École Polytechnique Fédérale de Lausanne

Conditional Synthetic Financial Time Series with Generative Adversarial Networks

A Master Thesis By:
Alexander Rusnak

Supervisors:
Dr. Stéphane Daul (Pictet Asset Management)
Prof. Frédéric Kaplan (EPFL)

EPFL

Digital Humanities Laboratory

Lausanne, July 29, 2022

1 Abstract

The creation of high fidelity synthetic data has long been an important goal in machine learning, particularly in fields like finance where the lack of available training and test data make it impossible to utilize many of the deep learning techniques which have proven so powerful in other domains. Despite ample research into different types of synthetic generation techniques, which in recent years have largely focused on generative adversarial networks, there remain key holes in many of the architectures and techniques being utilized. In particular, there are currently no techniques available which can generate multiple series concurrently while capturing the specific stylized facts of financial time series and which incorporate extra information that effect the series such as macroeconomic factors. In this thesis, we propose the Conditional Market Transformer-Encoder Generative Adversarial Network (C-MTE-GAN), a novel generative adversarial neural network architecture that satisfies the aforementioned challenges. C-MTE-GAN is able to capture the relevant univariate stylized facts such as lack of autocorrelation of returns, volatility clustering, fat tails, and the leverage effect. It is also able to capture the multivariate interactions between multiple concurrently generated series such as correlation and tail dependence. Lastly, we are able to condition the generated series both on a prior series of returns as well as on different types of relevant information that typically effect both the characteristics of the market and factor into asset allocation decision making. Furthermore, we demonstrate the effectiveness of data generated by C-MTE-GAN to augment training of a statistical arbitrage model and improve its performance in realistic portfolio allocation scenarios. The abilities of this architecture represent a substantial step forward in financial time series generation which will hopefully unlock many new applications of synthetic data within the realm of finance.

2 Introduction

2.1 Background

The modeling of financial time series is at the core of any quantitative investing strategy: whether it is portfolio allocation, return prediction, instrument pricing, backtesting, or risk analysis, they rely fundamentally on capturing information about the behaviour of financial time series and then making investing decisions based on this modeled understanding. Traditionally, the models which were used to capture and understand information about financial time series were closed form solutions predicated on explicit definitions of a few variables. However, mirroring the same process in many other fields, the rise of data driven techniques for financial modeling has radically changed the landscape of quantitative finance. Machine learning modeling has made it possible to holistically model many behaviours of financial time markets, but it has also starkly clarified the issue of low data availability in finance. There have been many attempts to rectify this issue and provide further test or training data for machine learning models, and one that has recently gained traction in academic literature as well as in direct application are generative adversarial networks (GAN) [9].

2.2 Use cases of Synthetic Financial Data

There are many potential use cases of synthetic data within finance; it is a field whose challenges make generating data particularly useful. In particular, there is often comparatively very little data available at most relevant time scales, and that data is often very imbalanced. For example, market crashes are relatively rare events in the modern history of finance but are perhaps the most important events to understand and model well if you hope to have success in the long term in finance. Furthermore, tasks like fraud detection are even more heavily imbalanced with thousands of regular transactions for every fraudulent one. Beyond class imbalances, it is often advantageous to test models under particular conditions or adjust decisions based on the status of the market, therefore conditional data is quite useful in the context of finance. There have been some attempts to use GANs for conditional forecasting, but other model architectures created specifically for this purpose are more successful. Generally the use cases for synthetic data in finance fall under 3 main headings: creating extra training data for other models, creating extra testing data for other models, and calibrating other models.

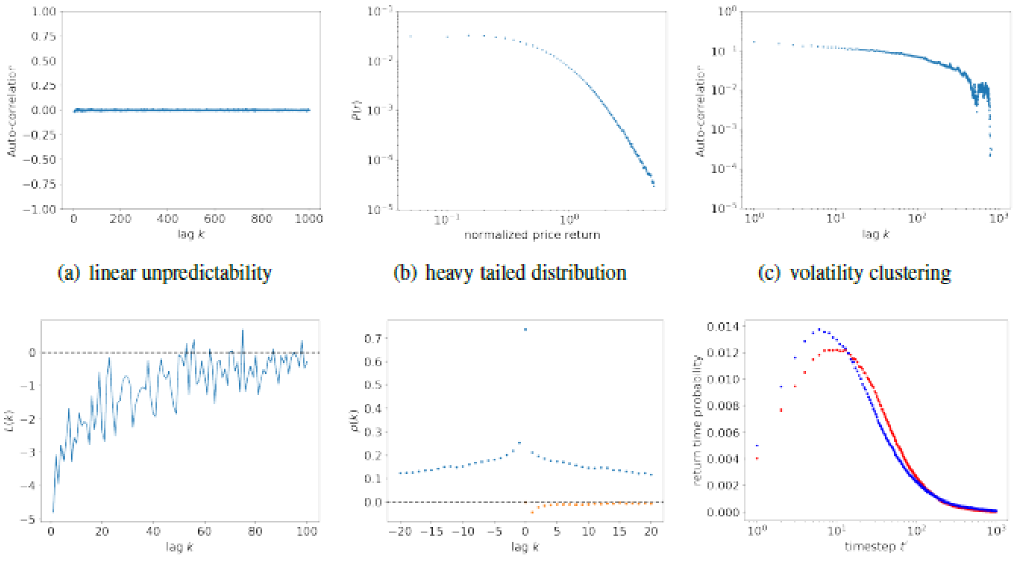
2.3 Unique Challenges of Financial Data

The generation of synthetic financial returns presents certain unique and specific challenges that are not relevant when considering other types of time series or data types in which GANs have shown incredible performance such as images. Firstly, there is a set of stylized facts about financial time series which must be captured in order to accurately recreate useful and realistic synthetic series. Secondly, there is no 'eye test' for financial series which can be used in lieu of an explicit analysis in the way there is for other data types. Thirdly, the usefulness of the generated data is heavily dependent on the expected use case, and thus mediates trade offs that are not as important when generating other data types.

The main univariate (i.e. on a single series) stylized facts that are important to capture about financial time series are: lack of auto-correlation of daily returns, fat tails in the distribution of returns, volatility clustering, and the leverage effect. The main multivariate (i.e. between multiple series) stylized facts are the correlation structure and tail dependence between the returns. In real financial time series, there is no correlation across time between the returns of a single asset. In contrast to a normal distribution, financial returns exhibit a fat tailed distribution where there are more very high or very low values (kurtosis). Financial returns also have heteroskedasticity; the variance of the sample is not constant and specifically appears as volatility clusters where periods of high or low volatility arise in conjunction with each other in time. In other words, the returns exhibit auto-correlation of volatility. Lastly, there is a negative correlation between lagged returns and volatility called the leverage effect. In multivariate situations it is also crucial to capture the correlation structure between distinct assets, which is often complex and varies from asset to asset strongly. Likewise, financial markets exhibit a property of tail dependence, the dependence between returns of different assets increases at extreme values. When all returns are within a relatively typical range, they are more uncoupled then when they are at the tail ends of the distribution. A simple example of this is a market crash, where most assets in a market simultaneously have very negative returns.

In comparison, generating a series such as energy consumption from an electric grid is far easier. It has correlation and clear structures like seasonality, it is less random. The predictive conditioning variables, such as weather data, are far more related to the actual behaviour of the series then something like macroeconomic data is for financial series. This data also has more consistent variance that does not cluster in time. It is clear generating realistic financial series is one of the hardest problems in time series modeling generally.

Another key challenge in generating financial series is that there is no suitable subjective heuristics for determining quality to guide architecture decisions, particularly in multivariate cases. A key motivation for utilizing GANs is in situations where it is not possible to explicitly define what makes data 'realistic', there is no loss function that accurately captures all the facets of 'realness.' For many machine learning tasks, such as classification, it is possible to say with certainty whether a prediction is correct relative to a set of labels for the particular sample being analyzed. For example, in the classic 'hot dog or not hot dog' example, it is relatively simple to provide a set of binary labels for images denoting whether they contain a hot dog or not. Thus, it is far simpler to train a model to predict this information, because you can make an explicit evaluation of its performance. But what about determining whether a given image of a hot dog is realistic? This is essentially impossible to define explicitly, and most attempts at doing so end up focusing on one specific



(a) linear unpredictability (b) heavy tailed distribution (c) volatility clustering

(d) leverage effect (e) coarse-fine volatility correlation (f) gain/loss asymmetry

Figure 1: Stylized Facts of Financial Time Series [16]

aspect of the data while neglecting others. This broadly defines the importance of having a learned discriminator for the realistic of generated data: it can capture many aspects of what it means to be realistic without having to define it and if one aspect becomes too dominant it can adjust its criteria to better evaluate the generated data. While this approach is powerful, it does have a downside: it renders the loss values of the discriminator mostly meaningless and it is not possible to reach an objective measure of convergence or quality. For data types like images or music, it is possible to use the rough heuristic of 'the eye test' to determine quality outside of statistical measures. You can listen to the generated music and subjectively assess how pleasing it is, you can look at a generated image of a hot dog and quickly but roughly evaluate its quality. But this is not really possible with long financial series, particularly when you are generating multiple series concurrently. Thus it is even more difficult to evaluate generating financial data relative to other data types.

A good way to determine the realisticness of generated data is to test how beneficial it is for the downstream test you intend to use it for. In practice, this often is using the generated data to augment the training dataset or test dataset for a model. In the case of financial data, the usefulness of the generated data is mediated by the downstream task you intend to use it for more strongly than in other time series tasks. We can take electrical load forecasting as a an example again, if you simulated an electrical load path, most of the downstream tasks are focused on the same aspects of the data; i.e. its usefulness is primarily predicated on correctly timing the seasonality to determine peaks and troughs of consumption. For financial series, it is not so simple. If you intend to use the data for testing the performance of a trading model, even something as simple as the rebalancing speed influences whether the

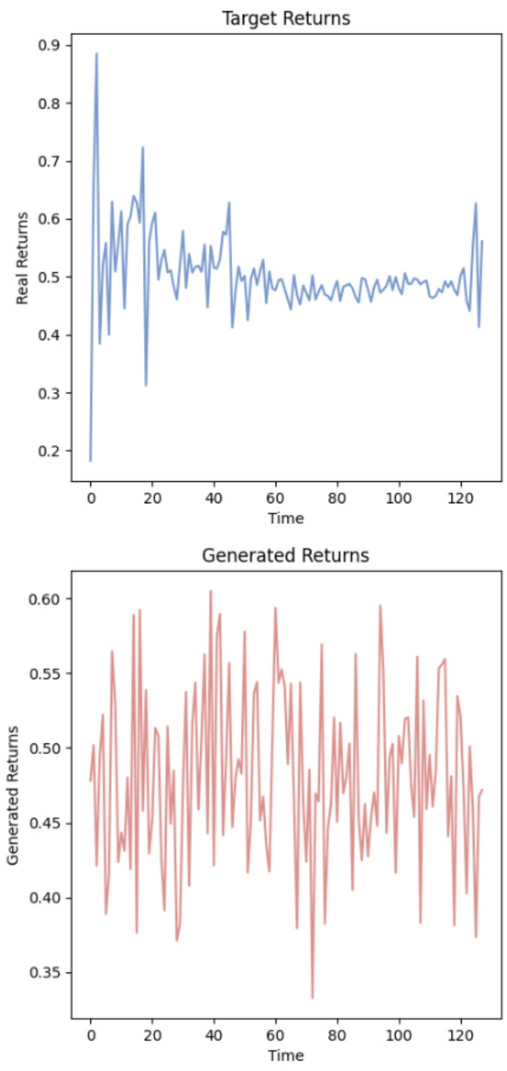


Figure 2: Real Series and a Generated Series without the Relevant Stylized Facts

data you generated is actually useful (i.e. whether it is important to more faithfully model long term trends or microsecond fluctuations). If you are looking to evaluate tail risk to a portfolio you need to be able to generate specifically extreme events whereas if you want to generate long paths relative to particular macroeconomic conditions, it is better for your model to have a more general representation of the whole distribution of the data. Because of the way the varied use cases of financial time series effect the importance of particular aspects of generated data, the usefulness of prior downstream task specification adds an additional layer of complexity to generating synthetic financial data.

Figure 3: Poorly Generated Images of Faces

2.4 Traditional Approaches to Generating Novel Financial Time Series

There have been many proposed techniques for rectifying data scarcity in finance, but two of the most widely adopted are utilizing a generalized autoregressive conditional hetereoskedasticity (GARCH) process to create synthetic series or using bootstrapping to resample the points from real series into novel paths. An ARCH process can be defined as a process where the variance at time t is conditional on the previous m data points. Simply put, GARCH is an autoregressive euation that can be used to generate time series that have close to the real statistics of financial time series, especially heteroskedasticity (volatility clustering). Though GARCH is powerful, it does not perfectly recreate the statistics of a real financial time series and it is overly simplistic (very few parameters) which makes the data it creates unsuitable for use in augmenting the training set of a model or to accurately create alternate synthetic backtest data. Especially because it can not be conditioned on external information like macroeconomic conditions.

$$y_t = x_t'b + \epsilon_t$$

$$\epsilon_t | \psi_{t-1} \sim \mathcal{N}(0, \sigma_t^2)$$

$$\sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \cdots + \alpha_q \epsilon_{t-q}^2 + \beta_1 \sigma_{t-1}^2 + \cdots + \beta_p \sigma_{t-p}^2 = \omega + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2$$

Figure 4: Equations for GARCH Process [1]

Another popular approach to generating synthetic financial time series is bootstrapping. At its base, bootstrapping is essentially just the resampling of points in the series to create novel series. If you resample single data points, you will preserve the overall distribution of the data, but you will lose many of the stylized facts of financial time series such as autocorrelation of volatility, that occur across the temporal dimension of the data. It is therefore common practice to resample the data in chunks of a particular length (block bootstrapping), but this can also raise problems if the blocks are a consistent size. The temporal statistics of the data will always be disturbed at the junction of the series sample, for instance if your block size is 10 points then the maximum total memory of autocorrelation is 10. The larger you make your block, the better you are able to preserve the features of the series, but the less novel the new series will be. Stationary bootstrap attempts to solve this issue by resampling with random block sizes, where the size of each block has a geometric distribution. This technique generates more novel series that better preserve the statistics of the dataset, but it is limited for multiple reasons. Firstly, It is not possible to generate a series with any type of of conditioning which limits its use in many possible applications of synthetic time series. Secondly, it is still fundamentally just resampling of data, so the actual novelty of the data while always be far lower than true generation, which also reduces the use cases in which it is applicable.

There are many other techniques which are used to resample or generate wholly novel sequences, but the two with the most adoption (which are not based on deep learning) are ARCH models and stationary bootstrap.

