Constructing Efficient Simulated Moments Using Temporal Convolutional Networks

Jonathan Chassot* and Michael Creel†

February 16, 2024

Link to latest version.

Abstract

We propose a method to estimate model parameters using temporal convolutional networks (TCNs). By training the TCN on simulated data, we learn the mapping from sample data to the model parameters that were used to generate this data. This mapping can then be used to define exactly identifying moment conditions for the method of simulated moments (MSM) in a purely data-driven manner, alleviating a researcher from the need to specify and select moment conditions. Using several test models, we show by example that this proposal can outperform the maximum likelihood estimator, according to several metrics, for small and moderate sample sizes, and that this result is not simply due to bias correction. To illustrate our proposed method, we apply it to estimate a jump-diffusion model for a financial series.

Keywords: temporal convolutional networks, method of simulated moments, jump-diffusion stochastic volatility, neural networks
JEL classification: C15, C45, C53, C58

*Faculty of Mathematics and Statistics, University of St.Gallen, Switzerland. jonathan.chassot@unisg.ch

†Universitat Autònoma de Barcelona, Barcelona School of Economics, and MOVE, Bellaterra (Barcelona) 08193, Spain. michael.creel@uab.cat.

1 Introduction

Understanding real-world phenomena through the lens of applied economics often necessitates the construction of comprehensive models that accurately reflect the inherent complexity of their empirical counterparts. However, an increase in model complexity often compromises the simplicity of statistical inference, posing a significant challenge for economists and econometricians. Particularly, when estimating parameters of an intricate model where a tractable likelihood and closed-form theoretical moments are non-existent, the challenge becomes even more substantial, thus warranting specialized methodologies. A widely used instance of such a methodology is the simulated method of moments (MSM; McFadden, 1989; Pakes & Pollard, 1989), which uses simulated data to replicate moments of the model. Similar to its predecessor, the generalized method of moments (GMM; Hansen, 1982), the choice of moment conditions is crucial in determining the estimator's efficiency, and it is an area that has received considerable attention in the literature. While it is well-known that overidentifying moment conditions do not harm the estimator's consistency, they can result in a significant increase in bias and variance in finite samples (Donald, Imbens, & Newey, 2009). To counteract this issue, several authors have put forth methods to select the best moment conditions from an overidentifying set (e.g., Carrasco, 2012; Cheng & Liao, 2015; DiTraglia, 2016; Hall, 2015). However, choosing the initial set of possible moment conditions remains a challenge, and the literature has not yet reached a consensus on the best approach to this problem. One fundamental reason this problem persists is rooted in the vast set of possible moments from which one could select. Owing to such complexity, a universally applicable method for this initial selection process remains elusive in the current literature.

This paper contributes to this ongoing research by proposing a data-driven approach that utilizes deep neural networks to formulate the set of moment conditions. We aim to construct an exactly identifying set of statistics that are not only informative but also lead to an MSM estimator that is approximately fully asymptotically efficient. By "construct", we mean that our method bears the advantage of circumventing the challenges of specifying and selecting the best moment conditions in traditional MSM methodologies. Instead, the ideal moment conditions are learned in an end-to-end manner by the network, which is trained on simulated data. To facilitate the application of our methodology, we have implemented our approach in a Julia package. This package is available for the research community and can be accessed at https://github.com/JLDC/DeepSimulatedMoments.jl, providing a practical tool for researchers and practitioners to apply our proposed method in various contexts.

While our primary focus lies within the framework of MSM and the Bayesian approach to method of moments estimators as established by Kwan (1999), Kim (2002) and Chernozhukov and Hong (2003), the technique we propose holds broad applicability to other estimation methods that rely on informative statistics such as Approximate Bayesian Computation (ABC; see Grazian & Fan, 2020; Sisson, Fan, & Beaumont, 2018, for a recent survey and textbook treatment, respectively) or Indirect Inference (II; Gourieroux, Monfort, & Renault, 1993; Gouriéroux & Monfort, 1997; Smith, 1993).

Our method involves fitting a deep neural network that leverages the entire raw sample data as input and generates the model's parameters as output. While the network's output is a point estimate of the true parameter vector, its interpretation as an exactly identifying moment condition allows for a straightforward application in the MSM framework and other simulation-based methods that allow for statistical inference. Although neural networks have been applied for parameter estimation before (e.g., Akesson et al., 2021; Creel, 2017, 2021; Fisher et al., 2020; B. Jiang et al., 2017; Wiqvist et al., 2019), our work represents the first application of temporal convolutional networks (TCNs) in this context. We demonstrate that our end-to-end TCN-based moment conditions offer superior performance compared to previously used approaches and lead to an estimator that is competitive with maximum likelihood estimators.

This paper is organized as follows. In Section 2, we review the literature on neural network-based parameter estimation, while Section 3 introduces the deep neural network architectures on which we focus. In Section 4, we present the results of our network-based estimators compared to maximum likelihood estimates for three test models with tractable likelihoods. Section 5 provides an empirical example, where we use the TCN-based moments to estimate a jump-diffusion model for S&P 500 data. Finally, Section 6 concludes the paper and discusses avenues for future research.

2 Neural Networks for Parameter Estimation

Neural networks are versatile tools with a wide range of applications. Several papers have used neural networks to estimate the parameters of statistical models, and many of them have presented results for the moving average order 2 model (presented below, in Section 4.1). After summarizing the methodologies, we present the results to give a rough and partial summary of the performance achieved thus far for this simple model.

In the first approach, researchers summarize the sample data into a vector of statistics before passing it to the network. The input is of fixed size, which means that the neural network can remain relatively simple (though it may still have hundreds or thousands of parameters) and will be comparatively quick and easy to train. Blum and François (2010) are the first to implement this technique of which we know. They use a single hidden layer feedforward neural network (FNN)[1] with only four nodes. In their work, kernel regression ameliorates the relatively poor fit that can be obtained from such a rudimentary network by additional processing of the network's output. Creel (2017, 2021) use FNNs that are much larger, with multiple hidden layers and hundreds of nodes per layer. These papers find that the FNN method can give relatively precise and reliable inferences on the parameters of the models under scrutiny. However, the approach that uses statistics as the input to the network is limited by the quality of the statistics used to summarize the sample information before it is provided to the network. If the statistics are approximately sufficient, then one would expect little loss of efficiency. However, ensuring that the chosen statistics are at least close to being sufficient may be difficult or impossible.

The second approach is more ambitious since it utilizes the complete sample information without summarization that may cause information loss. This method requires the network's input dimension to equal the number of variables the model provides multiplied by the sample size. As a result, the input size is significantly larger than in the first approach and grows with the sample size. Therefore, a sizeable and intricate network is required to adequately process the more complex form of the inputs. B. Jiang et al. (2017) was the first paper to use this technique, where the complete sample data serves as the input to the network. They use a simple FNN in which all observations are presented as an input vector without any positional encoding. However, the lack of structured information that identifies the boundaries between observations in the FNN architecture hinders its ability to learn the underlying structure of variables in the model. Other researchers have utilized more intricate neural network architectures, which are capable of processing structured data and are better suited for learning patterns and sequences. Fisher et al. (2020) use recurrent neural networks (RNNs)[2] to learn the quantiles of the parameters[3], while Akesson et al. (2021) use

[1]See Appendix D.1 for a formal definition of FNNs.

[2]See Appendix D.2 for a formal definition of RNNs.

[3]Quantiles may be learned using the "check function" or "pinball" loss function to train the network instead of the more commonly used mean-square error (MSE) loss function or similar variants, which target

convolutional neural networks (CNNs)[4] to tackle this problem. Finally, Wiqvist et al. (2019) propose using a neural network design called partially exchangeable networks (PENs), which may be appropriate when data has a Markovian structure. As such, this proposal considers structure in the data set, and the resulting model may benefit from this information. While these architectures can more easily learn from the structure of a data set than the simple FNN approach, they do so in different ways. RNNs and CNNs are well-known and widely used networks, most often for applications other than learning parameters, such as speech and image recognition (e.g. Goodfellow, Bengio, & Courville, 2016; Graves, Mohamed, & Hinton, 2013; Krizhevsky, Sutskever, & Hinton, 2012).

To end this section, we summarize existing results on parameter estimation that use the two approaches and motivate the methods put forth in this paper. We present a simple comparison based on a straightforward order 2 moving average model (referred to as MA(2), and presented in detail in Section 4.1, below). For this DGP, using samples of size $T = 100$, Akesson et al. (2021), in their Table I, present a comparison of the performance of their methods with those of B. Jiang et al. (2017) and Wiqvist et al. (2019). The measure for comparison is the mean absolute error (MAE), normalized by the MAE of the prior mean. Akesson et al. (2021) compute the normalized MAE for the jth component of the parameter vector $\boldsymbol{\theta}_i = (\theta_{i,1}, \theta_{i,2})^\top$ as follows[5]:

$$E_{\%} \approx \frac{4}{b_j - a_j} \frac{1}{n} \sum_{i=1}^{n} \left| \theta_{i,j} - \hat{\theta}_j(y_i) \right|, \qquad \theta_{i,j} \sim \text{Unif}(a_j, b_j) \tag{1}$$

where n is the number of simulated samples, y_i is the entire i^{th} simulated sample generated at the parameter value $\boldsymbol{\theta}_i$, drawn from a prior uniform distribution with lower and upper bounds $\mathbf{a} = (a_1, a_2)^\top$, and $\mathbf{b} = (b_1, b_2)^\top$, respectively, and $\hat{\boldsymbol{\theta}}(y_i) = (\hat{\theta}_1(y_i), \hat{\theta}_2(y_i))^\top$ is the output of the network given the ith simulated sample.[6]

the posterior mean, rather than posterior quantiles.

[4]See Appendix D.3 for a formal definition of CNNs.

[5]See equation (8) in their paper.

[6]The normalization factor is computed by simply using the lower and upper bounds for the MA(2) model's two parameters, treating the prior's support as a rectangle. Although the true support of the prior is the invertible region, i.e., a triangle, we maintain the original simplification as it makes our results comparable and provides a more conservative estimate than the true support. Appendix C shows how this normalizing

The measure in (1) is averaged across the two parameters of the MA(2) model. In Table 1, we reproduce the information from Table I of Akesson et al. (2021) and add results for the methods of Creel (2017) and the temporal convolutional networks proposed in this work. We also add the maximum likelihood estimator (MLE) for reference. The neural network results are based on 10^6 training samples. The results in Table 1 show that the approach based on summary statistics used in Creel (2017) can deliver competitive performance to the other methods reported, which all use the complete sample as input. However, this approach bears limitations in that the the choice of summary statistics significantly impact the results. Finding adequate summary statistics may not be evident for every DGP and this approach could be inadequate under more complex DGPs. The penultimate row of Table 1 shows that the TCN approach proposed in this paper outperforms all the other benchmarks and achieves performance on par (7.41% worse) with that of the fully asymptotically efficient MLE, displayed in the final row. As shown in Section 4, the TCN method can outperform MLE with additional training for this sample size.

Method	**Source**	$E_{\%}$
FNN	B. Jiang et al. (2017)	0.158
PEN	Wiqvist et al. (2019)	0.149
CNN	Akesson et al. (2021)	0.165
FNN, with statistics	Creel (2017)	0.128
TCN	This paper	**0.116**
MLE	This paper	**0.108**

Table 1: $E_{\%}$ for MA(2) model (5), $T = 100$, and 10^6 training samples, averaged over $n = 10$ replications.

3 Neural Network Architectures

This section describes the neural network topologies we utilize to construct moment conditions. To provide clarity for the reader, we focus on the rationale behind our choice of networks and their specific implementations. Appendix D supplements the necessary formal definitions of different layer and network types.

factor is derived.

For each data-generating process under scrutiny, we implement a long short-term memory network (LSTM; D.7) and a TCN (D.11). In their seminal paper, Hochreiter and Schmidhuber (1997) propose the LSTM as an extension of the vanilla RNN architecture (D.5), and, since their introduction, LSTMs have become a hallmark of sequence modeling. LSTMs have been particularly influential in time series problems due to their ability to capture long-term dependencies in data, making them a prevalent choice for various applications, including natural language processing and financial forecasting (e.g., Bucci, 2020; Chung et al., 2014; Sutskever, Vinyals, & Le, 2014; Vinyals et al., 2015).

While TCNs have garnered less attention than LSTMs in the literature, they are also considerably more recent, with the first concepts introduced by van den Oord et al. (2016), in their so-called WaveNet architecture. Bai, Kolter, and Koltun (2018) find that TCNs outperform other deep learning architectures in several sequence modeling tasks. TCNs address longstanding issues of LSTMs, such as allowing for parallelized training, which speeds up training time, and introducing an adjustable lookback window, which allows for greater flexibility in modeling different types of sequences. Furthermore, they typically require significantly fewer parameters to achieve on-par performance with LSTMs. Accordingly, this work primarily focuses on TCNs, and we include LSTM architectures as a deep learning baseline.

