Stress Testing Trading Strategies Using Conditional TimeGAN: Implementation of TimeGAN with Static Feature Integration for Market Regime Modelling

Course: CS787: Generative Artificial Intelligence

Team Members:

Nishvaan Sai H (200648)
Aman Kashyap (210108)
Jayant Malik (242110401)
Rohit Raj (210874)

December 15, 2025

Abstract

Algorithmic trading strategies are frequently validated against historical data, a practice that risks overfitting and provides a poor measure of resilience against rare, high-impact events. This project initially sought to address this by generating synthetic, stress-test scenarios using a conditional variant of the Time-series Generative Adversarial Network (TimeGAN). Later, we switch from using conditional TimeGAN to using normal TimeGAN with static feature integration, as theoretically described in the paper, to remain faithful to the original approach. However, the project encountered significant technical barriers: the reference TimeGAN implementation was based on the deprecated TensorFlow 1 framework, and more critically, it lacked the static feature integration required for conditional generation-a feature theoretically described in the original paper but absent from the public code. Consequently, the project's focus shifted to the non-trivial, foundational engineering task of modernising and architecturally completing the TimeGAN model. This report details the successful migration of the TimeGAN framework to TensorFlow 2, utilising an idiomatic, Keras-based architecture. We subsequently designed and implemented the paper's full static feature integration, applying it to condition the generative process on discrete market regimes ('Normal', 'Crisis' and 'Volatile') derived from yfinance market data. The modernised model was successfully trained, and a workflow was established to generate synthetic, regime-specific financial data. A rigorous, model-centric evaluation (including discriminative and predictive scores) and qualitative analysis (using t-SNE) were performed on the separated, regime-specific datasets. The model-centric evaluation results indicate that, although the model was able to capture the temporal dynamics moderately, it failed to accurately capture the probability distribution of the original dataset, a finding attributed to the hyperparameter tuning of the GAN and the supervisor loss function, as well as significant resource constraints. While the final application of backtesting was not completed, this work successfully produced a modern, architecturally complete TimeGAN, establishing a necessary and robust foundation for future research in regime-based synthetic data generation and financial stress testing.

Contents

1 Introduction

1.1 Problem Statement

The validation of quantitative trading strategies is fundamentally constrained by the finite nature of historical data. Algorithmic models are typically backtested against past events, a process that inherently risks overfitting to specific historical patterns and provides a false sense of security regarding future performance. This limitation is most acute when considering rare, systemic events such as market crashes or high-volatility ”black swan” scenarios. These events are, by definition, poorly represented in training data, yet they pose the greatest existential risk to an automated strategy.

Generative models, specifically Generative Adversarial Networks (GANs), offer a compelling solution to the data scarcity problem. By learning the underlying data-generating process of a financial time-series, a GAN can produce a theoretically infinite set of novel, plausible ”alternative histories”. A model capable of generating data conditioned on specific market characteristics (e.g., ’crisis’) would be a powerful tool for robust financial stress testing.

To achieve this, the Time-series Generative Adversarial Network (TimeGAN) was selected as a state-of-the-art framework. TimeGAN is designed explicitly to preserve the complex temporal dynamics and stepwise conditional distributions of sequential data. However, the pursuit of this application-centric goal revealed a critical, prerequisite technical problem. The public-facing reference implementation of TimeGAN presented two significant barriers:

1. **Technical Obsolescence:** The model was implemented in TensorFlow 1 (TF1), a framework that is now superseded by TensorFlow 2 (TF2). The codebase relies on deprecated libraries, session-based execution paradigms, and dependency conflicts that make modern development, extension, and maintenance intractable.
2. **Architectural Incompleteness:** The project’s goal required a conditional model. The original TimeGAN paper theoretically defines an architecture that accommodates both static features (e.g., labels) and temporal features. However, the public implementation omits this static-feature capability, supporting only the generation of unconditioned temporal data.

Therefore, the project was confronted with a dual problem: the high-level application problem of financial stress testing, and the immediate technical problem of a nonexistent, or at least non-functional, foundational tool.

1.2 Project Objective and Revised Scope

The project was initiated with the goal of developing a ”Conditional TimeGAN (cTimeGAN)” for stress testing trading strategies, as outlined in the project proposal. This would involve labelling historical data into market regimes and training the model to generate new sequences specific to those labels.

However, the technical limitations of the base TimeGAN implementation (as described in Section 1.1) necessitated a significant pivot. The original proposal’s ”cTimeGAN” was discovered to be, in fact, the full TimeGAN architecture as theoretically described in the 2019 NeurIPS paper. The paper’s formal problem formulation (Section 3) and proposed model (Section 4) explicitly define a system for jointly modelling static data (S) and temporal data ($X_{1:T}$). The team discovered that the public reference code had only implemented the temporal ($X_{1:T}$) portion.

Consequently, the project's scope was revised to focus on the substantial, foundational engineering work required to build this complete model. The primary objective shifted from applying a tool to constructing the tool itself. The revised objectives were as follows:

1. To execute a complete and robust migration of the TimeGAN codebase from TensorFlow 1 to an idiomatic, Eager-execution-based TensorFlow 2 framework.
2. To architect and implement the static feature generation capability (S) as theoretically described in the TimeGAN paper, integrating it with the temporal components ($X_{1:T}$).
3. To develop a data-processing pipeline using yfinance to source stock data and to engineer and label discrete ”market regime” categories as the static feature S.
4. To train this new ”Static-Feature-Integrated TimeGAN” (SF-TimeGAN) and conduct a rigorous, model-centric evaluation (as defined by the TimeGAN paper) to assess its ability to generate high-fidelity, regime-specific data.

