Delft University of Technology

MSc Applied Mathematics

MASTER THESIS

Enriching Financial Datasets with Generative Adversarial Networks

by
Fernando de Meer Pardo
4696700

May 2019

Supervisor: Cornelis W. Oosterlee
Company Supervisor: Rafael Cobo López

Acknowledgements

A Óscar, por ser mi pilar y mi guía.
A mi madre y a mi padre, por toda una vida de amor.
A Javi y a Leire, por ser mis compañeros de viaje.

Thanks to my supervisor Kees Osterlee, for patiently guiding me through the Thesis process, for being a great teacher and for giving me a chance.

Thanks to all the friends I made at TU Delft, my classmates from the Financial Engineering specialization, the people from Force Elektro and the Spartan Workout, for making me feel loved and at home in Delft.

Thanks to all the people at ETS Asset Management Factory, for believing in me and welcoming me with open arms.

Abstract

The scarcity of historical financial data has been a huge hindrance for the development algorithmic trading models ever since the first models were devised. Most financial models assume as hypothesis a series of characteristics regarding the nature of financial time series and seek extracting information about the state of the market through calibration. Through backtesting, a large number of these models are seen not to perform and are thus discarded. The remaining well-performing models however, are highly vulnerable to overfitting. Financial time series are complex by nature and their behaviour changes over time, so this concern is well founded.

In addition to the problem of overfitting, available data is far too scarce for most machine learning applications and impossibly scarce for advanced approaches such as reinforcement learning, which has heavily impaired the application of these novel techniques in financial settings.

This is where data generation comes into play. Generative Adversarial Networks, GANs, are a type of neural network architecture that focuses on sample generation. Through adversarial training, the GAN can learn the underlying structure of the input data and become able to generate samples very similar to those of the data distribution. This is specially useful in the case of high-dimensional objects, in which the dimensions are heavily inter-dependent, such as images, music and in our case financial time series.

In this work we want to explore the generating capabilities of GANs applied to financial time series and investigate whether or not we can generate realistic financial scenarios.

Keywords : Generative Adversarial Networks, WGAN-GP, Relativistic GAN, Data Augmentation, Overfitting, Time Series generation, Time Series Classification, Financial Machine Learning.

Contents

List of Figures

1 Introduction

Generating data structures with similar characteristics to those of a given dataset is a process that involves primarily two tasks, feature-extraction and feature-reproduction. These two tasks are fully automated through Generative Adversarial Networks, GANs for short [14].

The main application in which GANs have been proven to work well has been image generation, in which starting with simple MNIST numbers and going all the way to bedroom pictures [36] and celebrity photos [7, 22], many advancements have been developed to generate synthetic samples. Having a way to artificially create complex and rich artificial datasets can be a very useful tool when training other kinds of models, albeit because the original data is scarce, protected by privacy regulations or simply too expensive to recollect.

GAN's success with images has been theorized to be due to the strong spatial relationships of the inputs (meaning pixel values in photographs) and an easy measure of performance, simple visual inspection.

Applying GANs to time series data has also been explored, because of the lack of access to medical data due to privacy regulations in the case of [10], and similarly with banking data in [40]. The main challenge data generation faces is the possible lack of structure in the input data, but in our case, financial time series have been recognized to share some common traits, such as having heavy tails in the distribution of returns, volatility clustering, mean reversion and momentum (see [6, 29]) so we are not completely hopeless.

Our aim is having GANs that can reflect all the characteristics of financial time series we are able to calculate as well as the characteristics we may be unaware of, as GANs learn the underlying structure of our data, rather than just a set of features. If we are able to achieve this, the synthetic datasets we create could be used for a variety of purposes including model training, model selection, option pricing (in lieu of classical stochastic models) etc.

1.1 Research Questions

Having stated our aim we define the following research questions:

- *Can we generate realistic synthetic financial time series using GANs?*
- *How can we evaluate the quality of our generated series/ performance of our model?*
- *How can we use our synthetic series? Can we train models on them?*

1.2 Contributions

In this work we carry out the following:

First we implement Wasserstein's GAN [3] with Gradient Penalty [16] adapted to 1-dimensional time series; we carry out a sanity check with artificial data (sine curves) to test the model robustness. Once the sanity check is passed we apply our algorithm to return series of components of the SP500 for a fixed period and check the quality of our generated samples by comparing statistics.

Second, we induce variability on our dataset by taking rolling windows of our series. We verify the effectiveness of our methodology by producing new scenarios of a second dataset, the daily prices CBOE VIX Index. As the VIX is a mean-reverting index, we modify our Wasserstein GAN in order to work in a conditional setting [32].

Third, we discuss the challenges behind generating diverse multi-dimensional scenarios and propose a Relativistic Average GAN [21] employed in a conditional setting as a solution. Once again we test it on artificial data (sine curves for consistency) and on a real case, producing joint scenarios of the VIX and the SP500.

Fourth, we show an example of usage of synthetic data, we train our model on a supervised Time Series Classification task [20], identifying trends in financial time series, we first do it the classic way, dividing historic data on train and test sets and later we enrich our train set with synthetic series (created with the same train set) and check for performance improvements in the test set, this is known as the "Train on Synthetic, test on

Real" approach [10]. We carry out this experiment with the VIX index data.

1.3 Thesis Structure

In Chapter 2 we detail the mathematical foundations of our models. Section 2.1 presents an introduction to basic neural networks concepts, section 2.2 introduces the field of generative modelling and the basic concepts of Generative Adversarial Networks, section 2.3 analyzes recent advances on the understanding of GANs and some advanced setups derived from them. In Chapter 3 the experiments carried out in this work are described, sections 3.4.1 and 3.4.2 discuss the evaluations of our results. Chapter 4 summarizes our work and presents conclusions and Chapter 5 discusses possible further research directions and topics.

2 Background

In this chapter we start by giving a brief introduction to neural networks and some related concepts. Then we introduce deep generative models with a focus on GANs, going over how they function and how they are constructed. Finally we conclude with the technical details of the GAN versions we employ, explaining the breakthroughs that led to their development and their justification and usefulness in our work.

2.1 Basic concepts about neural networks

Linear models for regression are based on linear combinations of fixed nonlinear basis functions $\phi_j(x)$ and take the form

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=1}^{M} w_j \phi_j(\mathbf{x})\right) \tag{1}$$

where $f(\cdot)$ is a nonlinear activation function in the case of classification and is the identity in the case of regression. Neural Networks are an extension of this concept which make the basis functions $\phi_j(x)$ depend on parameters and then to allow these parameters to be adjusted, along with the coefficients $\{w_j\}$, during training.

This leads to the basic neural network model, made of fully connected layers, which can be described as a series of functional transformations. First we construct M linear combinations of the input variables $x_1, ..., x_D$ in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2}$$

where $j = 1, ..., M$,and the superscript (1) indicates that the corresponding parameters are in the first 'layer' of the network. We shall refer to the parameters $w_{ji}^{(1)}$ as weights and the parameters $w_{j0}^{(1)}$ as biases. The quantities a_j are known as activations. Each of the a_j's is then transformed using a differentiable, nonlinear activation function $h(\cdot)$ to give

$$z_j = h\left(a_j\right) \tag{3}$$

These quantities correspond to the outputs of the basis functions in (1) that, in the context of neural networks, are called hidden units. The z_j will play the role of inputs of the following layer of the networks, just as the x_i, $i = 1, ...D$ did before. The nonlinear functions $h(\cdot)$ are generally chosen to be sigmoidal functions, such as the logistic sigmoid or the tanh function. In our case, due to their popularity and widespread use in GAN models for Image Generation we will mostly use the ReLU activation function, a choice we will explain later.

We can combine these various stages to give the overall network function that, for sigmoidal output unit activation functions, takes the form

$$y_k(\mathrm{x}, \mathrm{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{4}$$

where the the σ represents the sigmoid function and the set of all weight and bias parameters have been grouped together into a vector w. Thus the neural network model is simply a nonlinear function from a set of input variables $\{x_i\}$ to a set of output variables $\{y_k\}$ controlled by a vector w of adjustable parameters.

This function can be represented in the form of a network diagram as shown in Figure 1.

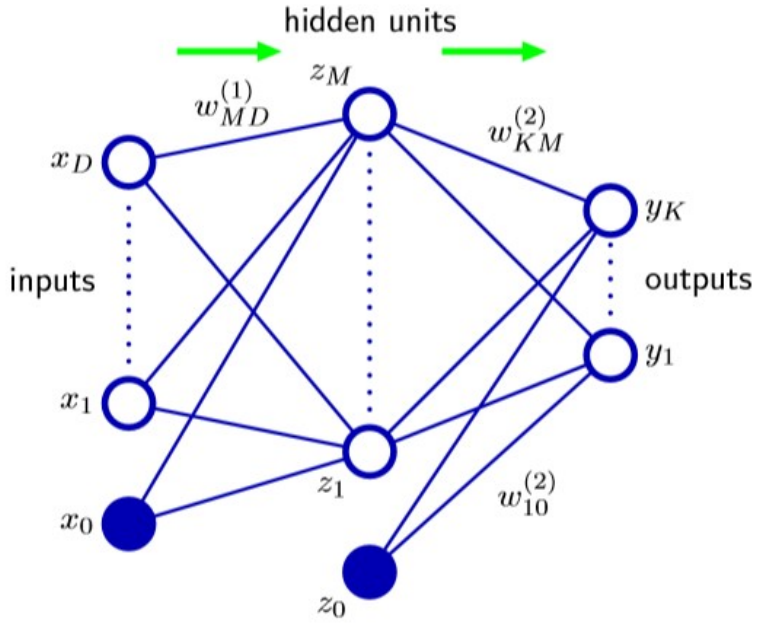


Figure 1: Fully Connected Neural Network Diagram

The process of evaluating (4) can be interpreted as a *forward propagation* of information through the network. The approximation properties of feed-forward networks have been widely studied ([11], [18]) and found to be very general. Neural networks are therefore said to be universal approximators. For example, a two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units. Although such theorems are reassuring, the key problem is how to find suitable parameter values w given a set of training data.

2.1.1 Parameter Optimization

So far, we have viewed neural networks as a general class of parametric nonlinear functions from a vector x of input variables to a vector y of output variables.Given a training set comprising a set of input vectors $\{\mathbf{x}_n\}$, where $n = 1, ..., N$, together with a corresponding set of target vectors $\{t_n\}$, a simple approach to the problem of determining the network parameters w is to either minimize a sum-of-squares error function if we are performing a regression task,

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{y}\left(\mathbf{x}_n, \mathbf{w}\right) - \mathbf{t}_n\|^2 \tag{5}$$

or a binary cross-entropy function if we are performing classification.

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln (1 - y_n)\} \tag{6}$$

First note that if we make a small step in weight space from w to $w + \delta w$ then the change in the error function is $\delta E \simeq \delta \mathbf{w}^{\mathrm{T}} \nabla E(\mathbf{w})$, where the vector $\nabla E(\mathbf{w})$ points in the direction of greatest rate of increase of the error function. Because the error $E(\mathbf{w})$ is a smooth continuous function of $\mathbf{w}$, its smallest value will occur at a point in weight space such that the gradient of the error function vanishes, so that

$$\nabla E(\mathbf{w}) = 0 \tag{7}$$

as otherwise we could make a small step in the direction of $-\nabla E(\mathbf{w})$ and thereby further reduce the error. Our goal is to find a vector w such that $E(w)$ takes its smallest value. However, the error function typically has a highly nonlinear dependence on the weights and bias parameters, and so there will be many points in

weight space at which the gradient vanishes (or is numerically very small). A minimum that corresponds to the smallest value of the error function for any weight vector is said to be a *global minimum*. Any other minima corresponding to higher values of the error function are said to be *local minima*.

For a successful application of neural networks, it may not be necessary to find the global minimum (and in general it will not be known whether the global minimum has been found) but it may be necessary to compare several local minima in order to find a sufficiently good solution.

Because there is clearly no hope of finding an analytical solution to the equation $\nabla E(\mathbf{w}) = 0$ we resort to iterative numerical procedures. The optimization of continuous nonlinear functions is a widely studied problem and there exists an extensive literature on how to solve it efficiently. Most techniques involve choosing some initial value $\mathbf{w}^{(0)}$ for the weight vector and then moving through weight space in a succession of steps of the form

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{8}$$

where τ labels the iteration step. Different algorithms involve different choices for the weight vector update $\Delta\mathbf{w}^{(\tau)}$. Many algorithms make use of gradient information and therefore require that, after each update, the value of $\nabla E(\mathbf{w})$ is evaluated at the new weight vector $\mathbf{w}^{(\tau+1)}$. The simplest approach to using gradient information is to choose the weight update to comprise a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla E\left(\mathbf{w}^{(\tau)}\right) \tag{9}$$

where the parameter $\eta > 0$ is known as the *learning rate*. After each such update, the gradient is re-evaluated for the new weight vector and the process repeated.

2.1.2 Error Backpropagation

We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a differentiable error function. Many error functions of practical interest, comprise a sum or an average of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}) \tag{10}$$

Here we shall consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function.
In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i \tag{11}$$

where z_i is the activation of a unit, or input, that sends a connection to unit j, and w_{ji} is the weight associated with that connection. The sum in (11) is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h\left(a_j\right) \tag{12}$$

Note that one or more of the variables z_i in the sum in (11) could be an input, and similarly, the unit j in (12) could be an output. For each pattern in the training set, we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (11) and (12). As already mentioned, this process is often called *forward propagation* because it can be regarded as a forward flow of information through the network. Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input data point. First we note that E_n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{13}$$

We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{14}$$

where the δ's are often referred to as errors for reasons we shall see shortly. Using (11), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{15}$$

Substituting (14) and (15) into (13), we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{16}$$

Equation (16) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (16). For the output units, we have, in the case of a sum-of-squares error function

$$\delta_k = y_k - t_k \tag{17}$$

This δ_k will depend upon the error function but can be analytically calculated. To evaluate the $\delta's$ for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{18}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 2. If we now substitute the definition of δ given by (14) into (18), and make use of (11)

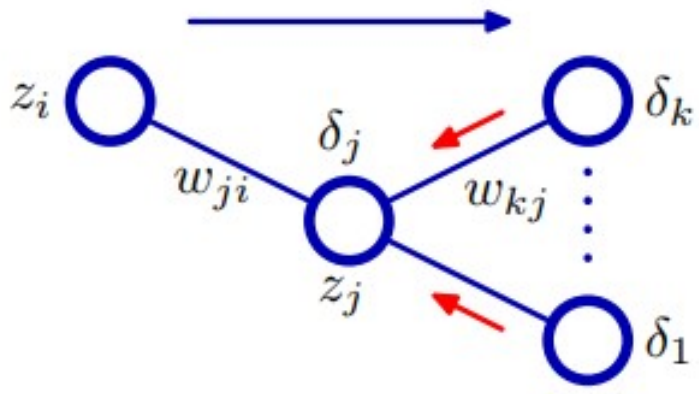


Figure 2: Illustration of the calculation of δ_j for hidden unit j
For hidden unit j δ_j is calculated by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

and (12), we obtain the following *backpropagation* formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{19}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the $\delta's$ backwards from units higher up in the network, as illustrated in Figure 2. Because we already know the values of the δ's for the output units, it follows that by recursively applying (19) we can evaluate the δ's for all of the hidden units in a feed-forward network, regardless of its topology. The backpropagation procedure can therefore be summarized as follows.

1. Apply an input vector x_n to the network and forward propagate through the network using (11) and (12) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (17)
3. Backpropagate the δ's using (18) to obtain δ_j for each hidden unit in the network.
4. Use (16) to evaluate the required derivatives.

2.1.3 Optimizers

In a practical setting, weight updates aren't carried out with classic gradient descent, as in 9. Instead, optimizers that use gradient information such as ADAM [24] or RMSProp [42] are employed, as they speed up training considerably.

1. RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class [42]. It is an adaptation of another optimizer, AdaGrad [8]. AdaGrad is an adaptive optimization algorithm. it adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. Let $\left\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(M)}\right\}$ be a batch of m samples with corresponding target variables $\left\{\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, ..., \mathbf{y}^{(M)}\right\}$, let a neural network with weights w be represented as $f\left(x^{(i)}; w\right)$, then the Adagrad updates are described as follows:

   - Let $\mathbf{g_{t,i}} = \frac{1}{m}\nabla_{w_{t,i}} \sum_i L\left(f\left(\mathbf{x}^{(i)}; \boldsymbol{w_{t,i}}\right), \mathbf{y}^{(i)}\right)$
   - $w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}+\epsilon}} \cdot g_{t,i}$

   Where L is the network loss function, $G_t \in \mathbb{R}^{d\times d} = \{G_{t,ii}\}$ is a diagonal matrix where each diagonal element i, is the sum of the squares of the gradients w.r.t. w_i up to time step t, while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. A value of $\eta = 0.01$ is usually chosen. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

   Instead of inefficiently storing all previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average in a RMSPROP update $E\left[g^2\right]_t$ at time step t then depends only on the previous average and the current gradient:

   - $E\left[g^2\right]_t = 0.9E\left[g^2\right]_{t-1} + 0.1g_t^2$
   - $w_{t+1} = w_t - \frac{\eta}{\sqrt{E[g^2]_t+\epsilon}} g_t$

   A good default value for the learning rate is $\eta = 0.00001$

2. Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like RMSprop, , Adam also keeps an exponentially decaying average of past gradients m_t. It computes the decaying averages of past and past squared gradients as follows:

   - $m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t$
   - $v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2$

   m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1.)
   They counteract these biases by computing bias-corrected first and second moment estimates:

   - $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
   - $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$

   This yields the Adam update rule:

   - $w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$

All the coding of backpropagation, the optimizers and all the necessary procedures involved in constructing, training and testing neural networks can be easily used with the help of auto-differentiation libraries such as Tensorflow [1] (or a higher level API such as Keras [5]) or Pytorch [35].

2.1.4 ReLU and LeakyReLU

ReLU is an acronym for Rectified Linear Unit, and is a type of activation function. Mathematically, it is defined as $y = max(0, x)$.

The ReLU activation function was proposed by Nair and Hinton 2010 (see [33]), and ever since, has been widely

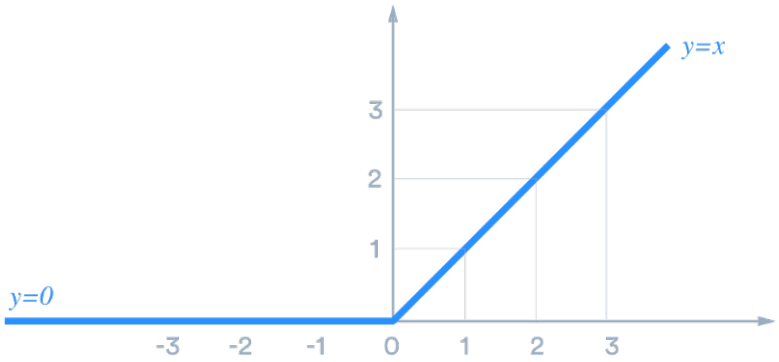


Figure 3: ReLU plot

used in deep learning applications, with state-of-the-art results. It offers better performance and generalization in deep learning compared to the Sigmoid and tanh activation functions [46], in addition to being faster to compute.