2.5 Deep Generative Models

Generative models based on neural networks have had tremendous success at creating realistic synthetic data across a range of data types and fields. The main paradigm in generative neural networks is centered around various self-supervised techniques, starting with autoencoders in the 1980s [8] before progressing to generative adversarial networks in 2014 with the landmark paper by Ian Goodfellow [9]. Autoencoders are an architecture of model which is predicated on directly replicating individual samples of data, and utilizing some form of reconstruction error for the loss function. Usually, autoencoders receive a sample of data into the encoder side of the network, which is structured to reduce the dimension of the data with each successive layer. The output of the final layer of the the encoder is traditionally called a latent vector. This output is then passed to the decoder which upscales the data through its successive layers, and then outputs its best attempt at a perfect mimicry of the the input sample. This approach is relatively successful at recreating samples, but it has many key deficiencies. Because each sample is mapped to a latent vector, the decoder becomes good at utilizing the particular vectors which correspond to the individual samples in the dataset. However, it is usually not good at utilizing vectors in between those between or not mapped to an input sample. Essentially, this means that the model is good at generating data that is exactly the same as the input samples, but not at generating data which is similar (i.e. conforming to the relevant statistical properties but otherwise novel). As an attempt to rectify this, a particular class of model called a variational autoencoder gained popularity. Instead of mapping the sample data to an explicit latent vector, the encoder maps the sample to only two values; a mean and standard deviation. A vector is then sampled from a normal distribution with the corresponding mean and standard deviation, and then passed to the decoder, which tries to recreate exactly the input sample. This regularizes the latent space, and helps the model learn a more general, less brittle latent representation of the data. This makes it easier to generate realistic outputs that are similar to the given dataset but not identical. It is also possible to sample many points from the latent distribution (i.e. the normal distribution with corresponding mean and standard deviation) and give those points directly to the decoder, with no input from the encoder. This makes it far easier to generate many new synthetic samples. However, this model is still not great at generating truly novel samples that are realistic but otherwise nothing like the given dataset. Furthermore, it is always limited by the type of reconstruction loss that is used to measure the 'realness' of the data. It is possible to use many types of reconstruction errors, distance measures, or statistical tests to create an explicit definition of the realisticness of the data, but ultimately all of these rely on expert knowledge and naturally preclude the model from learning any representations of the data which does not correspond to the pre-defined measures. Furthermore, in practice, there is usually no effective and all encompassing distance measure that results in subjectively realistic data at the level of other approaches.

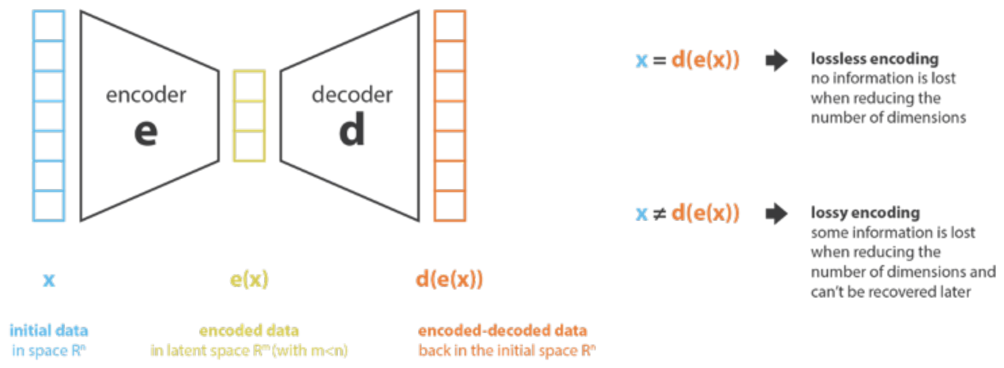

encoder
e
decoder
d
x
e(x)
d(e(x))
initial data
in space R^n
encoded data
in latent space R^m (with m<n)
encoded-decoded data
back in the initial space R^n
x = d(e(x))
lossless encoding
no information is lost when reducing the number of dimensions
x ≠ d(e(x))
lossy encoding
some information is lost when reducing the number of dimensions and can't be recovered later

Figure 5: Encoder-Decoder Architecture [2]

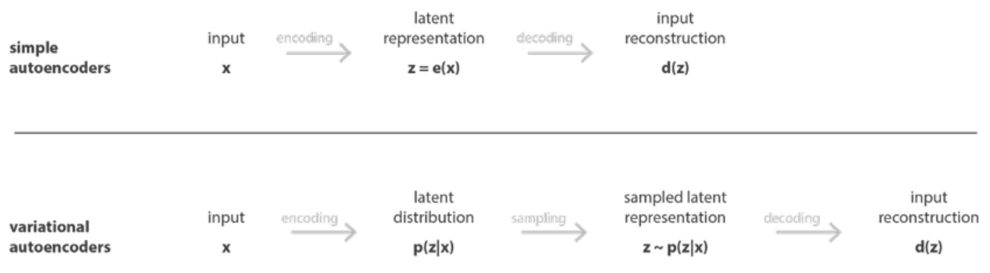


Figure 6: Difference between Autoencoder and VAE [2]

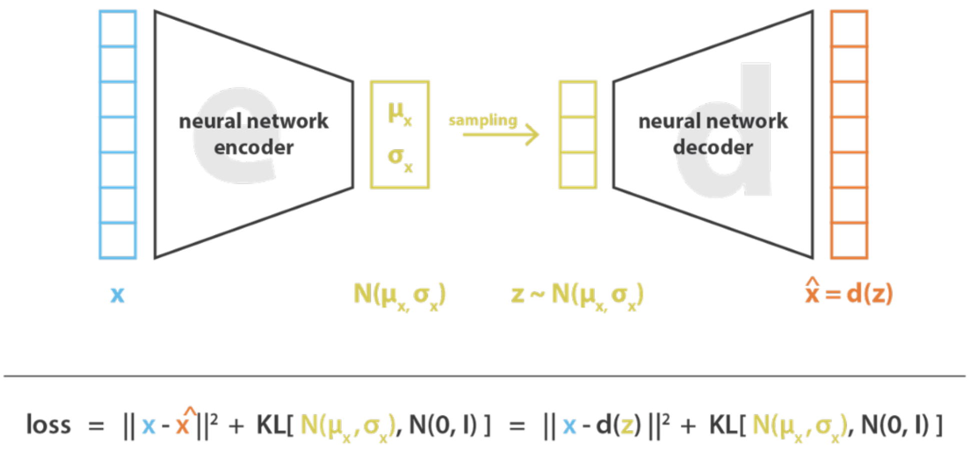

neural network encoder
μ_x
σ_x
sampling
neural network decoder
x
$N(\mu_x, \sigma_x)$
$z \sim N(\mu_x, \sigma_x)$
$\hat{x} = d(z)$
loss $= \| x - \hat{x} \|^2 + KL[N(\mu_x, \sigma_x), N(0, I)] = \| x - d(z) \|^2 + KL[N(\mu_x, \sigma_x), N(0, I)]$

Figure 7: VAE [2]

Generative adversarial networks are another broad architecture of neural network that seeks to solve the prior problems and to generate convincing and useful synthetic data. The structure of a GAN is similar to a variational autoencoder, but with some key differences. GANs do not have an encoder, and their latent space is made up of random vectors sampled from a single distribution. The generator of the GAN (roughly analogous to the decoder of a variational autoencoder) creates a rough facsimile of the target data from this latent vector. This generated data and the real data are passed to a binary classifier (usually called the discriminator) which attempts to determine which samples are real, and which are fake. The loss for the generator is usually the inverse of the binary cross entropy of the classifications, in other words: how effectively it is able to fool the discriminator into classifying the fake samples as real. The loss for the discriminator is the BCE on the classifications of the fake and real samples; how well the discriminator is able to classify fake and real samples. This dual network structure creates an inherent tension (adversarial relationship) between the generator and discriminator, and allows them to push each other to more effective performance. Because the efficacy of the generator is not tied to a one to one comparison of an input and output sample, it ideally learns a far more general and regular representation of the target dataset. Furthermore, because the measure of realisticness is not explicitly defined, it allows the generator to learn novel facets of the target dataset that would not have been measured (and thus not captured) by a non-discriminator based loss. GANs have shown their efficacy across many different domains, and have been able to generate convincing and useful data for many applications.

Despite their strong ability to generate synthetic samples, GANs have multiple limitations and challenges. The training of GANs is far more difficult and less straightforward than a typical machine learning problem due to their adversarial structure. Because the loss value for both the discriminator and generator are related to each

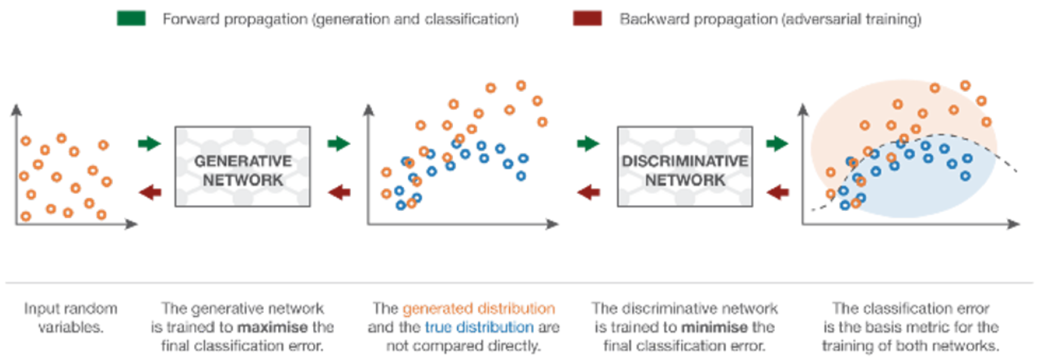

Forward propagation (generation and classification)
Backward propagation (adversarial training)
GENERATIVE NETWORK
DISCRIMINATIVE NETWORK
Input random variables.
The generative network is trained to **maximise** the final classification error.
The generated distribution and the true distribution are not compared directly.
The discriminative network is trained to **minimise** the final classification error.
The classification error is the basis metric for the training of both networks.

Figure 8: GAN Architecture [3]

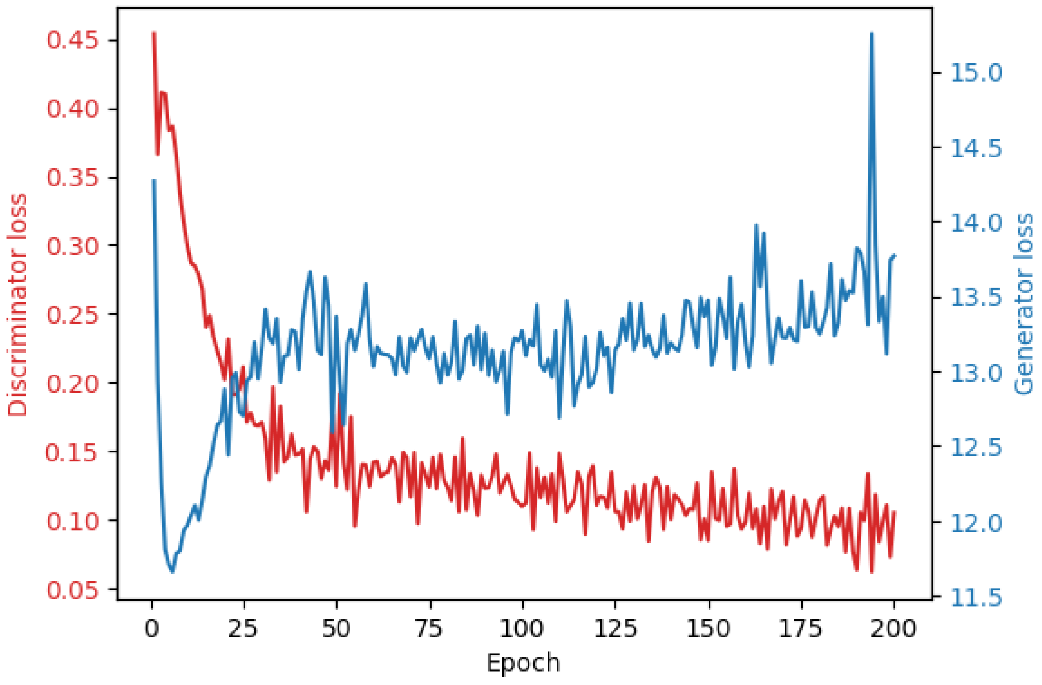


Figure 9: GAN Training [3]

other, their values are essentially meaningless. In a traditional machine learning task, such as classification, a lower loss value always represents better performance (outside of overfitting). However, in the training of a GAN, it is possible for the discriminator loss or generator loss to be either very high or very low while the generated samples still improve in quality with ever successive epoch. There is essentially no way to accurately determine when a vanilla GAN has converged to the optimal solution possible. In simple cases, reaching the point where the discriminator is simply guessing can mean that the generated data has achieved sufficient quality that is is indistinguishable from the real data, however in practice with complicated datasets this rarely happens or means that the training of the GAN has just broken down.

It is generally preferable to keep the loss values of the generator and discriminator close to each, to prevent degenerate training from one of the networks 'winning' their battle. If the discriminator is able to perfectly delineate the real and fake samples, no information will be passed back to the generator during back propagation and the generated samples will not improve. Likewise, if the generator is able to fool the discriminator into simply guessing, especially early in training before the discriminator has learned much about the target data, it prevents the discriminator from improving itself and making more accurate classifications. An ineffective discriminator is easy to fool, and thus the actual realisticness of the generated data will be low even if judging by the loss values it seems to be very effective. In the training of GANs there is also usually an imbalance between the discriminator and generator. With a sufficiently complicated target dataset, it is usually far harder to generate realistic to data than it is to discriminate which data is real and fake. To rectify this imbalance, sometimes unbalanced architectures are used where the discriminator is smaller or less complex than the generator. Outside of divergent discriminator and generator architectures, it is usually necessary to have differential learning rates and dropout rates for the two networks in favor the generator. Another potential pitfall in the training of GANs is the prevention of mode collapse. Mode collapse refers to a peculiar situation where the generator only learns to generate a subset of the target dataset. In this case, the generated samples are realistic and able to fool the discriminator but do not accurately recreate the total distribution of the target data. An example of this can be found in trying to generate fake samples of the classic MNIST handwritten digits image dataset. The images in this dataset all correspond to a single handwritten digit from 0 to 9. An effectively trained GAN would map each digit to a different part of the latent space, and could generate synthetic samples of each digit that are realistic. A GAN suffering from mode collapse would learn to generate only one type of digit realistically rather than every type, and thus not accurately model the target data. This problem will be discussed in more detail in the section discussing model architectures, because it influenced some of our decisions about the model. In general, the training of GANs is very unstable and requires careful monitoring in addition to a series of specific tricks to stabilize it.

An extension of the vanilla GAN architecture is a conditional GAN in which some extra information is passed to the generator along with the latent vector at each step, in order to influence the generated output. After generation, the real input condition is attached to the corresponding generated sample before it is passed to the discriminator in conjunction with the real conditions and corresponding real samples. The discriminator is then able to make judgements about not only the realisticness of the samples themselves, but also the realisticness of their relationship to the conditioning data. The conditioning can be any type of information, such as some sort of descriptive data that provides context; in finance it could be macroeconomic data, for image generation it could be the time of day for understanding lighting conditions. The idea of conditioning is more important and profound in the context of sequential data, because the condition can be the prior step in your sequential

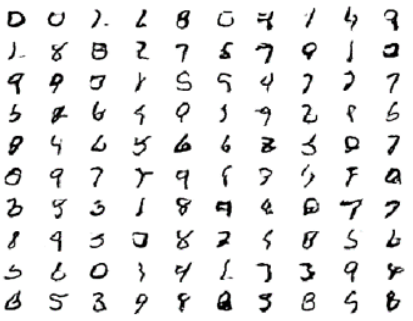


Figure 10: Generated MNIST Digits from a properly trained GAN.