More recently, attention-based architectures, particularly transformers (Vaswani et al., 2017), have started to take over LSTMs as state-of-the-art architecture for sequence modeling. This success, however, has come with the price of added complexity. The number of parameters in potent transformer architectures is typically an order of magnitude larger than in their LSTM and TCN counterparts. Due to this, transformers require significantly more computational power to be trained adequately, which often proves to be a barrier for consumer-grade workstations and also justifies their exclusion in this work.

Before discussing the specific implementations of LSTMs and TCNs, we briefly mention the process of hyperparameter tuning[7] as well as some architecture choices that influence both networks. First, we do not use L_2 weight regularization or dropout layers when training the networks. While it is commonly well-understood that such techniques help neural networks

[7] Hyperparameter tuning refers to the process of selecting the optimal parameters for a machine learning model. These parameters, known as hyperparameters, are not learned from the data but are set prior to the training process and can significantly impact the performance of the model. See Appendix A for the hyperparameters of interest.

generalize and be more effective out of sample (Srivastava et al., 2014), our setting exemplifies a special case. Since the data is simulated anew for each epoch, there is no overlap among individual training steps, making it difficult for the network to overfit. Accordingly, we avoid regularization methods, to speed up computations. Second, we conducted a grid-search hyperparameter tuning approach to determine the optimal values for the mini-batch size, the number of layers, the number of nodes or channels per layer, and optimization algorithms for each network. Additionally, for the TCN, we conducted a grid-search hyperparameter tuning approach to determine whether to include a residual connection and to obtain the optimal values for the kernel width and dilation factor. Table 8 displays the candidate and chosen hyperparameter values for both networks. Due to the number of candidates, we use early stopping to simplify the grid-search process and reduce computation time. Notably, many candidate values produce similarly satisfactory results, indicating that this specific choice of hyperparameters is not crucial for the success of our approach.

Once the final set of hyperparameters is chosen, we run the TCN for $0.4 \cdot 10^6$ epochs and the LSTM for $0.2 \cdot 10^6$ epochs. We found that training for longer is always beneficial during our tuning process.[8] However, it is essential to note that these benefits are marginal past an initial number of epochs; this, coupled with the fact that recurrent models are not parallelizable and thus train considerably slower, justifies the lower number of epochs for LSTMs. Finally, we evaluate the networks every 5 000 epochs for the TCN and 2 500 epochs for the LSTM, and we keep track of the best model on a validation set with 10 000 observations. During the last 1 000 epochs of training, we repeat this evaluation procedure at every epoch instead.

The output of the networks, $\hat{\boldsymbol{\theta}}$ has the same dimension as the parameter vector $\boldsymbol{\theta}$, and is multi-dimensional in all our DGPs. This multi-dimensionality of the output must be considered carefully when training the networks. Notably, the the components of the parameter vector can have different ranges and averaging a component-wise metric might lead to a loss function that is dominated by only a few components. This induces a challenge in training the network, as components with broader ranges might have a more significant impact on the loss, leading the networks to focus more on these components when minimizing the loss. We train our network using the ℓ_2-norm, which is equivalent to averaging the squared errors of each component. To address the issue of different ranges, we normalize the parameter vector

[8] Note that because the data is simulated anew for each epoch, the training set is not exhausted after a few first epochs. Effectively, this procedure is equivalent to training on a single dataset of $1\,024 \cdot 0.4 \cdot 10^6 = 409.6 \cdot 10^6$ observations.

before training the network. We randomly draw $m = 0.1 \cdot 10^6$ parameter vectors and compute the sample mean and component-wise standard deviation of these parameter vectors.[9]

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m}\boldsymbol{\theta}_i, \quad \boldsymbol{\sigma} = \frac{1}{\sqrt{m}}\left(\sum_{i=1}^{m}(\boldsymbol{\theta}_i - \boldsymbol{\mu})^{\circ 2}\right)^{\circ\frac{1}{2}},$$

with $(\cdot)^{\circ 2}$ and $(\cdot)^{\circ\frac{1}{2}}$ denoting the Hadamard square and square root, respectively.

Once these two quantities are computed, we can defined a component-wise normalizing function $z(\boldsymbol{\theta}) = (\boldsymbol{\theta} - \boldsymbol{\mu}) \oslash \boldsymbol{\sigma}$, where $\oslash$ represents the Hadamard division.

$$L(\tilde{\boldsymbol{\theta}}, \boldsymbol{\theta}) = \left\| z(\boldsymbol{\theta}) - \tilde{\boldsymbol{\theta}} \right\|_2,$$

where $\tilde{\boldsymbol{\theta}}$ is the raw output of the network and $\boldsymbol{\theta}$ is the true parameter vector that generated the underlying sample. Once the network is trained, we can pass its output through the inverse of the normalizing function to obtain the final estimate $\hat{\boldsymbol{\theta}} = z^{-1}(\tilde{\boldsymbol{\theta}})$. To simplify, we refer to $\hat{\boldsymbol{\theta}}$ as the output of the network in the following sections since the unnormalized output is uniquely used in training to mitigate the impact of the parameters' scale. This quantity is otherwise not of interest in this work and we report all results in terms of $\hat{\boldsymbol{\theta}}$.

3.1 LSTM

Our LSTM implementation is straightforward and depicted in Figure 1. We pass the input series through an initial dense layer with a hyperbolic tangent activation function, followed by two LSTM layers with a hidden size of 32. Lastly, the LSTM outputs are mapped back to a vector of reals using a dense layer with an identity activation function. Except for the input and output sizes of the first and last layers, the network's topology is unchanged between the different experiments. On the one hand, increasing the number of layers or neurons per layer does not directly increase the memory of the LSTM, i.e., it does not enable us to capture longer sequences. On the other hand, the inputs and outputs are relatively low-dimensional for all DGPs, implying that a moderately-sized network, like the proposed one, ought to be

[9] Importantly, these parameter draws do not require us to simulate any data. Hence, this step is computationally cheap.

sufficient to model the underlying relationship.

Figure 1 portrays the LSTM architecture implemented in our paper. The blue rectangles represent the different layers of the LSTM, where the numbers in parentheses stand for the size of the input and output nodes in each layer.

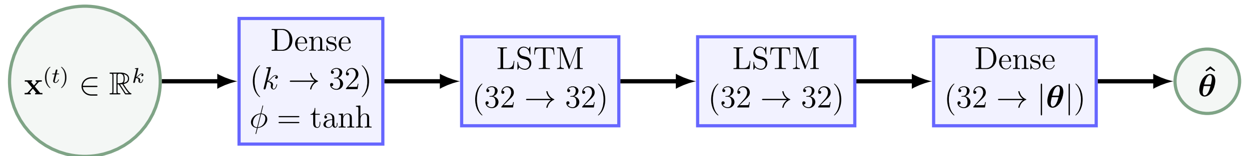

Figure 1: LSTM Architecture

3.2 TCN

The TCN architectures we implement follow a similar structure; however, they slightly differ depending on the observed sequence length. These differences lie in the number of layers chosen to achieve a particular receptive field size. As our hyperparameter tuning suggests, we fix the kernel width to 32 and the dilation factor to 2. According to (8), we then choose the smallest possible number of temporal blocks (see Figure 3) for the receptive field size to be at least as large as the entire sequence. Proceeding according to this formula gives 3, 4, 5, and 6 temporal blocks layers for the sequence lengths 100, 200, 400, and 800.

Figure 2 depicts the TCN architecture implemented in our paper. The blue rectangles represent the different layers of the TCN and the red rectangle is the matrix vectorization operation. The temporal blocks repeat 3 to 6 times depending on the sample size of the DGP. The outputs of the temporal blocks are vectorized, passed through a convolutional layer (D.8), and passed through two dense layers (D.2), where the first one is followed by a hard hyperbolic tangent activation function[10].

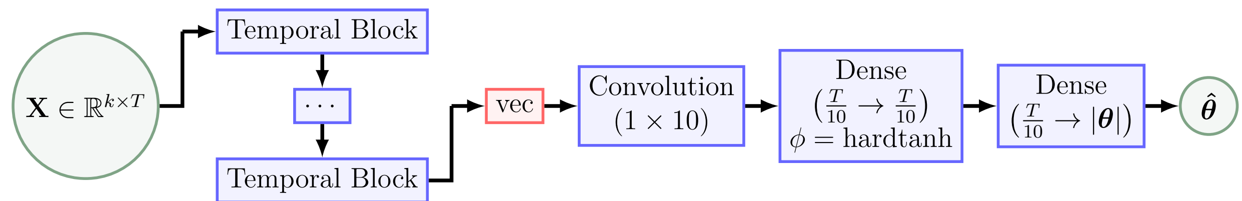

Figure 2: TCN Architecture

[10] $\text{hardtanh}(x) = \min(\max(-1, x), 1)$

Figure 3 illustrates the structure of temporal blocks in the TCN. We use a skip connection (He et al., 2016) with a 1×1 convolutional layer (D.8) and pass the inputs through two iterations of dilated causal convolution layers (D.10) and batch normalization layers (Ioffe & Szegedy, 2015) which are followed by a leaky ReLU[11] activation function. The same leaky ReLU is applied after the skip connection. This architecture is the same as Bai, Kolter, and Koltun (2018), with the minor difference that we use batch normalization for simplicity of implementation instead of weight normalization (Salimans & Kingma, 2016).

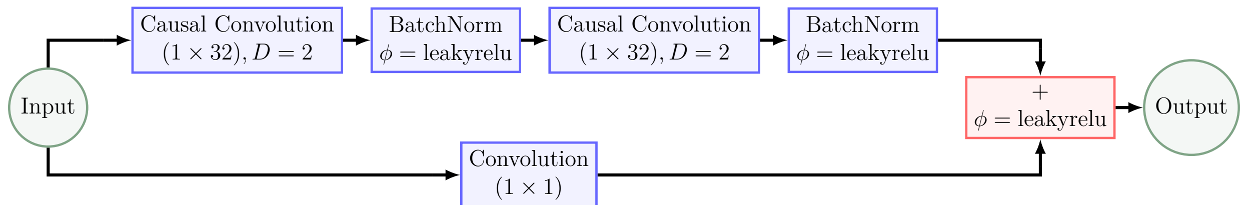

Figure 3: Temporal Block

4 Results for Models with Tractable Likelihoods

This section explores three data-generating processes for which the maximum likelihood estimator exists and can be computed: an MA(2), a Logit, and a GARCH(1,1). The goal is to see how well neural networks can do compared to an asymptotically unbiased and fully efficient baseline. The neural network estimators we propose can be computed with equal ease, whether or not the MLE can be computed. We conjecture that if neural network estimators can achieve performance equal to or better than MLE when a comparison is possible, they should also perform well in cases where the MLE is difficult compute or intractable.

Table 2 reports the results of our empirical experiment, averaged across the elements of the parameter vector. Given a true parameter $\boldsymbol{\theta}$ and a point estimate $\hat{\boldsymbol{\theta}}$, we gauge the estimator's effectiveness based on three metrics which we compute on the sample: the absolute bias, the

[11] $\text{leakyrelu}(x) = \max(0.01x, x)$

root-mean-square error (RMSE), and the normalized mean absolute error (NMAE)

$$\left|\text{bias}(\hat{\boldsymbol{\theta}})\right| = \left\|\mathbb{E}\left[\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\right]\right\|_1 \approx \left\|\frac{1}{n}\sum_{i=1}^{n}\hat{\boldsymbol{\theta}}_i - \boldsymbol{\theta}_i\right\|_1, \tag{2}$$

$$\text{RMSE}(\hat{\boldsymbol{\theta}}) = \sqrt{\mathbb{E}\left[\left\|\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\right\|_2^2\right]} \approx \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left\|\hat{\boldsymbol{\theta}}_i - \boldsymbol{\theta}_i\right\|_2^2}, \tag{3}$$

$$\text{NMAE}(\hat{\boldsymbol{\theta}}) = \gamma(\boldsymbol{\theta}) \cdot \mathbb{E}\left[\left\|\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\right\|_1\right] \approx \gamma(\boldsymbol{\theta}) \cdot \frac{1}{n}\sum_{i=1}^{n}\left\|\hat{\boldsymbol{\theta}}_i - \boldsymbol{\theta}_i\right\|_1, \tag{4}$$

where the normalizing factor $\gamma(\boldsymbol{\theta})$ is the component-wise inverse expected mean absolute error when the prior mean is used as an estimate. Given the sampling distribution of a parameter vector $\boldsymbol{\theta}$ with entries θ_i, the normalizing factor $\gamma(\boldsymbol{\theta})$ simplifies to a constant vector.[12] When $\theta_i \sim \text{Unif}(a_i, b_i)$ such as in the GARCH(1,1) and MA(2) DGPs, the factor simplifies to $\gamma(\boldsymbol{\theta})$, a vector with entries $\gamma(\theta_i) = \frac{4}{b_i - a_i}$, and when $\theta_i \sim \mathcal{N}(0, 1)$ such as in the Logit DGP, we obtain $\gamma(\theta_i) = \sqrt{\frac{\pi}{2}}$. Normalizing the MAE by this factor builds on previous works by Akesson et al. (2021) and allows for a straightforward interpretation: an NMAE equal to one implies no new information gained or lost, while a value below one suggests an improvement in accuracy compared to prior knowledge.

