} < VaR_{t+1|t}$ and 0 if $y_{t+1} \geq VaR_{t+1|t}$. Hence, N is the observed number of exceptions in the sample. As argued in Kupiec (1995) [10], the failure number follows a binomial distribution, $B(T, p)$.

Figure 4.80 shows the distribution of exceptions based on the binomial distribution. For a 5% VaR over 500 observations, we expect 25 exceptions. The red lines indicate the 95% confidence interval, with the lower bound at 16 and the upper bound at 35. If the actual number of exceptions falls within this range, it suggests that the GARCH model has accurately predicted the VaR.

VaR is a widely used measure of downside risk that estimates how much a portfolio might lose under normal market conditions within a set time period. Its three key components are: *a) time period*, *b) confidence level*, *c) loss amount (or loss percentage)*.

The VaR can also be estimated using historical data by simply calculating the 5% quantile value of the return distribution.

4.2.1 VaR Estimation Using Forecasting Simulation

In this analysis, we have 3020 trading days over a twelve-year period. We use 3000 days to train the GAN and then forecast the next day's returns, i.e., trading day 3001. Using the trained model, we generate 1000 next-day forecasts, from which we obtain a distribution of log returns and calculate the historical VaR.

Figure 4.81 shows the distribution of log returns generated for the next day (day 3001) based on 1000 simulations.

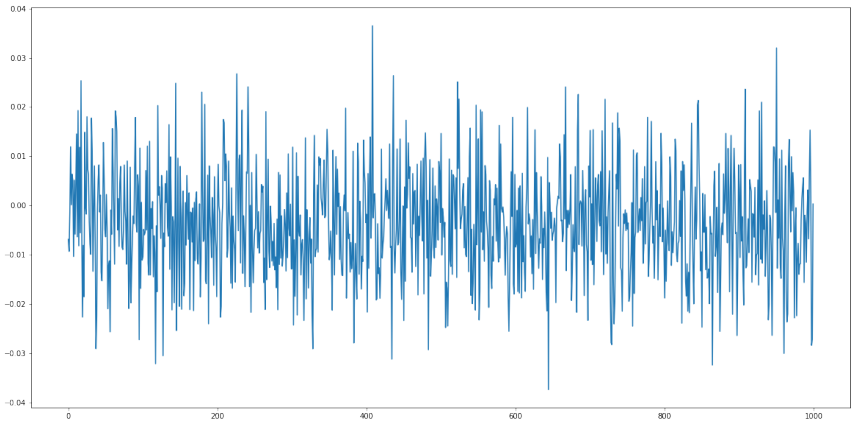


Figure 4.81: Log return distribution from 1000 simulations for trading day 3001.

In addition, Figure 4.82 provides the Kernel Density Estimate (KDE) for these forecasts, giving a smoothed representation of the distribution.

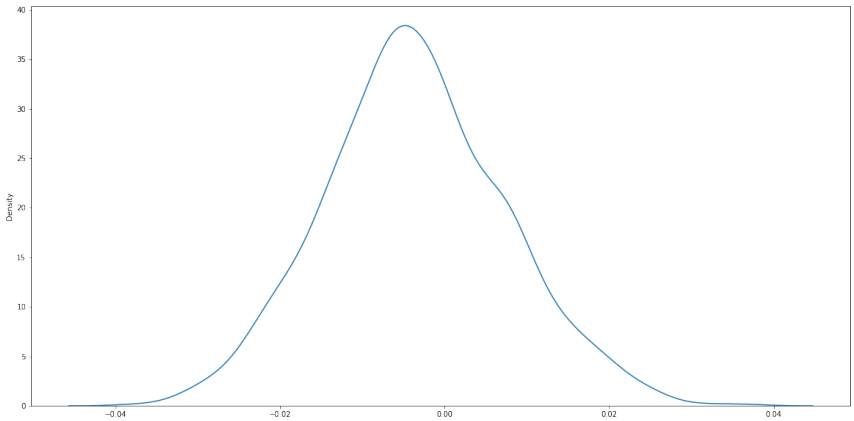


Figure 4.82: KDE of 1000 simulated log returns for trading day 3001.

From this distribution, we calculate the 1% and 5% daily VaR values for day 3001:

$$1\%\, Var: -2.83\%$$

$$5\%\, Var: -2.14.\%$$

The actual log return for day 3001 is -0.0085, which is higher than the calculated VaR values derived from the generated data. This means that the loss on day 3001 falls within the predicted risk thresholds. Additionally, the mean value of all forecasts is -0.0036, which represents the average expected log return from our 1000 simulations. This mean value is less negative than the actual return, suggesting that the overall forecast was for a smaller loss.

The VaR values are further illustrated in Figure 4.83, which shows the historical VaR at both the 95% and 99% confidence intervals based on the generated data.

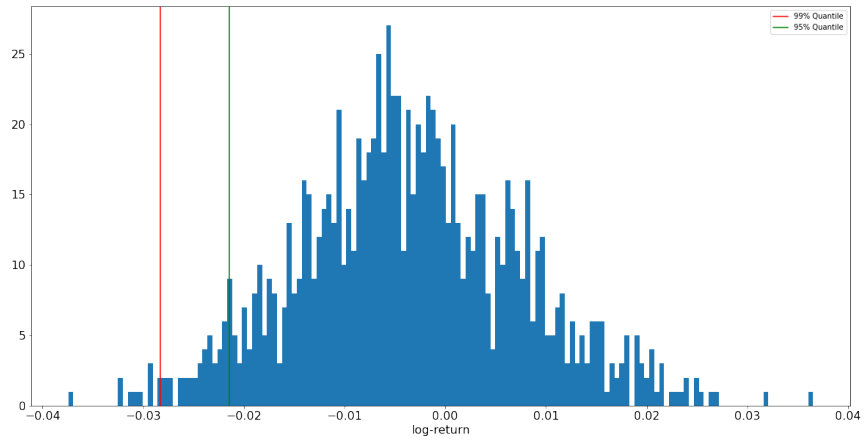


Figure 4.83: Historical VaR estimates at 95% and 99% confidence intervals based on generated data for trading day 3001.

To better understand the behavior of our forecasted data, we compare the distribution of the generated data with common probability distributions such as the normal, log-normal, beta, and t-distributions, as shown in Figure 4.84. It can be seen that the normal, log normal and beta distribution are close to model the underlying distribution. This is expected as our obtained distribution for the first day of forecasting is similar to the normal one. Still, the t-distribution, exhibits the best fit of our data, which can also be seen from the results presented in Table 4.13.

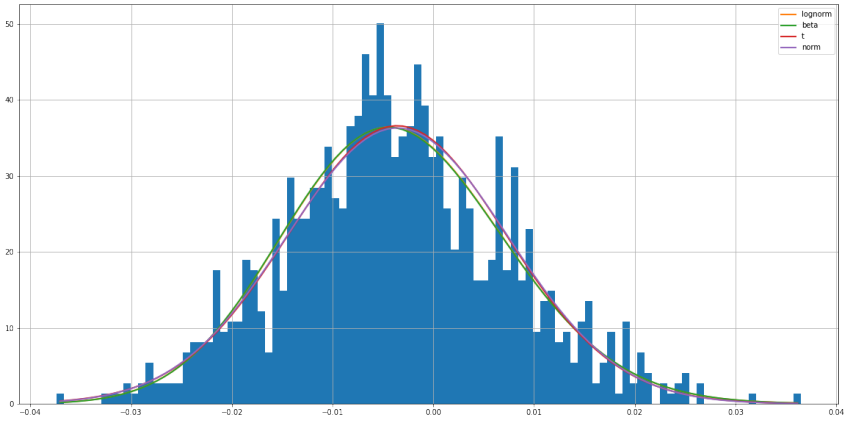


Figure 4.84: Comparison of the distribution of the generated data with t, beta, log-normal, and normal distributions.

Distributions	**Sum-Square Error**	**AIC**	**BIC**
t	1949.289	-334.073	688.188
beta	1950.722	-331.990	695.830
log normal	2001.885	-330.522	714.812
normal	2022.080	-329.122	717.942

Table 4.13: Results from comparing different distributions with the forecasted daily distribution.

Table 4.13 provides a comparison of the sum-square error, Akaike Information Criterion (AIC), and Bayesian Information Criterion (BIC) for each distribution. Lower values indicate a better fit with the t-distribution having the lowest sum-square error, AIC, and BIC, making it the best fit for our forecasted data. This information allows us to use the t-distribution in our VaR calculations, providing a more accurate estimation of potential losses. By understanding the distribution of our data, we can make more informed decisions and implement more effective risk management strategies.

Next, we forecast the log-return for the next day, i.e., day 3002, 1000 times, and obtain the results shown in Figure 4.85. As before, the obtained distribution from the forecasts for the next day is presented.

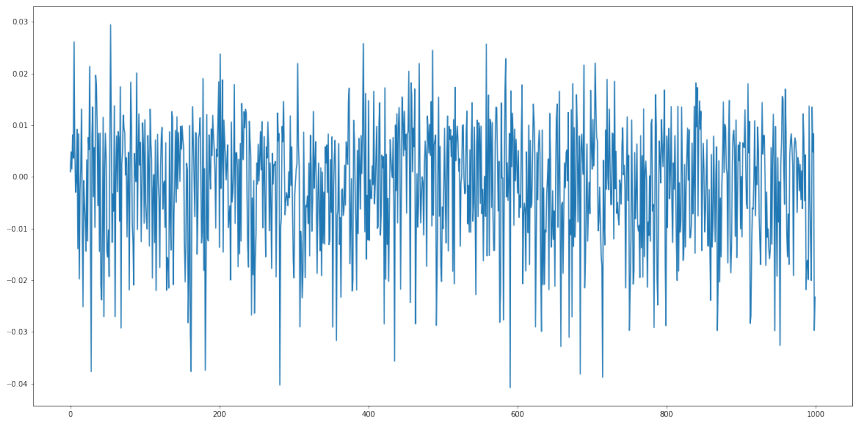


Figure 4.85: Log return distribution from 1000 simulations for trading day 3002.

Figure 4.86 shows the Kernel Density Estimate (KDE) of the forecasts we obtain from ForGAN for day 3002.

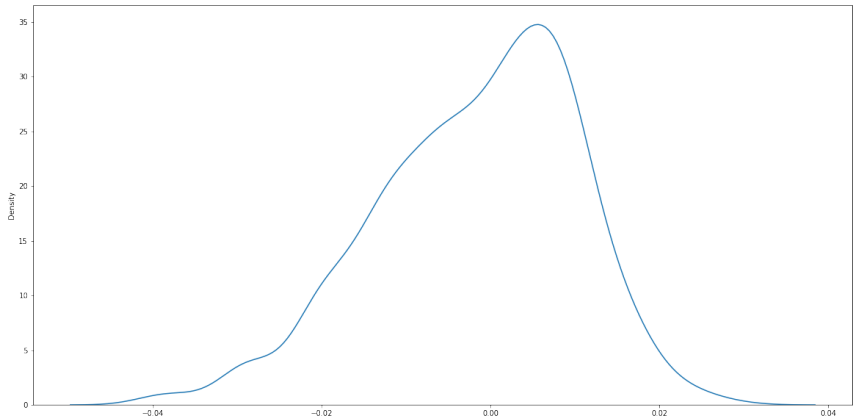


Figure 4.86: KDE of 1000 simulated log returns for trading day 3002.

From these forecasts, we calculate the 1% and 5% daily VaR as follows:

$$1\%\, Var: -3.17\%$$
$$5\%\, Var: -2.19.\%$$

The actual log-return for day 3002 is 0.01166, which exceeds the calculated VaR values. In this context, exceeds means that the actual return is higher (or less negative) than the VaR predictions, indicating that the actual return on this day was outside the risk thresholds we estimated from our simulations. Moreover, the mean value of all the forecasts is -0.00172, indicating a general expectation of a small negative return.

The calculated VaR values are plotted in Figure 4.87, which visually represents the historical VaR at both the 95% and 99% confidence intervals.

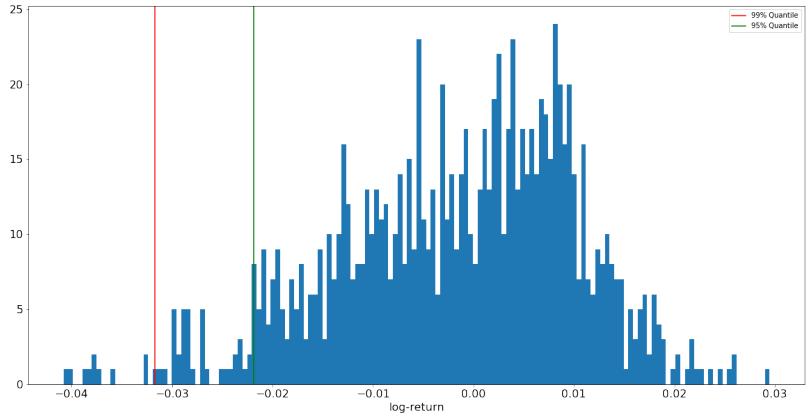


Figure 4.87: Historical VaR estimates at 95% and 99% confidence intervals based on generated data for trading day 3001.

We continue forecasting the next day's log-return, generating 1000 simulations per day, and calculating the Value-at-Risk (VaR) for a total of 20 consecutive trading days. In order to avoid a huge number of graphs we will just write the VaR values we get and the true log-increment value.

For the **third** day we get the following:

$$1\%\,\textit{Var:} -3.13\%$$
$$5\%\,\textit{Var:} -1.91\%$$
$$\textit{Real log-increment:} 0.02049.$$

The real log-increment exceeds the estimated VaR for both confidence levels.

For the **fourth** day we get the following:

$$1\%\,\textit{Var:} -4.05\%$$
$$5\%\,\textit{Var:} -2.00\%$$
$$\textit{Real log-increment:} 0.00308.$$

Again, the real log increment exceeds the VaR estimates.

For the **fifth** day we get the following:

$$1\%\,\textit{Var:} - 2.22\%$$
$$5\%\,\textit{Var:} - 1.60\%$$
$$\textit{Real log-increment:} - 0.00721.$$

Log-increment exceeds the value at risk estimate in this case.

For the **sixth** day we get the following:

$$1\%\,\textit{Var:} - 1.93\%$$
$$5\%\,\textit{Var:} - 1.17\%$$
$$\textit{Real log-increment:} 0.0095.$$

Log-increment exceeds the value at risk estimate in this case.

For the **seventh** day we get the following:

$$1\%\,\textit{Var:} - 1.95\%$$
$$5\%\,\textit{Var:} - 1.33\%$$
$$\textit{Real log-increment:} - 0.00918.$$

Log-increment exceeds the value at risk estimate in this case.

For the **eighth** day we get the following:

$$1\%\,\textit{Var:} - 3.17\%$$
$$5\%\,\textit{Var:} - 1.84\%$$
$$\textit{Real log-increment:} - 0.00745.$$

Log-increment exceeds the value at risk estimate in this case.

For the **ninth** day we get the following:

$$1\%\,\textit{Var:} - 2.99\%$$
$$5\%\,\textit{Var:} - 2.14\%$$
$$\textit{Real log-increment:} 0.0162.$$

Log-increment exceeds the value at risk estimate in this case.

For the **tenth** day we get the following:

$$1\%\,\textit{Var:} - 2.07\%$$
$$5\%\,\textit{Var:} - 1.30\%$$
$$\textit{Real log-increment:} - 0.00878.$$

Log-increment exceeds the value at risk estimate in this case.

For the **eleventh** day we get the following:

$$1\%\,\textit{Var:} - 2.72\%$$
$$5\%\,\textit{Var:} - 1.87\%$$
$$\textit{Real log-increment:} - 0.0103.$$

Log-increment exceeds the value at risk estimate in this case.

For the **twelfth** day we get the following:

$$1\%\,\textit{Var:} - 1.97\%$$
$$5\%\,\textit{Var:} - 1.31\%$$
$$\textit{Real log-increment:} - 0.0114.$$

Log-increment exceeds the value at risk estimate in this case.

For the **thirteenth** day we get the following:

$$1\%\,\textit{Var:} - 0.81\%$$
$$5\%\,\textit{Var:} - 0.63\%$$
$$\textit{Real log-increment:}\, 0.0176.$$

Log-increment exceeds the value at risk estimate in this case.

For the **fourteenth** day we get the following:

$$1\%\,\textit{Var:} - 3.24\%$$
$$5\%\,\textit{Var:} - 1.88\%$$
$$\textit{Real log-increment:}\, 0.0101.$$

Log-increment exceeds the value at risk estimate in this case.

For the **fifteenth** day we get the following:

$$1\%\,\textit{Var:} - 0.61\%$$
$$5\%\,\textit{Var:} - 0.07\%$$
$$\textit{Real log-increment:}\, 0.0062.$$

Log-increment exceeds the value at risk estimate in this case.

For the **sixteenth** day we get the following:

$$1\%\,\textit{Var:} - 2.09\%$$
$$5\%\,\textit{Var:} - 1.44\%$$
$$\textit{Real log-increment:}\, 0.0137.$$

Log-increment exceeds the value at risk estimate in this case.

For the **seventeenth** day we get the following:

$$1\%\,\textit{Var:} - 2.59\%$$
$$5\%\,\textit{Var:} - 1.60\%$$
$$\textit{Real log-increment:} - 0.0010.$$

Log-increment exceeds the value at risk estimate in this case.

For the **eighteenth** day we get the following:

$$1\%\,Var: -2.48\%$$
$$5\%\,Var: -1.51\%$$
$$Real\ log\text{-}increment: 0.00140.$$

Log-increment exceeds the value at risk estimate in this case.

For the **nineteenth** day we get the following:

$$1\%\,Var: -1.60\%$$
$$5\%\,Var: -1.05\%$$
$$Real\ log\text{-}increment: -0.00299.$$

Log-increment exceeds the value at risk estimate in this case.

For the **twentieth** day we get the following:

$$1\%\,Var: -1.49\%$$
$$5\%\,Var: -1.03\%$$
$$Real\ log\text{-}increment: -0.0026.$$

Log-increment exceeds the value at risk estimate in this case.

For 20 forecasted days, the actual log returns consistently exceed the calculated Value at Risk (VaR) estimates at both the 1% and 5% levels. This means that the actual returns were higher than the predicted risk thresholds in all these cases, indicating that the current VaR model may be overestimating the risk.

To improve the accuracy of our risk assessment, we plan to run additional experiments with a larger dataset. In this next experiment, we predict 15% of our data set, i.e., the last $N = 453$ days. The distribution of these log returns is shown in Figure 4.88, where we observe that the data is closest to the t-distribution.

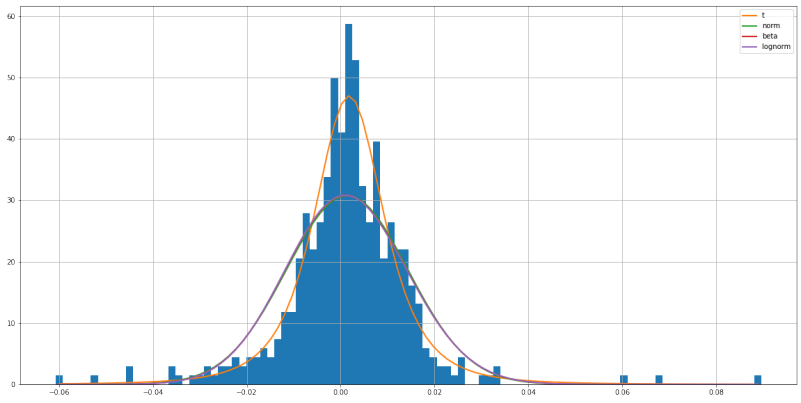


Figure 4.88: Comparison of the distribution of the historical log returns with t, beta, log-normal, and normal distributions for the last 453 days. The t-distribution provides the best fit to the data.

For each forecasted day we calculate the historical value at risk for 95% and 99% confidence intervals. An example for one of these forecasted days, trading day 2568, is presented in Figures 4.89 and 4.90.

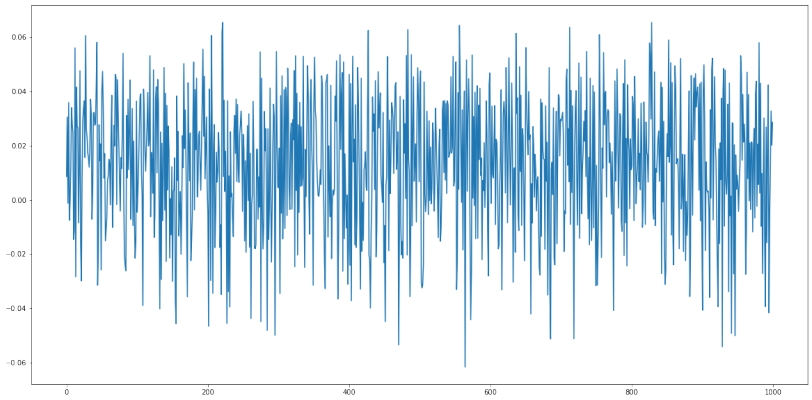


Figure 4.89: Log return distribution from 1000 simulations for the trading day 2568.

Figure 4.90 shows the KDE (Kernel Density Estimate) of the simulated log returns from ForGAN for trading day 2568. This provides a smoothed estimate of the distribution of the forecasted log returns.

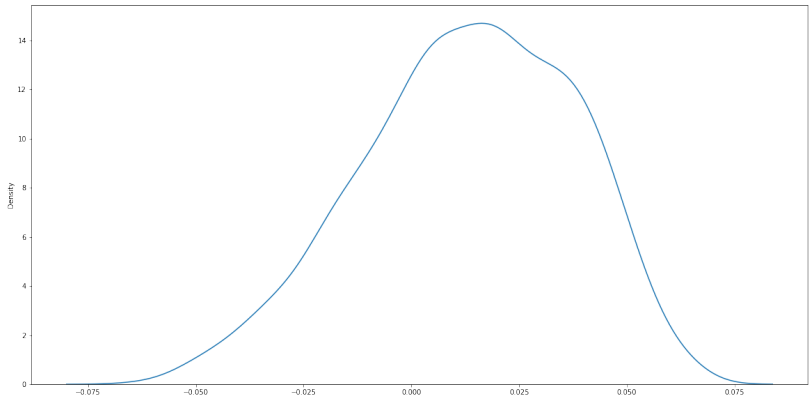


Figure 4.90: KDE of 1000 simulated log returns for trading day 2568.

The calculated VaR values for the trading day 2568 are as follows:

$$1\%\, Var: -0.0456$$
$$5\%\, Var: -0.0311.$$

The actual log-return for day 2568 is -0.0532, which is lower than the calculated VaR thresholds. This indicates that the loss on this day exceeded both the 1% and 5% risk estimates, suggesting that the risk was underestimated for this particular day. The mean of all forecasts for this day is 0.0130, indicating that the overall expectation was for a positive return. The VaR estimates based on the forecasted data are illustrated in Figure 4.91.

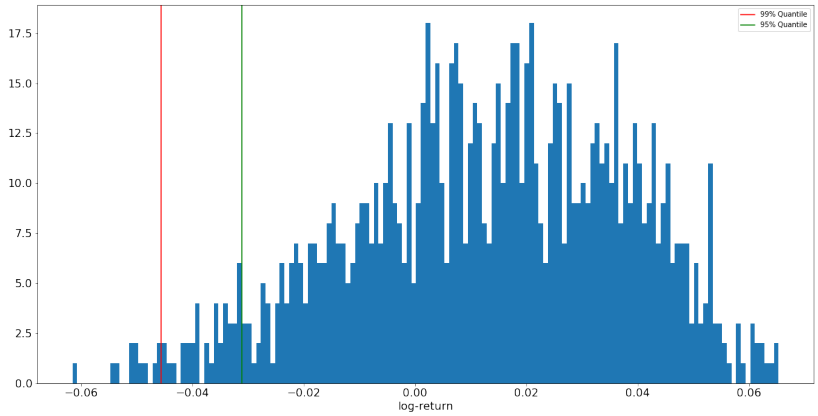


Figure 4.91: Historical value at risk for 95 % and 99 % confidence interval on generated data.

We repeat the same process for all **453** days and we get the following results. In the first case, for $\alpha = 0.05$ the real log-increment exceeds the value at risk estimate, $N_\alpha = 414$ times, which implies the following:

$$\frac{N_\alpha}{N} = \frac{414}{453} \approx 0.914.$$

In the second case, for $\alpha = 0.01$ the real log-increment exceeds the value at risk estimate, $N_\alpha = 432$ times, which implies the following:

$$\frac{N_\alpha}{N} = \frac{432}{453} \approx 0.954.$$

To further assess the model's robustness, we increased the number of forecasts per day from **1000** to **10000**. This change, however, did not significantly affect the results.

Over the same **453** days, for $\alpha = 0.05$, with **10000** forecasts per day, the real log-increment exceeds the value at risk estimate, $N_\alpha = 413$ times, which implies the following:

$$\frac{N_\alpha}{N} = \frac{413}{453} \approx 0.912.$$

Similarly, for $\alpha = 0.01$, with 10000 forecasts per day, the real log-increment exceeds the value at risk estimate, $N_\alpha = 432$ times, which implies the same as with 1000

forecasts per a day:

$$\frac{N_\alpha}{N} = \frac{432}{453} \approx 0.954.$$

Moreover, we transform our logarithmic data using the *Lambert W* function, which is a parametric non-linear transformation. It converts a random variable X into a so-called *Lambert W* random variable Y, which allows a very flexible approach to modeling skewed data [35]. Generally, Y is the heavy-tail version of the arbitrary random variable $X \sim F_X$. The tail behavior of the so-called heavy tail *Lambert* $W \times F_X$ random variable depends on a tail parameter $\delta \geq 0$ for $\delta = 0$, $Y \equiv X$, for $\delta > 0$ Y has heavier tails than X. *Lambert's W* function provides an explicit inverse transformation, which can be estimated by maximum likelihood. This inverse can remove heavy tails from data, and we use it for our *S & P 500* data set before estimating VaR with GANs as above.

After applying the **Lambert W** transformation, we analyze the Value at Risk (VaR) over **453** days: when $\alpha = 0.05$ the real log-increment exceeds the value at risk estimate, $N_\alpha = 412$ times, which translates to:

$$\frac{N_\alpha}{N} = \frac{412}{453} \approx 0.909.$$

For the second case, when $\alpha = 0.01$ the real log-increment exceeds the value at risk estimate, $N_\alpha = 425$ times, yielding:

$$\frac{N_\alpha}{N} = \frac{425}{453} \approx 0.938.$$

We compare the results of VaR forecasting using GANs with those obtained from a standard GARCH model.

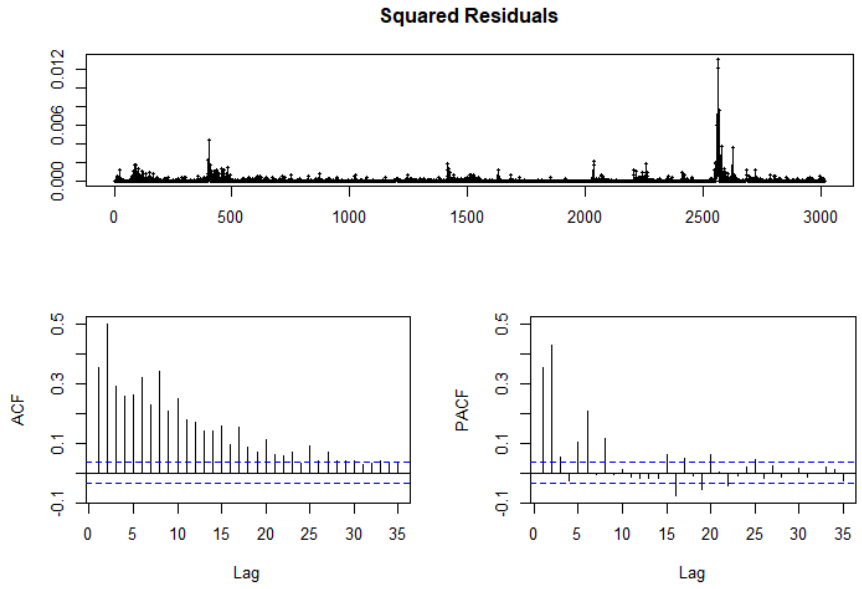


Figure 4.92: ACF and PACF of squared residuals from ARIMA(3,0,2) with non-zero mean, showing significant correlation.

By first identifying the most suitable autoregressive model using the Box-Jenkins approach, we found that the best-fitting model for our data is **ARIMA(3,0,2)** *with non-zero mean.*

Moreover, we performed diagnostics checks on the residuals of the model. GARCH models assume that the squared residuals are correlated, which is essential for the model's validity. To assess this, we examined the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) plots of the squared residuals. These plots, shown in Figure 4.92, clearly indicate significant correlation.

Using a standard GARCH(1,1) model with a constant mean for the generated log-returns, we obtained the following model coefficients:

	Estimate	Std. Error	t value	Pr(> |t|)
mu	0.0004	0.0001	3.1783	0.0015
omega	0.0000	0.0000	2.1358	0.0327
alpha1	0.1900	0.0114	16.7030	0.0000
beta1	0.7750	0.0083	93.8437	0.0000

Table 4.14: Estimated coefficients for sGARCH(1,1) model based on generated data.

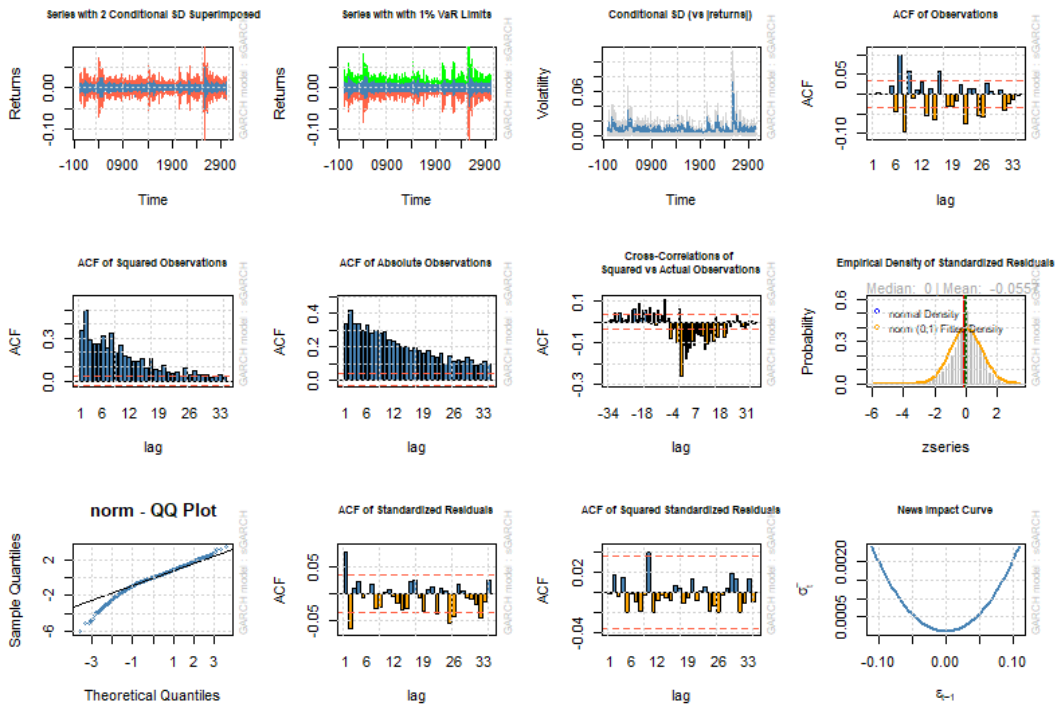


Figure 4.93: ACF and other diagnostic plots for sGARCH(1,1) residuals, confirming model adequacy.

Furthermore, we graph the auto-correlation function of the residuals, see Figure 4.93.

Our analysis indicates that the data closely follows a t-distribution, and we use its parameters to forecast Value at Risk (VaR). Similar to our GAN-based approach, we apply a rolling window technique to the last 453 observations of the dataset. This technique allowed us to perform rolling one-step-ahead forecasts of the conditional standard deviation, $\hat{\sigma}_{t+1|t}$, re-estimating the GARCH parameters every **50** observations to account for recent changes.

Figure 4.94 illustrates the forecasted 5% VaR from the *GARCH(1,1)* model. The black line represents the daily VaR forecasts, and the red points indicate the actual returns that fall below the forecasted VaR.

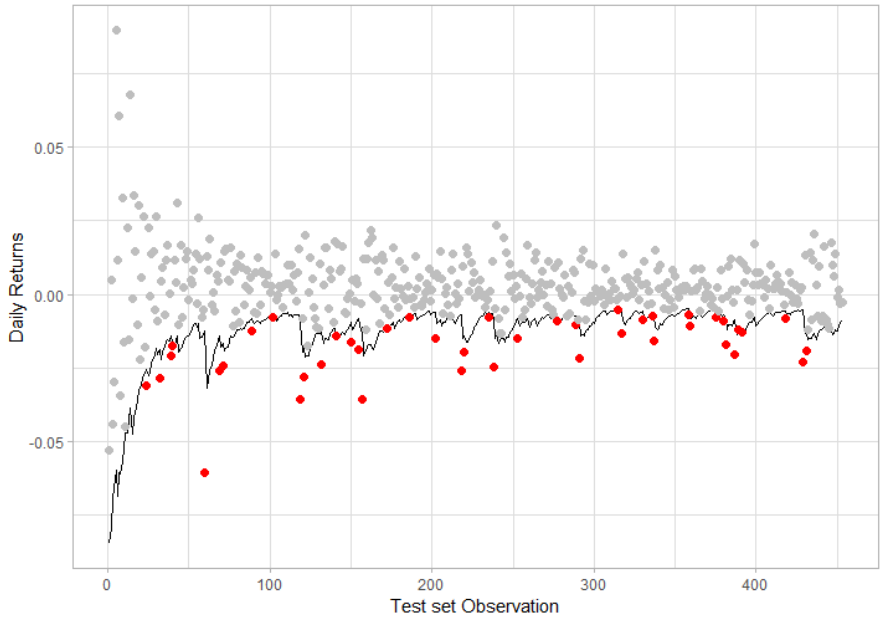


Figure 4.94: Forecasted 5% VaR from GARCH(1,1). Red points mark actual returns below the VaR threshold.

From the *GARCH(1,1)* model, we find that the real log-return exceeds the 5% VaR estimate **410** times out of **453** days. This results in an exceedance rate of:

$$\frac{N_\alpha}{N} = \frac{410}{453} \approx 0.905.$$

We apply the same procedure to forecast the $1\% VaR$ using the $GARCH(1,1)$ model. Since only the level of confidence changes, the ARIMA model and the GARCH's coefficients stay the same as those listed in 4.14. Figure 4.95 displays the daily forecasted 1% VaR from the standard $GARCH(1,1)$ model. The black line represents the forecasted 1% VaR, while the red points indicate the actual returns that fall below this threshold.

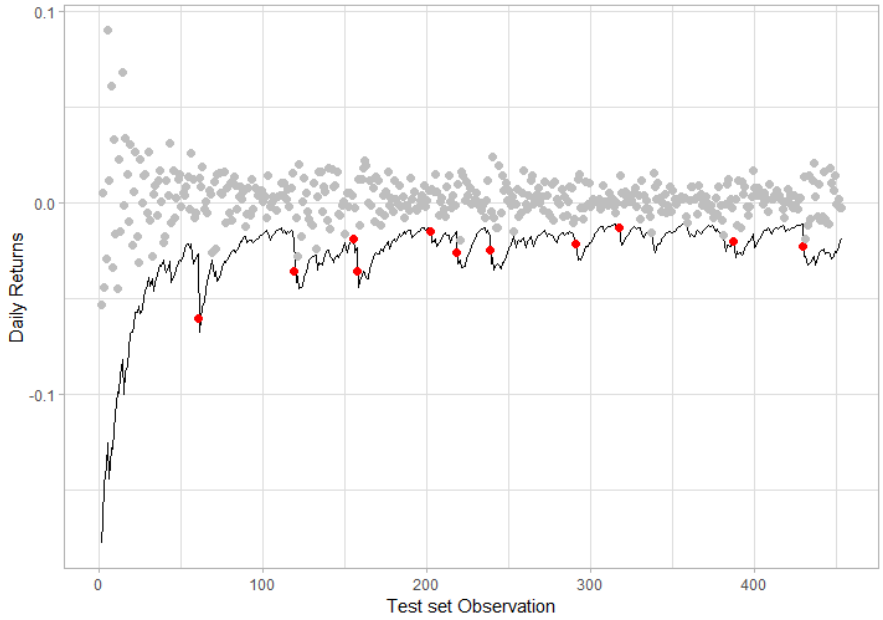


Figure 4.95: Forecasted 1% VaR from GARCH(1,1). Red points indicate actual returns below the VaR threshold.

From *GARCH(1,1)*, for $\alpha = 0.01$ the real log-increment exceeds the value at risk estimate, $N_\alpha = 442$ times, which implies the following:

$$\frac{N_\alpha}{N} = \frac{442}{453} \approx 0.976.$$

Next, we use the *GJR-GARCH* model with students t-distribution and we get the following results.

For *GJR-GARCH*, when $\alpha = 0.05$, the real log-increment exceeds the VaR estimate $N_\alpha = 408$ times, which implies the following:

$$\frac{N_\alpha}{N} = \frac{408}{453} \approx 0.901.$$

From GJR-GARCH, for $\alpha = 0.01$ the real log-increment exceeds the value at risk estimate, $N_\alpha = 444$ times, which implies the following:

$$\frac{N_\alpha}{N} = \frac{444}{453} \approx 0.98.$$

Summary and Implications

Both the *GARCH* and *GAN* models exhibit a tendency to underestimate Value at Risk (VaR), particularly at the extreme 1% confidence level. However, GAN models offer more accurate forecasts at the 5% level, providing a better reflection of potential extreme losses. Despite this improved accuracy, the GAN approach is computationally more complex and requires simulations and model retraining for each forecasted day, making it harder to implement compared to the more straightforward GARCH model.

Chapter 5

Portfolio Optimization with GANs

5.1 Introduction to Portfolio Theory

In a broad perception, any portfolio can be viewed as a set of distinct items. The portfolio's owner acquires these items based on their needs, wants, preferences, predispositions, possibilities, and expectations. From a financial perspective, such a portfolio consists of financial assets[1].The financial universe has two main underlying dimensions: time and risk.