The ReLU however does have a significant limitation. It may cause some gradients to vanish because of the flat slope of its negative side, which leads to some neurons being dead,causing the weight updates not to activate in future data points, thereby hindering learning as dead neurons give zero activation [25]. To resolve the dead neuron issues, the leaky ReLU was proposed.

Defined as $f_\alpha(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$ Leaky ReLUs allow a small, positive gradient when the unit is not active, facilitating training and the propagation of information through the network.

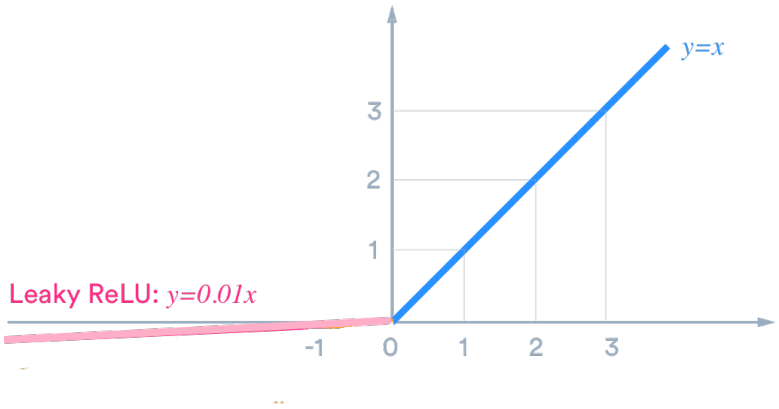


Figure 4: LeakyReLU plot

2.1.5 Batch Normalization

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization.

This phenomenon is known as *Internal Covariance Shift* (see [19]), and can be addressed by normalizing layer inputs.

Note that simply normalizing each input of a layer may change what the layer can represent. For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, Batch Normalization is designed so that *the transformation inserted in the network can represent the identity transform.* To accomplish this, for each activation $x^{(k)}$ a pair of parameters $\gamma^{(k)}, \beta^{(k)}$, which scale and shift the normalized value are learned. The process carried out in each Batch Normalization layer is detailed in **Algorithm 1**.

Input: Hidden Layer Outputs $\{x_{1\ldots x_m}\}$
Parameters to be learned: γ, β
Output: Normalized Inputs for the next layer, $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch

These parameters are learned along with the original model parameters through gradient descent, and restore the representation power of the network. Indeed, by setting $\gamma^{(k)} = \sqrt{\mathrm{Var}\left[x^{(k)}\right]}$ and $\beta^{(k)} = \mathrm{E}\left[x^{(k)}\right]$, we could recover the original activations, if that were the optimal thing to do.

Each normalized activation $\widehat{x}^{(k)}$ can be viewed as an input to a sub-network composed of the linear transform $y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$, followed by the other layers of the original network. As all these sub-networks of the Batch-Normalization layers have fixed means and variances, the introduction of normalized inputs accelerates the training of the sub-network and, consequently, the network as a whole.

2.1.6 Discrete Convolutions

Neural Networks can be generalized from the simple structure described in 2.1 and described as a series of *affine transformations* and other differentiable operations. In an affine transformation a vector is received as input and is multiplied with a matrix to produce an output (to which a bias vector is usually added before passing the result through an activation function). Images, sound clips and in our case financial time series have an intrinsic structure. In the case of time series, the temporal dimension is the most crucial piece of information that should be captured by the network. This is where discrete convolutions come into play.

A discrete convolution is a linear transformation that preserves this notion of ordering. It is sparse (only a few input units contribute to a given output unit) and reuses parameters (the same weights are applied to multiple locations in the input). A discrete convolution operator can be visualized as a kernel of learned weigts w sliding accros an input feature map (see Figure 5 for an illustration) but it actually is a sparse matrix with components determined by the kernel values.

Discrete Convolutions can be cleverly inverted to perform "Deconvolutions" or rather, Transposed Convolutions. The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution. Transposed convolutions work by swapping the forward and backward passes of a convolution. One way to put it is to note that the kernel defines a convolution, but whether it's a direct convolution or a transposed convolution is determined by how the forward and backward passes are computed.

For more details on Convolutions, see [9].

2.1.7 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of neural networks in which a series of Convolutions and Transposed Convolutions are carried out in order to map inputs to outputs. They have been widely studied in their applications regarding image-related problems [36]. CNNs have been successful in identifying faces, objects and traffic signs apart, powering vision in robots and self-driving cars among others. In this work we

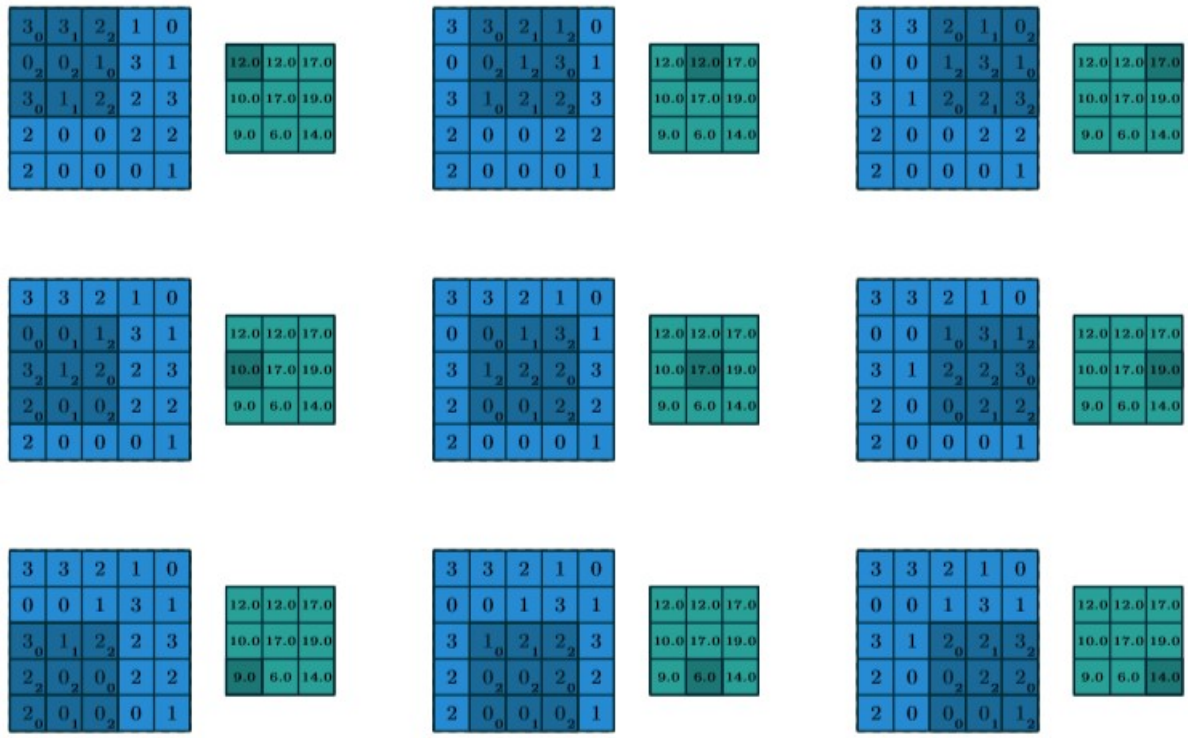


Figure 5: Illustration of a discrete convolution
The light blue grid is called the input feature map. To keep the drawing simple, a single input feature map is represented, but it is not uncommon to have multiple feature maps stacked one onto another. A kernel (shaded area) of values slides across the input feature map. At each location, the product between each element of the kernel and the input element it overlaps is computed and the results are summed up to obtain the output in the current location. The procedure can be repeated using different kernels to form as many output feature maps as desired. The final outputs of this procedure are called output feature maps (green grid).

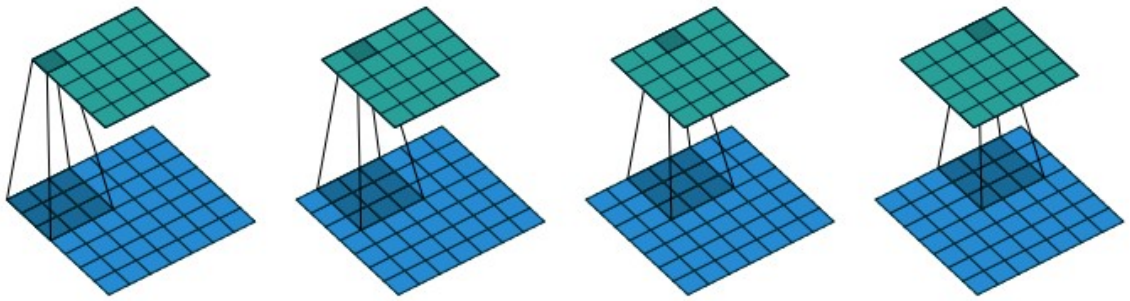


Figure 6: Illustration of a transposed convolution
The transpose of convolving a 3×3 kernel over a 5×5 input using full padding and unit strides.

use Convolutional Networks to process time series data and capture its geometric and time-dependent properties.

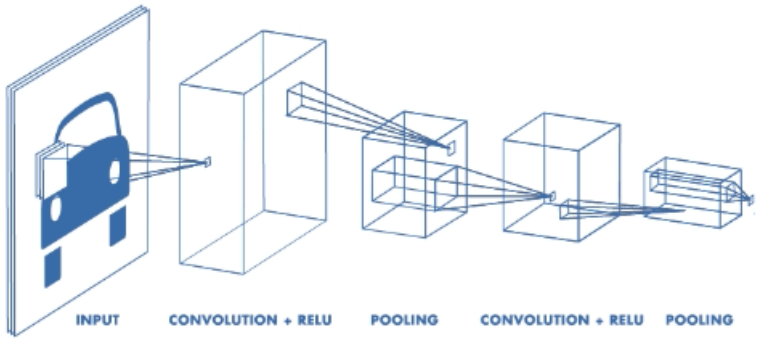


Figure 7: Visualization of a Convolutional Network

2.2 Generative Adversarial Networks

2.2.1 Introduction to Generative Modelling

Generative adversarial networks are an example of *generative models*. The term refers to any model that takes a training set, consisting of samples drawn from a distribution $\mathbb{P}_{data}$, and learns to represent an estimate of that distribution somehow. The result is a probability distribution $\mathbb{P}_{model}$.

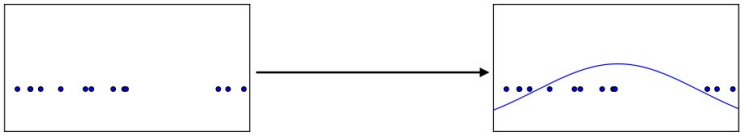


Figure 8: Density Estimation Example
Some generative models perform density estimation. These models take a training set of examples drawn from an unknown data-generating distribution p_{data} and return an estimate of that distribution. The estimate p_{model} can be evaluated for a particular value of $\mathbf{x}$ to obtain an estimate $p_{\text{model}}(\boldsymbol{x})$ of the true density $p_{\text{data}}(\boldsymbol{x})$. This figure illustrates the process for a collection of samples of one-dimensional data and a Gaussian model.

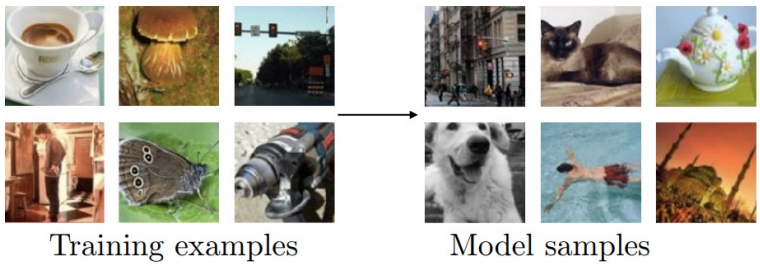


Figure 9: Generative Model Concept
Some generative models are able to generate samples from the model distribution. In this illustration of the process, we show samples from the ImageNet ([38]) dataset. An ideal generative model would be able to train on examples as shown on the left and then create more examples from the same distribution as shown on the right. At present, generative models are not yet advanced enough to do this correctly for ImageNet, so for demonstration purposes this figure uses actual ImageNet data to illustrate what an ideal generative model would produce.

In some cases, the model estimates $\mathbb{P}_{model}$ explicitly, as shown in Figure 8 . In other cases, the model is only able to generate samples from $\mathbb{P}_{model}$, as shown in Figure 9. Some models are able to do both. GANs focus

primarily on sample generation, though it is possible to design GANs that can do both. For more information on other kinds of generative models, see [12].

2.2.2 The GAN framework

The basic idea of GANs is to set up a game between two players. One of them is called the *generator*. The generator creates samples that are intended to come from the same distribution as the training data. The other player is the *discriminator*. The discriminator examines samples to determine whether they are real or fake. The discriminator learns using traditional supervised learning techniques, dividing inputs into two classes (real or fake).

Imagine we want to have a model that can generate images of realistic hand-written numbers, as in the MNIST [26] dataset. The generator would start by producing blurry images that the discriminator would easily classify as fake. Through training however, the generator would slowly learn how to make realistic hand-written numbers that can fool the discriminator . The process is illustrated in Figure 10.

The two players in the game are represented by two functions, each of which is differentiable both with respect to its inputs and with respect to its parameters. The discriminator is a function D that takes x as input and uses $w^{(D)}$ (the discriminator network weights) as parameters. The generator is defined by a function G that takes $\mathbf{z}$ (usually a random normal noise vector) as input and and uses $w^{(G)}$ (the generator network weights) as parameters.

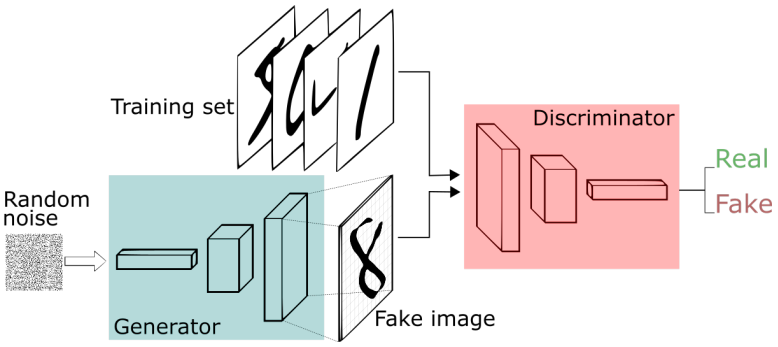


Figure 10: Generative Model Diagram

Both players have cost functions that are defined in terms of both players' parameters. The discriminator wishes to minimize $L^{(D)}\left(w^{(D)}, w^{(G)}\right)$ and must do so while controlling only $w^{(D)}$. The generator wishes to minimize $L^{(G)}\left(w^{(D)}, w^{(G)}\right)$ and must do so while controlling only $w^{(G)}$. Because each player's cost depends on the other player's parameters, but each player cannot control the other player's parameters, this scenario is most straightforward to describe as a game rather than as an optimization problem.

The solution to an optimization problem is a (local) minimum, a point in parameter space where all neighboring points have greater or equal cost. The solution to a game is a Nash equilibrium. Here, we use the terminology of local differential Nash equilibria (see, [37]).). In this context, a Nash equilibrium is a tuple $\left(w^{(D)}, w^{(G)}\right)$ that is a local minimum of $L^{(D)}$ w.r.t $w^{(D)}$ and a local minimum of $L^{(G)}$ w.r.t $w^{(G)}$.

2.2.3 The GAN Generator

The generator is simply a differentiable function G. When $\mathbf{z}$ is sampled from some simple prior distribution $\mathbb{P}_z$, (normally an easy-to-sample distribution such as $\mathcal{N}(0,1)$), $G(\mathbf{z})$ yields a sample of x drawn from $\mathbb{P}_{model}$. Typically, a deep neural network is used to represent G. There are very few restrictions on the design of the generator network. If we want $\mathbb{P}_{model}$ to have full support on x space we need the dimension of $G(\mathbf{z})$ to be at least as large as the dimension of x, and G must be differentiable, but those are the only requirements.

2.2.4 The GAN Discriminator

Similarly, the GAN discriminator is a differentiable function D, whose goal is to classify the real and fake samples accurately. it is also typically represented with a deep neural network D and there are very few restrictions on its construction, it must only take real and fake samples as input and output a score $D(x) \in [0,1]$ and be differentiable.

2.2.5 GAN Training Process

The GAN training process consists of simultaneous Stochastic Gradient Descent Updates. On each step, two minibatches are sampled: a minibatch of $\mathbf{x}$ values from the dataset and a minibatch of $\mathbf{z}$ values drawn from $\mathbb{P}_z$. Then two gradient steps are made simultaneously: one updating $w^{(D)}$ to reduce $L^{(D)}$ and one updating $w^{(G)}$ to reduce $L^{(G)}$. In both cases, it is possible to use the gradient-based optimization algorithm of your choice (such as Adam or RMSProp, see 2.1.3). Running more steps of one player than the other, as in training the discriminator for 5 batches for every update of the generator, is also a possibility and is used in many GAN versions, see [3].

2.2.6 GAN Discriminator Cost Function

All of the different standard GAN setups use the same cost for the discriminator, $L^{(D)}$. They differ only in terms of the cost used for the generator, $L^{(G)}$.
The cost used for the discriminator is:

$$L^{(D)}\left(w^{(D)}, w^{(G)}\right) = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{ data}}} \log D(\boldsymbol{x}) - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}} \log(1 - D(G(z))) \tag{20}$$

This is just the standard cross-entropy cost that is minimized when training a standard binary classifier with a sigmoid output. The only difference is that the classifier is trained on two minibatches of data; one coming from the dataset, where the label is 1 for all examples, and one coming from the generator, where the label is 0 for all examples.

2.2.7 GAN Generator Cost Function

So far we have specified the cost function for only the discriminator. A complete specification of the GAN setup requires that we specify a cost function also for the generator.

The simplest GAN version is a zero-sum game, in which the sum of all player's costs is always zero. In this version of the GAN,

$$L^{(G)} = -L^{(D)} \tag{21}$$

Initially suggested in [14], this setup has been shown to be very unstable (see [2]), GANs training is particularly complex and many things can go awry. In the following sections we will detail some of the pitfalls that can lead to an unsuccessful training of GANs as well as the changes carried out to solve them. These changes imply modifying the cost functions as we will see in the upcoming sections.

2.2.8 Non-convergence in GANs

Most deep models are trained using an optimization algorithm that seeks out a low value of a cost function. While many problems can interfere with optimization, optimization algorithms usually make reliable downhill progress. GANs however, require finding the equilibrium to a game with two players. Even if each player successfully moves downhill on that player's update, the same update might move the other player uphill. Sometimes the two players eventually reach an equilibrium, but in other scenarios they repeatedly undo each others' progress without arriving anywhere useful.

Simultaneous gradient descent converges for some games but not all of them. In the case of the minimax GAN game, [13] showed that simultaneous gradient descent converges if the updates are made in function space. In practice, the updates are made in parameter space, so the convexity properties that the proof relies on do not apply. Currently, there is neither a theoretical argument that GAN games should converge when the updates are made to parameters of deep neural networks, nor a theoretical argument that the games should not converge. Probably the most common form of harmful non-convergence encountered in the GAN game is mode collapse.