1.3 Achieved Work and Report Outline

This project successfully executed the foundational engineering objectives. A full migration to TensorFlow 2 was completed, and the static feature architecture was successfully implemented. A data pipeline was built to source and label five yfinance tickers with three distinct market regimes. The training and generation workflows were separated, and the model was trained to produce synthetic data with its regime-specific characteristics.

A model-centric evaluation was performed, separating the original and generated data by their static regime label to analyse performance on a granular level. The quantitative and qualitative results, however, were not as expected, indicating that the model struggled to produce realistic results, despite capturing the temporal dynamics of the data moderately. This is attributed to the inherent difficulty of loss function hyperparameter tuning and significant resource constraints that prevented exhaustive fine-tuning. The final, application-centric goal of backtesting trading strategies was not completed due to the extensive time required for this prerequisite engineering work.

This report documents these findings in full. Section 2 provides an academic background on the TimeGAN architecture. Section 3 details the methodology, including the data pipeline and the novel TF2/static-feature model implementation. Section 4 presents the experimental setup and the quantitative and qualitative results. Section 5 provides a transparent discussion of the project's limitations, the challenges in model convergence. Section 6 concludes with a summary of the project's contributions.

2 Background: Time-series Generative Adversarial Networks (TimeGAN)

TimeGAN paper link: https://proceedings.neurips.cc/paper_files/paper/2019/file/c9efe5f26cd17ba6216bbe2a7d26d490-Paper.pdf

TimeGAN paper public implementation by the author: https://github.com/jsyoon0823/TimeGAN

The Time-series Generative Adversarial Network (TimeGAN), proposed by Yoon et al. (2019), is a framework designed to generate realistic sequential data. Its primary innovation is the introduction of a learning environment that jointly optimises the objectives of a Generative Adversarial Network (GAN) and an autoencoder, explicitly incorporating the supervised, autoregressive nature of time-series data. This novel combination is intended to preserve the complex, stepwise temporal dynamics that standard GANs often fail to capture.

The TimeGAN architecture consists of four key network components:

1. **Embedding Network (e):** An autoregressive network (e.g., GRU or LSTM) that maps a sequence of real temporal features $x_{1:T}$ into a latent representation $h_{1:T}$
2. **Recovery Network (r):** A network that reconstructs the original feature sequence $\tilde{x}_{1:T}$ from the latent representation $h_{1:T}$. These first two components form a standard autoencoder.
3. **Sequence Generator (g):** An autoregressive network that generates a synthetic latent sequence $\hat{h}_{1:T}$ from a sequence of random vectors $z_{1:T}$.
4. **Sequence Discriminator (d):** A network (typically bidirectional) that attempts to distinguish between the real latent sequences $h_{1:T}$ (from the Embedder) and the synthetic latent sequences $\hat{h}_{1:T}$ (from the Generator).

The key to TimeGAN's efficacy is that the generative and discriminative processes occur in the latent space defined by the autoencoder, not in the (often high-dimensional) feature space. This is coordinated by a three-loss system:

1. **Reconstruction Loss ($\mathcal{L}_R$):** A standard supervised loss (e.g., L2 norm) between the original data and the reconstructed data $\tilde{x}$ This loss trains the Embedder and Recovery networks, forcing the latent space h to be an effective representation of x.
2. **Unsupervised Loss ($\mathcal{L}_U$):** The standard adversarial minimax loss. The Generator (g) minimises this loss (by fooling the Discriminator), while the Discriminator (d) maximises it (by correctly identifying fakes). This loss is applied in the latent space and ensures the distribution of synthetic latent sequences $\hat{h}$ matches the distribution of real latent sequences h.
3. **Supervised Loss ($\mathcal{L}_S$):** The most critical innovation. The Generator (g) is also trained in a "teacher-forcing" style, where it receives the real latent sequence $h_{1:t-1}$ and attempts to predict the real next h_t. This loss (an L2 norm in the latent space) explicitly forces the generator to learn the stepwise conditional distributions $p(x_t|x_{1:t-1})$ of the real data, which is essential for capturing temporal dynamics like momentum and volatility clustering.

Most importantly for this project, the TimeGAN paper's formal problem formulation (Section 3) is explicitly for data tuples of the form $(S, X_{1:T})$, where S is a vector space of static features and $X_{1:T}$ is a space of temporal features. The proposed model architecture (Section 4) defines static counterparts for each component: a static embedder $h_s = e_s(s)$, a static generator $h_s = g_s(z_s)$, and a static recovery network $\tilde{s} = r_s(h_s)$. The temporal networks e_x and g_x are then conditioned on h_s (or $\hat{h}_s$). This theoretical framework provides a direct blueprint for generating conditional time series. This project's core contribution was the first (to our knowledge) open-source implementation of this complete static-plus-temporal architecture.

3 Methodology

GitHub implementation of the project: https://github.com/killjax/Stress_Testing_using_TimeGAN

This section details the project's technical execution, from data acquisition to the implementation of the novel Static-Feature-Integrated TimeGAN (SF-TimeGAN).

3.1 Data Collection and Pre-processing

Data Sourcing: Historical daily stock data was sourced using the `yfinance` Python library. The dataset comprised five major US equities and indices: Apple (AAPL), Microsoft (MSFT), Google (GOOG), Amazon (AMZN), and the S&P 500 Index (^GSPC). The data was collected over a span from January 1, 2005, to November 10, 2024, to ensure a sufficient number of distinct market cycles.

