Adapting Adversarial Estimator for Time Series Data: A Recurrent Neural Network Approach

Jose Luis Martínez Martínez

July 2025

Abstract

This document addresses the open challenge of adapting the Adversarial Estimator for time series data. This estimator integrates the idea of GANs (Generative Adversarial Networks) into the parameter estimation framework. The key feature of this estimation method lies in the Discriminator, which acts as a classifier between real and fake (generated) data. Previous studies motivate interest in this approach as their results demonstrate good properties asymptotically and with finite samples. Not only does the estimator achieve consistency, but under an Oracle classifier, it was proven to be efficient.

Our contribution focuses on designing a Discriminator capable of incorporating the structure of serial correlation inherent in time series data. We propose utilizing a Recurrent Neural Network (RNN) with a Long Short-Term Memory (LSTM) structure as the Discriminator. LSTM networks were designed to allow for self-selection of memory use and solve the vanishing gradient problem common in basic RNNs. A critical methodological adaptation for training the LSTM Discriminator with serially correlated data is the employment of the Block Bootstrap method. This technique creates approximately independent and identically distributed (i.i.d.) training samples by resampling blocks of the original time series, thereby improving the stability and accuracy of the adversarial estimation process.

The proposed algorithm, inspired by the Generative Adversarial Networks (GANs) framework, iteratively trains the Discriminator to distinguish between bootstrapped real data and simulated data, while the Generator updates its parameters to "fool" the Discriminator. Preliminary simulation results for MA(1), AR(1), and AR(2) models, despite being computationally demanding, show more promising convergence behavior compared to initial attempts and suggest that this adapted framework is a viable direction for future research. These findings delineate clear avenues for future research, including more extensive statistical evaluations, adaptive hyperparameter tuning, and refined Discriminator architectures.

1 Introduction

This document aims to shed light on how to properly adapt the Adversarial Estimator when using time series data. This estimator, first proposed by Kaji, Manresa, and G. Pouliot 2023, integrates the idea of GANs (Generative Adversarial Networks) into the parameter estimation framework. The key feature of this estimation method lies in the Discriminator, which acts as a classifier between real and fake (generated) data.

This simulated procedure has been further studied and evaluated mainly by the original authors in two more papers: Kaji, Manresa, and G. A. Pouliot 2021 and Cigliutti and Manresa 2022. These papers motivate the interest in this approach, as their results demonstrate good properties asymptotically and with finite samples. Not only does the estimator achieve consistency, but under an Oracle classifier (an infeasible classification function)—and thus under a good Discriminator (one that eventually approaches this Oracle)—it was proven to be efficient. It has also shown competitive performance against other simulated methods in simulation exercises, outperforming all of them under a particular form of the Discriminator, namely a Neural Network (which further motivates investigation in this direction).

With this in mind, our work will focus on an open challenge that the creators of the Adversarial Approach highlighted in their paper (Kaji, Manresa, and G. Pouliot 2023):

"First, the theoretical results in this paper do not cover time series data.[...]. This is not to say that the adversarial framework cannot be extended thereto, but it would require a careful design of the discriminator to incorporate the structure of the serial correlation."

This is precisely what we will try to achieve, or at least try to provide some guidance on the direction we should pursue. The proposed solution consists of using a Neural Network with a particular recurrent structure, capable of capturing the serial dependencies about which the authors warned. This particular structure is the Long Short-Term Memory (LSTM) Network.

In what follows, we will explain the basics of Recurrent Neural Networks and how they work, justifying our choice. The rest of the document is organized as follows:

Section 2 introduces the Adversarial Approach to parameter estimation as conceived by its authors. Section 3 discusses Neural Networks and how they have been adapted to account for serially correlated data, introducing our Discriminator candidate: the LSTM Network. The knowledge exposed in this section has been mainly drawn from the following sources: Ghojogh and Ghodsi 2023, Greff et al. 2017, Lillicrap and Santoro 2019, and C. Williams 2024. Section

4 finally presents our contribution and summarizes the motivation behind our choice, along with the final algorithm used and some simulation results. We conclude with final comments in Section 5.

2 Adversarial Approach to Parameter Estimation

As mentioned, the **Adversarial Estimator** was first introduced by Kaji, Manresa, and G. Pouliot 2023 as an alternative simulated method; subsequent studies have also been primarily conducted by its original authors. The idea behind this estimator is anchored in **GANs (Generative Adversarial Networks)**, which are generative models that pit two Neural Networks (a Generator against a Discriminator) against each other, resulting in a powerful tool for mimicking real data.

Even though GANs were originally conceived for image generation, their underlying idea is so powerful that it extends far beyond that topic.

2.1 Introduction to GANs

To grasp the spirit of the Adversarial Estimator, a conceptual understanding of its underlying structure is essential. That's why this section provides a brief introduction to GANs. Our goal here isn't to delve into their technical details in depth, but rather to understand the main idea behind them so we can extrapolate it to our specific case: adapting them for model estimation.

GANs were introduced by Goodfellow et al. 2014 as a novel method for image generation. More precisely, GANs aim to create an effective imitator for some desirable real data that we wish to mimic (meaning we have an unknown distribution we want to artificially reproduce).

The method is based on the iterative and simultaneous training of two models:

- **Generator (G)**: Generates *fake data*.
- **Discriminator (D)**: Distinguishes between observed *real data* and generated *fake data*.

The Generator constantly tries to fool the Discriminator, which, in turn, continuously improves its classification skills. The idea is that this (virtuous) competition yields a very good Generator (capable of producing data that looks genuinely real) trained by a very strict Discriminator (capable of sharply detecting the true underlying distribution). This iterative, competitive structure is precisely what gives them their "Adversarial" name.