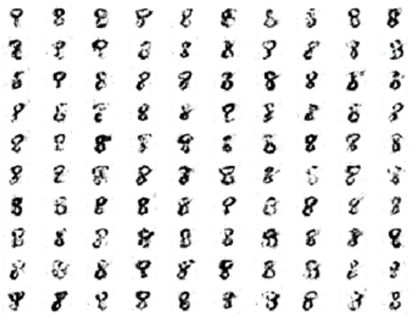


Figure 11: Generated samples from a GAN exhibiting mode collapse and only creating the digit 8.

data. In the context of video generation, it would be the preceding still frame. In the context of financial time series, it would be the prior steps in the sequence or sequences. This allows the network to have a better idea of what realistic data is across the time dimension and gives more control over the generation to create more specific sequences. Conditional generation also makes it possible to generate new sequences starting from a particular point in time, which is useful for many applications in finance if you want to simulate a particular point in time, including the present moment. Furthermore, it allows for auto-regressive generation of long sequences, instead of generating a sequence of hundreds or thousands of steps in one shot, smaller length sequences can be generated and then passed back as a conditioning input until a sequence of the desired length is achieved. This gives more flexibility to the generated data across the length of the sequence.

2.6 GANs in Finance Literature Review

In recent years, there has been a surge of interest in the application of GANs to financial data and financial time series. It is beyond the scope of this paper to complete

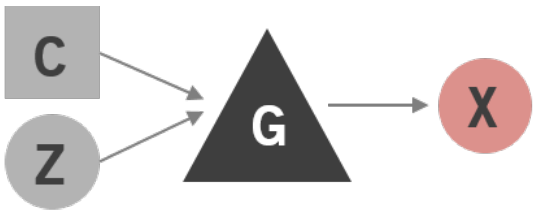


Figure 12: Conditioning the Generator (C = Condition Data, Z = Latent Vector, G = Generator, X = Generated Sample)

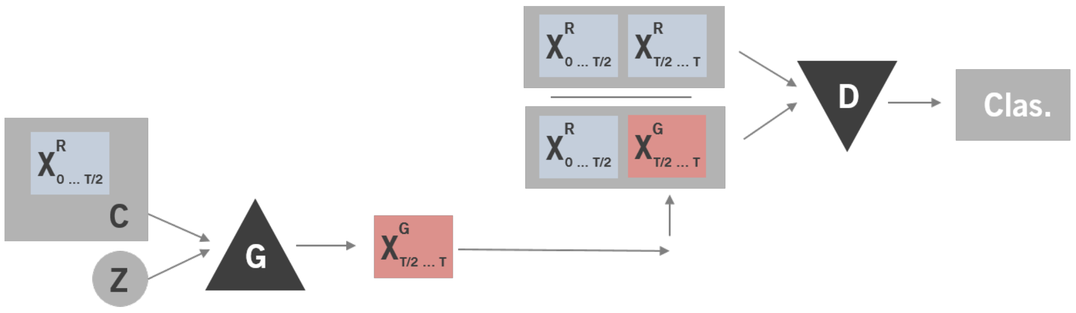


Figure 13: Conditioning on Prior Series

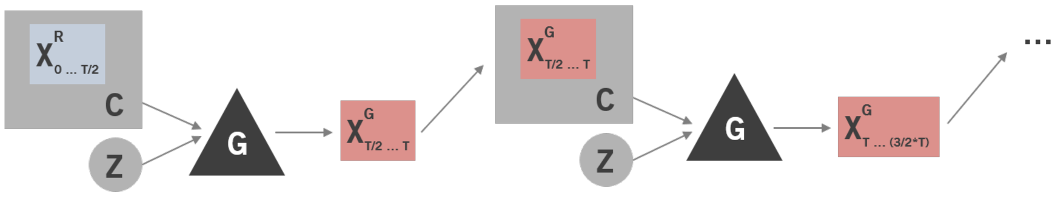
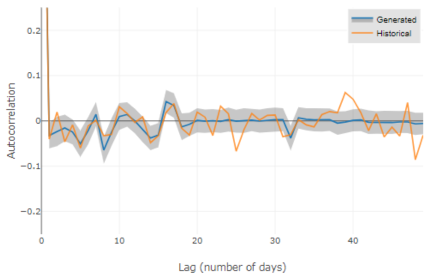


Figure 14: Autoregressive Generation of Long Sequences using Conditioning

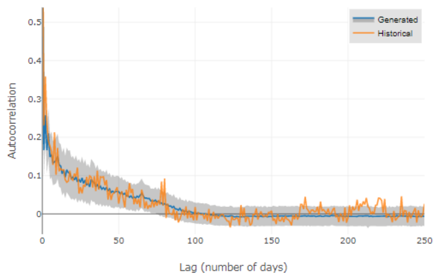
a total survey of every application of GANs in finance, especially considering there are already good references for this [6]. Instead, this section will focus on relevant papers for the generation of synthetic time series both for establishing benchmarks and tracing the genealogy of our approach. An early and relevant GAN paper in finance was *Quant GANs: Deep Generation of Financial Time Series* by Magnus Wiese et al [18]. This paper is based on the landmark Wavenet paper from Deepmind [14], and utilizes temporal convolutional networks in both the generator and discriminator to capture the temporal dynamics of the target data more accurately. There will be a deeper explanation of the TCN architecture in the modeling section of this paper. The key improvement of this technique were the ability of the TCN to capture long term dependence in the series, and thus model more complex stylized facts like autocorrelation of volatility while generating the series sequentially. This model architecture was able to capture the univariate stylized facts of financial time series, but it has some key limitations. Firstly, the model is relatively large and even more unstable in training than a traditional GAN. Secondly, The architecture explicitly is structured to only receive series, which makes both autoregressive generation from a latent space awkward and conditioning unnatural. The TCN architecture is designed for sequence to sequence problems where a prior sequence is given as input and then the next value in the series is predicted and then used as input again to generate autoregressively. However, this approach is predicated on the assumption that the input series is of the same type as the output series. In the case of the QuantGAN, the input sequence is a latent vector and the output is a real series, so it is not possible to append the output value to the end of the latent vector for autoregressive generation. It is necessary instead to shift the latent vector back by the generation step size, and append new latent variables to the end in order to make a partially novel latent vector. Then the corresponding new datapoints in the sample are appended to the previously generated series. This approach is not totally ineffective but it has two main limitations: first, the relative value of a particular latent variable at the position t=0 is not the same as at t=1 and so forth. In other words, when you slide the latent vector backwards in time, the corresponding generates series is not perfectly consistent with the series that was generated from the same latent variables at a different position in the time dimension. Second, the memory of the network is limited by the network window size, so any feature, stylized fact, or relationship in the series that is longer than the size of the sample generated at each step can't be captured. This can be mitigated by conditioning the network on the prior series as well as the latent vector, but this is also ungainly given the structure of the network, either the series must be appended to the latent vector or vice versa, implying they form one consistent sequence. Though this approach was successful for the univariate case and without conditioning, we found it insufficient for our purposes.

Another impressive paper also published in 2019 was *Modeling financial time-series with generative adversarial networks* by Shuntaro Takahashi et al [16]. The author propose a joint multilayer perceptron / convolutional neural network architecture for the generator that utilizes both network architectures concurrently and then combines their output before passing it to the discriminator, presumably with the fully connected network modeling large scale or long term relationships in the data and the convolutional network modeling more local relationships in the series. This approach is highly effective in its ability to capture stylized facts about the data, but it likewise also has some limitations. The generation window size of this approach is n=8192; it generates a single long series at once. This approach precludes local conditioning, or any sort of control over the series over time. Furthermore, their approach is univariate, and many applications of synthetic data require multivariate data or the synthesis of a market.

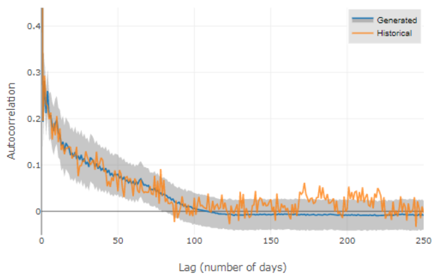
In 2020, Hao Ni et al published *Conditional Sig-Wasserstein GANs for Time Series Generation* [13]. Their apporach centered on conditional auotregressive generation using a feed-forward architecture for the neural network component. The condition data and generated data both were regular series but the loss was measured on the signatures of the real and fake data rather than its native form. A total exploration of the signature of a path is outside of the scope of this thesis, but it fundamentally



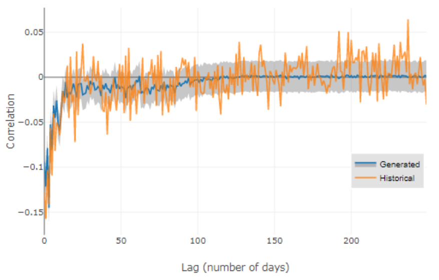
(a) Serial



(b) Squared

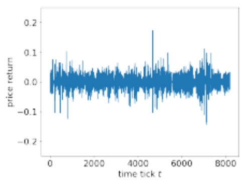


(c) Absolute

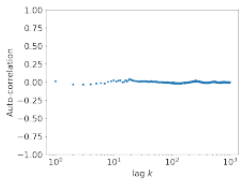


(d) Leverage Effect

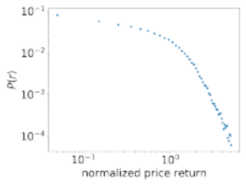
Figure 15: QuantGan Results



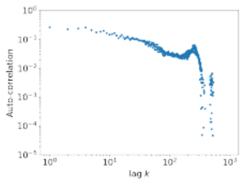
(a) time-series



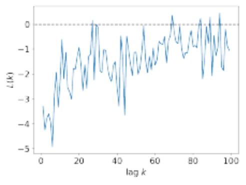
(b) linear unpredictability



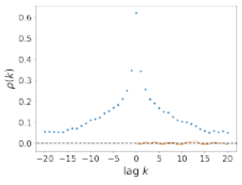
(c) heavy-tailed distribution



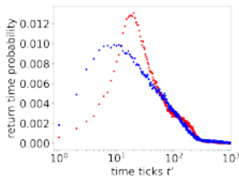
(d) volatility clustering



(e) leverage effect



(f) coarse-fine volatility correlation



(g) gain/loss asymmetry

Figure 16: FINGAN results

an embedding technique using iterated integrals to capture moments of the path. This makes the learning problem simpler for the generator, and allows them to use a fairly simple critic predicated on linear regression rather than a deep neural network. Since this is a wasserstein GAN, the loss function is not BCE and there is a critic instead of a discriminator. Instead of classifying each sample, the generator is seeking to minimize the wasserstein distance (earth mover's distance) between the outputs of the critic for the real and synthetic data. The critic is trying to differentiate the samples, and maximize the distance. The motivation for using wasserstein loss is that it prevents mode collapse and forces the model to attempt to generate the whole distribution. All of these innovations mean the model is able to capture the distribution and temporal dynamics relatively well, though they provided little information about the stylized facts we have mentioned outside of autocorrelation and correlation between the multiple series they generated (Standard and Poor 500 Index and Dow Jones Index). Another positive result is their ability to generate extremely long series that still conform to a realistic distribution, even up to 80,000 steps. However, since there is no explicit map from a signature of a path back to a path, all paths must be approximated by monte carlo simulation.

One month after the CSWGAN paper was published, Thierry Roncalli et el. published *Improving the Robustness of Trading Strategy Backtesting with Boltzmann Machines and Generative Adversarial Networks* [11]. This paper largely summarized the state of RBM and GAN generated synthetic series, but their headline model was a conditional deep convolutional wasserstein GAN that utilized a neural network discriminator and utilized real return data rather than signatures. Their model generated 2 series, with a generation window size of 5, and a latent vector dimension of 100. They did not provide information regarding the stylized facts of their generated series outside of autocorrelaton of returns (which they captured accurately). Our own implementation of this architecture confirmed that it was unable to capture more complex stylized facts, and did not scale well to more series. They demonstrate the usefulness of their generated data for estimating the probability distribution of performance and risk statistics in a simulated backtest, validating its usefulness in a more applicable scenario.

In July of 2021, Florian Eckerli et al. made *Generative Adversarial Networks in Finance: An Overview* [6] available. This paper neatly summarizes many of the application of GANs in finance, especially the more specific or targeted use cases that have evolved around the core problem of time series generation. The authors also make a comparison between 3 architectures of GANs for univariate, non-autoregressive generation of financial time series in the spirit of *Modeling financial time-series with generative adversarial networks.* They compare a simple deep convolutional GAN, a a wasserstein GAN with gradient penalty, and a self-attention GAN. They found that self-attention GAN was best able to capture the volatility clustering and general characteristics of the time series. Though the paper was not cutting edge in their approach to modeling, this result further confirmed the efficacy of self-attention and was prescient towards our own success utilizing self-attention in our networks.

In March of 2022, Paul Jeha et al. continued to demonstrate the usefulness of self-attention in the context of time series generation with their paper: *PSA-GAN: Progressive Self-Attention GANs for Synthetic Time Series* [15]. Though the focus of this paper was not explicitly financial time series, it introduced several key innovations that influenced our work. They utilized a relatively deeper architecture than other approaches, predicated on a main block with a spectral normalized convolutional layer and residual self-attention. The width of the layer size of their blocks grows progressively with each successive block, allowing them to build step by step from a smaller space to a long series. The training schedule of the blocks follows Karras et al. (2017) [10], the model trains for 1000 epochs at the smallest resolution, then a new block is appended with a larger size and the model resumes training at this higher level of resolution. They also incorporate conditioning on prior series values to enable autoregressive generation in their PSA-GAN-C model. The model also receives time features like time of day, day of the year, day of the week, etc. which are relevant for the type of time series they are looking to generate. This model showed great results for simulating multiple different time series that were generally seasonal and related to load balancing (i.e. solar panel output and electricity usage) and some of their techniques have proven useful when applied to financial time series in our research.

3 Datasets and Pipelining

We used multiple target datasets in our experiments in order to confirm the ability or inability of our models to capture specific stylized facts under controlled conditions, their ability to model multiple time series from different assets classes, or to utilize the generated data on downstream tasks. In this section we will explore the various datasets we utilized and the pipelining process we created to prepare these datasets for modeling.

3.1 Synthetic or Sampled Target Data

In order to capture the simplest efficacy of our models, we utilized two broad types of fake data as targets for our models. The first were series made of random variables sampled from either Student-t or gamma distributions. Both of these distributions were relatively easy to capture for any of our models because they had no temporal dynamics and were not extremely divergent from the normal distribution the latent vector was sampled from.

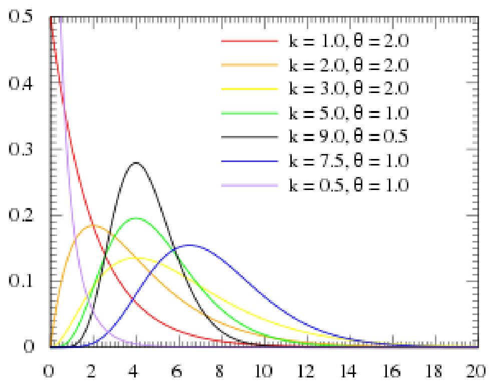


Figure 17: Gamma Distribution [4]

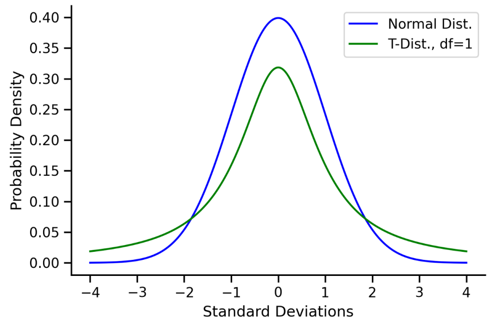


Figure 18: t Distribution [5]

We moved on from this to time series modeled from a GARCH process, testing multiple different parameter settings to elucidate the types of features that caused our GANs to fail. In particular, we created univariate and multivariate series that had small levels of or no autocorrelation, autocorrelation of volatility with differing levels of decay, leverage effect, trends in the returns, and differing levels of correlation between the distinct series in the multivariate cases. The largest multivariate dataset we created with a GARCH process had 32 time series. We will expand more deliberately on the performance of the relevant GANs on these different datasets with different parameters in the results section.