2.2.9 Mode Collapse

Mode collapse, also known as the **Helvetica scenario**, is a problem that occurs when the generator learns to map several different input $\mathbf{z}$ values to the same output point. In practice, complete mode collapse is rare, but partial mode collapse is common. Partial mode collapse refers to scenarios in which the generator makes for example, multiple images that contain the same color or texture themes, or multiple images containing different views of the same dog. The mode collapse problem is illustrated in Figure 11.
The mode collapse problem is probably among the most important issues with GANs that researchers have attempted to address in the recent years.

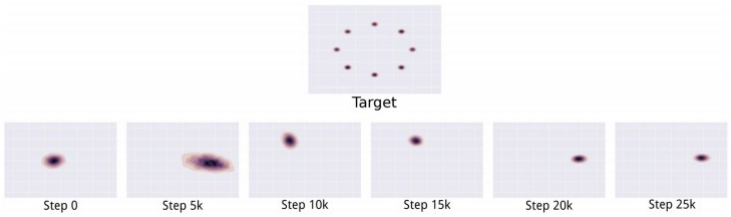


Figure 11: Mode Collapse Illustration
An illustration of the mode collapse problem on a two-dimensional toy dataset. In the top row, we see the target distribution $\mathbb{P}_{data}$ that the model should learn. It is a mixture of Gaussians in a two-dimensional space. In the lower row, we see a series of different distributions learned over time as the GAN is trained. Rather than converging to a distribution containing all of the modes in the training set, the generator only ever produces a single mode at a time, cycling between different modes as the discriminator learns to reject each one. Images from [31].

2.2.10 Theoretical Analysis of GAN Training

Definition 2.1 (Kullblack Leiber Divergence)**.** Let $\mathbb{P}$ and $\mathbb{Q}$ be distributions of two different absolutely continuous random variables with probability densities p and q. Then their Kullblack-Leibler (KL) divergence is defined to be:

$$D_{\mathrm{KL}}(\mathbb{P}\|\mathbb{Q}) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \tag{22}$$

Regarding GANs, this cost function has the good property that it has a unique minimum at $p_{model} = p_{data}$, and it doesn't require knowledge of the unknown p_{data} to estimate it (only samples). However, it is interesting to see how this divergence is not symmetrical between $\mathbb{P}_{data}$ and $\mathbb{P}_{model}$:

- If $p_{data}(x) > p_{model}(x)$, then x is a point with higher probability of coming from the data than being a generated sample. This is the core of the phenomenon commonly described as 'mode dropping': when there are large regions with high values of p_{data} , but small or zero values in p_{model}. It is important to note that when $p_{data}(x) > 0$ but $p_{model}(x) \to 0$, the integrand inside the KL grows quickly to infinity,

meaning that this cost function assigns an extremely high cost to a generator's distribution not covering parts of the data.

- If $p_{data}(x) < p_{model}(x)$, then x has low probability of being a data point, but high probability of being generated by our model. This is the case when we see our generator outputting a sample that doesn't look real. In this case, when $p_{data} \to 0$ and $p_{model}(x) > 0$, we see that the value inside the KL goes to 0, meaning that this cost function will pay extremely low cost for generating fake looking samples.

Definition 2.2 (Jensen-Shannon Divergence)**.** Let $\mathbb{P}$ and $\mathbb{Q}$ be distributions of two different absolutely continuous random variables with probability densities p and q. Then their Jensen-Shannon (JS) divergence is defined to be:

$$D_{JS}(p\|q) = \frac{1}{2}D_{KL}\left(p\|\frac{p+q}{2}\right) + \frac{1}{2}D_{KL}\left(q\|\frac{p+q}{2}\right) \tag{23}$$

The Jensen-Shannon divergence is clearly symmetric and is deeply connected to GAN training as we will see.

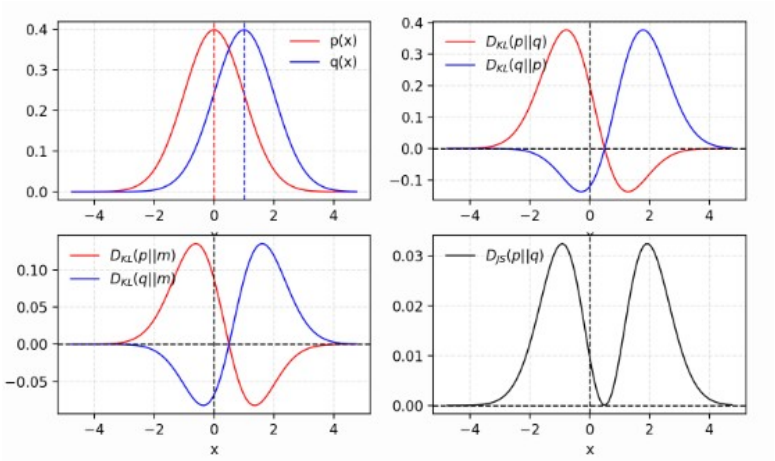


Figure 12: Jensen Shannon Divergence Illustration
Given two Gaussian distributions, p with mean=0 and std=1 and q with mean=1 and std=1. The average of two distributions is labelled as $m = (p+q)/2$. KL divergence D_{KL} is asymmetric but JS divergence D_{JS} is symmetric.

Remember how the loss functions of the discriminator and the generator are defined in (20) and (21). When combining both functions together, D and G are playing a **minimax game** in which we optimize the following loss function:

$$\begin{aligned}\min_G \max_D L(D,G) &= \mathbb{E}_{x\sim\mathbb{P}_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim\mathbb{P}_z(z)}[\log(1-D(G(z)))] \\ &= \mathbb{E}_{x\sim\mathbb{P}_{data}(x)}[\log D(x)] + \mathbb{E}_{x\sim\mathbb{P}_{model}(x)}[\log(1-D(x)]\end{aligned} \tag{24}$$

Note that $\mathbb{E}_{x\sim\mathbb{P}_{data}(x)}[\log D(x)]$ has no impact on G during gradient descent updates (i.e. G is not influenced by real data).
Let's now examine what is the optimal value for D:

$$L(G,D) = \int_x \left(p_{data}(x)\log(D(x)) + p_{model}(x)\log(1-D(x))\right) dx \tag{25}$$

lets us label

$$\tilde{x} = D(x), A = p_{data}(x), B = p_{model}(x) \tag{26}$$

Then our integrand becomes

$$\begin{aligned} f(\tilde{x}) &= A\log\tilde{x} + B\log(1-\tilde{x}) \\ \frac{df(\tilde{x})}{d\tilde{x}} &= A\frac{1}{\tilde{x}} - B\frac{1}{1-\tilde{x}} \\ &= \left(\frac{A}{\tilde{x}} - \frac{B}{1-\tilde{x}}\right) \\ &= \frac{A-(A+B)\tilde{x}}{\tilde{x}(1-\tilde{x})} \end{aligned} \tag{27}$$

Thus, setting $\frac{df(\tilde{x})}{d\tilde{x}} = 0$ we get the best value of the discriminator:

$$D^*(x) = \tilde{x}^* = \frac{A}{A+B} = \frac{p_{data}(x)}{p_{data}(x) + p_{model}(x)} \in [0,1] \tag{28}$$

Once the generator is trained to its optimal, p_{model} gets very close to p_{data}. When $p_{model} = p_{data}$, $D^*(x)$ becomes $1/2$ .This is illustrated in Figure 13.

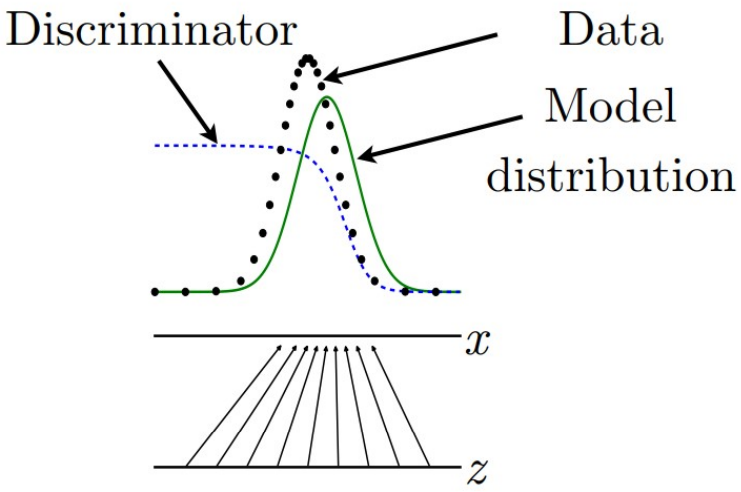


Figure 13: Density Ratio Estimation
In this example, we assume that both z and x are one dimensional for simplicity. The mapping from z to x (shown by the black arrows) is non-uniform so that $p_{\text{model}}(x)$ (shown by the green curve) is greater in places where z values are brought together more densely. The discriminator (dashed blue line) estimates the ratio between the data density (black dots) and the sum of the data and model densities. Wherever the output of the discriminator is large, the model density is too low, and wherever the output of the discriminator is small, the model density is too high. The generator can learn to produce a better model density by following the discriminator uphill; each $G(z)$ value should move slightly in the direction that increases $D(G(z))$. Figure reproduced from [13].

When both G and D are at their optimal values, we have $p_{model} = p_{data}$ and $D^*(x) = 1/2$ and the loss function becomes:

$$\begin{aligned} L(G, D^*) &= \int_x \left(p_{data}(x) \log\left(D^*(x)\right) + p_{model}(x) \log\left(1 - D^*(x)\right)\right) dx \\ &= \log\frac{1}{2} \int_x p_{data}(x) dx + \log\frac{1}{2} \int_x p_{model}(x) dx \\ &= -2\log 2 \end{aligned} \tag{29}$$

If we calculate the JS divergence between $\mathbb{P}_{model}$ and $\mathbb{P}_{data}$ we have that

$$\begin{aligned} D_{JS}\left(\mathbb{P}_{data} \| \mathbb{P}_{model}\right) =& \frac{1}{2} D_{KL}\left(p_{data} \| \frac{p_{data} + p_{model}}{2}\right) + \frac{1}{2} D_{KL}\left(p_{model} \| \frac{p_{data} + p_{model}}{2}\right) \\ =& \frac{1}{2}\left(\log 2 + \int_x p_{data}(x) \log \frac{p_{data}(x)}{p_{data}(x) + p_{model}(x)} dx\right) + \\ & \frac{1}{2}\left(\log 2 + \int_x p_{model}(x) \log \frac{p_{model}(x)}{p_{data}(x) + p_{model}(x)} dx\right) \\ =& \frac{1}{2}\left(\log 4 + L(G, D^*)\right) \end{aligned} \tag{30}$$

Thus,

$$L(G, D^*) = 2 D_{JS}\left(\mathbb{P}_{data} \| \mathbb{P}_{model}\right) - 2\log 2 \tag{31}$$

Essentially the loss function of GAN quantifies the similarity between the generative data distribution $\mathbb{P}_{model}$ and the real sample distribution $\mathbb{P}_{data}$ by JS divergence when the discriminator is optimal. The best G^* that replicates the real data distribution leads to the minimum $L(G^*, D^*) = -2\log 2$ which is aligned with equations above.

However, in practice, if we just train D till convergence, its error will go to 0, pointing to the fact that the JS Divergence between them is maxed out (It is easy to see that $0 \leq D_{JS}(p\|q) \leq \ln(2)$). The only way this can happen is if the distributions are not continuous (meaning their densities are not absolutely continuous functions), or they have disjoint supports. One possible cause for the distributions not to be continuous is if their supports lie on low dimensional manifolds. There is strong empirical and theoretical evidence to believe that $\mathbb{P}_{data}$ is indeed extremely concentrated on a low dimensional manifold for many datasets (see [34]).

Arjovsky and Bottou (2017) discussed the problem of the supports of p_{data} and p_{model} lying on low dimensional manifolds and how it contributes to the instability of GAN training thoroughly in [2]. Because both p_{data} and p_{model} rest in low dimensional manifolds, they are almost certainly gonna be disjoint (See Fig 14). When they have disjoint supports, we are always capable of finding a perfect discriminator that separates real and fake samples 100% correctly.

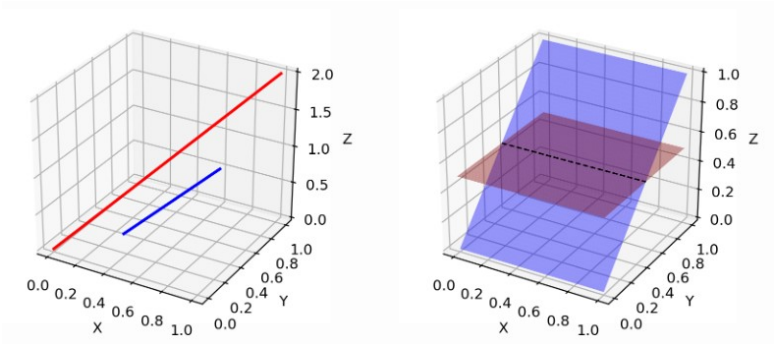


Figure 14: Manifold Overlap Illustration
Low dimensional manifolds in high dimension spaces can hardly have overlaps. (Left) Two lines in a three-dimension space. (Right) Two surfaces in a three-dimension space.

When the discriminator is perfect, we are guaranteed with $D(x) = 1, \forall x \in \mathbb{P}_{data}$ and $D(x) = 0, \forall x \in \mathbb{P}_{model}$. Therefore the loss function L falls to zero and we end up with no gradient to update the loss during learning iterations. Fig 15 demonstrates an experiment when the discriminator gets better, the gradient vanishes fast.

As a result, training a GAN faces a *dilemma*:

- If the discriminator behaves badly, the generator does not have accurate feedback and the loss function cannot represent the reality.
- If the discriminator does a great job, the gradient of the loss function drops down to close to zero and the learning becomes super slow or even jammed.

This dilemma makes GAN training very tough.

2.2.11 Wasserstein's distance

Definition 2.3. Let $\mathcal{X}$ be a compact metric set. Let Prob($\mathcal{X}$) denote the space of probability measures defined on X. The Earth-Mover (EM) distance or Wasserstein-1 is then defined as

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{32}$$

where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively $\mathbb{P}_r$ and $\mathbb{P}_g$.

Intuitively, $W(\mathbb{P}_r, \mathbb{P}_g)$ indicates how much "mass" must be transported from x to y in order to transform the distributions $\mathbb{P}_r$ into the distribution $\mathbb{P}_g$, thinking of the distributions as "piles of earth".

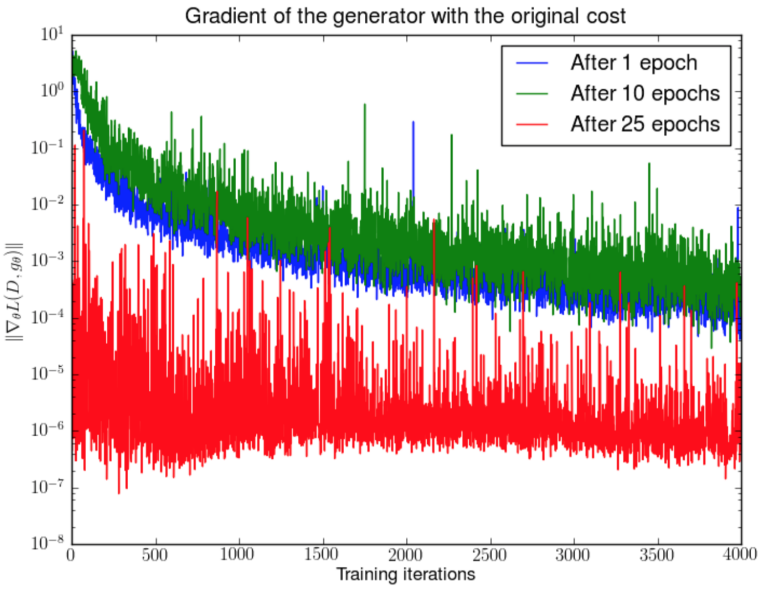


Figure 15: Vanishing Gradient Experiment
First, a DCGAN is trained for 1, 10 and 25 epochs. Then, with the generator fixed, a discriminator is trained from scratch and measure the gradients with the original cost function. We see the gradient norms decay quickly (in log scale), in the best case 5 orders of magnitude after 4000 discriminator iterations. (Image source: [2]

Let $\mu(x)$ and $\nu(x)$ be two probability distributions on X, suppose we want to move the mass of $\mu(x)$ so that it has the shape of $\nu(x)$. Suppose that there is a given cost function

$$c(x, y) \mapsto [0, \infty)$$

that gives the cost of transporting a unit mass from the point x to the point y. A transport plan to move μ into ν can be described by a function $\gamma(x, y)$ which gives the amount of mass to move from x to y. In order to be a transport plan, this function must satisfy

$$\int \gamma\left(x', x\right) \mathrm{d}x' = \nu(x)$$

$$\int \gamma\left(x, x'\right) \mathrm{d}x' = \mu(x)$$

This is equivalent to $\gamma(x, y)$ being a joint probability distribution with marginals μ and ν. The total cost of a transport plan $\gamma(x, y)$ will then be

$$\iint c(x, y)\gamma(x, y)\mathrm{d}x\mathrm{d}y = \int c(x, y)\mathrm{d}\gamma(x, y) \tag{33}$$

As γ is not unique, there must be an infimum to the set of the costs of all transport plans (since all costs are positives). Let Γ be the set of all transport plans (which is the set of all joint probability distribution with marginals μ and ν as mentioned), then the cost of the optimal plan is

$$C = \inf_{\gamma \in \Gamma(\mu, \nu)} \int c(x, y)\mathrm{d}\gamma(x, y) \tag{34}$$

If the function $c(x, y)$ is simply the distance between the two points, then the optimal cost is identical to the definition of the Wasserstein-1 distance.

Let us first look at a simple case where the probability domain is discrete. For example, suppose we have two distributions P and Q, each has four piles of dirt and both have ten shovelfuls of dirt in total. The numbers of shovelfuls in each dirt pile are assigned as follows:

$$P_1 = 3, P_2 = 2, P_3 = 1, P_4 = 4$$
$$Q_1 = 1, Q_2 = 2, Q_3 = 4, Q_4 = 3$$

In order to change P to look like Q, as illustrated in Fig 16 (taken from [45]) , we:

- First move 2 shovelfuls from P_1 to P_2 => (P_1, Q_1) match up.
- Then move 2 shovelfuls from P_2 to P_3 => (P_2, Q_2) match up.
- Finally move 1 shovelfuls from Q_3 to Q_4 => (P_3, Q_3) and (P_4, Q_4) match up.