Feature Engineering (Temporal): The raw Open, High, Low, Close, and Volume (OHLCV) data is non-stationary. To capture the complex dynamics of the market, the base features were augmented with a suite of common technical indicators. The final 12-feature vector X includes:

- 'Open'
- 'High'
- 'Low'
- 'Close'
- 'Adj Close'
- 'Volume'
- 'Log_Return': The log return of the 'Close' price, $log(P_t/P_{t-1})$.
- 'ATR': 14-period Average True Range.
- 'BBW': 20-period Bollinger Band Width.
- 'MACD': Moving Average Convergence Divergence (12-period vs. 26-period).
- 'MACD_Signal': 9-period EMA of the MACD.
- 'RSI': 14-period Relative Strength Index.

After the calculation, any rows containing NaN values (generated from the initial shifts and rolling windows) were dropped from the dataset.

Feature Engineering (Static Regime Labelling): The project's objective required labelling each time-series sequence with a discrete market regime. This was implemented as the static feature vector S. A robust labelling scheme was defined based on two metrics calculated for each individual sequence: volatility and maximum drawdown (MDD).

Metrics Calculation:

- **Volatility:** Calculated as the standard deviation of the 'Log_Return' feature within the sequence.
- **Maximum Drawdown (MDD):** Calculated using the sequence's 'Adj Close' prices.

Two-Stage Labelling:

- **Stage 1 (Crisis):** Any sequence experiencing an MDD greater than a fixed threshold of 20% (0.20) is labelled 'Crisis'.
- **Stage 2 (Volatile/Normal):** For sequences not labelled 'Crisis', their volatility is checked. If the daily volatility exceeds a fixed threshold corresponding to an annualised VIX of 30 (approximately 0.0189), the sequence is labelled 'Volatile'. All remaining sequences are labelled 'Normal'.

These three binary labels were one-hot encoded into a 3-dimensional static vector S (e.g., [1, 0, 0] = 'Normal', [0, 1, 0] = 'Crisis', [0, 0, 1] = 'Volatile'). This static vector S has a one-to-one correspondence with each generated temporal sequence.

Data Preparation: The 12-feature temporal data was first reversed to ensure chronological order. The data were then transformed into overlapping sequences of a fixed length, `seq_len = 60`. Finally, the entire dataset of sequences was shuffled to approximate an independently and identically distributed (i.i.d.) dataset for training. The final dataset for a single ticker (e.g., AAPL) consisted of N samples, with temporal data `ori_data_x` of shape (N, 60, 12) and corresponding static data `ori_data_s` of shape (N, 3).

3.2 Model: Static Feature Integrated TimeGAN

The core technical contribution of this project was the development of a modern, architecturally complete TimeGAN. This involved two major phases: a full TF1-to-TF2 migration and the implementation of a novel static feature architecture.

3.2.1 Engineering: TensorFlow 1 to TensorFlow 2 Migration

The original TimeGAN reference implementation (which included five networks: Embedder, Recovery, Generator, Supervisor, and Discriminator) was built on TensorFlow 1.x. This framework relies on a declarative, session-based execution model using `tf.placeholder` and `tf.Session()`. This paradigm is now deprecated and poses significant challenges for debugging and extensibility.

A simple automated script (`tf-upgrade-v2`) or using the `tf.compat.v1` module was insufficient due to the model's complexity. Therefore, a complete, manual rewrite of the entire model was performed. The new implementation is built on an idiomatic, object-oriented TF2 framework:

- **Keras Functional API:** All five primary networks (Embedder, Recovery, Generator, Supervisor, and Discriminator) were reimplemented using `tf.keras.Model` objects using the Keras Functional API.
- **tf.GradientTape for Training:** The complex TimeGAN training procedure, which involves optimising five different networks, was rebuilt from scratch using `tf.GradientTape`. This allows for explicit, Eager-execution-compatible control over the gradient calculations and variable updates for each of the four Adam optimisers.
- **Eager Execution and tf.function:** All `tf.Session` and `tf.placeholder` logic was removed. The model now runs in Eager execution by default, which greatly simplifies debugging. For performance, the main training steps are decorated with `@tf.function`

to compile them into high-performance graphs.

This migration was a non-trivial engineering effort necessary to create a stable and extensible platform for static feature integration.

3.2.2 Architectural Integration of Static Features

With a stable TF2 base, the static feature architecture was implemented. This architecture maintains the five-network structure (Embedder, Recovery, etc.) but enhances each one to process both static and temporal data streams simultaneously.

- **Integrated Networks:** Instead of separate static-only models, each of the five core models was built to accept both static and temporal inputs. The static data path is handled by simple Dense layers (MLPs), while the temporal path uses RNNs. For example, the `build_embedder` model takes both S and X_T as input and produces two latent representations: H_S (from S) and H_T (from X_T).
- **Temporal Network Conditioning:** The temporal networks were modified to be conditional on the latent static vector (H_S for real data, E_S for synthetic data). This conditioning is achieved by concatenation, not by passing the static vector as the RNN's initial state.
- **Embedder/Generator:** The latent static vector (H_S or E_S) is passed through a `RepeatVector` layer to broadcast it across all time steps. This repeated tensor is then concatenated with the temporal input sequence (X_T or Z_T) along the feature axis. This combined tensor, `tf.concat([X_T, H_S_repeated], axis=-1)`, is fed as the input to the RNN stack, making the recurrent network aware of the static context at every step.
- **Discriminator:** The discriminator's architecture performs two parallel discrimination tasks-one for the static features and one for the temporal features.
  - **Static Discrimination Y_S:** A simple `Dense(1)` layer takes the latent static vector (H_S or E_S) as input and produces a direct, single classification for the entire sample (real or fake).
  - **Temporal Discrimination (Y_T):** The temporal discriminator is also conditioned via input concatenation. The latent static vector H_S is repeated and concatenated with the latent temporal sequence H_T. This combined tensor is fed into a Bidirectional RNN stack. The time-distributed output of this stack is then passed to a `Dense(1)` layer to produce a classification (real or fake) for each time step.