Time is a fundamental dimension critical to all financial models, best explained by the concept of the time value of money (TVM). TVM is a financial principle stating that the present value of money is generally higher than its future value due to its potential to earn returns over time. Thus, one dollar today is typically worth more than one dollar a year from now because it can be invested to grow. However, if the interest rate is negative, the value of one dollar today might be higher than its future value due to the cost of holding or investing it. The value of one dollar invested today changes according to its interest rate. An interest rate reflects the relationship between cash flows occurring at different times, representing the return on investment or the cost of borrowing.

Considering the time dimension, we differentiate between discrete and continuous time. Consequently, interest rates can be categorized as discrete or continuous. TVM enables us to calculate either the future value (compounding) or the present value (discounting) of money.

Discrete Time

In a discrete-time setting, the values of financial assets are only observed at the discrete set of points in time. In this framework, interest is accrued to the principal[2]

[1]Financial asset - an investment asset whose value is derived from a contractual claim of what it represents.

[2]Principal - the amount of money initially invested

at these specific intervals. The interest rate itself can either be a random variable, assuming different values at each time point, or it can be a deterministic variable, with its future values known today. In the case of a random interest rate, it behaves like a discrete random variable, with different possible values at each interval.

- Discrete compounding
- the future value (FV) of one dollar invested for n years at the interest rate i compounded once per annum

$$FV = (1 + i)^n. \tag{5.1}$$

- FV of one dollar compounded m-times per annum for n years

$$FV = (1 + \frac{i}{m})^{mn}. \tag{5.2}$$

- Discrete discounting
- the present value (PV) of a future one dollar discounted for n years

$$PV = \frac{1}{(1 + i)^n}. \tag{5.3}$$

- PV of a future one dollar compounded m-times a year

$$PV = \frac{1}{(1 + \frac{i}{m})^{mn}}. \tag{5.4}$$

Continuous Time

In continuous time, the variable also occurs at specific points in time. For our analysis, we need to be able to observe the value of variables at every point in time. Nonetheless, in comparison to discrete time, between two points in time, there is an infinite number of other points of time, and the distance between them converges to zero.

The continuous time effect on compounding as $m \to \infty$ can be appropriately described using limits as follows:

$$\lim_{m \to \infty} \left(1 + \frac{i}{m}\right)^{mn} = e^{in}. \tag{5.5}$$

- Continuous compounding
- FV of one dollar continuously compounded for n years

$$FV = e^{in}. \tag{5.6}$$

- Continuous discounting
- PV of one dollar continuously compounded i

$$PV = e^{-in}. \tag{5.7}$$

Risk Dimension

The risk dimension is closely related to the time dimension. The essential property of the time dimension is its identification between two points in time: the present and the future. The present is known, but the future is uncertain and, therefore, risky. This uncertainty represents the potential deviation from our expectations about the future. These deviations can be positive or negative, but typically, only the negative deviations are considered undesirable.

The most prominent measure of risk in finance is the standard deviation (SD), which estimates the variability of returns for a given asset or investment approach. If an asset's value experiences large swings up and down (high volatility), it will have a higher standard deviation and, correspondingly, more risk than a stable asset whose value remains more constant. Other risk measures include Beta, semi-variance, value-at-risk (VaR), and expected shortfall (ES).

The concepts of utility functions, the expected utility criterion, and risk aversion play essential roles in the investment decision-making process.

Efficient Portfolios and an Optimal Portfolio

In constructing a portfolio, investors aim to maximize the expected return for a given level of risk they are willing to bear. Portfolios that achieve this goal are referred to as *efficient portfolios*.

To build an efficient portfolio, we need to make certain assumptions about investor behavior. A fair assumption is that investors are risk-averse. A *risk-averse investor*, when presented with two investments that have the same expected return but different levels of risk, will prefer the one with the lower risk. Among the efficient portfolios available, an *optimal portfolio* is the one that the investor prefers the most.

Utility Function

In economic theory, there are numerous scenarios where individuals and firms must make decisions. The "theory of choice" explains this decision-making process using a concept known as the utility function. A utility function is a mathematical way to quantify how an investor evaluates decisions and trade-offs by assigning values to different outcomes. It measures the degree of individual satisfaction or utility the investor receives from a specific level of wealth (W).

In general, utility functions have the following important properties:

- **Increasing Function of Wealth**: The utility function is an increasing function of wealth, meaning that more wealth leads to higher utility.
- **Decreasing Marginal Utility**: Marginal utility decreases as wealth increases. This is mathematically represented as:

$$u'(W_1) < u'(W_0)) \quad when \quad W_1 > W_0. \tag{5.8}$$

- **Concavity**: The utility function is concave, which reflects the investor's risk aversion. This property is expressed as:

$$U'' < 0. \tag{5.9}$$

  This is true assuming that the utility function U is twice continuously differentiable.

- **Individual Differences**: Different investors have different utility functions based on personal preferences and risk tolerances.

A utility function can be illustrated graphically by using an indifference curve. Figure 5.1 shows indifference curves u_1, u_2, and u_3, where the horizontal axis represents risk, while the vertical axis represents expected return. Each curve characterizes portfolios with different combinations of risk and return, all providing the same level of utility to the investor.

For instance, on curve u_1, points u and u' offer equal utility, even though u has a higher expected return and greater risk. The slope of the indifference curve indicates that investors require higher expected returns to accept higher risk.

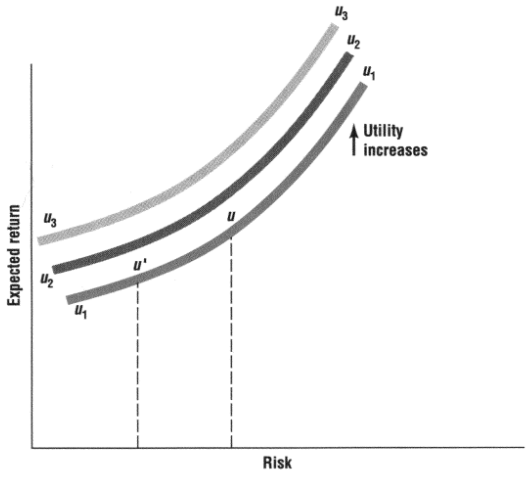


Figure 5.1: Indifference curves [97].

Commonly Used Utility Functions

There are several commonly used utility functions in finance, including:

- **Power Utility Function**: Commonly used to represent varying degrees of constant relative risk aversion.

$$u(W) = \frac{W^{1-\gamma}}{1-\gamma}, \quad \text{for} \quad \gamma > 0 \quad (\text{with} \quad \gamma \neq 1). \tag{5.10}$$

- **Quadratic Utility Function:** Often used for analytical tractability but can imply increasing absolute risk aversion, which is less realistic.

$$u(W) = W - \frac{b}{2}W^2, \quad b > 0. \tag{5.11}$$

- **Log Utility Function**: A special case of the power utility function when $\gamma = 1$, representing constant relative risk aversion (CRRA).

$$u(W) = ln(W). \tag{5.12}$$

- **Exponential Utility Function:** Represents constant absolute risk aversion (CARA).

$$u(W) = 1 - e^{-vW}. \tag{5.13}$$

Where W represents the investor's wealth. These utility functions help model investor behavior and are integral to various financial theories and applications, such as portfolio optimization and asset pricing

Expected Utility

The expected utility criterion is a fundamental hypothesis in mathematical economics concerning human preferences in decision-making under uncertainty. When faced with various possible outcomes from a decision made today, without knowing which outcome will occur, an individual will choose the one with the highest expected utility. Each outcome is given a probability of happening. Hence, the expected utility (EU) value is calculated by summing the products of each possible outcome's utility and its respective probability of occurrence. The decision to decide which action to take will also depend on the individual's risk aversion, as described below. To illustrate a general case, consider a lottery $L(x, y, \pi)$, where the outcome x has a probability of occurrence π and y has a probability of $(1-\pi)$. The expected utility is given by:

$$E[U(L)] = \pi u(x) + (1 - \pi)u(y). \tag{5.14}$$

The above equation is specified to as the **von Neumann-Morgernstern** (VNM) utility and represents the **expected utility criterion**. Let's adjust the VNM with respect to asset returns.

Consider an asset A at time t_0 with possible future returns $r_1, r_2, ..., r_n$ and the respective probabilities $\pi_{r_1}, \pi_{r_2}, ..., \pi_{r_n}$ at time t_1, respectively. Then, the expected utility is:

$$E[U(r_A)] = \pi_{r_1} u(r_1) + \pi_{r_2} u(r_2) + ... + \pi_{r_n} u(r_n) = \sum_{i=1}^{n} \pi_{r_i} u(r_i). \tag{5.15}$$

The expected utility criterion confirms that the individual is concerned with the utility of the final payoffs and the cumulative probability associated with achieving those utilities [28].

Risk Aversion

An essential feature of the risk dimension under the portfolio theory framework is the investor's relationship to risk. Every investor is distinct and has a diverse perception of risk, or in other words, a different degree of risk aversion. Risk aversion is the tendency to avoid risk. A risk-averse investor prefers a certain outcome equivalent to the expectation of a risky outcome over that risky outcome itself[33]. Loosely speaking, a risk-averse investor "penalizes" the expected rate of return of a risky portfolio by a certain percentage (or penalizes the expected profit by a dollar amount) to account for the risk involved. The greater the risk, the larger the penalty[37]. Naturally, the desire of every investor is to avoid unnecessary risks.

In investing, risk often equals price volatility. A volatile investment can lead to significant gains or substantial losses, while a conservative investment is expected to grow slowly and steadily over time. An investor is considered to be risk-averse if:

$$u(E[W]) \geq E[u(W)]. \tag{5.16}$$

This inequality indicates that the utility of the expected (certain) wealth $E[W]$ is higher than or equal to the expected utility of the uncertain wealth W.

The risk aversion coefficients measure the risk aversion:

- **Absolute risk aversion** coefficient

$$a(W) = -\frac{u''(W)}{u'(W)}. \tag{5.17}$$

- **Relative risk aversion** coefficient

$$\rho(W) = Wa(W) = -\frac{Wu''(W)}{u'(W)}. \tag{5.18}$$

- **Risk tolerance** coefficient

$$r(W) = \frac{1}{a(W)} = -\frac{u'(W)}{u''(W)}. \tag{5.19}$$

Utility functions incorporating the risk aversion coefficients:

- **Constant absolute risk aversion** (CARA)
  - Absolute risk aversion is constant at every wealth level
  - CARA is a function of the exponential utility function

$$u(W) = -e^{-aW}. \tag{5.20}$$

- **Constant relative risk aversion** (CRRA)
  - Relative risk aversion is constant at every wealth level

– CRRA is a function of the power utility function, which includes the logarithmic utility function as a special case when $\rho = 1$

$$u(W) = \frac{W^{1-\rho}}{1-\rho}. \tag{5.21}$$

Stock prices as Synonyms for Uncertainty Korn [42]. Stock prices are a prime example of uncertainty and unpredictability in daily life. Predicting stock prices accurately requires in-depth knowledge of a company's economic status, the markets it operates in, the behavior and sentiment of stock market participants, as well as political conditions, technological advancements, and various other factors. Due to the complexity of these influences, fundamentally modeling stock prices based on these factors is nearly impossible. As a result, low-dimensional parametric stochastic models serve as a practical alternative framework.

Markowitz's model is a one-period model, often referred to as a static model. In this model, decisions about investment strategies are made at the beginning of the period. The resulting consequences are then registered at the end of the period, with no additional reallocation of assets. Static models, like this one, are characterized by making decisions once, setting a strategy, and not engaging in further trading.

Diversification, investing in negatively correlated stocks, is essential for risk reduction. In this context, owning different financial assets is less risky than focusing on a single type. However, the accuracy of Markowitz's model depends on forecasted portfolio return and volatility. Rather than relying on a straightforward moving average forecast, Machine Learning can provide more accurate estimates. Assuming the underlying theory of Mean-Variance Optimization holds, utilizing Machine Learning (ML) can contribute to developing an optimal portfolio. This approach will be explored in more detail in the following section 5.4, and Chapter 5.3. Our main focus in the following part of the research is to use ML models for forecasting data to get more reliable estimates for Markowitz Portfolio Optimization. Note that the most crucial task of Mean-Variance Optimization is getting the parameters of securities, i.e., means, covariances of the returns of the stock prices.

Before we begin the discussion on the Markowitz theory, we state some assumptions for the market

- Markets are perfectly efficient, i.e., all information is transmitted perfectly, completely, instantly, and for no cost
- No transaction costs, no taxes
- Assets are perfectly divisible
- Investors are rational and risk-averse
- The market is liquid
- Investors are concerned only with mean returns and the variances and covariances of the returns

- Investors desire to maximize their expected utility, taking into account the trade-off between risk and return of their investment portfolios

Markowitz Mean-variance Model

Description of the Single-Period Market Models. To describe the Markowitz Mean-Variance Model, we primarily use Korn (2014) [42]. We consider a market where d different securities with prices $s_1, s_2, \ldots, s_d > 0$ are traded at the initial time $t = 0$. The security prices at the end time $t = T$, $S_1(T), S_2(T), \ldots, S_d(T)$, are unpredictable and are therefore modeled as non-negative random variables on a probability space $(\Omega, \mathcal{F}, P)$. In the following, the so-called *returns of the securities* are given by

$$R_i(T) := \frac{S_i(T) - s_i}{s_i}, \quad i = 1, \ldots, d, \tag{5.22}$$

i.e., their relative returns (including dividends), assuming that we know their expected values, variances, and covariances:

$$E(R_i(T)) = \mu_i \quad \text{for } i = 1, \ldots, d \tag{5.23}$$

$$\mathrm{Cov}(R_i(T), R_j(T)) = \sigma_{ij} \quad \text{for } i, j = 1, \ldots, d. \tag{5.24}$$

Remark 5.1.1. *The matrix $\sigma := (\sigma_{ij})_{i,j=1}^{d}$ is positive semi-definite as a variance-covariance matrix.*

Portfolio Weights and Return

Let $x > 0$ denote the initial wealth of an investor, which is distributed among d assets. The investor holds $\varphi_i \geq 0$ shares of the i-th security, and the total wealth invested across all assets is:

$$\sum_{i=1}^{d} \varphi_i s_i = x.$$

The portfolio weights π_i represent the proportion of total wealth allocated to each asset, are given by:

$$\pi_i = \frac{\varphi_i s_i}{x}, \quad i = 1, \ldots, d.$$

These weights satisfy:

$$\sum_{i=1}^{d} \pi_i = 1.$$

The portfolio return R^π is defined as the weighted sum of individual asset returns $R_i(T)$:

$$R^x = \sum_{i=1}^{d} \pi_i R_i(T).$$

Criteria for Portfolio Selection When choosing a portfolio, the primary goal is typically to maximize returns. However, focusing solely on the expected return can lead to investing entirely in the asset with the highest expected return, which could be very risky. Therefore, Markowitz proposed considering the variance of the portfolio return as an additional target criterion. Since the criteria of a high expected return and low variance of the portfolio return are generally conflicting, Markowitz's main idea is to either set a lower bound for the expected return (minimum return) and then select the portfolio vector with the minimal return variance, or to set an upper bound for the variance of the portfolio return and choose the portfolio vector with the highest expected portfolio return from the remaining options.

Markowitz diversification focuses on reducing portfolio risk by considering the covariance between asset returns. Instead of evaluating an asset's risk in isolation, this strategy combines assets with returns that are not perfectly correlated. By doing so, it aims to lower the overall portfolio variance without compromising returns

Problems in the Mean-Variance Approach

We now explicitly present the two variants of the mean-variance approach mentioned above:

1. Maximize the expected portfolio return $\mathbb{E}(R^\pi)$ with a given upper bound c_1 for the variance $\mathbb{V}\mathrm{ar}(R^\pi)$:

$$\begin{aligned} &\max_{\boldsymbol{\pi}\in\mathbb{R}^d} \mathbb{E}(R^\pi) \\ &\text{s.t.} \quad \pi_i \geq 0 \quad \text{for } i = 1, \ldots, d \quad \sum_{i=1}^{d} \pi_i = 1 \\ &\text{and} \quad \mathbb{V}\mathrm{ar}(R^\pi) \leq c_1. \end{aligned} \tag{5.25}$$

   Among all possible portfolios in $\pi \in \mathbb{R}^d$, only those that satisfy the constraints are considered, particularly those whose variance is below the given threshold. Among these, the portfolio that delivers the highest expected return is selected.

2. Minimize the variance of the portfolio return $\mathbb{V}\mathrm{ar}(R^\pi)$ with a given minimum expected return $\mathbb{E}(R^\pi) \geq c_2$:

$$\begin{aligned} &\min_{\boldsymbol{\pi}\in\mathbb{R}^d} \mathbb{V}\mathrm{ar}(R^\pi) \\ &\text{s.t.} \quad \pi_i \geq 0 \quad \text{for } i = 1, \ldots, d \quad \sum_{i=1}^{d} \pi_i = 1 \\ &\text{and} \quad \mathbb{E}(R^\pi) \geq c_2. \end{aligned} \tag{5.26}$$

Remark 5.1.2. *Both problems can be solved quickly and efficiently using commercial standard software. For the second version, even simple quadratic optimization*

methods are sufficient. The feasible region is non-empty if and only if there is at least one π that satisfies the constraints when

$$c_2 \leq \max_{1 \leq i \leq d} \mu_i. \tag{5.27}$$

If the matrix is even positive definite and the feasible region is non-empty, the problem has a unique solution. This also applies if, for example, the first security is risk-free (i.e., the price of the first security $S_1(T)$ is not random, as the first security represents cash or a savings account) and the variance-covariance matrix for the remaining securities is positive definite.

Effiecient Frontier Harry Markowitz (1959) in [3] introduced the concept of the efficient frontier in his work on Modern Portfolio Theory, which was an extension of his paper in 1952. Let's consider the case of two securities. For example, the investor could put 50% of his money in each company, or 25% in one company and 75% in the other, or 33% in one and 67% in the other. Without even considering a third security there are already an infinite number of possible portfolios that could be purchased. However, the investor needs to evaluate only a subset of the available portfolios due to the **efficient set theorem**, which states that: *An investor will choose his or her optimal portfolio from the set of portfolios that*

1. *Offer maximum expected return for a given level of risk*
2. *Offer minimum risk for a given level of expected return.*

The set of portfolios meeting these two conditions is known as the **efficient set** (also known as efficient frontier). A portfolio is *inneficient* if it is possible to obtain higher expected (or average) return with no greater variability or return, or obtain greater certainty of return with no less average or expected return [3].

By plotting each of the efficient portfolios on a chart, illustrating the relationship between expected return and the standard deviation of return (i.e., risk), the efficient frontier is obtained, see Figure 5.2. A rational investor will, of course, choose the portfolio that offers a higher expected rate of return for the same level of risk. An important point on the minimum-variance frontier is known as the global minimum-variance portfolio. This portfolio is characterized by having the lowest risk of all possible portfolios that can be created.

To give a mathematical foundation to the diversification effect described so far, we show in the following Theorem (see Korn(2014)[42]) that states that the standard deviation of the return of a portfolio of investment opportunities is always less than or equal to the weighted sum of the standard deviations of the individual investments, regardless of the form of the covariance matrix.

Theorem 5.1.3. *Let π be a portfolio vector with non-negative elements. Then for the standard deviation $\sqrt{\mathbb{V}ar(R^\pi)}$ of the portfolio return*

$$\sqrt{\mathbb{V}ar(R^\pi)} \leq \sum_{i=1}^{d} \pi_i \sqrt{\mathbb{V}ar(R_i)}. \tag{5.28}$$

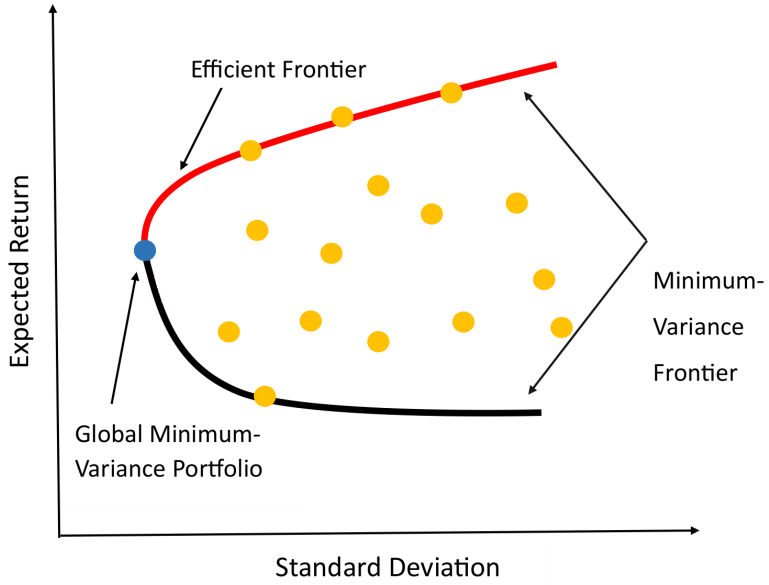


Figure 5.2: The Efficient Frontier.

5.2 Time-series Generative Adversarial Networks

TimeGAN is a novel generative model specifically designed for time-series data, combining elements of autoregressive models and Generative Adversarial Networks (GANs). While traditional autoregressive models focus on predicting the next step in a sequence, they are deterministic and not ideal for generating new data. GANs, on the other hand, excel at creating realistic samples but often overlook the unique temporal dependencies of time-series data.

TimeGAN bridges this gap by introducing a dual training approach: it uses an adversarial loss to ensure the realism of the generated sequences and a supervised stepwise loss to accurately capture the conditional distributions between time steps. This model also incorporates an embedding network, which maps high-dimensional data into a lower-dimensional latent space, making the learning process more efficient and better at preserving temporal dynamics.

TimeGAN consists of four main network components: an embedding function, a recovery function, a sequence generator, and a sequence discriminator. The key innovation of TimeGAN is its approach to training. Specifically, the autoencoding components (the embedding and recovery functions) are trained simultaneously with the adversarial components (the generator and discriminator). This simultaneous training enables TimeGAN to learn feature encoding, generate representa-

tions, and capture temporal dynamics effectively. The embedding network creates a latent space where the adversarial network operates, and the supervised loss ensures that the latent dynamics of both real and synthetic data are synchronized.

Importantly, TimeGAN effectively captures correlations within time-series data, including the interdependencies between different assets. By combining adversarial learning with a supervised approach, TimeGAN models the data distribution to reflect these complex relationships accurately. The embedding network aids in identifying and simplifying these correlations, optimizing the learning process.

In TimeGAN, there are three key types of losses [87]:

1. **Reconstruction Loss:** This loss is associated with the auto-encoder components (the embedder and recovery functions). It essentially measures how well the encoded data can be reconstructed compared to the original data.
2. **Supervised Loss:** This loss evaluates the generator's performance in predicting the next time step within the latent space, ensuring that the generated sequences closely follow the true temporal progression.
3. **Unsupervised Loss:** This loss reflects the adversarial relationship between the generator and discriminator networks. It represents the min-max game where the generator tries to produce realistic data to fool the discriminator, while the discriminator aims to differentiate between real and generated data.

In contrast to other types of GANs, which typically rely solely on unsupervised learning, TimeGAN's inclusion of reconstruction and supervised losses enables it to capture more complex temporal dynamics and correlations within time-series data.

5.2.1 Embedding and Recovery Functions

The embedding and recovery functions create mappings between feature and latent spaces, enabling the adversarial network to learn underlying temporal dynamics through lower-dimensional representations. Let $\mathcal{H}_S, \mathcal{H}_X$ denote the latent vector spaces for feature spaces S, X. The embedding function $e : S \times \mathcal{Q}_t X \to \mathcal{H}_S \times \mathcal{Q}_t \mathcal{H}_X$ transforms static and temporal features to latent codes $h_S, h_{1:T} = e(s, x_{1:T})$, implemented via a recurrent network:

$$h_S = e_S(s), \quad h_t = e_X(h_S, h_{t-1}, x_t).$$

Here, $e_S : S \to \mathcal{H}_S$ handles static features, and $e_X : \mathcal{H}_S \times \mathcal{H}_X \times X \to \mathcal{H}_X$ handles temporal features.

The recovery function $r : \mathcal{H}_S \times \mathcal{Q}_t \mathcal{H}_X \to S \times \mathcal{Q}_t X$ converts latent codes back to feature representations $\tilde{s}, \tilde{x}_{1:T} = r(h_S, h_{1:T})$ implemented via a feedforward network:

$$\tilde{s} = r_S(h_S), \quad \tilde{x}_t = r_X(h_t).$$

where $r_S : \mathcal{H}_S \to S$ and $r_X : \mathcal{H}_X \to X$ recover static and temporal embeddings.

5.2.2 Sequence Generator and Discriminator

The generator outputs into the embedding space, defined by vector spaces $\mathcal{Z}_S, \mathcal{Z}_X$. The generating function $g : \mathcal{Z}_S \times \mathcal{Q}_t\mathcal{Z}_X \to \mathcal{H}_S \times \mathcal{Q}_t\mathcal{H}_X$ uses random vectors to create synthetic latent codes $\hat{h}_S, \hat{h}_{1:T} = g(z_S, z_{1:T})$ implemented with a recurrent network:

$$\hat{h}_S = g_S(z_S), \quad \hat{h}_t = g_X(\hat{h}_S, \hat{h}_{t-1}, z_t)$$

where $g_S : \mathcal{Z}_S \to \mathcal{H}_S$ is for static features, and $g_X : \mathcal{H}_S \times \mathcal{H}_X \times \mathcal{Z}_X \to \mathcal{H}_X$ is for temporal features. Random vectors are sampled from chosen distributions, such as Gaussian and Wiener processes. The discriminator operates in the embedding space, using the function $d : \mathcal{H}_S \times \prod_t \mathcal{H}_X \to [0,1] \times \prod_t [0,1]$. This function takes static and temporal codes and outputs classifications $\tilde{y}_S$ and $\tilde{y}_{1:T}$ for real (h^*) or synthetic ($\hat{h}^*$) embeddings. It is implemented via a bidirectional recurrent network with a feedforward output layer, where:

$$\tilde{y}_S = d_S(\tilde{h}_S) \quad \text{and} \quad \tilde{y}_t = d_X(\overleftarrow{u}_t, \overrightarrow{u}_t).$$

The forward ($\overrightarrow{u}_t$) and backward ($\overleftarrow{u}_t$) hidden states are managed by recurrent functions $\overrightarrow{c}_X$ and $\overleftarrow{c}_X$.

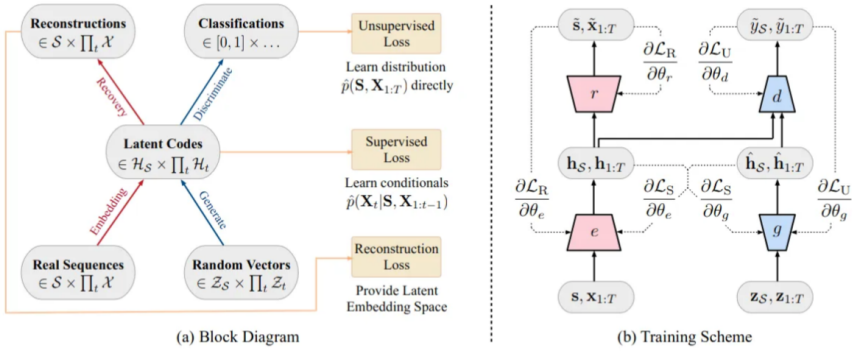


Figure 5.3: TimeGAN block diagram and training scheme [83].

5.2.3 Reconstruction and Loss Functions

The embedding and recovery functions in TimeGAN enable accurate reconstruction of the original data $s, x_{1:T}$ from their latent representations $h_S, h_{1:T}$. The primary goal is to minimize the *reconstruction loss* L_R, which ensures proper mapping between feature and latent spaces:

$$L_R = \mathbb{E}_{s,x_{1:T}\sim p}\left[\|s - \tilde{s}\|_2 + \sum_t \|x_t - \tilde{x}_t\|_2\right].$$

During training, the generator operates in two modes. In open-loop mode, it generates synthetic embeddings $\hat{h}_S, \hat{h}_{1:t-1}$ to produce the next synthetic vector $\hat{h}_t$, optimizing the *unsupervised loss* L_U:

$$L_U = \mathbb{E}_{s,x_{1:T}\sim p}\left[\log y_S + \sum_t \log y_t\right] + \mathbb{E}_{s,x_{1:T}\sim \hat{p}}\left[\log(1-\hat{y}_S) + \sum_t \log(1-\hat{y}_t)\right].$$

In closed-loop mode, the generator uses actual data embeddings $h_{1:t-1}$ to predict the next latent vector, optimizing the *supervised loss* L_S:

$$L_S = \mathbb{E}_{s,x_{1:T}\sim p}\left[\sum_t \|h_t - g_X(h_S, h_{t-1}, z_t)\|^2\right].$$

The **optimization** involves minimizing the combined reconstruction and supervised losses $\lambda L_S + L_R$ for the embedding and recovery functions, and training the generator and discriminator adversarially by minimizing $\eta L_S + \max_{\theta_d} L_U$. Here, λ and η are hyperparameters that balance the losses. This approach ensures that TimeGAN effectively learns to encode, generate, and capture temporal dynamics in the data.

5.3 Principal Component Analysis

Principal Component Analysis (PCA) is a statistical approach typically used to deduce the dimensionality of a data set while keeping the variation present in the data set as much as possible. Its primary goal is to transform a dataset containing numerous interrelated variables into a new set of uncorrelated variables known as principal components. These components are ordered sequentially, with the first component explaining the maximum variation present in the data. Each principal component is formed by combining the original variables linearly, where the coefficients indicate the relative importance of each variable within the component.

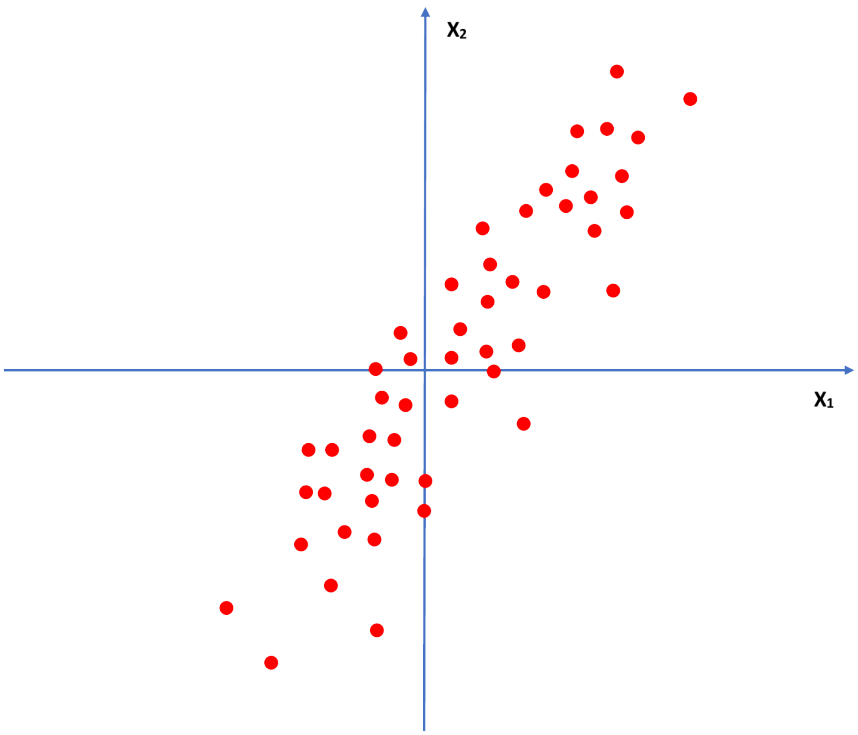


Figure 5.4: Plot of 50 observations on two variables x_1,x_2.

Assume that x represents a vector of p random variables. The objective is on understanding the variances of these p random variables and the patterns of covariances or correlations among them. Unless the variable p is small or the structure is quite simple, only studying the p variances and the totality of the $1/2 * p * (p-1)$ correlations or covariances might not be particularly informative. Nonetheless, an alternative strategy involves seeking a limited number ($<< p$) of derived variables that retain a substantial portion of the information these variances and correlations or covariances give.

While PCA considers covariances and correlations, its primary focus is on variances. The initial step involves identifying a linear function $\alpha_1' x$ of the elements of x that maximizes variance. Here, α_1 is represented as a vector with constants $\alpha_{11}, \alpha_{12}, ..., \alpha_{1p}$, and $'$ means transposed, such that

$$\alpha_1' x = \alpha_{11} x_1 + \alpha_{12} x_2 + ... + \alpha_{1p} x_p = \sum_{j=1}^{p} \alpha_{1j} x_j. \tag{5.29}$$

Subsequently, the process involves identifying a linear function $\alpha_2' x$ that is uncorrelated with $\alpha_1' x$ and has maximum variance. This process is repeated iteratively, seeking a linear function $\alpha_k' x$ at the $k-th$ stage, characterized by maximum variance while remaining uncorrelated with $\alpha_1' x, \alpha_2' x, ..., \alpha_{k-1}' x$. The derived variable $\alpha_k x$, identified at the $k-th$ stage, is called the $k-th$ Principal Component (PC).

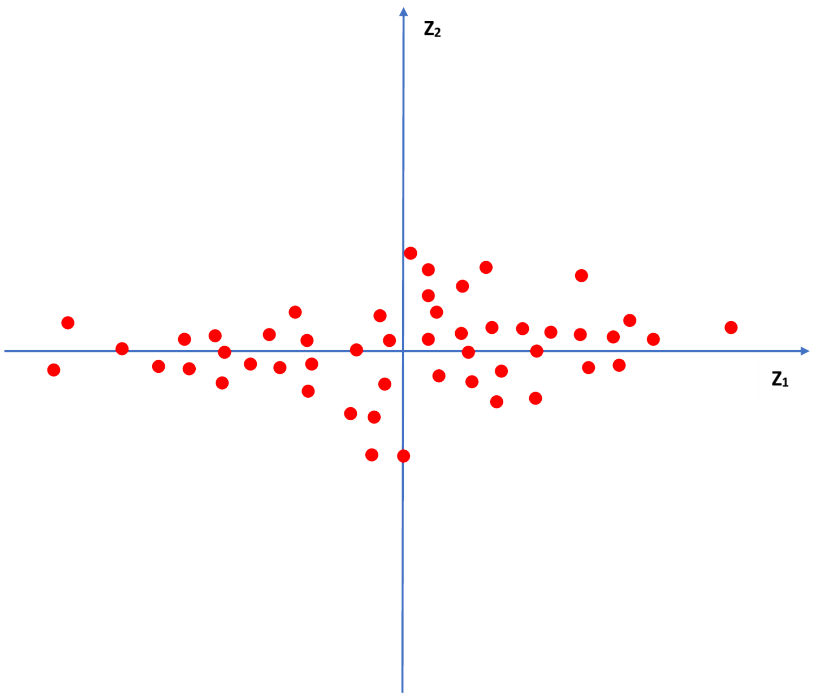


Figure 5.5: Plot of the 50 observations from Figure 1.1 with respect to their PCs z_1, z_2.

While it's possible to identify up to p PCs, the general expectation is that a smaller number of PCs. typically denoted as m, where $m << p$, can explain most of the variation in x.
Analyzing the scenario where $p = 2$ can be particularly beneficial as it allows for accurate representation and visualization of the data in two dimensions.

Figure 5.4 illustrates a plot of 50 observations on two highly correlated variables, x_1 and x_2. Both variables show significant variation, with x_2 indicating more variation than x_1. Upon transformation into PCs z_1 and z_2, the resulting plot is given in Figure 5.5.

It is clear that there is a more significant variation in the direction of z_1 than in either of the original variables, while there is minimal variation in the direction of z_2. In a broader sense, when a set of $p(> 2)$ variables shows considerable correlations, the initial PCs tend to capture the majority of the variation in the original variables. Conversely, the final PCs identify directions with minimal variation, identifying near-constant linear relationships among the original variables. The question now is how do we find the PCs? Let's initially consider the scenario where the vector of random variables x has a known covariance matrix Σ. This matrix has elements (i, j) representing the (known) covariance between the $i-th$ and $j-th$ component of x when $i \neq j$, and the variance of the $j-th$ element of x when $i = j$. For each $k = 1, 2, .., p$ the $k-th$ PC, denotes as z_k, can be expressed as $z_k = \alpha_k' x$, where α_k is an eigenvector of Σ corresponding to its $k-th$ largest eigenvalue λ_k. Additionally, if α_k is normalized to have unit length ($\alpha_k'\alpha_k = 1$), then the variance of z_k, denoted as $var(z_k) = \lambda_k$. The derivation of these results is given in the book by I.T. Jolliffe called *Principal Component Analysis* [24].

To derive the form of the PCs, let's start with $\alpha_1' x$. The vector α_1 maximizes $var[\alpha_1' x] = \alpha_1'\Sigma\alpha_1$. However, achieving this maximum needs imposing a normalization constraint. In this derivation, the constraint is set as $\alpha_1'\alpha_1 = 1$, indicating that the sum of squares of elements in α_1 equals 1. While other constraints, like $Max_j|\alpha_{1j}| = 1$, might be more suitable in different circumstances and can be substituted later on. Nonetheless, using constraints other than $\alpha_1'\alpha_1 = constant$ in the derivation complicates the optimization problem, creating a set of derived variables different from the PCs. To maximize $\alpha_1'\Sigma\alpha_1$ with the constraint $\alpha_1'\alpha_1 = 1$, the standard approach involves using the Lagrange multipliers technique. This implicates maximizing

$$\alpha_1'\Sigma\alpha_1 - \lambda(\alpha_1'\alpha_1 - 1),$$

where λ is a Lagrange multiplier. The differentiation with respect to α_1 gives

$$\Sigma\alpha_1 - \lambda\alpha_1 = 0,$$

or

$$(\Sigma - \lambda I_p)\alpha_1 = 0,$$

where I_p is the (pxp) identity matrix. Thus, λ is an eigenvalue of Σ and α_1 is the corresponding eigenvector. To identify the eigenvector giving $\alpha_1' x$ with maximum variance, maximize

$$\alpha_1'\Sigma\alpha_1 = \alpha_1'\lambda\alpha_1 = \lambda\alpha_1'\alpha_1 = \lambda$$

ensuring that λ must be as large as possible. Therefore, λ_1 is the eigenvector corresponding to the largest eigenvalue of Σ, and $var(\alpha_1' x) = \alpha_1' \Sigma \alpha_1 = \lambda_1$, the largest eigenvalue. In general, the $k-th$ PC of x is $\alpha_k' x$, and $var(\alpha_k' x) = \lambda_k$, where λ_k is the $k-th$ largest eigenvalue of Σ, and α_k is the corresponding eigenvector. This holds true for $k = 2$, and the proof for $k \geq 3$ is slightly more intricate but follows a similar logic.

