MTSS-GAN: Multivariate Time Series Simulation Generative Adversarial Networks

Derek Snow, Alan Turing Institute

Abstract

MTSS-GAN is a new generative adversarial network (GAN) developed to simulate diverse multivariate time series (MTS) data with finance applications in mind. The purpose of this synthesiser is two-fold, we both want to generate data that accurately represents the original data, while also having the flexibility to generate data with novel and unique relationships that could help with model testing and robustness checks. The method is inspired by stacked GANs originally designed for image generation. Stacked GANs have produced some of the best quality images, for that reason MTSS-GAN is expected to be a leading contender in multivariate time series generation.

In this paper we propose a framework of two network encoders, two generators and two discriminators locked into two adversarial models. The framework has the flexibility to branch out to include additional targeted conditional features. Each additional feature layer will require an additional encoder, generator, and discriminator layer. At the end we discard all the models except the two generators; the one feeds into the other to generate the final synthetic dataset. The code for this model has been made available for further experimentation by researchers and practitioners[1]. This is the first MTS GAN that has been developed in the new TensorFlow 2 framework.

Introduction

Simple conditional GANs allow us to generate specific outputs using noise code and one-hot labels as input (Mirza & Osindero, 2014). The benefit of MTSS-GAN over conditional GAN's is that the adjustments can be made both with one-hot labels and disentangled latent code[2]. There has been research looking into conditional univariate simulations in finance, but not yet for conditional multivariate simulations (Fu et al., 2019).

There has also been synthesisers developed in the medical community to develop conditional multivariate time series, but without the flexibility afforded by using disentangled latent code (Esteban et al., 2017). Disentangled code is useful because you can change specific data attributes while not affecting others. This is possible because the disentangled code is either inputs to network encoders or outputs from network encoder and therefore acts more like features than gaussian noise.

MTSS-GAN uses a pretrained encoder classifier to disentangle latent codes. MTSS-GAN is a stack of models each with an encoder and a GAN similar to the stacked GAN framework for image generation (Huang et al., 2017). Each GAN is trained in an adversarial manner by using the input and output data of the corresponding encoder[3]. MTSS-GANs can also be stacked to allow for various types of features, but even when this is not done, disentangled code can still be adjusted and the effect on generated data can be recorded[4].

[1] GitHub: https://github.com/firmai/mtss-gan/; this paper should be read in parallel with the code which includes settings and specification that if included in this paper, would see to a tripling in its length.

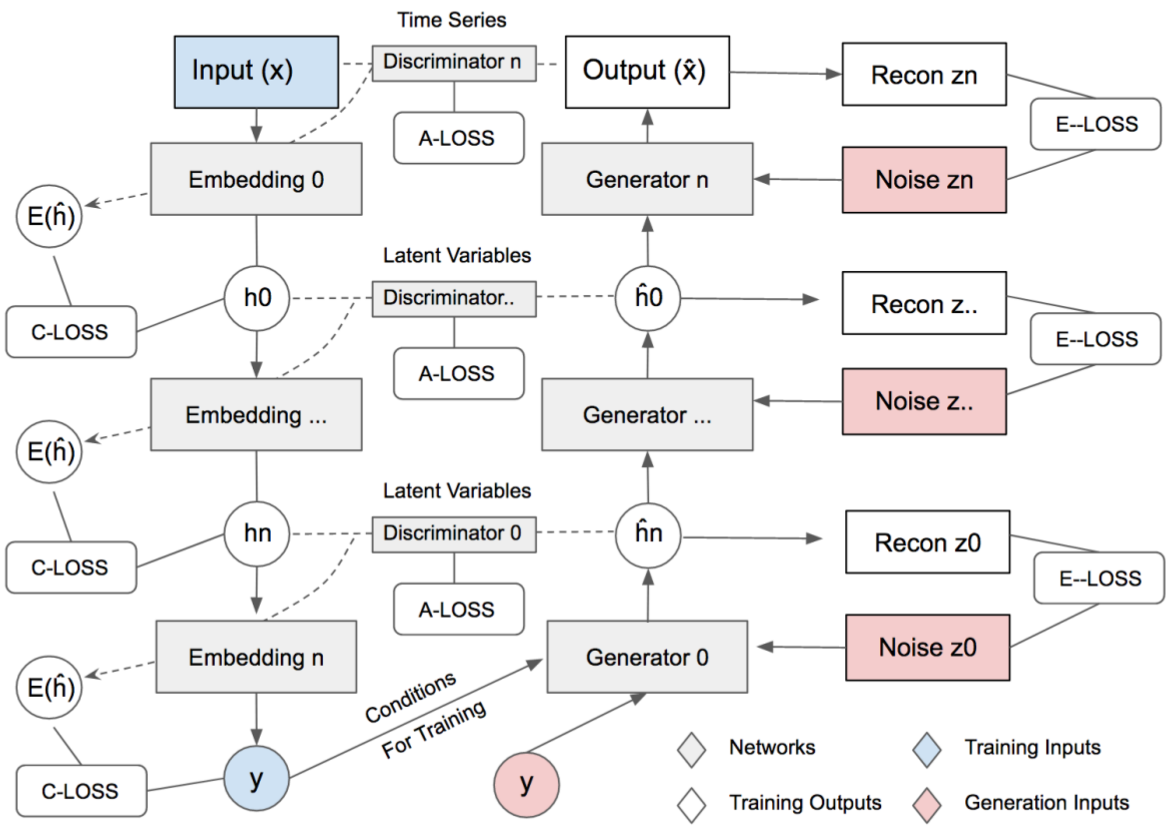
[2] InfoGan a method that maximises mutual information also disentangles latent code, it is both easier to implement and faster to train that MTSS-GAN but does not produce the same quality of data.