For each of the three DGPs we present below, we compare the maximum likelihood estimate of the parameters to that of the TCNs and LSTMs detailed in Section 3. We do so using four sample sizes for each process: $T \in \{100, 200, 400, 800\}$, and each approach is evaluated on the same $n = 5\,000$ test samples. As MLE is a $\sqrt{n}$-consistent estimator, by doubling the sample size, we expect the metrics to decrease by a factor $\frac{1}{\sqrt{2}}$, or approximately 30%.

4.1 MA(2)

One of the examples that has been widely used in the literature on parameter estimation using neural networks (Akesson et al., 2021; Creel, 2017; B. Jiang et al., 2017; Wiqvist et al., 2019) is a moving average model of order 2 (MA(2)). We use this same model, to facilitate

[12] See Appendix C for the detailed derivation of this factor on a single component of the parameter vector.

comparing our methods with previous results. Data is generated by the model

$$y_t = u_t + \theta_1 u_{t-1} + \theta_2 u_{t-2}, \quad u_t \sim \mathcal{N}(0,1)$$
$$\theta_1 \in [-2,2], \quad \theta_2 \in [-1,1], \quad \theta_2 \pm \theta_1 \geq -1. \tag{5}$$

The parameter restrictions define the invertible region, and the prior of $\boldsymbol{\theta} = (\theta_1, \theta_2)^\top$ is a uniform distribution over this region.

The results for this DGP are illustrated in the first third of Table 2. We observe that the metrics for the MLE decrease with sample size, as expected from $\sqrt{n}$-consistent estimators. While the LSTM achieves the lowest absolute bias for $T = 200$, it is outperformed by MLE or TCN everywhere else. In particular, the TCN displays an edge across all metrics over MLE for the smallest sample size. However, as the sample size grows, MLE achieves better results, with the TCN being relatively close.

4.2 Logit

In order to have an example typical of cross-sectional models, we consider the simple logit model. The logit model generates data according to

$$y_t = \mathbb{1}\left(\varepsilon_t < \frac{1}{1+\exp(-\boldsymbol{x}_t^\top \boldsymbol{\theta})}\right), \quad \varepsilon_t \sim \mathcal{N}(0,1)$$

where $\mathbb{1}(\cdot)$ is the indicator function. Our DGP specifies a regressor vector $\boldsymbol{x}_t \in \mathbb{R}^3$ that comprises three independent standard normal draws. The prior for the parameter $\boldsymbol{\theta}$ is also a vector of three independent standard normal draws.

As reported in the middle third of Table 2, we obtain a similar pattern for the Logit model as we did for the MA(2) model. TCN and MLE generally outperform the LSTM, with the TCN achieving the best RMSE and NMAE for all but the largest sample size. Even in our largest sample, $T = 800$, the results of the TCN are particularly close to that of MLE. However, considering the absolute bias, we find MLE to perform significantly better except for the smallest sample size, $T = 100$.

4.3 GARCH(1,1)

The standard GARCH(1,1) model (Bollerslev, 1986; Engle, 1982) is given by

$$y_t = \sqrt{h_t}\varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0,1)$$
$$h_t = \omega + \alpha\varepsilon_{t-1}^2 + \beta h_{t-1}$$

For computational simplicity, we parameterize the model as $\boldsymbol{\theta} = (v, \phi, \pi)^\top$, where $v = \omega/(1-(\alpha+\beta))$ is the long-run variance, $\phi = \alpha + \beta$ is the sum of the two underlying parameters, and $\pi = \frac{\beta}{\alpha+\beta}$ is the share of β. This reparametrization allows us to define an independent uniform prior distribution for each parameter while respecting the stationarity constraints and being able to recover the original underlying parameters using $\omega = (1-\phi)v$, $\alpha = (1-\pi)\phi$, and $\beta = \pi\phi$. The prior on θ is $v \sim \text{Unif}(0.001, 1)$, $\phi \sim \text{Unif}(0, 0.99)$, and $\pi \sim \text{Unif}(0, 1)$. The prior for v ensures that the long-run variance is reasonable for financial series, and the priors on ϕ and π ensure that the model is stationary.

In contrast to the other two DPGs, we find that, while the metrics for MLE decline as the sample size grows, they do so slower than expected for a $\sqrt{n}$-consistent estimator. This discrepancy is explained by the fact that the MLE is often on the bounds of the parameter space, which are imposed in estimation, which occurs especially frequently for the smaller sample sizes. Because the restrictions bind more often for smaller sample sizes, RMSE and NMAE in those cases are smaller than what they would be if we used an unrestricted ML estimator. We impose the true restrictions that reduce RMSE and NMAE for the ML estimator to make the comparison with the neural network estimators fair since they also use the restrictions implied by the prior's support. The TCN comfortably outperforms other estimators for all metrics and sample sizes, achieving a bias an order of magnitude smaller and roughly 30% less RMSE and NMAE than the MLE across the board. We also note that while the LSTM performs significantly worse than the TCN, it surpasses MLE everywhere.

MA(2)

	Absolute bias				Root-mean-square error				Normalized mean absolute error			
	$T = 100$	$T = 200$	$T = 400$	$T = 800$	$T = 100$	$T = 200$	$T = 400$	$T = 800$	$T = 100$	$T = 200$	$T = 400$	$T = 800$
MLE	0.01155	0.00516	0.00219	0.00091	0.10296	0.06410	0.04235	0.02998	0.12116	0.07569	0.04968	0.03487
TCN	0.00237	0.00415	0.00133	0.00102	0.09026	0.06405	0.04425	0.03225	0.10680	0.07637	0.05254	0.03809
LSTM	0.00356	0.00037	0.00369	0.00406	0.14912	0.10405	0.07037	0.04948	0.17787	0.12458	0.08431	0.05911

Logit

	Absolute bias				Root-mean-square error				Normalized mean absolute error			
	$T = 100$	$T = 200$	$T = 400$	$T = 800$	$T = 100$	$T = 200$	$T = 400$	$T = 800$	$T = 100$	$T = 200$	$T = 400$	$T = 800$
MLE	0.00492	0.00316	0.00336	0.00038	0.38582	0.24914	0.17264	0.11915	0.36150	0.24023	0.16803	0.11617
TCN	0.00342	0.01396	0.00620	0.00660	0.31828	0.23259	0.16604	0.11985	0.31215	0.22826	0.16273	0.11667
LSTM	0.00797	0.01224	0.00309	0.00781	0.43817	0.33195	0.24132	0.17963	0.42837	0.32438	0.23480	0.17412

GARCH(1, 1)

	Absolute bias				Root-mean-square error				Normalized mean absolute error			
	$T = 100$	$T = 200$	$T = 400$	$T = 800$	$T = 100$	$T = 200$	$T = 400$	$T = 800$	$T = 100$	$T = 200$	$T = 400$	$T = 800$
MLE	0.03405	0.02024	0.02401	0.02467	0.29091	0.26311	0.23400	0.21029	0.84547	0.72790	0.61103	0.51515
TCN	0.00255	0.00194	0.00242	0.00437	0.19757	0.17357	0.15536	0.13636	0.62494	0.53183	0.45566	0.38712
LSTM	0.01330	0.01134	0.01228	0.01529	0.21653	0.19502	0.17587	0.15649	0.70179	0.61698	0.53949	0.45756

Table 2: Absolute bias, RMSE, and NMAE

Each metric is computed element-wise. The table reports the average over the elements of the parameter vector θ, and the lowest values for each sample size and metric are highlighted in gray.

5 Empirical Example: A Jump-Diffusion Model for S&P 500 Data

This section presents an example of a computationally challenging model that better reflects real-world research problems: estimating a jump-diffusion model of S&P 500 returns. This example aims to demonstrate the feasibility of the methods for moderately complex models. We train a TCN to recognize the parameters that generated a sample from a jump diffusion model. We then use the output of the neural network to define moment conditions for Bayesian Limited Information estimation (Kwan (1999), Kim (2002)), implemented using Markov chain Monte Carlo sampling, as proposed by Chernozhukov and Hong (2003). We refer to this procedure as Bayesian MSM estimation. The jump-diffusion model we use is the same as was used in Creel (2021), who explored FNNs, based on summary statistics to define moment conditions for Bayesian MSM estimation. In contrast, our paper proposes methods that do not necessitate the specification of summary statistics, the TCN constructs the statistics for the Bayesian MSM procedure directly from the sample.

5.1 The Jump-Diffusion Model

The following equations describe the jump-diffusion model:

$$\begin{aligned} dp_t &= \mu\, dt + \sqrt{\exp h_t}\, dW_{1t} + J_t\, dN_t \\ dh_t &= \kappa(\alpha - h_t) + \sigma\, dW_{2t}. \end{aligned}$$

Here p_t is 100 times the logarithmic price, h_t is the logarithmic volatility, J_t is the jump size, and N_t is a Poisson process with jump intensity λ_0. The stochastic processes W_{1t} and W_{2t} are two standard Brownian motions with correlation ρ. Whenever a jump occurs, its size is determined by $J_t = a\lambda_1\sqrt{\exp h_t}$, where a is either $+1$ or -1, with an equal probability of 0.5. Hence, the jump size depends on the current standard deviation, and positive and negative jumps are equiprobable.

The model is solved on a continuous time 24-hour basis, encompassing both trading and non-trading days, using the SRIW1[13] solver. To account for potential measurement error,

[13]In particular, we use the SRIW1 strong order 1.5 solver from the DifferentialEquations.jl package for the Julia language (Rackaukas & Nie, 2017).

we introduce the concept of observed log price, denoted as p_t^*. It is defined as the true log price, p_t, with the addition of measurement error: $p_t^* = p_t + \nu_t$, where $\nu_t \sim N(0, \tau^2)$. We consider cases where τ can be either positive or zero, allowing for scenarios with or without measurement error. Observed log price, p_t^*, is sampled at 10-minute ticks during the trading hours of trading days, and these intra-day observations are used to compute daily realized volatility (RV) and daily bipower variation (BV). At the close of trading days, we record the observed log price, RV, and BV values. The model generates 1 000 daily observations for returns, RV, and BV. Returns are calculated as the daily first difference of the observed log price. Due to the logarithmic scaling of the price by 100, returns are expressed on a daily percentage basis. With these three variables (returns, RV, and BV), we aim to estimate the eight parameters: $\boldsymbol{\theta} = (\mu, \kappa, \alpha, \sigma, \rho, \lambda_0, \lambda_1, \tau)$.

5.2 Data

We use two overlapping samples from the SDPR S&P 500 ETF Trust (SPY), an exchange-traded fund that tracks the S&P 500 stock index. We obtain the data from the NYSE Trade and Quote (TAQ) database following the procedure outline in Barndorff-Nielsen et al. (2009) to aggregate the data into 10-minute intervals. The first sample spans from December 17, 2013, to December 5, 2017, and the second sample spans from January 12, 2016 to December 31, 2019. We choose these time windows such that each sample encompasses 1 000 trading days, which aligns with the sequence length generated by the model. Figure 4 visualizes the data, where we can observe volatility clusters and apparent jumps. The presence of spikes in RV and BV during periods of volatility, and the differences between both measures, highlight the usefulness of employing these intra-day volatility measures for identifying the jump parameters within the model. Table 3 presents descriptive statistics for the two samples.