This brief overview should be sufficient to grasp the intuition and understand the following application to model estimation. For a more visual understanding, we recommend checking Figure 1, which represents the iterative process of a GAN. (For this figure, we assume the Discriminator is already efficient, but in practice, the Discriminator also undergoes an improvement process.) It's worth noting that these two models were originally represented by Neural Networks (something we'll discuss later, and which relates to the main contribution of this document). So, even though it's not explicitly mentioned, the reader should keep in mind that the 'training' process is driven by some type of machine-learning algorithm (meaning GANs also use an error measure to improve their performance, i.e., a Loss function). More technical details, such as how these structures learn or are trained mathematically, will be explained later.

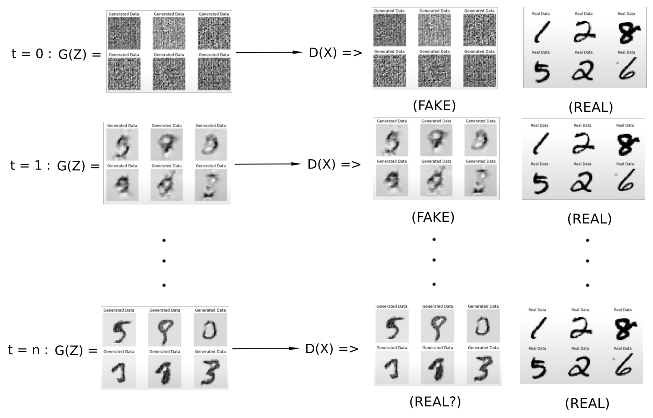

Figure 1: This image represents an abstraction of the Generator's learning process. Initially, it produces essentially white noise, which the Discriminator easily identifies. However, as the Generator improves, its representations become more complex, making the Discriminator's task much harder.

2.2 The Adversarial Estimator

Now we'll describe how to adapt these GANs to the parameter estimation context, as expressed in Kaji, Manresa, and G. Pouliot 2023.

2.2.1 Estimator Depiction

It's important to recognize that the main novelty (and contribution) introduced by this Adversarial method is the Discriminator's involvement. This component guides the simulation process and determines whether the parameters proposed by the simulation best fit the real data we feed into the model—which is the core purpose of any data-based estimator.

The estimator is motivated by the following framework:

Imagine we have a series of n **real data** points $\{X_i\}_{i=1}^n$, which are essentially draws from an unknown distribution. Our goal is to estimate parameters capable of describing this unknown reality. For this purpose, we design a parametric model that we believe (based on theory) accurately describes reality. From this model, we obtain a series of m parameter-dependent (θ) simulated data points; these will form our **fake data** $\{X_{i,\theta}\}_{i=1}^m$. In essence, *this model will represent our Generator.*

The next task is to assign the correct parameter values to that model. This will be performed by what we call a **Discriminator**. Intuitively, a Discriminator is a function capable of labeling data as either *real* or *fake.* The estimated parameters will be those that best deceive this Discriminator, meaning they make our *fake* series most similar to the *real* series. Formally, a discriminator is defined as a function $D : \mathcal{X} \to [0, 1]$ such that $D(x)$ represents the likelihood of x being an actual observation; $D(x) = 1$ means x is classified as *real* with certainty, while $D(x) = 0$ means x is classified as *fake* with certainty. Later, we'll see that the elicitation of this classifier is not trivial, and the estimator's performance will depend on it. Theoretically, however, the Discriminator could be any function that, given data points, returns 0 or 1. In this document, we are interested in cases where this Discriminator is a Neural Network.

2.2.2 Adversarial Procedure

Formally, the Adversarial Estimator can be expressed with the following min-max problem:

$$\min_{\theta\in\Theta} \max_{D\in\mathcal{D}_n} \mathbb{E}_{X_i\sim P_0}[\log D(X_i)] + \mathbb{E}_{X_{i,\theta}\sim P_\theta}[\log(1 - D(X_{i,\theta}))]$$

In practice, we'll be dealing with the sample counterpart of this, so our estimator will ultimately be represented by:

$$\hat{\theta} = \arg\min_{\theta\in\Theta} \max_{D\in\mathcal{D}_n} \frac{1}{n}\sum_{i=1}^{n} \log D(X_i) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(X_{i,\theta}))$$

Where:

- $\Theta \equiv$ The space of all possible parameters.
- $\mathcal{D}_n \equiv$ The space of all possible Discriminators within the family we are considering.

This estimation can be performed by traditional maximization whenever we're dealing with a Discriminator that has a plain functional form (e.g., a Logit function). However, as we mentioned, here we'll explore cases where the Discriminator is represented by a Neural Network. In such scenarios, a training algorithm must be employed, and things will operate a bit differently (e.g., the maximization procedure will be driven by a Loss function). This brief explanation is just to let the reader know that a more detailed re-explanation will follow, as needed, which is why this subsection currently contains limited technical detail.

3 The Adaptation of Neural Networks for Inter-Dependent Data Points

This section introduces Neural Networks and some of their variants designed to account for serially correlated data. While it serves a didactic purpose, it also implicitly motivates the use of these structures as candidates for our **Discriminator** within the context of time series data.

3.1 Introduction to Neural Networks

Again, our primary goal isn't to provide a technically detailed explanation of what Neural Networks are. Instead, an introductory overview will help to establish a common understanding of what we'll be using and how it operates.

3.1.1 Architecture

Neural Networks are theoretical structures formed by **neurons**, **layers**, and **weights** (along with **biases**). Through a training process, these structures are capable of learning a function that relates some input to some output. A crucial feature is their ability to incorporate and absorb nonlinear relationships within their architecture, reflecting them in their results.

The linear algebra underpinning them can be summarized as follows:

$$\mathbf{a}^{[l]} = \sigma(\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]})$$

Here, $\mathbf{a}^{[l]}$ represents the vector of outputs from layer l (the values assigned to neurons in layer l), while $\mathbf{W}^{[l]}$ and $\mathbf{b}^{[l]}$ represent the weight matrix (weights connecting each neuron in the previous layer to neurons in the current layer) and the vector of biases (typically one fixed value for each neuron in the current layer), respectively, in layer l. The function $\sigma(\cdot)$ transforms each neuron's value within the layer and is responsible for potential nonlinearities in the architecture (e.g., if the function is nonlinear); these functions are called **activation functions**. As you might notice, the term $\mathbf{a}^{[l-1]}$ is also included, indicating a recursive process where all previous layers influence subsequent ones. This creates a framework where complexity can grow exponentially with the number of layers and neurons we choose to include.