[3] Additional adjustments are made in the adversarial learning model with the interaction of generators and discriminators to minimise conditional label loss and conditional latent feature loss.

[4] This is also the first multivariate TS GAN that has been developed in the new TensorFlow 2.

If, for example, a normal GAN uses 200-dimensional noise code, then all the features are entangled within these dimensions; the salient attributes are not disentangled. With MTSS-GAN we separate the code into entangle and disentangled interpretable latent code to teach the generator what to synthesise. The n-dimensional entangled code is still necessary to represent all the attributes that have not been disentangled. Each GAN is trained independently in the usual discriminator-adversarial manner with its own latent code.

Figure 1



To train the MTSS model, we start with the original data and feed it into the embedder; the embedding system then gets trained in its entirety against labelled conditions. The discriminators at each layer is then trained separately, after which the generators learns to duplicate the embedding outputs though an adversarial process. The discriminator models learn from two losses, the traditional discriminator loss (binary cross entropy), and entropy loss (mean squared error). The adversarial joint-training model learns from three losses, the conditional loss (mean squared error and categorical cross entropy), the adversarial loss (binary cross entropy), and the entropy loss (mean squared error). The end result is a range of generators, each feeding into the next to produce the final outcome. The generator process starts by feeding noise data and your preferred conditions into the generator.

The first version of MTSS-GAN uses one feature encoder that is broken down into an encoder of latent code and an encoder of labels pertaining to data conditions. Additional encoders could be included to represent additional features. The framework in its entirety is presented in *figure 1* and makes reference to the fact that additional layers could be included. Each GAN therefore uses latent code to condition its generator output. The latent code of each GAN can be used to alter specific attributes. Even the noise code can be used to alter the attributes being created, but altering the noise code is less pure and we would in fact have to investigate the changes to the generated data post-hoc. We can adjust the noise code at every encoded level.

Method

The specific flavour of MTSS-GAN that we have been developed uses only two encoders[5]. The original data is converted to latent features using the first network encoder, and the latent features is converted to class labels using the second encoder. Because the first encoder is a input to the second encoder, a combined model is created that takes the input of the first encoder and produces the output of the second encoder. The second encoder is essentially a classifier, as a result the full encoder can be trained using the real labels. This is where the training of the encoders stops. At this stage they are fully trained and ready to be used in the adversarial models.

The next step is to use generators to invert the encoder back. We start with the class labels and walk back to the latent features and thereafter back to the original data. The two generators, therefore, simply inverts the two encoders to retrieve real looking time series data. For that reason, these generators can be thought of as network decoders whose parameters are learned through an adversarial process. The first generator seeks to learn the latent feature representation of the first encoder by incorporating real class labels and noise code. The second generator seeks to invert the second encoder by starting with latent features and noise code to produce real looking data.

The purpose of this inversion is to expose additional unentangled input parameters to the user so that they can prespecify certain data attributes that they want to generate. All the generators are trained via an adversarial process using discriminators. The generators are pinned against discriminators that seeks to identify how far off the generator model outputs are from their real counterparts. For the first generated model the real counterpart is the latent features encoded using real data, for the second generated model, the real counterpart is the original data.