The second PC, $\alpha_2' x$, maximizes $\alpha_2' \Sigma \alpha_2$ while being uncorrelated with $\alpha_1' x$, expressed as $cov[\alpha_1' x, \alpha_2' x] = 0$. The covariance is given by

$$cov[\alpha_1' x, \alpha_2' x] = \alpha_1' \Sigma \alpha_2 = \alpha_2' \Sigma \alpha_1 = \alpha_2' \lambda_1 \alpha_1 = \lambda_1 \alpha_2' \alpha_1 = \lambda_1 \alpha_1' \alpha_2.$$

Consequently, any of the equations

$$\alpha_1' \Sigma \alpha_2 = 0, \quad \alpha_2' \Sigma \alpha_1 = 0$$

$$\alpha_1' \alpha_2 = 0 \quad \alpha_2' \alpha_1 = 0$$

could be used to specify zero correlation between $\alpha_1' x$ and $\alpha_2' x$. Choosing the last of these equations (an arbitrary choice) and recognizing the need for a normalization constraint, the quantity to be maximized is

$$\alpha_2' \Sigma \alpha_2 - \lambda(\alpha_2' \alpha_2 - 1) - \phi \alpha_2' \alpha_1,$$

where λ and ϕ are Lagrange multipliers. Differentiating with respect to α_2 gives

$$\Sigma \alpha_2 - \lambda \alpha_2 - \phi \alpha_1 = 0,$$

and multiplying this equation on the left by α_1 results in

$$\alpha_1' \Sigma \alpha_2 - \lambda \alpha_1' \alpha_2 - \phi \alpha_1' \alpha_1 = 0.$$

Since the first two terms are zero, and $\alpha_1' \alpha_1 = 1$, this reduces to $\phi = 0$. Consequently, $\Sigma \alpha_2 - \lambda \alpha_2 = 0$, or equivalently $(\Sigma - \lambda I_p)\alpha_2 = 0$, indicating that λ is once more an eigenvalue of Σ, and α_2 is the corresponding eigenvector.

Once again, $\lambda = \alpha_2' \Sigma \alpha_2$, and it must be as large as possible. Assuming Σ does not have repeated eigenvalues, λ cannot equal λ_1. If it did, α_2 would be equal to α_1, violating the constraint $\alpha_1' \alpha_2 = 0$. Therefore, λ is the second largest eigenvalue of Σ, and α_2 is the corresponding eigenvector.

As mentioned earlier, it can be demonstrated that for the third, fourth, ..., $p-th$ PCs the vectors of coefficients $\alpha_3, \alpha_4, ..., \alpha_p$ are the eigenvectors of Σ corresponding to $\lambda_3, \lambda_4, ..., \lambda_p$, the third and fourth largest, ..., and the smallest eigenvalue, respectively. Furthermore, $\text{var}[\alpha_k' x] = \lambda_k$ for $k = 1, 2, ..., p$.

The derivation of Principal Component (PC) coefficients and variances as eigenvectors and eigenvalues of a covariance matrix is standard. However, alternative derivations that do not involve differentiation are presented in Flury (Section 2.2) [7] and Diamantaras and Kung (Chapter 3)) [12].

5.4 TimeGAN and ForGAN-PCA for Portfolio Optimization

As discussed earlier in section 5.1, a crucial task of portfolio optimization is getting the parameters of the securities, i.e., the means, and covariances of the return of the stock prices, especially when dealing with a larger number of assets. Nevertheless, in many cases, we do not have enough data points to get reliable estimates. As demonstrated in Chapter 4, Generative Adversarial Networks (GANs) are a good method for generating synthetic data, which can help overcome this limitation by providing additional data points.

In this first application of GANs for portfolio optimization using the Markowitz Mean-Variance model, we focus on historical time series of stock prices. The idea is to generate synthetic data by training a GAN model on the log-increments (logarithmic returns) of the stock prices. Once the GAN is trained, we can generate additional log increments, effectively expanding the dataset. With this larger dataset, we aim to estimate the parameters (mean returns and covariances) more effectively, which can improve the reliability of constructing an efficient portfolio.

The correlations between stocks play a crucial role in portfolio diversification, and thus, capturing this structure is essential for accurate parameter estimation. To maintain the correlation structure among the assets, we apply Principal Component Analysis (PCA) to the log-increments. PCA transforms the correlated stock returns into a set of orthogonal (uncorrelated) principal components. This transformation allows us to generate additional data for each principal component independently using GANs. Afterward, we can combine the generated components to reconstruct the full dataset, ensuring that the original correlation structure between the assets is preserved in the expanded data set.

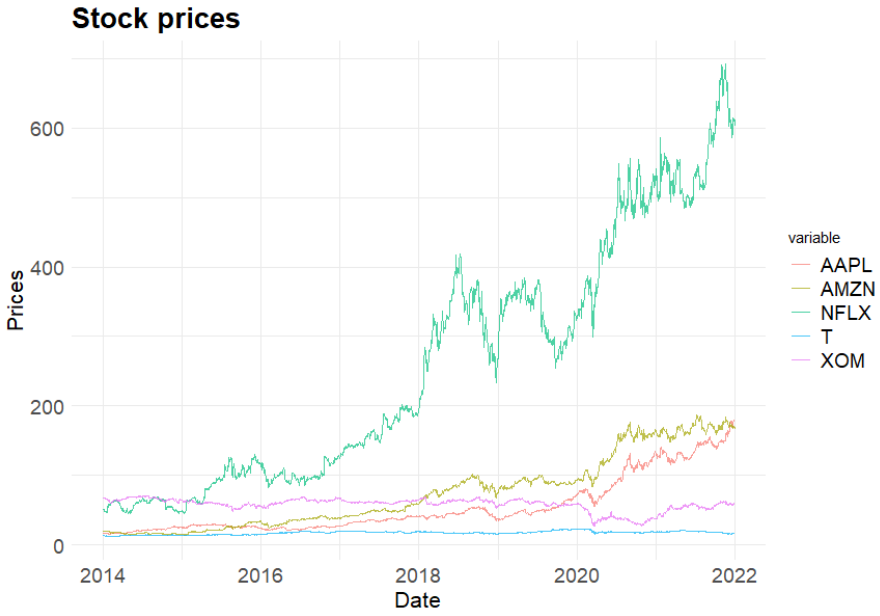


Figure 5.6: Stock prices for Apple, Netflix, Amazon, Exon Mobil Corporation, and AT & T Corporation.

For this analysis, we use historical stock prices from the period 2014-2022, focusing on five prominent companies: Apple, Netflix, Amazon, Exxon Mobil Corporation, and AT & T Corporation (American Telephone and Telegraph Company). The stock prices of these companies are shown in Figure 5.6.

For our analyses, we use log return transformation for our data since log returns are much more likely to be normally distributed and independent as a time series than prices or simple returns. See Figure 5.7.

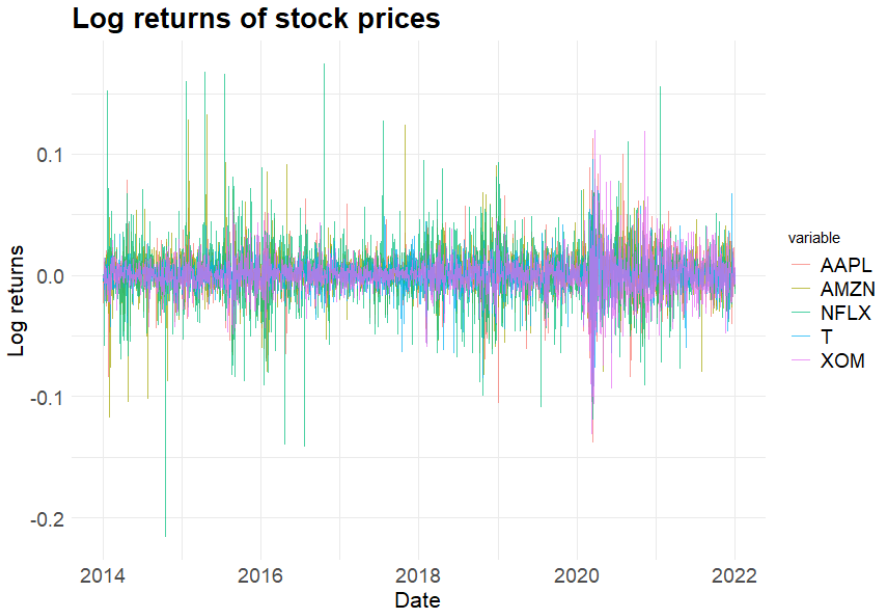


Figure 5.7: Log returns of stock prices.

Our goal is to obtain the parameters of the securities, specifically the means and covariances of stock price returns. This is generally a difficult task due to the often limited availability of data points, which hinders the derivation of reliable estimates. To address this challenge, we will use ForGAN to generate additional data for our log returns, thereby enabling more accurate parameter estimation. We have chosen ForGAN due to its architecture, which conveniently generates data for a specific future period, such as the next day. This capability aligns perfectly with our portfolio optimization process, where decisions are made on a daily basis. With ForGAN, historical data, denoted as $x_0, x_1, ..., x_t$, is used as a condition input for both the discriminator and the generator. The generator also takes a random noise vector, in our case a Gaussian distribution with mean zero and standard deviation one, and returns the forecast x_{t+1} based on the condition c, i.e., the actual data. Then, the discriminator outputs a probability score between 0 and 1, indicating the likelihood that the input data is real.

We will focus on the log returns of three stocks: Apple, Amazon, and Netflix. From the historical data that we see in Figure 5.7, we use 50% of them as train data, 10% for validation, and the remaining 40% for testing.

To check if the log returns are indeed close to a normal distribution, in Figure 5.8, we have shown the histograms and Q-Q plots of our train log returns. We can see that Apple and Amazon show log returns that are approximately normally distributed with some deviations at the tails. The histograms are bell-shaped but with slightly fatter tails, and the Q-Q plots confirm heavier tails than the normal distribution. The log returns of Netflix show more significant deviations from normality. The histogram has a sharp peak and fatter tails, and the Q-Q plot indicates substantial departures from the normal distribution, especially at the extremes. So, the log returns for AAPL and AMZN are reasonably close to normal with heavier tails, and NFLX exhibits more significant deviations from normality.

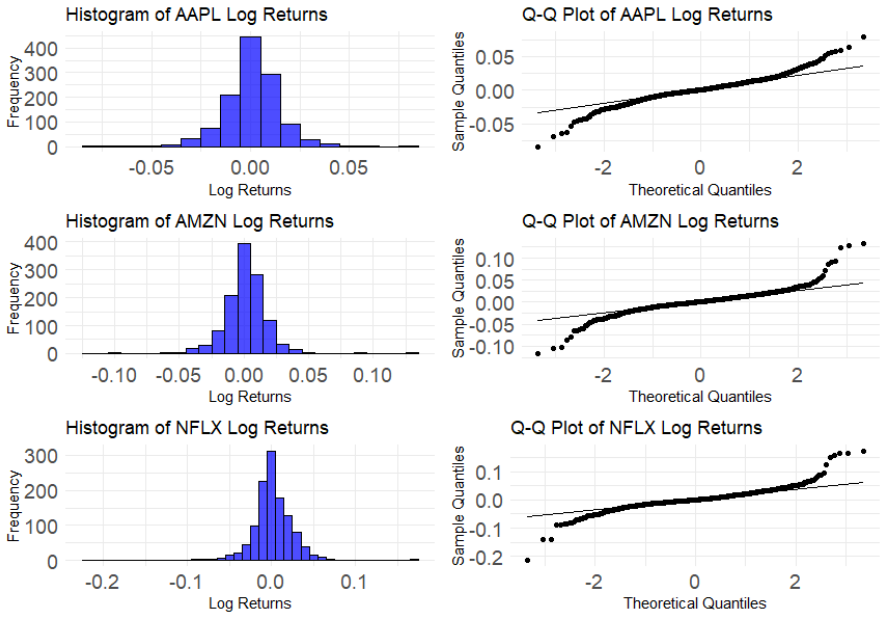


Figure 5.8: Histograms and Q-Q plots of the train log returns.

In Tables 5.1, 5.3, and 5.4, we have shown the estimated means, covariances, and correlations from the train log returns. Table 5.3 displays the annualized covariance matrix of the log returns for AAPL, AMZN, and NFLX. Each value represents the covariance between the pairs of assets on an annual basis, calculated by scaling daily covariances by 252 trading days.

	AAPL	AMZN	NFLX
Mean	0.0009	0.0012	0.0016

Table 5.1: Estimated means of the train log returns.

Correlation offers a more standardized view, which can be more insightful for comparing the strength of relationships across different pairs of stocks. From the

correlation matrix, we can see that all three pairs of stocks have moderate positive correlations, indicating that their returns tend to move in the same direction, but not perfectly. The strongest correlation is between AAPL and AMZN (0.5156), suggesting a relatively stronger relationship compared to the others.

	AAPL	AMZN	NFLX
AAPL	0.0002	0.0001	0.0001
AMZN	0.0001	0.0003	0.0002
NFLX	0.0001	0.0002	0.0007

Table 5.2: Estimated covariance matrix of the train log returns.

	AAPL	AMZN	NFLX
AAPL	0.0529	0.0252	0.02772
AMZN	0.02520	0.08568	0.05040
NFLX	0.02772	0.05040	0.17640

Table 5.3: Annualized covariance matrix of historical train data.

	AAPL	AMZN	NFLX
AAPL	1.0000	0.3750	0.2894
AMZN	0.3750	1.0000	0.4119
NFLX	0.2894	0.4119	1.0000

Table 5.4: Correlation matrix of the training log returns.

From the Table 5.4, we observed a moderate positive correlation between the variables. To generate additional log returns while maintaining this correlation structure, we will use Principal Component Analysis (PCA), as detailed in Section 5.3. While PCA is often employed for dimensionality reduction, in our case, we use it specifically to maintain the correlation structure among the log returns of the stocks. PCA transforms the original correlated variables into a set of uncorrelated principal components. These components are orthogonal to each other, which means they are statistically independent. This transformation is crucial for our analysis because it helps us maintain the underlying correlation structure among the stocks in the portfolio when generating synthetic data.

Alternatively, we can address the correlation between log returns using a specialized type of Generative Adversarial Network (GAN) known as *TimeGAN*. TimeGAN is designed specifically for time series data and excels at capturing complex temporal dependencies and correlations. Unlike traditional GANs, TimeGAN incorporates recurrent neural networks to model the sequential nature of time series data effectively. This makes TimeGAN particularly well-suited for generating

realistic time series data that maintains the intricate relationships observed in the original log returns. We will later compare the performance of TimeGAN with that of the combination of PCA and univariate GANs.

We start our PCA analysis by standardizing the data. This step is performed on the training dataset, which includes the initial 1208 data points of our log return stock prices, as shown in Figure 5.9. Standardization is essential because without it, variables with larger variances can dominate the results and lead to skewed outputs. By scaling the data to a comparable range, we address this issue. Specifically, we perform mean centering, where the mean is subtracted from each data point. This step doesn't significantly alter the data but ensures that the first Principal Component captures the direction of maximum variance without the bias of scale differences.

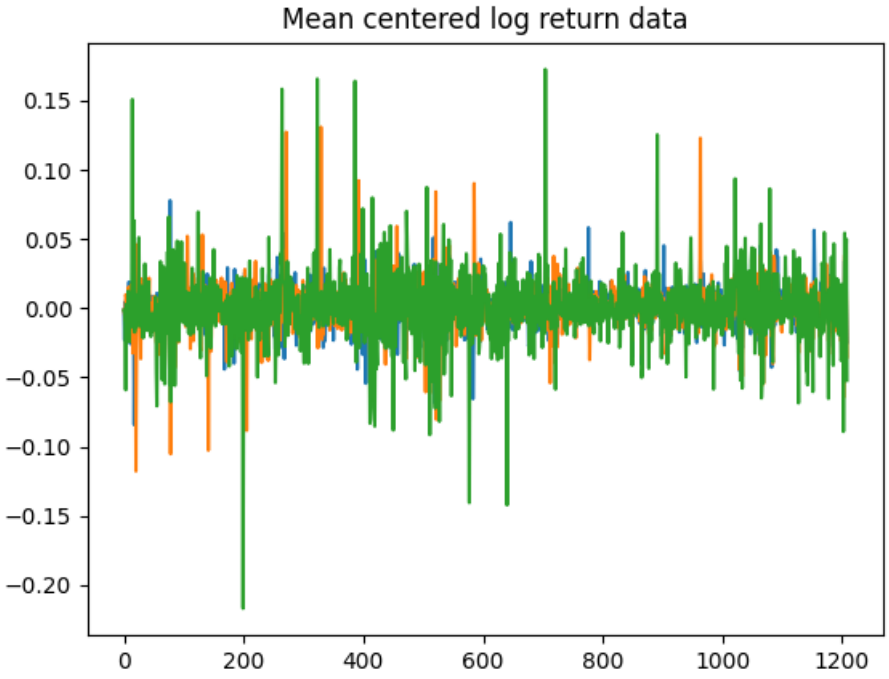


Figure 5.9: Mean centered train log return data.

The core of PCA involves calculating the eigenvectors and eigenvalues of a covariance matrix, which is symmetric and positive semi-definite. The eigenvectors indicate the directions of maximum variance (principal components), while the eigenvalues measure the variance along these directions. The first principal component corresponds to the largest eigenvalue and represents the direction of maximum variance, while the subsequent components capture progressively less variance.

We then perform the eigen decomposition of the covariance matrix for our training log returns, obtaining the eigenvectors and eigenvalues displayed in Table 5.5.

Eigen vectors	-0.224	-0.883	-0.411
	-0.411	0.469	-0.782
	-0.883	0.006	0.469
Eigen Values	0.00082	0.00015	0.00026

Table 5.5: Eigenvectors and eigenvalues of train log return data.

Table 5.6 shows the variance explained by each principal component, along with the cumulative variance. The first principal component explains approximately 66.015% of the total variance. The second component adds 21.659%, and the third contributes 12.326%. Cumulatively, the first component alone accounts for 66.015% of the variance, the first two components together explain 87.674%, and the three components combined cover 100% of the variance.

Principal Component	1	2	3
Variance Explained (%)	66.015	21.659	12.326
Cumulative Variance Explained (%)	66.015	87.674	100.000

Table 5.6: Variance explained and cumulative variance explained by each principal component.

Once we have the principal components, we use Generative Adversarial Networks (GANs) to generate synthetic data for each component individually. This approach ensures that the independence of the components, established by PCA, is maintained. After generating the synthetic data, we combine the components and apply the inverse PCA transformation to recreate the full dataset.

With the synthetic dataset ready, we estimate its means and covariances.

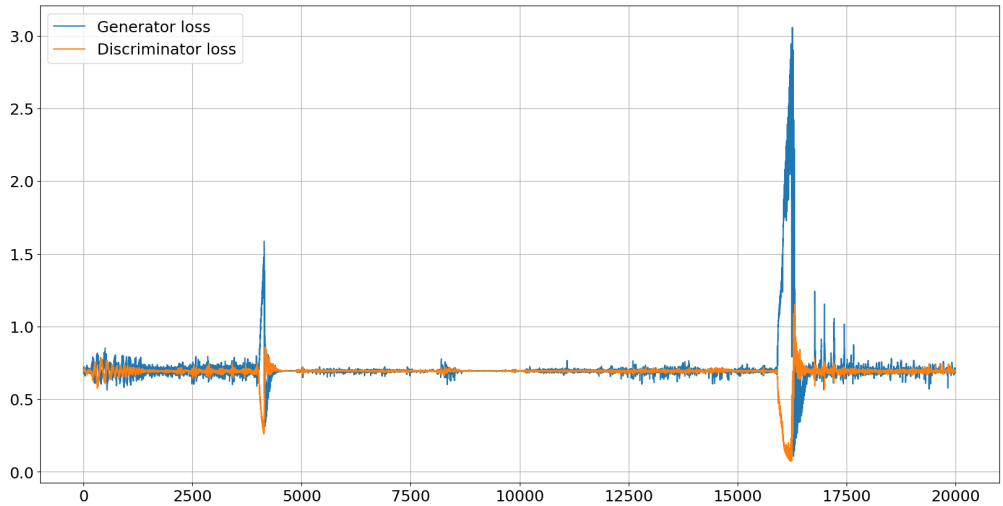


Figure 5.10: Disciminator's and Generatos's loss during the training process for the first PC.

For each principal component (PC), we generate 1000 additional paths for the next 5000 days using ForGAN, as detailed in Section 4. Figures 5.10, 5.11, and 5.12 show the training losses for both the discriminator and generator across the different PCs. In all cases, the discriminator's loss is more stable compared to the generator's. While the losses for the third principal component exhibit more fluctuations initially, both losses eventually converge, indicating a successful training process.

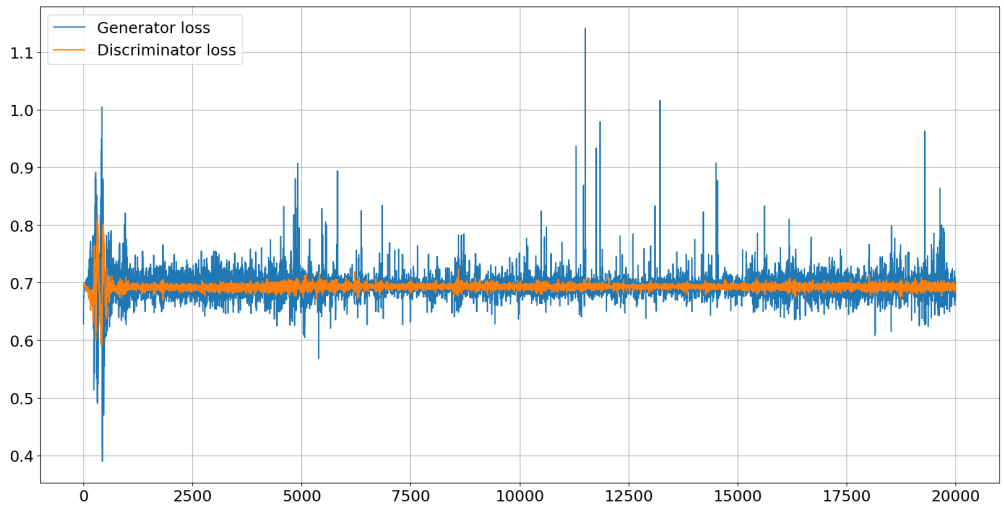


Figure 5.11: Disciminator's and Generatos's loss during the training process for the second PC.

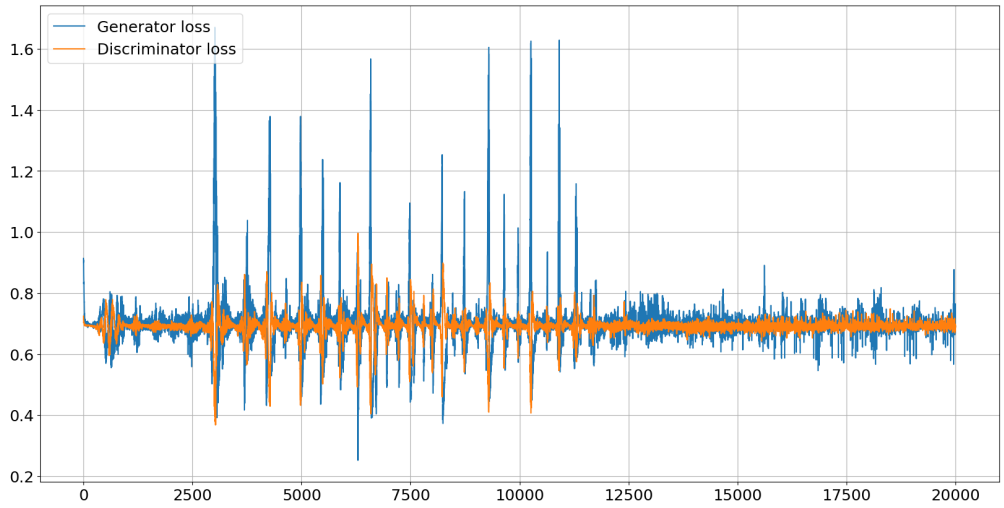


Figure 5.12: Disciminator's and Generatos's loss during the training process for the third PC.

The generated dataset consists of 1000 paths of log prices for each principal component. After reconstructing the data to its original form, Figure 5.13 presents the histograms and Q-Q plots of the generated data.

Comparing this with the train data in Figure 5.8, we see that the generated data closely resemble the training data in terms of overall shape and distribution. However, while the generated data captures the central tendencies well, it does not fully capture the tail behaviors observed in the train data. Overall, the generated data appears to be a simplified version of the actual log returns.

Further we look at the estimated means, covariances, and correlations of generated log returns, shown in Tables 5.7, 5.8, and 5.9, respectively, and compare them with the ones from historical train data.

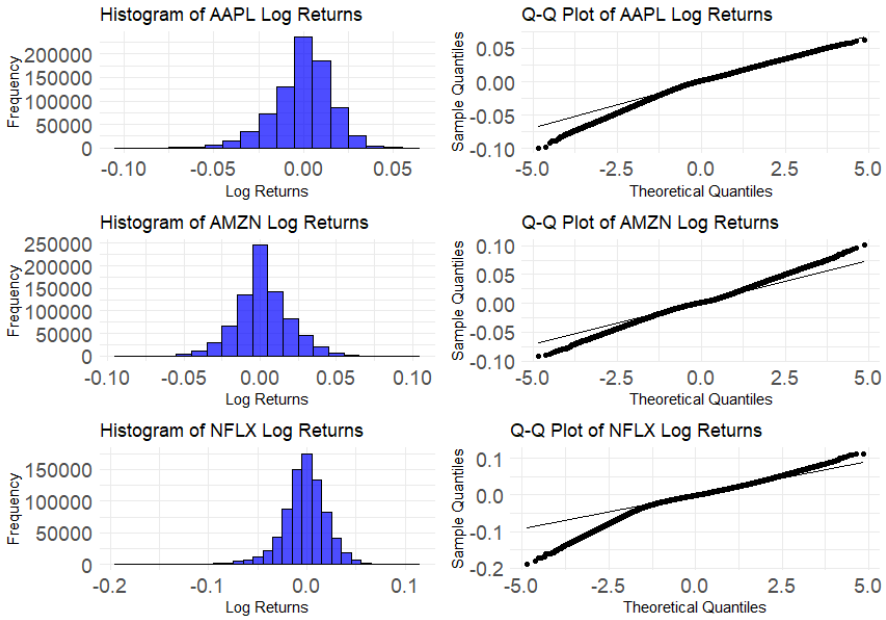


Figure 5.13: Histograms and Q-Q plots of the generated log returns.

The generated data shows a shift in mean log returns compared to the training data, with AAPL and AMZN showing more negative values and NFLX showing a higher mean.

Comparing the **correlation** values in Table 5.9 (generated data) with the values in blue from the training data, we observe that the correlations between AAPL and the other two stocks (AMZN and NFLX) are reduced in the generated data. Conversely, the relationship between AMZN and NFLX is slightly stronger in the generated data. This shift suggests a potential change in how these assets might interact in the generated scenarios, compared to the original training data.

	AAPL	AMZN	NFLX
Mean	-0.001 (0.0009)	-0.0028 (0.0012)	0.0038 (0.0016)

Table 5.7: Estimated means of the generated log returns (train data values in blue).

	AAPL	AMZN	NFLX
AAPL	0.0266 (0.0529)	0.0053 (0.0252)	0.0113 (0.0277)
AMZN	0.0053 (0.0252)	0.0338 (0.0857)	0.0206 (0.0504)
NFLX	0.0113 (0.0277)	0.0206 (0.0504)	0.0691 (0.1764)

Table 5.8: Annualized estimated covariance matrix of the generated log returns (train data values in blue).

	AAPL	AMZN	NFLX
AAPL	1.0000	0.1755 (0.3750)	0.2644 (0.2894)
AMZN	0.1755 (0.3750)	1.0000	0.4262 (0.4119)
NFLX	0.2644 (0.2894)	0.4262 (0.4119)	1.0000

Table 5.9: Correlation matrix of the generated log returns (train data values in blue).

When generating data with Principal Component Analysis (PCA) and Generative Adversarial Networks (GANs), several factors affect the results. These include the choice of hyperparameters during GAN training, the amount of training data for both PCA and GANs and the size of the condition window used in GANs. We focus on more intensive training for the first principal component (PC), which captures most of the data's variance, to ensure that the generated data aligns closely with the distribution of the original data. By fine-tuning these factors and focusing on the most important components, especially the first PC, we can significantly improve the quality of the generated data.

To optimize the portfolio we will use both data sets, the train log returns and the generated log returns from GANs.

To assess the risk and return of the data generated from GANs, we first assign three random weights to each asset, ensuring the sum of the weights equals 1. The portfolio return is calculated as follows:

$$r_W = \left(\sum_{i=1}^{n} w_i * r_i + 1\right)^{252} - 1. \tag{5.30}$$

where the r_i is the mean return of the *i-th* asset and w_i is the weight of the *i-th* asset in the portfolio. The result is incremented by 1, raised to the power of 252 (assuming 252 trading days in a year for daily returns), and then decremented by 1 to obtain the annualized return.

We calculate the portfolio risk using the following formula:

$$\sigma_W = \sqrt{W^T * \Sigma * W} \tag{5.31}$$

where W is the column vector of weights, Σ is the covariance matrix which we annualized by multiplying with 252. The matrix multiplication represents the

weighted sum of covariances, and the square root is applied to obtain the standard deviation, representing the portfolio's risk.

Table 5.8 shows the annualized covariance matrices for both the train (in blue) and generated data. We observe that covariances between stocks are generally lower in the generated data, suggesting weaker relationships or even slight inverse relationships compared to historical data. In the following part, we see that those differences impact portfolio diversification, since the generated data portrays a less correlated market environment.

For our example, using the log returns generated by GANs and assigning three random weights to Apple, Amazon, and Netflix, we obtain:

$$\begin{aligned} &\textit{Portfolio risk:} \quad 0.1341 \\ &\textit{Portfolio return:} \quad 0.1851. \end{aligned}$$

Next, we evaluate over 5000 randomly picked portfolios, i.e., we explore various combinations of asset allocations (weights). We estimate the returns and risks for each portfolio and assume a 0% risk-free rate to calculate the Sharpe Ratios:

$$S = \frac{r_W}{\sigma_W}$$

where S is the Sharpe Ratio. The tables below present the results for the optimal portfolios: the portfolio with the minimal variance (the Minimum Variance Portfolio) and the one with the highest Sharpe Ratio (the Tangency Portfolio). The corresponding weights of each asset, the risk, returns, and Sharpe Ratio of each portfolio are shown below.

We should note here that the lower correlation of the assets in generated data reduces the diversification benefits, resulting in potentially higher risk for a given level of return. Investors might prefer portfolios from the historical data efficient frontier for more stable and diversified investments.

The Minimum Variance Portfolio					
AAPL	AMZN	NFLX	Return	Risk	SharpeRatio
0.432	0.388	0.180	0.0947	0.169	0.559

Table 5.10: Statistics of the Minimum Variance Portfolio for generated data.

The Tangency Portfolio					
AAPL	AMZN	NFLX	Return	Risk	SharpeRatio
0.00490	0.991	0.00422	0.712	0.271	2.63

Table 5.11: Statistics of the Tangency Portfolio for generated data.

Using our historical data set, we assign three random weights to each asset, ensuring the sum of the weights equals 1. When we calculate the risk and return, as

shown in Equations 5.30 and 5.31, we get the following results:

$$\begin{aligned} \textit{Portfolio risk:} \quad & 0.313 \\ \textit{Portfolio return:} \quad & 0.348. \end{aligned}$$

Additionally, when evaluating 5000 randomly selected portfolios, we estimate the returns and risks, assuming a 0% risk-free rate. We obtain the succeeding portfolios as the optimal ones from all the random 5000 portfolios.

The Minimum Variance Portfolio					
AAPL	AMZN	NFLX	Return	Risk	SharpeRatio
0.682	0.269	0.0497	0.279	0.211	1.32

Table 5.12: Statistics of the Minimum Variance Portfolio for true historical data.

The Tangency Portfolio					
AAPL	AMZN	NFLX	Return	Risk	SharpeRatio
0.421	0.361	0.218	0.321	0.227	1.42

Table 5.13: Statistics of the Tangency Portfolio for train historical data.

The efficient frontier is shown in Figure 5.14, with the two red points representing the *Minimum Variance Portfolio* and the *Tangency Portfolio*.

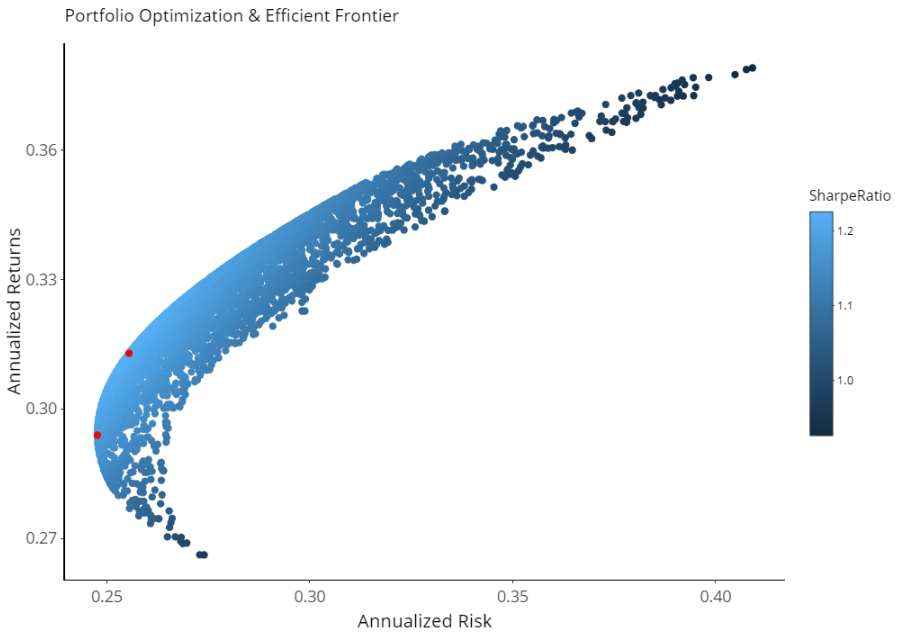


Figure 5.14: Efficient frontier for the portfolio obtained from train data.

To compare the optimal portfolios derived from the historical data and the generated data, we examine the optimal portfolios which are detailed in Tables 5.10,

5.11, 5.12, and 5.13. We extract the corresponding portfolio weights and compute the cumulative returns for both the complete historical data and the test set, which includes the last 40% of the log returns (not included in the PCA).

We calculate the cumulative returns in the following way:

Weighted Portfolio Returns Given weights w_i for assets $i = 1, 2, \ldots, n$, the weighted log return for the portfolio at time t is:

$$R_t = \sum_{i=1}^{n} w_i r_{i,t} \tag{5.32}$$

where $r_{i,t}$ is the log return of asset i at time t.

Cumulative Returns from Log Returns To calculate cumulative returns directly from log returns, we use the property that the sum of log returns over time gives the log of the cumulative return. The cumulative return CR_t at time t is given by:

$$CR_t = \exp\left(\sum_{k=1}^{t} R_k\right) \tag{5.33}$$

where R_k is the weighted log return of the portfolio at time k.

In our analysis, Figures 5.15 and 5.16 compare the cumulative returns of the minimum variance portfolios derived from historical training data and GAN-generated data over different periods.

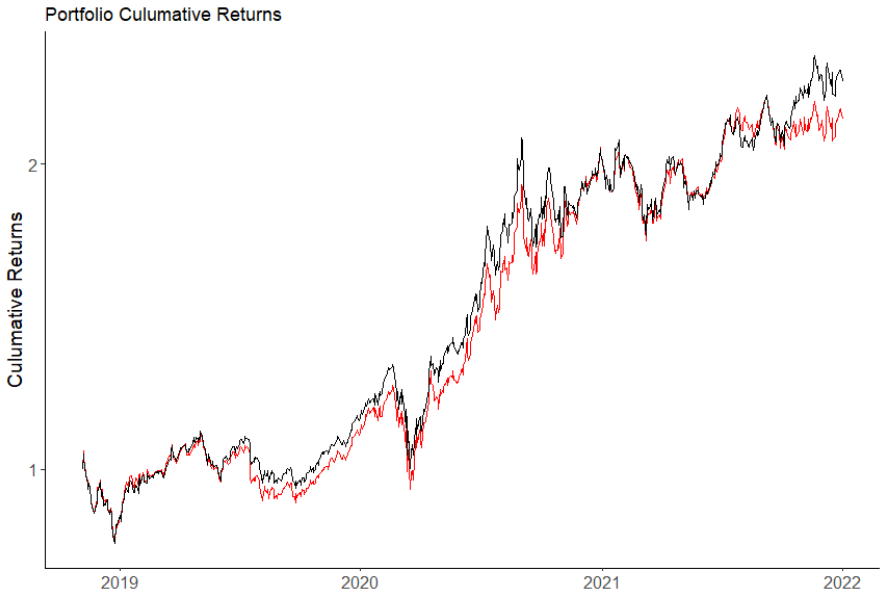


Figure 5.15: Cumulative return of the portfolio over the test data period. The **red** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the historical data, while the **black** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the GAN-generated data.

These comparisons provide insights into how these portfolios perform under different market conditions, helping to assess the reliability of GAN-generated data in replicating historical patterns.

Figure 5.15 shows the cumulative returns of the portfolio over the test data period. The red line represents the cumulative returns calculated using the weights of the optimal portfolio (minimum variance portfolio) derived from the train historical data. The black line represents the cumulative returns calculated using the weights of the minimum variance portfolio derived from the GAN-generated data.