Table 1: GARCH Datasets

Dataset Name	**Composition**	**Length**
Autocorr. of Vol GARCH	1 GARCH series	40960 Days
Return Trend GARCH	1 GARCH series	40960 Days

3.2 Real Target Data

From here, we started testing our GANs on real financial time series. We always used the daily total returns rather than prices for all of these series. Our univariate series of choice was the history of the S&P 500 Index, and our initial multivariate dataset also included the Dow Jones Industrial Index, the Volatility Index (VIX), and the US Dollar - Japanese Yen currency pair. To expand to 8 series, we added the US Dollar - Euro currency pair, the NASDAQ Index, the MSCI World Index, and the Euro - Japanese Yen currency pair. Lastly, we expanded to 16 dimensions of real series by adding the US Dollar - Great British Pounds pair, GBP - EUR pair, US 10 year treasury bond, US 2 year treasury bond, United Kingdom 10 year bond, Japan 10 year bond, German 10 Year bond, and German 2 year bond. Having a broad mix of asset classes and geographic locales created a complicated set of relationships between the different series to provide a real challenge to the GANs.

We utilized another real dataset in order to test the efficacy of our generated data on a downstream task. The task we chose to test was the training of a statisical arbitrage model predicated on a simple neural network. This same model had shown strong results out of sample when trained on real data numbering in the thousands of assets, and tested by predicting optimal portfolio weights on the same cross section of assets. This model needed a decomposed set of 3 series for each asset: the base returns, active returns, and residual returns. The base returns of an asset are just the normal returns on a day to day basis, the active returns are the component of the base returns attributable generally to the market (i.e. the monthly beta for the asset times the whole market returns), and the residuals are the part of the returns that are unique to each individual asset (i.e. the base returns minus the active returns). In order to generate a convincing facsimile of this data, we took the total return series of all the equities in the universe and clustered then into 8 different buckets using K-means. Then we randomly selected 8 random assets from these buckets (!! look up actual asset vals!!) and decomposed them into active and residual returns. We then passed the 8 active return series and 8 residual return series histories to the GAN to generate data for use in training and testing our statistical arbitrage model.

Table 2: Real Datasets

Dataset Name	**Composition**	**Length**
Standard and Poor's 500 Index	1 Equity Index	8925 Days
4 Real Multi-Asset	2 Equity Indices, Volatility Index, 1 Currency Pair	8192 Days
8 Real Multi-Asset	4 Equity Indices, VIX, 3 Currency Pairs	5888 Days
16 Real Multi-Asset	4 Equity Indices, VIX, 5 Currency Pairs, 6 Bonds	5888 Days
16 Decomposed Equities	8 Equities (Active Returns and Residual Returns)	4945 Days

3.3 Conditioning Data

We used multiple types of conditioning data beyond just the prior steps in the series to influence the generation from our networks. Firstly and most simply, we used a binary conditioning representing business cycles taken from the US National Bureau of Economic Research. The two binary options are expansion and contraction, with an expansion being defined at the period from the last trough of the series to the peak of the series, and a contraction is the opposite: from peak to trough. This data occurs at a monthly time frame, and tracks US economic activity. In particular it "is based on a range of monthly measures of aggregate real economic activity published by the federal statistical agencies. These include real personal income less transfers (PILT), nonfarm payroll employment, real personal consumption expenditures, wholesale-retail sales adjusted for price changes, employment as measured by the household survey, and industrial production. There is no fixed rule about what measures contribute information to the process or how they are weighted in our decisions."

We also used more fine grain and international macroeconomic markers to condition our networks. We took the short term interest rate (three month ICE LIBOR) of the US dollar, GB pound, Swiss franc, Japanese yen, and the Euro. Likewise, we used the term spread of these same currencies (price of the yield on 10 year bonds minus the short term interest rate). We included the US consumer comfort index which "measures Americans' perceptions on three important variables: the state of the economy, personal finances and whether it's a good time to buy needed goods or services." and the US Manufacturing Purchasing Managers Index (PMI) which measures the activity level of purchasing in the manufacturing sector. We also used the US default yield spread, and the 1 year difference in US money supply. Lastly, we took some data about the MS World Index to have a general picture of the stock market: the earnings to price ratio, the price to book ratio, the dividend yield, the net proceeds, and the net issue.

3.4 Data Pipelining and Preprocessing

Regardless of the type of data we passed through our network, we had a relatively consistent pipeline for processing and preparing the data for training. Firstly, any data being passed to the model was min-max scaled to values between 0 and 1. We then partitioned the data into discrete samples of the relevant generation window size from a continuous time series or set of time series. There were two options in the pipeline at this step: whether to use overlapping samples or not. In some datasets and with some problems, such as when trying to concurrently generate conditions with a monthly sampling rate, it made more sense to use non overlapping samples as the input and target data, whose generation window size corresponded to the sample rate (i.e. 2 months of data starting on the day of first sample for the macroeconomic conditions). However, in most cases, we used overlapping samples of data as our target to expand our dataset as much as possible. In this case, we create a new generation window starting at each day in our series. s

4 Modeling

4.1 General Model Architecture

As discussed in the introduction, our approach is predicated on a generative adversarial network structure and most of our modeling research focused on improving the training of the two networks or modifying the network of the generator, discriminator, or both. Each of the models was designed to be input size agnostic; i.e. the outputs of the models / size of some layers scale automatically to match the target data shape. The tables detailing the layers of each model below all correspond to an generation window size of 128, a latent vector size of 32, and a single input series unless otherwise specified. It is important to note that in multivariate examples, each series had a corresponding latent vector; so for a set of N series, the latent vector would be of size(32, N). We also included a network size flag in our implementation to set the interior size of the network independently of the target size, unless otherwise specified this variable was set to 512 in each of the network size descriptions.

4.2 Training

The training loop we utilized is predicated on a balanced update rule, with the same number of training steps per epoch for the generator and discriminator. In the discriminator update step, the discriminator sees a set of real samples and a set of fake samples from the generator. The loss is calculated by summing the binary cross entropy of the real predictions relative to a set of targets of only 1, and the BCE from the generated samples relative to a set of targets of only 0, and dividing it by two. In the generator update step, the discriminator sees a set of generated samples and the loss that is propagated at that step is just the BCE of the predictions on the generated samples relative to a set of targets of only 1 (i.e. the inverse of the discriminator step). In general, we tended to utilize imbalanced learning rates in favor of the generator but this was a decision made on a model by model basis. We always added dropout to our networks, usually with a balance favoring the generator as well but likewise dependant on the architecture.

4.3 Stabilizer

As part of our exploration into stabilizing the training of GANs, we built a learning rate stabilizer to manage the training of our networks rather than setting a learning rate schedule a priori. The stabilizer takes a measurement of the loss values for the generator and the discriminator every ten epochs, and if they are imbalanced in favor of either network, adds a tally in favor of the losing network. If the tally of either network makes it to fifteen (i.e. 150 epochs of imbalanced training), the learning rate of the winning network is penalized by 0.00002 and the the learning rate of the losing network is boosted by 0.00002. Though this generally did not lead to different training results than manual monitoring of the the training of the network, it did prevent degenerate training where one network overpowered the other, and made it easier to train over the long term without closely monitoring the models.

4.4 Deep Convolutional GAN

The first network we implemented was a simple deep convolutional GAN, with an imbalanced archtiecture between the generator and discriminator. The generator had a single fully connected layer as the first layer, one 1d convolutional layer followed by two 1d transpose convolutional layers, and another fully connected final layer before a sigmoid activation function for the output (this was consistent across all networks because the target values were min-max scaled). The generator had batch normalization and dropout after the linear, convolutional, and first transpose convolutional layers. The nonlinearities were leaky ReLUs across the board. The discriminator had 2 1d convolutional layers followed by one linear layer and a sigmoid activation (because it was a binary classifier). There was dropout and batch normalization after each of the convolutional layers.

Table 3: DCGAN Generator

Layer Name	**Size Details**	**Kernel**	**Stride**	**Padding**	**Bias**
FC 1	Input: 96, Output: 16384	N/A	N/A	N/A	True
Conv 1	In Channels: 512, Out Channels: 256	2	2	0	False
Transpose Conv 1	In Channels: 256, Out Channels: 128	2	1	0	False
Transpose Conv 2	In Channels: 128, Out Channels: 1	2	1	1	False
FC 2	Input: 16, Output: 64	N/A	N/A	N/A	True

Table 4: DCGAN Discriminator

Layer Name	**Size Details**	**Kernel**	**Stride**	**Padding**	**Bias**
1d Conv 1	In Channels: 1, Out Channels: 256	4	2	1	False
1d Conv 2	In Channels: 256, Out Channels: 512	2	1	0	False
FC 1	Input: 16384, Output: 1	N/A	N/A	N/A	True

4.5 Temporal Convolutional Network

The implementation of a TCN GAN (roughly analogous to QuantGAN) we made was significantly different than the DCGAN. Before we dive into the specifics of this model, it is pertinent to discuss the general structure of a temporal convolutional network.

TCNs rely on dilated convolutions to carry information forward in time and capture long range dependencies in sequential data. The dilation size is small at first, so that the convolutional filters focus on local relationships, but they become larger and larger with each with each successive layer. The greater the dilation size, the greater the step in between the variables which are passed into each convolutional filter.

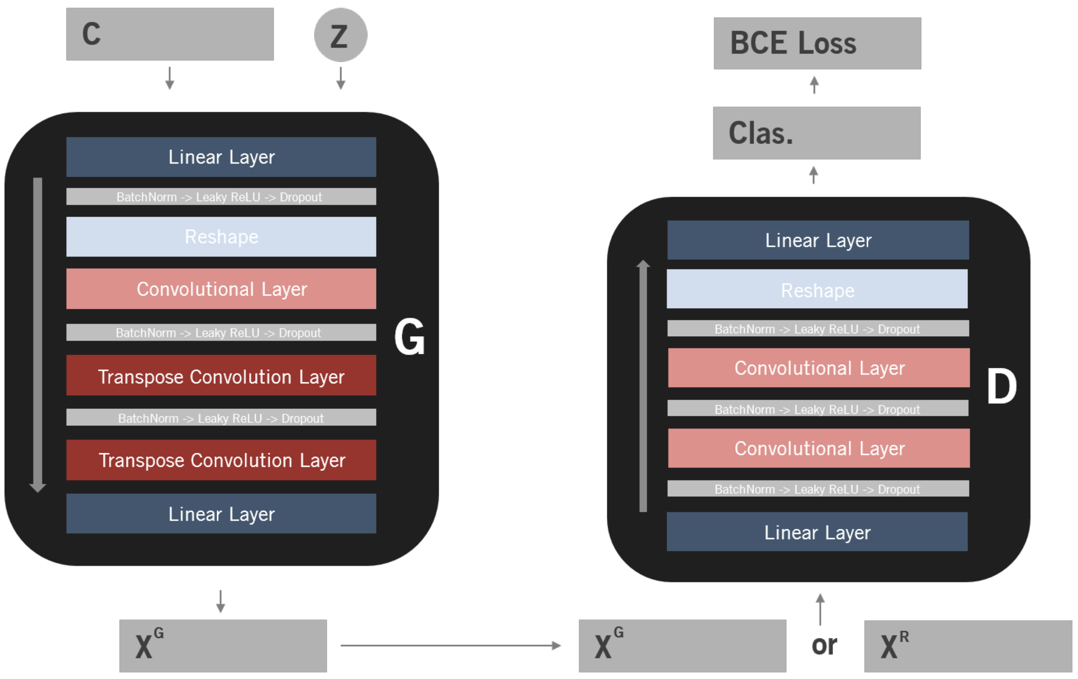

C
Z
Linear Layer
BatchNorm -> Leaky ReLU -> Dropout
Reshape
Convolutional Layer
BatchNorm -> Leaky ReLU -> Dropout
Transpose Convolution Layer
BatchNorm -> Leaky ReLU -> Dropout
Transpose Convolution Layer
Linear Layer
G
BCE Loss
Clas.
Linear Layer
Reshape
BatchNorm -> Leaky ReLU -> Dropout
Convolutional Layer
BatchNorm -> Leaky ReLU -> Dropout
Convolutional Layer
BatchNorm -> Leaky ReLU -> Dropout
Linear Layer
D
X^G
X^G
or
X^R

Figure 19: DCGAN Architecture

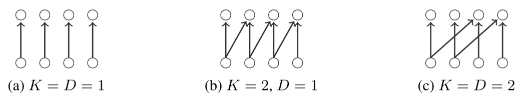


Figure 20: Demonstration of dilation [14]

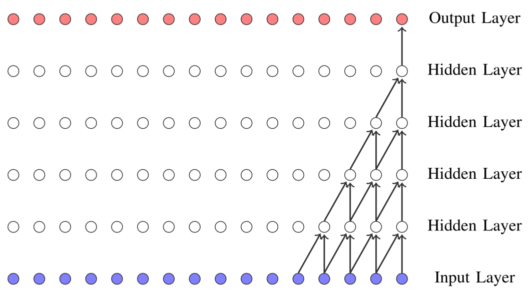


Figure 21: Vanilla TCN with D = 1 [14]

Our implementation of a TCN based GAN is slightly different than the one employed in the QuantGAN paper. The interior of their generator was also based on TCNs, and they also tested a pure TCN based network, but the final architecture they used was named a stochastic volatility network. It was composed of two separate TCNs, one of which generated the drift and volatility terms at each time step for the target series, and another which generated just the innovation term. In conjunction, these 3 terms made it possible for them to create a time series from their SVNN generator. The discriminator was a standard TCN binary classifier. The QuantGAN

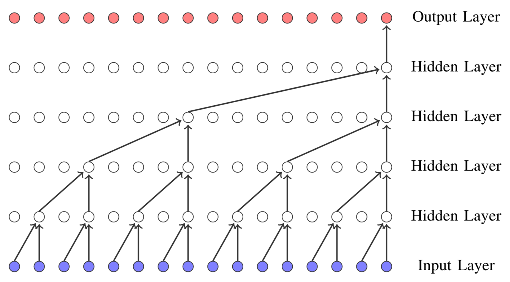


Figure 22: Vanilla TCN with D = 2 [14]

architecture was successful in modeling the heavy tails of the distribution, the lack of autocorrelation in returns, and the volatility clusters up to a certain memory threshold.

In our implementation, we used a slightly larger single generator that went directly from the latent space to the return series, rather than the SVNN. The latent dimension for this model is of a different shape than the others. It was the total length of the generation window size (in this case 128) with a width of 4. In other words, there were 4 latent variables for each time step in the sequence. The backbone of the architecture was a temporal block, each of which contain a stack of 2 sets of multiple components: a 1d convolutional layer (with a kernel size of two, stride of 1, a variable dilation term, and padding of 1) with weight normalization, followed by a 'chomp' layer which essentially removes the padding from the convolution, a parametric ReLU activation function, and dropout. The whole network is made up of a stack of these temporal blocks (in this case 6 for both networks) with the dilation of each block starting at two and increasing by a power of 2 with each successive block. Lastly, there is a single linear layer and a sigmoid activation for both the generator and discriminator.