For didactic reasons, let's expand this definition for a highly simplified example (Figure 2). This will give us:

- **Input:** $a^0 = x$
- **Hidden Layer:**

$$\sigma(W^1 \cdot a^0 + b^1) = \sigma\left(\begin{bmatrix} w_{11}^1 \\ w_{21}^1 \\ w_{31}^1 \end{bmatrix} \cdot x + \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \end{bmatrix}\right) = \begin{bmatrix} \sigma(w_{11}^1 \cdot x + b_1^1) \\ \sigma(w_{21}^1 \cdot x + b_2^1) \\ \sigma(w_{31}^1 \cdot x + b_3^1) \end{bmatrix}$$

- **Output:**

$$\sigma(W^2 \cdot a^1 + b^2) = \sigma\left(W^2 \cdot \left[\sigma(W^1 \cdot a^0 + b^1)\right] + b^2\right) = \sigma\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \end{bmatrix} \cdot \begin{bmatrix} \sigma(w_{11}^1 \cdot x + b_1^1) \\ \sigma(w_{21}^1 \cdot x + b_2^1) \\ \sigma(w_{31}^1 \cdot x + b_3^1) \end{bmatrix} + b^2\right)$$

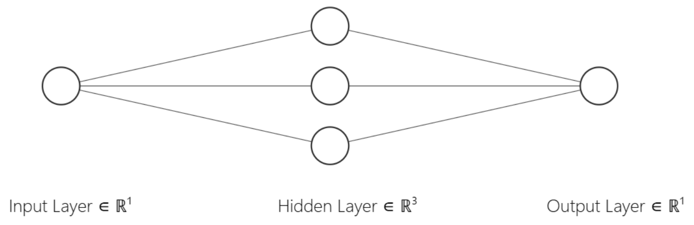

Figure 2: An example of a Simple Neural Network with 1 input, 1 output, and 1 hidden layer with 3 neurons.

So, we end up with an estimated function that looks like this:

$$D(x) = \sigma\left(w_{11}^2 \cdot \sigma(w_{11}^1 \cdot x + b_1^1) + w_{12}^2 \cdot \sigma(w_{21}^1 \cdot x + b_2^1) + w_{13}^2 \cdot \sigma(w_{31}^1 \cdot x + b_3^1) + b^2\right)$$

Here, w_{jk} denotes the weight connecting neuron k in layer $l-1$ to neuron j in layer l.

3.1.2 Training Process

This structure (this functional form) means nothing on its own; what makes Neural Networks truly useful is the method they employ to determine these parameter values (weights and biases). We refer to this 'learning' process as Neural Network Training. To illustrate the importance of a proper training method, consider that their popularity only truly bloomed once Rumelhart, Hinton, and R. J. Williams 1986 developed a reliable learning method: **Backpropagation**, which (with its variations) is still used today.

Any Backpropagation-related learning method is driven by the idea of **gradient descent**, which means precisely what it sounds like: we follow a method that iteratively reduces the gradient. At this point, one might ask, the gradient of what? This question brings us to one of the most important features of any supervised machine learning process: the **Loss function**. Loss functions tell the Neural Network how accurate its estimation is. In simple terms, this function must be an error measure between predicted outputs and real observed data. We are interested in decreasing the gradient of this Loss function to reach a point where the error is minimized. So, this process attempts to estimate the Neural Network parameters by updating them in a direction where, at each step, the gradient becomes less steep. Formally, the updating (learning) process can be described as:

$$\theta^* = \theta - \eta \left(\frac{\partial \mathcal{L}}{\partial \theta} \right)$$

Where θ represents the Neural Network parameters, $\mathcal{L}$ is the Loss function, and η is the **Learning Rate**, which dictates the size of the descent step.

As you might notice, this procedure involves partial derivatives, and since we are dealing with a recursive structure where many elements are intertwined, the **chain rule** will be pivotal in deriving these updates.

For a more compact notation, assume that $\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$ and thus $\mathbf{a}^{[l]} = \sigma(\mathbf{z}^{[l]})$. Then, continuing with the previous example, our updating values will be computed in the following manner:

- **Output Layer (Layer 2):**

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(2)}} = \frac{\partial \mathcal{L}}{\partial D(x)} \cdot \frac{\partial D(x)}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w_{jk}^{(2)}}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \frac{\partial \mathcal{L}}{\partial D(x)} \cdot \frac{\partial D(x)}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial b^{(2)}}$$

- **Hidden Layer (Layer 1):**

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{(1)}} = \underbrace{\left(\frac{\partial \mathcal{L}}{\partial D(x)} \cdot \frac{\partial D(x)}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a_j^{(1)}} \right)}_{\frac{\partial \mathcal{L}}{\partial a_j^{(1)}}} \cdot \frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} \cdot \frac{\partial z_j^{(1)}}{\partial w_{jk}^{(1)}}$$

$$\frac{\partial \mathcal{L}}{\partial b_j^{(1)}} = \underbrace{\left(\frac{\partial \mathcal{L}}{\partial D(x)} \cdot \frac{\partial D(x)}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a_j^{(1)}} \right)}_{\frac{\partial \mathcal{L}}{\partial a_j^{(1)}}} \cdot \frac{\partial a_j^{(1)}}{\partial z_j^{(1)}} \cdot \frac{\partial z_j^{(1)}}{\partial b_j^{(1)}}$$

Notice that the deeper the Neural Network (the more neurons and hidden layers we introduce), the larger the size of the nested relationships. This is because the impact of more immediate neurons on the final output (and thus on the Loss function) will be carried through a vast number of subsequent relationships. This relates to the earlier comment on exponentially increasing complexity.