We previously described the optimization process and provided the weights tables. Cumulative returns are then calculated using these optimal weights on the test data set, the last 794 days of our data set.

By comparing the red and black lines in Figure 5.15, we observe that the weights from the GAN-generated data lead to slightly higher cumulative returns over the test period than those from the historical data. However, the cumulative returns from both datasets are very close, indicating that the GAN-generated data captures the structure of the training data well.

Figure 5.16 displays the cumulative returns of the portfolio over the complete historical data period.

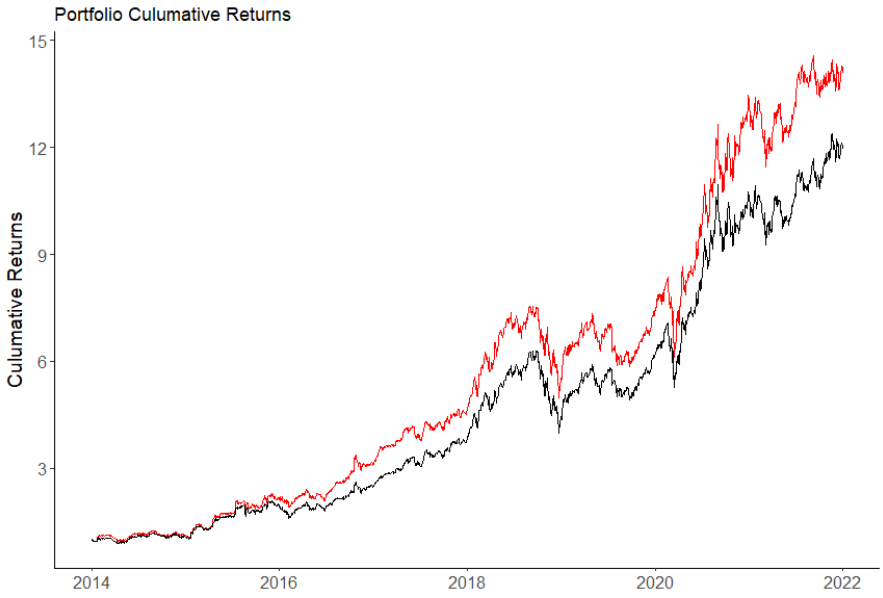


Figure 5.16: Cumulative return of the portfolio over the complete historical data period. The **red** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the historical data, while the **black** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the GAN-generated data.

The same process is followed here, and we see that the red line (based on historical data weights) generally shows higher cumulative returns than the black line (GAN-

based weights) over the entire period. However, the black line converges toward the red line in recent years, indicating that the GAN-generated data performs comparably in the long term.

The higher cumulative returns from GAN weights in the test data period (as seen in Figure 5.15) suggest potential advantages of using GAN-generated data for short-term investment horizons. As both data sets are quite similar, they lead to approximately the same performance. However, for long-term investments, historical data-based portfolios might be preferred for more reliable and diversified returns, as evidenced by the long-term cumulative performance in Figure 5.16.

Another way to compare the performance of GAN-generated data for portfolio optimization is to specify a level of risk for both the generated data and the historical training data. By doing this, we can obtain the portfolio weights for each stock that achieves this specified risk level. Once we have these weights, we can calculate the cumulative return of these portfolios over the test period, similar to our previous approach.

We have chosen a risk level of 0.223. As before, we calculate the portfolio's cumulative return for the test data set.

In Figure 5.17, the black line, representing the cumulative returns from the GAN weights, consistently outperforms the red line, which represents the returns from the train historical weights.

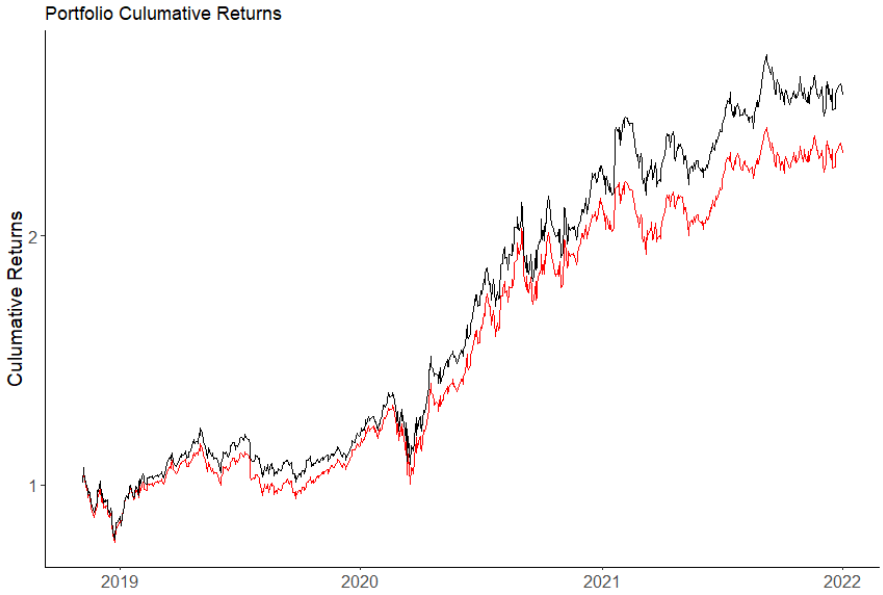


Figure 5.17: Cumulative return of the portfolio over the test historical data period. The **red** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the historical data for a specific level of risk 0.223, while the **black** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the GAN-generated data for a specific level of risk 0.223.

We have calculated cumulative returns for the test set across different risk levels. The results demonstrate that, in most cases, the portfolios optimized using GAN-generated data yield higher cumulative returns compared to those optimized using train historical data.

While the results indicate that the GAN-generated data generally yield higher cumulative returns, it is important to note that the correlation structure between the log returns is not perfectly captured. This discrepancy suggests that, although the generated data may show favorable return characteristics, it may not fully replicate the underlying relationships observed in the historical data. Additionally, it is worth mentioning that for some specific risk levels, the GAN-generated weights do not always outperform the historical data. However, in general, the results from both the train and GAN-generated data are very similar.

TimeGAN for generating stock prices

As outlined in Section 5.2, TimeGAN includes two additional networks: an embedding network, which compresses high-dimensional time-series data into a lower-dimensional latent space, and a recovery network, which decodes this latent representation back into the original data space. Additionally, a supervisor network ensures that the generated latent space accurately captures the temporal structure of the data.

To evaluate TimeGAN's effectiveness in generating time series, we apply t-SNE [31] and PCA [20] analyses to both the true and generated data.

As previously mentioned PCA reduces the data's dimensionality while retaining as much variability as possible. t-SNE (t-Distributed Stochastic Neighbor Embedding) is a non-linear dimensionality reduction technique that is particularly good at preserving local structures and separating clusters in the data.

Figures 5.18 and 5.19 illustrate that the synthetic data generated by TimeGAN retains the overall structure and relationships of the original data. Both datasets are projected into a 2D space for visualization.

For PCA, we use the first two principal components to reduce the data's dimensionality and then plot the results in a scatter plot, with the original data shown in red and the synthetic data in blue.

t-SNE further reduces the data to a two-dimensional space, focusing on preserving local data structures. The plot visualizes these reduced dimensions.

The t-SNE plot, in particular, demonstrates that the synthetic data preserves local structures fairly well, which is important for time-series data where the order of the data points matters. However, some differences are present, particularly in regions where synthetic data extends beyond the original data's range.

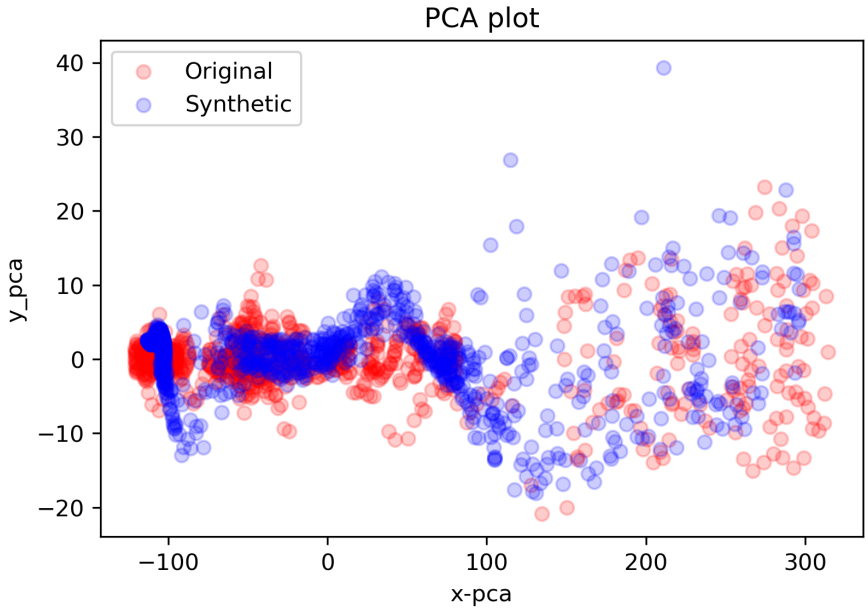


Figure 5.18: PCA plot comparing the original and TimeGAN-generated data.

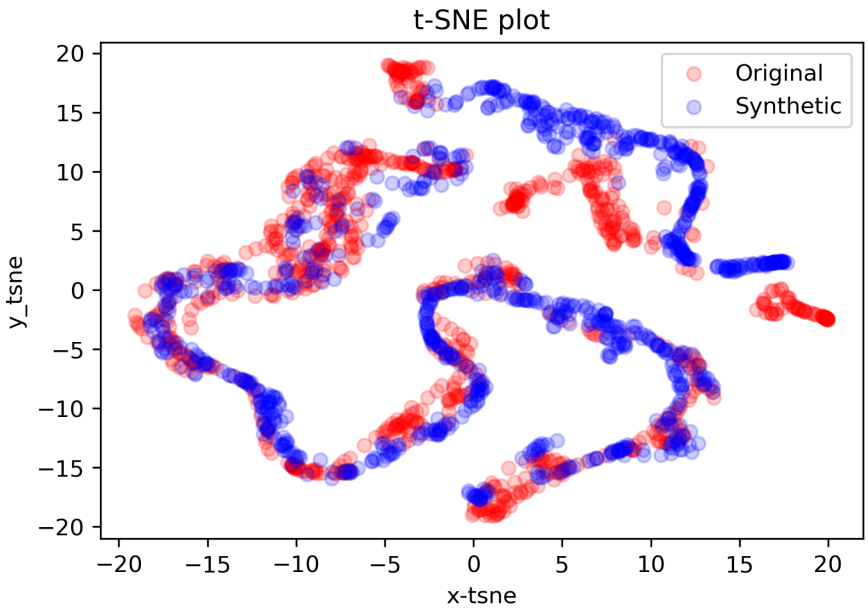


Figure 5.19: t-SNE plot showing the distribution of original versus TimeGAN-generated data.

To generate the data shown in Figures 5.18 and 5.19, we first applied a *MinMaxScaler* to normalize the data. Later, we used log returns instead, which helped stabilize the variance, make the data less skewed, and handle outliers better, making it easier to model.

However, regardless of the normalization technique used, TimeGAN tends to pro-

duce synthetic data with stronger correlations than those present in the original dataset. While the good performance in PCA and t-SNE indicates that the model effectively captures the overall structure and distribution of the data, the observed stronger correlations suggest that the synthetic data may lack sufficient diversity or that the model is overly emphasizing certain relationships.

Further, we generated data using TimeGAN based on log returns and analyzed their means and correlations, as shown in Tables 5.14 and 5.15. The generated data shows much stronger relationships between the stocks than the training data, suggesting it may have overestimated how closely the stocks move together. On the other hand, the mean log returns are lower in the generated data compared to the training data, as reflected in the blue values.

	AAPL	AMZN	NFLX
AAPL	1.0000	0.8765 (0.3750)	0.7878 (0.2894)
AMZN	0.8765 (0.3750)	1.0000	0.7738 (0.4119)
NFLX	0.7878 (0.2894)	0.7738 (0.4119)	1.0000

Table 5.14: Correlation matrix of the generated log returns from TimeGAN (training data values in blue).

	AAPL	AMZN	NFLX
Mean	-0.0010 (0.0009)	-0.0031 (0.0012)	-0.0020 (0.0016)

Table 5.15: Means of the generated log returns from TimeGAN (training data values in blue).

Summary and Implications

Both ForGAN with PCA and TimeGAN offers effective methods for generating synthetic data, each with distinct strengths and challenges. Our method involves first using PCA to analyze the data and then working with each principal component individually before putting everything back together. This approach allows us to have clear control over each step of the process and we don't reduce the dimensions of the data along the way. As a result, it provides a more reliable framework for estimating parameters crucial for portfolio optimization (PO), particularly when accurate asset correlations are important.

In contrast, TimeGAN uses a different approach. It first compresses the time series data into a smaller, lower-dimensional space using an embedding network. Then, it tries to decode or reconstruct the data back to its original form. This compression step introduces an extra layer of abstraction, which can sometimes lead to more potential errors or inaccuracies in the final result, including a less accurate representation of the correlations between different time series.

In essence, while TimeGAN excels in capturing temporal patterns, the potential loss of correlation information can impact its effectiveness in portfolio optimization, where the relationships between assets are critical for diversification and risk management.

5.5 Portfolio Optimization Using GANs and PCA for Stock Price Data

Since our previous analysis demonstrated that ForGAN-PCA performs well for generating synthetic data over a longer time horizon, we now delve deeper into this method by focusing on a shorter time frame, generating data for the next day only. Generating next-day data with ForGAN-PCA aligns with real-time portfolio optimization strategies, where short-term forecasts are critical for day-to-day decision-making.

As mentioned earlier, before performing PCA, it is essential to standardize the data. In this step, we use the training dataset, which consists of the initial 1768 data points of our log return stock prices, as illustrated in Figure 5.7. Without standardization, the data can be disproportionately affected by the variances of the original variables. Therefore, mean centering is performed to ensure that the first Principal Component aligns with the direction of maximum variance.

We are using log-returns of the stock prices, thus using the Standard Scaler, which standardizes a feature by subtracting the mean and then scaling to unit variance, i.e., dividing all the values by the standard deviation would increase the scale of our data. We want to keep the same scale, so we perform mean centering by only subtracting the mean, which doesn't affect the data significantly, see Figure 5.20.



Figure 5.20: Comparison of original log returns, standardized log returns (using Standard Scaler), and mean-centered log returns for the training data.

Next, the essence of PCA is calculating the eigenvectors and eigenvalues of a covariance (or correlation) matrix. Every covariance matrix is symmetric and positive semi-definite, and it is used to determine the direction of a linear relationship between two variables. Moreover, it has orthogonal eigenvectors. The eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues assess their magnitude. In other words, the eigenvalues explain the data variance along the new feature axes. We perform the eigen decomposition of the covariance matrix and get eigen vectors and eigen values. Each eigenvector

	Apple	Amazon	Netflix
Apple	0.00033	0.00018	0.00019
Amazon	0.00018	0.00038	0.00026
Netflix	0.00019	0.00026	0.00071

Table 5.16: Covariance matrix of the log returns for the training data.

represents a direction of variance. The eigenvector corresponding to the most significant eigenvalue will give the direction of maximum variance, which is the first principal component. Then, the eigenvector corresponding to the second-largest eigenvalue will provide the direction of the second-largest variance, i.e., the second principal component. And so on.

We must sort the eigenvectors in descending order based on their corresponding eigenvalues. This ensures that the principal components representing the greatest variance come first. However, since our goal is not dimensionality reduction but preserving the full structure of the data, we retain all components for further analysis and processing.

Eigenvalues in descending order:
0.00095
0.00029
0.00017

Table 5.17: Eigenvalues in descending order for principal component analysis.

We select the top k eigenvectors based on the desired level of compression. The optimal number of components is determined by computing the explained variance for each feature, which is done by dividing the eigenvalues by the sum of all eigenvalues. The cumulative sum of these values helps us understand how much variance each principal component captures.

Principal Components	PC 1	PC 2	PC 3
Explained Variance (%)	67.47	88.21	100

Table 5.18: Explained variance for each principal component.

Table 5.18 shows that the first principal component explains 67.47% of the variance in the data, while the first two components together explain 88.21%, and all three combined explain 100%. Afterward, we use the projection matrix, constructed from the top k eigenvectors, to transform the input data into the new feature subspace. Let's reduce the 3-dimensional feature space to a 1-dimensional subspace by selecting the top eigenvector, which captures 67.47% of the variance.

The transformed data is shown in Figure 5.21. After reconstructing the data back into its original dimensions (Figure 5.22), we calculate the reconstruction loss using

the mean squared error (MSE) between the original and reconstructed data, which yields a value of 0.000153.

In this analysis, we continue to utilize Principal Component Analysis (PCA) on the training dataset. PCA transforms the original correlated variables into uncorrelated principal components. These components are orthogonal to each other, meaning they are statistically independent, which is crucial for our analysis.

Once we have the principal components, we use Generative Adversarial Networks (GANs) to generate new synthetic data for each component individually. Generating data for each principal component separately helps retain the independent properties of these components. After generating the synthetic data for each principal component, we reconstruct the complete data set by combining these components. This step involves transforming the generated principal component data back into the original data space, effectively reversing the PCA transformation. The result is a synthetic data set that mimics the statistical properties of the original training data.

It is important to reiterate that in this case, PCA is not used for dimensionality reduction but rather to maintain the independence of the generated data components. As mentioned earlier, in cases with many principal components, one could prioritize those with the highest explained variance to capture the most significant patterns in the data. However, in this analysis, we retain all components.

After reconstructing the synthetic dataset, we estimate its means and covariances to ensure the generated data closely mirrors the original data's statistical properties.

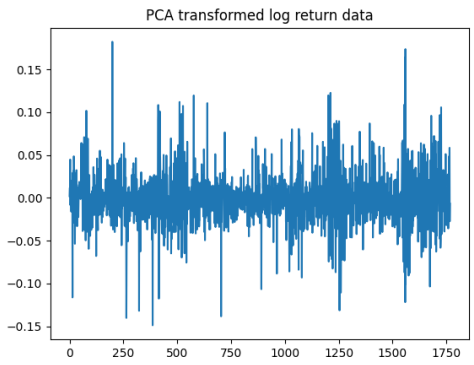


Figure 5.21: Log return data after applying PCA, transformed into the principal component space.

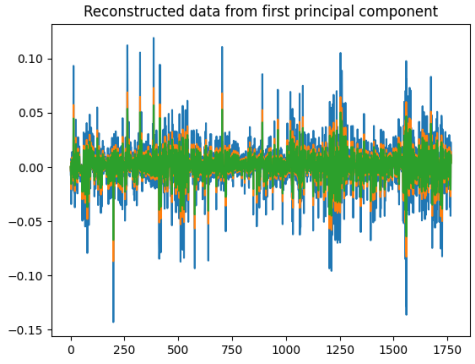


Figure 5.22: Reconstructed log returns using only the first principal component, capturing the direction of maximum variance.

Remember that every covariance matrix is symmetric, positive, semi-definite, and has orthogonal eigenvectors. Our goal is to compute the eigenvectors and eigenvalues of this matrix to capture the structure of our dataset after mean centering, as shown in Figure 5.20.

	Apple	Amazon	Netflix
Apple	0.00033	0.00018	0.00019
Amazon	0.00018	0.00038	0.00026
Netflix	0.00019	0.00026	0.00071

Table 5.19: Covariance matrix of the training dataset, showing the relationships between AAPL, AMZN, and NFLX stock returns.

The covariance matrix of our data is now shown in Table 5.19, we have only positive coefficients. The correlation matrix, shown in Table 5.20, reveals moderate positive correlations between AAPL, AMZN, and NFLX returns.

	AAPL	AMZN	NFLX
AAPL	1.0000	0.5156	0.3891
AMZN	0.5156	1.0000	0.5051
NFLX	0.3891	0.5051	1.0000

Table 5.20: Correlation matrix of the training dataset, illustrating the strength of relationships between AAPL, AMZN, and NFLX stock returns.

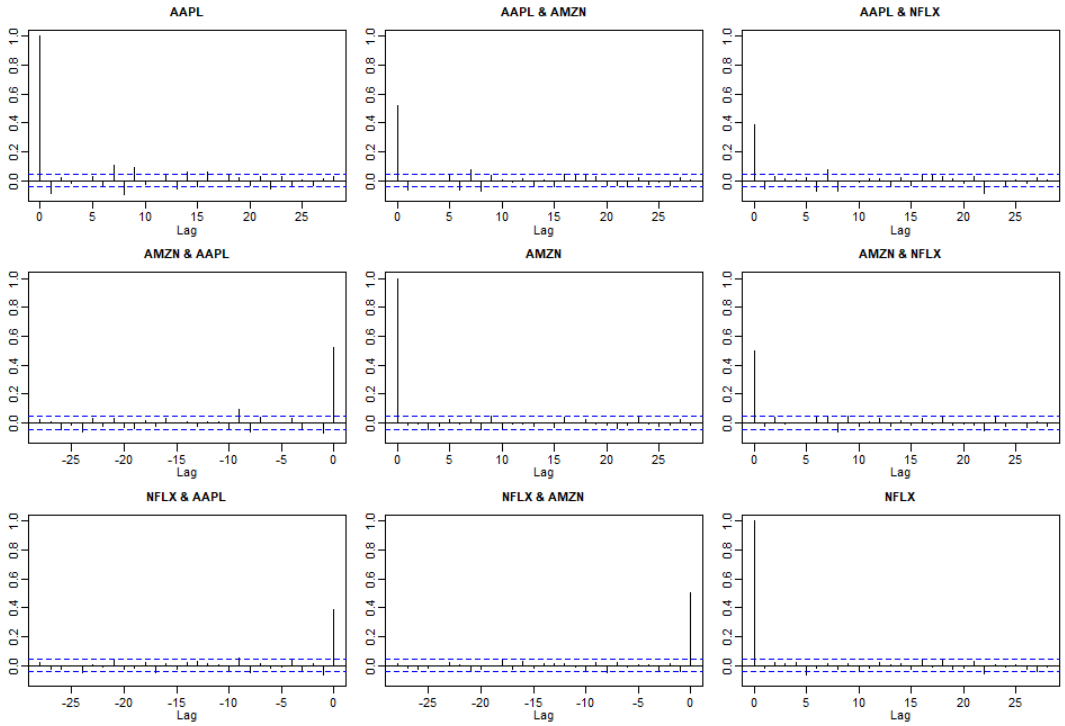


Figure 5.23: Autocorrelation function (ACF) for the complete dataset, showing how the correlation between stock returns diminishes over time.

These moderate correlations suggest that while the stock returns tend to move in the same direction, they do not do so perfectly. This is beneficial for portfolio diversification, as some level of uncorrelated movement is desired to reduce overall portfolio risk.

Furthermore, we will examine the autocorrelation function for the entire dataset, as shown in Figure 5.23, and for the training data in Figure 5.24. These observations indicate that the training data maintains a similar autocorrelation structure to the full dataset, which is important for ensuring consistency in the following analysis. The ACF plots show that the autocorrelations for most pairs (e.g., AAPL, AMZN, NFLX) quickly decay to near zero as the lag increases, meaning that the past values of the stock prices do not have a strong or lasting influence on future prices over time. This unpredictability makes it challenging to generate accurate synthetic data.

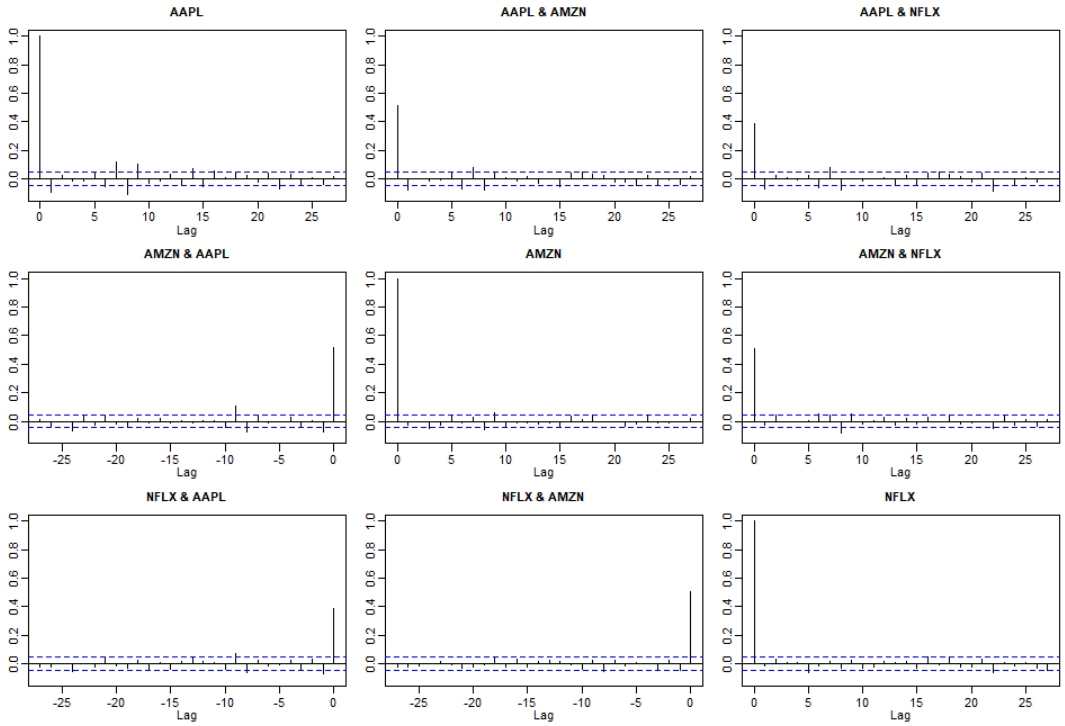


Figure 5.24: Autocorrelation function (ACF) for the training dataset, reflecting a similar decay in correlations as the complete dataset.

Moreover, we do the eigen decomposition of the covariance matrix and get eigen vectors and eigen values shown in Table 5.21.

Eigen vectors	-0.375	-0.689	-0.619
	-0.483	0.716	-0.505
	-0.791	-0.109	0.601
Eigen Values	0.0009	0.00017	0.00029

Table 5.21: Eigenvalues and eigenvectors of the covariance matrix, which represent the principal components of the data.

Afterward, we use a projection matrix to transform the input data onto the new feature subspace. First, we select the first eigenvector, the second eigenvector, and so on. The obtained PCA transformed log return data are shown in Figure 5.25.

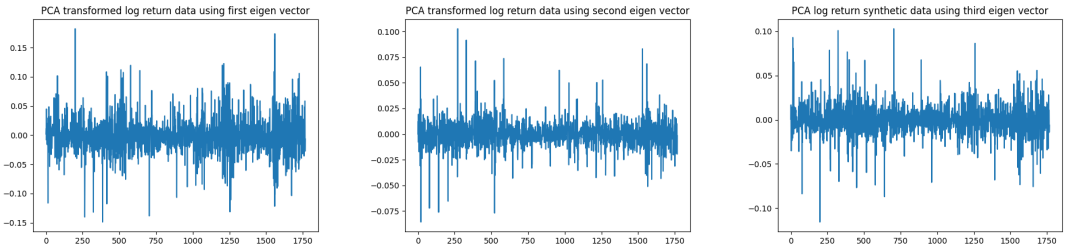


Figure 5.25: PCA-transformed log return data for the first, second, and third principal components.

We aim to generate more data using GANs, then perform a back-transformation and use all the information we have to optimize our portfolio. However, to better understand the performance of PCA, we will first invert the PCA transformation and reconstruct the original data before moving on to GANs.

To reconstruct the train log returns to their original form, we reverse the PCA transformation by taking the dot product of the transposed eigenvectors with the transformed data (i.e., the data projected onto each principal component). This process is illustrated in Figure 5.26.

Recall that we initially centered the data by subtracting the mean. Therefore, to accurately reconstruct the original data, we must add this mean back after performing the dot product. Since the eigenvectors are orthogonal, their dot product with themselves yields an identity matrix, which proves the correct reconstruction of the data from the principal components.

Subsequently, we will use the PCA-transformed train data set we get from each PC and use GANs to generate more data. We start with the PCA-transformed data from the first principal component, generating 1000 forecasts for the next day, followed by the second and third principal components.

When generating new data, we impose two statistical tests as conditions: an F-test and a t-test. These tests ensure that the mean and variance of the generated data are not significantly different from the true data. We iterate over various hyperparameters and select those that satisfy this condition. Different hyperparameters

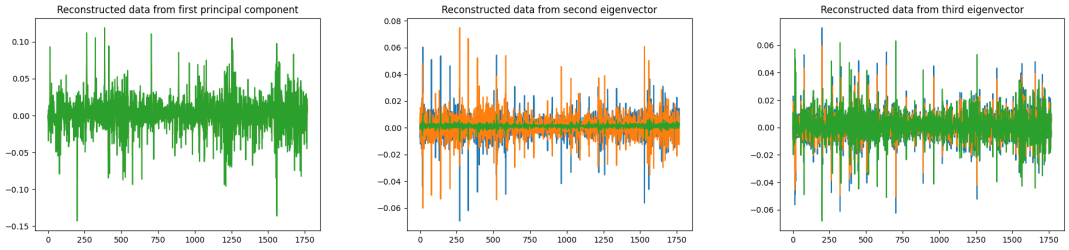


Figure 5.26: Reconstructed training log returns from the first, second, and third principal components, after performing the inverse PCA transformation.

are used for each PC, and the ForGAN model (as described in Section 4) is used to generate the synthetic data.

Hyperparameters	PC 1	PC 2	PC 3
Rolling window	4	4	4
Batch size	2	2	2
Noise size	4	2	4
Learning rate	0.001	0.001	0.001
Generator's cell	4	4	4
Discriminator's cell	4	3	3
Discriminator's training iterations	10	12	12

Table 5.22: Hyperparameters used for GAN training for each principal component (PC).

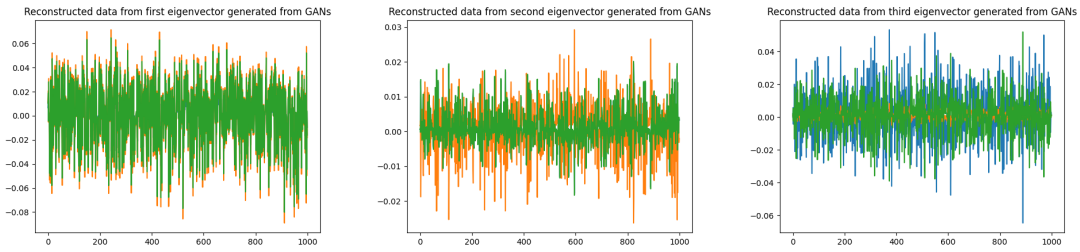


Figure 5.27: Reconstructed Log Return Data from PCA for GAN-Generated Data.

After generating 1000 additional data points for the next day from each PC, we reconstruct the data to its original form, as seen in Figure 5.27. The first eigenvector explains approximately 67.47% of the variance, making it a critical component. In cases with a large number of stocks, focusing on the first component might be a good approach.

Following the reconstruction, we will concatenate the information we already have, i.e., log returns for the first 1768 days with the generated data from GANs for the next day. We will then estimate the means and covariances for the following

three scenarios and optimize 5000 randomly selected portfolios for each case.The scenarios are as follows:

- **First case** Using only the training data (i.e., the log returns for the first 1768 days).
- **Second case** Combining the training data with the GAN-generated data for the next day.
- **Third case** Using only the GAN-generated data.

The efficient frontiers for each case are shown in Figures 5.28, 5.29, and 5.30, with the red points representing the optimal portfolios—specifically, the minimum variance portfolio and the maximum Sharpe ratio. In general, the GAN-generated data provides higher returns for lower levels of risk. The second case offers a good balance between maintaining similarity to the historical data and achieving enhanced returns. In the following part, we will analyze how well the generated data captures the structure of the training data.

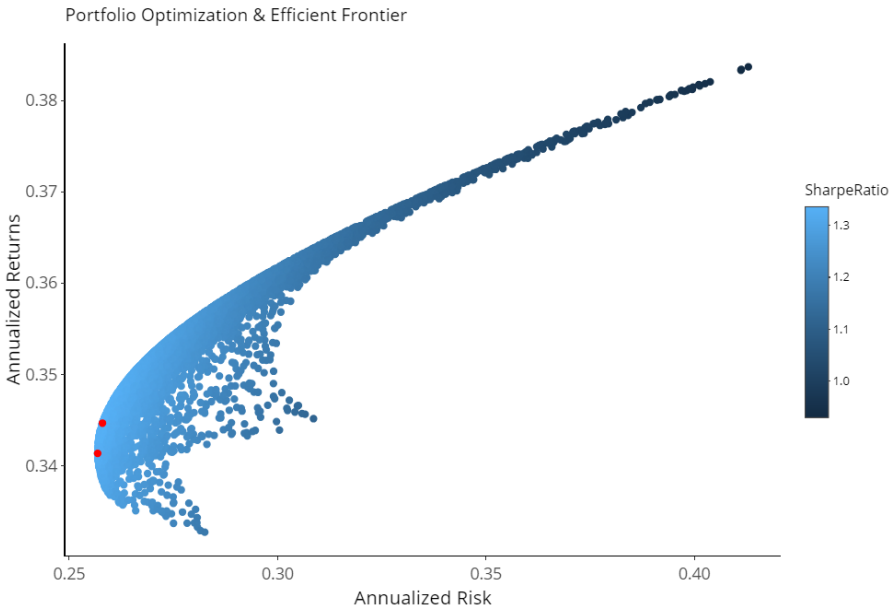


Figure 5.28: Efficient Frontier generated using only the train historical data for portfolio optimization (first case).

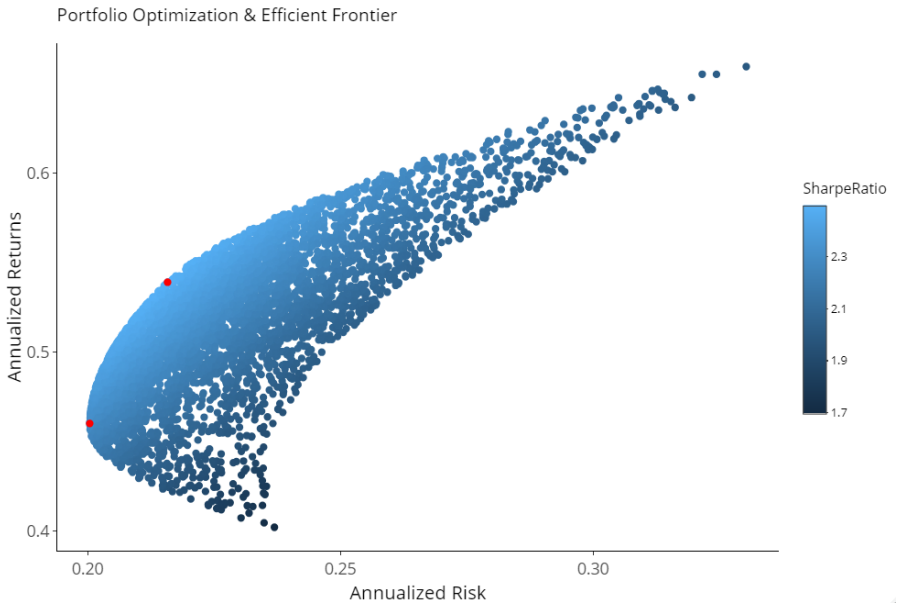


Figure 5.29: Efficient Frontier generated using the merged data from train historical data and GAN-generated data (second case).

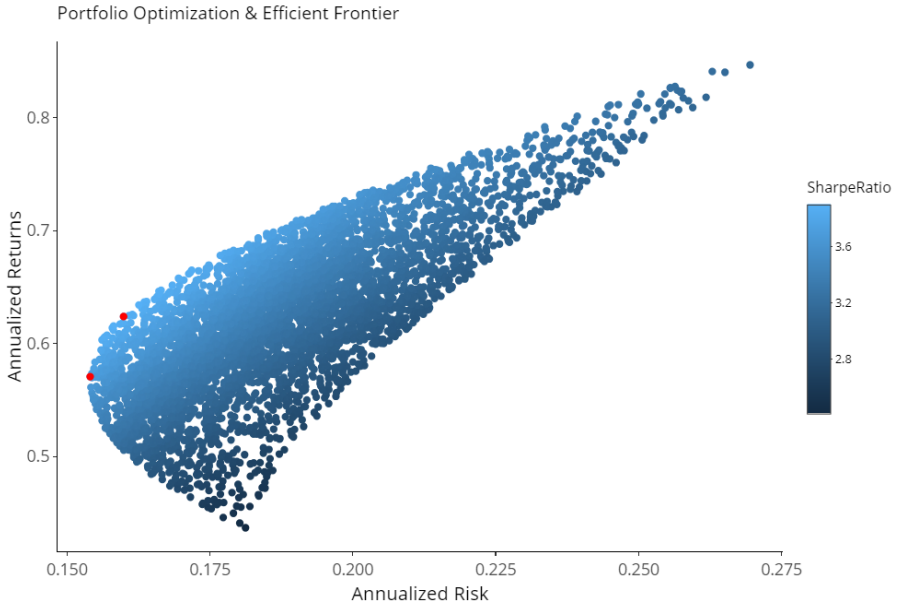


Figure 5.30: Efficient Frontier from generated data with GANs (third case).

Next, we calculate the Cumulative Returns as described in Equation 5.33 for various time frames: one day (the day following the training data, which is the first day of the test data), ten days, one hundred days, and 247 days (the full test period). Cumulative Returns (CR) provide a straightforward metric to assess portfolio performance over time, making them ideal for comparing the reliability of the generated data. See Figure 5.31 for the results. To calculate the Cumulative Returns, we use the weights obtained from the optimal portfolios—specifically, the

minimum variance portfolio and the maximum Sharpe ratio portfolio. The results indicate that for shorter time frames (one and ten days), portfolios based on the optimal weights derived from the training data achieve higher Cumulative Returns. However, over longer periods (100 and 247 days), the portfolios optimized using GAN-generated data yield higher Cumulative Returns. Nonetheless, the results are quite similar across all time frames, indicating that the GAN-generated data captures the structure of the training data well.

CR for the first case	CR for the second case	CR for the third case
0.9772231	0.9771222	0.976742

Table 5.23: Cumulative Returns (CR) for the next day across the three cases: first case (training data only), second case (training data merged with GAN-generated data), and third case (GAN-generated data only).