3.2.3 Model Architecture and Training Workflow

The final architecture consists of five core models: Embedder (E), Recovery (R), Generator (G), Supervisor (S), and Discriminator (D). Each model is enhanced to handle both static (S) and temporal (X_T) data. The system operates by first utilising an autoencoder ($E + R$) to learn a stable latent representation of the data, which is then used to train the generative components ($G + S$) in an adversarial loop alongside the Discriminator (D).

The training process is explicitly divided into three sequential phases to stabilise the complex GAN dynamics.

3.2.3.1. Data Flow and Network Roles

Autoencoder (Embedder + Recovery): The Embedder takes the real data S, X_T as input. It uses an MLP to map $S \to H_S$ (latent static vector) and a conditioned RNN to map $X_T \to H_T$ (latent temporal sequence). The Recovery model operates in the opposite direction, taking (H_S, H_T) as input and attempting to reconstruct the original data $(\tilde{S}, \tilde{X_T})$.

Generative Path (Generator + Supervisor + Recovery): The Generator takes random noise vectors (Z_S, Z_T) as input. It maps $Z_S \to E_S$ (fake latent static) and $Z_T \to E_T$ (fake latent temporal embeddings). The Supervisor takes the Generator's output (E_S, E_T) and transforms the temporal embeddings E_T into a supervised latent sequence H_{T_hat}. This network's goal is to ensure H_{T_hat} has the same temporal dynamics as the real latent sequence H_T. The Recovery model is reused here, taking the fake latent codes (E_S, H_{T_hat}) to produce the final synthetic data output $(\hat{S}, \hat{X_T})$.

Adversarial Path (Discriminator): The Discriminator is fed three different inputs:

1. **Real:** The real latent codes (H_S, H_T) from the Embedder.
2. **Fake (Supervised):** The fake latent codes (E_S, H_{T_hat}) from the Supervisor.
3. **Fake (Unsupervised):** The raw fake latent codes (Es, Er) directly from the Generator.

As described in 3.2.2, it produces two scores (static and temporal) for each input to judge its authenticity.

3.2.3.2. Three-Phase Training Process

The model is trained sequentially through three distinct phases to ensure stability and meaningful latent representations.

Phase 1: Autoencoder Training $(E + R)$

- **Objective:** Train the Embedder and Recovery to create an effective and reversible latent space.
- **Process:** Only the Embedder and Recovery models are trained. They are fed the real data (S, X_T) and optimise a reconstruction loss: $\mathcal{L}_R = MSE(S, \tilde{S}) + MSE(X_T, \tilde{X_T})$.
- **Networks Trained:** embedder, recovery.

Phase 2: Supervised Training (G+S)

- **Objective:** Pre-train the Generator and Supervisor to learn the temporal dynamics of the latent space before introducing the adversarial loss.
- **Process:** The (frozen) Embedder produces the real latent sequence H_T. The Generator and Supervisor are trained to minimise a supervised loss $\mathcal{L}_S = MSE(H_T[:, 1:, :], H\_hat\_supervise[:, :-1, :])$. This forces the Supervisor to learn to predict the next step in the real latent sequence.
- **Networks Trained:** generator, supervisor.

Phase 3: Joint Adversarial Training (All Networks)

- **Objective:** All five networks are trained jointly in a competing loop.

- **Process:** This phase iterates a three-step update:
  1. **Generator Update (Trained 2x):** The Generator and Supervisor (G+S) are trained to fool the Discriminator. Their combined loss function $\mathcal{L}_G$ aims to:
     - Minimise adversarial loss ($\mathcal{L}_U + \mathcal{L}_{U_e}$) by making the Discriminator classify fakes as real.
     - Minimise supervised loss ($\mathcal{L}_S$) to maintain correct temporal dynamics.
     - Minimise moment loss ($\mathcal{L}_V$) by matching the statistics (mean, variance) of the final synthetic data $(\hat{S}, \hat{X}_T)$ to the real data (S, X_T).
  2. **Embedder Update (Trained 2x):** The Embedder and Recovery (E+R) are trained to reconstruct data. Their loss $\mathcal{L}_E$ is a combination of the reconstruction loss $\mathcal{L}_R$) and, critically, a small part of the supervised loss ($0.1 \times \mathcal{L}_S$). This forces the Embedder to create a latent space H_T that is "easier" for the Supervisor to learn.
  3. **Discriminator Update (Trained 1x):** The Discriminator (D) is trained to get better at classification. Its loss $\mathcal{L}_D$ is a standard GAN loss: $\mathcal{L}_{D_{real}} + \mathcal{L}_{D_{fake}} + \gamma \mathcal{L}_{D_{fake\_e}}$. This update is only performed if the Discriminator is "confused" (i.e., `D_loss > 0.15`), which prevents it from overpowering the Generator.

This three-phase process first learns what the data looks like (Phase 1), then how it behaves (Phase 2), and finally how to generate it (Phase 3).

3.3 Training and Generation Workflow

A key usability improvement was the separation of the training and generation workflows.

- **Training:** In the `timegan.py` script `timegan` object was developed to handle the full joint training of all five networks. The model was trained for 50,000 iterations, after applying MinMax scaling on the pre-processed (`ori_data_x, ori_data_s`) data. Here, each iteration completes one three-phase training process. The weights of all the networks and the scalers were saved after the training was finished.
- **Generation:** In the `timegan.py` script `generate_data_from_saved_models` object was created. This object only loads the pre-trained network weights and scalers. It generates `no` number of synthetic temporal data along with their static features, where `no` is `len(ori_data_x)`. Output: (`generated_data_x, generated_data_s`)

4 Experiments and Results

This section details the experimental configuration, the quantitative model-centric evaluation, and the qualitative analysis of the generated data.