We were able to achieve strong results in the univariate case with this approach, but ultimately it failed when trying to expand to multiple series and was very unstable in training (i.e. likely for one network to defeat the other). As mentioned previously, it was also awkward / ineffective to introduce conditioning into this network because of the inherent sequential assumption.

4.6 Wasserstein GAN with Gradient Penalty

Wasserstein GANs are a special class of GANs who have a significantly different loss function than a regular GAN. While a regular GAN uses a discriminator (i.e. binary classifier) and binary cross entropy loss. The Wasserstein loss is based on the Wasserstein distance (also known as earth mover's distance) between the two distributions which come from the critic network. The critic network is usually not hugely different from a discriminator network in terms of architecture, but does not have an activation layer as the final part of the network (i.e. the critic network output is unbounded). The job of the critic is not to predict the individual class of each sample, but rather to have the distribution of the outputs relative to the generated samples be as far as possible by Wasserstein distance from the distribution of outputs relative to the real samples. Likewise, the generator is attempting to minimize the distance between the distributions.

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

Figure 23: Wasserstein Distance between distributions Pr and Pg.

This approach provides two main benefits: it forces the generator to consider the whole distribution and thus prevents mode collapse and it provides an informative loss value that converges in an intelligible manner (in contrast to typical GAN losses which do not provide much information about convergence or the quality of generated samples). Unfortunately, optimizing the Wasserstein distance across all lambdas is computationally intractable. Because of this, it is necessary to enforce 1 lipschitz continuity on the discriminator, so that the space of possible lambdas is constrained. In the first implementation of Wasserstein GANs, weight clipping was used to explicitly constrain the weights within the correct range. This approach was inelegant, and often resulted in vanishing or exploding gradients up to the max and min available values of the clipping. It also limits the complexity of the functions that the network can map. So a gradient penalty was proposed as a solution by Gulrajani et al (2017). The gradient penalty is calculated by taking a linear interpolation between the real and generated samples, and then passing these interpolated samples to the critic. Then you take the l2 norm of the gradients relative to these critic outputs, subtract one from them, and square these terms. Then the loss is augmented by the mean of this modified gradient norm, as a way to penalize the model learning weights that fall outside of the 1 lipschitz constraint. This process is captured in the loss term below:

$$\mathcal{L} = \mathop{\mathbb{E}}_{\tilde{\mathbf{x}}\sim\mathbb{P}_g}[f(\tilde{\mathbf{x}})] - \mathop{\mathbb{E}}_{\mathbf{x}\sim\mathbb{P}_r}[f(\mathbf{x})] + \lambda \mathop{\mathbb{E}}_{\hat{\mathbf{x}}\sim\mathbb{P}_{\hat{x}}}[(||\nabla_{\hat{\mathbf{x}}} f(\hat{\mathbf{x}})||_2 - 1)^2]$$

Figure 24: Critic loss function for a Wasserstein GAN with gradient penalty.

The architecture we implemented to test the efficacy was the same as the architecture for the standard DCGAN, with the exception of the output activation in the discriminator / critic as mentioned. Obviously, the loss function was different as described above. The main motivation in using a Wasserstein GAN is to prevent mode collapse, stabilize training, as well as ensure a more general and flexible training of the generator network. We ultimately opted to solve these issues using spectral normalization, which we will detail in the next section.

4.7 Spectral Normalization

Spectral normalization controls the lipschitz constant of both the generator and discriminator explicitly but in a much more effective and elegant way than simple weight clipping and a much more computationally efficient way than gradient penalty. Unlike other common normalization techniques used in deep learning, spectral normalization is applied to the weights of each layer rather than the outputs of layers (i.e. the hidden state). At its essence, spectral normalization is just replacing the weights (W) of a layer with W/ σ(W), where σ(W) is the largest singular value of W.

$$\underbrace{\|A\|}_{\text{norm}} = \max_{x\neq 0} \frac{\|Ax\|}{\|x\|}$$

Figure 25: Spectral Normalization of matrix A.

$$\begin{array}{ccccccc} \mathbf{A} & & \mathbf{U} & & \mathbf{S} & & \mathbf{V}^{\mathrm{T}} \\ \begin{pmatrix} x_{11} & x_{12} & x_{1n} \\ & \ddots & \\ x_{m1} & & x_{mn} \end{pmatrix} & = & \begin{pmatrix} u_{11} & & u_{m1} \\ & \ddots & \\ u_{1m} & & u_{mm} \end{pmatrix} & & \begin{pmatrix} \sigma_1 & & 0 \\ & \ddots \; \sigma_r \; \ddots & \\ 0 & & 0 \end{pmatrix} & & \begin{pmatrix} v_{11} & & v_{1n} \\ & \ddots & \\ v_{n1} & & v_{nn} \end{pmatrix} \\ m \times n & & m \times m & & m \times n & & n \times n \end{array}$$

Figure 26: Singular Value Decomposition of matrix A.

In practice, utilizing spectral normalization is enough to prevent mode collapse and encourage the network weights to maintain a healthy distribution that allows the generator and discriminator to learn higher quality and more general representations of the target dataset. Both because of its demonstrated efficacy in other GAN implementations and our success using it, we chose to progress with a regular discriminator / BCE loss in conjunction with spectral normalized weights rather than using wasserstein loss with gradient penalty.

4.8 Self Attention GAN

Self-attention GANs were introduced in 2018 by Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena for use in image generation techniques [19]. Self-attention modules have gained a lot of hype in recent years due to their incorporation in large scale language and vision modules, and their flexibility at learning representations that occur non-adjacently in the data. In particular, their success in modeling long range dependencies in language models seems to make them an obvious choice to boost performance of our GANs which are focused sequences. In addition, self attention has been shown to be useful for time series generative modeling in general in the PSA-GAN paper [15] and for financial time series in particular in Temporal Fusion Transformers [12] and Generative Adversarial Networks in Finance [6], amongst others. Before diving into our own usage of self attention in this particular architecture, it is necessary to understand a basic self attention model and multihead self attention.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Figure 27: Empirical definition of the attention module.

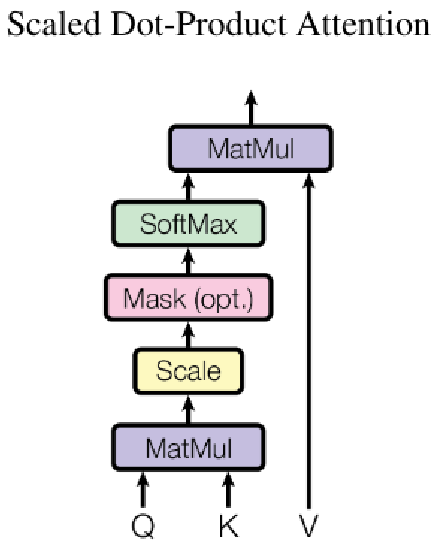


Figure 28: Self Attention Module Diagram [17]

A single head attention module involves passing your data into 3 separate sub-networks; the query, key, and value networks. The query output is matrix multiplied with the transpose of the key network output, the output of this term is usually called the ’energy.’ After taking the softmax of this matrix, it is called the ’attention.’ After matrix multiplying the ’attention’ matrix by the value network output, you receive what is generally called the ’attention map.’ In practice, this network structure is extremely flexible and is able to find relationships in the data that a regular feed forward or convolutional structure would not be able to. It is able to replicate convolutional maps where necessary and find relationships in adjacent data points, but also is particularly adept in finding filters that relate distant points.

Even more powerful than a single head self attention is multihead attention, which is simply multiple single head attention modules in parallel whose output is then summed and passed into a single linear layer to output one single output. In practice, this allows the network to learn multiple sets of attention maps i.e. one focused on local relationships, one on distant relationships in the same series, and one focused on relationships between different series in a multivariate problem. This approach is over-powered, and can be over-parameterized for univariate series so we generally used only single head attention when generating a single series. It is also important to note that both types of self-attention modules return outputs of the same shape

Table 5: Single Head Self Attention Module

Layer Name	Size Details	Kernel	Stride	Padding	Bias
Query 2d Conv.	In Channels: 512, Out Channels: 64	1	0	0	True
Key 2d Conv.	In Channels: 512, Out Channels: 64	1	0	0	True
Value 2d Conv.	In Channels: 512, Out Channels: 512	1	0	0	True

Table 6: Multi Head Self Attention Module

Layer Name	Size Details	Kernel	Stride	Padding	Bias
Query Linear	Input: 512, Output: 512	N/A	N/A	N/A	True
Key Linear	Input: 512, Output: 512	N/A	N/A	N/A	True
Value Linear	Input: 512, Output: 512	N/A	N/A	N/A	True
Output Linear	Input: 512, Output: 512	N/A	N/A	N/A	True

Table 7: SAGAN Generator

Layer Name	Size Details	Kernel	Stride	Padding	Bias
FC 1	Input: 96, Output: 16384	N/A	N/A	N/A	True
Conv 1	In Channels: 512, Out Channels: 256	2	2	0	False
Single Head Self Attn.	Input: 512, Output: 512	N/A	N/A	N/A	N/A
Single Head Self Attn.	Input: 512, Output: 512	N/A	N/A	N/A	N/A
Transpose Conv 1	In Channels: 256, Out Channels: 128	2	1	0	False
Transpose Conv 2	In Channels: 128, Out Channels: 1	2	1	1	False
FC 2	Input: 16, Output: 64	N/A	N/A	N/A	True

Table 8: SAGAN Discriminator

Layer Name	Size Details	Kernel	Stride	Padding	Bias
1d Conv 1	In Channels: 1, Out Channels: 256	4	2	1	False
Single Head Self Attn.	Input: 512, Output: 512	N/A	N/A	N/A	N/A
1d Conv 2	In Channels: 256, Out Channels: 512	2	1	0	False
FC 1	Input: 16384, Output: 1	N/A	N/A	N/A	True

as the input, essentially an in place transformation of the data. This makes it easy to incorporate residual connections (i.e. adding the input back into the output of the module) which are important for maintaining informational flow from the input to the later layers in particularly deep networks.

In our implementation, we added single head self attention modules with residual connections to both the generator and discriminator of the DCGAN structure we defined earlier. In the generator, we placed two modules after the first convolutional layer. In the discriminator, we placed one module also after the first convolutional layer. We determined that this interior placement was more effective than using the self attention module as the first layer (where its impact was largely washed out by later layers) or as the final layer (where it was too powerful in changing the output and would overfit to the target set). In addition, we spectral normalized all of the layers in the network. In our single head attention network, we changed the traditional structure of a self attention module to convolutional layers for the query, key, and value networks instead of linear layers.

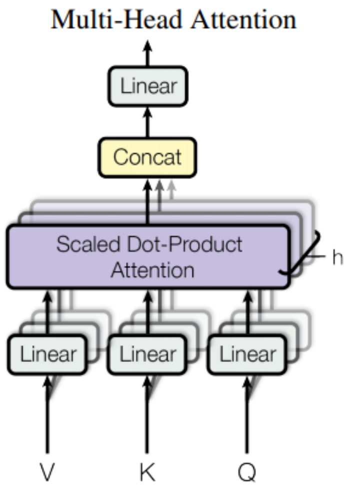


Figure 29: Multi-Head Self Attention [17]

4.9 Transformer Encoder GAN

The last architecture we implemented took inspiration from transformer architectures which have seen great success in many sequence modeling tasks, as well as NLP and image related tasks. We stopped short of implementing the self supervised training paradigm traditionally associated with transformers, and likewise did not use a codebook or vector quantization like the VQGAN-Transformer Model from Taming Transformers for High-Resolution Image Synthesis [7]. Our approach was simpler, we recreated a transformer block from a vanilla BERT, and then used a succession of these blocks for both the generator and discriminator. We tested multiple structures of blocks; some using convolutional self attention and some using linear layers for the query, key, and value, and also tried both convolutional and linear layers in the upscaling / downscaling subsection of the transformer block. We also tried various block arrangements: different depths, imbalanced depths where the discriminator had fewer blocks than the generator, and growing and shrinking block sizes (where the hidden size of the layers in each block started small (roughly equivalent to the input size), became larger by a power of two for every block until the central block, and then the blocks decreased in size until they were at the output size). Ultimately the architecture that had the best results was using linear layers in all the positions we mentioned, with 6 blocks for the generator and 4 for the discriminator. As with the prior model, we utilized spectral normalization for all the layers.

The transformer block architecture started with a multihead self-attention module followed by a linear layer with dropout. The residual from the input was added after this linear layer. This was followed by layer normalization. Then there was a group of two linear layers, the first upscaling the hidden value followed by gelu activation, and then another linear layer that downscaled back to the original size.

Table 9: Transformer Block

Layer Name	Size Details	Kernel	Stride	Padding	Bias
MultiHead SA	Input: 512, Output: 512	N/A	N/A	N/A	N/A
Projection Linear	Input: 512, Output: 512	N/A	N/A	N/A	True
Upscale Linear	Input: 512, Output: 2048	N/A	N/A	N/A	True
Downscale Linear	Input: 2048, Output: 512	N/A	N/A	N/A	True

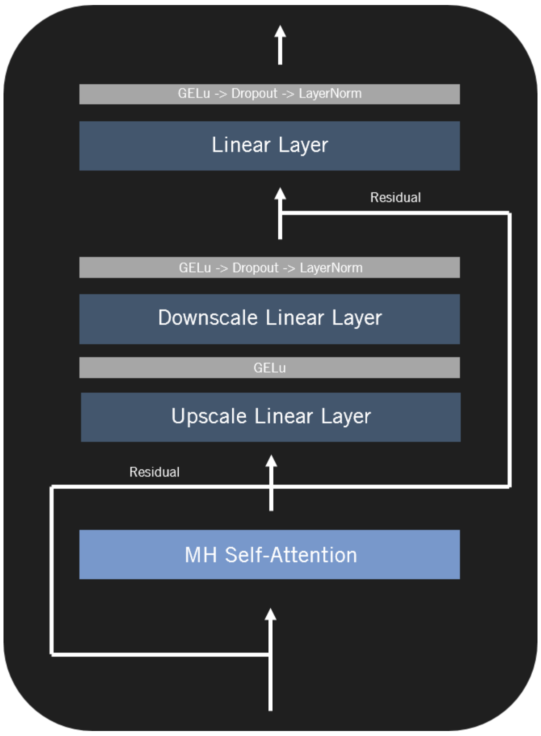


Figure 30: Transformer Encoder Block

This was followed by dropout and a residual connection from the value prior to the upscale-downscale group. With the growing / shrinking blocks architecture, there was another linear 'scaler' which changed the hidden size to the needed scale for the next block.