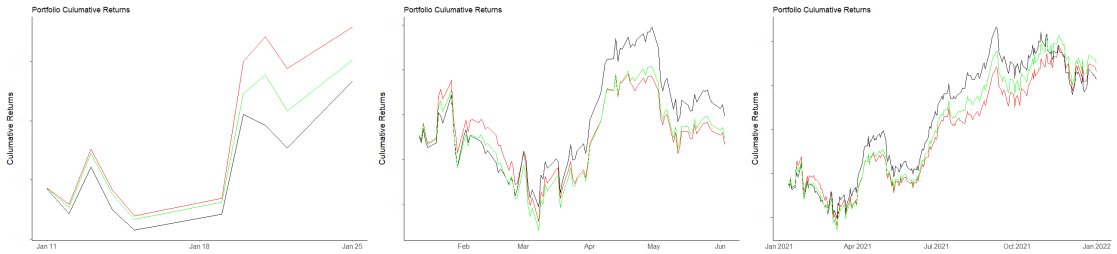


Figure 5.31: Portfolio Cumulative Returns, where red represents the train data (first case), green the train merged with generated data (second case) and black GANs generated data (third case).

To compare the relationships between assets in the generated data and the training data, we calculated the correlation matrices for the generated data, shown in Table 5.25. The results indicate that the correlations in the generated data are generally stronger than those observed in the training data. In Table 5.24, we present the correlation matrix for the merged data (training data combined with GAN-generated data). The correlations in the merged data are slightly higher and more uniform compared to those in the training data, suggesting that the generated data captures the overall structure while introducing a tighter relationship between the assets.

	AAPL	AMZN	NFLX
AAPL	1.0000	0.6041 (0.5156)	0.4888 (0.3891)
AMZN	0.6041 (0.5156)	1.0000	0.6042 (0.5051)
NFLX	0.4888 (0.3891)	0.6042 (0.5051)	1.0000

Table 5.24: Correlation matrix of the merged data (second case) with training data values in blue.

	AAPL	AMZN	NFLX
AAPL	1.0000	0.7700 (0.5156)	0.6598 (0.3891)
AMZN	0.7700 (0.5156)	1.0000	0.7601 (0.5051)
NFLX	0.6598 (0.3891)	0.7601 (0.5051)	1.0000

Table 5.25: Correlation matrix of the generated data (third case) with training data values in blue.

We can use various visualization methods and statistical tests to compare the distribution of the reconstructed merged data (obtained from each eigenvector) with the actual log returns. These methods include:

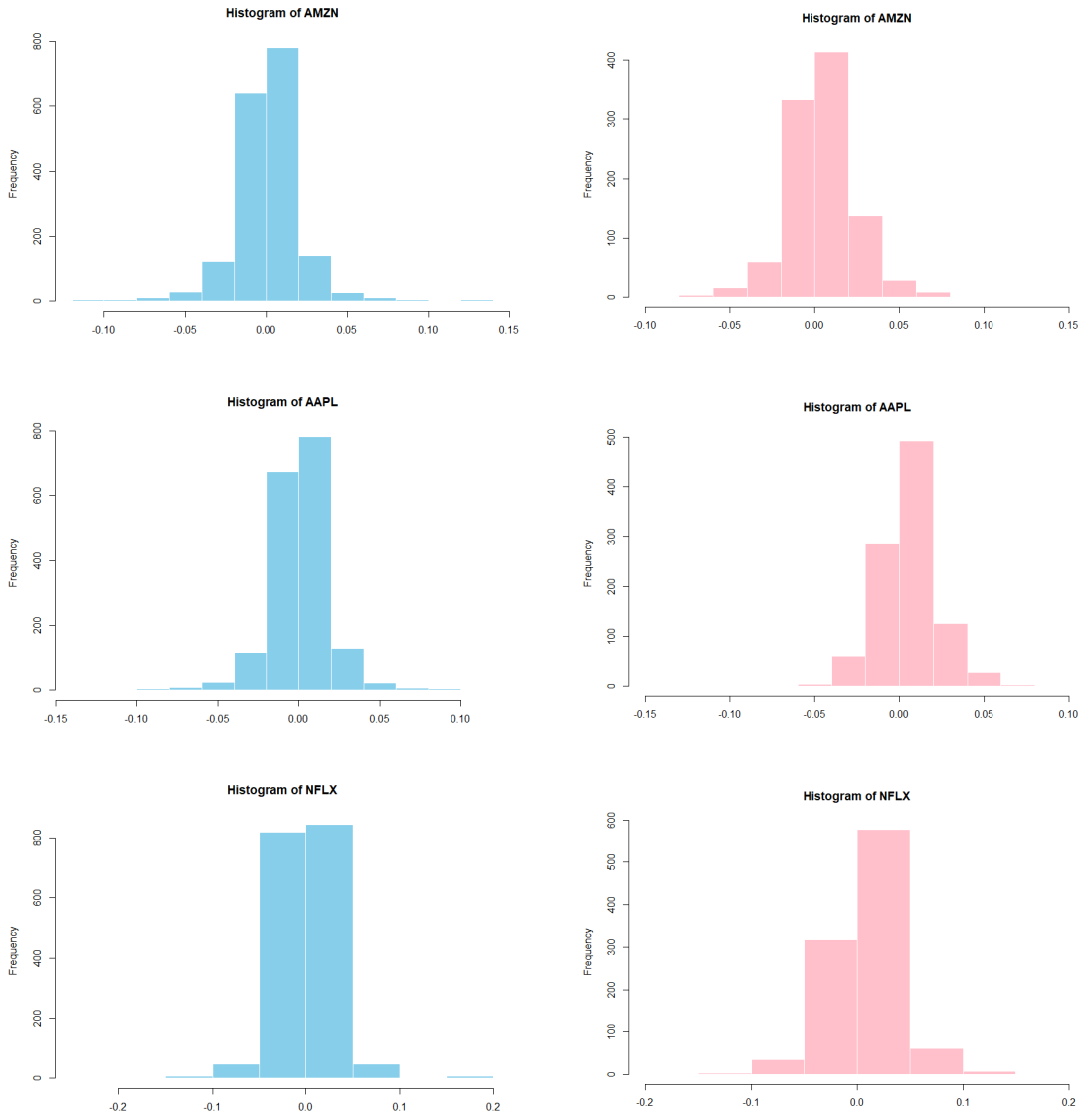


Figure 5.32: Histograms comparing AMZN, AAPL, and NFLX log returns: training data in blue, GAN-generated data in pink.

- **Visualization Techniques**: Empirical distribution functions, histograms, and boxplots.
- **Statistical Tests**: Kolmogorov–Smirnov test and Student's t-test.

In Figure 5.32, we present histograms for each stock: the training data is shown in *blue*, while the reconstructed data (obtained through PCA and GANs) is shown in *pink*.

The results from the *t-test* indicate no significant difference between the means of the log returns and the reconstructed data. On the other hand, the *Kolmogorov-Smirnov test* could pick the difference in the variances of the data. Hence, it rejects the null hypothesis, i.e., log returns and reconstructed data do not come from the same distribution according to this test. The KS test is sensitive to differences in the tails of the distributions. As we can see from the histograms the GANs capture the distribution of the data well, but not perfectly, the GAN-reconstructed data exhibit lighter tails and the KS test detects this discrepancy.

Improving the results with hyperparameter tuning

We experimented with different sets of hyperparameters for training the GAN, using a parameter grid to fine-tune the model. For each eigenvector, we applied these varied hyperparameters to generate new synthetic data. After merging this newly generated data with the historical training data, we optimized 5000 random portfolios to construct the Efficient Frontier shown in Figure 5.33. The resulting Efficient Frontier closely resembles the one obtained from the original training log returns, as shown in Figure 5.28.

Additionally, the correlation matrix for the merged data of train log returns with the GAN-generated data, presented in Table 5.26, shows only minor differences compared to the train log returns. This indicates that hyperparameter tuning has improved the generated data, allowing it to capture the structure of the training data more accurately while maintaining stable asset relationships.

	AAPL	AMZN	NFLX
AAPL	1.0000	0.5695 (0.5156)	0.3701 (0.3891)
AMZN	0.5695 (0.5156)	1.0000	0.6017 (0.5051)
NFLX	0.3701 (0.3891)	0.6017 (0.5051)	1.0000

Table 5.26: Correlation matrix of the train log returns merged with generated data using a different set of hyperparameters (training data values in blue).

When calculating cumulative returns, we find that portfolios optimized using GAN-generated data often deliver higher returns compared to those based on historical train data. However, the results from both datasets remain quite similar overall, indicating that the generated data captures the key patterns of the training data effectively. Notably, for longer time periods, portfolios with weights derived from the generated data usually yield higher cumulative returns than those based

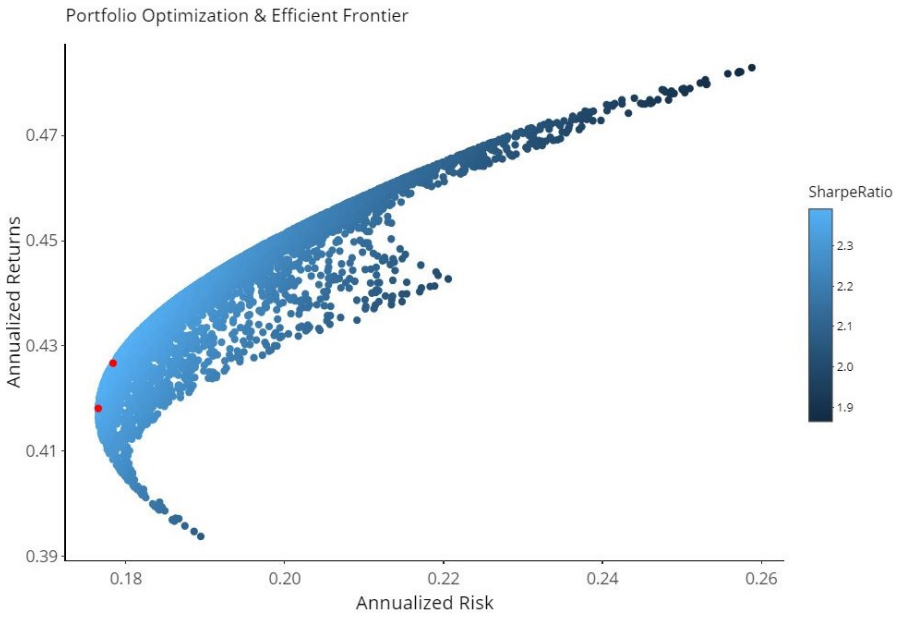


Figure 5.33: Efficient frontier for train log returns merged with generated data, using a different set of hyperparameters without any statistical test as e condition.

on historical data, suggesting that the stronger correlations in the generated data may enhance long-term investment performance.

Impact of Increasing Iterations on Generated Data

For the results we presented above, we used 1000 training steps. When increasing the number of training steps, we observed more stable training, improved convergence, and better generalization performance. Figure 5.34 illustrates how the training process evolves for the first eigenvector with different iteration counts. However, we must keep in mind that too many iterations may lead to overfitting, impacting the model's ability to generalize.

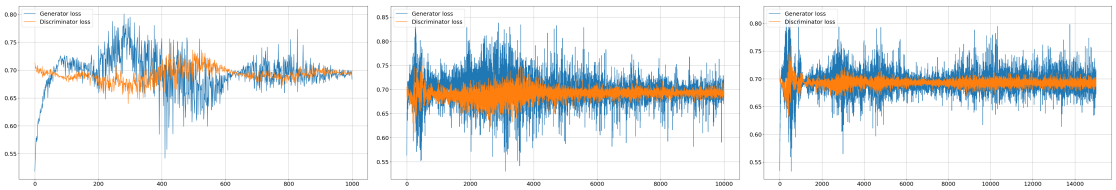


Figure 5.34: Discriminator's and Generator's loss during the training process across different iteration counts.

Increasing the number of iterations enhances the quality of the generated data, improving the histograms and producing a different Efficient Frontier, as seen in Figure 5.36.

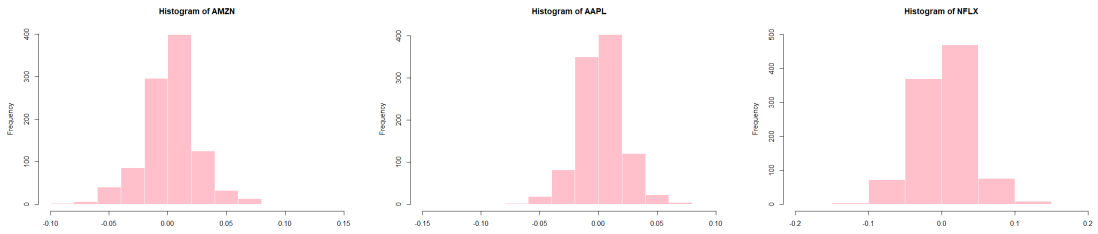


Figure 5.35: Histogram of AMZN, AAPL, and NFLX, from generated data with more training iterations.

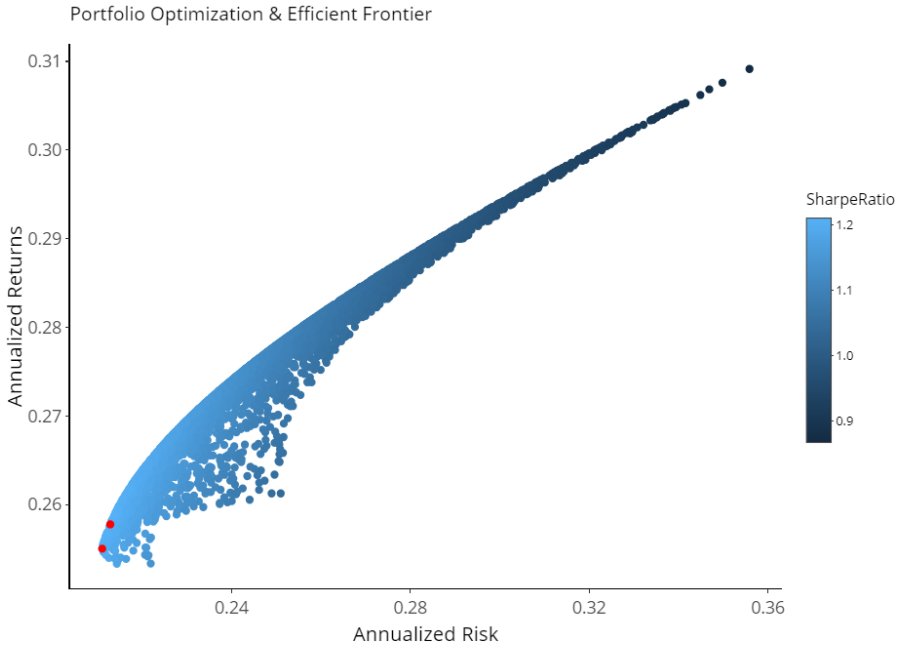


Figure 5.36: Efficient Frontier from generated data with more iterations, merged with the train data.

In this section, we generated data for the next day only. By extending the time window to 300 days, we observed a further improvement in the distribution of the data. Figure 5.37 and Table 5.27 present the histograms and mean estimates of this extended time window. The estimated mean from the generated data with an extended time window is closest to the actual data.

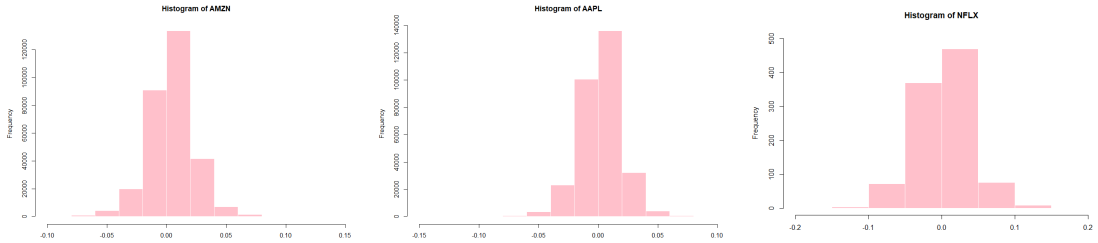


Figure 5.37: Histogram of AMZN, AAPL, and NFLX, from generated data with a 300-day time window.

	Apple	Amazon	Netflix
Complete data	0.001149	0.00105	0.00122
Train data	0.001139	0.001176	0.001294
Generated data	0.0018577	0.0014331	0.002468
Generated data with more iterations	0.0007451	0.000771	0.0009453
Generated data with a longer time window	0.0007196	0.0014025	0.001424

Table 5.27: Mean estimates of the complete, train, and generated data.

We get a better Efficient Frontier from the data with a longer time window, showing lower risk and higher returns, resulting in a higher Sharpe Ratio, as seen in Figure 5.38. Moreover, the generated data captures the structure of the train data effectively, leading to similar results. This raises the question of which data to trust more—while the generated data appear to enhance long-term performance, the close resemblance to the training data suggests both sets are reliable, making it difficult to definitively favor one over the other. In the next section, we will attempt to answer this question using synthetic data where we know that the conditions are satisfied.

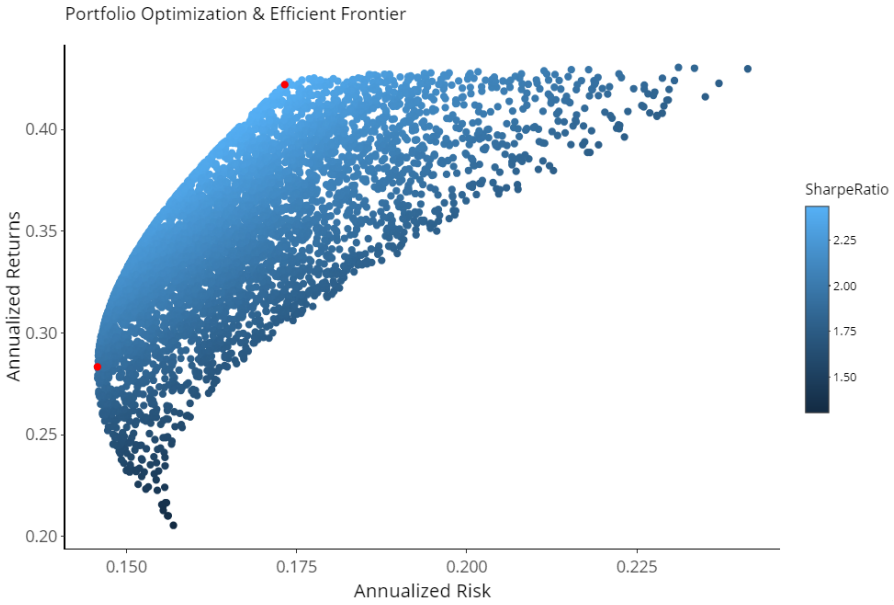


Figure 5.38: Efficient Frontier from generated data with an extended time window, merged with the train data.

Summary and Implications

In this section, we applied ForGAN-PCA to generate synthetic data for stock returns, focusing on both short-term (next day) and extended (300-day) time frames. Our results demonstrate that the generated data aligns well with the statistical properties of the training data, offering similar risk-return trade-offs and efficient frontiers. Hyperparameter tuning further improved the quality of the generated

data, enhancing its ability to capture the structure of the original dataset.

Through various tests, including cumulative return analysis and correlation matrix comparisons, we found that GAN-generated data not only closely resembles the training data but often provides better long-term returns. This suggests that synthetic data has the potential to improve portfolio optimization outcomes, especially for extended time horizons.

Despite these positive results, the similarity between the efficient frontiers and cumulative returns of the training and generated data raises an important question: can we confidently trust the generated data over historical data? While the generated data shows stronger correlations and higher returns in some cases, it remains difficult to definitively conclude which data set is more reliable. In the next section, we will further explore this question using purely synthetic data, where conditions can be controlled and evaluated.

5.5.1 Portfolio Optimization Using GANs and PCA for Synthetic Data: Examples and Analysis

First Example: Generating Log Returns Using TimeGAN and PCA with ForGAN for Synthetic Data

The primary objective of this section, as in previous sections, is to leverage Generative Adversarial Networks (GANs) to simulate additional log returns. However, in this case, we generate synthetic data where the conditions are more controlled and known. By generating more data, we aim to create a larger dataset, allowing for more reliable estimation of key parameters for portfolio optimization. To test our method of generating data with GANs for portfolio analysis, we also simulate synthetic data for the log returns of three assets using predefined means, volatilities, and correlations. The parameters for the log returns are as follows:

1. μ: The mean of the log returns, representing the expected growth rate of the asset.
2. σ: The volatility of the log returns, representing the standard deviation or risk of the asset.
3. ρ: The correlation coefficients between different assets, which ensure the assets move together in a way that reflects real-world data.

The specific values of μ, σ, and ρ used in our simulation are listed in Table 5.28, we have chosen a strong correlation between the log-increments to be able to compare the TimeGAN with our method, PCA with ForGAN, under conditions of high correlation.

The log returns for each asset are generated using the following equation:

$$r_i(t) = \mu_i - \frac{1}{2}\sigma_i^2 + \sigma_i Z_i(t)$$

Parameter	Value	Description
μ_1	0.05	Mean for asset 1
μ_2	0.04	Mean for asset 2
μ_3	0.03	Mean for asset 3
σ_1	0.25	Volatility for asset 1
σ_2	0.3	Volatility for asset 2
σ_3	0.2	Volatility for asset 3
ρ_{12}	0.9	Correlation coefficient between assets 1 and 2
ρ_{13}	0.8	Correlation coefficient between assets 1 and 3
ρ_{23}	0.7	Correlation coefficient between assets 2 and 3

Table 5.28: Parameters for Synthetic Data Generation.

where:

- $r_i(t)$ is the log return for asset i,
- μ_i is the mean (drift) for asset i,
- σ_i is the volatility for asset i,
- $Z_i(t)$ are correlated standard normal random variables.

The correlation between the log returns is imposed by applying Cholesky decomposition [3]. The idea is to transform independent random variables into correlated ones by applying the Cholesky decomposition to the correlation matrix.

$$Z(t) = L \cdot U(t)$$

where:

- L is the lower triangular matrix from the Cholesky decomposition of the correlation matrix,
- $U(t)$ is a vector of independent standard normal variables,
- $Z(t)$ is a vector of correlated normal variables after transformation.

The correlation matrix of our synthetic data is presented below 5.29. It demonstrates that we can effectively impose the desired correlations among the assets.

Figure 5.39 shows the generated synthetic log returns for three assets. The figure illustrates the dependency structure and simulated behavior of the log returns over time, based on 5000 simulated data points.

[3]is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is helpful for efficient numerical solutions, e.g., Monte Carlo simulations.

	X1	X2	X3
X1	1.0	0.905	0.801
X2	0.905	1.0	0.707
X3	0.801	0.707	1.0

Table 5.29: Correlation matrix of synthetic log returns after applying Cholesky decomposition.

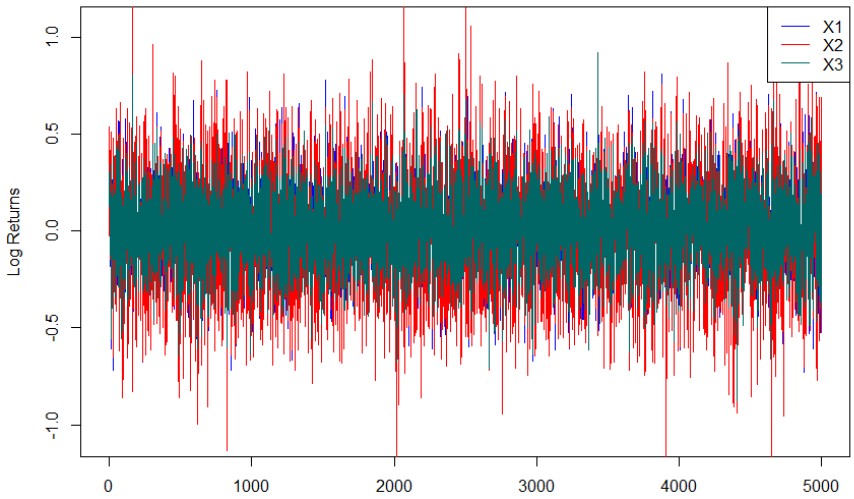


Figure 5.39: Generated synthetic log returns for three assets with imposed correlations using Cholesky decomposition.

The covariance matrix of the mean-centered synthetic log-return data is shown in Table 5.30. Following the same procedure as in Section 5.4, we apply Principal Component Analysis (PCA) to the synthetic log returns. We use the first 2000 data points from the generated synthetic data, remove the mean for standardization, and get the following eigenvectors 5.31 from the covariance matrix 5.30.

The eigenvectors represent the directions of maximum variance in the data, while the eigenvalues indicate the amount of variance explained by each corresponding eigenvector.

As shown in Table 5.32, the first principal component (PC1) explains the majority of the variance (88.94%), indicating that it captures the most significant patterns in the data. The second and third components explain smaller portions of the variance, with 8.29% and 2.77%, respectively. This suggests that most of the information in the dataset is concentrated in the first component, while the remaining components add less to the overall variance.

The principal components Z_1, Z_2, and Z_3 are linear combinations of the centered log returns X_1, X_2, and X_3 based on the eigenvectors derived from the covariance

	X1	X2	X3
X1	0.0635	0.0688	0.0390
X2	0.0688	0.0909	0.0408
X3	0.0390	0.0408	0.0381

Table 5.30: Covariance matrix of the mean-centered synthetic log returns used for PCA analysis.

Eigen vectors	0.592	0.802	0.0795
	0.705	-0.468	-0.532
	0.3897	-0.371	0.843
Eigen Values	0.1712	0.0053	0.0159

Table 5.31: Eigenvectors and eigenvalues derived from the covariance matrix of the synthetic data.

matrix of the synthetic data. Using the eigenvectors from Table 5.31, the principal components are given by:

- $Z_1 = 0.592X_1 + 0.705X_2 + 0.3897X_3$
- $Z_2 = 0.802X_1 - 0.468X_2 - 0.371X_3$
- $Z_3 = 0.0795X_1 - 0.532X_2 + 0.843X_3$

After transforming the data using PCA, we trained a Generative Adversarial Network (GAN) for each principal component (PC) individually and generated additional data points for a period of 300 days. Following this, we converted the generated data back into the log-returns space by applying the inverse transformation of the principal components. This back transformation allows us to create a larger universe of data from which we estimate the parameters for portfolio optimization.

We tested the GAN's performance for both 5000 and 50000 training steps to generate the data. In both cases, we observed a stable training process; however, increasing the number of iterations beyond 5000 did not enhance the quality of the generated data. As a result, we determined that 5000 training steps were sufficient to achieve reliable outcomes. The loss curves of the generator and dis-

Principal Component	**Explained Variance (%)**
PC1	88.94
PC2	8.29
PC3	2.77

Table 5.32: Explained variance for each principal component from PCA of synthetic log returns.

criminator during the training process for the 5000-step case are illustrated in Figures 5.40, 5.41, and 5.42.

In the figures, we observe stable loss patterns for both the generator and discriminator, indicating convergence during the training process.

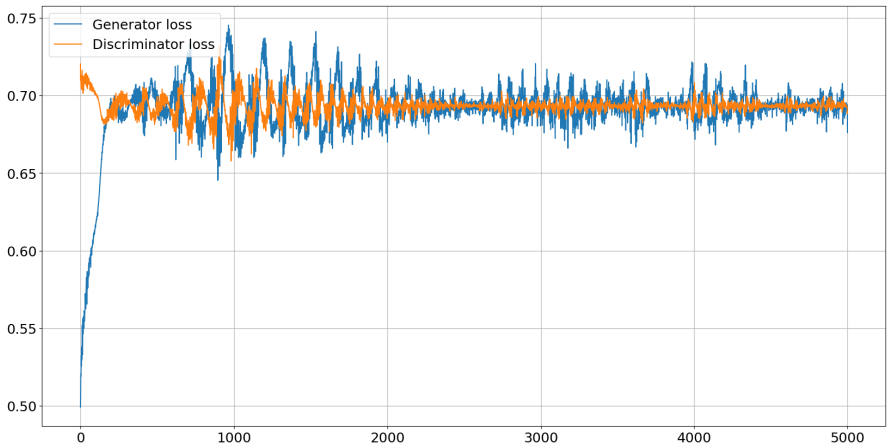


Figure 5.40: Loss curves of the generator and discriminator during training for the first principal component (PC) over 5000 steps.

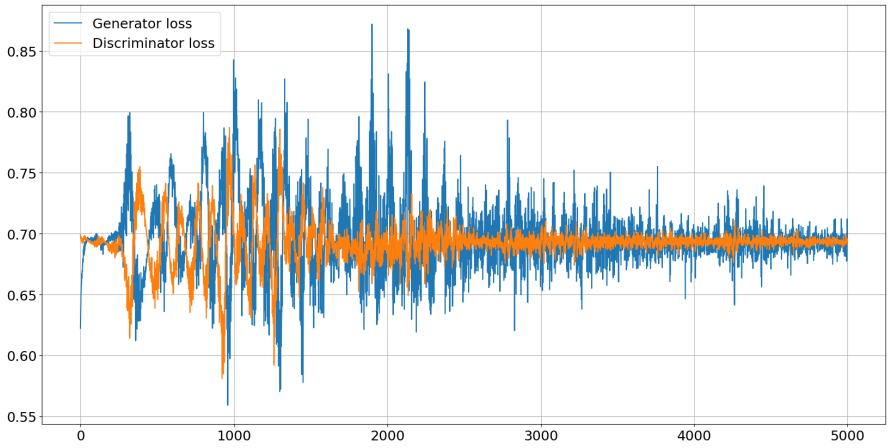


Figure 5.41: Loss curves of the generator and discriminator during training for the second principal component (PC) over 5000 steps.

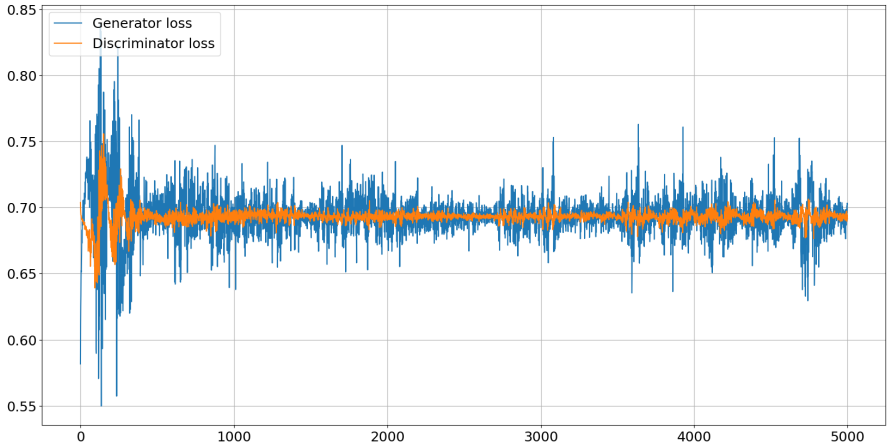


Figure 5.42: Loss curves of the generator and discriminator during training for the third principal component (PC) over 5000 steps.

To assess the quality of the generated data, we compare key statistical properties such as correlations, means, and covariances between the training data (2000 data points) and the data generated using PCA and ForGAN. The results are summarized in the following tables.

	X1	X2	X3
X1	1.0000 (1.0)	0.8782 (0.9)	0.7344 (0.8)
X2	0.8782 (0.9)	1.0000 (1.0)	0.5913 (0.7)
X3	0.7344 (0.8)	0.5913 (0.7)	1.0000 (1.0)

Table 5.33: Correlation matrix comparing generated data and training data (in blue).

	X1	X2	X3
Mean value	0.0062 (0.0205)	0.0054 (-0.0084)	-0.0214 (0.0076)

Table 5.34: Means of the generated data compared to the training data (in blue).

	X1	X2	X3
X1	0.0474 (0.0635)	0.0489 (0.0688)	0.0282 (0.0389)
X2	0.0489 (0.0688)	0.0685 (0.0909)	0.0268 (0.0408)
X3	0.0282 (0.0389)	0.0268 (0.0408)	0.0331 (0.0381)

Table 5.35: Covariance matrix comparing generated data and training data (in blue).

Based on the presented tables, 5.33, 5.34, and 5.35, we can say that the generated data are fairly close to the training data in terms of correlation. However, there are some deviations in terms of the means and covariances. For example, the covariance values of the generated data are noticeably smaller than those of the training data (blue), this means that the generated data exhibits less variability and weaker relationships between variables than the training data.

Additionally, it is worth noting that the means have higher relative deviations compared to the covariances. This observation will become more apparent when we delve into the continuous time portfolio problem.

To further compare our methods, we generate data using TimeGAN. In the previous section (Section 5.4), TimeGAN was utilized to simulate stock price data for Apple, Amazon, and Netflix. Here, we will generate data for the synthetic log-returns shown in Figure 5.39.

The training and the generation process remain the same as we have described in the previous sections 5.2, 5.4. We again used PCA and t-SNE to compare the training synthetic log-returned data and TimeGAN-generated data visually.

Figures 5.43 and 5.44 show that the synthetic data generated by TimeGAN keep the overall structure and relationships of the original data.

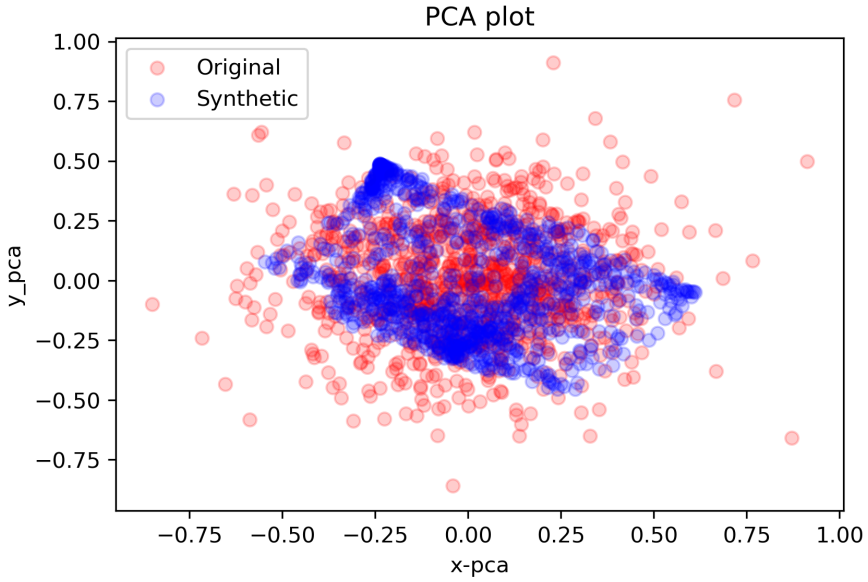


Figure 5.43: PCA plot comparing the original and TimeGAN-generated data.

For the PCA, we reduce the data's dimensionality using the first two principal components and plot the results in a scatter plot, where the original data is represented in red and the synthetic data in blue. The PCA plot shows that the red (original) and blue (generated) data points overlap quite well in the central region. However, the generated data are more dense in specific areas compared to the original ones, this means that TimeGAN (as we have mentioned before)

creates a stronger correlation within the generated data compared to the original. So, TimeGAN is performing well at replicating the main structure of the original data. Still, since the generated data show fewer outliers, it might be missing to capture some of the finer characteristics of the original dataset.

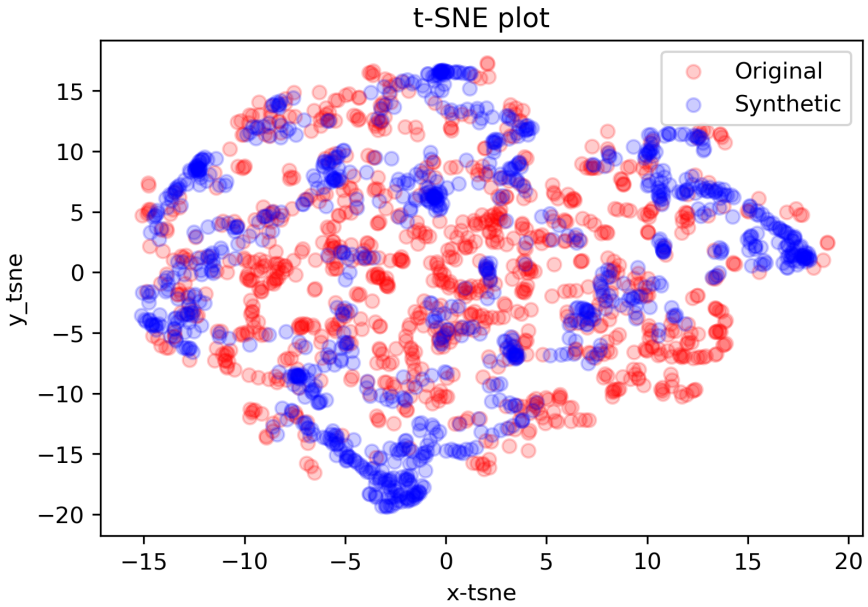


Figure 5.44: t-SNE plot showing the distribution of original versus TimeGAN-generated data.

t-SNE further reduces the data to a two-dimensional space, focusing on preserving local data structures. The plot visualizes these reduced dimensions. Similarly, the t-SNE plot demonstrates that the blue data projects the red ones quite closely in a lower-dimensional space, meaning that the model has learned the patterns of the original data. However, there are regions where the two distributions do not overlap.

To evaluate the quality of the generated data from TimeGAN, we compare the statistical measures—means, covariances, and correlations—of both the training and generated datasets.

	X1	X2	X3
Mean value	0.0483 (0.0205)	0.0196 (-0.0084)	0.027 (0.0076)

Table 5.36: Means of the generated data compared to the training data (in blue).

From Tables 5.36, 5.37, and 5.38, we see that the means and covariance values between the training and generated data are fairly close, with only some small deviations. However, the TimeGAN tends to overfit the relationships between variables compared to the original data, as noted in Section 5.4 for the stock

prices example. So we can say that TimeGAN has more or less generated three perfectly dependent variables, however it produced better estimates for the means.

	X1	X2	X3
X1	1.0000 (1.0)	0.9996 (0.9)	0.9993 (0.8)
X2	0.9996 (0.9)	1.0000 (1.0)	0.9991 (0.7)
X3	0.9993 (0.8)	0.9991 (0.7)	1.0000 (1.0)

Table 5.37: Correlation matrix comparing generated data and training data (in blue).