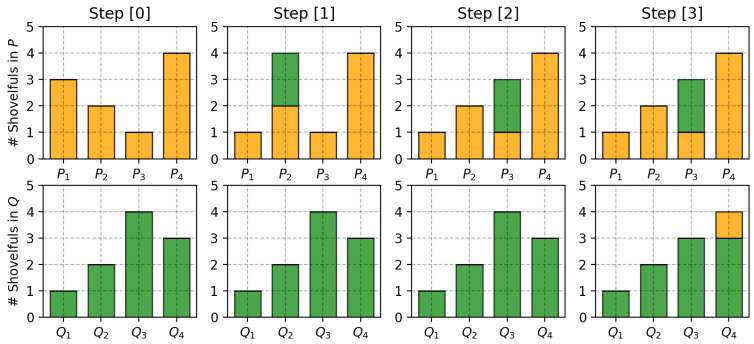


Figure 16: Earth Mover Distance graphic

If we label the cost to pay to make P_i and Q_i match as δ_i, we would have $\delta_{i+1} = \delta_i + P_i - Q_i$ and in the example:

$$\delta_0 = 0$$
$$\delta_1 = 0 + 3 - 1 = 2$$
$$\delta_2 = 2 + 2 - 2 = 2$$
$$\delta_3 = 2 + 1 - 4 = -1$$
$$\delta_4 = -1 + 4 - 3 = 0$$

Finally the Earth Mover's distance is $W = \sum |\delta_i| = 5$

Even when two distributions are located in lower dimensional manifolds without overlaps, the Wasserstein distance can still provide a meaningful and smooth representation of the distance in-between. [3] exemplified the idea with a simple example.

Example 2.1. Suppose we have two probability distributions, P and Q:

$$\forall(x, y) \in P, x = 0 \text{ and } y \sim U(0, 1)$$
$$\forall(x, y) \in Q, x = \theta, 0 \leq \theta \leq 1 \text{ and } y \sim U(0, 1)$$

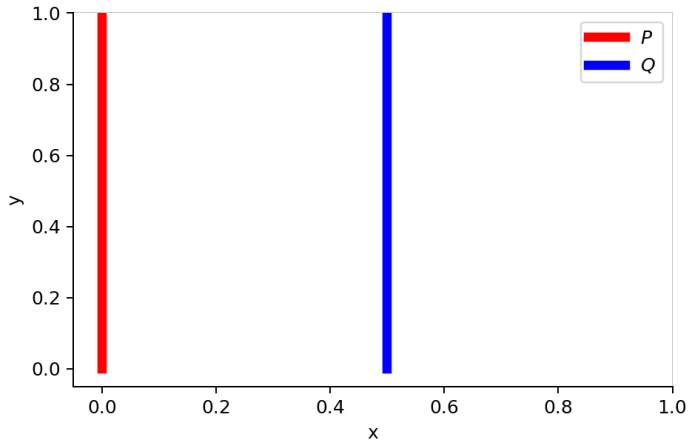


Figure 17: Wasserstein Distance Example
There is no overlap between P and Q when $\theta \neq 0$

When $\theta \neq 0$:

$$D_{KL}(P\|Q) = \sum_{x=0, y-U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty$$
$$D_{KL}(Q\|P) = \sum_{x=0, y-U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty$$
$$D_{JS}(P, Q) = \frac{1}{2}\left(\sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{1/2} + \sum_{x=0, y-U(0,1)} 1 \cdot \log \frac{1}{1/2}\right) = \log 2$$
$$W(P, Q) = |\theta|$$

But when $\theta = 0$, the two distributions are fully overlapped:

$$D_{KL}(P\|Q) = D_{KL}(Q\|P) = D_{JS}(P, Q) = 0$$
$$W(P, Q) = 0 = |\theta|$$

D_{KL} gives us infinity when two distributions are disjoint. The value of D_{JS} has sudden jump, not differentiable at $\theta = 0$. Only the Wasserstein's metric provides a smooth measure, which allows for a stable learning process using gradient descent.

2.3 Advanced GANs Setups

2.3.1 Wasserstein GAN

It is intractable to exhaust all the possible joint distributions in $\Pi\left(\mathbb{P}_{data}, \mathbb{P}_{model}\right)$ to compute $\inf_{\gamma \sim \Pi(p_{data}, p_{model})}$ in 32, thus the authors in [3] proposed a transformation of the formula based on the Kantorovich-Rubinstein duality (see [39]) to:

$$W\left(\mathbb{P}_{data}, \mathbb{P}_{model}\right) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim \mathbb{P}_{data}}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_{model}}[f(x)] \tag{35}$$

The function f in the new form of Wasserstein metric is demanded to satisfy $\|f\|_L \leq K$ meaning it should be K-Lipschitz continuous.

A real-valued function $f : \mathbb{R} \to \mathbb{R}$ is called K -Lipschitz continuous if there exists a real constant $K \geq 0$ such that, for all $x_1, x_2 \in \mathbb{R}$

$$\left|f\left(x_1\right) - f\left(x_2\right)\right| \leq K\left|x_1 - x_2\right| \tag{36}$$

Here K is known as a Lipschitz constant for function $f(.)$. Functions that are everywhere continuously differentiable are Lipschitz continuous, because the derivative, estimated as $\frac{|f(x_1)-f(x_2)|}{|x_1-x_2|}$, has bounds. However, a Lipschitz continuous function may not be everywhere differentiable, such as $f(x) = |x|$.

Suppose this function f comes from a family of K-Lipschitz continuous functions, $\{f_w\}_{w \in W}$, parameterized by w. In the modified Wasserstein-GAN, the "discriminator" model is used to learn w to find a good f_w and the loss function is configured as measuring the Wasserstein distance between $\mathbb{P}_{data}$ and $\mathbb{P}_{model}$.

$$L\left(\mathbb{P}_{data}, \mathbb{P}_{model}\right) = W\left(p_{data}, p_{model}\right) = \max_{w \in W} \mathbb{E}_{x \sim p_{data}}\left[f_w(x)\right] - \mathbb{E}_{z \sim p_{data}(z)}\left[f_w\left(g_\theta(z)\right)\right] \tag{37}$$

Thus the “discriminator” is not a direct critic of telling the fake samples apart from the real ones anymore. Instead, it is trained to learn a K-Lipschitz continuous function to help compute Wasserstein distance. As the loss function decreases in the training, the Wasserstein distance gets smaller and the generator model’s output grows closer to the real data distribution.

It is not trivial how to maintain the K -Lipschitz continuity of f_w during the training, but [3] proposes a simple solution, after every gradient update, clip the weights w to a small window, such as $[-0.01, 0.01]$, resulting in a compact parameter space W and thus f_w obtains its lower and upper bounds to preserve the Lipschitz continuity.

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator’s parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 18: Wasserstein GAN Pseudocode,image from [3].

Compared to the original GAN algorithm, the WGAN undertakes the following changes:

- After every gradient update on the critic function, clamps the weights to a small fixed range, $[-c, c]$.
- Uses a new loss function derived from the Wasserstein distance. The "discriminator" model does not play as a direct critic but estimates the Wasserstein metric between real and generated data distribution.
- Empirically the authors recommended RMSProp optimizer on the critic, rather than a momentum based optimizer such as Adam which could cause instability in the model training, although this is not theoretically justified.

However, as the authors point out: "Weight clipping is a clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality. If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used (such as in RNNs) ... and we stuck with weight clipping due to its simplicity and already good performance".

To sum up, clipping is simple but it introduces some problems. The model may still produce poor quality samples (images in the paper) and not converge, in particular when the hyper parameter c is not tuned correctly. The weight clipping behaves as a weight regulation. It reduces the capacity of the model f and limits the capability to model complex functions. This is improved upon with the following algorithm.

2.3.2 Wasserstein GAN with Gradient Penalty

Proposed in [16], this version of Wasserstein's GAN uses gradient penalty instead of the weight clipping to enforce the Lipschitz constraint. This is because a differentiable function f is 1-Lipschtiz *if and only if* it has gradients with norm at most 1 almost everywhere. In order to circumvent tractability issues, the authors enforce a soft version of the constraint with a penalty on the gradient norm for random samples, $\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}$. The new objective becomes:

$$L = \mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}} \sim \mathbb{P}_g} [D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \mathbb{P}_r} [D(\boldsymbol{x})] + \lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}} \sim \mathbb{P}_{\hat{\boldsymbol{x}}}} \left[\left(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1 \right)^2 \right] \tag{38}$$

Where the first two terms correspond to the original critic loss and the last one to the gradient penalty. The authors implicitly define $\mathbb{P}_{\hat{x}}$ sampling uniformly along straight lines between pairs of points sampled from the data distribution $\mathbb{P}_r$ and the generator distribution $\mathbb{P}_g$. This is justified in the paper, along with setting a value of $\lambda = 10$ and advising against batch normalization (simply normalizing the outputs of the hidden layers in the network) in the discriminator network.

WGAN-GP had been observed to present the best convergence behavior out of all the variants of WGAN at the time its paper was published, and it accounts for 200+ citations. As explained however in [21], WGAN-GP is part of the Integral Probability metrics (IPM)-based GANs family and although effective and robust, it is a very computationally expensive algorithm.

2.3.3 Relativistic Standard GAN

In this section we are going to use the following notation:

- $D(x) = \text{sigmoid}(C(x))$ is the discriminator output, where $C(x)$ is the non-transformed discriminator output (which we call critic as per [3])
- We refer to a loss function as saturating if when real and fake data are perfectly classified, the loss has zero gradient. (21) is one example of a saturating loss.
- We also refer to the standard GAN setup presented with loss functions as in (21) and (20) as SGAN.

Definition 2.4 (Integral probability metrics)**.** IPMs are statistical divergences represented mathematically as:

$$IPM_F(\mathbb{P} \| \mathbb{Q}) = \sup_{C \in \mathcal{F}} \mathbb{E}_{x \sim \mathbb{P}}[C(x)] - \mathbb{E}_{x \sim \mathbb{Q}}[C(x)] \tag{39}$$

where $\mathcal{F}$ is a class of real-valued functions. It can be observed that both discriminator and generator loss functions are unbounded and would diverge to $-\infty$ if optimized directly. However, IPMs assume that the discriminator is of a certain class of functions $\mathcal{F}$ so that it does not grow too quickly which prevents the loss functions from diverging. Each IPM applies a different constraint to the discriminator (e.g., WGAN assumes a Lipschitz D, WGAN-GP assumes that D has gradient norm equal to 1 around real and fake data).

Jolicoeur-Martineau in [21] proposed one of the most insightful changes to the GAN procedure made to date. The paper argues that training the generator should not only increase the probability that fake data is real but also decrease the probability that real data is real. In other words, the key missing property of the Standard GAN is that the probability of real data being real $D(x_{real})$ should decrease as the probability of fake data being real $D(x_{fake})$ increases. It supports this reasoning with the three following arguments:

1. **Prior knowledge argument** : During training, the discriminator is trained in order to accurately classify samples as real or fake. A powerful enough generator would fool the discriminator into classifying all samples as real, but this behavior is illogical considering the *a priori* knowledge that half of the samples in the mini-batch are fake. If the discriminator perceives all samples shown as equally real, it should assume that each sample has probability $\frac{1}{2}$ of being real. However, in SGAN and other non-IPM-based GANs, the generator is simply trained so that the discriminator output of fake data is close to 1.
   Assuming that the generator is trained with a strong learning rate or for many iterations; in addition to both real and fake samples being classified as real, fake samples may appear to be more realistic than real samples, i.e., $C(x_{fake}) > C(x_{real})$ for most x_{real} and x_{fake}. In that case, considering that half of the samples are fake, the discriminator should assign a higher probability of being fake to real samples rather than classify all samples as real.

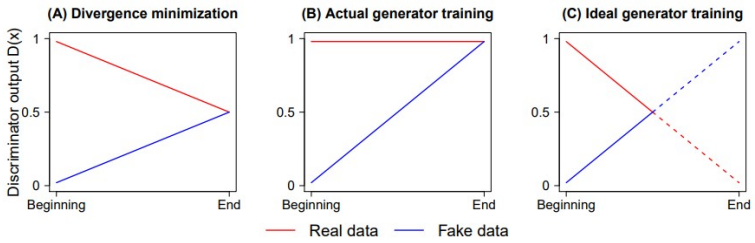


Figure 19: Expected discriminator output
Expected discriminator output of the real and fake data for the a) direct minimization of the Jensen–Shannon divergence, b) actual training of the generator to minimize its loss function, and c) ideal training of the generator to minimize its loss function (lines are dotted when they cross beyond the equilibrium to signify that this may or may not be necessary). Image from [21].

   In summary, by not decreasing $D(x_{real})$ as $D(x_{fake})$ increases, SGAN completely ignores the a priori knowledge that half of the mini-batch samples are fake. Unless one makes the task of the discriminator more difficult (using regularization or lower learning rates), the discriminator does not make reasonable predictions. On the other hand, IPM-based GANs implicitly account for the fact that some of the samples must be fake because they compare how realistic real data is compared to fake data. This provides an intuitive argument to why the discriminator in SGAN (and GANs in general) should depend on both real and fake data.

2. **Divergence minimization argument** : As we have already seen in SGAN, we have that the optimal discriminator loss function is equal to the Jensen–Shannon divergence (see, 31). Assuming an optimal discriminator, if the JSD is minimized ($JSD(\mathbb{P}_{data}\|\mathbb{P}_{model}) = 0$) then $D(x_{real}) = D(x_{fake}) = \frac{1}{2}$ for all $x_{real} \in \mathbb{P}_{data}$ and $x_{fake} \in \mathbb{P}_{model}$ and if it is maximized ($JSD(\mathbb{P}_{data}\|\mathbb{P}_{model}) = \log(2)$) then $D(x_{real}) = 1, D(x_{fake}) = 0$ for all $x_{real} \in \mathbb{P}_{data}$ and $x_{fake} \in \mathbb{P}_{model}$.
   Thus, if we were directly minimizing the divergence from maximum to minimum, we would expect $D(x_{real})$ to smoothly decrease from 1 to .50 for most x_{real} and $D(x_{fake})$ to smoothly increase from 0 to $\frac{1}{2}$ for most x_{fake} (Figure 19a) However, when minimizing the saturating loss in SGAN, we are only increasing $D(x_{fake})$, we are not decreasing $D(x_{real})$ (Figure 19b). Furthermore, we are bringing $D(x_{fake})$ closer to 1 rather than $\frac{1}{2}$. This means that SGAN dynamics are very different from the minimization of the JSD. To bring SGAN closer to divergence minimization, training the generator should not only increase $D(x_{fake})$ but also decrease $D(x_{real})$ (Figure 19c).

3. **Gradient argument** : This last argument is based on the analysis of the gradients for both SGAN and IPM-based GANs. It can be shown that the gradients of the discriminator and generator in non-saturating SGAN are respectively:

$$\nabla_w L_D^{GAN} = -\mathbb{E}_{x_r \sim \mathbb{P}_{data}} \left[(1 - D(x_r)) \nabla_w C(x_r) \right] + \mathbb{E}_{x_f \sim \mathbb{Q}_\theta} \left[D(x_f) \nabla_w C(x_f) \right] \tag{40}$$

$$\nabla_\theta L_G^{GAN} = -\mathbb{E}_{z\sim\mathbb{P}_z}\left[(1 - D(G(z)))\nabla_x C(G(z)) J_\theta G(z)\right] \tag{41}$$

where w and θ are the discriminator and generator weights respectively and J is the Jacobian.
It can also be shown that the gradients of the discriminator and generator in IPM-based GANs are respectively:

$$\nabla_w L_D^{IPM} = -\mathbb{E}_{x_r\sim\mathbb{P}_{data}}\left[\nabla_w C\left(x_r\right)\right] + \mathbb{E}_{x_f\sim\mathbb{Q}_\theta}\left[\nabla_w C\left(x_f\right)\right] \tag{42}$$

$$\nabla_\theta L_G^{IPM} = -\mathbb{E}_{z\sim\mathbb{P}_z}\left[\nabla_x C(G(z)) J_\theta G(z)\right] \tag{43}$$

where $C(x) \in \mathcal{F}$ (the class of functions assigned by the IPM).
From these equations, it can be observed that SGAN would share the same gradient as IPM-based GANs given that:

(a) $D\left(x_r\right) = 0, D\left(x_f\right) = 1$ in the discriminator step of SGAN

(b) $D\left(x_f\right) = 0$ in the generator step of SGAN.

(c) $C(x) \in \mathcal{F}$

Assuming that the discriminator and generator are trained to optimality in each step and that it is possible to perfectly distinguish real from the fake data (strong assumption, but generally true early in training); we have that $D\left(x_r\right) = 1, D\left(x_f\right) = 0$ in the generator step and that $D\left(x_r\right) = 1, D\left(x_f\right) = 1$ in the discriminator step for most x_r and x_f (Figure 19b). Thus, the only missing assumption is that $D\left(x_r\right) = 0$ in the discriminator step.

This means that SGAN could be equivalent to IPM-based GANs, in certain situations, if the generator could indirectly influence $D\left(x_r\right)$. Considering that IPM-based GANs are generally more stable than SGAN, it would be reasonable to expect that making SGAN closer to IPM-based GANs could improve its stability.

In IPMs, both real and fake data equally contribute to the gradient of the discriminator's loss function. However, in SGAN, if the discriminator reaches optimality, the gradient completely ignores real data. This means that if $D\left(x_r\right)$ does not indirectly change when training the discriminator to reduce $D\left(x_f\right)$ (which might happen if real and fake data have different supports or if D has a very large capacity), the discriminator will stop learning what it means for data to be "real" and training will focus entirely on fake data. In which case, fake samples will not become more realistic and training will get stuck. On the other hand if $D\left(x_r\right)$ always decreases when $D\left(x_f\right)$ increases, real data will always be incorporated in the gradient of the discriminator loss function. The author also supports this argument with experimental evidence in the paper [21].

The modifications needed to make discriminator relativistic (i.e., having the output of D depend on both real and fake data) is to sample from real/fake data pairs $\tilde{x} = (x_r, x_f)$ and define it as $D(\tilde{x}) = \text{sigmoid}\left(C\left(x_r\right) - C\left(x_f\right)\right)$

In the paper the author argues that we can interpret this modification in the following way: **the discriminator estimates the probability that the given real data is more realistic than a randomly sampled fake data.**

Similarly, we can define $D_{rev}(\tilde{x}) = \text{sigmoid}\left(C\left(x_f\right) - C\left(x_r\right)\right)$ as the probability that the given fake data is more realistic than a randomly sampled real data. An interesting property of this discriminator is that we do not need to include D_{rev} in the loss function through $\log\left(1 - D_{rev}(\tilde{x})\right)$ because we have that $1 - D_{rev}(\tilde{x}) = 1 - \text{sigmoid}\left(C\left(x_f\right) - C\left(x_r\right)\right) = \text{sigmoid}\left(C\left(x_r\right) - C\left(x_f\right)\right) = D(\tilde{x})$; thus, $\log(D(\tilde{x})) = \log\left(1 - D_{rev}(\tilde{x})\right)$

The discriminator and generator (non-saturating) loss functions of the Relativistic Standard GAN (RSGAN) can be written as:

$$L_D^{RSGAN} = -\mathbb{E}_{(x_r,x_f)\sim(\mathbb{P}_{data},\mathbb{P}_{model})}\left[\log\left(\text{sigmoid}\left(C\left(x_r\right)-C\left(x_f\right)\right)\right)\right] \tag{44}$$

$$L_G^{RSGAN} = -\mathbb{E}_{(x_r,x_f)\sim(\mathbb{P}_{data},\mathbb{P}_{model})}\left[\log\left(\text{sigmoid}\left(C\left(x_f\right)-C\left(x_r\right)\right)\right)\right] \tag{45}$$

More generally, relativistic GANs (with symmetric loss functions, the formulation can be made more general) can be formulated as

$$L_D^{RGAN*} = \mathbb{E}_{(x_r,x_f)\sim(\mathbb{P}_{data},\mathbb{P}_{model})}\left[f_1\left(C\left(x_r\right)-C\left(x_f\right)\right)\right] \tag{46}$$

and

$$L_G^{RGAN*} = \mathbb{E}_{(x_r,x_f)\sim(\mathbb{P}_{data},\mathbb{P}_{model})}\left[f_1\left(C\left(x_f\right)-C\left(x_r\right)\right)\right] \tag{47}$$

Algorithm 1 shows how to train RGANs of this form.