4.10 Generating Conditions and Series with the Transformer Encoder Architecture

When conditioning on outside information, we simply pass the conditions into the normal architectural structure of the network at an increased size. However, we also wanted to test our ability to generate conditions for completely synthetic long term market simulations. Initially we tried to generate the conditions from the same generator we used for series generation. We did this by simply expanding the generation window size of the generator and considering the trailing outputs as the conditions, reshaping them for the discriminator. We also tried adding a separate set of layers at the end of the transformer blocks which created the conditions in

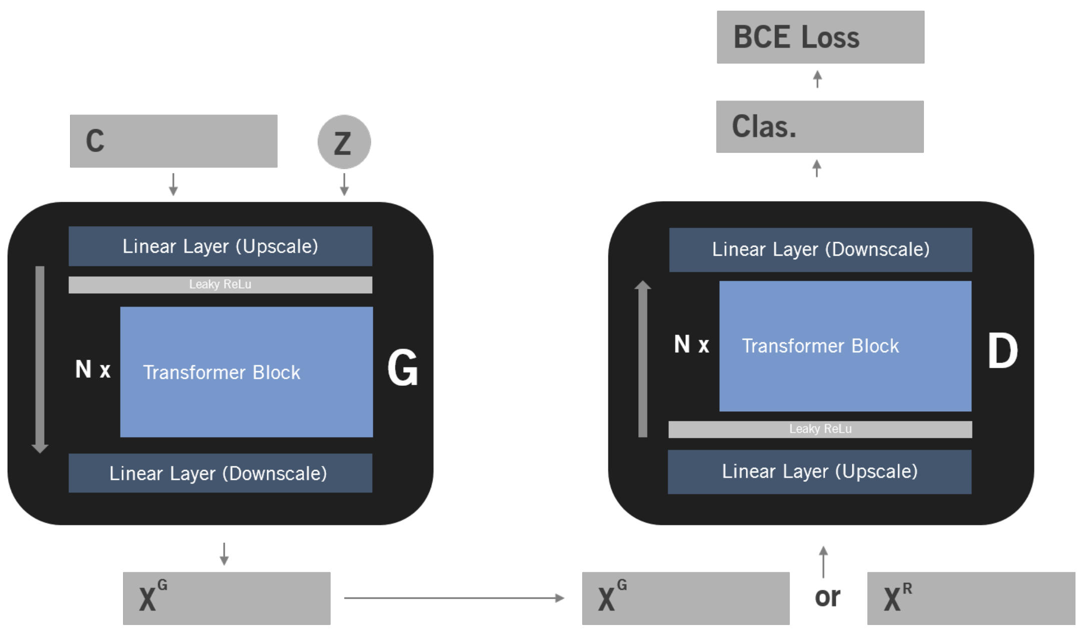


Figure 31: Transformer Encoder GAN

parallel with the last layers that created the series. This approach was moderately successful, but always impacted the series generation negatively because some of the capacity of the network was being reserved for information on the conditions that was potentially not relevant to the series themselves. So lastly we tried a totally separate GAN where the generator only made the conditions given the prior series and prior conditions, and the discriminator saw the real prior series and conditions, the real target series and the generated conditions relative to that data. This approach was far more successful in maintaining the quality of the generated series while still producing synthetic conditioning data as well. By default the conditioning GAN had the same architecture as the series GAN but with 1/8 of the hidden size (i.e. 64 in the case of a hidden size of 512 for the main network).

5 Results

We found that we were able to train our TCGAN and SAGAN to capture the stylized facts of univariate financial time series For multivariate series the SAGAN with good quality up to 4 assets and the TEGAN with good quality up to 16 assets. We demonstrated that the TEGAN model was able to generate multi-asset markets as well as just equities. Furthermore, we were able to train with macroeconomic conditions as inputs, and match by clustering the generated and real series relative to the macroeconomic conditions. Lastly, we showed that our data was of sufficient quality to increase the performance (measured by sharpe ratio) of a statistical arbitrage strategy by augmenting the real training data with our generated data.

5.1 Univariate Series

We tested series modeled from a GARCH model as test series with explicit characteristics in order to determine the strengths and weaknesses of particular models. In this section we will look at the DCGAN, TCGAN, the SAGAN, and the TEGAN. Though we tested multiple different GARCH series, the primary series we tested had a heavy tailed distribution, no autocorrelation in the returns and a decay of autocorrelation for volatility. Without conditioning, the DCGAN fails to capture autocorrelation of volatility where other models are able to accurately model it. With conditioning it is even more apparent that long term trends are not captured by the DCGAN.

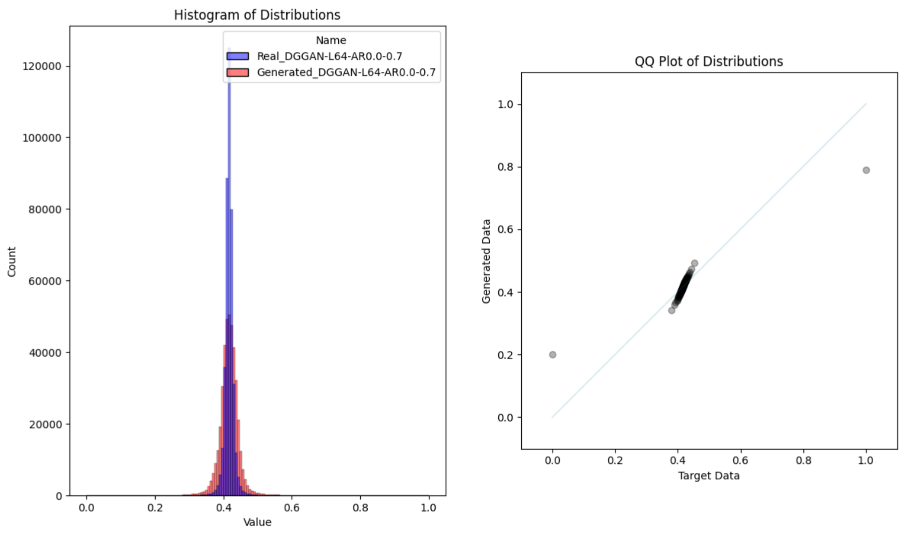


Figure 32: DCGAN GARCH Distribution

We can see from the above graphs that the SAGAN is the most effective for the generation of univariate series. They do much better capturing the temporal dynamics

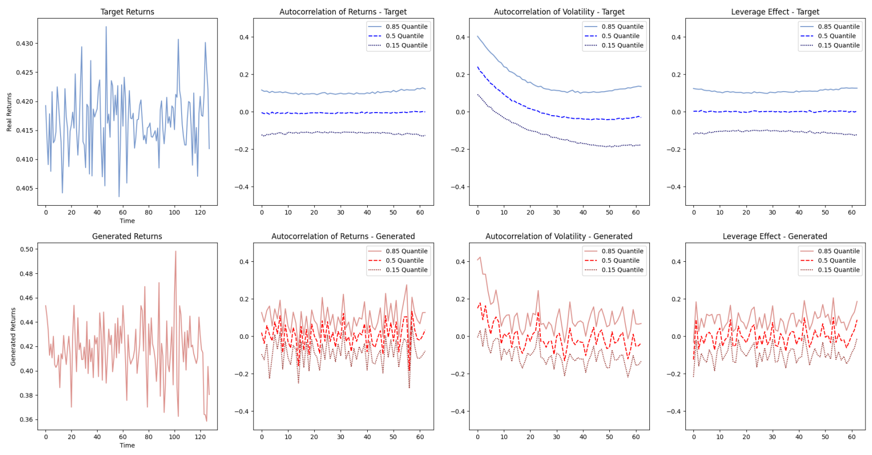


Figure 33: DCGAN GARCH Series

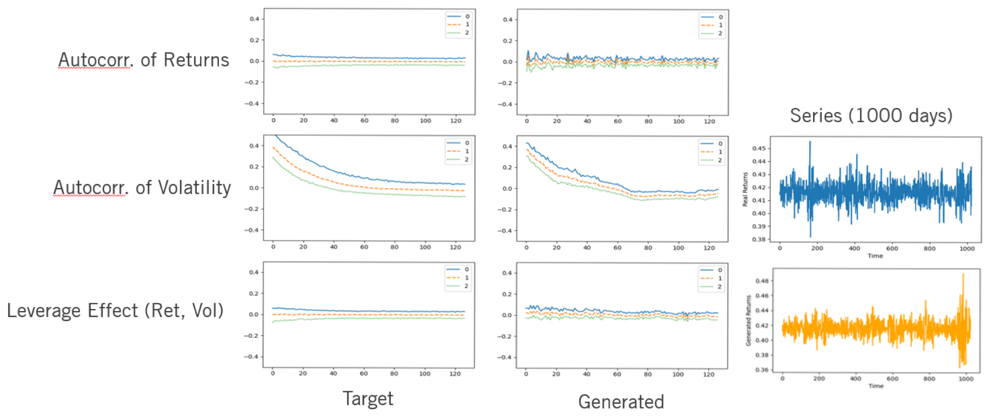


Figure 34: SAGAN GARCH

such as the autocorrelation of volatility or leverage effect. The TEGAN overfits to the target data, and is unable to generate sequences that have any variation in autocorrelation or autocorrelation of volatility - i.e it generates many series that are almost identical. Because much of the transformative power of the TEGAN comes when many blocks are stacked together but stacking many blocks essentially overparamaterizes for univariate series, it is not the best architecture for this task.

We also tested a GARCH process with trends to see if the model could learn from only situationally specific series (i.e. in this case a crash and recovery)

We see here again that the SAGAN is effective for univariate generation and most accurately recreates the v shaped trend we introduced into our GARCH process, especially evidenced by comparing the autocorrelation plots of the different samples

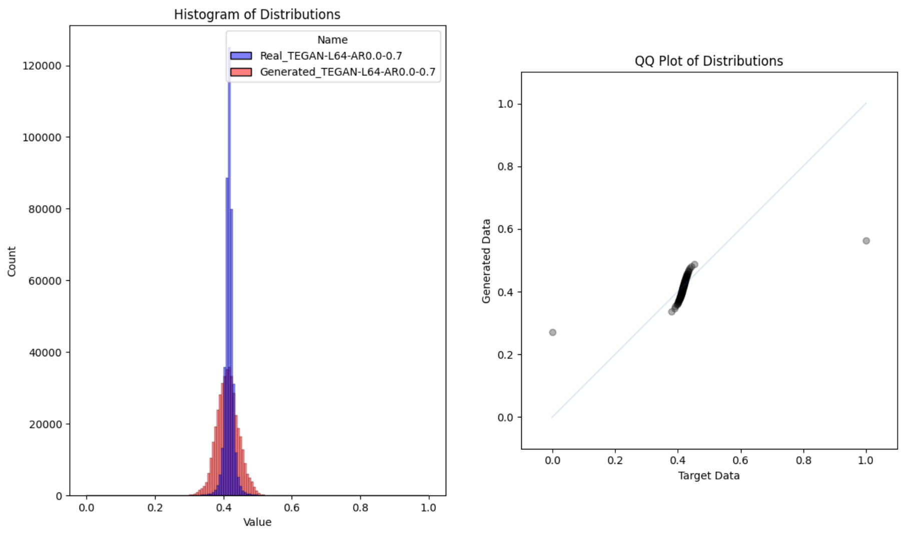


Figure 35: TEGAN GARCH Distribution

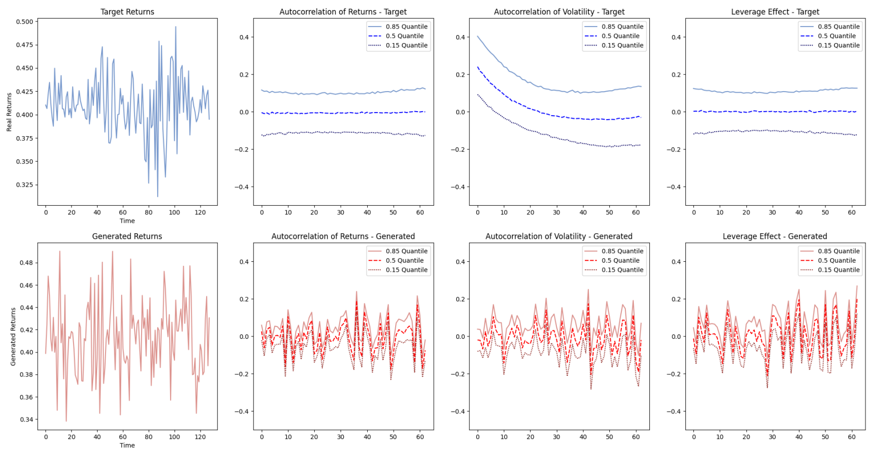


Figure 36: TEGAN GARCH Series

of generated data.

All of the prior examples have been comparisons of short series generated independently, rather than generated as a long sequence autoregressively. Likewise the training did not incorporate any training based on conditioning using the prior steps in the series. The following examples were trained and generated with a condition on prior steps in the series. Here we only compare our TCGAN and SAGAN models, as we determined they were the most effective at modeling univariate series.

Once again, we see the SAGAN outperforming the TCGAN, though the SAGAN

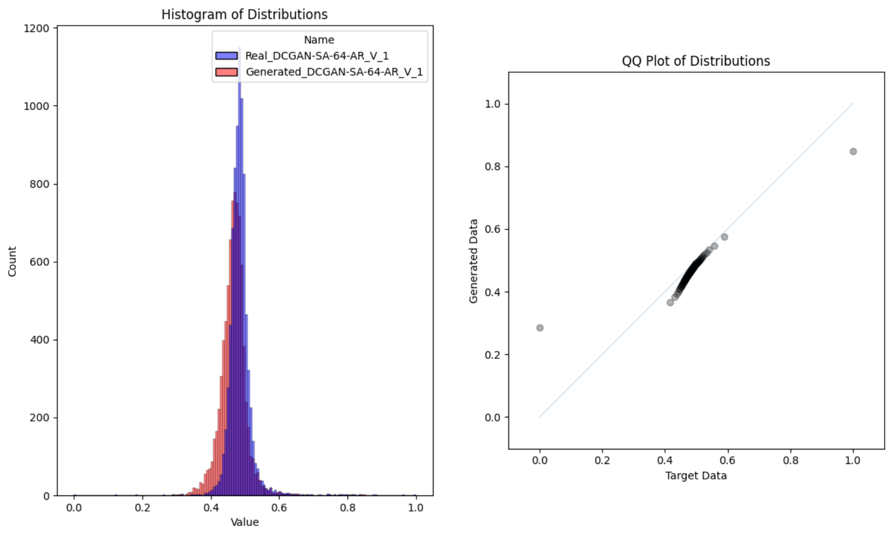


Figure 37: SAGAN GARCH V Distribution

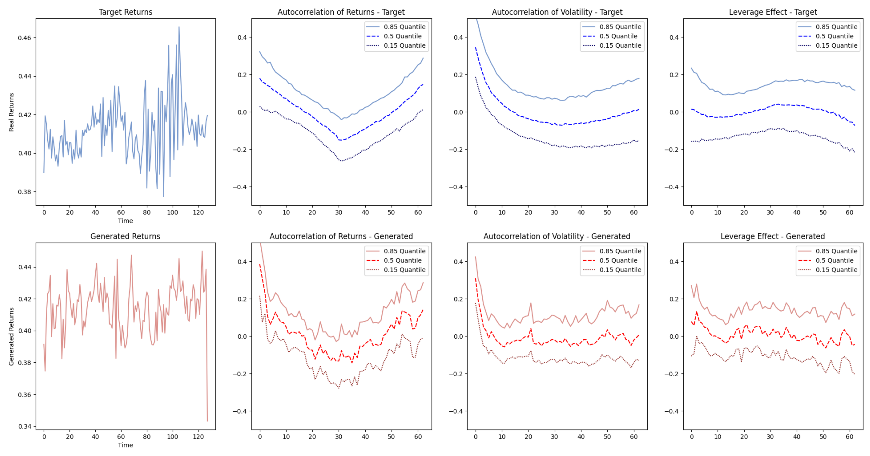


Figure 38: SAGAN GARCH V Series

does introduce a small 'kink' in the center of the autocorrelation function because it tends to generate similar series between generation window steps - i.e. the model generates a series that is more similar to its condition in an exact rather than statistical fashion. This effect is less pronounced with the regular GARCH series than with the trend GARCH, potentially because there is actually a natural spike in the autocorrelation of the series in the center of the time dimension. It is also less pronounced with shorter generated series, however, as the autoregressive generation goes through more and more steps, it tends to settle into a homogeneous series which creates higher autocorrelation between samples. We can see that the SAGAN