	X1	X2	X3
X1	0.0412 (0.0635)	0.0478 (0.0688)	0.0288 (0.0390)
X2	0.0478 (0.0688)	0.0556 (0.0909)	0.0334 (0.0408)
X3	0.0288 (0.0390)	0.0334 (0.0408)	0.0201 (0.0381)

Table 5.38: Covariance matrix comparing generated data and training data (in blue).

Summary and Implications

PCA with ForGAN effectively captures the correlation structure of the original dataset, which is crucial for risk management in portfolio optimization. However, while it generally captures the overall structure, it may slightly underestimate volatilities and covariances, leading to some deviations that could affect the optimal portfolio weights.

TimeGAN, on the other hand, appears to focus more on capturing temporal dependencies rather than exact statistical properties. However, an important limitation of TimeGAN is that it tends to produce a nearly singular correlation matrix, meaning that the relationships between variables are oversimplified. As observed in our analysis, this often results in a numerically low-rank matrix, close to 1, indicating that TimeGAN overfits the interdependencies among variables.

This overfitting issue arises because TimeGAN tends to reduce the dimensionality of the data during the generation process. This dimensionality reduction compresses the time series data into a smaller, lower-dimensional space using an embedding network, followed by an attempt to reconstruct the original data. While this method captures important patterns, it adds a layer of complexity that can lead to inaccuracies, especially in how it represents the relationships between different time series. As a result, even though TimeGAN is good at capturing temporal dependencies, its tendency to overfit the relationships among variables may prevent it from capturing the full variability and detailed features of the original data.

Second Example: Generating Log Returns Using PCA with ForGAN for Synthetic Data

In this example, we simulate synthetic log returns for three assets with predefined means, volatilities, and correlations. Unlike the previous case, we use lower correlation values to explore how different correlation structures influence the generated data. We generate 5000 data points to enable a more comprehensive analysis. The correlation structure is imposed on uncorrelated standard normal variables using Cholesky decomposition. The chosen parameters are detailed in Table 5.39.

Parameter	**Value**	**Description**
μ_1	0.05	Mean for asset 1
μ_2	0.04	Mean for asset 2
μ_3	0.03	Mean for asset 3
σ_1	0.1	Volatility for asset 1
σ_2	0.15	Volatility for asset 2
σ_3	0.2	Volatility for asset 3
ρ_{12}	0.5	Correlation coefficient between assets 1 and 2
ρ_{13}	0.3	Correlation coefficient between assets 1 and 3
ρ_{23}	0.4	Correlation coefficient between assets 2 and 3

Table 5.39: Parameters for synthetic data generation.

The log returns for each asset are generated using the following equation:

$$X_{t,i} = \left(\mu_i - \frac{1}{2}\sigma_i^2\right) \cdot t + \sigma_i \cdot W_{t,i},$$

where $X_{t,i}$ represents the log return for asset i at time t, μ_i is the mean (drift) for asset i, σ_i is the volatility for asset i, and $W_{t,i}$ is the correlated random variable (Wiener process) for asset i at time t.

The simulated log returns are illustrated in Figure 5.45.

From Table 5.40, we can see that we were able to impose the wanted correlation to our data quite well.

	X1	X2	X3
X1	1.000	0.501	0.291
X2	0.501	1.000	0.391
X3	0.291	0.391	1.000

Table 5.40: Correlation matrix of dependent synthetic data.

For the PCA of the simulated log returns, we compute the eigenvalues and eigenvectors of the covariance matrix, following the same methodology used in previous

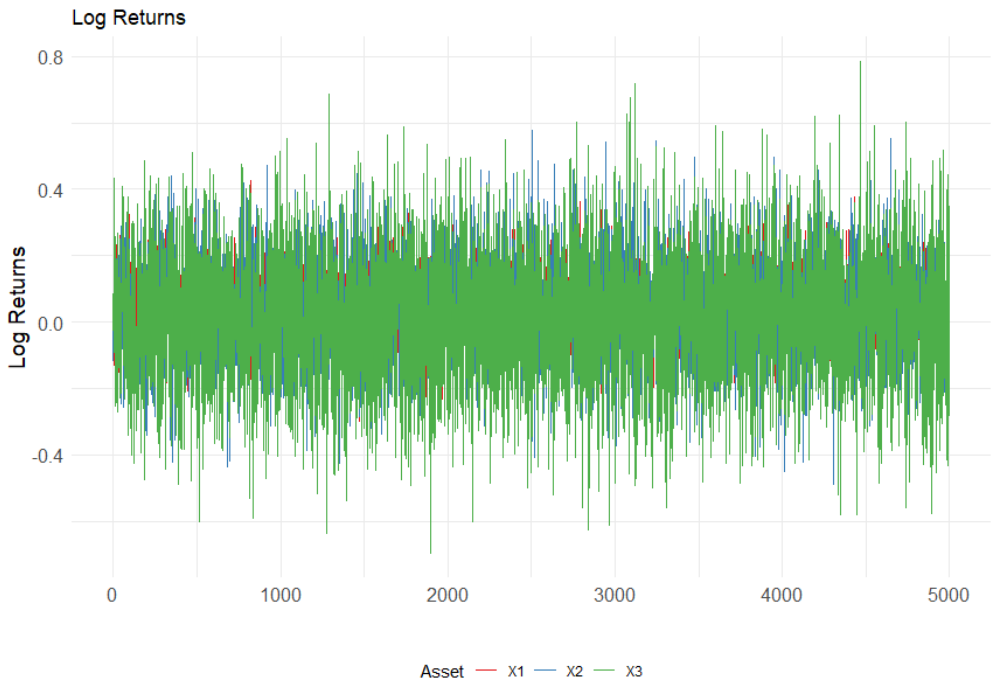


Figure 5.45: Simulated annual log returns for three assets, reflecting specified means, volatilities, and correlations.

examples. We specifically chose 1768 data points for training to stay consistent with the case of log increments of stock prices. Before applying PCA, we standardize the data by removing the mean, ensuring that the data is centered for an accurate covariance matrix calculation. This consistent approach across cases ensures coherence in our analysis.

	X1	X2	X3
X1	0.010	0.007	0.006
X2	0.007	0.022	0.012
X3	0.006	0.012	0.043

Table 5.41: Covariance matrix of the synthetic log returns for assets X1, X2, and X3.

Eigen vectors	0.050	-0.216	-0.903
	0.006	-0.428	0.429
	-0.006	0.066	0.998
Eigen Values	0.018	-0.877	0.013

Table 5.42: Eigenvectors and eigenvalues of the covariance matrix for the synthetic log returns.

The principal components Z_1, Z_2, and Z_3 are computed as linear combinations of

the centered log returns X_1, X_2, and X_3 utilizing the eigenvectors obtained from the covariance matrix of the synthetic data. Based on the eigenvectors listed in Table 5.42, the formulations for the principal components are as follows:

- $Z_1 = 0.050X_1 + 0.0.006X_2 - 0.006X_3$
- $Z_2 = -0.216X_1 - 0.428X_2 + 0.066X_3$
- $Z_3 = -0.903X_1 + 0.429X_2 + 0.998X_3$

From each principal component, we will generate data for a longer time window of 300 days and repeat this process 1000 times. Following this, we reconstruct the data back into log returns. In Figure 5.46, we compare the histogram of the distribution of the training log returns for each asset with the histogram of the generated log returns. Our results indicate that we can capture the distribution of the data quite well.

The tables presented below compare key statistical measures-means, correlations, and covariances-between the generated data and the training data. As shown in Table 5.43, the generated data's means are very close to those of the training data, indicating that the GAN was able to replicate the average behavior of the assets with high accuracy. The correlation matrix (Table 5.44) also shows that the relationships between the assets in the generated data align well with those in the training data, with small deviations in correlation values.

The covariance matrix (Table 5.45) further confirms that the generated data closely mirrors the variance and covariances of the training data, showing consistency in the magnitude of variability and the interrelationships among the three assets.

	X1	X2	X3
Mean value	0.049 (0.046)	0.031 (0.027)	0.011 (0.009)

Table 5.43: Means of the generated data compared to the training data (in blue).

	X1	X2	X3
X1	1.000 (1.000)	0.533 (0.497)	0.285 (0.289)
X2	0.533 (0.497)	1.000 (1.000)	0.415 (0.383)
X3	0.285 (0.289)	0.415 (0.383)	1.000 (1.000)

Table 5.44: Correlation matrix comparing generated data and training data (in blue).

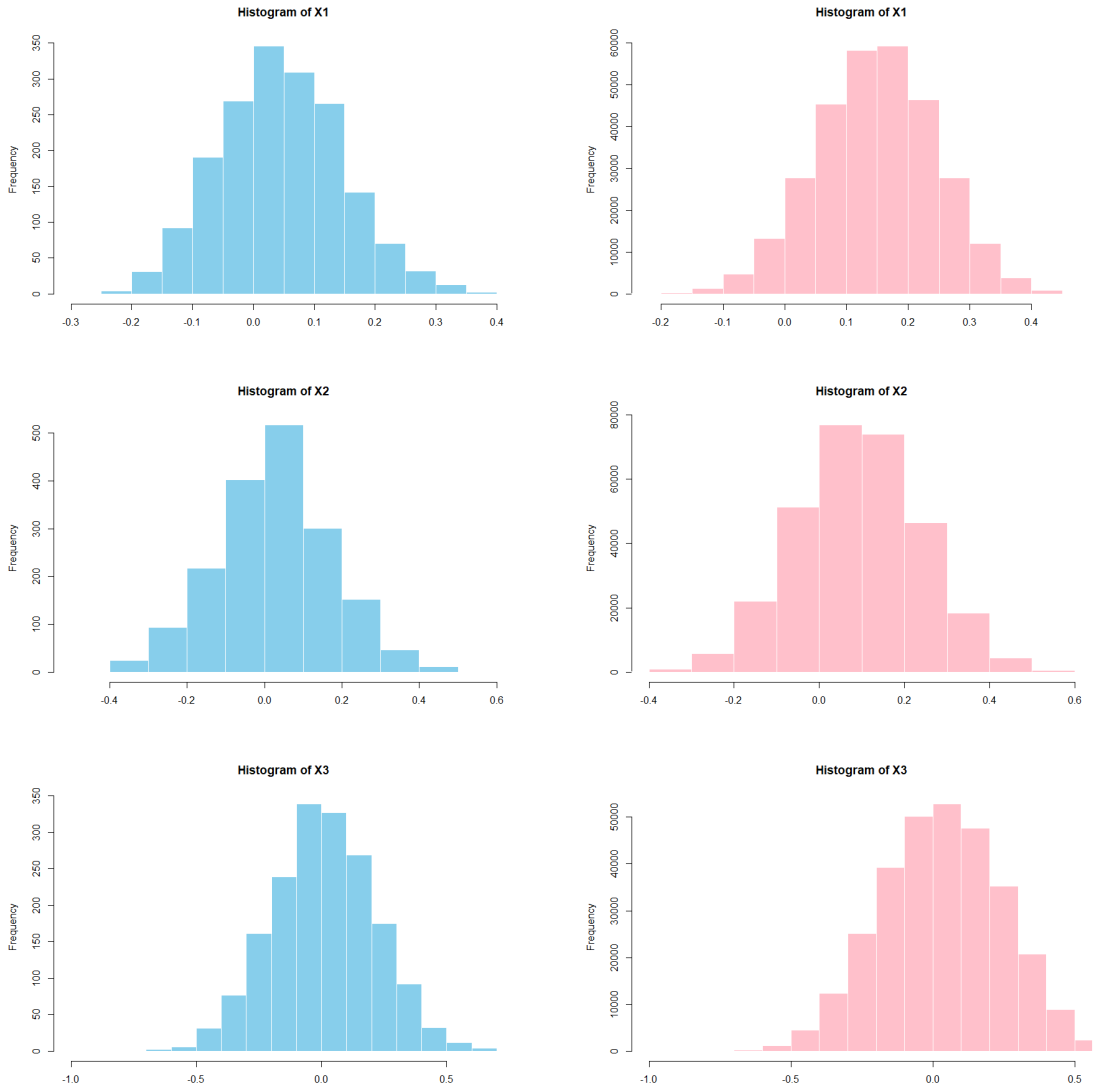


Figure 5.46: Histograms comparing training log returns (blue) and generated log returns (pink) for assets X1, X2, and X3.

	X1	X2	X3
X1	0.008 (0.0101)	0.007 (0.0074)	0.006 (0.0060)
X2	0.007 (0.0074)	0.022 (0.0219)	0.013 (0.0117)
X3	0.006 (0.0060)	0.013 (0.0117)	0.047 (0.0428)

Table 5.45: Covariance matrix comparing generated data and training data (in blue).

In Figure 5.47, we present the Efficient Frontiers for both the training and generated log returns. To construct these frontiers, we generated 5000 random portfolios by assigning random weights to the three assets. For each portfolio, we calculated

the portfolio's mean return and covariance matrix, allowing us to estimate the risk and return for each portfolio. The portfolios that maximize the Sharpe ratio and minimize variance are marked in red on the Efficient Frontiers, representing their means and variances. The shapes and the small sizes of the efficient frontier can be attributed to the fact that the third asset performs significantly worse than the other two, with the first asset clearly dominating. The corresponding portfolio weights (vectors) for these optimal portfolios are computed and presented in Table 5.46.

Data	**X1**	**X2**	**X3**	**Return**	**Risk**	**SharpeRatio**
Min Var (Train)	0.811	0.119	0.0697	0.0408	0.0976	0.418
Max SR (Train)	0.969	0.00124	0.0293	0.0445	0.0996	0.447
Min Var (Generated)	0.884	0.0729	0.0427	0.0464	0.0551	0.842
Max SR (Generated)	0.969	0.00124	0.0293	0.0483	0.0555	0.870

Table 5.46: Comparison of Train Data (in blue) and Generated Data for Min Variance and Max Sharpe Ratio portfolios, including the corresponding portfolio weights, risk, and returns.

The comparison between the training and generated data frontiers shows that the generated data captures the essential characteristics of the training data well. Both frontiers exhibit a similar curve, indicating that the GAN successfully modeled the risk-return dynamics present in the training data.

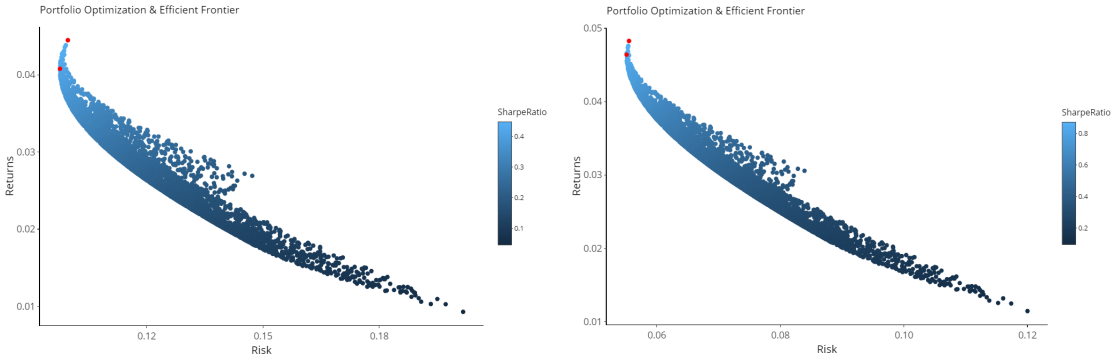


Figure 5.47: Comparison of Efficient Frontiers for the training and generated log returns, highlighting optimal portfolios with minimum variance and maximum Sharpe ratio (red points).

Summary and Implications

The comparison of histograms, correlation, covariance matrices, and Efficient Frontiers shows that the GAN-based model (PCA with ForGAN) effectively replicates the statistical properties of the training data. The generated log returns closely match the training data in terms of mean values, covariance, and correlation. While there are minor deviations in the covariance matrix and the shape of the

Efficient Frontiers, this is expected since the generated data covers only 300 days, not the entire period. The synthetic data generated closely aligns with the training data, offering a useful tool for improving the estimation of key parameters like returns and risk. This expanded dataset can support more reliable portfolio allocation decisions which will be the subject of the second example below.

5.5.2 Portfolio Optimization and Expected Growth Rate Maximization

In previous sections, we observed that the generated data often yielded better estimates for key parameters, resulting in Efficient Frontiers with higher returns and lower risk. However, the question remains: which data is more reliable—the historical training data or the generated data from GANs? In this section, we explore the reliability of the data generated by GANs by comparing it with synthetic data where the exact solution is known. This controlled approach will allow us to assess the accuracy of our generated data in a portfolio optimization context.

We consider two risky securities with prices modeled by geometric Brownian motion, which reflects the stochastic nature of financial markets. The prices of the securities are given by:

$$\begin{aligned} S_1(t) &= s_1 e^{\left(b_1 - \frac{1}{2}\sigma_1^2\right)t + \sigma_1 W_1(t)} \\ S_2(t) &= s_2 e^{\left(b_2 - \frac{1}{2}\sigma_2^2\right)t + \sigma_2\left(\rho W_1(t) + \sqrt{1-\rho^2} W_2(t)\right)} \end{aligned} \tag{5.34}$$

Here, s_1 and s_2 are the initial prices of the two securities, b_1 and b_2 represent the drift rates, σ_1 and σ_2 denote the volatilities, and $W_1(t)$ and $W_2(t)$ are standard Brownian motions. The correlation between the two assets is captured by ρ. The correlation ρ between the two assets is essential as it impacts diversification and, consequently, the portfolio's risk level. By adjusting ρ, we can see how different correlation structures affect optimization results.

The evolution of wealth in a portfolio that allocates wealth according to a strategy π is given in Equation 5.35.

$$\begin{aligned} X^{\pi}(t) = x \cdot e^{\int_0^t \left(b_2 + \pi(s)(b_1 - b_2) - \frac{1}{2}(\pi(s)(\sigma_1 - \sigma_2\rho) + \sigma_2\rho)^2 \right.} \\ \left. + (\sigma_2\sqrt{1-\rho^2} - \pi(s)\sigma_2\sqrt{1-\rho^2})^2)ds + \int_1^t \ldots dW(s) \right. \end{aligned} \tag{5.35}$$

This equation captures the dynamics of portfolio growth, where x is the initial wealth and $\pi(s)$ represents the fraction of wealth invested in security S_1 at time s.

To determine the optimal investment strategy that maximizes the expected growth rate of the portfolio, we seek to solve the following optimization problem:

$$\max_{\pi(.)} \mathbb{E}(\ln(X^{\pi}(T))) \quad \text{with solution} \quad \pi^*(t) = \frac{b_1 - b_2 - \sigma_1\sigma_2\rho + \sigma_2^2}{\sigma_1^2 - 2\sigma_1\sigma_2\rho + \sigma_2^2} \tag{5.36}$$

The optimal strategy $\pi^*(t)$ gives the fraction of wealth to be allocated to the first security S_1 at time t. This solution balances the expected returns against the risks associated with the securities, taking into account their volatilities and correlation.

First, we generate the prices of the two risky securities using the equations provided in 5.34. After simulating the prices, we take the log returns, as illustrated in Figure 5.48. The parameters used for generating the prices are summarized in Table 5.47, where b_1 and b_2 represent the expected returns, σ_1 and σ_2 represent the volatilities, and ρ denotes the correlation between the returns of the two securities.

b_1	σ_1	b_2	σ_2	ρ
0.05	0.3	0.04	0.2	0.5

Table 5.47: Parameters used for generating the prices of securities S1 and S2.

In this case, we have set the expected returns $b_1 = 0.05$ for security S1 and $b_2 = 0.04$ for security S2, indicating a slightly higher expected return for S1. The volatilities are set to $\sigma_1 = 0.3$ and $\sigma_2 = 0.2$, showing that S1 has a higher risk compared to S2. The correlation between the securities is set at $\rho = 0.5$, indicating a moderate positive relationship between their returns.

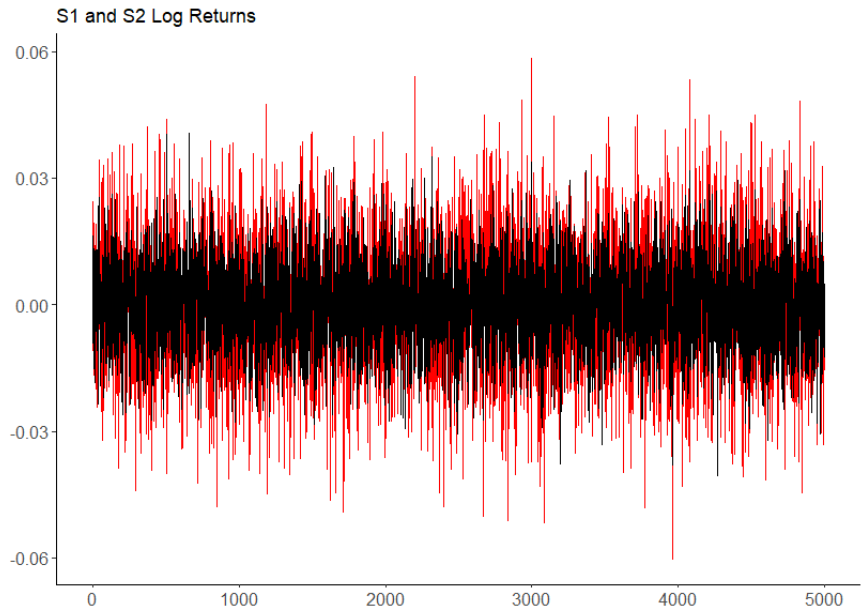


Figure 5.48: Log returns for securities S1 and S2, illustrating their distribution and characteristics.

We utilize 500 data points for principal component analysis (PCA). As previously established, we will use the transformed PC data to generate 2000 data points for the next day through Generative Adversarial Networks (GANs).

The covariance matrix of the log returns for S1 and S2 is shown in Table 5.48.

	S1	S2
S1	0.000234	0.000068
S2	0.000068	0.000103

Table 5.48: Covariance matrix for the log returns of securities S1 and S2.

In addition, the eigenvectors and eigenvalues corresponding to the covariance matrix of the log returns of S1 and S2 are presented in Table 5.49. The eigenvalues indicate the amount of variance captured by each principal component, with the first eigenvalue capturing the most variance, while the eigenvectors represent the direction of the principal components in the data space.

Eigen vectors	0.920	-0.391
	0.391	0.920
Eigen Values	0.000263	0.0000738

Table 5.49: Eigenvalues and corresponding eigenvectors of the covariance matrix for the log returns of S1 and S2.

The explained variance of the principal components is summarized in the following Table 5.50:

Explained variance	78.061	21.939
Cumulative explained variance	78.061	100.

Table 5.50: Explained variance and cumulative explained variance of the principal components.

The first principal component accounts for 78.061% of the total variance, making it crucial for accurately modeling the asset returns. This component will have a dominant influence on the portfolio's expected return and risk. While the second component accounts for 21.939%, indicating that the first component captures the majority of the data structure. However, the portion of variance explained by the second component will not allow for a dimension reduction.

We generate additional synthetic data using 1500 iterations for each principal component. The training process, shown in Figure 5.49, demonstrates a stable training process, particularly for the first principal component, which captures the majority of the data's variance.

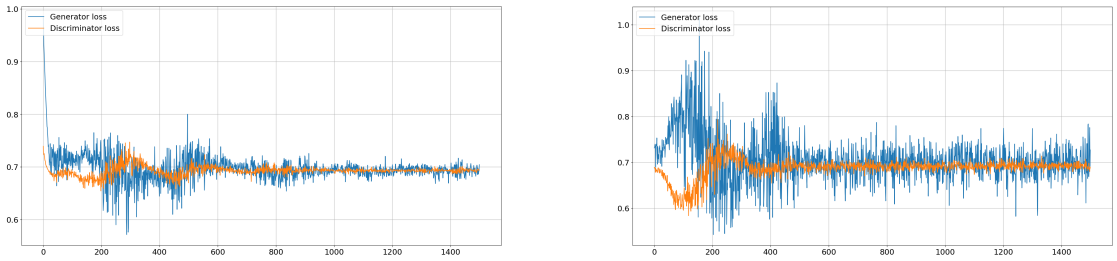


Figure 5.49: Generator's and Discriminator's loss curves during the training process for the first (left) and second (right) principal components.

The histograms of the training data and the reconstructed generated data are presented in Figure 5.50. While the histograms for the training and generated data show noticeable similarities, they are not identical. This discrepancy is expected since we are generating data for only the next day. Additionally, the generated dataset contains more data points, resulting in a different scale on the y-axis compared to the training data histogram.

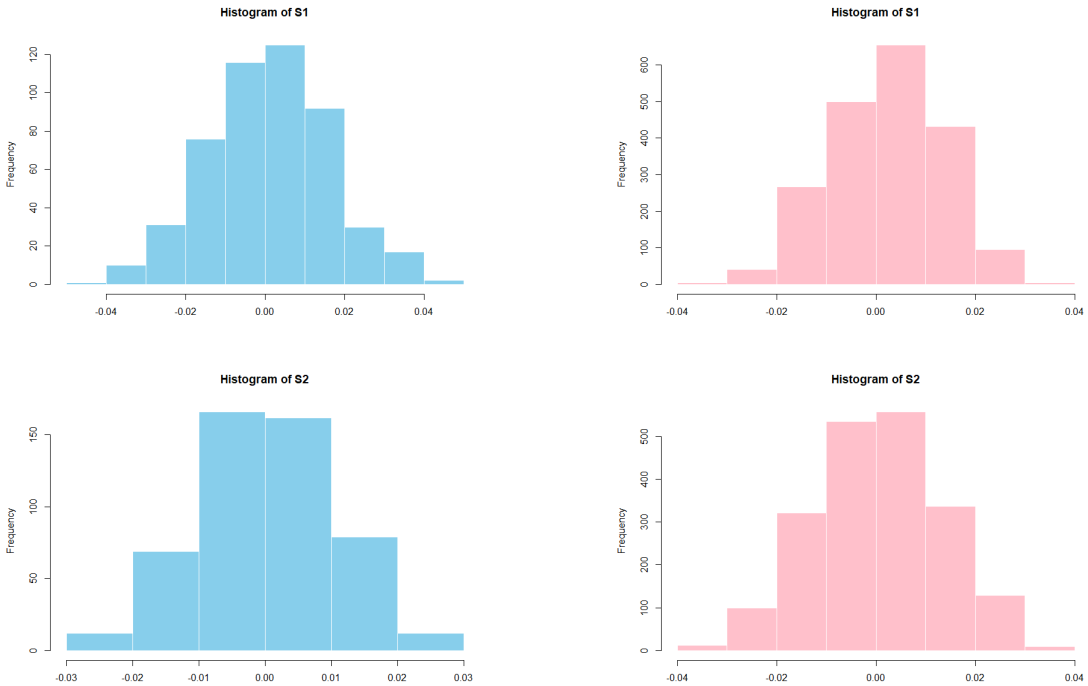


Figure 5.50: Histograms of log returns for securities S1 (top row) and S2 (bottom row): blue bars represent training data, while pink bars represent generated data.

When calculating the optimal π^* with the parameters from Table 5.47 we get the following value:

$$\pi^* = 0.2857.$$

This optimal investment strategy indicates that we should allocate 28.57% of our wealth to the first asset and the remaining 71.43% to the second one.

From the simulated synthetic train log return data and the additional data points generated from GANs using the PCA method, we estimate the parameters $b_1, b_2, \sigma_1, \sigma_2, \rho$, and $\hat{b_1}, \hat{b_2}, \hat{\sigma_1}, \hat{\sigma_2}, \hat{\rho}$, respectively (Table 5.51). Compared to the target parameters (Table 5.47), the estimates from the train data are closer to the actual values, especially for the drift and volatility terms. The GAN-generated data deviates more, particularly underestimating correlation and overestimating σ_2, which may affect portfolio diversification.

It is important to note that the estimated parameter values in the table below are rounded for presentation, but we maintain precision up to seven decimal places when calculating the optimal π values to ensure accuracy.

	b_1	σ_1	b_2	σ_2	ρ
Generated train data	0.045	0.29	0.019	0.19	0.49
Generated GANs data	0.026	0.22	0.03	0.24	0.141

Table 5.51: Estimated parameters from the generated data, comparing values from simulated train data and data generated by GANs.

The optimal investment allocation values calculated from the generated data are as follows:

$$\text{GANs generated data} \quad \pi^* = 0.52, \qquad \text{train generated data} \quad \pi^* = 0.502.$$

This suggests that the optimal allocation for the GAN-generated data is 52% in the first asset, indicating a slightly more aggressive investment strategy compared to the 50.2% allocation from the training-generated data. However, both estimated optimal portfolios deviate substantially from the optimal one of 0.2857.

To assess whether we can improve the estimated parameters by increasing the number of training data points from 500 to 1000, we refer to Table 5.52.

	b_1	σ_1	b_2	σ_2	ρ
Generated train data	0.0436	0.293	0.0199	0.197	0.434
Generated GANs data	0.022	0.204	0.0227	0.208	0.0267

Table 5.52: Estimated parameters from generated data with increased training points.

From the parameters listed in Table 5.52, we obtain the following optimal π^*.

$$\text{GANs generated data} \quad \pi^* = 0.499, \qquad \text{train generated data} \quad \pi^* = 0.502.$$

The results show a slight improvement, as the optimal π^* derived from the generated data is now closer to the theoretically optimal allocation ($\pi^* = 0.2857$). This suggests that more training data enhances the model's accuracy. Moreover, while the parameter estimates obtained from the simulated train data are more accurate, the optimal portfolio strategy derived from the GAN-generated data consistently remains closer to the theoretically optimal one. This trend will be observed in the following cases as well, further supporting the effectiveness of the GAN-generated data in portfolio optimization.

Improved results

Moreover, we observed enhancements in the estimated parameters by fine-tuning our approach during the generation process. The updated estimated parameters from the generated data are summarized in Table 5.53.

	b_1	σ_1	b_2	σ_2	ρ
Generated train data	0.0436	0.293	0.0199	0.197	0.434
Generated GANs data	0.0366	0.267	0.0163	0.172	0.476

Table 5.53: Estimated parameters from generated data with hyperparameter tuning.

From the parameters in Table 5.53, we derive the following optimal π^*. The GAN-generated data now provides closer estimates for the drift terms (b_1, b_2) and the correlation (ρ) compared to the training data. However, the volatilities (σ_1, σ_2) from the GAN-generated data, while improved, still deviate slightly more from the target values than the estimates from the training data.

$$\text{GANs generated data} \quad \pi^* = 0.488, \qquad \text{train generated data} \quad \pi^* = 0.502.$$

To achieve these improved results, we explored various hyperparameters during the training of the GAN, while also paying close attention to the histograms of the generated data. This way we got a π^* closer to the theoretically optimal value.

Generating and Analyzing Synthetic Log Returns: Second Case

In this example, we directly generate 1000 log returns for the two securities, S_1 and S_2, using the same parameters as previously detailed in Table 5.47. This approach contrasts with the first case, where we generated prices and subsequently calculated the log returns. Figure 5.51 illustrates the log returns of both securities, and we use those data for further analysis.

Further, we generate an additional 5000 data points using GANs for the next day from each principal component, applying the same methodology as in the previous example. After generating these data points, we reconstruct them back into their original log return format for further analysis.

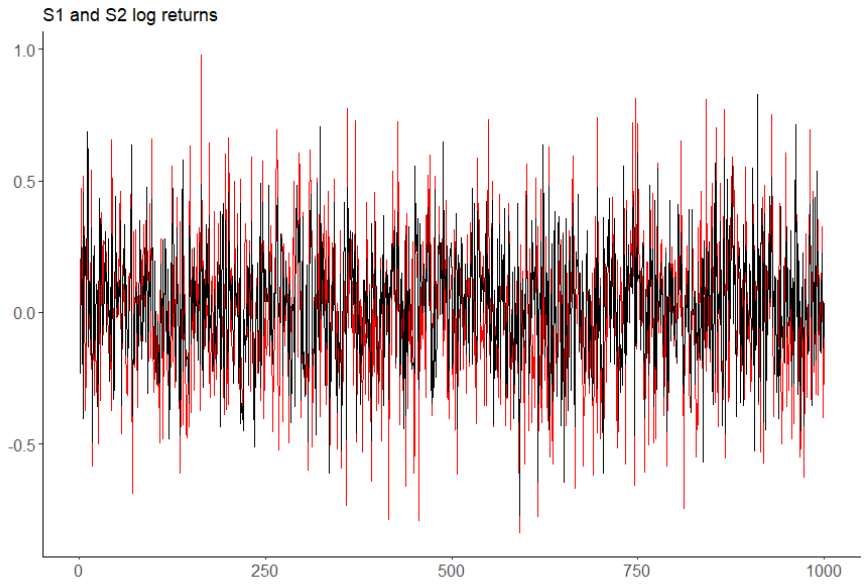


Figure 5.51: Log returns for securities S1 (in black) and S2 (in red).

We proceed to calculate the covariance matrix to understand the relationship between the two assets. The covariance matrix is presented in the Table 5.54 below.

	S1	S2
S1	0.0885	0.0487
S2	0.0487	0.0572

Table 5.54: Covariance matrix of S1 and S2 log returns.

Next, we compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components that capture the variance within the log returns. Table 5.55 displays these values, revealing the directions in which the variance is maximized (eigenvectors) and the magnitude of that variance (eigenvalues). A higher eigenvalue indicates a principal component that explains more variance in the data.

Eigen vectors	0.8079	-0.589
	0.589	0.8079
Eigen Values	0.124	0.0216

Table 5.55: Eigenvectors and eigenvalues of the covariance matrix for S1 and S2 log returns.

We also calculate the explained variance for each principal component to assess how much variance each component captures from the total dataset. As shown

in the following Table 5.56, the first principal component explains 85.139% of the variance, indicating it is the most significant factor in the data.

Explained variance	85.139	14.86
Cumulative explained variance	85.139	100.

Table 5.56: Explained variance and cumulative explained variance for the principal components derived from the log returns.

Finally, we generate an additional 5000 data points for the next day from each principal component using GANs and reconstruct the data to analyze its performance. This approach ensures that we maintain the variance structure captured by the principal components while generating synthetic data that closely resembles the original log returns.

We generate the data using 3000 iterations for each principal component, and the training process is illustrated in Figure 5.52. The training shows a stable convergence for both the Generator and the Discriminator.

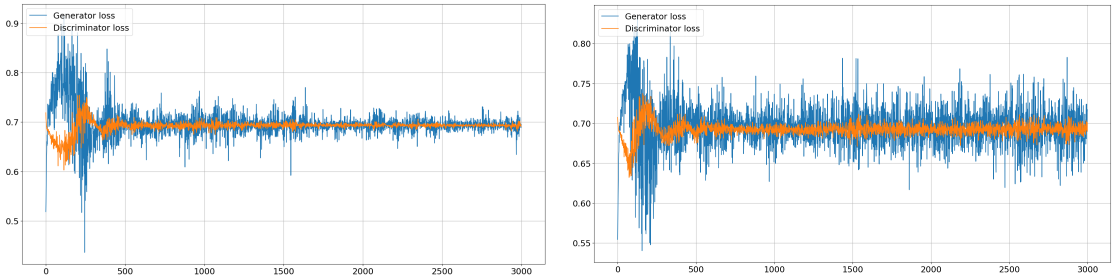


Figure 5.52: Generator's and Discriminator's loss during the training process over 3000 iterations for the first and second principal components.

The histograms of the train and reconstructed generated data are shown in Figure 5.53. As in the previous example, we observe a similarity between the training data and generated data, although they are not exactly the same due to the focus on generating data for the next day only. Overall, the generated data exhibit a distribution close to the training data, though slight deviations in shape and frequency.

When calculating the optimal π^* with the initially choosen parameters from Table 5.47, we got the following value $\pi^* = 0.2857$.

Similar to the previous example, from both the simulated train data and the reconstructed generated data (using GANs with the PCA method), we will estimate the parameters $b_1, b_2, \sigma_1, \sigma_2, \rho$, and $\hat{b_1}, \hat{b_2}, \hat{\sigma_1}, \hat{\sigma_2}, \hat{\rho}$, respectively. These estimates are summarized in Table 5.57. The generated training data provides better estimates for the drift terms, but the GAN-generated data yields slightly better estimates for the correlation. Both datasets offer similar volatility estimates that are relatively close to the target values. However, the drift terms show the largest deviation from the target values, even in the simulated training data.

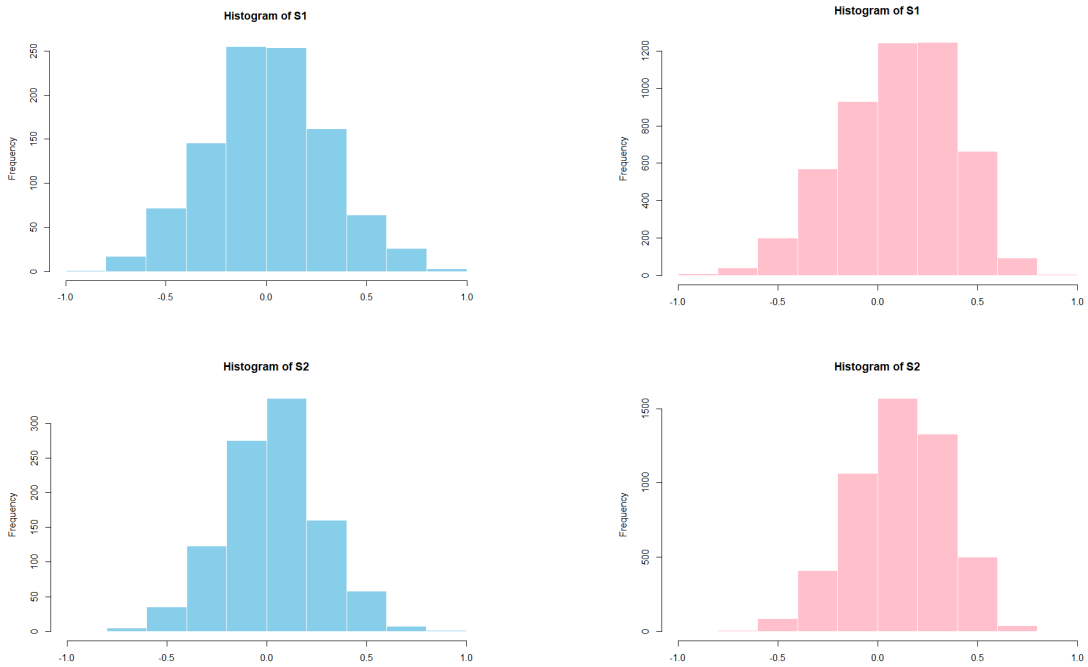


Figure 5.53: Histograms of S1 and S2 log returns: *blue* represents the training data, and *pink* represents the generated data for the next day.

	b_1	σ_1	b_2	σ_2	ρ
Generated train data	0.054	0.29	0.058	0.24	0.68
Generated GANs data	0.035	0.29	0.043	0.23	0.61

Table 5.57: Estimated parameters from simulated training data and GAN-generated data.