Algorithm 1 Training algorithm for non-saturating RGANs with symmetric loss functions

Require: The number of D iterations n_D ($n_D = 1$ unless one seeks to train D to optimality), batch size m, and functions f which determine the objective function of the discriminator (f is f_1 from equation 10 assuming that $f_2(-y) = f_1(y)$, which is true for many GANs).
while θ has not converged **do**
  for $t = 1, \ldots, n_D$ **do**
    Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}$
    Sample $\{z^{(i)}\}_{i=1}^m \sim \mathbb{P}_z$
    Update w using SGD by ascending with $\nabla_w \frac{1}{m}\sum_{i=1}^m \left[f(C_w(x^{(i)}) - C_w(G_\theta(z^{(i)})))\right]$
  end for
  Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}$
  Sample $\{z^{(i)}\}_{i=1}^m \sim \mathbb{P}_z$
  Update θ using SGD by ascending with $\nabla_\theta \frac{1}{m}\sum_{i=1}^m \left[f(C_w(G_\theta(z^{(i)})) - C_w(x^{(i)}))\right]$
end while

Figure 20: RGAN training pseudocode, image from [21].

2.3.4 Relativistic Average GAN

In [21] Jolicoeur-Martineau argues that although the relative discriminator provides the missing property that we want in GANs (i.e., G influencing $D\left(x_r\right)$) its interpretation is different from the standard discriminator. Rather than measuring "the probability that the input data is real", it is now measuring "the probability that the input data is more realistic than a randomly sampled data of the opposing type (fake if the input is real or real if the input is fake)". To make the relativistic discriminator act more globally, the author proposes the Relativistic average Discriminator (RaD) which compares the critic of the input data to the average critic of samples of the opposite type. The discriminator loss function for this approach can be formulated as:

$$L_D^{\text{RaSGAN}} = -\mathbb{E}_{x_r\sim\mathbb{P}_{data}}\left[\log\left(\overline{D}\left(x_r\right)\right)\right] - \mathbb{E}_{x_f\sim\mathbb{P}_{model}}\left[\log\left(1-\overline{D}\left(x_f\right)\right)\right] \tag{48}$$

where

$$\overline{D}(x) = \begin{cases} \text{sigmoid}\left(C(x) - \mathbb{E}_{x_f\sim\mathbb{P}_{model}} C\left(x_f\right)\right) & \text{if } x \text{ is real} \\ \text{sigmoid}\left(C(x) - \mathbb{E}_{x_r\sim\mathbb{P}_{data}} C\left(x_r\right)\right) & \text{if } x \text{ is fake.} \end{cases} \tag{49}$$

RaD has a more similar interpretation to the standard discriminator than the relativistic discriminator. With RaD, **the discriminator estimates the probability that the given real data is more realistic than fake data, on average**. This approach is also favored because it has $O(m)$ complexity, unlike other approaches explored in [21].

Algorithm 2 Training algorithm for non-saturating RaGANs

Require: The number of D iterations n_D ($n_D = 1$ unless one seek to train D to optimality), batch size m, and functions f_1 and f_2 which determine the objective function of the discriminator (see equation 10).

while θ has not converged **do**

  for $t = 1, \dots, n_D$ **do**

    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}$

    Sample $\{z^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_z$

    Let $\overline{C_w(x_r)} = \frac{1}{m}\sum_{i=1}^{m} C_w(x^{(i)})$

    Let $\overline{C_w(x_f)} = \frac{1}{m}\sum_{i=1}^{m} C_w(G_\theta(z^{(i)}))$

    Update w using SGD by ascending with

    $$\nabla_w \frac{1}{m}\sum_{i=1}^{m}\left[f_1(C_w(x^{(i)}) - \overline{C_w(x_f)}) + f_2(C_w(G_\theta(z^{(i)})) - \overline{C_w(x_r)})\right]$$

  end for

  Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}$

  Sample $\{z^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_z$

  Let $\overline{C_w(x_r)} = \frac{1}{m}\sum_{i=1}^{m} C_w(x^{(i)})$

  Let $\overline{C_w(x_f)} = \frac{1}{m}\sum_{i=1}^{m} C_w(G_\theta(z^{(i)}))$

  Update θ using SGD by ascending with

  $$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m}\left[f_1(C_w(G_\theta(z^{(i)})) - \overline{C_w(x_r)}) + f_2(C_w(x^{(i)}) - \overline{C_w(x_f)})\right]$$

end while

Figure 21: RaGAN training pseudocode, image from [21].

2.3.5 Conditional GAN

In all the GAN setups described so far, the input to the generator is always defined as $\left\{z^{(i)}\right\}_{i=1}^{m} \sim \mathbb{P}_z$ ($\mathbb{P}_z$ normally being an easy-to-sample distribution such as $\mathcal{N}(0,1)$) per batch where m is the batch size. This way, we are simply asking a well trained G to generate a random sample from $\mathbb{P}_{data}$. It is possible however to condition this generation on additional information that is also added as an input to G.
In the conditional setup, training is conducted in the same way, only adding the conditional information as input to both G and D as part of each sample. There are few restrictions on the way this conditional information may be added as input to the networks, it may be through concatenation to the noise input as in [32], as input to a hidden layer as in [23] etc.

In many papers, Conditional GANs have been proven to be more robust and able to produce better quality samples than classical GANs. Conditional information is very diverse, it can be in the form of class as in [15], it can be a corrupted image in order to perform reconstruction as in [27] or it can be as a base photo to "beautify" it as in [7]. In our case as we will detail in an upcoming section we will teach our GAN to generate different dimensions of a time series by conditioning on one dimension, which we will call the "base" dimension.

2.4 Time Series Classifiers

2.4.1 ResNet

Presented in [44], the ResNet (Residual Network) architecture is a special type of neural network designed for Time Series Classification (TSC). The main characteristic of ResNets is the shortcut residual connection between consecutive convolutional layers. Actually, the difference with the usual convolutions (such as in FCNs) is that a linear shortcut is added to link the output of a residual block to its input thus enabling the flow of the gradient directly through these connections, which makes training a Deep Neural Network much easier by reducing the vanishing gradient effect [17].
The network is composed of three residual blocks followed by a Global Average Pooling layer and a final softmax classiffier whose number of neurons is equal to the number of classes in a dataset. Each residual block is first composed of three convolutions whose output is added to the residual block's input and then fed to the next layer. The number of filters for all convolutions is fixed to 64, with the ReLU activation function that is preceded by a batch normalization operation. In each residual block, the filter's length is set to 8, 5 and 3 respectively for the first, second and third convolution.

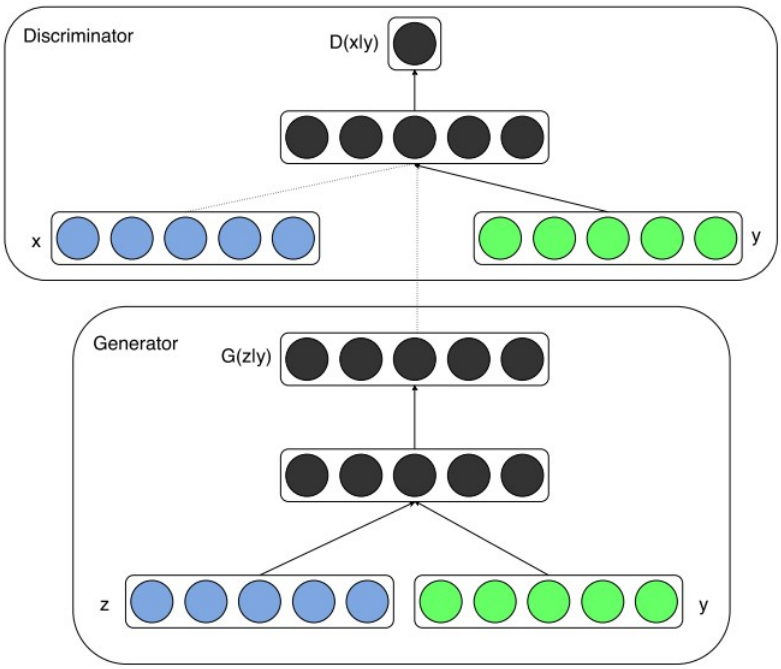


Figure 22: Conditional GAN Diagram
In this diagram, the conditional information is supplied to both networks by concatenation with the inputs of each network.

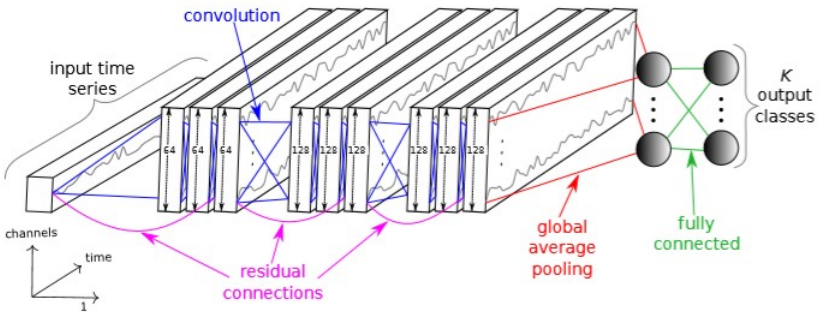


Figure 23: ResNet architecture

In a thorough empirical comparison of many TSC models (see [20]) the ResNet was found to be the best performer in a wide variety of datasets. This is the reason why we choose it to answer the third of our Research Questions 1.1. We will detail in the upcoming section how to improve the base results achieved by the ResNet in financial real-world dataset by creating new scenarios with WGAN-GP and RaGAN.

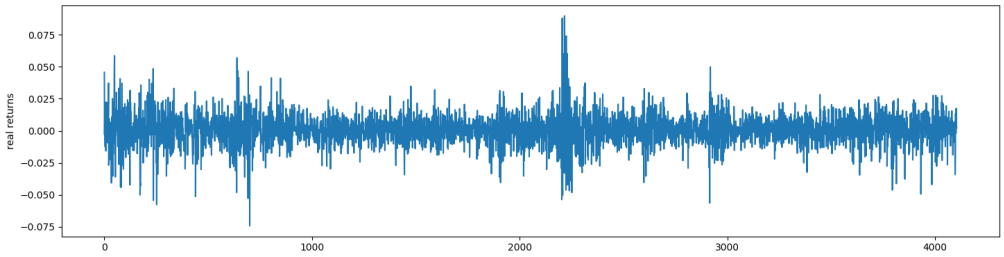
3 Experiments

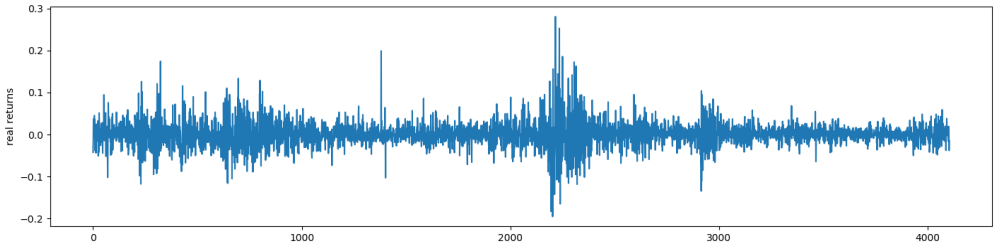
In this chapter we carry out experiments with financial datasets of our interest. We want to verify whether or not the WGAN-GP (2.3.2) and the RaGAN (2.3.4) can capture the structure of financial time series and produce diverse scenarios for use to make use of. Evaluation of our procedures is however complex, there is no clear consensus on the way to quantitatively score samples. GANs are harder to evaluate than other generative models because it can be difficult to estimate the likelihood for GANs as they do not provide an explicit representation of $\mathbb{P}_{model}$. [41] describes many of the difficulties with evaluating generative models and [28], [4] focus on GANs in particular.

In spite of these hardships, in our first experiment we will first try to measure quality by showing a series of distributional and time series statistics in order to illustrate how similar the real and synthetic samples are. In our second and third experiments, we will choose to carry out the "Train on Synthetic, Test on real" approach, detailed in Section 3.4, to quantify the quality of our models. All experimental setups as well as further details are described in Appendix C.

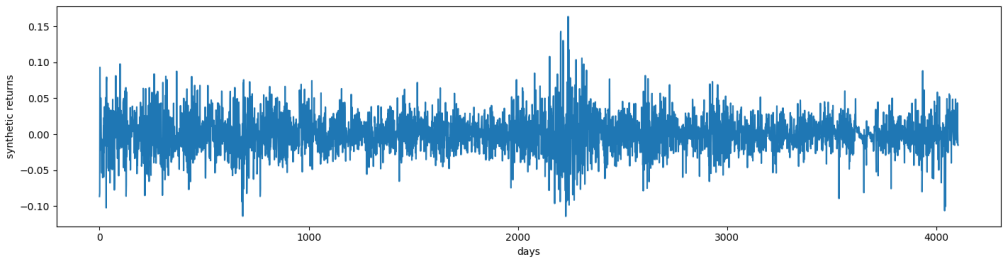
3.1 SP500 Stocks Generation with WGAN-GP

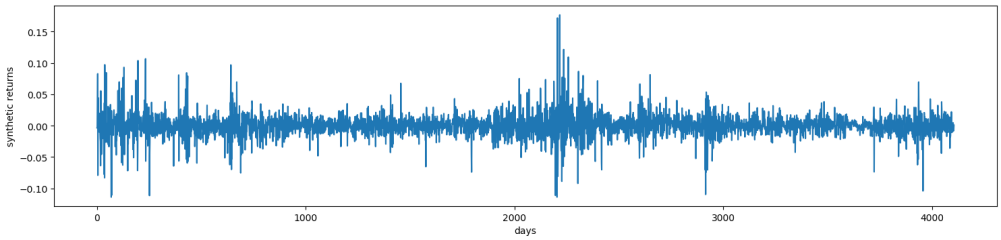
The first dataset of our interest are the daily returns of a set of 330 stocks belonging to the SP500 for a period of 4106 days from 3/1/2000 to 29/4/2016. Some samples of our dataset look like this:





We train our WGAN-GP with networks designed to process 1-dimensional time series (network architecture as well as training settings will be detailed in Appendix C) and then ask the generator to sample 330 new synthetic samples from $\mathbb{P}_{model}$. Some samples look like the following;





The synthetic samples are similar to those of $\mathbb{P}_{data}$, we can right away see behaviors such as volatility clustering being reproduced as well as the same order of trends/macro states. Giving a quantitative measure of "sample quality" is however not clear as we have already mentioned, this is why in the following subsection we are going to give a summary of statistics concerning the distribution of synthetic returns as well as some time-dependent properties, in order to see how similar they are to those of the real returns.

3.1.1 Statistical Analysis of Synthetic Series

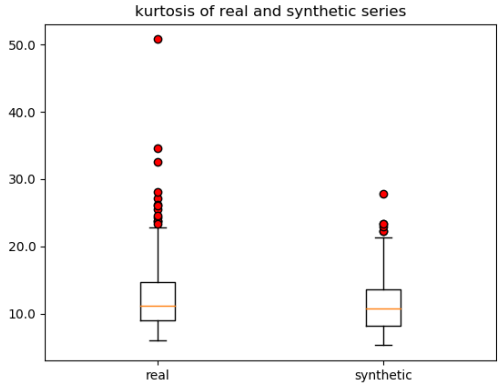


Figure 24: Kurtosis boxplot of real and synthetic series
The kurtosis is defined as the fourth standarized moment $\frac{\mathrm{E}\left[(X-\mu)^4\right]}{\left(\mathrm{E}\left[(X-\mu)^2\right]\right)^2}$

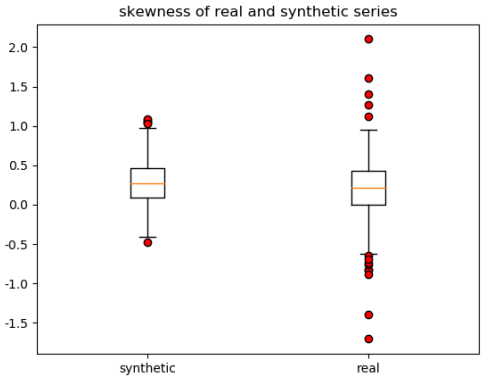
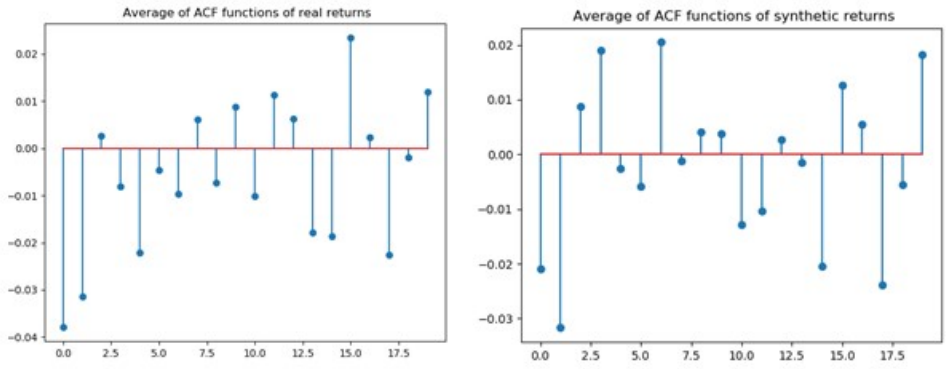
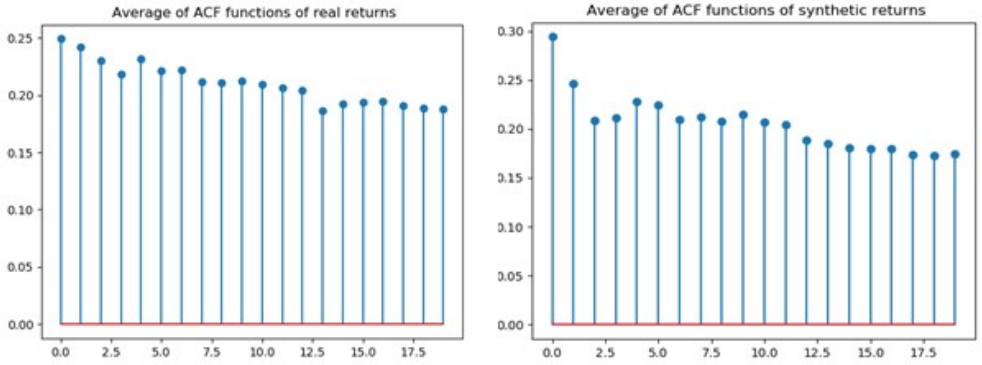


Figure 25: Skewness boxplot of real and synthetic series
The skewness is defined as the third standarized moment $\frac{\mathrm{E}\left[(X-\mu)^3\right]}{\left(\mathrm{E}\left[(X-\mu)^2\right]\right)^{3/2}}$

Figure 26: Monthly Returns correlations of real and synthetic series
We calculate the monthly returns of each series and then calculate the correlation coefficient $\mathrm{corr}(X,Y) = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathrm{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$ for each of the 330 series of each dataset.