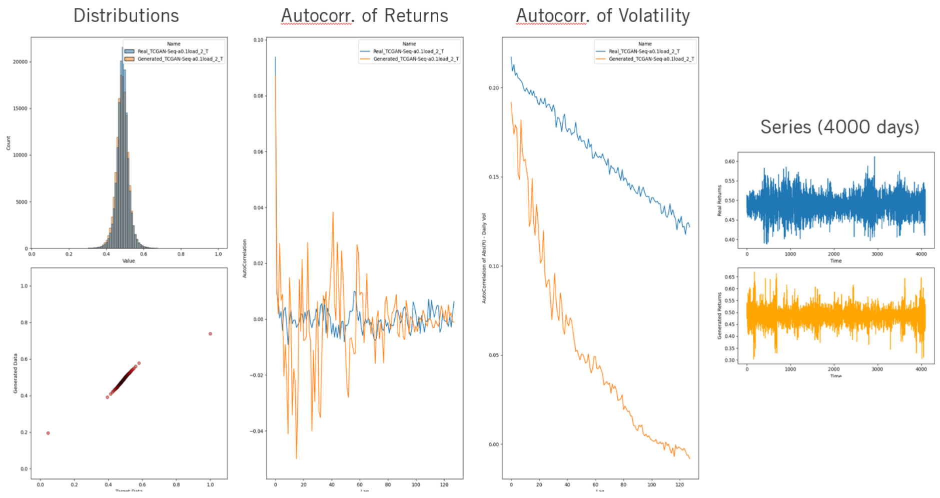


Figure 39: Autoregressive TCGAN GARCH

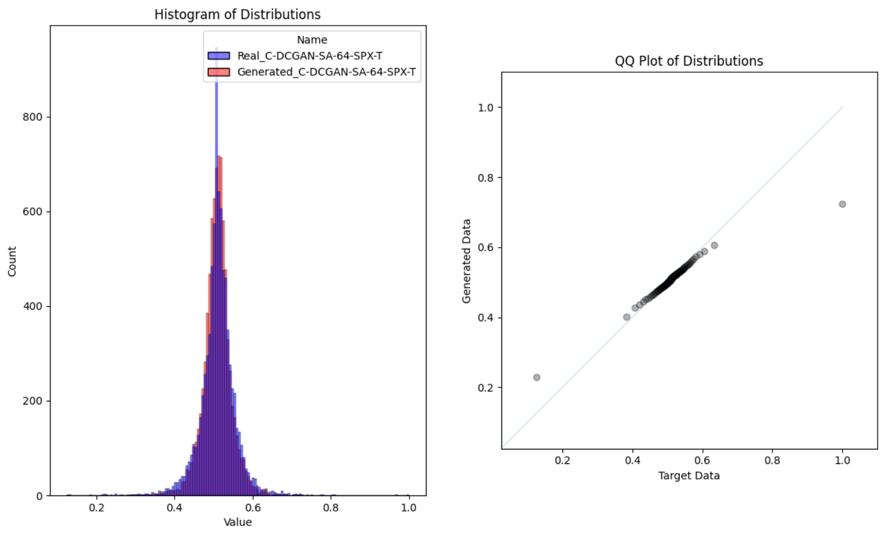


Figure 40: C-SAGAN GARCH V Distribution

model is able to capture the longer term autocorrelation of volatility that the TC-GAN cannot because the way the TCGAN series are generated precludes them from capturing any memory beyond the length of the generation window size.

When testing on the total returns of the S&P Index, the SAGAN outperforms:

5.2 Multivariate Series

When generating multivariate series, we first tested the 4 series multivariate dataset consisting of the S&P Index, the Dow-Jones Index, the Volatility index, and the USD-Yen currency pair. We found the SAGAN and TEGAN to be most effective

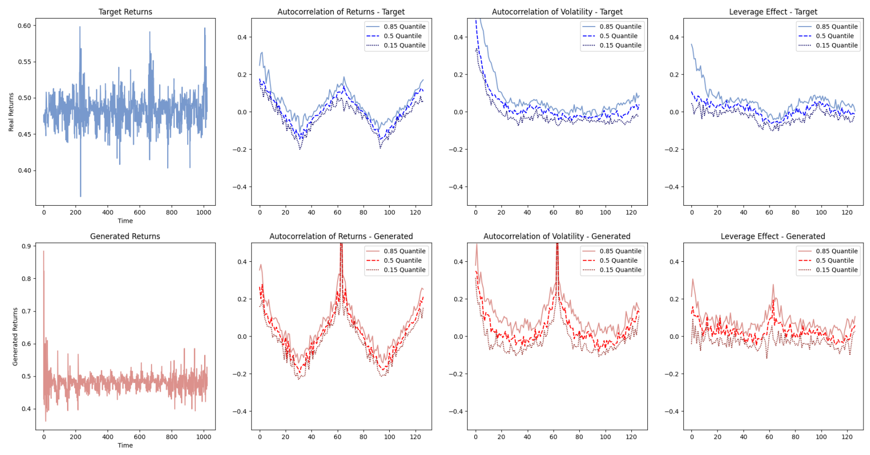


Figure 41: C-SAGAN GARCH V Series

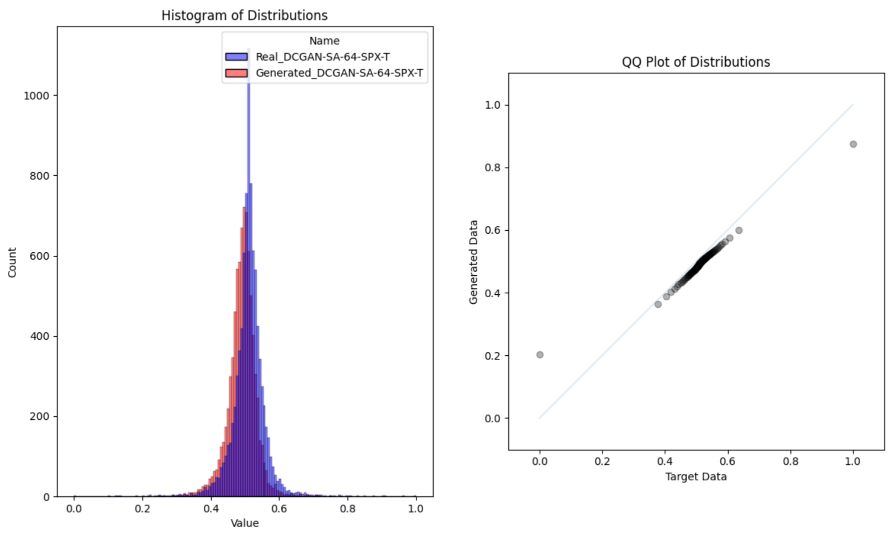


Figure 42: C-SAGAN S&P Distribution

for this task.

Both models were able to capture the inter-series correlation relatively well, will also generating series that had differing levels of various other stylized facts relative to each of the corresponding assets.

We continued our multivariate testing using 8 assets, and here we begin to see the higher capacity and flexibility of the TEGAN take prominence over the SAGAN. The quality of the individual series degraded slightly the more assets we add because there is some tradeoff between correlation for particular series and having highly

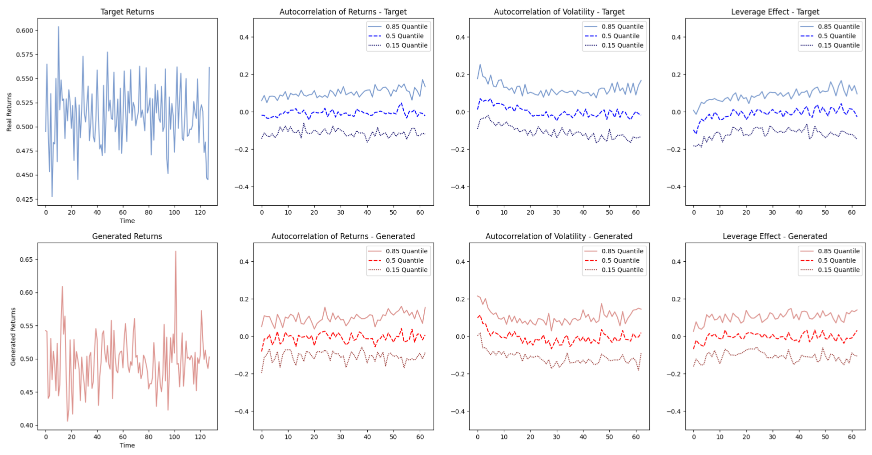


Figure 43: C-SAGAN S&P Series

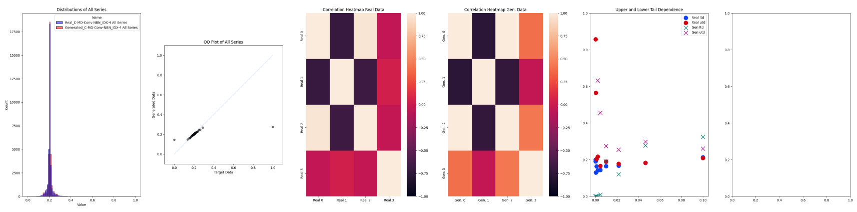


Figure 44: C-SAGAN 4 Assets

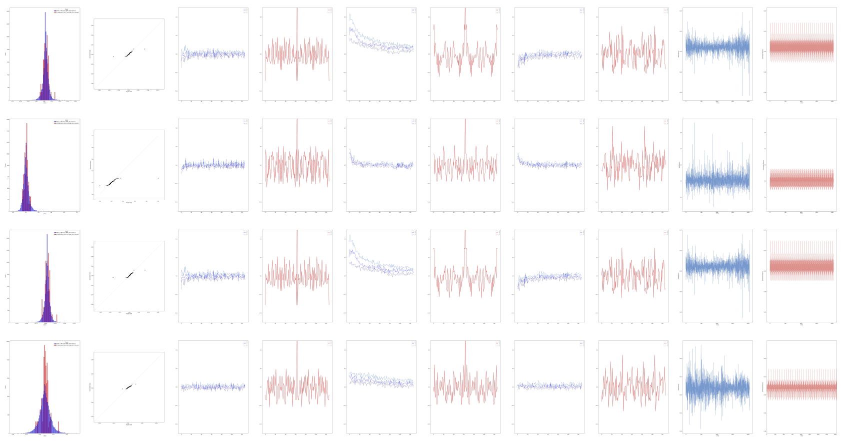


Figure 45: C-SAGAN 4 Assets

differentiated stylized facts.

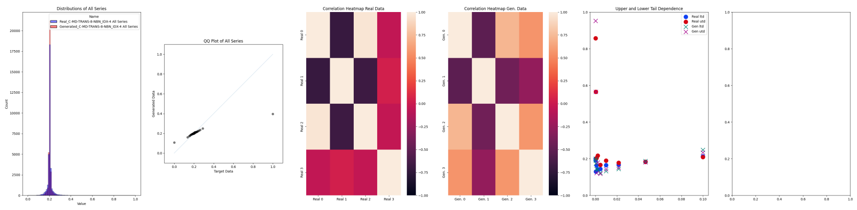


Figure 46: C-MTE-GAN 4 Assets Distribution

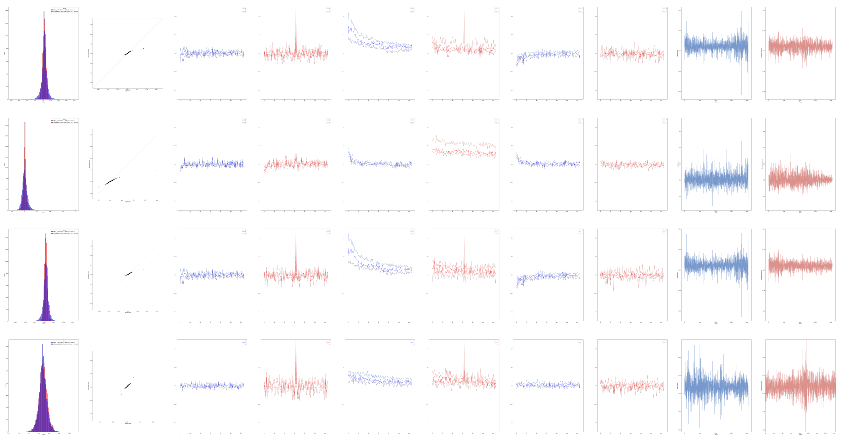


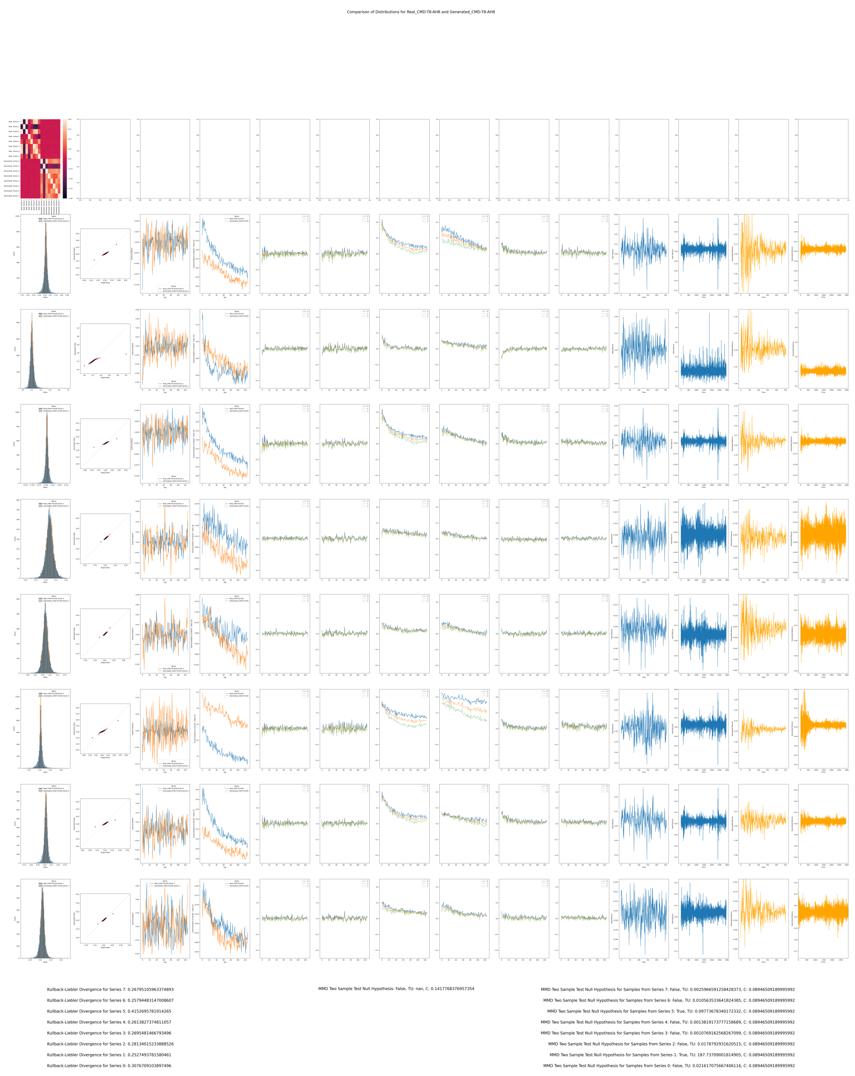
Figure 47: C-MTE-GAN 4 Assets Distribution

As we proceed to 16 series, the SAGAN begins to break down in its ability to learn independent series and devolves to generating the same series for asset, while the TEGAN is able to maintain its ability to generate multiple assets independently.