From these parameters, we calculate the optimal portfolio allocation π^*. For the generated training data, we obtain $\pi^* = 0.09$, while for the GAN-generated data, the optimal allocation is $\pi^* = 0.105$. Notably, the π^* value from the GAN-generated data is closer to the theoretically optimal $\pi^* = 0.2857$.

$$\text{GANs generated data} \quad \pi^* = 0.105, \qquad \text{train generated data} \quad \pi^* = 0.09.$$

Generating and Analyzing Synthetic Log Returns: Third Case

In this example, we generate the training data (log-returns) using different parameters compared to the previous cases, as shown in Table 5.58. Notably, we set the correlation parameter $\rho = 0$, which allows us to explore whether the absence of correlation between the two assets impacts the performance of the GAN model. By removing the correlation, we aim to investigate if previous results—where $\rho > 0$—benefited from the correlation structure and whether this leads to better synthetic data generation.

b_1	σ_1	b_2	σ_2	ρ
0.05	0.2	0.04	0.15	0

Table 5.58: Parameters used for simulating security prices in the third example.

Figure 5.54 presents the generated log-returns for the two securities, S_1 and S_2. As before, we will apply PCA to the first 1000 data points to extract the principal components.

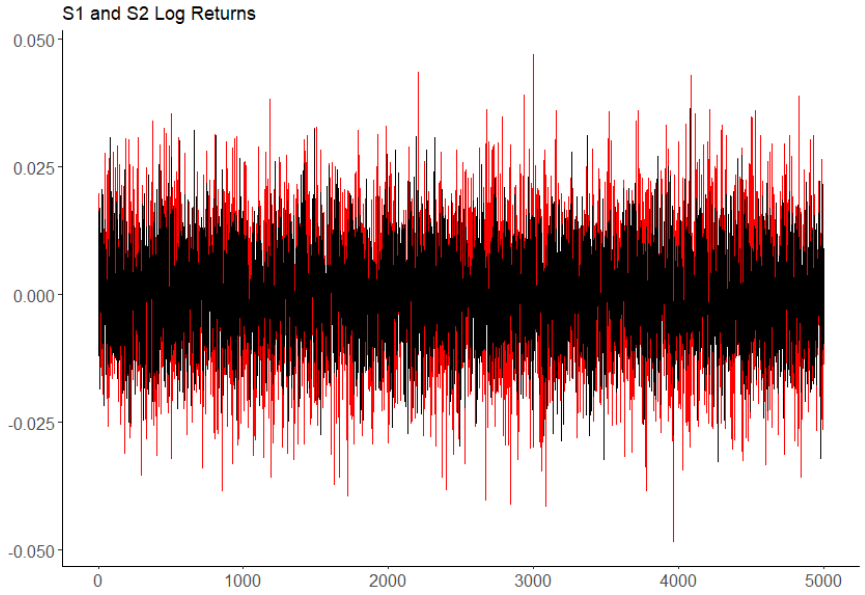


Figure 5.54: Log-returns for securities S1 (black) and S2 (red) in the third example.

Next, we compute the covariance matrix for the log-returns, shown in Table 5.59.

	S1	S2
S1	0.000151	-0.0000083
S2	-0.0000083	0.000094

Table 5.59: Covariance matrix of S1 and S2 log-returns in the third example.

We then perform PCA on the log-return data and compute the eigenvectors and eigenvalues, presented in Table 5.60. These eigenvectors correspond to the directions of maximum variance in the data, while the eigenvalues indicate the amount of variance captured by each principal component.

Eigen vectors	0.99	0.139
	-0.139	0.99
Eigen Values	0.00015	0.000093

Table 5.60: Eigenvectors and eigenvalues of the covariance matrix for S1 and S2 log-returns.

The explained variance, as shown in Table 5.61, highlights that the first principal component captures 62.19% of the total variance, making it the most significant factor driving the variation in the log-return data.

Explained variance	62.194	37.805
Cumulative explained variance	62.194	100.

Table 5.61: Explained variance and cumulative explained variance for the principal components derived from the log-returns in the third example.

We generate $10,000$ data points for the next day from each principal component using GANs, running 4000 iterations to ensure stable training. The training process for the Generator and Discriminator is depicted in Figure 5.55.

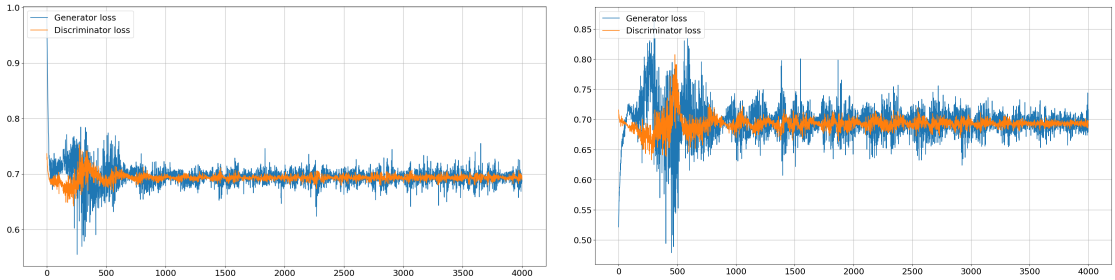


Figure 5.55: Generator's and Discriminator's loss during the training process for 4000 iterations in the third example.

Finally, the histograms in Figure 5.56 show the comparison between the training data and the GAN-generated data for S_1 and S_2. While the generated data closely follows the overall shape of the training data, minor differences are observed.

From the simulated training data and the reconstructed generated data from GANs using the PCA method, we estimate the parameters $b_1, b_2, \sigma_1, \sigma_2, \rho$, and $\hat{b_1}, \hat{b_2}, \hat{\sigma_1}, \hat{\sigma_2}, \hat{\rho}$ as shown in Table 5.62. Both the simulated training data and the GAN-generated data provide relatively close estimates for the volatilities σ_1 and σ_2 compared to the target values (Table 5.58), with the estimates from the GAN-generated data being slightly more accurate. However, the drift terms b_1 and b_2 deviate further from the target values in both cases. The correlation ρ estimates show some deviation from the target value of zero, particularly in the training data. Overall, while the volatilities are well captured, the drift terms show the largest deviations from the target values. Despite this, the optimal strategy derived from the GAN-generated data consistently remains closer to the theoretically

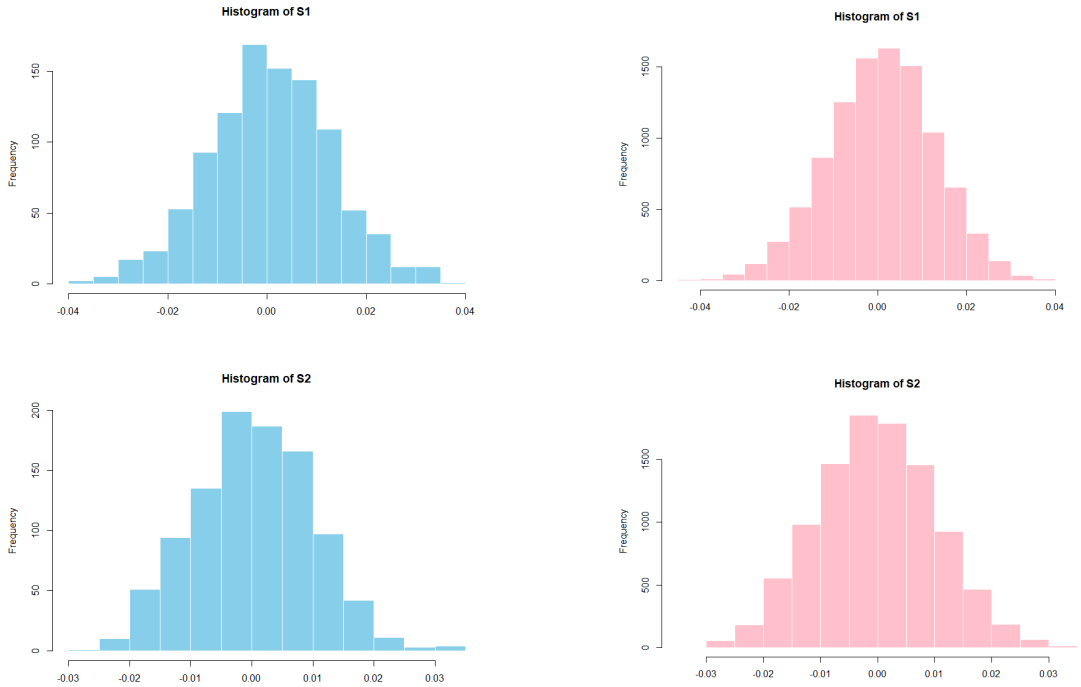


Figure 5.56: Histograms of S1 and S2 (log-returns): *blue* represents the training data, and *pink* represents the GAN-generated data for the next day.

optimal value, with the parameters from Table 5.62 yielding an optimal allocation of $\pi^* = 0.52$.

	b_1	σ_1	b_2	σ_2	ρ
Generated train data	0.0197	0.195	0.0121	0.154	-0.069
Generated GANs data	0.019	0.189	0.013	0.164	-0.056

Table 5.62: Estimated parameters from simulated training data and GAN-generated data.

$$\text{GANs generated data} \quad \pi^* = 0.52, \qquad \text{train generated data} \quad \pi^* = 0.50.$$

Confidence Interval Calculation for Estimated Parameters

So far, our comparisons between the training and generated data have focused on point estimates—single numerical values. However, it is well-known that estimating drift coefficients μ_i is particularly challenging in diffusion models. This raises the important question of how reliable these point estimates truly are. To address this uncertainty, we will now shift our focus to calculating confidence intervals, which provide a range within which the true parameter values are likely to fall,

We calculate the 95% and 80% confidence intervals for the parameters b_1, b_2, σ_1,σ_2 based on both the training and generated data. These intervals provide insights into the reliability of our parameter estimates.

We have a dataset with 1000 data points for the train data and 10000 for the generated ones. The standard deviations σ_1 and σ_2 and the parameters b_1 and b_2 are calculated using the following formulas:

$$\sigma_1 = \text{sd}(S_1) \times \sqrt{252}$$
$$\sigma_2 = \text{sd}(S_2) \times \sqrt{252}$$

$$b_1 = \text{annualizedmean}(S_1) + 0.5 \times \sigma_1^2$$
$$b_2 = \text{annualizedmean}(S_2) + 0.5 \times \sigma_2^2.$$

Next, we compute the standard error of the mean for S_1 and S_2, considering the corresponding time increment $\Delta t = 1/252$:

$$\text{SE}_{\text{mean_S1}} = \frac{\text{sd}(S_1)}{\sqrt{n * \Delta t}}$$
$$\text{SE}_{\text{mean_S2}} = \frac{\text{sd}(S_2)}{\sqrt{n * \Delta t}}$$

where n is the number of data points.

The 95% confidence intervals for the means of S_1 and S_2 are calculated as follows:

$$\text{CI}_{\text{mean_S1}} = \text{mean}(S_1) \pm 1.96 \times \text{SE}_{\text{mean_S1}}$$
$$\text{CI}_{\text{mean_S2}} = \text{mean}(S_2) \pm 1.96 \times \text{SE}_{\text{mean_S2}}$$

for the 80% CI only the value 1.96 changes to 1.282.

Assuming the variance term of σ^2 is negligible, the standard errors for b_1 and b_2 are approximated by the standard errors of the means:

$$\text{SE}_{b_1} = \text{SE}_{\text{mean_S1}} * \Delta t$$
$$\text{SE}_{b_2} = \text{SE}_{\text{mean_S2}} * \Delta t.$$

Finally, the 95% confidence intervals for b_1 and b_2 are given by:

$$\text{CI}_{b_1} = b_1 \pm 1.96 \times \text{SE}_{b_1}$$
$$\text{CI}_{b_2} = b_2 \pm 1.96 \times \text{SE}_{b_2}.$$

When we compute the confidence intervals (CIs) for the parameters b_1 and b_2 using both train and generated data, we obtain the following results.

For the **95%** CIs we have:

- **Train Data:**

$$\begin{aligned} \mathrm{CI}_{b_1} &= [-0.172,\, 0.212], \\ \mathrm{CI}_{b_2} &= [-0.139,\, 0.163]. \end{aligned}$$

- **Generated Data:**

$$\begin{aligned} \mathrm{CI}_{b_1} &= [-0.0402,\, 0.0779], \\ \mathrm{CI}_{b_2} &= [-0.0378,\, 0.0643]. \end{aligned}$$

For the **80%** CIs we have:

- **Train Data:**

$$\begin{aligned} \mathrm{CI}_{b_1} &= [-0.106,\, 0.145], \\ \mathrm{CI}_{b_2} &= [-0.087,\, 0.111]. \end{aligned}$$

- **Generated Data:**

$$\begin{aligned} \mathrm{CI}_{b_1} &= [-0.0197,\, 0.0575], \\ \mathrm{CI}_{b_2} &= [-0.02,\, 0.0466]. \end{aligned}$$

Given the data standard deviations σ_1 and σ_2, and assuming a size n of data points, the confidence intervals for the standard deviations are computed using the chi-squared distribution. For a given confidence level, the confidence intervals for the standard deviations σ_1 and σ_2 are calculated as follows:

First, we define the confidence level:

$$\alpha = 1 - \text{confidence_level}.$$

Next, we calculate Chi-Squared Critical values:

$$\begin{aligned} \chi^2_{\text{lower}} &= \text{qchisq}\left(\frac{\alpha}{2},\, n-1\right), \\ \chi^2_{\text{upper}} &= \text{qchisq}\left(1-\frac{\alpha}{2},\, n-1\right). \end{aligned}$$

Finally, we compute the Confidence Intervals:

$$\begin{aligned} \mathrm{CI}_{\sigma_1} &= \left(\sqrt{\frac{(n-1)\sigma_1^2}{\chi^2_{\text{upper}}}},\, \sqrt{\frac{(n-1)\sigma_1^2}{\chi^2_{\text{lower}}}}\right), \\ \mathrm{CI}_{\sigma_2} &= \left(\sqrt{\frac{(n-1)\sigma_2^2}{\chi^2_{\text{upper}}}},\, \sqrt{\frac{(n-1)\sigma_2^2}{\chi^2_{\text{lower}}}}\right). \end{aligned}$$

Here, $n-1$ represents the degrees of freedom.

When we compute the confidence intervals (CIs) for the parameters σ_1 and σ_2 using both train and generated data, we obtain the following results:

For the **95%** CI we have:

- **Train Data:**

$$\text{CI}_{\sigma_1} = [0.187,\ 0.204],$$
$$\text{CI}_{\sigma_2} = [0.147,\ 0.161].$$

- **Generated Data:**

$$\text{CI}_{\sigma_1} = [0.187,\ 0.192],$$
$$\text{CI}_{\sigma_2} = [0.162,\ 0.166].$$

For the **80%** CI:

- **Train Data:**

$$\text{CI}_{\sigma_1} = [0.189,\ 0.201],$$
$$\text{CI}_{\sigma_2} = [0.149,\ 0.158].$$

- **Generated Data:**

$$\text{CI}_{\sigma_1} = [0.188,\ 0.191],$$
$$\text{CI}_{\sigma_2} = [0.162,\ 0.165].$$

Comparison of Confidence Intervals for Train and Generated Data

Based on the analysis of the confidence intervals (CIs) for the generated and train data, we can make the following observations regarding the reliability of the estimated parameters (see Table 5.58 for the target values):

- For the generated data, the CIs for the parameters b_1 and b_2 are narrower and closer to their respective target values when compared to the train data. This indicates a higher reliability of these parameter estimates in the generated data. Similarly, the CI for σ_1 is also quite reliable, as it is narrow and close to the target value.
- However, in the case of σ_2, the CIs for the generated data do not contain the target value, suggesting a less reliable estimate for this parameter.
- On the other hand, the train data shows wider CIs for all parameters. While the CIs for σ_1 and σ_2 remain relatively narrow and close to the target values, indicating reliable estimates, the wider CIs for b_1 and b_2 reflect a lower level of reliability in comparison to the generated data.

In conclusion, the generated data using GANs with PCA provides more reliable estimates for most parameters, particularly for b_1, b_2, and σ_1, as their CIs are narrower and centered closer to the target values. However, the reliability of the estimate for σ_2 in the generated data is lower, as its CI does not cover the target value.

Merging Train and Generated Data

What happens if we merge the train and generated data into one sample?

In that case, for the **95%** CI we get the following results:

- $$\begin{aligned} \text{CI}_{b_1} &= [-0.0376, 0.0754], \\ \text{CI}_{b_2} &= [-0.0353, 0.0615]. \end{aligned}$$

- $$\begin{aligned} \text{CI}_{\sigma_1} &= [0.187, 0.193], \\ \text{CI}_{\sigma_2} &= [0.161, 0.165]. \end{aligned}$$

Merging the train and generated data results in narrower 95% confidence intervals (CIs), improving the precision of the parameter estimates. The CIs for b_1 and b_2 have become tighter, indicating reduced variability and greater reliability in the estimates. For σ_1 and σ_2, the intervals remain narrow and close to the target values. However, σ_2 still slightly overestimates the target value of 0.15. Overall, the merged sample increases the reliability of the parameter estimates.

Generating and Analyzing Synthetic Log Returns: Fourth Example

In this case, we simulate the training data (log-returns) using parameters with a very high correlation coefficient, $\rho = 0.95$, as shown in Table 5.63. The high correlation allows us to analyze the effect of strongly linked asset price movements on the simulated data and the GAN model's performance.

b_1	σ_1	b_2	σ_2	ρ
0.05	0.2	0.04	0.15	0.95

Table 5.63: Parameters for generating security prices (log-returns) with a strong correlation ($\rho = 0.95$).

As in the previous examples, we use the generated training data (the first 1000 data points) for PCA. Subsequently, we generate an additional 10,000 data points from each principal component using GANs for the next day. This will allow us to observe how the strong correlation impacts the generation process and the resulting data.

	S1	S2
S1	0.000151	0.000105
S2	0.000105	0.0000822

Table 5.64: Covariance matrix of S1 and S2 (log-returns) with a strong correlation ($\rho = 0.95$).

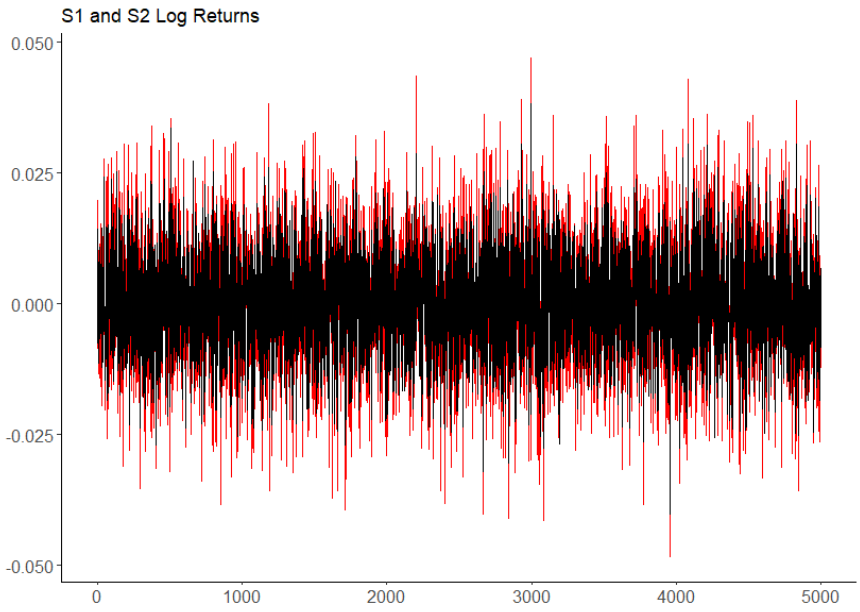


Figure 5.57: S1 and S2 log returns generated for the fourth example, reflecting the strong correlation.

The eigenvectors and eigenvalues of the covariance matrix are shown in Table 5.65. Notably, the first principal component explains 97.407% of the variance, which is expected given the strong correlation between the two assets.

Eigen vectors	0.8098	-0.587
	0.587	0.8098
Eigen Values	0.0002274	0.0000061

Table 5.65: Eigenvectors and eigenvalues of the covariance matrix for S1 and S2 (log-returns) with a high correlation.

The explained variance for each principal component is summarized below. The first principal component captures almost all of the variance (97.407%), while the second principal component explains only 2.592%.

Explained variance	97.407	2.592
Cumulative explained variance	97.407	100.

Table 5.66: Explained variance and cumulative explained variance for the principal components.

For the data generation process, we used 6000 iterations for each principal component. The training progress is shown in Figure 5.58. The training is stable for both the Generator and the Discriminator, particularly for the first principal component (PC), which captures the majority of the variance in the data. This demonstrates

the model's ability to effectively handle highly correlated data, especially given the strong correlation ($\rho = 0.95$) in this example.

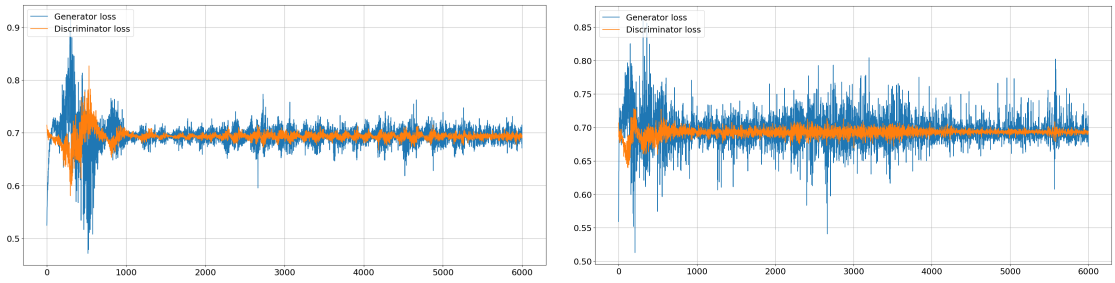


Figure 5.58: Generator's and Discriminator's loss during the training process for the fourth example with a high correlation.

The histograms of the train and reconstructed generated data for both S_1 and S_2 log returns are presented in Figure 5.59. Compared to the previous case with zero correlation, we observe that the generated data distribution more closely resembles the training data. The high correlation of $\rho = 0.95$ seems to improve the GAN's ability to capture the underlying structure of the log returns, as reflected in the histogram shapes. Although some minor differences still exist.

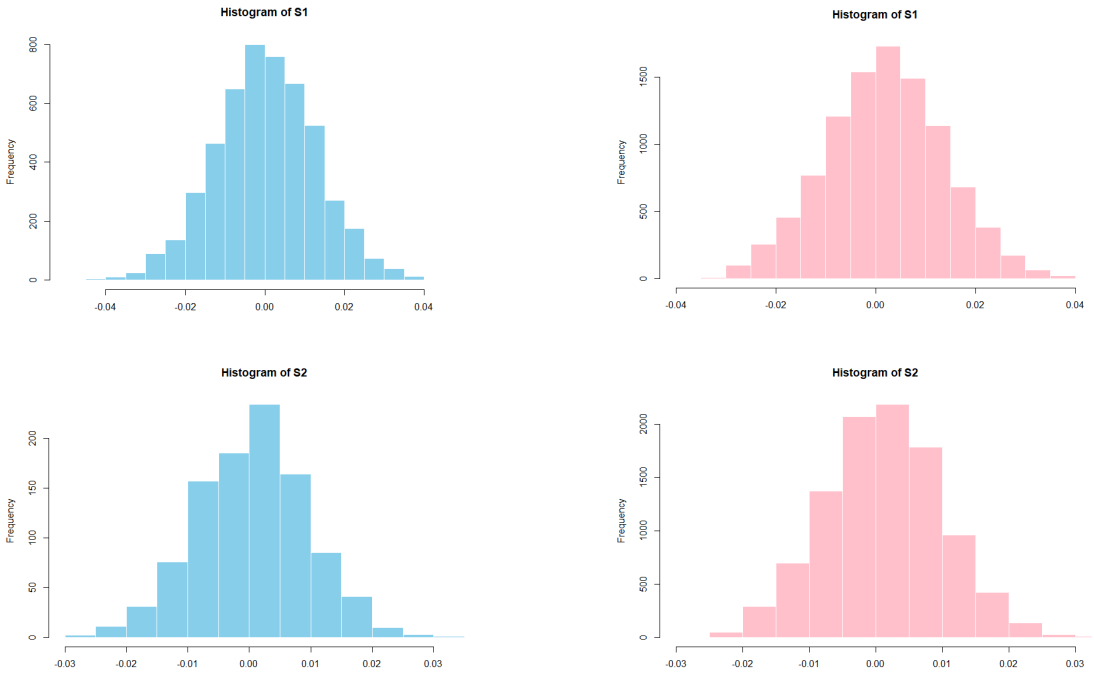


Figure 5.59: Comparison of histograms for S1 and S2 log-returns: the *blue* bars represent the training data, while the *pink* bars represent the GAN-generated data for the next day.

When analyzing the estimated parameters for the train and generated data (see Table 5.67), we observe that the values for b_1, σ_1, b_2, and σ_2 from the training data are slightly better aligned with the original parameters compared to the generated

data, except for the correlation (ρ). The correlation value estimated from the generated data ($\rho = 0.92$) is slightly less accurate compared with the one from the training data ($\rho = 0.943$) when compared with the true value ($\rho = 0.95$). However, despite this minor discrepancy, the GAN model still demonstrates its capability to effectively capture the overall correlation structure.

	b_1	σ_1	b_2	σ_2	ρ
Generated train data	0.0197	0.195	0.011	0.143	0.943
Generated GANs data	0.019	0.187	0.011	0.138	0.92

Table 5.67: Estimated parameters for b_1, b_2, σ_1, σ_2, and ρ from the training data (generated log-returns) and the GAN-generated data.

The calculated optimal portfolio allocation, π^*, for the generated and train data are 0.61 and 0.515, respectively. Both values are relatively close to the theoretically optimal $\pi^* = 0.73$. However, the optimal π^* from the generated data is closer to the theoretically expected value.

$$\text{GANs generated data} \quad \pi^* = 0.61, \qquad \text{train generated data} \quad \pi^* = 0.515$$

Confidence intervals for the estimated parameters

First for the **95%** CI:

- **Train Data:**

$$\begin{aligned} \text{CI}_{b_1} &= [-0.172,\, 0.211], \\ \text{CI}_{b_2} &= [-0.130,\, 0.152]. \end{aligned}$$

- **Generated Data:**

$$\begin{aligned} \text{CI}_{b_1} &= [-0.0389,\, 0.0779], \\ \text{CI}_{b_2} &= [-0.0324,\, 0.054]. \end{aligned}$$

For the **95%** CI:

- **Train Data:**

$$\begin{aligned} \text{CI}_{\sigma_1} &= [0.187,\, 0.204], \\ \text{CI}_{\sigma_2} &= [0.137,\, 0.150]. \end{aligned}$$

- **Generated Data:**

$$\begin{aligned} \text{CI}_{\sigma_1} &= [0.186,\, 0.190], \\ \text{CI}_{\sigma_2} &= [0.137,\, 0.141]. \end{aligned}$$

Based on the confidence intervals (CIs), the reliability of the parameter estimates varies between the train and generated data. For both b_1 and b_2, the CIs from the generated data are narrower and closer to the expected values, suggesting greater accuracy in the estimates. Similarly, for σ_1, the generated data shows tighter CIs compared to the train data, indicating higher reliability. However, for σ_2, both the train and generated data show similar ranges, with the generated data providing slightly more consistent intervals, though both are relatively close to the original values.

When we generate for a longer time window, specifically for 300 days, we achieve improved parameter estimates, as shown in Table 5.68. The histograms of the generated data are shown in Figure 5.60, and we are able to capture the distribution of the train data quite well.

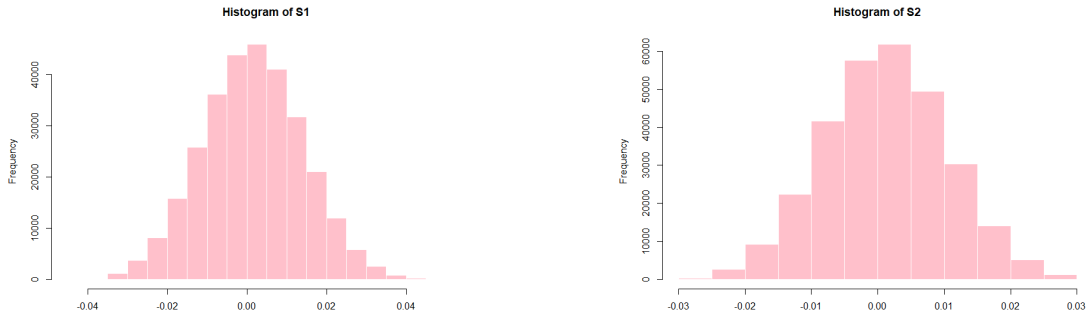


Figure 5.60: Histogram of S1 and S2 log returns, generated for a longer time window (300 days).

The estimated parameters from the generated data, summarized in Table 5.68, demonstrate that these estimates are closer to the target values (see Table 5.63) Notably, the correlation parameter ρ is estimated very accurately. This results in an optimal portfolio strategy with $\pi^* = 0.55$, which is closer to the theoretically optimal value compared to the one we get from synthetic training data.

	b_1	σ_1	b_2	σ_2	ρ
Generated GANs data	0.021	0.199	0.012	0.147	0.944

Table 5.68: Estimated parameters from generated data over a longer time window (300 days).

For the generated data with a longer time window we get the following **95%** CI:

- $$\begin{aligned} \mathrm{CI}_{b_1} &= [-0.041,\, 0.084], \\ \mathrm{CI}_{b_2} &= [-0.034,\, 0.058]. \end{aligned}$$

- $$\begin{aligned} \text{CI}_{\sigma_1} &= [0.197,\ 0.203], \\ \text{CI}_{\sigma_2} &= [0.145,\ 0.150]. \end{aligned}$$

For the generated data with a longer time window merged with the train data we get the following **95%** CI:

- $$\begin{aligned} \text{CI}_{b_1} &= [-0.038, 0.081], \\ \text{CI}_{b_2} &= [-0.032, 0.055]. \end{aligned}$$

- $$\begin{aligned} \text{CI}_{\sigma_1} &= [0.197, 0.202], \\ \text{CI}_{\sigma_2} &= [0.145, 0.150]. \end{aligned}$$

The generated data for a longer time window provides the closest match to target values, particularly for σ_1 and σ_2, demonstrating high precision and reliability. In contrast, estimates from the train data are further from the target values, indicating less reliability and potential bias. The merged data gives us similar estimates to the generated data but with slight improvements in precision for σ_1 and σ_2.

Summary and Implications

The performance of GAN-generated data can be described as okay, but not exceptional, particularly since we are primarily generating data for the next day only. While GANs capture the volatility structure quite well, in some cases, the synthetic training data produce parameter estimates (b_1, b_2, σ_1, σ_2) that are closer to the theoretical values. However, when it comes to capturing correlations, GANs consistently outperform, reliably modeling the relationships between assets.

One key observation is that GANs always result in a better optimal investment strategy compared to the training data. However, accurately estimating the drift remains a challenging task. Even with the synthetic simulated log returns, the drift estimates are not close to the target values, so we do not expect significant improvements using GANs in this area either. Nonetheless, they do manage to capture the broader relationship between drift and volatility reasonably well.

Furthermore, the confidence intervals (CIs) for the GAN-generated data are generally narrower and more precise, particularly for volatility estimates, highlighting the reliability of the results.

Chapter 6

Conclusion

We conclude the thesis by summarizing our key findings and their relevance to the research aims and questions. We discuss the value, contribution, and limitations of our studies, and suggest future research opportunities.

In the previous chapters, we derived in detail the theoretical foundations of Generative Adversarial Networks (GANs), exploring their applications in time series simulations, Value at Risk (VaR) forecasting, and portfolio optimization. Before diving into the theory of GANs, in Chapter 2 we have described the essential concepts of artificial neural networks (ANN), machine learning, and deep learning, laying the foundation for understanding the architecture and components of GANs, which were elaborated on in Chapter 3.

One of the key contributions of this thesis is the detailed application of *ForGAN*, a type of Conditional GAN (cGAN), which demonstrated how simulated data for future time periods can support portfolio optimization decisions. In Chapter 4, we investigated the factors that influence the accuracy of the simulated data, using two datasets: the more regular, seasonal 15-minute intraday electricity consumption in Germany, and the volatile S&P 500 index, with complex, non-stationary properties.

Our results showed that data transformations are not always beneficial and that removing outliers may result in the loss of valuable information-particularly in financial datasets where extreme market movements are important for accurate forecasting. ForGAN's performance was influenced by hyperparameter choices. Larger condition sizes did not consistently improve accuracy, but increasing the number of training iterations did. Additionally, high-frequency data captured more detailed patterns, resulting in improved forecasting performance. In contrast, the S&P 500 dataset, marked by its volatility, presented greater challenges for GAN-based modeling, demonstrating the difficulty of capturing these complex patterns. Given the critical role of hyperparameter tuning in determining model performance, implementing a hyperparameter grid search proved to be highly beneficial. This approach allowed us to systematically explore a range of parameters and identify optimal configurations for improved forecasting accuracy.

This thesis highlights the importance of dataset characteristics when using GANs for time series forecasting. ForGAN performed better on regular datasets, such as electricity consumption, where seasonality and patterns are consistent. In contrast, financial datasets like the S&P 500, with their inherent volatility, heavy tails, and sudden market shifts, proved more challenging for ForGAN to model effectively. The unpredictable behavior and non-stationarity of financial markets made it more difficult to achieve high accuracy compared to energy data.

In Chapter 5, our main research question is *"Can we use GANs for Portfolio Optimization?"*. Two portfolio optimization problems were described : *Markowitz mean-variance approach*, and *Maximization of the Expected Growth Rate*. The crucial task in both approaches is the estimation of the parameters, such as means, and covariances. By generating additional data using GANs, we expanded the data universe to obtain more reliable estimates. First, we used the log increments of the stock prices for Apple, Amazon, and Netflix. To generate additional data and keep the information regarding the correlation of the assets we used the Principal Component Analysis (PCA), which is also described in this Chapter. We were able to generate additional log returns that closely followed the distribution of the historical ones and got improved risk-return proportion.

Moreover, to compare our method of using the *ForGAN* with PCA, we described another type of GAN, namely *TimeGAN*. TimeGAN uses an embedding network to represent the data in a lower-dimensional space, which helps identify and maintain the temporal relationships and correlations between different time points. We highlighted the strengths and challenges of both methods.

The approach of ForGAN with PCA handles and analyzes individual principal components before reconstructing them back-to-log return space and this provides clear control over the process without reducing dimensionality. As a result, it provides a more reliable framework for estimating parameters crucial for portfolio optimization, particularly when accurate asset correlations are important. TimeGAN, on the other hand, compresses the data into a lower-dimensional space using an embedding network before reconstructing it. While this method efficiently captures temporal patterns, the compression can sometimes lead to inaccuracies, particularly in capturing correlations, which are essential for portfolio optimization. Therefore, we showed that ForGAN with the PCA excels in maintaining the correlation structure needed for portfolio optimization, while TimeGAN, though better at capturing temporal dependencies, may introduce challenges in preserving the full interdependencies across time series

Despite the good results, the similarity between the distributions and estimated parameters for train historical log-returns and generated log-returns, led us to the following important research question *"Which data can be trusted more - the historical log-returns of the stock prices, or the generated data?"*. To further explore this question we looked at the problem of maximizing the expected growth in continuous-time diffusion model where we know all market parameters and also the explicit form of the optimal portfolio process. We could then generate data with the correct distribution from this setting by a usual Monte Carlo method.

Having this simulated synthetic data as our training data, we were able to train a GAN. This controlled approach allowed us to assess the accuracy of our generated data in a portfolio optimization context, Where we got better *optimal portfolio allocation* from the generated data using GANs compared to the train synthetic data.

We mainly focused on point estimates when comparing training data with the data generated by GANs. However, as it is well-known that estimating drift coefficients in a diffusion setting is notoriously difficult, this raised the question: how much can we trust these point estimates? To address this, we calculated confidence intervals (CIs) for the estimated parameters for both the historical training data and the GAN-generated data. Our analysis showed that the CIs for the GAN-generated data were generally narrower and more precise, indicating that the GAN-generated data might offer more reliable parameter estimates in certain settings.

We also compared ForGAN with LSTMs for next-day time series forecasting. While LSTMs provided reasonable forecasts, they tended to predict in a mean-wise manner and struggled with capturing the full distribution of the data. Additionally, LSTMs required significantly more computational time to train compared to ForGAN. In contrast, ForGAN was more effective at capturing the underlying data distribution, making it a more suitable choice for generating diverse and realistic time series simulations.

This thesis shows that ForGAN, TimeGAN, LSTM, and other neural networks are capable of learning data quite well, but they often have to fight very hard against specialized traditional methods. It is not always the case that neural networks outperform specialized traditional methods which are well-established. For instance, the GARCH model continues to perform well and remains relatively easy to train.

Further research could aim to enhance GAN models for handling highly volatile and non-stationary datasets like financial time series. Combining GANs with advanced volatility models such as GARCH could improve their ability to capture heavy tails and sudden market shifts. Additionally, automating hyperparameter tuning with methods like Bayesian optimization could lead to more efficient and accurate performance. Additional research is needed in the direction of the use of GANs for portfolio optimization, such as expanding the use of synthetic data generated by GANs in other areas of portfolio management, such as stress testing or scenario analysis, could yield new insights into the robustness of investment strategies.