For each series we calculate the correlation between returns of consecutive days and then take the average of all coefficients.



For each series we calculate the correlation between absolute returns of consecutive days and then take the average of all coefficients.

We observe that the WGAN-GP is capturing the structure of our original series, and we could enlarge our dataset this way. We are however trying to simulate "novel" financial scenarios and our WGAN-GP logically mimics the structure of the only scenario we have shown it, namely the US economy between 2000-2016.

This is why, in our next experiment, we are going to modify our dataset before feeding it to our WGAN-GP. By taking rolling windows of our series, we make $\mathbb{P}_{data}$ more diverse, so that the WGAN-GP may learn that for a given period, our series of interest may behave in very different ways.

3.2 VIX Scenarios with Conditional WGAN-GP

In order to illustrate the process of generating diverse financial scenarios with WGAN-GP we choose another dataset of interest, the daily closing prices of the Chicago Board of Options Exchange Volatility Index, more commonly known as the VIX, for the period 02/01/2004-04/04/2019. We choose 02/01/2004 as the start of our dataset because it was when the VIX started being calculated with its current methodology.

As explained by the CBOE, intraday VIX Index values are based on snapshots of SPX option bid/ask quotes every 15 seconds and are intended to provide an indication of the fair market price of expected volatility at particular points in time. There are many ways in which investors may choose to participate in the VIX such as VIX-based options, futures or ETFs and it is considered one of the most important ways in which investors may interact with market volatility. Because of this, the behaviour of the VIX has been widely studied, and many volatility models have been proposed to model its dynamics.

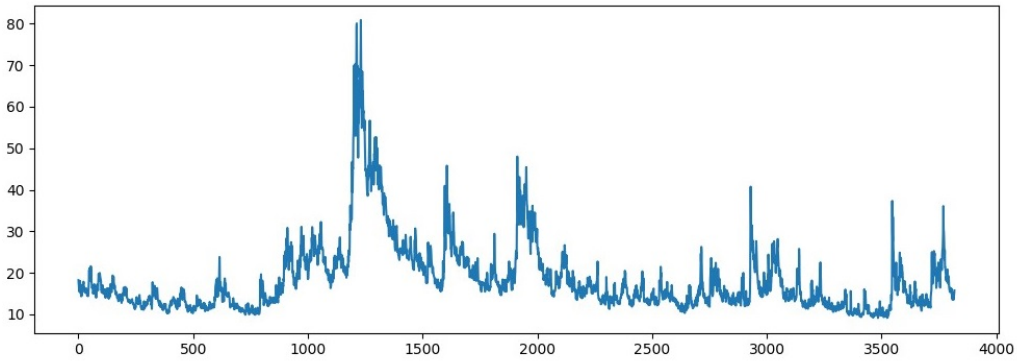


Figure 27: VIX daily closing prices for the period 02/01/2004-04/04/2019

We now build our dataset by taking segments of the closing prices for 1000 days, rolling forward 100 days at a time, so that all series share 100 days with the previous and following series. From these closing prices we calculate daily returns. In this way, we get a dataset consisting of snapshots with different behaviours of the VIX over time. We employ our WGAN-GP in a conditional setup; the VIX tends to be mean-reverting, hence the price level at which your series starts is important. If at the start of the series the VIX is valued at 60 points, it is more likely to descend in the following days, given its past behaviour. In order to make our WGAN-GP conditional, as explained in 2.3.5, we simply add to each series of returns its starting level, and concatenate it along the noise vector as input to the generator and the discriminator.

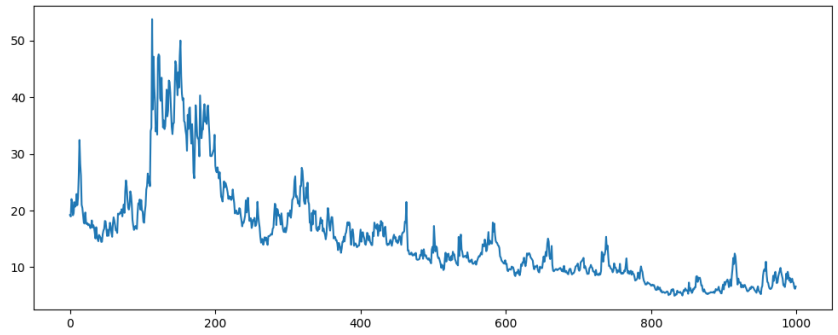


Figure 28: Synthetic VIX sample No 1

After training, we ask our model to produce samples of $\mathbb{P}_{model}$ by sampling from a vector of 50 random instances of $\mathbb{P}_z$ ($\mathcal{N}(0,1)$ in our case) and by randomly sampling starting prices from our dataset. We get a series of returns as the generator output and calculate prices evolving from the starting price, the resulting synthetic series can be visualized in Figures 28,29, 30 and 31. A video illustrating the outputs resulting from an interpolation of the initial price from 60 to 10 and a fixed noise seed can be seen here.

These results are at least visually promising. We could again visualize some statistics about the synthetic series and compare them to those of the original, but in this case the "stylized facts" of the VIX have not been as widely studied as those of stock prices. Instead we will demonstrate their quality in section 3.4.

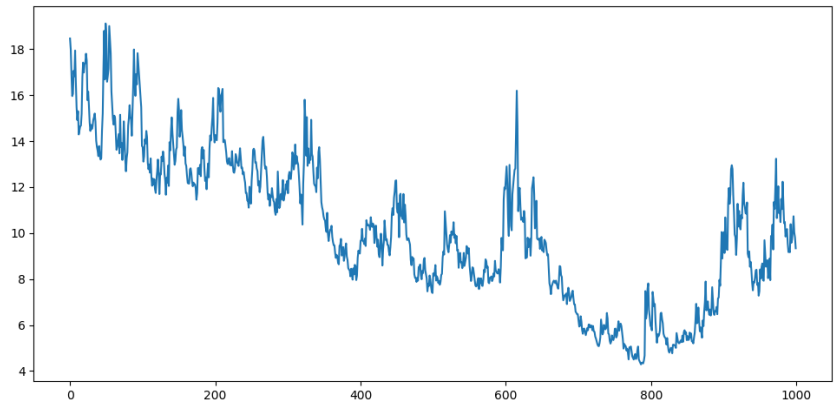


Figure 29: Synthetic VIX sample No 2

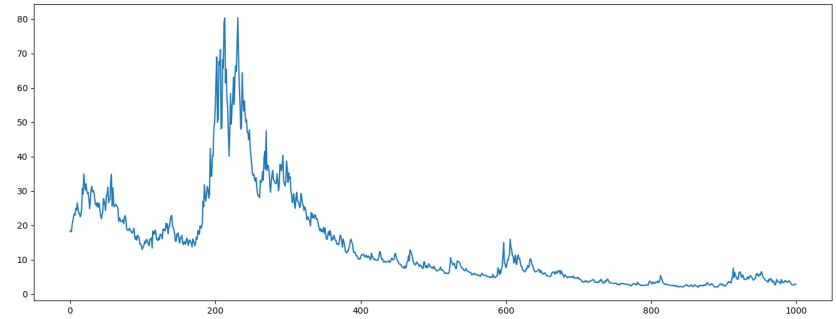


Figure 30: Synthetic VIX sample No 3

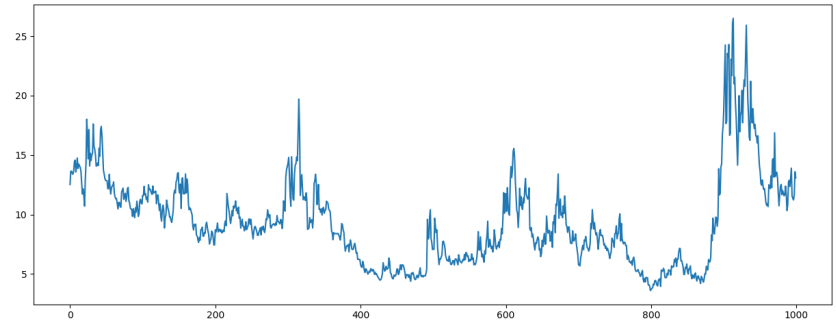


Figure 31: Synthetic VIX sample No 4

3.3 VIX and SP500 Scenarios

Our last objective regarding generating synthetic data is the generation of diverse and multidimensional scenarios. One approach we could take would be repeating the steps of Section 3.2 with multidimensional-series, modifying our networks so that they can handle matrix-shaped data. This approach resulted in mode collapse, as the networks had to be scaled up significantly and thus made training very costly and unstable even with WGAN-GP and Relativistic Average GAN.

In order to overcome this sort of "curse of dimensionality" in the multidimensional case, we propose the following generation procedure:

Given a multidimensional series, we carry out the following steps:

1. We first take rolling periods of the series as in 3.2, then choose one of the dimensions of the series as the "base" dimension and all the others as "associate" dimensions.
2. We construct a dataset for each "associate" dimension by taking the synchronous periods of the "base" and the corresponding "associate" dimension.
3. We train a Relativistic Average GAN in a conditional setup by supplying the "base" dimension as conditional information to the generator and have it generate the "associate" dimension. We then input both series to the discriminator. This way the generator can learn what the "associate" dimension behaviour would be given a "base" dimension.
4. We now train a WGAN-GP as in section 3.2 only on the "base" dimension and generate scenarios (i.e. samples of $\mathbb{P}_{model}$) of the "base" dimension.
5. Finally, for each sample of the "base" dimension, we conditionally generate each of the "associate" dimensions with each of the trained Relativistic Average GANs by giving as input the "base" dimension.

These steps can be summarized in the form of pseudocode in **Algorithm 2**.

Input: Multidimensional Time Series
Output: Synthetic Scenarios of the input Time Series
begin
  Choose one dimension as "base" dimension, the remaining will be the "associate" dimensions.
  foreach *d in "associate" dimensions* **do**
    1. Construct a dataset by taking synchronous rollig periods of the "base" dimension and d.
    2. Train a Relativistic Average GAN in a conditional setup conditioning on the "base" dimension and having the generator generate the corresponding d period.
  end
  Train a WGAN-GP as in section 3.2 on the "base" dimension and generate samples from $\mathbb{P}_{model}$
  foreach *Sample of* $\mathbb{P}_{model}$ *of the "base" dimension* **do**
    foreach *d in "associate" dimensions* **do**
      Conditionally generate, with the previously trained Relativistic Average GAN, the corresponding the d period by conditioning on the "base" dimension sample
    end
  end
end

Algorithm 2: Proposed Multidimensional Time Series Generation Procedure.

In order to illustrate our procedure we choose as a dataset a 2-dimensional time series composed of the VIX and the SP500 indices. These two indices have a pronounced negative correlation. Whenever the SP500 has a prolonged downwards period, the market volatility increases, so the VIX does as well. This is illustrated in Figure 32.

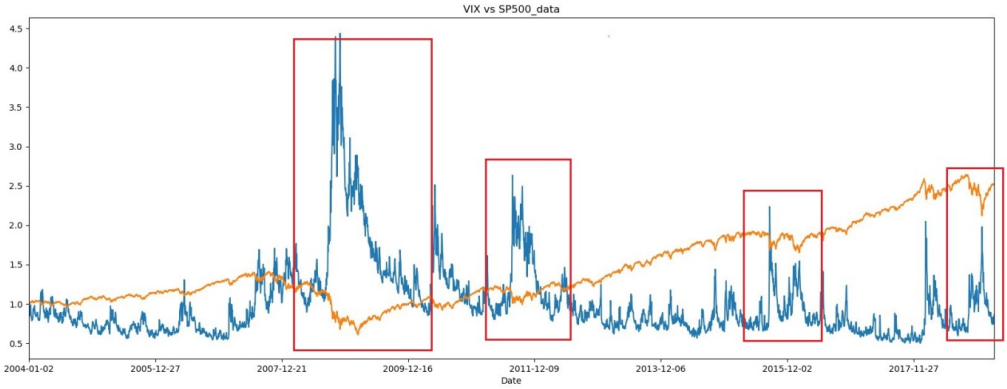


Figure 32: Relationship between the VIX and the SP500

We choose the daily returns of the SP500 as the "base" series and the returns of the VIX as the "associate" series. We construct the dataset by taking rolling periods of 100 days advancing 100 days every time, making pairs of "base" series from the SP500 and "associate" series from the VIX following Steps 1 and 2. After having trained our Relativistic Average GAN as in Step 3, we can ask it to conditionally generate "associated" series, by giving the "base" series as inputs. We illustrate the results in Figures 33, 34 and 35. The Figures show the resulting normalized rolling series of prices given the returns produced by the Relativistic Average GAN.

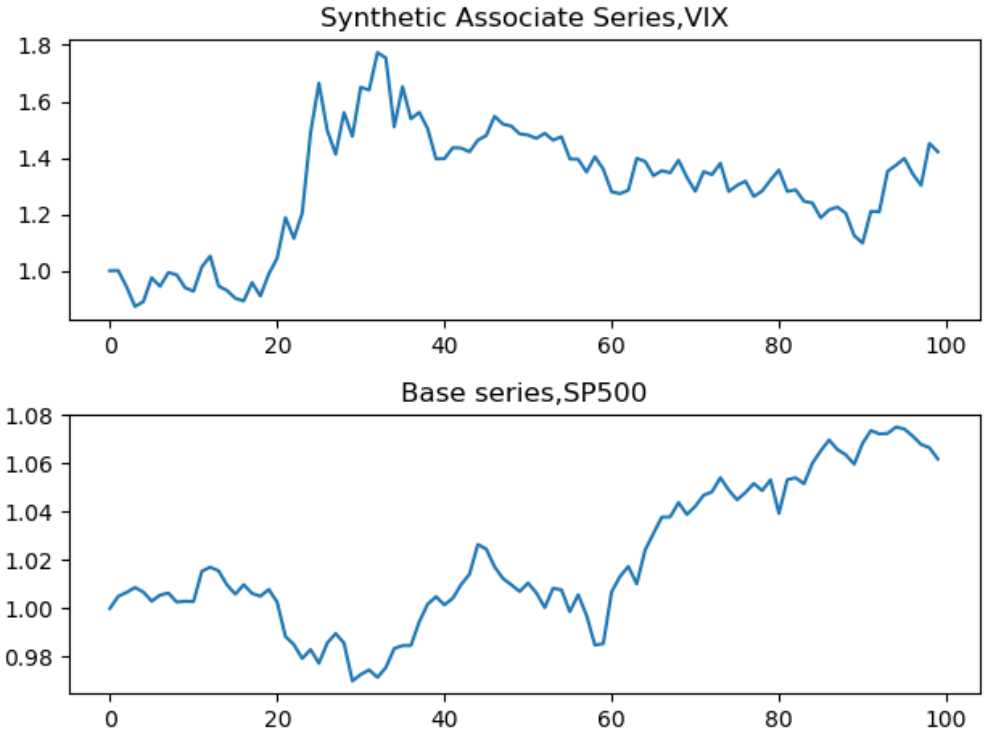


Figure 33: Synthetic VIX sample conditioned on the SP500 No 1

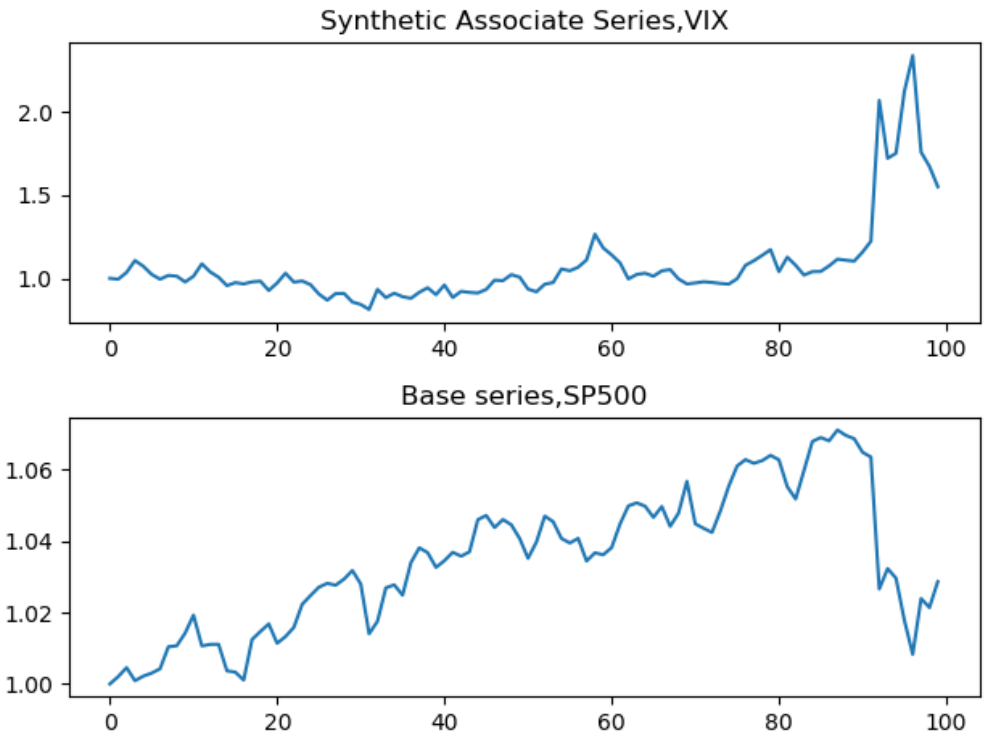


Figure 34: Synthetic VIX sample conditioned on the SP500 No 2

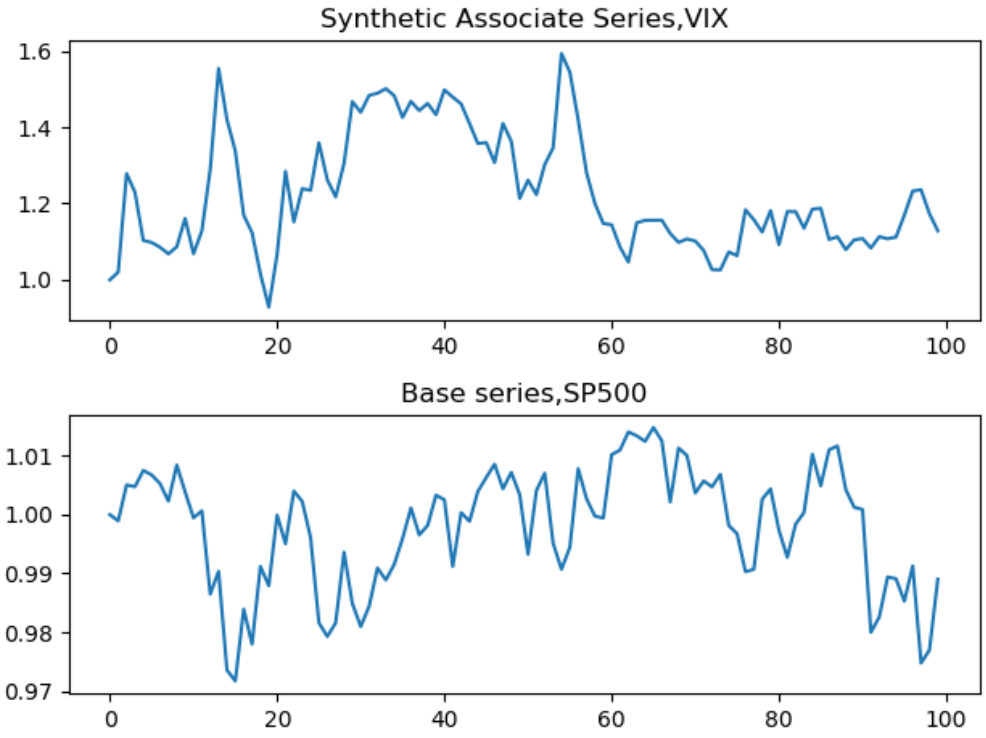


Figure 35: Synthetic VIX sample conditioned on the SP500 No 3

Following Step 4, now we need to generate new “base” series as in Section 3.2 . As in Step 3 we have trained the Relativistic Average GANs on returns of both the SP500 and the VIX, along with the SP500 returns we need to generate the starting point of the VIX for each SP500 "base" series.