The first adversarial model is concerned with the probability that the synthetic latent features produced by the generated model matches that of the first encoder (adversarial loss) and that the latent features encoded to synthetic labels are similar to the labels encoded from the latent variables as derived from the original data (conditional loss). The second adversarial model is concerned first with the probability that synthetic time series data generated by the model matches that of the original data (adversarial loss) and that the synthetic time series data encoded to synthetic latent variables are similar to the latent variables encoded from the original data (conditional loss).

The reason we are concerned both with the adversarial and conditional losses is because the adversarial loss helps us to produce realistic looking data, whereas the conditional loss helps us to create generative models that respect the conditions imposed onto them. However, the conditional loss function introduces an additional problem in that the input noise is ignored in favour of the conditional values, therefore we have to introduce an entropy loss function to ensure that the generator does not ignore the noise code.

We need the model to pay attention to the noise code to increase the diversity of the samples being produced and ensure that the model does not overfit on the latent conditions. The conditional loss and entropy loss are therefore in competition for network attention. To measure the entropy loss, we recover the noise code from the outputs of the generator in an auxiliary network attached to a discriminator[6]. As a result, we would like to balance the loss at an appropriately acceptable level instead of allowing their importances to be purely a function of adversarial loss.

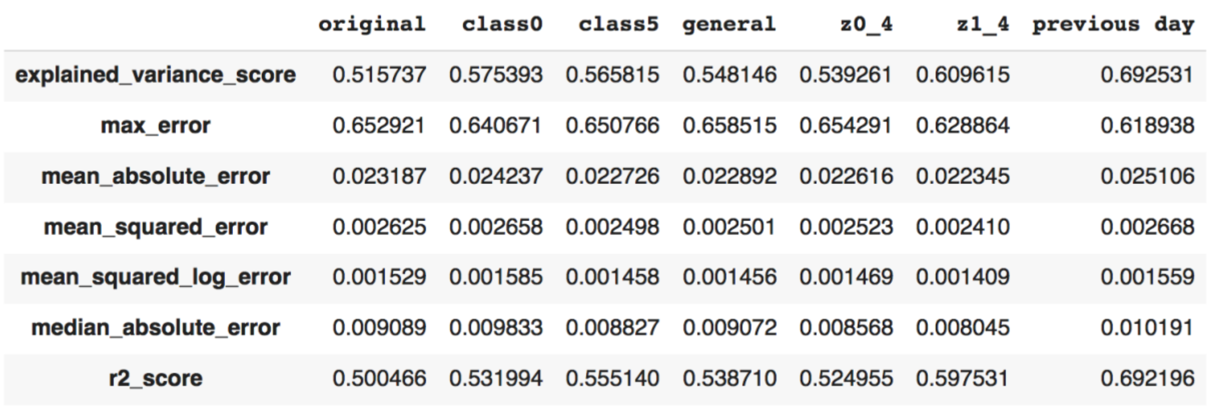
[5] The model, *figure 1*, shows how the model can ordinarily be expanded to include more layers.
[6] The difference between the input noise and the recovered noise is measured in MSE

Utility

In an attempt to compare the model with related research, we have decided to use the same Google stock data as has been used by the best performing medical time-series synthesiser (Yoon et al., 2019). The RNN model the researcher used for the predictive score achieved 0.038 mean absolute error (MAE), whereas the worst model we used achieved a MAE of around 0.024. The purpose is not so much to compare these results, but rather the difference between the original datasets and the generated dataset for each respective study. TimeGAN, the proposed model by Yoon et al., performed 5% worse than the original dataset, our proposed method, MTSS-GAN, generally performs at the same level as the original data. This is mostly likely a result of our encoding-decoding process that removes the noise from the reconstructed dataset leading to better generalisable performance.

In this section we condition the MTSS model on a few arbitrary recipes to compare the utility and similarity across different generated datasets. We have generated five different datasets from the model based on certain conditions. The first, *class 0,* is conditioned on returns of -10% and less over a period of 24 days; the second, *class 5* is conditioned on +10% return or more over a 24 day period; the third, *general*, is conditioned on a non-stratified noise class; for the fourth generator, *z0-4*, the noise code of the first generator is replaced with the value four; and in the fifth generator, *z1-4* the noise code of the second generator is also replaced with the value four. These conditions are arbitrarily chosen. Our purpose is just to see the similarity of these datasets using the first three conditions, and the diversity of the datasets using the last two.