	2013-12 to 2017-12						**2016-01 to 2019-12**					
	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.	Mean	Min.	1st Qu.	Median	3rd Qu.	Max.
Returns	0.038	−4.166	−0.279	0.045	0.438	3.825	0.052	−4.199	−0.243	0.059	0.453	4.798
RV	0.335	0.007	0.102	0.196	0.369	9.771	0.400	0.007	0.094	0.182	0.383	7.935
BV	0.326	0.006	0.092	0.185	0.359	9.476	0.392	0.006	0.087	0.171	0.377	8.834

Table 3: Sample average and quartiles of returns, realized volatility (RV) and bipower variation (BV) for SPY

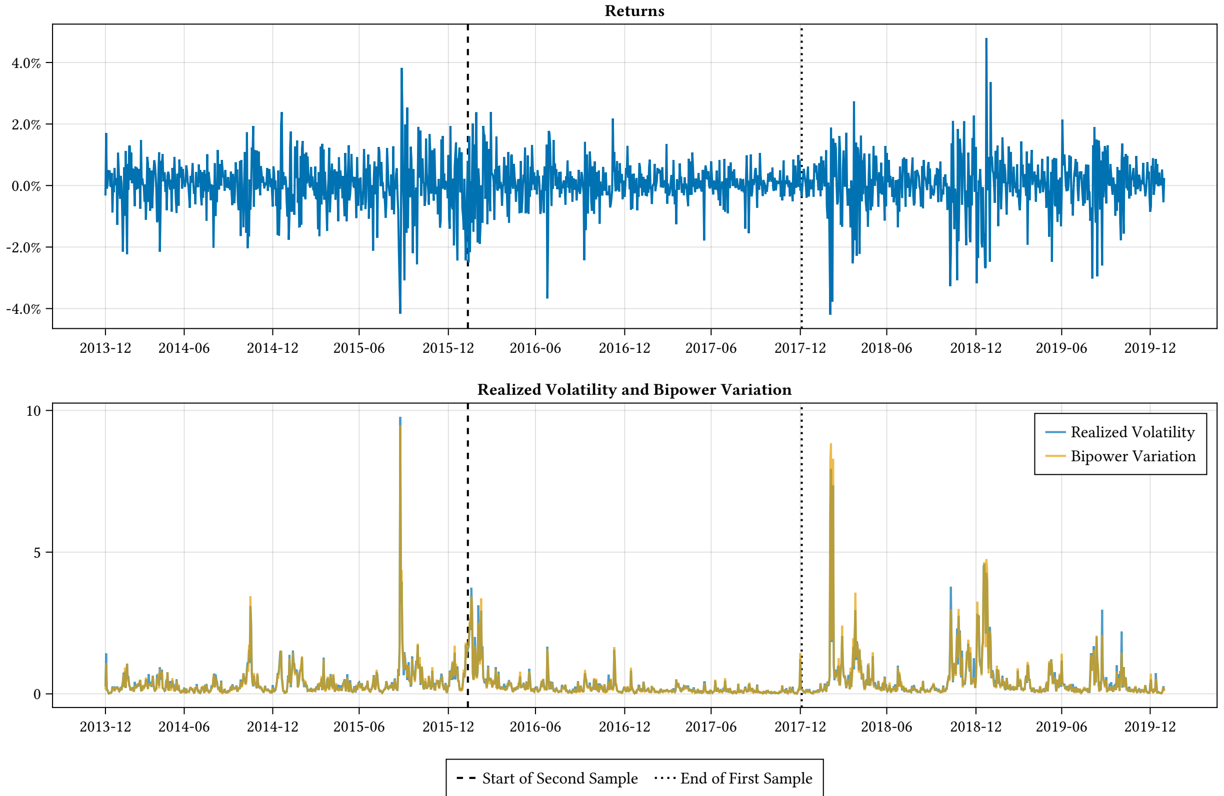

Figure 4: Returns, realized volatility, and bipower variation for SPY. (2013-12-17 to 2019-12-31)

5.3 Training the Network

Due to the high dimensionality of the parameter space, we use a Sobol' sequence (Sobol', 1967) to draw parameter combinations from the prior distribution and generate the training data. Table 4 depicts the bounds of the prior for each parameter. While these bounds are informed by the results of Creel (2021), adjustments have been made for some parameters. Notably, the lower bound for κ has been slightly increased to mitigate potential numerical issues arising from excessive persistence in the generated data. Additionally, the upper bounds for λ_0 and τ have been raised based on insights gained from the Markov chain Monte Carlo (MCMC) sampling process (discussed in detail in Section 5.4.2), as the original bounds were found to be overly restrictive. Except for λ_0 and τ, the remaining parameters are incorporated into the DGP as described in Section 5.1. However, for λ_0 and τ, if a negative value is drawn from the prior, the parameter is set to zero before generating data. This particularity enables DGPs without jumps and measurement errors and, effectively, the

actual priors for these two parameters possess atoms of probability at zero. This scheme is used simply for convenience in implementing the code. It is worth mentioning that in cases where λ_0 is negative, the parameter λ_1 becomes unidentifiable as no jumps occur.

Parameter	μ	κ	α	σ	ρ	λ_0	λ_1	τ
Lower bound	-0.05	0.01	-6.0	0.1	-0.99	-0.02	2.0	-0.02
Upper bound	0.05	0.30	0.0	4.0	0.50	0.10	6.0	0.20

Table 4: Bounds of uniform priors

The TCN architecture is the one described in Section 2 with 7 temporal blocks to ensure that the receptive field size exceeds the sequence length of 1 000. We draw roughly $102.4 \cdot 10^6$ parameter combinations using Sobol' sequences and the bounds in Table 4 to generate training data. Compared with the $406.9 \cdot 10^6$ draws used for the simpler DGPs as described in Section 4, we train the network on a significantly smaller number of observations. This is due to the increased complexity of the DGP, which requires more time to generate data. Since the parameters are drawn independently from each other, this approach results in some combinations that are not sensible from a financial perspective. To account for this, we discard the combinations that generate data with extreme values. In particular, we discard combinations that generate data which exhibits a maximum absolute 10-minute return greater than 100% or an average RV or BV greater than 50. Lastly, before feeding the data into the network, we logarithmize the volatility measures and standardize returns, log-RV, and log-BV to have zero mean and unit variance.

We then fit the TCN to the training data using the best hyperparameters found in Table 8. The mini-batch size of 1 024 implies that the network is fed around $0.1 \cdot 10^6$ mini-batches before our training data is exhausted. Once the data is exhausted, we proceed with a second round on the same data. We have found this repeated training to marginally improve the results without bearing the cost of generating additional data. We use a validation set of 15 000 samples to monitor the training process and keep track of the best model.

5.4 Results

5.4.1 TCN Results

While the direct TCN estimate of the parameters, $\hat{\boldsymbol{\theta}}_{\text{TCN}}$, is not our final estimate, it is still of interest to examine its performance. If the network is able to accurately estimate the parameters, it would provide a strong indication that the statistic is informative and that the indirect inference procedure is likely to yield good results.

	μ	κ	α	σ	ρ	λ_0	λ_1	τ
Abs. Bias	0.00049	0.00130	0.00010	0.00420	0.00242	0.00039	0.12571	0.00024
RMSE	0.00832	0.03658	0.24377	0.11405	0.07694	0.01533	0.80229	0.00128
NMAE	0.22309	0.36503	0.11722	0.08548	0.46267	0.61191	0.80395	0.01702

Table 5: Out-of-sample absolute bias, RMSE, and NMAE.

Aggregated performance measures for 5 000 out-of-sample predictions of the TCN.

Table 5 illustrates the out-of-sample performance of the TCN on 5 000 parameter combinations drawn randomly from the prior. Figure 5 displays the actual values of the parameters against the predicted values for each individual process as well as the estimated kernel density using a Gaussian kernel. The estimated parameters are close to the actual values, and the kernel density estimates are centered around the diagonal, indicating evidence that the TCN is able to accurately estimate the parameters of the jump-diffusion model. The Table and Figure also reveal that the TCN faces some difficulty in estimating the jump size parameter λ_1. We further investigate this issue by conditioning the λ_1 estimates on the true values of the jump intensity λ_0. For values of λ_0 smaller than 0.01, we would expect to not observe any jumps over the course of 1 000 observations, and the jump size parameter would thus remain unidentifiable[14]. Figure 6 depicts the estimated λ_1 values for different values of λ_0. As expected, the TCN is unable to properly estimate λ_1 when λ_0 is small. However, for larger values of λ_0, the TCN is able to estimate λ_1 with reasonable accuracy.

[14]Note however, that due to the innate randomness of the jump-diffusion model, there will be processes where a jump occurs even in the case where $\lambda_0 < 0.01$ and there will be cases where no jump occurs while $\lambda_0 \geq 0.01$.

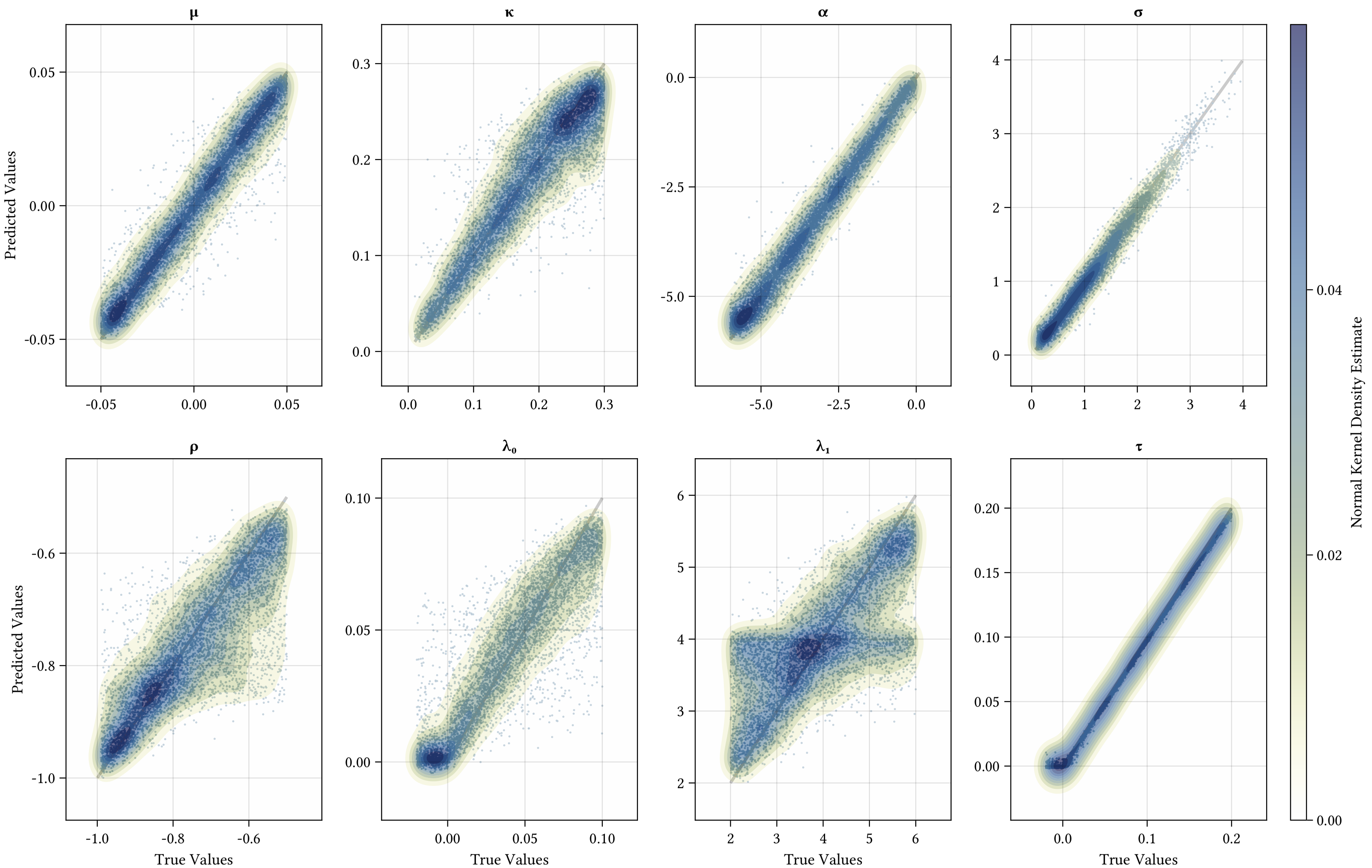

Figure 5: True values ($\boldsymbol{\theta}$) against predicted values ($\hat{\boldsymbol{\theta}}_{\text{TCN}}$) for the jump-diffusion model parameters. Out-of-sample data with 5 000 randomly drawn parameter combinations from the prior in Table 4.

5.4.2 The Bayesian MSM Procedure

The direct TCN estimate, $\hat{\boldsymbol{\theta}}_{\text{TCN}}$, is a statistic representing the output of a finite-dimensional function derived from the available data. Consequently, it does not fall under the category of extremum estimators, and its distribution is unknown, making the direct application of frequentist asymptotic theory impossible. In order to facilitate inference, we define moment conditions that are similar to those of indirect inference (Gourieroux, Monfort, & Renault, 1993)[15]. The moment conditions are defined as

$$m(\boldsymbol{\theta}) = \hat{\boldsymbol{\theta}}_{\text{TCN}} - \frac{1}{S}\sum_{s=1}^{S} \hat{\boldsymbol{\theta}}^{s}_{\text{TCN}}(\boldsymbol{\theta}), \tag{6}$$

where the $\hat{\boldsymbol{\theta}}^{s}_{\text{TCN}}(\boldsymbol{\theta})$ terms denote replications of the output of the trained TCN, evaluated using independent simulated samples, each generated under the DGP at the parameter $\boldsymbol{\theta}$. The sample statistic, $\hat{\boldsymbol{\theta}}_{\text{TCN}}$, obtained from the observed data, is assumed to have been generated by the DGP at the true parameter value, $\boldsymbol{\theta}_0$.