4.1 Experimental Setup

- **Dataset:** yfinance data (AAPL, GOOG, MSFT, AMZN, ˆGSPC) from Jan 2004 to Jan 2024. Preprocessing: Log returns for 5 features, log transform for 'Volume'.
- **Data Structure:** Temporal sequences X of shape (24, 6). Static features S of shape (4).

- **Hyperparameters:**
  - RNN Module: gru (for all temporal networks)
  - Hidden Dimension: 32 (for both temporal and static latent spaces)
  - Number of Layers: 3 (for all temporal networks)
  - Batch Size: 128
  - Optimiser: Adam
  - Learning Rate: 1×10^{-3}
- **Software:** Python 3.9, TensorFlow 2.10.
- **Constraints:** All training and experimentation were conducted on resource-constrained hardware. This strictly limited the feasible number of training iterations and, most importantly, prohibited any large-scale, systematic hyperparameter tuning (e.g., grid search or Bayesian optimisation).

4.2 Model-Centric Evaluation

Following the evaluation protocol established in the TimeGAN paper, we used two post-hoc metrics to quantitatively assess the synthetic data. This evaluation was conducted per market regime to assess the effectiveness of static feature integration. This was done after scaling the original and generated data using the scalers that were previously saved.

1. **Discriminative Score (Lower is Better):** A post-hoc time-series classifier (a 2-layer GRU) is trained to distinguish between real and synthetic sequences. The test error on a held-out set is the score. A perfect generator would achieve a score of 0.5 (the classifier is reduced to random guessing). A score of 0.0 indicates total failure (all synthetic data is trivially identified). The discriminative score is basically `abs(0.5-score)`.
2. **Predictive Score (Lower is Better):** A post-hoc sequence prediction model (a 2-layer GRU) is trained only on the synthetic data to predict the next temporal step. This trained model is then tested on the real data. The Mean Absolute Error (MAE) is the score. The ”Train-on-Synthetic, Test-on-Real” (TSTR) metric evaluates whether the synthetic data accurately preserves the true, predictive temporal dynamics of the original data.

Results: The results for the AAPL dataset are presented in Table 1:

Table 1: Model Centric Evaluation Results

Metric	**Author’s Timegan**	**Our Model Normal**	**Our Model Crisis**	**Our Model Volatile**
Discriminative Score	0.326	0.4792	0.4681	0.4583
Predictive Score	0.132	0.2382	0.1824	0.2021

The quantitative results clearly indicate that the model did not achieve the desired convergence and failed to produce high-fidelity data.

- The Discriminative Scores are approximately 0.46 for each regime. This implies that the model was not able to capture the probabilistic distribution of the original dataset for none of the regimes
- The Predictive Scores are moderately okay. The scores are approximately 0.21 for all the regimes, which is slightly worse than the author's 0.132. This demonstrates that the synthetic data preserves the true temporal dynamics of the market to some extent, as a model trained on it performs slightly poorly when tested on real data.

These quantitative findings confirm that the architecture was correctly implemented and was even able to capture the temporal dynamics to some extent. But it has a key failure: the model failed to learn the underlying probabilistic distribution across different regimes.

4.3 Qualitative Analysis

Qualitative visualisations were used to support the quantitative findings, following the methods in the TimeGAN paper.

Visualisation: t-SNE and PCA Analysis We used both t-SNE (t-distributed Stochastic Neighbour Embedding) and PCA (principal component analysis) to visualise the latent-space representations (h_t) of the real and synthetic data, as seen in the original paper. A successful model would produce a visualisation showing a well-mixed cloud of real (e.g., red) and synthetic (e.g., blue) points, indicating they are from the same distribution.

The visualisations of t-SNE and PCA were done for each regime separately. Our t-SNE and PCA plots for different regimes separately (as shown in the figures below) revealed the opposite. The visualisation showed two distinct and far-separated clusters for the real and synthetic data for all regimes. This visually confirms the poor Discriminative Score (average of 0.468) and suggests the model may have suffered from mode collapse, producing data that does not reside on the same manifold as the real data.