Table 1



	original	class0	class5	general	z0_4	z1_4	previous day
explained_variance_score	0.515737	0.575393	0.565815	0.548146	0.539261	0.609615	0.692531
max_error	0.652921	0.640671	0.650766	0.658515	0.654291	0.628864	0.618938
mean_absolute_error	0.023187	0.024237	0.022726	0.022892	0.022616	0.022345	0.025106
mean_squared_error	0.002625	0.002658	0.002498	0.002501	0.002523	0.002410	0.002668
mean_squared_log_error	0.001529	0.001585	0.001458	0.001456	0.001469	0.001409	0.001559
median_absolute_error	0.009089	0.009833	0.008827	0.009072	0.008568	0.008045	0.010191
r2_score	0.500466	0.531994	0.555140	0.538710	0.524955	0.597531	0.692196

The explained_variance_score computes the explained variance regression score. The max_error function computes the maximum residual error that captures the worst case error between the predicted value and the true value. The mean_absolute_error function calculates a risk metric corresponding to the expected value of the absolute error loss or l1-norm loss. The mean_squared_error function computes mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error or loss. The mean_squared_log_error function computes a risk metric corresponding to the expected value of the squared logarithmic (quadratic) error or loss. The median_absolute_error is particularly interesting because it is robust to outliers. The loss is calculated by taking the median of all absolute differences between the target and the prediction. The r2_score function computes the coefficient of determination.[7]

The results in the *table 1* show the performance of a one-day-ahead multi-dimensional recurrent neural network model. Here we also include the performance of the original data in the first column, and a robustness check in the last column by taking the previous day's value as the next day's prediction. The results show that performance improvements of more than 3%-10% can be achieved

[7] As defined and used in https://scikit-learn.org/

when training a model using *z1-4* generated data depending on the metric. This initial check is done to see if the generated data provides good predictive utility.

Similarity

The last page shows a range of 30 plots to determine the similarity across the original and five generated models. The generated models are the same as those chosen above for the utility tests. The first chart-row looks at temporal correlation, the second row looks at feature correlation, the third is a t-SNE decomposition, the fourth a PCA decomposition[8], and the last looks at time-step correlations.

In the first four chart columns the purpose is to achieve similar looking datasets to that of the original datasets[9]. In the first three columns after the original dataset model; the data used to compare the similarity with the generated data have been selected based on the conditions imposed on the generative model to allow for a fair visual comparison. In the last two chart columns the purpose is to generate data with unique relationships; in which case we prefer more distinct patterns when compared to the original dataset.

The model generally achieve its dual-purpose of being able to generate similar looking data while also being flexible enough to generate diverse outputs. We are able to generate similar looking data after 2000 training steps with a batch size of 64. The *general* model's visual similarity is not as strong as the TimeGAN synthesizer; this is the result of the conditioning functionality in MTSS-GAN that is essential in generating new diverse datasets with specific attributes[10].

Conclusion

Many small adjustments can be made to the model in terms of learning parameters, network layer type and depth, the number of dimensions in noise code, and the preselection of conditions. The model as presented is a collection of a few core components but also includes a range of arbitrary choices. Future research can test different loss functions and can compare the use of different RNN models within the generator and discriminator models[11]. We can similarly make different design choices such as introducing weight clipping or gradient-penalties. It is our expectation that every field will have a different set of optimal parameters depending on the problem they want to solve.

This paper highlights that generative models can be used not just to produce similar looking data, but also to produce diverse datasets that could help us to develop robust models with generalisable performance for any data-centric financial domain[12]. I also foresee a future where generated data becomes part of the financial toolset where original data can be augmented to better highlight patterns and signals for prediction models to identify and act on. The conditions in the GAN model could be automatically selected based on how well the generated data performs on a validation set. It is my hope that the model as specified can be improved on with further iterations, and that more tests can be run to highlight the advantage of using a stacked GAN approach for generative multivariate time series simulations.

[8] The t-SNE and PCA decompositions are set up exactly like that of Yoon et al. (2019)
[9] The first chart column compare the original data with itself.
[10] The model trains in less than 10 minutes on Google Colab; it is very possible that this gap can be closed by increasing the training steps and other model parameters.
[11] This model used a GRU layer; for more information on the model architecture, see the code.
[12] Although novel in pattern, the diverse data still maintains the underlying logic of the original data.

References

Esteban, C., Hyland, S. L., & Rätsch, G. (2017). Real-valued (medical) time series generation with recurrent conditional gans. *ArXiv Preprint ArXiv:1706.02633*.