Kwan (1999) and Kim (2002) show how a limited information likelihood function may be constructed based upon moment conditions. These results require assumptions such that a central limit theorem will apply to the moment conditions, uniformly in a neighborhood of $\boldsymbol{\theta}_0$. We will proceed, without proof, as if these assumptions hold, for the moment conditions in (6). Some support for the validity of this speculation is in the results presented by Creel (2021), where confidence intervals based on similar methods exhibit accurate coverage. However, it is important to note that the asymptotic theory for estimators relying on deep neural networks to obtain parametric estimates is currently an active area of research (e.g., Farrell, Liang, & Misra, 2021).

Under the aforementioned assumptions, we can define the Bayesian limited information

[15] While the work by Gourieroux, Monfort, and Renault (1993) primarily focuses on extremum estimators derived from an auxiliary model, W. Jiang and Turnbull (2004) highlight how the statistic (in our case $\hat{\boldsymbol{\theta}}_{\text{TCN}}$) can also be an explicit function of the data. This broader perspective allows for a more flexible application of the indirect inference framework, accommodating scenarios where the statistic of interest is directly computed from the available data.

log-likelihood as:

$$L_n(\boldsymbol{\theta}) = -\frac{1}{2}n \cdot m(\boldsymbol{\theta})^\top \hat{\boldsymbol{\Sigma}}(\boldsymbol{\theta})^{-1} m(\boldsymbol{\theta}), \tag{7}$$

where $\hat{\boldsymbol{\Sigma}}(\boldsymbol{\theta})$ is a continuously updated estimator of $\boldsymbol{\Sigma}(\boldsymbol{\theta}) = \lim_{n\to\infty} \mathrm{Var}\left(\sqrt{n}m(\boldsymbol{\theta})\right)$. This criterion aligns with those examined by Chernozhukov and Hong (2003) (specifically, Section 4.1). Leveraging the methods proposed in that paper, we can now conduct inference on $\boldsymbol{\theta}_0$. The inference procedure employs ordinary MCMC techniques, wherein the acceptance or rejection step is based on treating (7) as the log-likelihood function.

To obtain an estimator of $\boldsymbol{\Sigma}(\boldsymbol{\theta})$, first consider the following expression:

$$\begin{aligned} \mathrm{Var}\left(\sqrt{n}m(\boldsymbol{\theta}_0)\right) &= \mathrm{Var}\left(\sqrt{n}\hat{\boldsymbol{\theta}}_{\mathrm{TCN}} - \frac{1}{S}\sum_{s=1}^{S}\sqrt{n}\hat{\boldsymbol{\theta}}^s_{\mathrm{TCN}}(\boldsymbol{\theta}_0)\right) \\ &= \left(1 + \frac{1}{S}\right)\mathrm{Var}\left(\sqrt{n}\hat{\boldsymbol{\theta}}_{\mathrm{TCN}}\right), \end{aligned}$$

where the simplification is due to the $\hat{\boldsymbol{\theta}}_{\mathrm{TCN}}$ and $\hat{\boldsymbol{\theta}}^s_{\mathrm{TCN}}$ variables being independent and identically distributed. To estimate $\mathrm{Var}\left(\sqrt{n}\hat{\boldsymbol{\theta}}_{\mathrm{TCN}}\right)$, we employ the sample covariance of a large number of independent and identically distributed draws of $\sqrt{n}\hat{\boldsymbol{\theta}}^s_{\mathrm{TCN}}(\boldsymbol{\theta})$, where $s = 1, \ldots, S$, and $\boldsymbol{\theta}$ is the value currently being considered for the unknown $\boldsymbol{\theta}_0$.

In summary, the steps involved in our Bayesian MSM procedure are as follows:

1. Obtain $\hat{\boldsymbol{\theta}}_{\mathrm{TCN}}$ from the neural network, and set the initial value of the MCMC chain, $\boldsymbol{\theta}^{(1)}$, to this value.

2. Define the symmetric random walk proposal distribution $\boldsymbol{\theta}^* \sim N(\boldsymbol{\theta}^{(i)}, t\boldsymbol{\Sigma}_p)$, where $\boldsymbol{\theta}^*$ represents the proposal, $\boldsymbol{\theta}^{(i)}$ is the current value in the Markov chain, and t is a scalar tuning parameter for the proposal density. $\boldsymbol{\Sigma}_p$, the proposal covariance, is the sample covariance of a large number of draws of $\sqrt{n}\hat{\boldsymbol{\theta}}^s_{\mathrm{TCN}}(\boldsymbol{\theta}^{(1)})$.

3. Draw a Markov chain using the Metropolis-Hastings acceptance/rejection algorithm, using the proposal distribution from step 2.

4. Check for convergence of the chain. If this is the case, stop. Otherwise, re-initialize the chain, setting $\boldsymbol{\theta}^{(1)}$ to the mean of the previous chain (dropping a burnin proportion from the initial part of the chain), adjust the tuning parameter t to improve the acceptance

rate, if needed, and go to step 3.

5.4.3 The Bayesian MSM Results

Table 6 presents the posterior means and medians of the final MCMC chain, along with the corresponding quantiles that define the 95% confidence intervals, following the procedure of Chernozhukov and Hong (2003). Figure 7 illustrates the posterior densities and highlights the 95% confidence intervals for the parameters of interest, for the two samples. We see how the drift parameter, μ, increases when moving from the first sample to the second one, which is consistent with the fact that returns are typically higher in the second sample (see Table 3). Likewise, the mean volatility, α, and the variance of the volatility, σ, are higher in the second sample, which also aligns with the higher intra-day average measures of volatility in the second sample. The jump frequency parameter, λ_0, is also higher in the second sample, which appears reasonable considering Figure 4. The measurement error, τ, is a significant factor in both sample periods.

	2013-12 to 2017-12				**2016-01 to 2019-12**			
	Mean	Median	2.5%	97.5%	Mean	Median	2.5%	97.5%
μ	−0.01454	−0.01445	−0.03639	0.00710	−0.00509	−0.00451	−0.02922	0.01780
κ	0.17403	0.17330	0.14237	0.21043	0.17166	0.17080	0.13522	0.21434
α	−1.19645	−1.19032	−1.52353	−0.91770	−1.10610	−1.09599	−1.43820	−0.80754
σ	0.92747	0.92637	0.82753	1.03575	0.95327	0.95118	0.85587	1.05564
ρ	−0.79534	−0.79799	−0.85202	−0.71998	−0.73734	−0.73939	−0.82190	−0.64171
λ_0	0.00563	0.00470	0.00081	0.01519	0.00798	0.00660	0.00129	0.02332
λ_1	3.25268	2.99758	2.03745	5.68069	2.89514	2.66586	2.02759	4.95701
τ	0.03038	0.03043	0.02820	0.03235	0.03243	0.03246	0.03071	0.03422

Table 6: MCMC results for the jump diffusion model on both SPY samples

The MCMC results further allow us to explore relationships between the parameters. Figure 8 depicts the contours of the joint posterior densities of μ and α, and that of σ and τ, for the two samples. We observe that larger values of the drift parameter, μ, are associated with lower values of mean volatility, σ. This suggests that the two parameters act as substitutes, to a certain extent, in the model, and that it is relatively difficult to isolate the effects of each parameter. This is not suprising, as, through the leverage effect, higher returns due to a positive exogenous shock are associated with a lower volatility, because the shocks to price and volatility are strongly negatively correlated, as indicated by the value of the parameter

ρ. Likewise, we see that measurement error, τ, acts as a partial complement to the variance of volatility, σ. While this relationship has a less obvious explanation, our goal here is to highlight how the MCMC methodology offers a convenient way to explore and identify the existence of such relationships.

5.4.4 Comparison with Standard Moments

Lastly, we conclude this section by comparing our TCN-based moments to more conventional moments, based on low-order statistics of the data. This comparison contrasts the Bayesian Method of Simulated Moments (MSM) estimator, with TCN-based moments against a set of moments utilizing a vector of empirical sample moments in place of the TCN-fitted parameters. The latter comprises 12 elements, encapsulating means, standard deviations, and first-order autocorrelations for each of the three model-generated variables, in addition to the three correlations across variable pairs.

We generate 100 samples from the jump-diffusion model, adopting the parameter values from the posterior means listed in the first column of Table 6 as the true parameter values. For each of the 100 replications, both MSM variants employ identical proposal densities —the multivariate normal random walk, as defined in Section 5.4.2. The proposal covariance, Σ_p, is derived from the sample covariance of TCN-fitted parameters across 1 000 independent samples, each based on the true parameter vector. This approach yields a favorable proposal density, fostering efficient mixing and high acceptance rates. Constructing such a proposal is straightforward with the TCN estimator but poses challenges for the standard MSM estimator due to the absence of a robust initial estimate for computing covariance.

We process MCMC chains with a length of 200 post-burn-in for each sample. The posterior mean serve as the point estimator in each chain. This approach results in average acceptance rates of 27% for the TCN variant and 25% for the standard version. Given our focus on posterior mean estimation rather than tail behaviors, and the observed satisfactory acceptance rates, we confine chain lengths to 200.

The ensuing analysis, summarized in Table 7, indicates that both estimators exhibit comparable bias. However, the RMSE is consistently lower for the TCN-based estimator, averaging 73% of that observed in the standard variant. This experiment thus underscores the efficiency of estimation when employing TCN-based moments compared to standard moments.

	Standard Moments		TCN Moments	
Parameter	Bias	RMSE	Bias	RMSE
μ	0.003	0.015	-0.002	0.011
κ	0.001	0.021	0.005	0.018
α	-0.072	0.212	0.023	0.142
σ	0.021	0.077	0.005	0.057
ρ	-0.008	0.047	0.006	0.032
λ_0	0.004	0.006	0.002	0.005
λ_1	0.325	0.800	0.065	0.636
τ	0.000	0.001	-0.000	0.001

Table 7: Comparison of standard and TCN moments for the jump diffusion model over 100 replications.

6 Conclusion

In this study, we introduce a novel approach to estimating model parameters within simulation-based econometric frameworks. Utilizing deep learning methods, our method circumvents the difficulties tied to the specification and selection of optimal moments commonly found in traditional methodologies. Our empirical results affirm that the TCN-constructed moment conditions outperform previously used network architectures and are competitive with maximum likelihood estimators, when the likelihood is tractable.

The marriage of deep learning and simulation-based methods enables efficient statistical inference, owing to the rich and varied datasets generated through simulation. Post-training, our method produces these moment conditions at a minimal computational cost, making it highly suitable for scenarios requiring frequent parameter recalibration, a scenario where maximum likelihood methods falter due to their computational expense.

While promising, our results come with caveats. A pressing issue is the theoretical validation of our approach, especially concerning the assumed applicability of the Central Limit Theorem (CLT) to our network-based moment conditions, an assumption, which – while it appears plausible – has yet to be proven.

To sum up, our study stands as a pioneering effort in the end-to-end application of deep learning techniques to simulation-based econometrics. We not only showcase the practical advantages of our method but also highlight the need for robust theoretical frameworks that

can substantiate the empirical strengths we have demonstrated.

References

Akesson, M., Singh, P., Wrede, F., & Hellander, A. (2021). Convolutional Neural Networks as Summary Statistics for Approximate Bayesian Computation. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *14*(8).

Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Barndorff-Nielsen, O. E., Hansen, P. R., Lunde, A., & Shephard, N. (2009). Realized kernels in practice: Trades and quotes. *Econometrics Journal*, *12*(3).

Blum, M. G., & François, O. (2010). Non-linear regression models for Approximate Bayesian Computation. *Statistics and Computing*, *20*(1), 63–73.

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, *31*(3), 307–327.

Bucci, A. (2020). Realized Volatility Forecasting with Neural Networks. *Journal of Financial Econometrics*, *18*(3), 502–531.

Calin, O. (2020). *Deep Learning Architectures: A Mathematical Approach*. Springer International Publishing.

Carrasco, M. (2012). A regularization approach to the many instruments problem. *Journal of Econometrics*, *170*(2), 383–398.

Cheng, X., & Liao, Z. (2015). Select the valid and relevant moments: An information-based LASSO for GMM with many moments. *Journal of Econometrics*, *186*(2), 443–464.

Chernozhukov, V., & Hong, H. (2003). An MCMC approach to classical estimation. *Journal of Econometrics*, *115*(2), 293–346.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*, 1–9.

Creel, M. (2017). Neural nets for indirect inference. *Econometrics and Statistics*, *2*, 36–49.