3.2 Introduction to Recurrent Neural Networks

Unlike standard Neural Networks that treat each input value as independent of the rest of the sample, **Recurrent Neural Networks (RNNs)** were created to account for serial correlation in data.

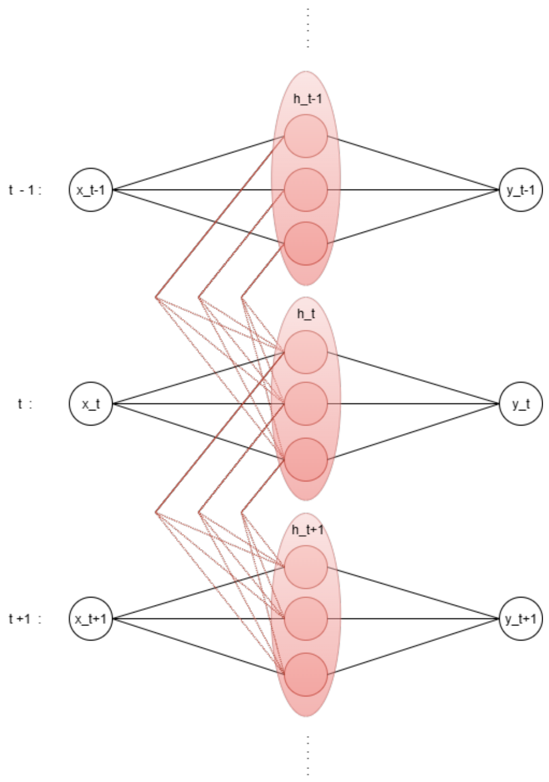

Figure 3: An example of a Simple Recurrent Neural Network with 1 input, 1 output, and a hidden state with 3 neurons. Illustrated for some periods.

3.2.1 Architecture

To account for these inter-dependencies in data, the structure of the Neural Network changes slightly, or rather, adds another dimension: **time (t)**. Time is introduced through an additional layer called the **hidden state**. This is the main feature of RNNs and is responsible for keeping track of previous results. Specifically, this hidden state is a dynamic function of previous inputs and previous hidden states. Thus, the Neural Network will take some input at each step (each period) and produce some output, while the hidden state continuously enriches its memory, thereby enhancing subsequent decisions with past experience. This is why these types of structures are said to absorb temporal relationships: the hidden state acts like a local memory (a past experience tracker). Perhaps Figure 3 is helpful for a more visual understanding.

In general, you can think of an RNN as a series (as many as the number of periods we have) of Neural Networks chained by this hidden state. The idea is that the final output is refined at each period given our sequential inputs, so in practice, the final (period-wise) output is taken as the actual output for the entire RNN.

The structure is also founded on weights and biases. These parameters will describe three possible relationships: Inputs with Hidden State, Current Hidden State with Past Hidden State, and Current Hidden State with Outputs. We end up with the following functional form (for a particular period t):

$$h_t = \sigma(Wh_{t-1} + Ux_t + b_h)$$
$$y_t = \sigma(Vh_t + b_y)$$

Here, W represents the weights relating hidden states, U represents the weights relating inputs with the hidden state, V represents the weights relating the hidden state with outputs, and b_h and b_y are the biases for the current hidden state and the output, respectively. Notice that none of them appear with a time subscript; this is not by chance: **in RNNs, weights and biases are shared across all periods**; they are fixed values, not dynamic. Also, observe that:

$y_t = \sigma(Vh_t + b_y)$
$= \sigma(V(\sigma(Wh_{t-1} + Ux_t + b_h)) + b_y)$
$= \sigma(V(\sigma(W(\sigma(Wh_{t-2} + Ux_{t-1} + b_h)) + Ux_t + b_h)) + b_y)$
$= \dots$

This should remind you of the recursive structure of classical Neural Networks; in a sense, this is the same concept, but now relationships occur in a "vertical" (across time) way rather than a "horizontal" (across layers within a single time step) one.

3.2.2 Training Process

This part is challenging. We won't delve deeply into how these structures are actually trained (the derivations become more complex and are not the focus of this work). However, to provide some insight, the common procedure is also based on gradient descent and is called **Backpropagation Through Time (BPTT)**. The difference lies in the computation of the derivatives, as the presence of hidden states complicates matters slightly, and for these recurrent models, the Loss function is typically a summation of all local Losses across time steps.

Nonetheless, a significant issue arises in training these models, almost purely related to a phenomenon called **vanishing gradient** (Pascanu, Mikolov, and Bengio 2013). Intuitively, this problem occurs when the Network attempts to keep track of long-past periods, and due to its construction, it becomes unable to learn from those periods, ultimately losing its memory ability. This is the main reason why basic RNNs are not used directly, but rather through their variants.

3.3 LSTM Networks

We hope the above explanation serves as a motivation for applying RNNs in a time series context, where data dependency is crucial for identification. Even though the idea behind these primitive RNNs is powerful, in practice, it introduces many issues (as mentioned before). To solve these problems, variations were proposed. In particular, one of the most widely used is the **LSTM Network (Long Short-Term Memory Network)**, which solves the vanishing gradient problem, and is what we will use. The justification for using LSTMs is more practical than theoretical. With this, we want to make it clear that the real improvement lies in adapting the Discriminator with a Recurrent Structure; the specific architecture of that Recurrent Structure will certainly affect the estimator's performance, but in a practical way (the core idea should hold for any recurrent type). That being said, we will now explain the features that LSTMs bring (Figure 4).