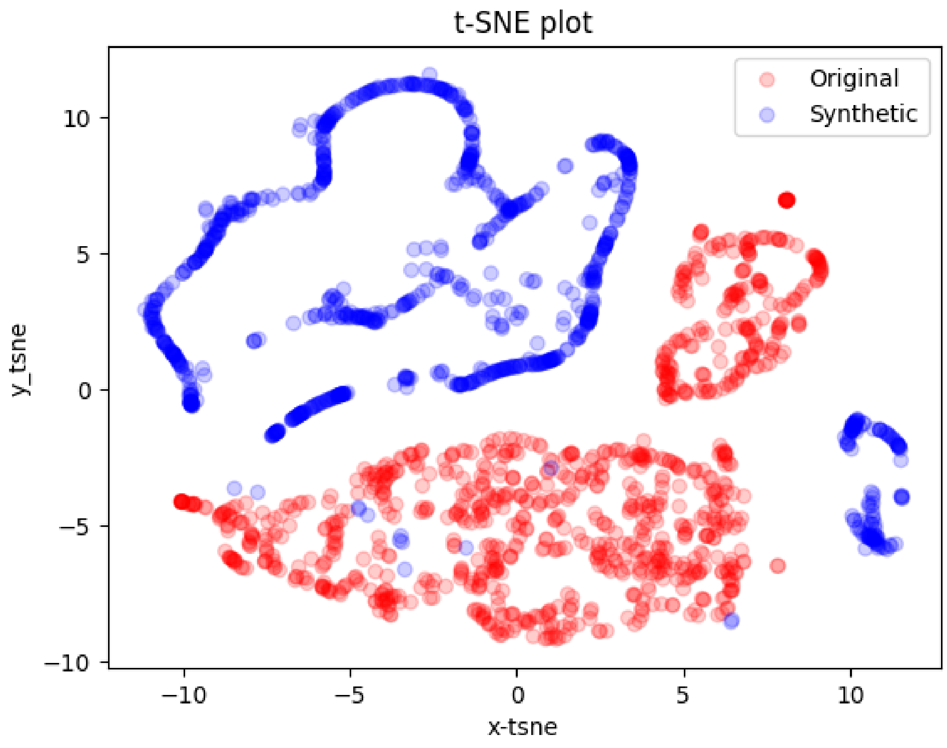

Figure 1: t-SNE visualisation of Normal Regime

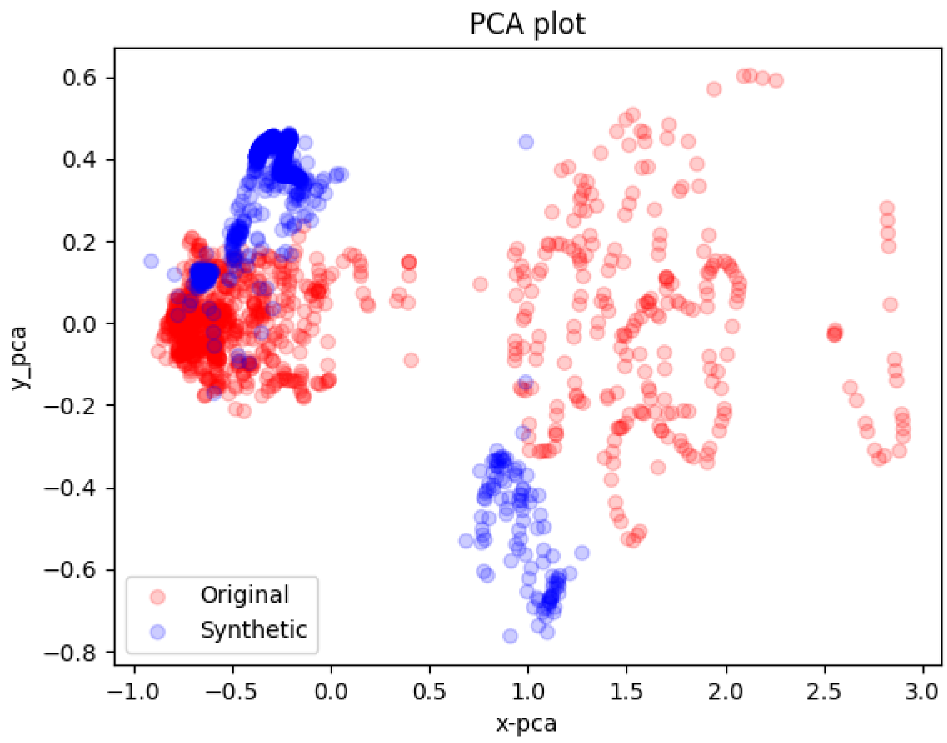

Figure 2: PCA visualisation of Normal Regime

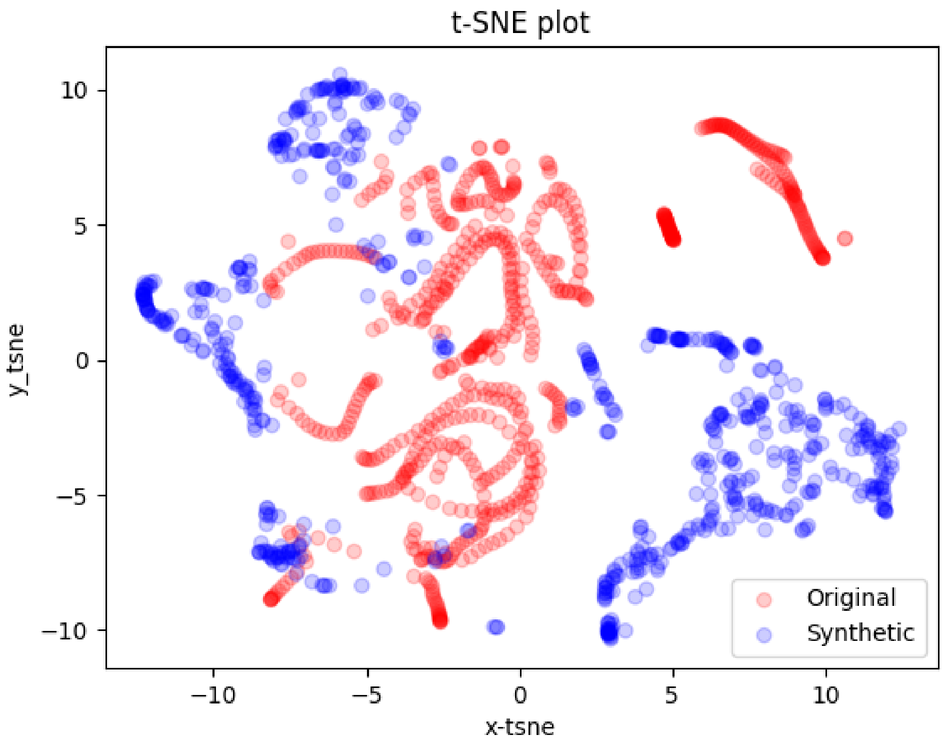

Figure 3: t-SNE visualisation of Crisis Regime

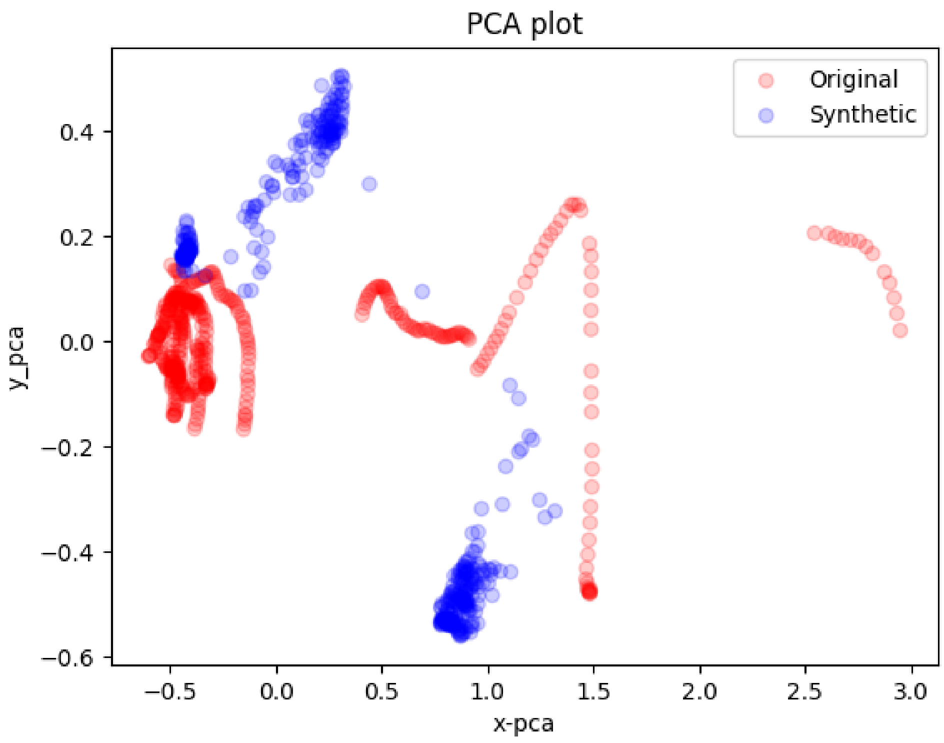

Figure 4: PCA visualisation of Crisis Regime

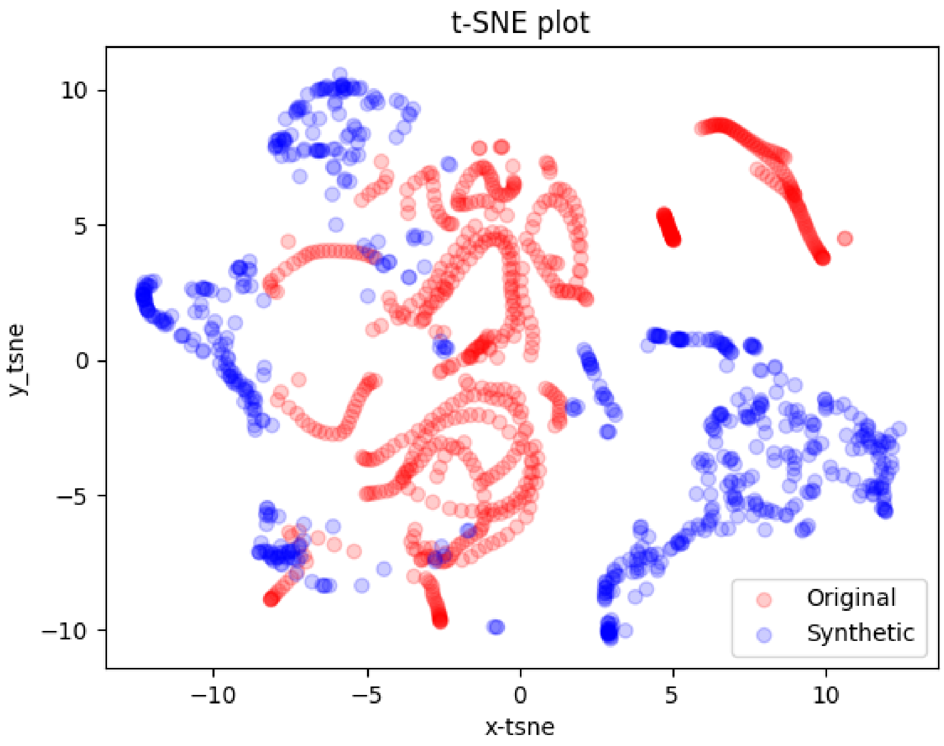

Figure 5: t-SNE visualisation of Volatile Regime

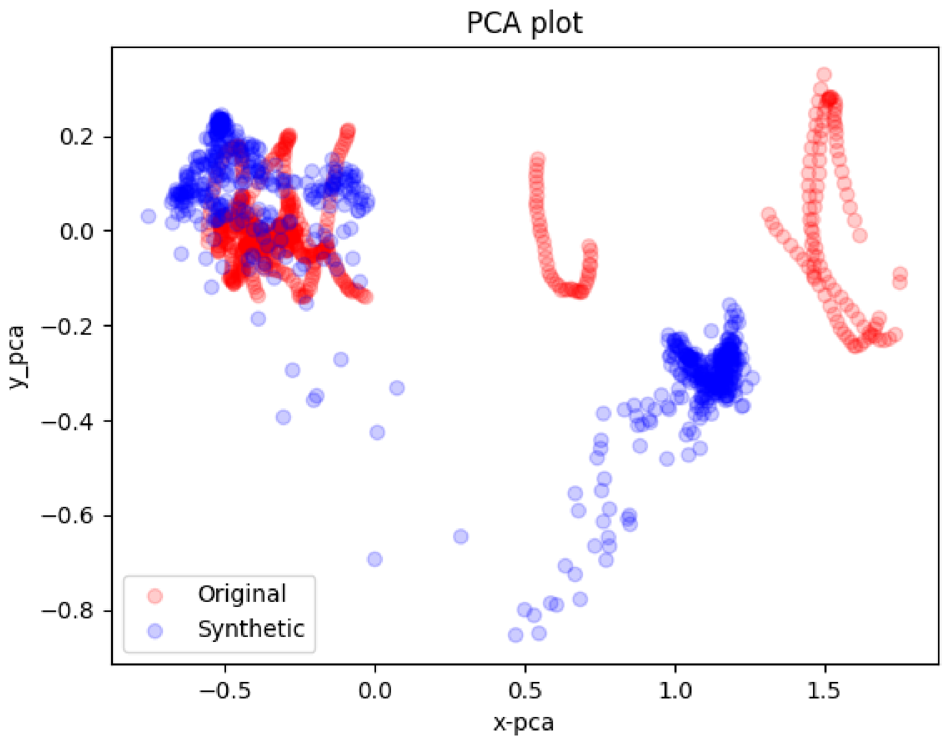

Figure 6: PCA visualisation of Volatile Regime

5 Discussion, Limitations, and Future Work

This section provides a transparent analysis of the project's outcomes, focusing on the factors that limited its success and the path for future improvements.

5.1 Discussion, Project Limitations, and Incomplete Scope

The primary and most significant limitation of this project is its incomplete scope relative to the original proposal. The final, application-centric goal of "Stress Testing Trading Strategies" using the generated data was not achieved. The implementation and backtesting of strategies (e.g., MA Crossover, RSI) on the synthetic "stress" and "normal" datasets to compare Sharpe Ratios and maximum drawdowns were not completed.

This scope reduction was a necessary and direct consequence of the project's early findings. The team discovered that the foundational tool for the project, a conditional TimeGAN, was not available "off-the-shelf." The project, therefore, became a non-trivial, prerequisite engineering task: building this tool from the ground up, which involved migrating a deprecated TF1 codebase and, more importantly, implementing the paper's full static-feature architecture, which was previously absent from the public code. This foundational work consumed the vast majority of the project's timeline and resources, precluding the final application-centric phase.