Finally following Step 5 we conditionally generate the VIX returns, given our synthetic SP500 series and the VIX starting point and calculate the rolling series of the VIX prices. The results can be visualized in Figures 36, 37, 38, 39.

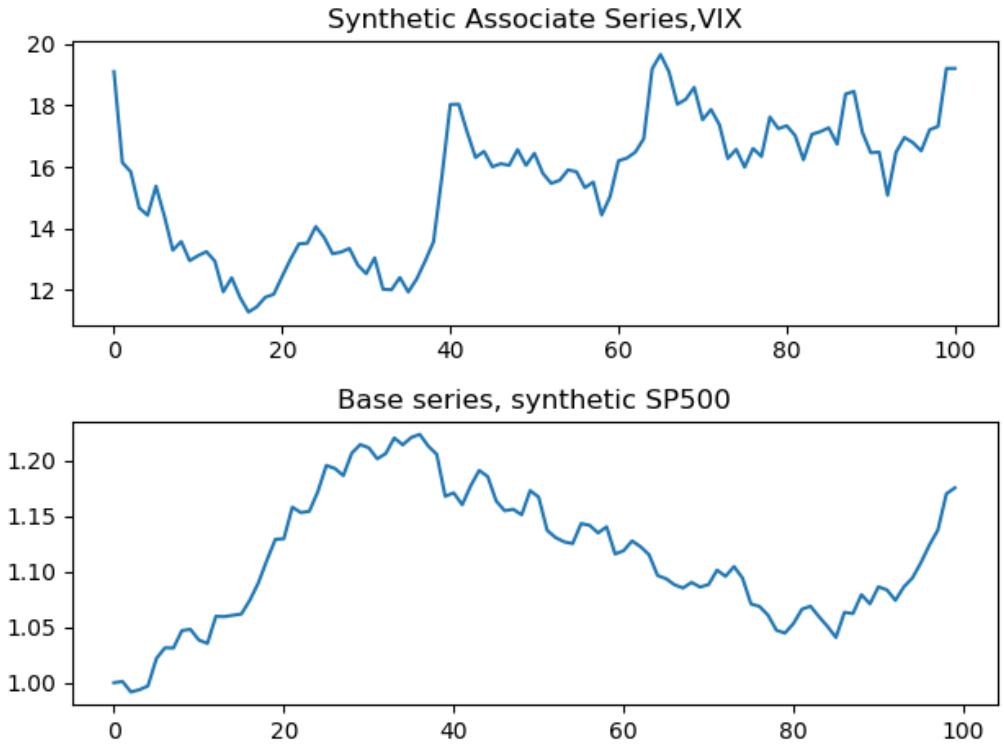


Figure 36: Synthetic VIX and SP500 series No1

The generated samples clearly reflect the negative correlation present in the real data, but as we have already mentioned, there is no clear consensus on the way to quantitatively evaluate GAN performance, see [28], [4] . This is why in the next section we carry out one of the most popular evaluation techniques.

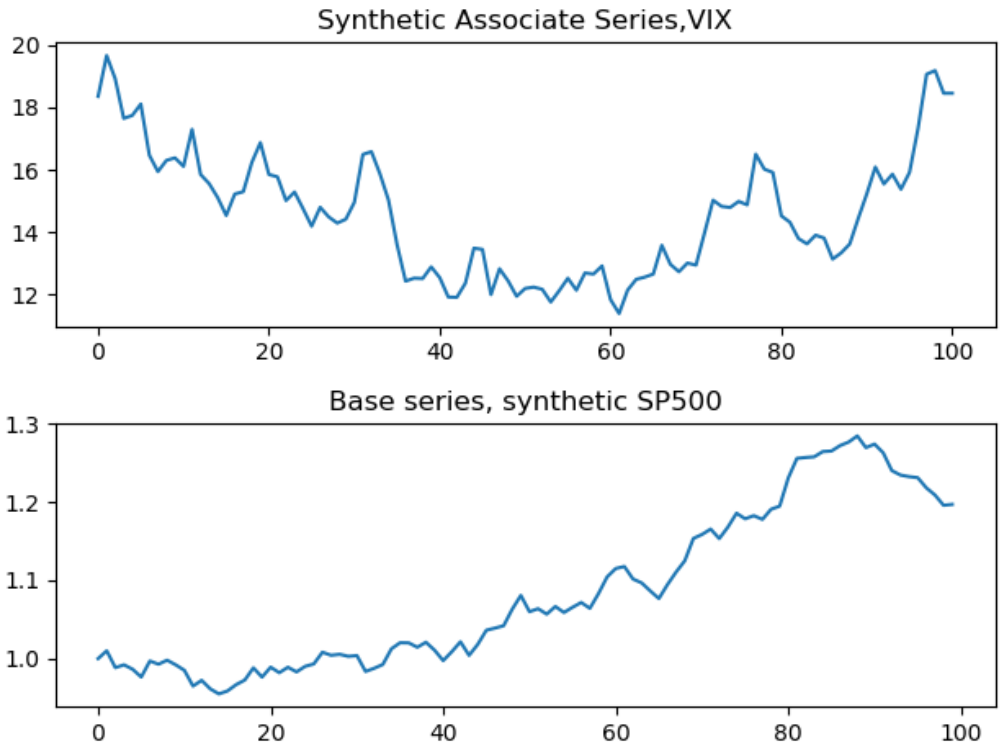


Figure 37: Synthetic VIX and SP500 series No2

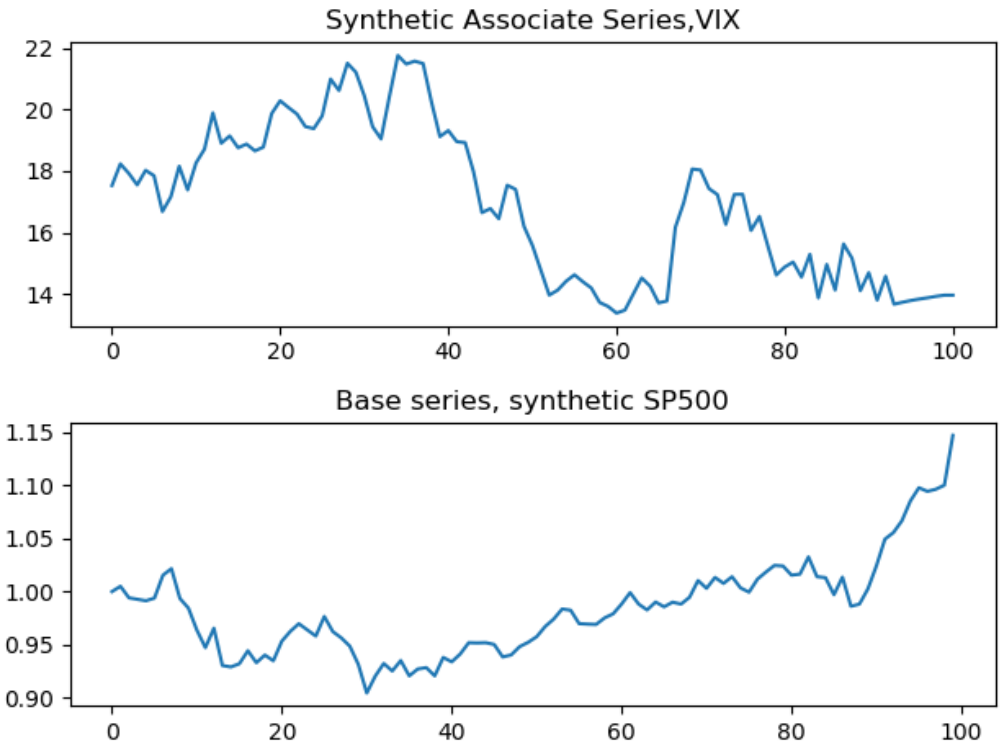


Figure 38: Synthetic VIX and SP500 series No3

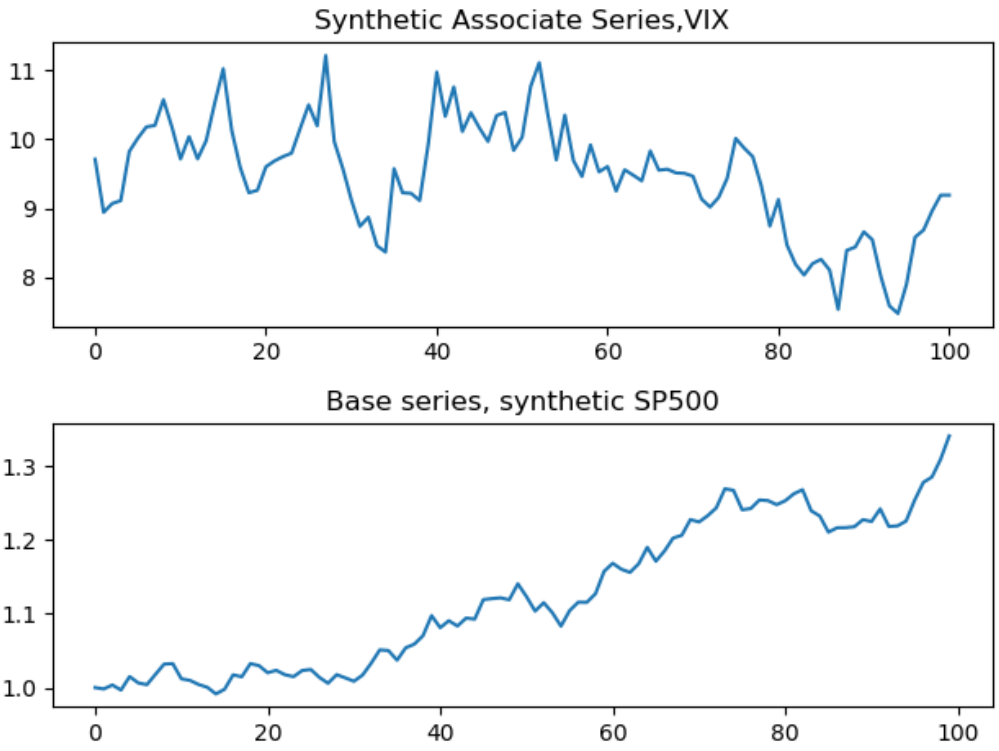


Figure 39: Synthetic VIX and SP500 series No4

3.4 "Train on synthetic, Test on Real"

Training neural networks on synthetic data is an active research field in which much research has been done in the case of images, given the success of image-generating GANs, see [43], [30]. A successful training on synthetic data has been used in many cases as a measure of performance for GANs, since if the GAN can generate data able to train another model, then the synthetic data can be considered as coming from $\mathbb{P}_{data}$.

This approach was first proposed in the case of Time Series in [10]. It consists on performing supervised training of a classifier on a synthetic dataset generated by the GAN and then testing the trained model on a held-out set of real data. This evaluation metric is among the most popular, since it demonstrates that the synthetic data produced by the GAN may be used in real applications.

3.4.1 VIX Scenarios with Conditional WGAN-GP Evaluation

In order to evaluate the quality of the series generated in Section 3.2, the supervised task we choose is forecasting VIX movements with the ResNet from section 2.4.1. The VIX tends to be mean-reverting and it stays in the 10-15 points range for long periods of time, hence the options with strike prices in this range are very popular. Knowing whether the VIX will be above or under 15 points say, a month ahead, could be very valuable. Given the last 30 prices of the VIX, can the ResNet predict at what level the VIX will be 20 trading days from today?

First, we perform training in the classic way, splitting our dataset in a training set from 01/02/2004 to 10/20/2015 and a test set from 20/11/2015 to 04/04/2019. We split each dataset in periods of 30 days (non-overlapping in the training set and overlapping in the test set) and then divide our samples in two classes :

1. **Class 0**: if the VIX price 20 days ahead the last day of each period was below 15.
2. **Class 1**: if the VIX price 20 days ahead the last day of each period was above 15.

We conduct training for 2000 iterations, the loss/accuracy plots can be seen in Figure 40.

The accuracy stabilizes close to 60% and the loss plot suggests some overfitting to the training data. These results are coherent, financial data is ever-changing by nature, and for this reason, overfitting is to be expected.

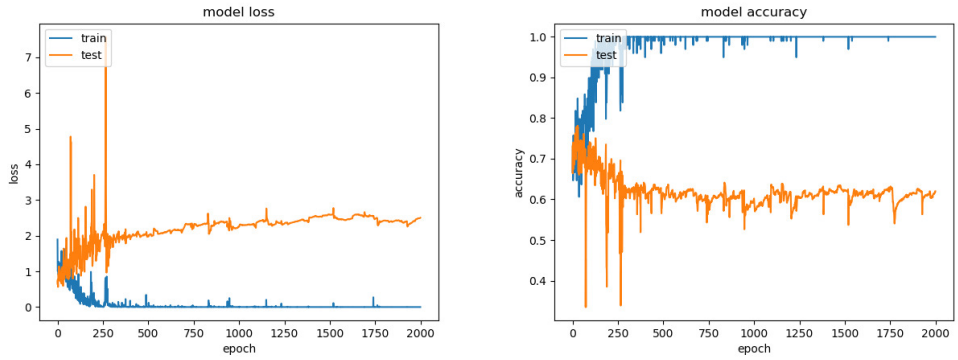


Figure 40: Loss/Accuracy plots of the original training set

We now try the "Train on Synthetic, test on Real" approach, by first generating VIX price series (using only training data as input to the GAN) as in section 3.2. Sampling then from $\mathbb{P}_{model}$ we add the samples to the training set and carry out the ResNet training as before. The loss/accuracy plots can be seen in Figure 41.

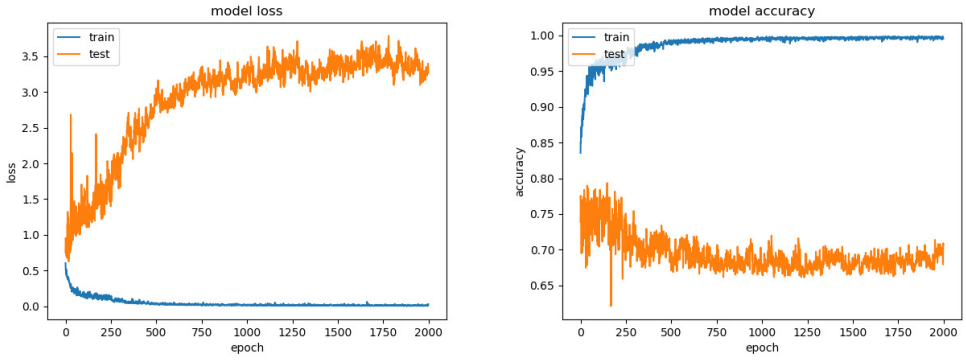


Figure 41: Loss/Accuracy plots of the enlarged training set

We can see that the synthetic data is also able to train our ResNet in a similar fashion as the real data, passing the quality test. Not only that, the enlarged dataset allows the ResNet to reach a higher accuracy, around 70%, on the test set. This tells us that **our synthetic data is not only similar to the training data that we used to generate it, but also more similar to the test data than the training data**. The synthetic samples produced by our GAN mitigate overfitting and make the model generalize better. This is behaviour is consistent among different training runs, see Appendix B.1.

Still, as evidenced by the plots, random guessing is as accurate as forecasting with the ResNet, so it would not be a great rule to use as an investment strategy.

3.4.2 VIX and SP500 Scenarios Evaluation

In order to evaluate the quality of the series generated in Section 3.3 the supervised task we choose is forecasting the SP500 movements with the ResNet from section 2.4.1. Given the last 30 prices of the VIX and the last 30 returns of the SP500, can the ResNet predict the trend of the SP500 20 days ahead?

In order to perform this forecast we split our dataset in non-overlapping periods of 30 days and then divide our samples in three classes:

1. **Class 0**: if the SP500 loses more than **2.5**% of its value on the following 20 days.
2. **Class 1**: if the SP500 price stays within a **2.5**% range on the following 20 days.
3. **Class 2**: if the SP500 appreciates more than **2.5**% of its value on the following 20 days.

We perform training in the classic way, splitting our dataset in a training set from 01/02/2004 to 10/20/2015 and a test set from 20/11/2015 to 04/04/2019. We conduct training for 2000 iterations, the loss/accuracy plots can be seen in Figure 42.

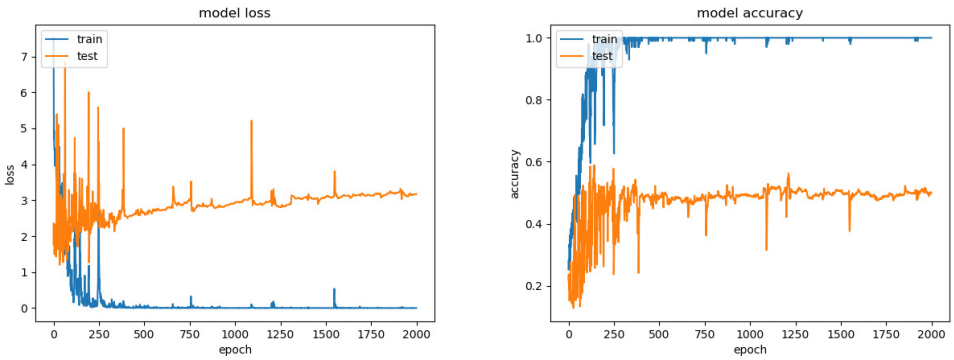


Figure 42: Loss/Accuracy plots of the original training set

We now try the "Train on Synthetic, test on Real" approach, generating SP500 and VIX series (using only training data as input to the GAN) following Algorithm 2. We add the samples to the training set and carry out the ResNet training as before. The loss/accuracy plots can be seen in Figure 43.