Michael Creel acknowledges financial support from the Spanish Ministry of Science, Innovation and Universities and FEDER through grant PGC2018-094364-B-100 and from the Spanish Agencia Estatal de Investigación (AEI), through the Severo Ochoa Programme for Centres of Excellence in R&D (Barcelona School of Economics CEX2019-000915-S).

Creel, M. (2021). Inference using simulated neural moments. *Econometrics*, *9*(4), 1–15.

DiTraglia, F. J. (2016). Using invalid instruments on purpose: Focused moment selection and averaging for GMM. *Journal of Econometrics*, *195*(2), 187–208.

Donald, S. G., Imbens, G. W., & Newey, W. K. (2009). Choosing instrumental variables in conditional moment restriction models. *Journal of Econometrics*, *152*(1), 28–36.

Engle, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*, *50*(4), 987–1007.

Farrell, M. H., Liang, T., & Misra, S. (2021). Deep Neural Networks for Estimation and Inference. *Econometrica*, *89*(1), 181–213.

Fisher, T., Luedtke, A., Carone, M., & Simon, N. (2020). *Deep Learning for Marginal Bayesian Posterior Inference with Recurrent Neural Networks.*

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* The MIT Press.

Gourieroux, C., Monfort, A., & Renault, E. (1993). Indirect Inference. *Journal of Applied Econometrics*, *8*.

Gouriéroux, C., & Monfort, A. (1997). *Simulation-based econometric methods.* Oxford University Press.

Graves, A., Mohamed, A. R., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. *IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings.*

Grazian, C., & Fan, Y. (2020). A review of approximate Bayesian computation methods via density estimation: Inference for simulator-models. *Wiley Interdisciplinary Reviews: Computational Statistics*, *12*(4), 1–16.

Hall, A. R. (2015). Econometricians Have Their Moments: GMM at 32. *Economic Record*, *91*(S1), 1–24.

Hansen, L. P. (1982). Large Sample Properties of Generalized Method of Moments Estimators. *Econometrica*, *50*(4), 1029–1054.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*, *1*, 448–456.

Jiang, B., Wu, T. Y., Zheng, C., & Wong, W. H. (2017). Learning summary statistic for approximate Bayesian computation via deep neural network. *Statistica Sinica*, *27*(4), 1595–1618.

Jiang, W., & Turnbull, B. (2004). The indirect method: Inference based on intermediate statistics-a synthesis and examples. *Statistical Science*, *19*(2), 239–263.

Kim, J. Y. (2002). Limited information likelihood and Bayesian analysis. *Journal of Econometrics*, *107*(1-2), 175–193.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, *25*.

Kwan, Y. K. (1999). Asymptotic Bayesian analysis based on a limited information estimator. *Journal of Econometrics*, *88*(1), 99–121.

McFadden, D. (1989). A Method of Simulated Moments for Estimation of Discrete Response Models Without Numerical Integration. *Econometrica*, *57*(5), 995–1026.

Pakes, A., & Pollard, D. (1989). Simulation and the Asymptotics of Optimization Estimators. *Econometrica*, *57*(5), 1027–1057.

Rackaukas, C., & Nie, Q. (2017). DifferentialEquations.jl - a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, *5*.

Salimans, T., & Kingma, D. P. (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *Advances in Neural Information Processing Systems*, 901–909.

Sisson, S. A., Fan, Y., & Beaumont, M. (Eds.). (2018). *Handbook of Approximate Bayesian Computation*. Taylor & Francis.

Smith, A. A. (1993). Estimating Nonlinear Time-Series Models Using Simulated Vector Autoregressions. *Journal of Applied Econometrics*, *8*, 63–84.

Sobol', I. M. (1967). On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, *7*(4), 86–112.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*, 1929–1958.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, *27*.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., & Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio. *arXiv preprint arXiv:1609.03499*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser,., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *2017-Decem*(Nips), 5999–6009.

Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and tell: A neural image caption generator. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 3156–3164.

Wiqvist, S., Mattei, P. A., Picchini, U., & Frellsen, J. (2019). Partially exchangeable networks and architectures for learning summary statistics in approximate Bayesian computation. *International Conference on Machine Learning*, 6798–6807.

A Hyperparameter Tuning

	TCN	LSTM	
Hyperparameter	**Chosen values**		**Candidate values**
Mini-batch size	1 024	1 024	32, 64, 128, 256, 512, 1 024, 2 048
Optimizer	AdamW	AdamW	Adam, AdamW, RMSProp, AdaGrad
Learning rate	10^{-3}	10^{-3}	$5 \cdot 10^{-3}, 10^{-3}, 5 \cdot 10^{-4}, 10^{-4}$
TCN-specific			
Channels	32		2, 4, 8, 16, 32, 64
Dilation factor	2		2, 4, 8, 16
Kernel width	32		2, 4, 8, 16, 32, 64
Residual connection	True		False, True
LSTM-specific			
LSTM Layers		2	1, 2, 3
Nodes		32	8, 16, 32, 64

Table 8: Candidate and chosen hyperparameters values

B Plots

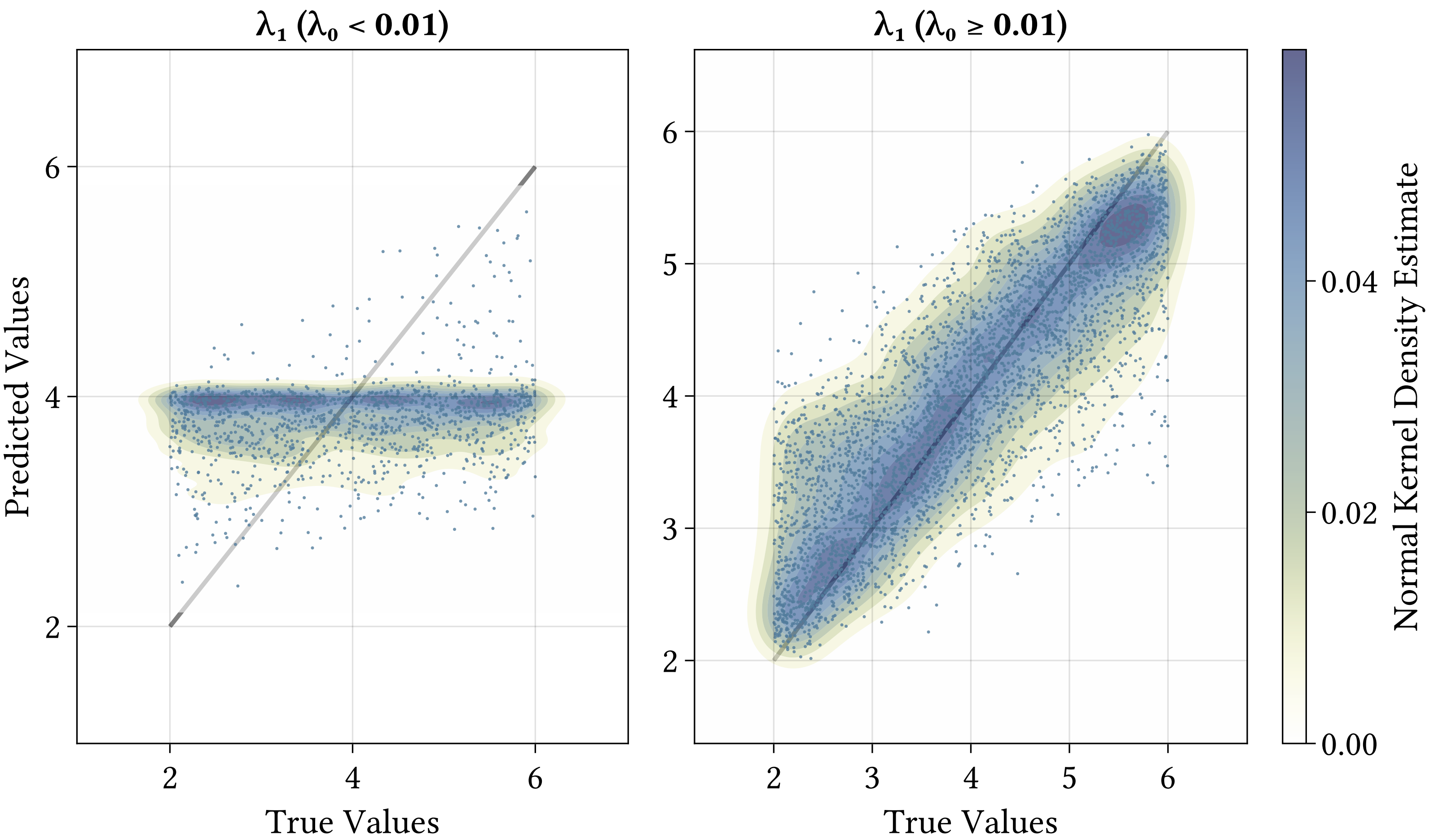

Figure 6: True values against predicted values for the jump size parameter λ_1 conditional on the jump intensity λ_0.

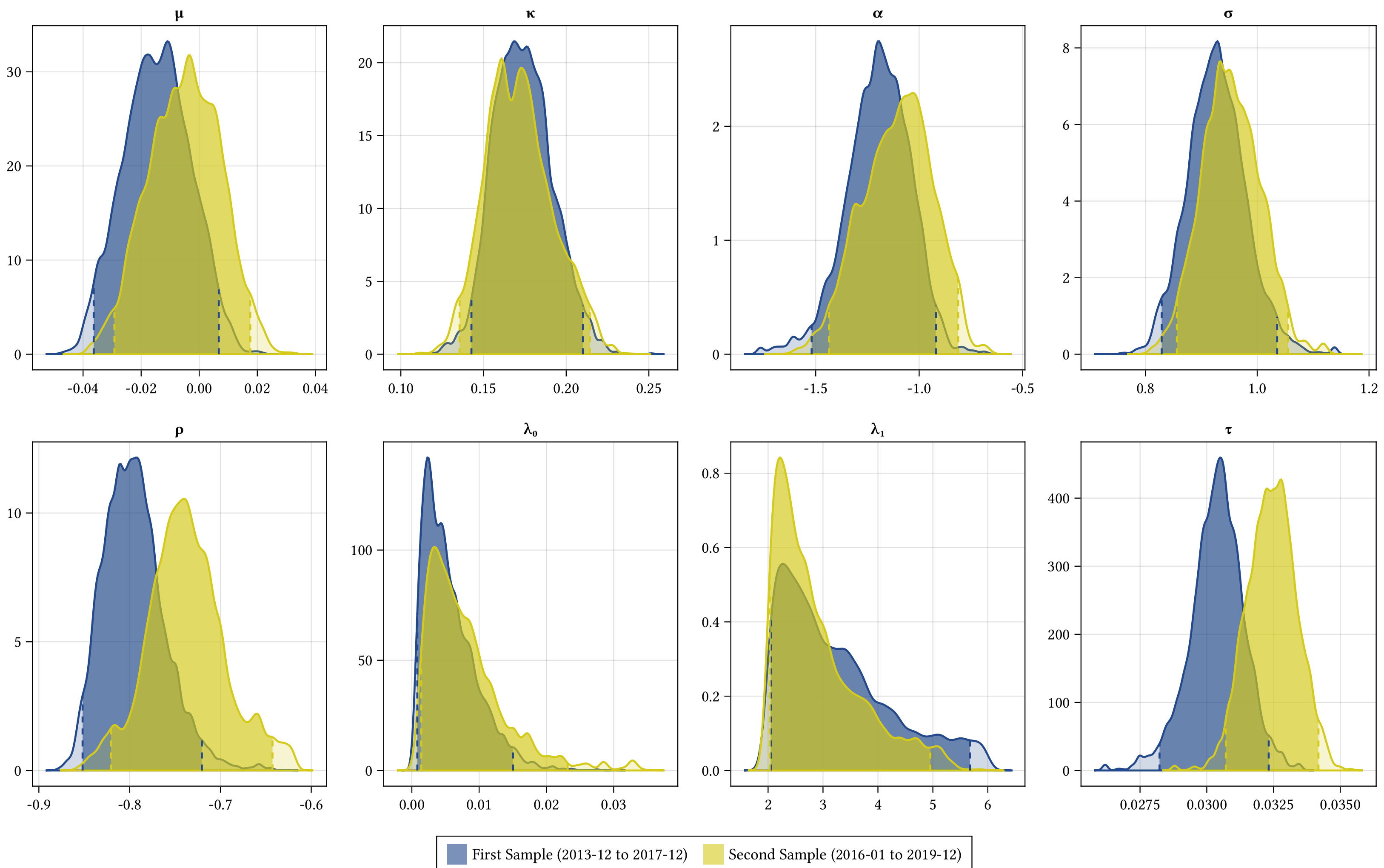

Figure 7: Posterior densities with highlighted 95% confidence intervals for the parameter of the jump-diffusion model for both SPY data samples.