List of Figures

2.1 Working principle of an artificial neuron [36]. 10
2.2 Example of simple artificial neural network. 10
2.3 Artificial Neuron. 11
2.4 From left to right: the rectified linear unit (ReLU), the logistic, and the hyperbolic tangent activation functions [90]. 12
2.5 Gradient descent. An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum. 13
2.6 Approximate minimization. 14
2.7 A deep neural network with three hidden layers. 20
2.8 Recurrent (RNN) and Feed-forward (FNN) topology of an artificial neural network. 21
2.9 Visualisation of differences between Feedforward NNs and Recurrent NNs [80]. 21
2.10 A recurrent network with no outputs. This recurrent network just processes information from the input x by incorporating it into the state that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of a single timestep. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance [48]. . . . 22
2.11 The computational graph for computing the training loss of a recurrent network that maps an input sequence x to a corresponding output sequence o [48]. 23
2.12 LSTM Architecture [73]. 26
2.13 The LSTM forget gate operation. 27
2.14 The LSTM input gate operation. 28
2.15 The LSTM update cell operation. 28
2.16 The LSTM output gate operation. 29
2.17 Gated Recurrent Unit [79]. 30

3.1 Swiss chocolate GAN analogy: Discriminator (chocolate expert) and Generator (chocolate company) cooperate to create Swiss-style chocolate [93]. 32
3.2 Basic GANs architecture. 33
3.3 An illustration of the gradient method of Adversarial Net. 39
3.4 Minimax objective versus the generated data distribution. 40

3.5 An illustration of the mode collapse problem on a two-dimensional toy dataset [49] . . . 41
3.6 DCGANs for 1, 10 and 25 epochs. G is fixed while D is trained from scratch and using original cost function to compute the gradients [55]. 42
3.7 DCGANs for 1, 10 and 25 epochs. G is fixed, D is trained from scratch and using -log D cost function to compute the gradients [55]. 42
3.8 Proposed Taxonomy. . . . 43

4.1 ForGan architecture where the condition c is handed to the generator G and discriminator D. . . . 45
4.2 (a): The architecture of the generator. (b): The architecture of the discriminator. . . . 46
4.3 One-month electricity consumption for 15-minute periods. . . . 47
4.4 The validation set and generated set on the first step. . . . 48
4.5 The validation set and generated set on the last step. . . . 48
4.6 Probability distribution of one step ahead forecast learned from ForGAN, one generated path. . . . 49
4.7 Discriminator's loss during the training process. . . . 50
4.8 Generator's loss during the training process. . . . 50
4.9 The validation set and generated set on the first step. . . . 51
4.10 The validation set and generated set on the last step. . . . 51
4.11 Probability distribution of one step ahead forecast learned from ForGAN with 1000 training iterations. . . . 52
4.12 Mean of five forecasted paths versus actual one using ForGAN with 1000 training iterations. . . . 52
4.13 Discriminator's loss during the training process. . . . 53
4.14 Generator's loss during the training process. . . . 53
4.15 Disciminator's and Generator's loss during the training process with 1000 training iterations. . . . 54
4.16 Disciminator's and Generator's loss during the training process with 5000 training iterations. . . . 54
4.17 Probability distribution of one step ahead forecast learned from ForGAN 5000 training iterations. . . . 55
4.18 The validation set and generated set on the first step. . . . 55
4.19 The validation set and generated set on the last step. . . . 56
4.20 Probability distribution of one step ahead forecast learned from ForGAN with increased condition size to **672** during training. . . . 57
4.21 Mean of five forecasted paths versus actual one using ForGAN with increased condition size to **672** during training. . . . 57
4.22 Three years of electricity consumption for 15-minute periods. . . . 58
4.23 Box plots regarding weeks, months, and years for 15-minute electricity consumption. . . . 59
4.24 Box plot of the complete dataset, confirming no significant outliers. 59
4.25 Heatmap showing correlations between days of the week and months of the year for electricity consumption. . . . 60
4.26 The validation set and generated set on the first step. . . . 61

4.27 The validation set and generated set on the last step. 61
4.28 Probability distribution of one step ahead forecast learned from ForGAN. 62
4.29 Mean of five forecasted paths versus actual one using ForGAN. . . . 62
4.30 Discriminator's loss during the training process. 63
4.31 Generator's loss during the training process. 63
4.32 Auto-correlation plot of the real and generated data of five random paths. 63
4.33 Auto-correlation plot of the real data. 64
4.34 Auto-correlation plot of the generated data. 64
4.35 Cross-correlation plot of the real and generated data for ten random paths. 64
4.36 Three years of daily electricity consumption. 65
4.37 Box plots illustrating the distribution of daily electricity consumption across different weeks, months, and years. 65
4.38 The validation set and generated set on the first step. 66
4.39 The validation set and generated set on the last step. 67
4.40 Probability distribution of one-step-ahead forecast generated by ForGAN for daily aggregated electricity consumption data. 67
4.41 Mean of fifty forecasted paths compared to the actual path using ForGAN for daily aggregated electricity consumption data. 68
4.42 Generator's and Discriminator's loss during the training process. . . 68
4.43 Auto-correlation plot of the real and generated data of five random paths. 69
4.44 Auto-correlation plot of the real data. 69
4.45 Auto-correlation plot of the generated data. 69
4.46 Cross-correlation plot of the real and generated data for ten random paths. 70
4.47 Standard and Poor's 500 Index 71
4.48 Box plots showing weekly and monthly variations in the S&P 500 Index. 72
4.49 Log returns of the Standard and Poor's 500 Index from January 1871 to January 2018. 73
4.50 Comparison of the validation set and the first generated path for the log-returns of the S&P 500 index. 73
4.51 Comparison of the validation set and the last generated path for the log-returns of the S&P 500 index. 74
4.52 Generator's and Discriminator's loss during the training process. . . 74
4.53 Probability distribution of one step ahead forecast learned from ForGAN for log-returns of S & P 500 index. 75
4.54 Mean of fifty forecasted paths versus actual one using ForGAN for log-returns of S & P 500 index. 75
4.55 Cross-correlation plot of the real and generated data for ten random paths. 76
4.56 S & P 500 close prices for five years. 76
4.57 S & P 500 returns and log returns of close prices for five years. . . . 77

4.58 The validation set and generated set on the first step. 78
4.59 The validation set and generated set on the last step. 78
4.60 Discriminator's loss during the training process. 79
4.61 Generator's loss during the training process. 79
4.62 Generator's and Discriminator's loss during the training process. . . 79
4.63 Probability distribution of one step ahead forecast learned from ForGAN for log-returns of S & P 500 index. 80
4.64 Mean of five forecasted paths versus actual one using ForGAN for log-returns of S & P 500 index. 80
4.65 Improved Generator's and Discriminator's loss during the training process. 81
4.66 S & P 500 close prices over twelve years. The prices show upward trends and increasing variance, indicating non-stationarity. 83
4.67 S & P 500 log returns of close prices over twelve years. The log returns display a more stationary behavior. 84
4.68 The validation set and generated set on the first step. 85
4.69 The validation set and generated set on the last step. 86
4.70 Discriminator loss over 3000 training iterations for log-returns of the S&P 500 index. 86
4.71 Generator loss over 3000 training iterations for log-returns of the S&P 500 index. 86
4.72 Generator and discriminator losses during training for log-returns of the S&P 500 index. 87
4.73 Probability distribution of one-step-ahead forecasts learned from ForGAN for the log-returns of the S&P 500 index. 87
4.74 Comparison of the real data distribution with t, beta, log-normal, and normal distributions. 89
4.75 Comparing the distribution of the generated data with t, beta, log normal, and normal distribution. 89
4.76 Historical value at risk for 95 % and 99 % confidence intervals for the true data. 90
4.77 Historical value at risk for 95 % and 99 % confidence intervals for the generated data. 91
4.78 Monte-Carlo simulation of the prices for true data. 92
4.79 Monte-Carlo simulation of the prices for generated data. 93
4.80 Distribution of probabilities for exceptions given by the binomial distribution. 95
4.81 Log return distribution from 1000 simulations for trading day 3001. 96
4.82 KDE of 1000 simulated log returns for trading day 3001. 97
4.83 Historical VaR estimates at 95% and 99% confidence intervals based on generated data for trading day 3001. 97
4.84 Comparison of the distribution of the generated data with t, beta, log-normal, and normal distributions. 98
4.85 Log return distribution from 1000 simulations for trading day 3002. 99
4.86 KDE of 1000 simulated log returns for trading day 3002. 99

4.87 Historical VaR estimates at 95% and 99% confidence intervals based on generated data for trading day 3001. 100
4.88 Comparison of the distribution of the historical log returns with t, beta, log-normal, and normal distributions for the last 453 days. The t-distribution provides the best fit to the data. 103
4.89 Log return distribution from 1000 simulations for the trading day 2568. 104
4.90 KDE of 1000 simulated log returns for trading day 2568. 104
4.91 Historical value at risk for 95 % and 99 % confidence interval on generated data. 105
4.92 ACF and PACF of squared residuals from ARIMA(3,0,2) with non-zero mean, showing significant correlation. 106
4.93 ACF and other diagnostic plots for sGARCH(1,1) residuals, confirming model adequacy. 107
4.94 Forecasted 5% VaR from GARCH(1,1). Red points mark actual returns below the VaR threshold. 108
4.95 Forecasted 1% VaR from GARCH(1,1). Red points indicate actual returns below the VaR threshold. 109

5.1 Indifference curves [97]. 113
5.2 The Efficient Frontier. 120
5.3 TimeGAN block diagram and training scheme [83]. 122
5.4 Plot of 50 observations on two variables x_1,x_2. 123
5.5 Plot of the 50 observations from Figure 1.1 with respect to their PCs z_1, z_2. 124
5.6 Stock prices for Apple, Netflix, Amazon, Exon Mobil Corporation, and AT & T Corporation. 127
5.7 Log returns of stock prices. 128
5.8 Histograms and Q-Q plots of the train log returns. 129
5.9 Mean centered train log return data. 131
5.10 Disciminator's and Generatos's loss during the training process for the first PC. 132
5.11 Disciminator's and Generatos's loss during the training process for the second PC. 133
5.12 Disciminator's and Generatos's loss during the training process for the third PC. 133
5.13 Histograms and Q-Q plots of the generated log returns. 134
5.14 Efficient frontier for the portfolio obtained from train data. 137
5.15 Cumulative return of the portfolio over the test data period. The **red** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the historical data, while the **black** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the GAN-generated data. 138

5.16 Cumulative return of the portfolio over the complete historical data period. The **red** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the historical data, while the **black** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the GAN-generated data. 139
5.17 Cumulative return of the portfolio over the test historical data period. The **red** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the historical data for a specific level of risk 0.223, while the **black** line represents the cumulative returns calculated using the weights of the optimal portfolio derived from the GAN-generated data for a specific level of risk 0.223. 140
5.18 PCA plot comparing the original and TimeGAN-generated data. . . 142
5.19 t-SNE plot showing the distribution of original versus TimeGAN-generated data. 142
5.20 Comparison of original log returns, standardized log returns (using Standard Scaler), and mean-centered log returns for the training data. 144
5.21 Log return data after applying PCA, transformed into the principal component space. 146
5.22 Reconstructed log returns using only the first principal component, capturing the direction of maximum variance. 146
5.23 Autocorrelation function (ACF) for the complete dataset, showing how the correlation between stock returns diminishes over time. . . 147
5.24 Autocorrelation function (ACF) for the training dataset, reflecting a similar decay in correlations as the complete dataset. 148
5.25 PCA-transformed log return data for the first, second, and third principal components. 149
5.26 Reconstructed training log returns from the first, second, and third principal components, after performing the inverse PCA transformation. 150
5.27 Reconstructed Log Return Data from PCA for GAN-Generated Data. 150
5.28 Efficient Frontier generated using only the train historical data for portfolio optimization (first case). 151
5.29 Efficient Frontier generated using the merged data from train historical data and GAN-generated data (second case). 152
5.30 Efficient Frontier from generated data with GANs (third case). . . . 152
5.31 Portfolio Cumulative Returns, where red represents the train data (first case), green the train merged with generated data (second case) and black GANs generated data (third case). 153
5.32 Histograms comparing AMZN, AAPL, and NFLX log returns: training data in blue, GAN-generated data in pink. 154
5.33 Efficient frontier for train log returns merged with generated data, using a different set of hyperparameters without any statistical test as e condition. 156

5.34 Discriminator's and Generator's loss during the training process across different iteration counts. 156
5.35 Histogram of AMZN, AAPL, and NFLX, from generated data with more training iterations. 157
5.36 Efficient Frontier from generated data with more iterations, merged with the train data. 157
5.37 Histogram of AMZN, AAPL, and NFLX, from generated data with a 300-day time window. 157
5.38 Efficient Frontier from generated data with an extended time window, merged with the train data. 158
5.39 Generated synthetic log returns for three assets with imposed correlations using Cholesky decomposition. 161
5.40 Loss curves of the generator and discriminator during training for the first principal component (PC) over 5000 steps. 163
5.41 Loss curves of the generator and discriminator during training for the second principal component (PC) over 5000 steps. 163
5.42 Loss curves of the generator and discriminator during training for the third principal component (PC) over 5000 steps. 164
5.43 PCA plot comparing the original and TimeGAN-generated data. . . 165
5.44 t-SNE plot showing the distribution of original versus TimeGAN-generated data. 166
5.45 Simulated annual log returns for three assets, reflecting specified means, volatilities, and correlations. 169
5.46 Histograms comparing training log returns (blue) and generated log returns (pink) for assets X1, X2, and X3. 171
5.47 Comparison of Efficient Frontiers for the training and generated log returns, highlighting optimal portfolios with minimum variance and maximum Sharpe ratio (red points). 172
5.48 Log returns for securities S1 and S2, illustrating their distribution and characteristics. 174
5.49 Generator's and Discriminator's loss curves during the training process for the first (left) and second (right) principal components. . . 176
5.50 Histograms of log returns for securities S1 (top row) and S2 (bottom row): blue bars represent training data, while pink bars represent generated data. 176
5.51 Log returns for securities S1 (in black) and S2 (in red). 179
5.52 Generator's and Discriminator's loss during the training process over 3000 iterations for the first and second principal components. . 180
5.53 Histograms of S1 and S2 log returns: *blue* represents the training data, and *pink* represents the generated data for the next day. . . . 181
5.54 Log-returns for securities S1 (black) and S2 (red) in the third example. 182
5.55 Generator's and Discriminator's loss during the training process for 4000 iterations in the third example. 183
5.56 Histograms of S1 and S2 (log-returns): *blue* represents the training data, and *pink* represents the GAN-generated data for the next day. 184

5.57 S1 and S2 log returns generated for the fourth example, reflecting the strong correlation. 189
5.58 Generator's and Discriminator's loss during the training process for the fourth example with a high correlation. 190
5.59 Comparison of histograms for S1 and S2 log-returns: the *blue* bars represent the training data, while the *pink* bars represent the GAN-generated data for the next day. 190
5.60 Histogram of S1 and S2 log returns, generated for a longer time window (300 days). 192

List of Tables

2.1 A summary of the problems faced by different strategies to deep generative modeling for all of the main model operations. 19

4.1 Performance of the ForGAN model across different metrics for the test set, showing the mean and standard deviation (where applicable) based on 100 runs. 49
4.2 Performance metrics for ForGAN after 1000 training iterations. . . 51
4.3 Performance metrics for ForGAN after 5000 training iterations. . . 53
4.4 Performance metrics for ForGAN with increased condition size to **672** during training. 56
4.5 S & P 500 Data Set. 77
4.6 Descriptive statistics of the S & P 500 log returns. 85
4.7 Error results from the generated path using ForGANs. 88
4.8 Comparison of different distributions with the true dataset. 89
4.9 Comparison of different distributions with the generated dataset. . . 90
4.10 Monte-Carlo simulated prices for actual data. 92
4.11 Monte-Carlo simulated prices for generated data. 94
4.12 sGARCH(1,1) model's coefficient for generated data. 95
4.13 Results from comparing different distributions with the forecasted daily distribution. 98
4.14 Estimated coefficients for sGARCH(1,1) model based on generated data. 107

5.1 Estimated means of the train log returns. 129
5.2 Estimated covariance matrix of the train log returns. 130
5.3 Annualized covariance matrix of historical train data. 130
5.4 Correlation matrix of the training log returns. 130
5.5 Eigenvectors and eigenvalues of train log return data. 132
5.6 Variance explained and cumulative variance explained by each principal component. 132
5.7 Estimated means of the generated log returns (train data values in blue). 134
5.8 Annualized estimated covariance matrix of the generated log returns (train data values in blue). 135
5.9 Correlation matrix of the generated log returns (train data values in blue). 135

5.10 Statistics of the Minimum Variance Portfolio for generated data. . . 136
5.11 Statistics of the Tangency Portfolio for generated data. 136
5.12 Statistics of the Minimum Variance Portfolio for true historical data. 137
5.13 Statistics of the Tangency Portfolio for train historical data. 137
5.14 Correlation matrix of the generated log returns from TimeGAN (training data values in blue). 143
5.15 Means of the generated log returns from TimeGAN (training data values in blue). 143
5.16 Covariance matrix of the log returns for the training data. 145
5.17 Eigenvalues in descending order for principal component analysis. . 145
5.18 Explained variance for each principal component. 145
5.19 Covariance matrix of the training dataset, showing the relationships between AAPL, AMZN, and NFLX stock returns. 147
5.20 Correlation matrix of the training dataset, illustrating the strength of relationships between AAPL, AMZN, and NFLX stock returns. . 147
5.21 Eigenvalues and eigenvectors of the covariance matrix, which represent the principal components of the data. 149
5.22 Hyperparameters used for GAN training for each principal component (PC). 150
5.23 Cumulative Returns (CR) for the next day across the three cases: first case (training data only), second case (training data merged with GAN-generated data), and third case (GAN-generated data only). 153
5.24 Correlation matrix of the merged data (second case) with training data values in blue. 153
5.25 Correlation matrix of the generated data (third case) with training data values in blue. 154
5.26 Correlation matrix of the train log returns merged with generated data using a different set of hyperparameters (training data values in blue). 155
5.27 Mean estimates of the complete, train, and generated data. 158
5.28 Parameters for Synthetic Data Generation. 160
5.29 Correlation matrix of synthetic log returns after applying Cholesky decomposition. 161
5.30 Covariance matrix of the mean-centered synthetic log returns used for PCA analysis. 162
5.31 Eigenvectors and eigenvalues derived from the covariance matrix of the synthetic data. 162
5.32 Explained variance for each principal component from PCA of synthetic log returns. 162
5.33 Correlation matrix comparing generated data and training data (in blue). 164
5.34 Means of the generated data compared to the training data (in blue). 164
5.35 Covariance matrix comparing generated data and training data (in blue). 164
5.36 Means of the generated data compared to the training data (in blue). 166

5.37 Correlation matrix comparing generated data and training data (in blue). . . . 167
5.38 Covariance matrix comparing generated data and training data (in blue). . . . 167
5.39 Parameters for synthetic data generation. . . . 168
5.40 Correlation matrix of dependent synthetic data. . . . 168
5.41 Covariance matrix of the synthetic log returns for assets X1, X2, and X3. . . . 169
5.42 Eigenvectors and eigenvalues of the covariance matrix for the synthetic log returns. . . . 169
5.43 Means of the generated data compared to the training data (in blue). 170
5.44 Correlation matrix comparing generated data and training data (in blue). . . . 170
5.45 Covariance matrix comparing generated data and training data (in blue). . . . 171
5.46 Comparison of Train Data (in blue) and Generated Data for Min Variance and Max Sharpe Ratio portfolios, including the corresponding portfolio weights, risk, and returns. . . . 172
5.47 Parameters used for generating the prices of securities S1 and S2. . 174
5.48 Covariance matrix for the log returns of securities S1 and S2. . . . 175
5.49 Eigenvalues and corresponding eigenvectors of the covariance matrix for the log returns of S1 and S2. . . . 175
5.50 Explained variance and cumulative explained variance of the principal components. . . . 175
5.51 Estimated parameters from the generated data, comparing values from simulated train data and data generated by GANs. . . . 177
5.52 Estimated parameters from generated data with increased training points. . . . 177
5.53 Estimated parameters from generated data with hyperparameter tuning. . . . 178
5.54 Covariance matrix of S1 and S2 log returns. . . . 179
5.55 Eigenvectors and eigenvalues of the covariance matrix for S1 and S2 log returns. . . . 179
5.56 Explained variance and cumulative explained variance for the principal components derived from the log returns. . . . 180
5.57 Estimated parameters from simulated training data and GAN-generated data. . . . 181
5.58 Parameters used for simulating security prices in the third example. 182
5.59 Covariance matrix of S1 and S2 log-returns in the third example. . 182
5.60 Eigenvectors and eigenvalues of the covariance matrix for S1 and S2 log-returns. . . . 183
5.61 Explained variance and cumulative explained variance for the principal components derived from the log-returns in the third example. 183
5.62 Estimated parameters from simulated training data and GAN-generated data. . . . 184

5.63 Parameters for generating security prices (log-returns) with a strong correlation ($\rho = 0.95$). 188
5.64 Covariance matrix of S1 and S2 (log-returns) with a strong correlation ($\rho = 0.95$). 188
5.65 Eigenvectors and eigenvalues of the covariance matrix for S1 and S2 (log-returns) with a high correlation. 189
5.66 Explained variance and cumulative explained variance for the principal components. 189
5.67 Estimated parameters for b_1, b_2, σ_1, σ_2, and ρ from the training data (generated log-returns) and the GAN-generated data. 191
5.68 Estimated parameters from generated data over a longer time window (300 days). 192

References

[1] Louis Augustin Cauchy. "Compte Rendu 'a l'Acad´emie des Sciences". In: *Compte Rendu des Séances de l'Académie des Sciences* (1847).

[2] Harry Markowitz. "Portfolio Selection". In: (1952).

[3] Harry M. Markowitz. *Portfolio Selection: Efficient Diversification of Investments*. Yale University Press, 1959.

[4] R. Engle. "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation". In: *Econometrica* 50.4 (1982), pp. 987–1007.

[5] T. Bollerslev. "Generalized Autoregressive Conditional Heteroskedasticity". In: *Journal of Econometrics* 31.3 (1986), pp. 307–327.

[6] D. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[7] B. Flury and H. Riedwyl. *Multivariate Statistics: A Practical Approach*. Chapman & Hall, 1988.

[8] S. Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. Diploma thesis, TU Munich, 1991.

[9] K. Doya. "Universality of Fully-Connected Recurrent Neural Networks". In: *IEEE Transactions on Neural Networks* (1993).

[10] Paul Kupiec. "Techniques for Verifying the Accuracy of Risk Measurement Models". In: (1995). Available at SSRN: `https://ssrn.com/abstract=6697`.

[11] Philippe Jorion. "Risk2: Measuring the Risk in Value at Risk". In: *Financial Analysts Journal* 52.6 (1996), pp. 47–56.

[12] S. Y. Kung K. I. Diamantaras. *Principal Component Neural Networks: Theory and Applications*. John Wiley & Sons, 1996.

[13] S. Hochreiter and J. Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[14] Thom M. Mitchell. *Machine Learning*. Carnegie Mellon University: WCB McGraw-Hill, 1997.

[15] W. Duch and N. Jankowski. "Survey of Neural Transfer Functions". In: *Neural Computing Surveys* 2 (1999), pp. 163–212.

[16] F. Gers, J. Schmidhuber, and F. Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (1999), pp. 2451–2471.

[17] Philip Best. *Implementing Value at Risk*. 2nd. Chichester, England: John Wiley & Sons, 2000.

[18] C. Gourieroux and et al. "Sensitivity analysis of Values at Risk". In: *Journal of Empirical Finance* 7.3 (2000), pp. 225–245.

[19] S. Kustrin and R. Beresford. "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research". In: *Journal of Pharmaceutical and Biomedical Analysis* 22 (2000), pp. 717–727.

[20] F. Bryant and P. Yarnold. "Principal-component analysis and exploratory and confirmatory factor analysis". In: *Statistical Methods for Multivariate Analysis* (2001).

[21] K. Dowd. "Estimating VaR with Order Statistics". In: *Journal of Derivatives - J DERIV* 8 (2001), pp. 23–30.

[22] C. Granger and S. Poon. "Forecasting Financial Market Volatility: A Review". In: *Journal of Economic Literature* (2001).

[23] J. Berkowitz and J. O'Brien. "How Accurate Are Value-at-Risk Models at Commercial Banks?" In: *The Journal of Finance* 57 (2002), pp. 1093–1111.

[24] I.T. Jolliffe. *Principak Component Analysis*. Springer, 2002.

[25] D. Endres and J. Schindelin. "A New Metric for Probability Distributions". In: *IEEE Transactions on Information Theory* 49.7 (2003), pp. 1858–1860.

[26] S. X. Chen and C. Y. Tang. "Nonparametric Inference of Value at Risk for Dependent Financial Returns". In: *Journal of Financial Econometrics* 3.2 (2005), pp. 227–255.

[27] K. Inui, M. Kijima, and A. Kitano. "VaR is subject to a significant positive bias". In: *Statistics & Probability Letters* 72 (2005), pp. 299–311.

[28] Th.Post and H. Levy. "Does Risk Seeking Drive Stock Prices? A Stochastic Dominance Analysis of Aggregate Investor Preferences and Beliefs". In: *The Review of Financial Studies* 18.3 (2005), pp. 925–953.

[29] *Basel Committee on Banking Supervision. Basel II: International Convergence of Capital Measurement and Capital Standards: A Revised Framework - Comprehensive Version*. Bank for International Settlements, 2006. URL: `Available%20at:%20https://www.bis.org/publ/bcbs128.pdf`.

[30] Philippe Jorion. *Value at Risk, 3rd Ed.: The New Benchmark for Managing Financial Risk*. 3rd ed. McGraw-Hill, 2006.

[31] L. van der Maaten and G. Hinton. "Viualizing data using t-SNE". In: *Journal of Machine Learning Research* 9 (Nov. 2008), pp. 2579–2605.

[32] A. Graves and et al. "A Novel Connectionist System for Unconstrained Handwriting Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.5 (2009), pp. 855–868.

[33] J. Werner. "Risk and Risk Aversion When States of Nature Matter". In: *Economic Theory* 41.2 (2009), pp. 231–246.

[34] V. Nair and G. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *International Conference on Machine Learning*. 2010.

[35] G. Goerg. "Lambert W random variables - a new family of generalized skewed distributions with applications to risk estimation". In: *The Annals of Applied Statistics* 5.3 (2011), pp. 2197–2230.

[36] A. Krenker, J. Bešter, and A. Kos. *Introduction to the Artificial Neural Networks*. InTech Europe, 2011.
[37] Z. Bodie, A. Kane, and A. Marcus. *Investments*. McGraw-Hill, 2013.
[38] A. Graves and et al. "Speech Recognition with Deep Recurrent Neural Networks". In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 2013, pp. 6645–6649.
[39] K. Cho and et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *arXiv preprint arXiv:1409.1259* (2014).
[40] J. Chung and et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *arXiv preprint arXiv:1412.3555* (2014).
[41] I. Goodfellow and et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. 2014.
[42] R. Korn. *Moderne Finanzmathematik Theorie und praktische Anwendungen Band I*. 2014.
[43] I. Sutskever, O. Vinyals, and Q. Le. "Sequence to Sequence Learning with Neural Networks". In: *arXiv preprint arXiv:1409.3215* (2014).
[44] J. Chung et al. "Gated Feedback Recurrent Neural Networks". In: (2015).
[45] Ch. Olah. *Understanding LSTM Networks*. `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. [Online; accessed : July 10, 2024]. 2015.
[46] A.Radford, L. Metz, and S. Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *Proceedings of the 4th International Conference on Learning Representations (ICLR)*. 2016.
[47] G. Chen. "A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation". In: *arXiV* (2016). Preprint, arXiv:1606.00661.
[48] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.
[49] L. Metz and et al. *Unrolled Generative Adversarial Networks*. 2016. arXiv: `arXiv:1611.02163 [stat.ML]`. URL: `https://arxiv.org/abs/1611.02163`.
[50] S. Nadarajah and S. Chan. *Extreme Events in Finance*. Hoboken, New Jersey: John Wiley & Sons, 2016, pp. 283–356.
[51] T. Salimans and et al. "Improved techniques for training GANs". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2234–2242.
[52] C. Vondrick, H. Pirsiavash, and A. Torralba. "Generating Videos with Scene Dynamics". In: *Neural Information Processing Systems*. 2016.
[53] J. Wu et al. "Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling". In: *CoRR* (2016). URL: `http://arxiv.org/abs/1610.07584`.
[54] M. Arjovsky and et al. "Wasserstein generative adversarial networks". In: *International Conference on Machine Learning*. 2017, pp. 214–223.
[55] M. Arjovsky and L. Bottou. "Towards Principled Methods for Training Generative Adversarial Networks". In: *ArXiv* abs/1701.04862 (2017). URL: `https://api.semanticscholar.org/CorpusID:18828233`.

[56] J. Donahue, P. Krähenbühl, and T. Darrell. "Adversarial Feature Learning". In: *arXiv* (2017). Preprint, arXiv:1605.09782.
[57] Abhijit Ghatak. *Machine Learning with R*. Springer, 2017.
[58] M. Heusel and et al. "GANs trained by a two time-scale update rule converge to a local Nash equilibrium". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6627–6638.
[59] S. Huang and et al. "Stacked Generative Adversarial Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 5077–5086.
[60] S. Hyland, C. Esteban, and G. Rätsch. "Real-Valued (Medical) Time Series Generation with Recurrent Conditional GANs". In: *arXiv* abs/1706.02633 (2017).
[61] P. Isola et al. "Image-to-Image Translation with Conditional Adversarial Networks". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).
[62] G. Montavon, W. Samek, and K. Müller. "Methods for Interpreting and Understanding Deep Neural Networks". In: *CoRR* (2017).
[63] A. Srivastava and et al. "VEEGAN: Reducing Mode Collapse in GANs using Implicit Variational Learning". In: *Neural Information Processing Systems*. 2017.
[64] J. Zhu et al. "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks". In: *IEEE International Conference on Computer Vision (ICCV)*. 2017.
[65] J. Adler and et al. "Banach Wasserstein GAN". In: *Advances in Neural Information Processing Systems*. Vol. 2018-Decem. 2018, pp. 6754–6763.
[66] A. Caterini and D. Chang. *Deep Neural Networks in a Mathematical Framework*. University of Oxford: Springer, 2018.
[67] F. Chollet. *Deep Learning with Python*. Manning Publications Co., 2018.
[68] DataDrivenInvestor. *How do LSTM Networks Solve the Problem of Vanishing Gradients?* Accessed: 2024-07-08. 2018. URL: https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577.
[69] Ch. Donahue, J. McAuley, and M. Puckette. "Synthesizing Audio with Generative Adversarial Networks". In: *arXiv* (2018). Preprint, arXiv:1802.04265.
[70] Josh Kalin. *Generative Adversarial Networks Cookbook*. Packt, 2018.
[71] T. Karras and et al. "Progressive Growing of GANs for Improved Quality, Stability, and Variation". In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. 2018.
[72] T. Karras et al. "Progressive Growing of GANs for Improved Quality, Stability, and Variation". In: *CoRR* (2018).
[73] E. Thomas. *Understanding LSTM: An In-depth Look at its Architecture, Functioning, and Pros & Cons*. Accessed: 2024-07-08. 2018. URL: https://medium.com/@ebinbabuthomas_21082/understanding-lstm-an-in-depth-look-at-its-architecture-functioning-and-pros-cons-5424dd7215e6.

[74] Alireza Koochali et al. "Probabilistic Forecasting of Sensory Data With Generative Adversarial Networks – ForGAN". In: *IEEE Access* 7 (2019), pp. 63868–63880. ISSN: 2169-3536. DOI: 10.1109/access.2019.2915544. URL: http://dx.doi.org/10.1109/ACCESS.2019.2915544.

[75] T. Che and et al. "Mode Regularized Generative Adversarial Networks". In: *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. 2019.

[76] C. Donahue, J. McAuley, and M. Puckette. *Adversarial Audio Synthesis*. 2019. arXiv: 1802.04208 [cs.SD]. URL: https://arxiv.org/abs/1802.04208.

[77] S. Khan. "Calculate Value-at-Risk Using Wasserstein Generative Adversarial Networks (WGAN-GP) for Risk Management System". In: (2019). Accessed: 2024-07-05. URL: https://blog.chatbotslife.com/calculate-value-at-risk-using-wasserstein-generative-adversarial-networks-wgan-gp-for-risk-2b1d320fde59.

[78] J. Langr and V. Bok. *GANs in Action*. Manning, 2019.

[79] Niels Olaf. "Understanding GRU Networks". In: *Towards Data Science* (2019). Accessed: 2024-07-12. URL: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.

[80] R. Schmidt. "Recurrent Neural Networks (RNNs): A gentle Introduction and Overview". In: *ArXiv* abs/1912.05911 (2019).

[81] D. Warde-Farley and et al. "Improving generative adversarial networks with denoising feature matching". In: *5th International Conference on Learning Representations (ICLR)*. 2019.

[82] M. Wiese et al. "Quant GANs: Deep Generation of Financial Time Series". In: *Quantitative Finance* (2019).

[83] J. Yoon and D. Jarrett. *Time-series Generative Adversarial Networks*. 2019.

[84] A. Zhang and et al. *Dive into Deep Learning*. 2019.

[85] D. Saxena and J. Cao. "Generative Adversarial Networks (GANs): Challenges, Solutions, and Future Directions". In: *CoRR* abs/2005.00065 (2020).

[86] Nishchal J and N. Bhandari. "Computational Complexity of Gradient Descent Algorithm". In: (2021). Preprint available. DOI: 10.36227/techrxiv.14544000.

[87] Shalini Sinha. *Synthetic Time Series Data: A GAN Approach*. https://towardsdatascience.com/synthetic-time-series-data-a-gan-approach-869a984f2239. Accessed: August 15, 2024. 2021.

[88] Sh. Dubey, S. Singh, and B. Chaudhuri. "Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark". In: *Neurocomputing* (2022).

[89] Robert Buch et al. "Estimating the Value-at-Risk by Temporal VAE". In: *Risks* 11.5 (2023), p. 79. DOI: 10.3390/risks11050079.

[90] D. Guidotti, L. Pandolfo, and L. Pulina. "Leveraging Satisfiability Modulo Theory Solvers for Verification of Neural Networks in Predictive Maintenance Applications". In: *Information* (2023).

[91] D. Saxena and J. Cao. *Generative Adversarial Networks (GANs): Challenges, Solutions, and Future Directions*. 2023.

[92] M. Topal. *What is Mode Collapse in GANs*. Accessed: 2024-07-23. 2023. URL: https://medium.com/@miraytopal/what-is-mode-collapse-in-gans-d3428a7bd9b8.

[93] Ralf Korn. *Can we use GANs for Portfolio Optimization?* https://actuview.com/videos/can-we-use-gans-for-portfolio-optimization-3845. Access restricted to authorized users only. Sept. 2024. URL: https://actuview.com/videos/can-we-use-gans-for-portfolio-optimization-3845.

[94] Mbali Kalirane. *Gradient Descent vs. Backpropagation: What's the Difference?* https://www.analyticsvidhya.com/blog/2023/01/gradient-descent-vs-backpropagation-whats-the-difference/. [Online; accessed 05 Apr, 2023].

[95] Chris V. Nicholson. *A Beginner's Guide to LSTMs and Recurrent Neural Networks*. https://wiki.pathmind.com/lstm. [Online; accessed : 06 November, 2019].

[96] Chinnu Pittapally. *Backpropagation and Gradient Descent*. https://medium.com/@cpittapa/backpropagation-and-gradient-descent-369e33fb7466. [Online; accessed Feb 23, 2022].

[97] Frank J. Fabozzi and James L. Grant. "Modern Portfolio Theory, Capital Market Theory, and Asset Pricing Models". In: *Equity Portfolio Management*. John Wiley & Sons, Year. Chap. 2.

Scientific and Professional Career

Education

08/2021 – Present	PhD student of Prof. Dr. Ralf Korn, *RPTU Kaiserslautern-Landau*
10/2018 – 05/2021	M.Sc. in Actuarial and Financial Mathematics, *TU Kaiserslautern*
10/2014 – 09/2017	B.Sc. in Financial Mathematics in Banking and Insurance, *University of Prishtina "Hasan Prishtina"*
09/2011 – 06/2014	Natural Sciences, *Gymnasium "Zenel Hajdini," Gjilan*

Experience

04/2023 – Present	Research Assistant in the Financial Mathematics Group, *Fraunhofer ITWM, Kaiserslautern*
04/2022 – 03/2023	Research Assistant in the Financial Mathematics Group, *RPTU Kaiserslautern-Landau*
03/2021 – 06/2021	Research Assistant, *TU Kaiserslautern*
05/2020 – 07/2020	Research Assistant, *Fraunhofer ITWM, Kaiserslautern*
01/2018 – 05/2018	Consultant Intern, Euro Commerce, *Gjilan*
10/2017 – 12/2017	Intern, Anti-Money Laundering Dept., *ProCredit Bank, Prishtina*

Wissenschaftlicher und Beruflicher Werdegang

Ausbildung

08/2021 – heute	Doktorand bei Prof. Dr. Ralf Korn, *RPTU Kaiserslautern-Landau*
10/2018 – 05/2021	Masterstudium Finanz- und Versicherungsmathematik, *TU Kaiserslautern*
10/2014 – 09/2017	Bachelorstudium Finanzmathematik Im Bereich Banken und Versicherungen, *University of Prishtina "Hasan Prishtina"*
09/2011 – 06/2014	Abitur, *Gymnasium "Zenel Hajdini", Gjilan*

Erfahrung

04/2023 – heute	Wissenschaftlicher Hilfskraft in der AG Finanzmathematik, *Fraunhofer ITWM, Kaiserslautern*
04/2022 – 03/2023	Wissenschaftlicher Hilfskraft in der AG Finanzmathematik, *RPTU Kaiserslautern-Landau*
03/2021 – 06/2021	Wissenschaftliche Hilfskraft, *TU Kaiserslautern*
05/2020 – 07/2020	Wissenschaftliche Hilfskraft, *Fraunhofer ITWM, Kaiserslautern*
01/2018 – 05/2018	Praktikum, Euro Commerce, *Gjilan*
10/2017 – 12/2017	Praktikum, Anti-Money-Laundering-Abteilung, *ProCredit Bank, Prishtina*