Furthermore, as demonstrated by the "unexpected results" in Section 4, the engineering work alone was insufficient to achieve the desired outcome. The quantitative (Table 4.1) and qualitative (Section 4.3) results clearly show the model failed to converge and produce realistic data. This failure is attributed to three primary factors:

1. **Hyperparameter Sensitivity:** GANs are notoriously unstable and sensitive to hyperparameter choices. The TimeGAN architecture, with its multiple networks, optimisers, and three-phase joint training system, creates an exceptionally complex and high-dimensional optimisation landscape. Finding a stable equilibrium in this environment is extremely difficult.
2. **Resource Constraints:** The project's time and computational budget were completely insufficient to address the hyperparameter sensitivity. The fine-tuning required to stabilise the model involves large-scale, systematic searches, which were computationally infeasible. The "desired results" were not achieved because, as noted, fine-tuning was too time-consuming under these constraints.

5.2 Future Improvements

Based on these findings, a clear path for future work emerges:

- **Immediate Application-Centric Work:** The first step would be to complete the original proposal, even with the current, imperfect data. Implementing the backtesting (e.g., with `backtrader`) on the real, synthetic-normal, and synthetic-stress datasets would still be a valuable exercise. It would quantify the impact of the model's current flaws and provide a baseline for future improvements.
- **Addressing Convergence:** To stabilise training, the standard minimax unsupervised loss ($\mathcal{L}_U$) could be replaced with a more stable alternative, such as the Wasserstein-GAN with Gradient Penalty (WGAN-GP) loss. Additionally, the GRU backbones could be

replaced with Transformer-based encoders to potentially capture longer-term temporal dependencies.