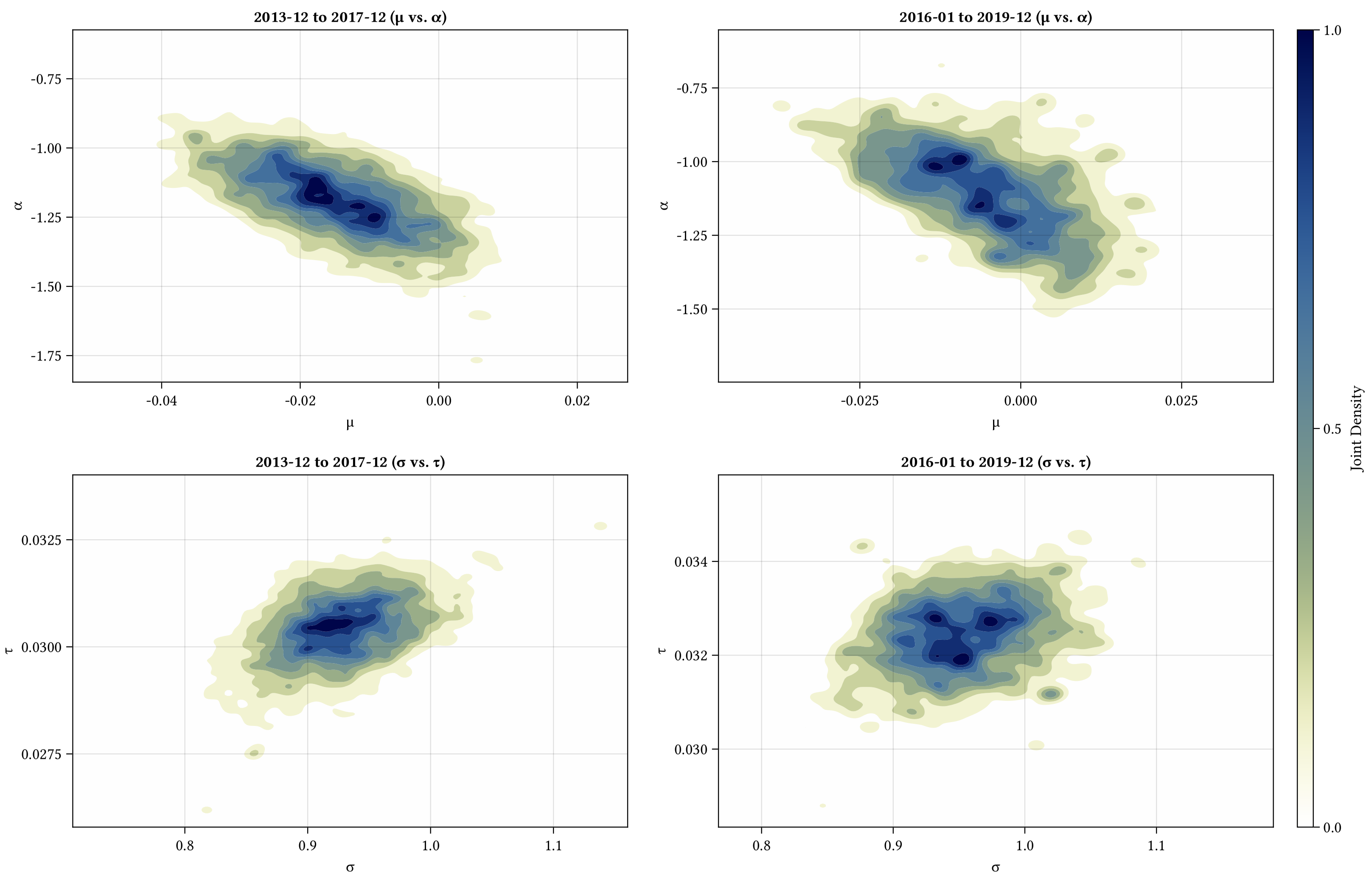

Figure 8: Selected bivariate posterior densities for the jump-diffusion model for both SPY data samples.

C Normalized Mean Absolute Error Derivation

This Appendix derives the normalizing factor in the normalized mean absolute error formula (4).

In our experiments, the parameters are randomly drawn from a pre-specified distribution. In the case of the GARCH(1,1) and MA(2), the parameters we use follow a uniform distribution, while they follow a standard normal distribution for the Logit DGP. The normalizing factor in (4) is the expected mean absolute error one makes when using the prior mean of the parameter as an estimate for the true parameter value.

C.1 Uniform distribution

Let the parameter θ follow a uniform distribution on $[a, b]$. The expected mean absolute error of the prior mean $\hat{\theta} = \mathbb{E}[\theta] = \frac{a+b}{2}$ is given by

$$
\begin{aligned}
\mathbb{E}\left[\left|\hat{\theta} - \theta\right|\right] &= \int_a^b \left|\hat{\theta} - \theta\right| \cdot \frac{1}{b-a} \, d\theta \\
&= \int_a^{\hat{\theta}} \left(\hat{\theta} - \theta\right) \cdot \frac{1}{b-a} \, d\theta + \int_{\hat{\theta}}^b \left(\theta - \hat{\theta}\right) \cdot \frac{1}{b-a} \, d\theta \\
&= - \frac{\left(\hat{\theta} - \theta\right)^2}{2(b-a)} \Bigg|_a^{\hat{\theta}} + \frac{\left(\theta - \hat{\theta}\right)^2}{2(b-a)} \Bigg|_{\hat{\theta}}^b \\
&= \frac{\left(\frac{a+b}{2} - a\right)^2 + \left(b - \frac{a+b}{2}\right)^2}{2(b-a)} \\
&= \frac{\left(\frac{b-a}{2}\right)^2}{b-a} = \frac{b-a}{4},
\end{aligned}
$$

thus, when $\theta \sim \text{Unif}[a, b]$, we obtain the normalizing factor

$$
\gamma(\theta) = \frac{4}{b-a}.
$$

C.2 Standard normal distribution

Let the parameter θ follow a normal distribution with mean 0 and variance 1. The expected mean absolute error of the prior mean $\hat{\theta} = \mathbb{E}[\theta] = 0$ is given by

$$\begin{aligned}
\mathbb{E}\left[\left|\hat{\theta} - \theta\right|\right] &= \int_{-\infty}^{\infty} \left|\hat{\theta} - \theta\right| \cdot \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-\frac{\theta^2}{2}} \, d\theta \\
&= \frac{1}{\sqrt{2\pi}} \left(\int_{-\infty}^{0} -\theta \mathrm{e}^{-\frac{\theta^2}{2}} \, d\theta + \int_{0}^{\infty} \theta \mathrm{e}^{-\frac{\theta^2}{2}} \, d\theta \right) \\
&= \frac{2}{\sqrt{2\pi}} \int_{0}^{\infty} \theta \mathrm{e}^{-\frac{\theta^2}{2}} \, d\theta \\
&= \frac{2}{\sqrt{2\pi}} \mathrm{e}^{-\frac{\theta^2}{2}} \Big|_{0}^{\infty} = \sqrt{\frac{2}{\pi}},
\end{aligned}$$

thus, when $\theta \sim N(0,1)$, we obtain the normalizing factor

$$\gamma(\theta) = \sqrt{\frac{\pi}{2}}.$$

D Neural Network Definitions

This Appendix provides formal definitions for the neural network topologies used throughout the paper. The following contents build upon Calin (2020) and Goodfellow, Bengio, and Courville (2016), and the interested reader is referred to these works for a more in-depth discussion of the deep learning topologies that follow.

D.1 Feedforward Neural Networks

Feedforward neural networks (FNNs), often dubbed multilayer perceptrons (MLPs), are the most straightforward neural network family. While we do not use FNNs in this work, it is beneficial to provide a short definition as their building blocks are ubiquitous and lay the foundation for the more advanced topologies we present.

FNNs comprise $L \in \mathbb{N}$ layers that each combine affine transformations and so-called activation functions. Each layer can be viewed as a map $g : \mathbb{R}^{N_I} \to \mathbb{R}^{N_O}$, where $N_I, N_O \in \mathbb{N}$ are

the layer's input and output vectors dimensions' respectively.

Definition D.1 (Activation function)**.** An activation function is a map $\phi : \mathbb{R} \to \mathbb{R}$ that is differentiable almost everywhere.

Definition D.2 (Dense layer)**.** Let $N_I, N_O \in \mathbb{N}$ be the input and output vectors' dimensions, respectively. Let $\mathbf{W} \in \mathbb{R}^{N_O \times N_I}$ be the layer's weights, $\mathbf{b} \in \mathbb{R}^{N_O}$ be the layer's biases, and ϕ be an activation function. A dense layer is a map $g : \mathbb{R}^{N_I} \to \mathbb{R}^{N_O}$ given by

$$g(\mathbf{x}) = \phi\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right),$$

where ϕ is applied element-wise, and $\mathbf{x} \in \mathbb{R}^{N_I}$ is the input data.

Definition D.3 (FNN)**.** Let $L \in \mathbb{N}$ be the number of layers of an FNN, let g_ℓ be the ℓth layer of the FNN ($\ell = 1, \ldots, L$), and let $N_I^{(\ell)}, N_O^{(\ell)} \in \mathbb{N}$ be the dimensions of the input and output vectors for the ℓth layer. An FNN is a map $f : \mathbb{R}^{N_I^{(1)}} \to \mathbb{R}^{N_O^{(L)}}$ defined by

$$f(\mathbf{x}) = g_L \circ g_{L-1} \circ \cdots \circ g_2 \circ g_1(\mathbf{x}),$$

where $\circ$ denotes the composition operator for functions and $\mathbf{x} \in \mathbb{R}^{N_I^{(1)}}$ is the input data of the FNN.

Notice that the above definition allows each layer to have a different number of inputs and outputs as well as individual activation functions. A deep FNN creates a progressively more abstract reparametrization of the input data by passing it through a series of alternating affine transformations and nonlinear functions.

D.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks particularly well-suited for processing sequential data. Compared to FNN, RNNs can easily accommodate variable-length data sequences without modifying the network architecture. Consider that we now observe a sequence of input vectors $\{\mathbf{x}^{(t)} : t = 1, \ldots, T\}$. While it is theoretically possible to fit an FNN to this input, this network will struggle to capture the information that might be encoded in the time dimension of this sequence.

Conversely, RNNs process every input $\mathbf{x}^{(t)}$ in a sequential manner. One key component is their so-called hidden state, a vector of weights encapsulating past information and updating

itself as the network processes the sequence step by step. This hidden state allows the network to extract and keep track of sequential information from the data.

Definition D.4 (Recurrent layer)**.** Let $N_I, N_H, N_O \in \mathbb{N}$ be the dimensions of input, hidden, and output vectors. Let $\mathbf{W}_I \in \mathbb{R}^{N_H \times N_I}$, $\mathbf{U} \in \mathbb{R}^{N_H \times N_H}$, and $\mathbf{W}_O \in \mathbb{R}^{N_O \times N_H}$ be the weight matrices parametrizing the input-to-hidden, the hidden-to-hidden, and the hidden-to-output connections, respectively. Let $\mathbf{b}_H \in \mathbb{R}^{N_H}$ and $\mathbf{b}_O \in \mathbb{R}^{N_O}$ be the hidden and output biases of the layer, and ϕ_H, ϕ_O be activation functions. A recurrent layer is a recurrent map $g : \left(\mathbb{R}^{N_I}\right)^{\mathbb{Z}} \to \left(\mathbb{R}^{N_O}\right)^{\mathbb{Z}}$ given by

$$\begin{aligned} \mathbf{h}^{(t)} &= \phi_H\left(\mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{W}_I\mathbf{x}^{(t)} + \mathbf{b}_H\right) \\ g(\mathbf{x}^{(t)}) &= \phi_O\left(\mathbf{b}_O + \mathbf{W}_O\mathbf{h}^{(t)}\right), \end{aligned}$$

where ϕ_H and ϕ_O are applied element-wise.

As this definition illustrates, recurrent layers incorporate a feedback loop. Indeed, as the data is processed time step by time step, the hidden state $\mathbf{h}^{(t)}$ is updated using past information, and, in turn, the network's output depends on the hidden state's current value. Akin to the FNN, an RNN can be viewed as the composition of multiple layers with at least one recurrent layer.