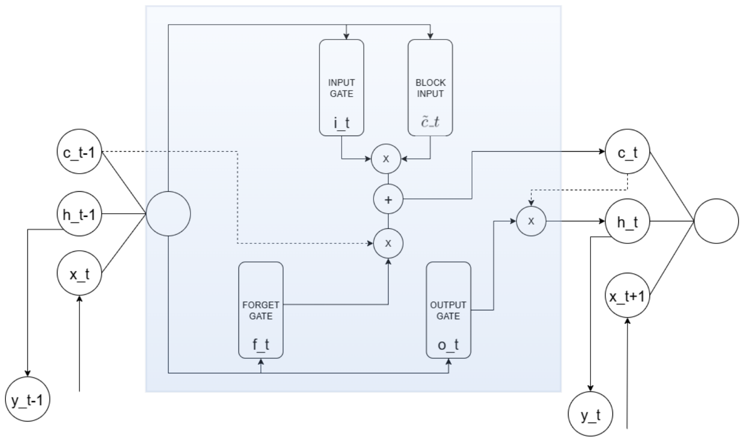

Figure 4: LSTM architecture. The blue box represents the LSTM Cell. Illustrated for input and output of one period.

LSTM Networks, as their name suggests, were designed to allow for self-selection of memory use. This means these Neural Networks incorporate mechanisms that mimic the decision of whether to activate (Long-Term) or deactivate (Short-Term) the influence of past experience (the hidden state). This is achieved by constructing a structure based on **Cells** and **Gates**.

You can think of Cells as each time/period slot. Inside these Cells, a team of Gates will process different aspects of the time series inputs. Each Gate will have the following functional form:

- **Input Gate:**

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + (p_i \odot c_{t-1}) + b_i)$$

- **Forget Gate:**

$$f_t = \sigma(W_f h_{t-1} + U_f x_t + (p_f \odot c_{t-1}) + b_f)$$

- **Output Gate:**

$$o_t = \sigma(W_o h_{t-1} + U_o x_t + (p_o \odot c_{t-1}) + b_o)$$

- **Block Input (Candidate Cell State):**

$$\tilde{c}_t = \tanh(W_c h_{t-1} + U_c x_t + b_c)$$

Where i_t, f_t, and o_t are signals bounded from 0 to 1, and $\tilde{c}_t$ is a signal bounded from -1 to 1. All are used for controlling the impact of certain values. Then

p_i, p_f, p_o are **peephole weights**; they control how much of the previous cell state c_{t-1} influences the current gate. This control is represented by the element-wise multiplication ($\odot$). c_t represents the current cell state value (computed later), but the rest of the notation is consistent with previous analysis.

Also, notice that even though the primary inputs to all gates are the same, the way each gate affects the final output is different. Consequently, the optimal updates will vary, and the parameters related to each gate will also be distinct.

These Gates are then used to compute the values that actually form and feed the Cell:

- **Current Cell State:**

$$c_t = (f_t \odot c_{t-1}) + (i_t \odot \tilde{c}_t)$$

- **Hidden State (Output of the Cell):**

$$h_t = o_t \odot \tanh(c_t)$$

- **Block Output (Final output of the LSTM layer for this step):**

$$y_t = V h_t + b_y$$

We end up with the following parameters to learn:

- **Input-to-Hidden weights:** $\mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_o, \mathbf{U}_c$
- **Hidden-to-Hidden weights:** $\mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_c$
- **Peephole weights:** $\mathbf{p}_i, \mathbf{p}_f, \mathbf{p}_o$
- **Biases:** $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o, \mathbf{b}_c, \mathbf{b}_y$ (if b_y is used for y_t)

As you can see, these additional features add a lot of complexity to the Neural Network, and consequently, training can be computationally intensive. However, it's true that by introducing this cell state (c_t), the Neural Network becomes capable of self-managing the importance of new data relative to past data, which represents a gain in 'intelligence' (i.e., a better understanding of the serial-dependence structure).

4 A Modified Discriminator for Time Series

This section outlines the primary focus of this document and what we consider to be its core contribution (or at least the proposed direction): a suitable method to adapt the **Adversarial Estimator** (specifically, its classification component) for accurate performance with time series data.

As Kaji, Manresa, and G. A. Pouliot 2021 concluded, the design of the **Discriminator** is pivotal for the estimator's performance. A poorly constructed classifier can lead to a shallow understanding of the true (real) data distribution, resulting in inadequate parameter matching. This intuition is further explored in Kaji, Manresa, and G. Pouliot 2023, where they address the question: "What if $\mathcal{D}$ Is Not Rich Enough?" They conclude that the efficiency and asymptotic properties of the Adversarial Estimator depend on $\mathcal{D}$ eventually being able to reach what is termed the *Oracle Discriminator* (D_θ). This Oracle Discriminator represents an impossible-to-build optimal classifier (as its construction would require prior knowledge of the real data's actual distribution). If D_θ is not included within $\mathcal{D}$ (the family of discriminators under consideration), the estimator cannot achieve efficiency. Therefore, careful consideration of the Discriminator is crucial, and given the inherent structure of time series, some adaptation is necessary. This adaptation should focus on designing a Discriminator capable of capturing the intertemporal dependencies that characterize such data.

As previously motivated, our proposed solution is to utilize an **LSTM Network** as the Discriminator. Theoretically, this adaptation should at least be sufficient to manage the time series data dependency and thereby help the procedure perform an accurate estimation. We will validate this hypothesis by running several simulations, hoping to verify that our conceptual framework holds true, at least in a controlled environment.

4.1 Block Bootstrap

BUT before continuing, one thing must be addressed:

The training process for Neural Networks, including the LSTM networks used for our Discriminator, typically assumes an independent and identically distributed (i.i.d.) training set. However, in time series analysis, researchers commonly work with a single, long series. If we simply "chop" this long series into subseries to create a training dataset, these subsets will inherently exhibit serial correlation, violating the i.i.d. assumption required for optimal Neural Network training. This correlation among the training set for the Discriminator can lead to issues with convergence and hinder proper performance.