We also tested 32 and 64 series, but because the performance degraded for all models, we decided to exclude them from this report. Regardless, we demonstrated the ability to faithfully generate multi asset markets autoregressively in which the series maintain their stylized facts to a relatively high degree of fidelity.

5.3 Conditioning on Macroeconomic Features

Our next test of our GAN models was to try to condition the series on macroeconomic data, and from here on out we only utilized the TEGAN model because we determined that it was the most effective for multivariate generation. As mentioned in the data section, we used 20 total macroeconomic features for conditioning, with a total of 213 unique condition sets. Below are the summary statistics for the conditioning data prior to the min-max scaling we used to make them more palatable to the GAN:



```
Kullback-Liebler Divergence for Series 7: 0.26795105963374893
Kullback-Liebler Divergence for Series 6: 0.25794483147008607
Kullback-Liebler Divergence for Series 5: 0.4152695781914265
Kullback-Liebler Divergence for Series 4: 0.2613827374811057
Kullback-Liebler Divergence for Series 3: 0.2695481466793496
Kullback-Liebler Divergence for Series 2: 0.28134015233888526
Kullback-Liebler Divergence for Series 1: 0.2527493781580461
Kullback-Liebler Divergence for Series 0: 0.3076709103897496

MMD Two Sample Test Null Hypothesis: False, TU: nan, C: 0.1417768376957354

MMD Two Sample Test Null Hypothesis for Samples from Series 7: False, TU: 0.0025966591258428373, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 6: False, TU: 0.010563533641824385, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 5: True, TU: 0.09773678340172332, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 4: False, TU: 0.0013819173777158689, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 3: False, TU: 0.0010769162568267099, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 2: False, TU: 0.0178792931620515, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 1: True, TU: 187.73709001814905, C: 0.08946509189995992
MMD Two Sample Test Null Hypothesis for Samples from Series 0: False, TU: 0.021617075667406116, C: 0.08946509189995992
```

Figure 48: C-MTE-GAN 8

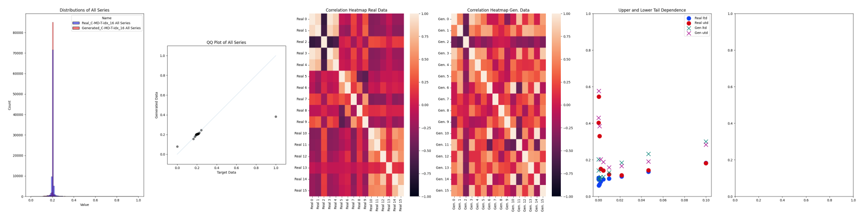


Figure 49: C-MTE-GAN 16 Series

	E2P_MSWLD	P2B_MSWLD	DY_MSWLD	NET_PROCEEDS_MSWLD
mean	4.927508	3.722322	2.366313	2.453906e+09
std	1.136927	1.122281	0.325488	2.277287e+09
min	1.453079	1.340361	1.811659	4.335296e+08
max	6.934146	6.605351	3.435784	1.051643e+10

	NET_ISSUE_MSWLD	CHF_SHORT_RATE	USD_SHORT_RATE
mean	-2.905748e+09	0.239369	1.643787
std	3.338342e+10	1.045565	1.640385
min	-1.311380e+11	-0.855400	0.188380
max	1.386030e+11	2.955000	5.621250

	GBP_SHRT_RT	JPY_SHRT_RT	EUR_SHRT_RT	CYCLE
mean	2.017983	0.134735	1.106463	0.906494
std	2.090709	0.243009	1.598845	0.291173
min	0.025500	-0.191000	-0.566290	0.000000
max	6.692500	0.996250	5.273750	1.000000

	CHF_TERM_SPREAD	USD_TERM_SPREAD	GBP_TERM_SPREAD
mean	0.851988	1.278643	0.769799
std	0.630768	1.181119	1.183459
min	-0.443000	-1.162250	-1.974250
max	2.551330	3.584370	3.506000

	JPY_TERM_SPREAD	EUR_TERM_SPREAD	USD_DEFAULT_YLD_SPRD
mean	0.677122	0.875790	107.271846
std	0.514426	0.823572	45.827057
min	-0.175860	-1.261750	49.120000
max	1.833750	2.728000	349.130000

	USD_CONS_CMFRT	USD_MAN_PMI	USD_MONEY_SPLY_DIFF
mean	40.877594	53.465633	716.438965
std	10.700742	4.696770	704.598870
min	23.000000	34.500000	157.300000
max	67.300000	61.400000	4157.200000

We wanted a flexible way to determine the efficacy of the conditioning, particularly how well the generator was able to generate series that were similar to the real series relative to each condition. The way we approached this comparison was by using k-means clustering to determine how similar the series were: we flattened the real and generated multivariate series into a single long array, and then clustered all of them together into 20 different clusters. Then we lined up the real samples and generated samples relative to a particular macroeconomic condition, and compared the clusters assignments of the real and generated samples. This was not a trivial problem, and it actually formed a decent approximation of generator convergence for training as it took thousands of epochs of training before any of the generated samples started to be assigned to matching clusters for their target.

In order to visualize these cluster matches, we take all the macroeconomic conditions and use t-SNE to reduce them into a 3 dimensional subspace. We then plot those 3 dimensions, with blue corresponding to an incorrect cluster match and orange to a correct cluster match. This makes it possible to see which subsets of conditions the model fails to recreate an accurate series in relation to.

We see here that with 16 assets, the model is able to capture the relevant statistical behaviour to a relatively strong degree as well as responsiveness to the macroeconomic conditions.

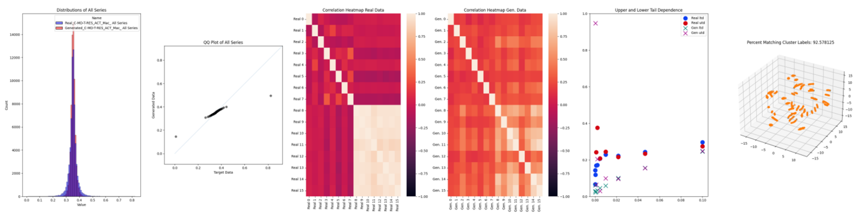


Figure 50: C-MTE-GAN 8 Active / 8 Residual Dsitribution + Clusters

This level of control of synthetic financial time series or accurate conditioning on macroeconomic conditions is unprecedented in the literature centered around the deep learning based generation of synthetic financial data. We demonstrate its usefulness in the next section regarding statistical arbitrage.

5.4 Statistical Arbitrage Data Augmentation

In order to determine the efficacy of generated data in a real world use case, we decided to augment the training data of a statistical arbitrage model to determine if we could improve its performance with our generated data. To this end, we took a universe of equities and clustered their returns using k-means into 8 buckets, then selected one equity from each cluster in order to capture an uncorrelated subset of the possible equity universe. When then decomposed this 8 equity returns into and active return component and a residual return component as described in the data section, and trained our C-MTE-GAN model on these 16 series.

The statistical arbitrage model we used was based on a relatively simple neural network, with two 1d convolutional layers, one self attention module (also based on 1d convolutional layers), and two linear layers. We trained an ensemble of 8 models and then used them as a voting classifier for making investing decisions. The models make independent predictions for each possible asset in the portfolio, which were then used as weights for the custom loss function. The loss function for the models was based on the sharpe ratio relative to the portfolio weights, so the models were optimized to construct optimal portfolios.

Layer Name	Size Details	Kernel	Stride	Padding	Bias
1d Conv 1	In Channels: 1, Out Channels: 8	2	1	0	true
1d Conv 2	In Channels: 8, Out Channels: 8	2	1	0	False
Single Head Self Attn.	Input: 8, Output: 8	N/A	N/A	N/A	N/A
Linear 1	Input: 8, Output: 16	N/A	N/A	N/A	True
Linear 2	Input: 16, Output: 1	N/A	N/A	N/A	True

The training of the model was designed to test the effect of increasing the quantity of available data for the model. We took training samples of 100 days, 200 days, 500 days, 1000 days, and 2000 days. We used a limited subset series for the training data, but a full investing universe of 1000 equities when evaluating the model's portfolio construction abilities after training.

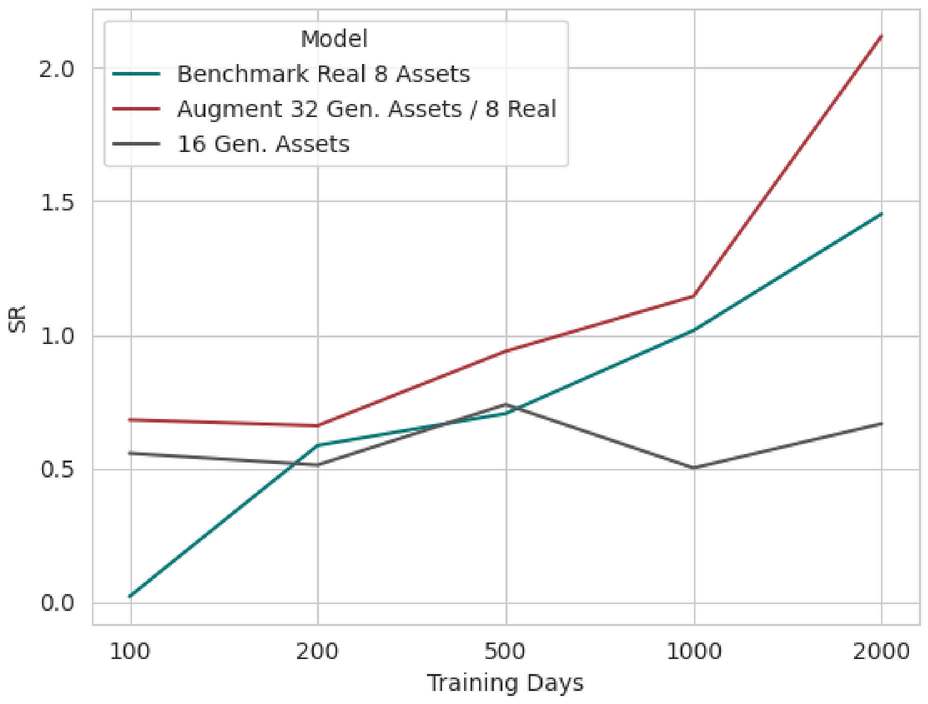


Figure 51: Performance by Sharpe Ratio of Augmented Statistical Arbitrage Strategy

We then tested a version of the statistical arbitrage ensemble where each of the sub networks was trained on the same 8 real assets and a different set of 8 series of synthetic returns (all taken from the same model). So each model saw a different set of synthetic series, and we hoped this would make their predictions more generalizable for the out of sample testing.

Lastly, we used our conditional generator to create long series based on real economic conditions. In general, the series were lower quality than the series which were generated normally, however we used the conditioning to generate some specific types of series. Firstly we generated an equal number of returns corresponding to either expanding or contracting business cycles, and used those to augment individual models in the ensemble. Secondly, we generated only samples corresponding to business cycle contractions to augment our training data with.

6 Conclusion

Through our research, we were able to explore the current state of synthetic data research in finance and create our own novel architectures that generated multi-asset markets conditionally on prior series and on macroeconomic data. We were able to capture stylized facts in univariate series, and in generating multiple series at once though to a lesser extent. Still we were able to generate relatively high fidelity series at a higher dimension than had been achieved in previous research. We were also able to accurately control the generation of series corresponding to macroeconomic conditions, and utilized k-means clustering to confirm the similarity of these conditionally generated series. We tested the efficacy of our models in augmenting the training of a statistical arbitrage model, where it had particular success in very low data availability environments. There are many more experiments and use cases that could extend from this research and the architectures we have developed. In conclusion, we have moved progress in the generation of synthetic financial time series forward.

References

[1] URL: https://en.wikipedia.org/wiki/Autoregressive_conditional_heteroskedasticity.

[2] URL: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73.

[3] URL: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73.

[4] URL: https://en.wikipedia.org/wiki/Gamma_distribution.

[5] URL: https://tjkyner.medium.com/the-normal-distribution-vs-students-t-distribution-322aa12ffd15.

[6] Florian Eckerli and Joerg Osterrieder. *Generative Adversarial Networks in finance: an overview.* 2021. DOI: 10.48550/ARXIV.2106.06364. URL: https://arxiv.org/abs/2106.06364.

[7] Patrick Esser, Robin Rombach, and Björn Ommer. *Taming Transformers for High-Resolution Image Synthesis.* 2020. DOI: 10.48550/ARXIV.2012.09841. URL: https://arxiv.org/abs/2012.09841.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016.

[9] Ian J. Goodfellow et al. *Generative Adversarial Networks.* 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[10] Tero Karras et al. *Progressive Growing of GANs for Improved Quality, Stability, and Variation.* 2017. DOI: 10.48550/ARXIV.1710.10196. URL: https://arxiv.org/abs/1710.10196.

[11] Edmond Lezmi et al. *Improving the Robustness of Trading Strategy Backtesting with Boltzmann Machines and Generative Adversarial Networks.* 2020. DOI: http://dx.doi.org/10.2139/ssrn.3645473.

[12] Bryan Lim et al. *Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting.* 2019. DOI: 10.48550/ARXIV.1912.09363. URL: https://arxiv.org/abs/1912.09363.

[13] Hao Ni et al. *Conditional Sig-Wasserstein GANs for Time Series Generation.* 2020. DOI: 10.48550/ARXIV.2006.05421. URL: https://arxiv.org/abs/2006.05421.

[14] Aaron van den Oord et al. *WaveNet: A Generative Model for Raw Audio.* 2016. DOI: 10.48550/ARXIV.1609.03499. URL: https://arxiv.org/abs/1609.03499.

[15] Jeha Paul et al. *PSA-GAN: Progressive Self Attention GANs for Synthetic Time Series.* 2022. arXiv: 2108.00981 [cs.LG].

[16] Kumiko Tanaka-Ishii Shuntaro Takahashi Yu Chen. *Modeling financial time-series with generative adversarial networks.* 2019. DOI: https://doi.org/10.1016/j.physa.2019.121261.

[17] Ashish Vaswani et al. *Attention Is All You Need.* 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[18] Magnus Wiese et al. “Quant GANs: deep generation of financial time series”. In: *Quantitative Finance* 20.9 (2020), pp. 1419–1440. DOI: 10.1080/14697688.2020.1730426. URL: https://doi.org/10.1080%2F14697688.2020.1730426.

[19] Han Zhang et al. *Self-Attention Generative Adversarial Networks*. 2018. DOI: 10.48550/ARXIV.1805.08318. URL: https://arxiv.org/abs/1805.08318.