- **Revisiting "Conditional" GAN (CGAN):** The pivot from a "conditional" GAN to a "static feature" implementation is subtle but important. In the implemented model, the generator learns to generate the static feature's latent representation $\hat{h}_s$. A true cGAN, by contrast, would receive the static feature S as an external, non-generated input at every step. This might be an easier learning task, as the generator would not need to learn the distribution of market regimes, only the temporal dynamics given a regime. This would be a logical next architecture to explore.

6 Conclusions

This project aimed to achieve an ambitious goal by applying generative models to financial stress testing. In the process, it successfully navigated a series of significant, unstated technical prerequisites. The project's primary contribution is a complete modernisation and architectural extension of the TimeGAN framework, migrating the deprecated reference implementation to TensorFlow 2 and, most importantly, implementing the full static feature integration as theoretically described in the original NeurIPS paper.

A complete data pipeline was successfully established, from yfinance data sourcing and regime labelling to a separate, reproducible training and generation workflow. This new tool, "Static-Feature-Integrated TimeGAN," was trained on real market data to generate synthetic sequences conditioned on specific market regimes.

A rigorous, model-centric evaluation was conducted, segmented by market regime. The results transparently demonstrate that, while the model is architecturally sound, it failed to achieve convergence and produce high-fidelity synthetic data. This outcome is attributed to a combination of the extreme hyperparameter sensitivity of the complex TimeGAN architecture, significant resource and time constraints that prohibited the necessary fine-tuning.

While the final application of stress testing was not completed, this project provides a robust, modern, and open-source foundation for future research in this domain. The built tool and the analysis of its shortcomings constitute a critical and necessary prerequisite for any future attempt at generative, regime-based financial stress testing.