To overcome this challenge and construct an approximately i.i.d. training sample while preserving the underlying correlation structure of the time series, we employ the *Block Bootstrap* method. Block Bootstrap works by dividing the original time series into blocks and then resampling these blocks with replacement to create new series. This approach allows the Discriminator to be trained on a dataset that better approximates the i.i.d. assumption, thereby improving the stability and accuracy of the adversarial estimation process. This could seem as something minor, but its omission brings convergency issues and miss-accuracy for the hole Adversarial Procedure.

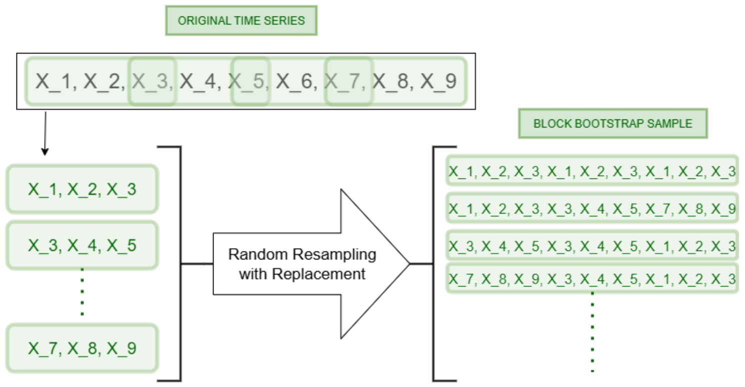
ORIGINAL TIME SERIES
X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9
BLOCK BOOTSTRAP SAMPLE
X_1, X_2, X_3
X_3, X_4, X_5
X_7, X_8, X_9
Random Resampling with Replacement
X_1, X_2, X_3, X_1, X_2, X_3, X_1, X_2, X_3
X_1, X_2, X_3, X_3, X_4, X_5, X_7, X_8, X_9
X_3, X_4, X_5, X_3, X_4, X_5, X_1, X_2, X_3
X_7, X_8, X_9, X_3, X_4, X_5, X_1, X_2, X_3

Figure 5: Illustration of the Block Bootstrap Sampling Process.

4.2 Used Algorithm

This subsection details the practical implementation of our Adversarial process. The incorporation of a Neural Network necessitates a more ”machine-like” (programming-oriented) mindset, requiring an understanding of the iterative looping criterion.

As discussed in previous sections, the adaptation of the Adversarial Estimator for time series focuses on integrating an LSTM network as the Discriminator. A critical aspect for training neural networks with serially correlated data is the preparation of the training samples to approximate independent and identically distributed (i.i.d.) conditions. This is achieved through resampling with replacement using the Block Bootstrap method. The estimation procedure then follows an iterative approach where the Discriminator and the Generator are competitively trained. The objective is to find the Generator’s parameters that best ”fool” the Discriminator, meaning the simulated data becomes indistinguishable from the real data. The general algorithm used is inspired by the Generative Adversarial Networks (GANs) framework and is adapted for time series parameter estimation. The algorithm is presented below in a format similar to the works of Kaji, Manresa, and G. Pouliot 2023:

Algorithm 1 Adversarial estimation with a neural network discriminator for time series data

```
Input: $X_{master}$: master observed time series (long series), $T_\theta$: generator model,
       $D$: discriminator
Input: $\eta_D$, $\eta_G$: learning rates; $N_{\text{gen\_steps}}$, $N_{\text{epochs\_D}}$: iteration parameters
Input: $B$: batch size (for mini-batches), block_size (for Block Bootstrap), se-
       ries_length (for individual samples), $N_{\text{samples}}$ (number of samples per
       step), tolerance_params, patience
Output: $\hat{\theta}$: estimated generator parameters
1  Initialize $\theta_0$, $D$ with random weights $k \leftarrow 0$, epochs_no_improve_gen $\leftarrow 0$,
   prev_theta $\leftarrow \theta_0$
2  while $k < N_{gen\_steps}$ and epochs_no_improve_gen < patience do
3      Generate simulated data $X_{sim} = T_{\theta_k}(Z)$ with $Z \sim P_Z$ (of
       size $N_{\text{samples}}$) Generate bootstrapped real data $X_{\text{real\_boot}}$ =
       BlockBootstrap($X_{master}$, block_size, series_length, $N_{\text{samples}}$)
4      for epoch = 1 to $N_{epochs\_D}$ do
5          Combine real and simulated data: $X_{\text{combined}} = [X_{\text{real\_boot}}, X_{sim}]$ Assign
           labels: $y_{\text{real}} = 1$, $y_{\text{sim}} = 0$ Create minibatches of $(X_{\text{combined}}, y_{\text{combined}})$
           foreach minibatch $(x_b, y_b)$ do
6              Compute $D(x_b)$ Compute discriminator loss: $\mathcal{L}_D =$
               $-\frac{1}{B}\sum_{i=1}^{B}[y_b \log D(x_b) + (1 - y_b)\log(1 - D(x_b))]$ Update $D$
               using gradient descent with $\eta_D$
7      Generate $X_{\text{sim\_for\_grad}} = T_{\theta_k}(Z)$ Compute $D(X_{\text{sim\_for\_grad}})$ and loss $\mathcal{L}_G$ =
       $\frac{1}{M}\sum_{i=1}^{M}\log(1 - D(X_{\text{sim\_for\_grad}}))$ Update $\theta_k$: $\theta_{k+1} \leftarrow \theta_k - \eta_G \nabla_\theta \mathcal{L}_G$
8      if $|\theta_{k+1} - prev\_theta| < tolerance\_params$ then
9          epochs_no_improve_gen $\leftarrow$ epochs_no_improve_gen +1
10     else
11         epochs_no_improve_gen $\leftarrow 0$
12     prev_theta $\leftarrow \theta_{k+1}$  $k \leftarrow k + 1$
13 return $\hat{\theta} = \theta_k$
```

Key elements of the algorithm are listed here:

- **Master Real Data (X_{master})**: This refers to the single, long observed time series from which the real data for training the Discriminator is derived. This series is assumed to represent the true underlying process.