Definition D.5 (RNN)**.** Let $L \in \mathbb{N}$ be the number of layers of an RNN, let g_ℓ be the ℓth layer of the RNN ($\ell = 1, \ldots, L$), and let $N_I^{(\ell)}, N_O^{(\ell)} \in \mathbb{N}$ be the dimensions of the input and output vectors for the ℓth layer. An RNN is a recurrent map $f : \left(\mathbb{R}^{N_I^{(1)}}\right)^{\mathbb{Z}} \to \left(\mathbb{R}^{N_O^{(L)}}\right)^{\mathbb{Z}}$ defined by

$$f(\mathbf{x}^{(t)}) = g_L \circ g_{L-1} \circ \cdots \circ g_2 \circ g_1(\mathbf{x}^{(t)}),$$

where at least one layer g_ℓ is an RNN layer.

Lastly, we raise two essential points. First, an RNN typically yields one output for each time step, i.e., it is a sequence-to-sequence model. Fortunately, the RNN can easily be accommodated to deal with sequence-to-one modeling, e.g., by running the network on the entire data set and keeping only the final output. Second, recurrent and dense layers are often combined to build an RNN, e.g., passing the final output through one or many dense layers. Hence, an RNN can be built using layers defined in D.2 and D.4.

Although RNNs significantly improve over typical FNNs regarding sequence modeling, they are not without downsides. On the one hand, RNNs cannot benefit from parallelization like

other families of networks due to their inherent sequentiality, i.e., it is necessary to compute the hidden state from the past time step before computing the next one. On the other hand, RNNs are typically plagued by the problem of vanishing or exploding gradients, which can make learning challenging, especially as the sequence length of the underlying data grows.

Gated RNNs are an extension of the vanilla RNN topology that aims to tackle the vanishing and exploding gradients by introducing gating units to help the network forget about less relevant past information. Long short-term memory networks, introduced by Hochreiter and Schmidhuber (1997), are a well-established type of gated RNN particularly adapted to deal with longer input sequences. LSTMs implement *forget*, *update*, and *output* gates. Furthermore, on top of the hidden state of recurrent layers, LSTMs also include a so-called *cell state*.

Definition D.6 (LSTM Layer)**.** Let $N_I, N_O \in \mathbb{N}$ be the number of input and output dimensions[16]. Let $\mathbf{U}_f$, $\mathbf{U}_i$, $\mathbf{U}_o$, and $\mathbf{U}_c \in \mathbb{R}^{N_O \times N_I}$ be the weight matrices parametrizing the connections from the hidden state to the forget, input, and output gates and to the cell state. Let $\mathbf{W}_f$, $\mathbf{W}_i$, $\mathbf{W}_o$, and $\mathbf{W}_c \in \mathbb{R}^{N_O \times N_I}$ be the analogs for the connections from the input data to the respective gates and cell state. Let $\mathbf{b}_f$, $\mathbf{b}_i$, $\mathbf{b}_o$, and $\mathbf{b}_c \in \mathbb{R}^N_O$ denote the biases for each gate and the cell state. $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic function, and $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is the hyperbolic tangent function. The equations that govern the different gates and the cell state of the LSTM layer are as follows:

$$
\begin{aligned}
\mathbf{f}^{(t)} &= \sigma\left(\mathbf{U}_f \mathbf{h}^{(t-1)} + \mathbf{W}_f \mathbf{x}^{(t)} + \mathbf{b}_f\right) \\
\mathbf{i}^{(t)} &= \sigma\left(\mathbf{U}_i \mathbf{h}^{(t-1)} + \mathbf{W}_i \mathbf{x}^{(t)} + \mathbf{b}_i\right) \\
\mathbf{o}^{(t)} &= \sigma\left(\mathbf{U}_o \mathbf{h}^{(t-1)} + \mathbf{W}_o \mathbf{x}^{(t)} + \mathbf{b}_o\right) \\
\tilde{\mathbf{c}}^{(t)} &= \tanh\left(\mathbf{U}_c \mathbf{h}^{(t-1)} + \mathbf{W}_c \mathbf{x}^{(t)} + \mathbf{b}_c\right) \\
\mathbf{c}^{(t)} &= \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)} \\
\mathbf{h}^{(t)} &= \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}_t)
\end{aligned}
$$

The final output of the LSTM layer at each time step t is its hidden state $\mathbf{h}^{(t)}$, i.e., an LSTM layer is a recurrent map $g : \left(\mathbb{R}^{N_I}\right)^{\mathbb{Z}} \to \left(\mathbb{R}^{N_O}\right)^{\mathbb{Z}}$ given by:

$$
g(\mathbf{x}^{(t)}) = \mathbf{h}^{(t)}.
$$

[16]The dimensions of the hidden state vector match that of the output.

The above definition shows that an LSTM layer can only output values in the range $(-1, 1)$. Due to this, LSTMs typically incorporate additional layers, e.g., a dense layer following the LSTM layer.

Definition D.7 (LSTM)**.** Let $L \in \mathbb{N}$ be the number of layers of an LSTM, let g_ℓ be the ℓth layer of the RNN ($\ell = 1, \ldots, L$), and let $N_I^{(\ell)}, N_O^{(\ell)} \in \mathbb{N}$ be the dimensions of the input and output vectors for the ℓth layer. An LSTM is a recurrent map $f : \left(\mathbb{R}^{N_I^{(1)}}\right)^{\mathbb{Z}} \to \left(\mathbb{R}^{N_O^{(L)}}\right)^{\mathbb{Z}}$ defined by

$$f(\mathbf{x}^{(t)}) = g_L \circ g_{L-1} \circ \cdots \circ g_2 \circ g_1(\mathbf{x}^{(t)}),$$

where at least one layer g_ℓ is an LSTM layer.

D.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a family of networks that process data using convolutions. Convolutions are particularly useful to compress information through weighted averages of local information. While CNNs are typically associated with tasks such as image recognition, they can also extract patterns from time series data. Convolutional layers are the defining element of CNNs. Each layer has a kernel that can essentially be seen as a sliding window computing a weighted average of the surrounding observations.

Definition D.8 (Convolutional layer)**.** Let $\mathbf{X} = \begin{bmatrix}\mathbf{x}^{(1)} & \cdots & \mathbf{x}^{(T)}\end{bmatrix} \in \mathbb{R}^{N_I \times T}$ be the matrix of the concatenated sequence $\{\mathbf{x}^{(t)} : t = 1, \ldots, T\}$. Let $\mathbf{W} \in \mathbb{R}^{N_h \times N_w}$ be the convolution's *kernel* with height N_h, and width N_w, where $N_h, N_w \in \mathbb{N}$, and let ϕ be an activation function. A convolutional layer is a map $g : \mathbb{R}^{N_I \times T} \to \mathbb{R}^{(N_I - N_h + 1) \times (T - N_w + 1)}$ such that the element in the mth row and n^{th} column of the output $g(\mathbf{X})$ is given by

$$g(\mathbf{X})_{m,n} = \phi\left(\sum_{i=1}^{N_h}\sum_{j=1}^{N_w} \mathbf{X}_{m-i,n-j} \cdot \mathbf{W}_{i,j}\right).$$

Definition D.9 (CNN)**.** Let $L \in \mathbb{N}$ be the number CNN layers, and g_ℓ be the ℓth layer of the CNN ($\ell = 1, \ldots, L$). The first $\ell^c < L$ layers are convolutional (D.8), and the last $L - \ell^c$ layers are dense (D.2). Let $\text{vec}(\cdot)$ represent the vectorization of a matrix, i.e., its conversion to a column vector. Let $N_I \in \mathbb{N}$ be the number of rows in the input matrix $\mathbf{X}$ and $N_O^{(L)} \in \mathbb{N}$ be the output dimension of the final dense layer. A CNN is a map $f : \mathbb{R}^{N_I \times T} \to \mathbb{R}^{N_O^{(L)}}$ such

that

$$f(\mathbf{X}) = g_L \cdots \circ g_{\ell^c+1} \circ \text{vec} \circ g_{\ell^c} \circ \cdots \circ g_2 \circ g_1(\mathbf{X}).$$

Temporal convolutional neural networks are a particular type of CNNs explicitly built to model time series data. In some cases, the name WaveNet is also used to describe TCNs, referring to van den Oord et al. (2016), the first work introducing a similar topology. The so-called *causal convolution* is the primary mechanism that makes TCNs well-suited for time series modeling. A causal convolution is similar to the convolution used in typical convolutional layers (D.8).

Definition D.10 (Causal convolutional layer)**.** Let $\mathbf{X}$, $\mathbf{W}$, and ϕ be as in Definition D.8. A causal convolutional layer with dilation $D \in \mathbb{N}$ is a map $g : \mathbb{R}^{N_I \times T} \to \mathbb{R}^{(N_I - N_h + 1) \times (T - D(N_w - 1))}$ such that the element in the mth row and n^{th} column of the output $g(\mathbf{X})$ is given by

$$g(\mathbf{X})_{m,n} = \phi\left(\sum_{i=1}^{N_h} \sum_{j=1}^{N_w} \mathbf{X}_{m-i,n-D(N_w-j)} \cdot \mathbf{W}_{i,j}\right).$$

Finally, combining causal convolutional layers and dense layers, we obtain the TCN.

Definition D.11 (TCN)**.** Let $L \in \mathbb{N}$ be the number of layers of a TCN, and g_ℓ be the ℓth layer of the TCN ($\ell = 1, \ldots, L$). The first $\ell^c < L$ layers are causal convolutional layers (D.10), and the last $L - \ell^c$ layers are dense (D.2). Let $N_I \in \mathbb{N}$ be the number of rows in the input matrix $\mathbf{X}$, and $N_O^{(L)} \in \mathbb{N}$ be the output dimension of the final dense layer. A TCN is a map $f : \mathbb{R}^{N_I \times T} \to \mathbb{R}^{N_O^{(L)}}$ such that

$$f(\mathbf{X}) = g_L \cdots \circ g_{\ell^c+1} \circ \text{vec} \circ g_{\ell^c} \circ \cdots \circ g_2 \circ g_1(\mathbf{X}).$$

Let $D_\ell \in \mathbb{N}$ be the dilation of the ℓth layer. A TCN defined such that $\frac{D_\ell}{D_{\ell-1}} = D \in \mathbb{N}$ for all $1 < \ell \leq \ell^c$ is said to have a *dilation factor* of D.

A critical advantage of TCNs is that due to their architecture not implementing any feedback loop, they can be parallelized efficiently, providing significant increases in training speed compared to recurrent architectures. Furthermore, as Definition D.11 and Figure 9 illustrate, adjusting the kernel width, the number of layers or the dilation factor allows us to easily accommodate the length of the sequence processed by the network. This flexibility marks a clear advantage for TCNs compared to RNNs and LSTMs when processing longer sequences. The receptive field of a TCN refers to the lookback window that the network can process.

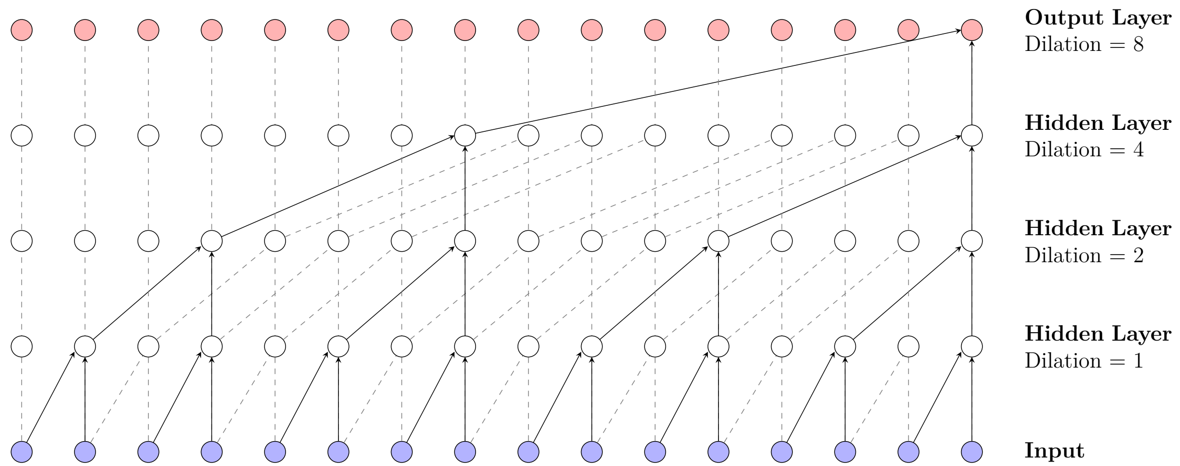

Figure 9: TCN with 4 causal convolutional layers with kernels of width 2 and dilation factor 2, yielding a receptive field size of 16.

For instance, in Figure 9, the TCN has a receptive field size of 16, as the output for time $t+1$ takes as input the past 16 values, $x_{t-15}, \ldots, x_t$. For a TCN with L causal convolutional layers, kernel width N_w, and dilation factor $D > 1$, its receptive field size RFS is given by:

$$\text{RFS} = 1 + \frac{(N_w - 1) \cdot \left(D^L - 1\right)}{D - 1} \tag{8}$$