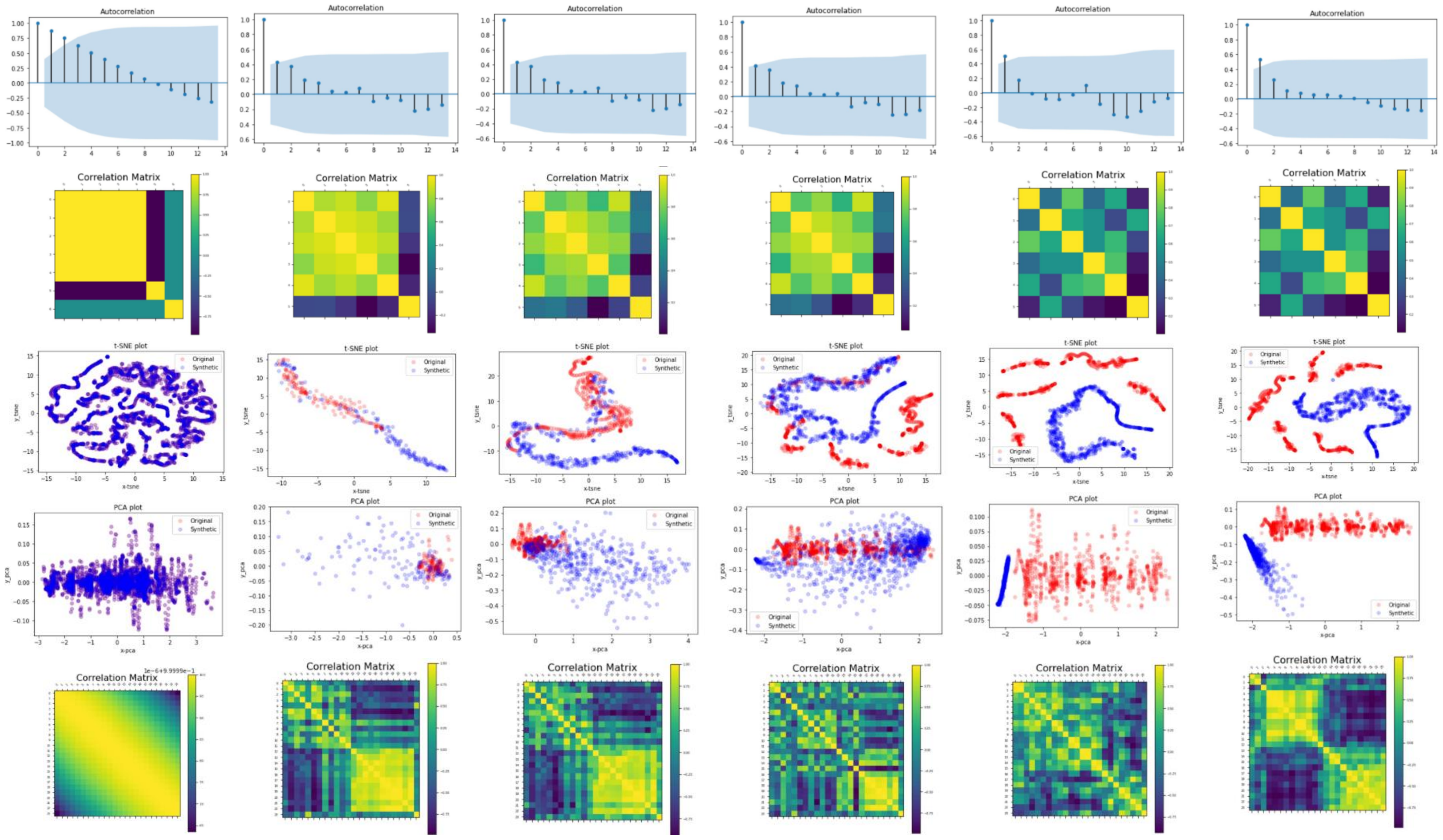
Fu, R., Chen, J., Zeng, S., Zhuang, Y., & Sudjianto, A. (2019). Time Series Simulation by Conditional Generative Adversarial Net. *ArXiv Preprint ArXiv:1904.11419*.

Huang, X., Li, Y., Poursaeed, O., Hopcroft, J., & Belongie, S. (2017). Stacked generative adversarial networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 5077–5086.

Mirza, M., & Osindero, S. (2014). Conditional generative adversarial nets. *ArXiv Preprint ArXiv:1411.1784*.

Yoon, J., Jarrett, D., & van der Schaar, M. (2019). Time-series Generative Adversarial Networks. *Advances in Neural Information Processing Systems*, 5509–5519.

Original Class 0 Class 5 General Z0-4 Z1-4


Autocorrelation
Correlation Matrix
t-SNE plot
PCA plot
Original
Synthetic
x-tsne
y_tsne
x-pca
y_pca