- **Generator (T_θ)**: This is our parametric model, which attempts to replicate the distribution of the real data. It is parameterized by θ. The Generator takes random noise (Z) and transforms it into simulated time series samples, each of a predefined 'series_length'.

- **Discriminator (D)**: This is a **Long Short-Term Memory (LSTM)** neural network. Its function is to classify input time series (samples of 'series_length‘) as ”real” (coming from X_{master} via bootstrapping) or ”fake” (generated by T_θ).

- **Real Data Preparation (Block Bootstrap)**: Crucially, in each generator step, a new set of ”real” training samples ($X_{\text{real\_boot}}$) is created by applying the **Block Bootstrap** method to the X_{master} series (Algorithm 1, line 3). This involves dividing the master series into blocks and then resampling these blocks with replacement to generate N_{samples} sub-series, each of ‘series_length‘. This procedure helps to break the serial correlation between training batches, approximating an i.i.d. setting for the Discriminator's optimization, while preserving the internal temporal dependencies within each sample.

- **Discriminator Training**: In each generator step, the Discriminator is trained for $N_{\text{epochs\_D}}$ steps to distinguish between the bootstrapped real data ($X_{\text{real\_boot}}$) and data simulated by the Generator in its current state (X_{sim}). The **Binary Cross-Entropy (BCE) Loss function** is used, where the Discriminator tries to maximize $\log D(X_i)$ for real data and $\log(1 - D(X_{i,\theta}))$ for simulated data.

- **Generator Update**: After training the Discriminator, the Generator's parameters are updated. Its goal is to ”fool” the Discriminator by generating data that the Discriminator classifies as real. This is achieved by minimizing $\log(1 - D(X_{\text{sim}}))$. Backpropagating the gradient of this loss function through the Discriminator (without updating the Discriminator's weights in this step) allows θ to be adjusted in the correct direction.

- **Stopping Criterion**: The Generator's training process stops when the estimated parameter θ stabilizes. This stabilization is monitored by a tolerance threshold for the change in θ between consecutive steps, and a ”patience” parameter that defines how many steps without significant improvement are allowed before termination.

4.3 Simulation Results

The empirical results of our proposed adversarial estimation approach for time series data, particularly after incorporating the Block Bootstrap method, have shown more promising convergence behavior compared to initial attempts. While still computationally demanding and requiring careful hyperparameter tuning, the stability issues previously observed have been significantly mitigated. However, conducting extensive dispersion measurements, such as those typically derived from Monte Carlo simulations (which would require numerous successful runs), remains computationally intensive and was not feasible within the scope of this work. Consequently, we present only a limited set of qualitative results.

Nevertheless, these illustrations, in the form of tracking the parameter values over generator steps for various time series models, offer crucial insights into the estimator's behavior and suggest that this adapted framework is indeed a viable direction for future research, even if full convergence or statistical robustness cannot be fully demonstrated at this stage.

The general setup for the following simulations is as follows: A total sample size of 1000 is used, consisting of 500 real (simulated from the true model) data points and 500 generated data points.

- **Moving Average Model (MA(1))**

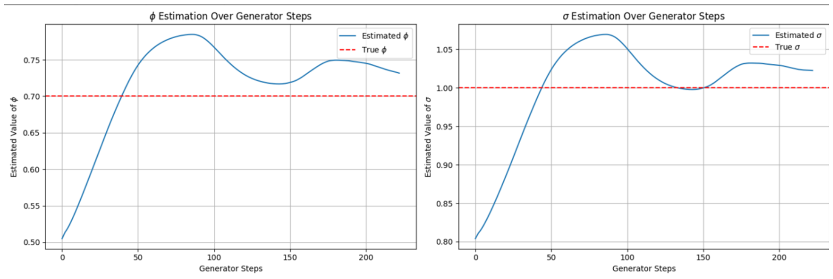

Figure 6: Adversarial Estimation process for a MA(1) model. The plot shows the estimated values of ϕ and σ over generator steps. True values are indicated by the red dashed line. Estimated $\hat{\phi} = 0.7317$ (True $\phi = 0.7$).Estimated $\hat{\sigma} = 1.024$ (True $\sigma = 1$). Sample size: 1000 (500 real, 500 generated). Block-size for training: 10.

- **Autoregressive Model (AR(1))**

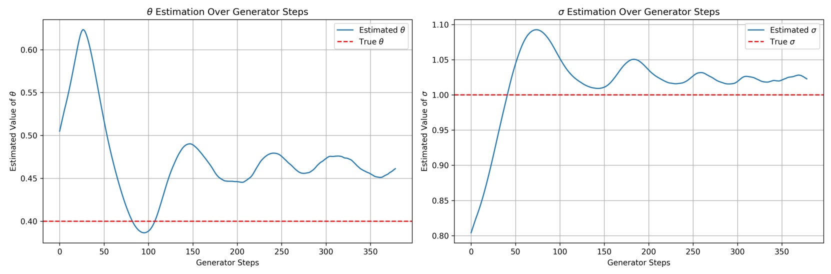

Figure 7: Adversarial Estimation process for an AR(1) model. The plot shows the estimated values of θ and σ over generator steps. True values are indicated by the red dashed line. Estimated $\hat{\theta} = 0.4613$ (True $\theta = 0.4$).Estimated $\hat{\sigma} = 1.0229$ (True $\sigma = 1$). Sample size: 1000 (500 real, 500 generated). Block-size for training: 25.