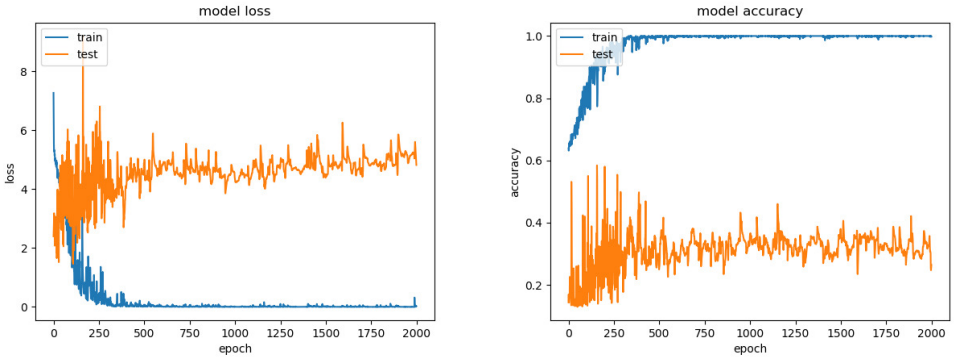


Figure 43: Loss/Accuracy plots of the original training set

In this experiment, our results are worse, as the ResNet does not reach the accuracy achieved by real data, meaning that our synthetic data is not as similar to real data as in earlier experiments. Event though, the samples produced by Algorithm 2 were visually coherent, they did not pass the "Train on Synthetic, test on Real" approach test. This is behaviour is consistent among different training runs, see Appendix B.2.

4 Conclusions

In this work we explored the application of Generative Adversarial Networks to time series data, in order to perform data augmentation of financial datasets.

We proposed first for the unidimensional case the use of Wasserstein's GAN with Gradient Penalty coupled with 1-dimensional convolutional networks in order to work on time series data. We checked for statistics comparing the generated series to the original ones and then proposed taking a rolling window of the original series in order to induce variability of scenarios.

We proposed a conditional implementation of the Relativistic Average GAN, in order to avoid mode collapse in the multidimensional case. Our method involves first training a conditional Relativistic Average GAN on pairs of "base" and "associated" series for each dimension, so that it learns the relationship between dimensions of the time series, then generating a varied unidimensional dataset with WGAN-GP and finally obtaining the remaining dimensions by running one Relativistic Average GAN for each dimension.

In Section 3.4 we showed an approach to measuring the performance of our generation procedures. In addition, this approach illustrated how synthetic data can help deep learning models mitigate overfitting and generalize better. These results open up many research paths, given the flexibility of our procedures, possibly leading to new applications of deep learning and reinforcement learning techniques in financial settings, as lack of training data and overfitting have been two of the biggest concerns which have set back this kind of developments.

5 Further Research

Our work could be further extended by researching the following topics:

1. *Improved Multidimensional Time Series Generation Procedure.*

   An improved procedure in the spirit of Algorithm 2, that could consistently pass the "Train on Synthetic, Test on Real" approach test would be a big research milestone. Perhaps instead of conditionally sampling the different dimensions as in our approach, simply building a network that could perform classic generation as in 3.2 of multidimensional series while avoiding Mode Collapse.

2. *Improved architectures and technical tricks for the Generator and Discriminator Networks.*

   In the GAN community, many papers focus solely on implementing new improved architectures for the networks used as generator and discriminator and demonstrating the usefulness of certain techniques, such as Batch Normalization, Residual Connections, in order to beat state-of-the-art FiD and Inception scores. While some of these breakthroughs are universal to all GANs, they're mostly focused on image generation and time series data presents a completely different structure from which we think a lot could be researched.

3. *Baseline Performance Metric.*

   Development of GANs for Time Series generation relies upon having a common baseline metric of success, in the same way as the Fréchet Inception Distance and the Inception Score are used in Image Generation. Perhaps having an standard classifier such as the ResNet and taking AUC scores of a fixed hyperparameter classification between real and synthetic data could be an approach.

4. *Improved Time Series Classification and Prediction Deep models.*

   While parallel to our research, having new time series specific models to justify the improvement of data augmentation techniques would probably be a great incentive.

A Experiments on Sine Curves

A.1 WGAN-GP on sine curves

In order to perform a sanity check of our generative models and assure their correct implementation, we tested their performance on a simple dataset on which the expected behaviour is easily verified. The dataset we chose are sine curves with random period and amplitude, some samples of the dataset can be visualized in Figures 44, 45.

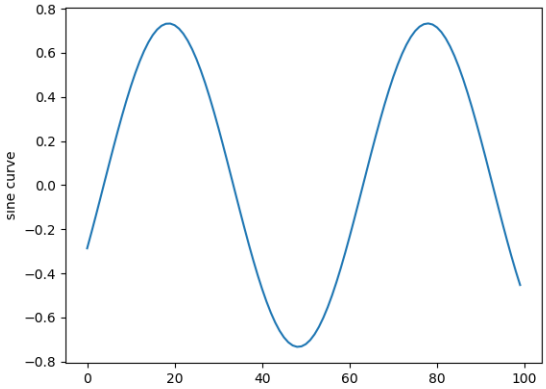


Figure 44: Sample of $\mathbb{P}_{data}$

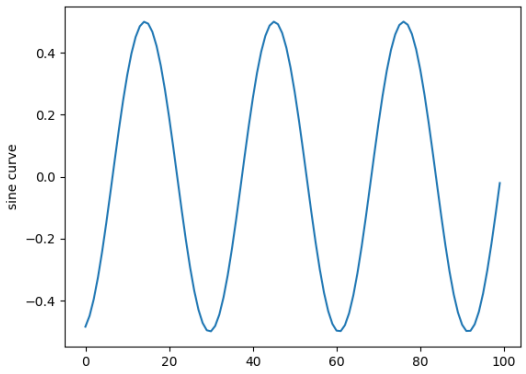


Figure 45: Sample of $\mathbb{P}_{data}$

We train our WGAN-GP of section 2.3.2 on this dataset and then ask it to generate samples from $\mathbb{P}_{model}$, the results can be visualized in Figures 46, 47. The samples from $\mathbb{P}_{model}$ look similar to those of $\mathbb{P}_{data}$, some imperfections are noticeable but they also have different periods and amplitudes as well as the oscillating behavior.

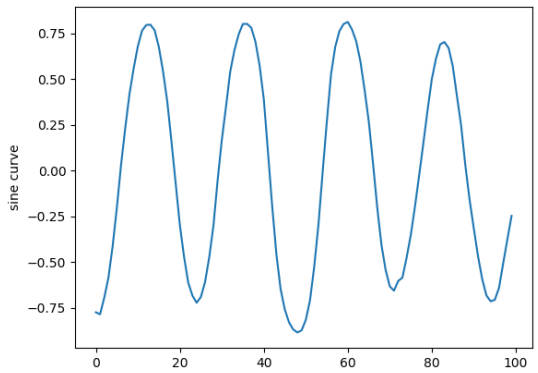


Figure 46: Sample of $\mathbb{P}_{model}$

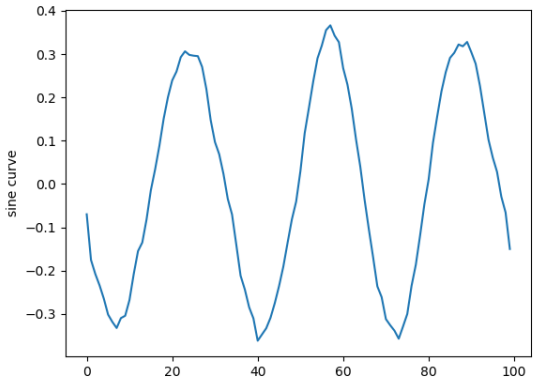


Figure 47: Sample of $\mathbb{P}_{model}$

A.2 Relativistic Average GAN on sine curves

Similarly as in the previous section, we perform a sanity check with our implementation of the Relativistic Average GAN in a conditional setup. We take the same dataset as in the previous section and construct samples of "base" and "asssociated" curves by taking the sine curves and their additive inverse. We only carry out the Steps 1-3 of section 32. After training the model, we generate samples from $\mathbb{P}_{model}$, the results can be visualized in Figures 48, 49.

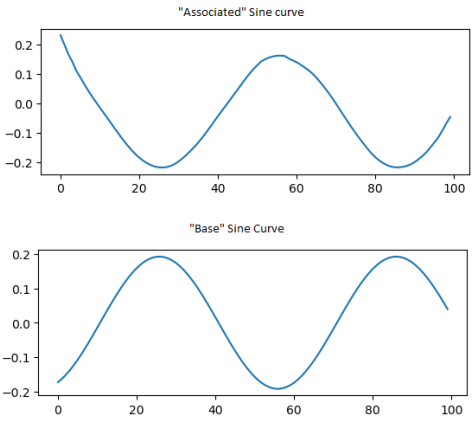


Figure 48: Sample of $\mathbb{P}_{model}$

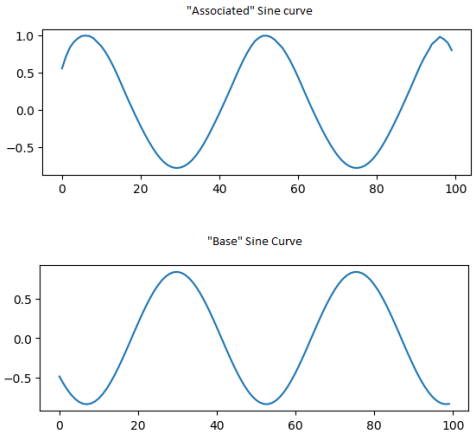


Figure 49: Sample of $\mathbb{P}_{model}$

B Loss/Accuracy Plots for the ResNet Experiments

B.1 VIX Scenarios with Conditional WGAN-GP Evaluation

Additional Loss/Accuracy plots of different training runs on the ResNet in the experiment of Section 3.4.1.

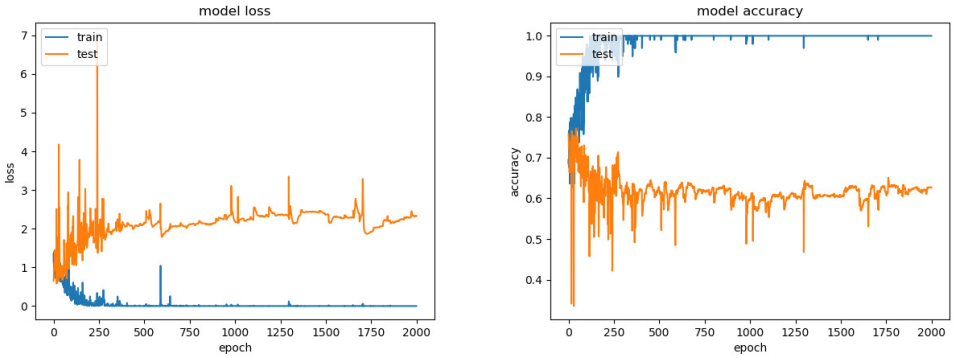


Figure 50: Loss/Accuracy plots on the original training set

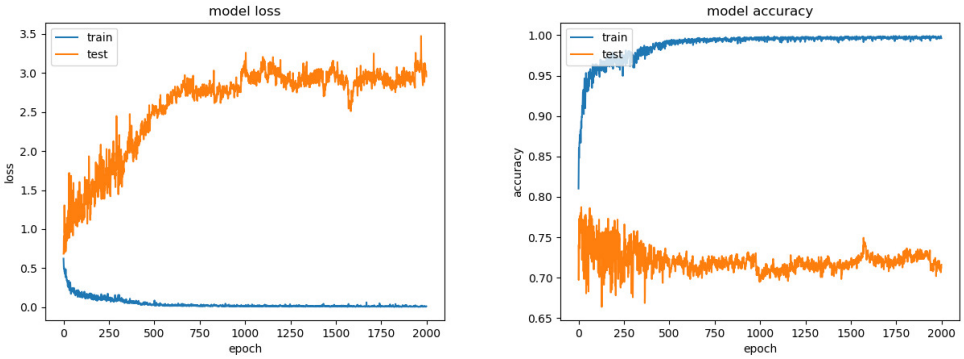


Figure 51: Loss/Accuracy plots on the enlarged training set

B.2 VIX and SP500 Scenarios with WGAN-GP and RaGAN Evaluation

Additional Loss/Accuracy plots of different training runs on the ResNet in the experiment of Section 3.4.2.

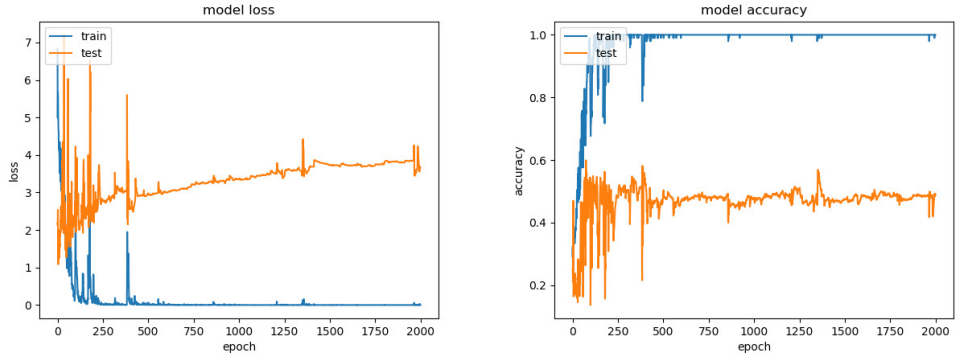


Figure 52: Loss/Accuracy plots on the original training set

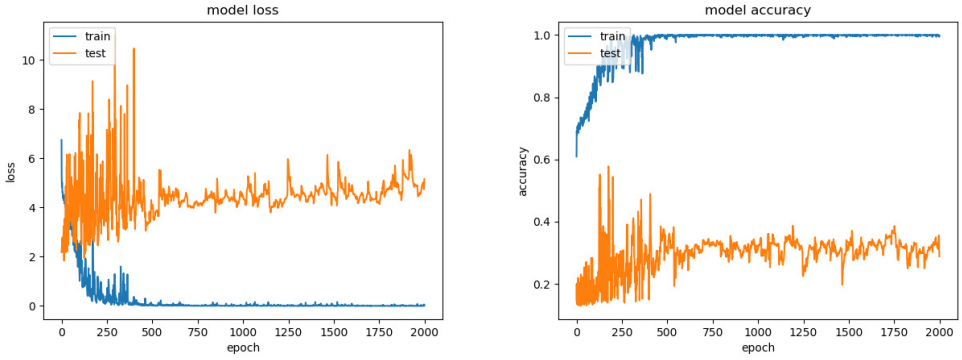


Figure 53: Loss/Accuracy plots on the enlarged training set

C Experimental Setups

Architectural details of the procedures shall remain undisclosed due to confidentiality issues. For inquiries, please contact ETS Asset Management Factory .

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

Martín Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *CoRR*, abs/1701.04862, 2017.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. *arXiv e-prints*, page arXiv:1701.07875, Jan 2017.

Ali Borji. Pros and Cons of GAN Evaluation Measures. *arXiv e-prints*, page arXiv:1802.03446, Feb 2018.

François Chollet et al. Keras. https://keras.io, 2015.

Laurie Davies and Walter Krämer. Stylized Facts and Simulating Long Range Financial Data. *arXiv e-prints*, page arXiv:1612.05229, Dec 2016.

Nir Diamant, Dean Zadok, Chaim Baskin, Eli Schwartz, and Alex M. Bronstein. Beholder-GAN: Generation and Beautification of Facial Images with Conditioning on Their Beauty Level. *arXiv e-prints*, page arXiv:1902.02593, Feb 2019.

John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv e-prints*, page arXiv:1603.07285, Mar 2016.

Cristóbal Esteban, Stephanie L. Hyland, and Gunnar Rätsch. Real-valued (Medical) Time Series Generation with Recurrent Conditional GANs. *arXiv e-prints*, page arXiv:1706.02633, Jun 2017.

Shintaro Funahashi, Charles Bruce, and P S Goldman-Rakic. Funahashi s, bruce cj, goldman-rakic ps. mnemonic coding of visual space in the monkey's dorsolateral prefrontal cortex. j neurophysiol 61: 331-349. *Journal of neurophysiology*, 61:331–49, 03 1989.

Ian Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1701.00160, Dec 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1406.2661, Jun 2014.

Guillermo L. Grinblat, Lucas C. Uzal, and Pablo M. Granitto. Class-Splitting Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1709.07359, Sep 2017.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved Training of Wasserstein GANs. *arXiv e-prints*, page arXiv:1704.00028, Mar 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, page arXiv:1512.03385, Dec 2015.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, page arXiv:1502.03167, Feb 2015.

Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *arXiv e-prints*, page arXiv:1809.04356, Sep 2018.

Alexia Jolicoeur-Martineau. The relativistic discriminator: a key element missing from standard GAN. *arXiv e-prints*, page arXiv:1807.00734, Jul 2018.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. *arXiv e-prints*, page arXiv:1710.10196, Oct 2017.

Tero Karras, Samuli Laine, and Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1812.04948, Dec 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec 2014.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image Inpainting for Irregular Holes Using Partial Convolutions. *arXiv e-prints*, page arXiv:1804.07723, Apr 2018.

Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are GANs Created Equal? A Large-Scale Study. *arXiv e-prints*, page arXiv:1711.10337, Nov 2017.

Hans Malmsten and Timo Teräsvirta. Stylized facts of financial time series and three popular models of volatility. *SSE/EFI Working Paper Series in Economics and Finance*, (503), Aug 2004.

Nikolaus Mayer, Eddy Ilg, Philipp Fischer, Caner Hazirbas, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. What Makes Good Synthetic Training Data for Learning Disparity and Optical Flow Estimation? *arXiv e-prints*, page arXiv:1801.06397, Jan 2018.

Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1611.02163, Nov 2016.

Mehdi Mirza and Simon Osindero. Conditional Generative Adversarial Nets. *arXiv e-prints*, page arXiv:1411.1784, Nov 2014.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines vinod nair. volume 27, pages 807–814, 06 2010.

Hariharan Narayanan and Sanjoy Mitter. Sample complexity of testing the manifold hypothesis. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1786–1794. Curran Associates, Inc., 2010.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1511.06434, Nov 2015.

Lillian J. Ratliff, Samuel A. Burden, and S. Shankar Sastry. On the Characterization of Local Nash Equilibria in Continuous Games. *arXiv e-prints*, page arXiv:1411.2168, Nov 2014.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *arXiv e-prints*, page arXiv:1409.0575, Sep 2014.

R. M. S. T. Rachev. *Duality theorems for Kantorovich-Rubinstein and Wasserstein functionals*. Instytut Matematyczny Polskiej Akademi Nauk, 1990.

Luca Simonetto. Generating spiking time series with generative adversarial networks: an application on banking transactions. Master's thesis, University of Amsterdam, Sept 2018.

Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models. *arXiv e-prints*, page arXiv:1511.01844, Nov 2015.

T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training Deep Networks with Synthetic Data: Bridging the Reality Gap by Domain Randomization. *arXiv e-prints*, page arXiv:1804.06516, Apr 2018.

Zhiguang Wang, Weizhong Yan, and Tim Oates. Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline. *arXiv e-prints*, page arXiv:1611.06455, Nov 2016.

Lilian Weng. From gan to wgan. https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html#kullbackleibler-and-jensenshannon-divergence, 2019.

M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton. On rectified linear units for speech processing. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3517–3521, May 2013.