- **Autoregressive Model (AR(2))**

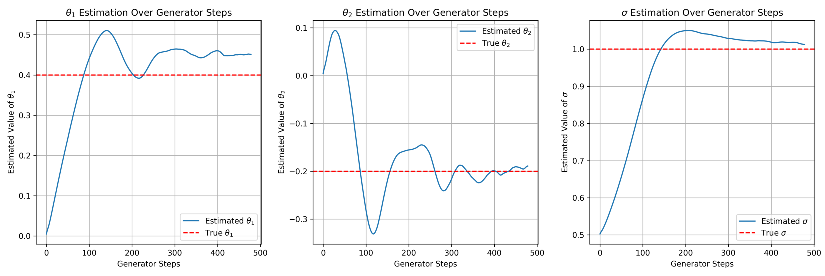

Figure 8: Adversarial Estimation process for an AR(2) model. The plot shows the estimated values of θ_1, θ_2, and σ over generator steps. True values are indicated by the red dashed line. Estimated $\hat{\theta_1} = 0.4512$ (True $\theta_1 = 0.4$). Estimated $\hat{\theta_2} = -0.1889$ (True $\theta_2 = -0.2$). Estimated $\hat{\sigma} = 1.0127$ (True $\sigma = 1$). Sample size: 1000 (500 real, 500 generated). Block-size for training: 10.

These preliminary results, while not conclusive regarding the estimator's asymptotic properties or efficiency due to the limited scope of simulations and absence of robust dispersion measures (such as those from Monte Carlo studies), highlight the significant potential of applying adversarial estimation to time series when the Discriminator's training data adequately accounts for serial correlation. The observed trends, where estimated parameters oscillate around their true values, strongly suggest that the proposed adaptation, particularly with the Block Bootstrap, points towards the correct path for developing a stable and accurate adversarial estimator for time series contexts. These findings delineate clear avenues for future research, including more extensive statistical evaluations, adaptive hyperparameter tuning, and refined Discriminator architectures.

5 Comments and Conclusion

The empirical results derived from our proposed adversarial estimation approach for time series data, especially after incorporating the Block Bootstrap method, have shown more promising convergence behavior compared to initial attempts. While still computationally demanding and requiring careful hyperparameter tuning, the stability issues previously observed have been significantly mitigated. However, conducting extensive dispersion measurements, such as those typically derived from Monte Carlo simulations (which would require numerous successful runs), remains computationally intensive and was not feasible within the scope of this work. Consequently, we present only a limited set of qualitative results. Nevertheless, these illustrations, in the form of tracking the parameter values over generator steps for various time series models, offer crucial insights into the estimator's behavior and suggest that this adapted framework is indeed a viable direction for future research, even if full convergence or statistical robustness cannot be fully demonstrated at this stage.

The design of the Discriminator is pivotal for the estimator's performance. A poorly constructed classifier can lead to a shallow understanding of the true (real) data distribution, resulting in inadequate parameter matching. The efficiency and asymptotic properties of the Adversarial Estimator depend on D eventually being able to reach what is termed the Oracle Discriminator (D_θ). If D_θ is not included within D (the family of discriminators under consideration), the estimator cannot achieve efficiency. Therefore, careful consideration of the Discriminator is crucial, and given the inherent structure of time series, some adaptation is necessary. This adaptation should focus on designing a Discriminator capable of capturing the intertemporal dependencies that characterize such data.

As previously motivated, our proposed solution is to utilize an LSTM Network as the Discriminator. Theoretically, this adaptation should at least be sufficient to manage the time series data dependency and thereby help the procedure perform an accurate estimation. We will validate this hypothesis by running several simulations, hoping to verify that our conceptual framework holds true, at least in a controlled environment.

These preliminary results, while not conclusive regarding the estimator's asymptotic properties or efficiency due to the limited scope of simulations and absence of robust dispersion measures (such as those from Monte Carlo studies), highlight the significant potential of applying adversarial estimation to time series when the Discriminator's training data adequately accounts for serial correlation. The observed trends, where estimated parameters oscillate around their true values, strongly suggest that the proposed adaptation, particularly with the Block Bootstrap, points towards the correct path for developing a stable and accurate adversarial estimator for time series contexts. These findings delineate clear avenues for future research, including more extensive statistical evaluations, adaptive hyperparameter tuning, and refined Discriminator architectures.

References

Cigliutti, Ignacio and Elena Manresa (2022). *Adversarial Method of Moments*. Tech. rep. Working Paper. New York University (NYU). URL: `https://www.nachocigliutti.com/research/amm/`.

Ghojogh, Benyamin and Ali Ghodsi (2023). *Recurrent Neural Networks and Long Short-Term Memory Networks: Tutorial and Survey*. To appear as a part of an upcoming textbook on deep learning. University of Waterloo working paper/tutorial. URL: `https://www.youtube.com/@bghojogh%20--%20Ver%20canales%20de%20YouTube%20de%20los%20autores%20para%20posibles%20actualizaciones%20o%20recursos%20relacionados.`.

Goodfellow, Ian J. et al. (2014). "Generative Adversarial Nets". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems*. Vol. 2. NIPS '14. Cambridge: MIT Press, pp. 2672–2680.

Greff, Klaus et al. (2017). "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10, pp. 2221–2232. DOI: `10.1109/TNNLS.2017.2695635`.

Kaji, Tetsuya, Elena Manresa, and Guillaume Pouliot (2023). "An Adversarial Approach to structural Estimation". In: *Econometrica* 91.6, pp. 1–23.

Kaji, Tetsuya, Elena Manresa, and Guillaume A. Pouliot (2021). "Adversarial Inference Is Efficient". In: *AEA Papers and Proceedings* 111, pp. 621–625. DOI: `10.1257/pandp.20211037`.

Lillicrap, Timothy P. and Adam Santoro (2019). "Backpropagation through time and the brain". In: *Current Opinion in Neurobiology* 55, pp. 82–89. DOI: `10.1016/j.conb.2019.01.011`.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio (2013). "On the difficulty of training recurrent neural networks". In: *arXiv preprint arXiv:1312.6103*.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323, pp. 533–536.

Williams, Cameron (2024). "Neural Networks to Hedge and Price Stock Options". Supervised by Dr Bay Chao. Master's Thesis. School of Mathematics, The University of Manchester.