Recurrent Neural Network Architectures

Chapter · November 2017

DOI: 10.1007/978-3-319-70338-1_3

CITATIONS
0

READS
311

5 authors, including:

Filippo Maria Bianchi
UiT The Arctic University of Norway
57 PUBLICATIONS **329** CITATIONS
SEE PROFILE

Enrico Maiorino
Harvard Medical School
28 PUBLICATIONS **478** CITATIONS
SEE PROFILE

Michael Christian Kampffmeyer
UiT The Arctic University of Norway
37 PUBLICATIONS **147** CITATIONS
SEE PROFILE

Antonello Rizzi
Sapienza University of Rome
186 PUBLICATIONS **1,408** CITATIONS
SEE PROFILE

An overview and comparative analysis of Recurrent Neural Networks for Short Term Load Forecasting

Filippo Maria Bianchi[1a], Enrico Maiorino[b], Michael C. Kampffmeyer[a], Antonello Rizzi[b], Robert Jenssen[a]

[a] *Machine Learning Group, Dept. of Physics and Technology, UiT The Arctic University of Norway, Tromsø, Norway*
[b] *Dept. of Information Engineering, Electronics and Telecommunications, Sapienza University, Rome, Italy*

Abstract

The key component in forecasting demand and consumption of resources in a supply network is an accurate prediction of real-valued time series. Indeed, both service interruptions and resource waste can be reduced with the implementation of an effective forecasting system. Significant research has thus been devoted to the design and development of methodologies for short term load forecasting over the past decades. A class of mathematical models, called Recurrent Neural Networks, are nowadays gaining renewed interest among researchers and they are replacing many practical implementation of the forecasting systems, previously based on static methods. Despite the undeniable expressive power of these architectures, their recurrent nature complicates their understanding and poses challenges in the training procedures. Recently, new important families of recurrent architectures have emerged and their applicability in the context of load forecasting has not been investigated completely yet. In this paper we perform a comparative study on the problem of Short-Term Load Forecast, by using different classes of state-of-the-art Recurrent Neural Networks. We test the reviewed models first on controlled synthetic tasks and then on different real datasets, covering important practical cases of study. We provide a general overview of the most important architectures and we define guidelines for configuring the recurrent networks to predict real-valued time series.

Keywords: Short Term Load Forecast, Recurrent Neural Networks, Time Series Prediction, Echo State Networks, Long Short Term Memory, Gated Recurrent Units, NARX Networks.

1. Introduction

Forecasting the demand of resources within a distribution network of energy, telecommunication or transportation is of fundamental importance for managing the limited availability of the assets. An accurate Short Term Load Forecast (STLF) system [1] can reduce high cost of over- and under-contracts on balancing markets due to load prediction errors. Moreover, it keeps power markets efficient and provides a better understanding of the dynamics of the monitored system [2]. On the other hand, a wrong prediction could cause either a load overestimation, which leads to the excess of supply and consequently more costs and contract curtailments for market participants, or a load underestimation, resulting in failures in gathering enough provisions, thereby more costly supplementary services [3, 4]. These reasons motivated the research of forecasting models capable of reducing this financial distress, by increasing the load forecasting accuracy even by a small percent [5, 6, 7, 8, 9].

The load profile generally follows cyclic and seasonal patterns related to human activities and can be represented by a real-valued time series. The dynamics of the system generating the load time series can vary significantly during the observation period, depending on the nature of the system and on latent, external influences. For this reason, the forecasting accuracy can change considerably among different samples even when using the same prediction model [10]. Over the past years, the STLF problem has been tackled in several research areas [11] by means of many different model-based approaches, each one characterized by different advantages and drawbacks in terms of prediction accuracy, complexity in training, sensitivity to the parameters and limitations in the tractable forecasting horizon [12].

Autoregressive and exponential smoothing models represented for many years the baseline among systems for time series prediction [13]. Such models require to properly select the lagged inputs to identify the correct model orders, a procedure which demands a certain amount of skill and expertise [14]. Moreover, autoregressive models make explicit assumptions about the nature of system under exam. Therefore, their use is limited to those settings in which such assumptions hold and where *a-priori* knowledge on the system is available [15]. Taylor [16] showed that for long forecasting horizons a very basic averaging model, like AutoRegressive Integrated Moving Average or Triple Exponential Smoothing, can outperform more sophisticated alternatives. However, in many complicated systems the properties of linearity and even stationarity of the analyzed time series are not guaranteed. Nonetheless, given their simplicity, autoregressive models have been largely employed as practical implementations of forecast systems.

The problem of time series prediction has been approached within a function approximation framework, by relying on the embedding procedure proposed by Takens [17]. Takens' theorem transforms the prediction problem from time extrapolation to phase space interpolation. In particular, by properly sampling a time dependent quantity $s(t)$, it is possible to predict the value of the k-th sample from the previous samples, given an appropriate choice of the sampling frequency τ and the number of samples m: $s[k] = f(s[k-\tau], \ldots, s[k - m \cdot \tau])$. Through the application of phase-space embedding, regression methods, such as Support Vector Regression (an extension of Support Vector Machines in the continuum) have been applied in time series prediction [18], either by representing the sequential input as a static domain, described by frequency and phase, or by embedding sequential input values in time windows of fixed length. The approach can only succeed if there are no critical temporal dependencies exceeding the windows length, making the SVM unable to learn an internal state representation for sequence learning tasks involving time lags of arbitrary length. Other universal function approximators such as Feed-Forward Artificial Neural Networks [19] and ANFIS (Adaptive Network-Based Fuzzy Inference System) [20] have been employed in time series prediction tasks by selecting a suitable interval of past values from the time series as the inputs and by training the network to forecast one or a fixed number of future values [21, 22, 23, 24, 25, 26, 27]. The operation is repeated to forecast next values by translating the time window of the considered inputs [28]. While this approach proved to be effective in many circumstances [29, 30, 31, 32], it does not treat temporal ordering as an explicit feature of the time series and, in general, is not suitable in cases where the time series have significantly different lengths. On this account, a Recurrent Neural Network (RNN) is a more flexible model, since it encodes the temporal context in its feedback connections, which are capable of capturing the time varying dynamics of the underlying system [33, 34].

RNNs are a special class of Neural Networks characterized by internal self-connections, which can, in principle, any nonlinear dynamical system, up to a given degree of accuracy [35]. RNNs and their variants have been used in many contexts where the temporal dependency in the data is an important implicit feature in the model design. Noteworthy applications of RNNs include sequence transduction [36], language modeling [37, 38, 39, 40], speech recognition [41], learning word embeddings [42], audio modeling [43], handwriting recognition [44, 45], and image generation [46]. In many of these works a popular variant of RNN was used, called Long-Short Term Memory [47]. This latter has recently earned significant attention due to its capability of storing information for very long periods of time.

As an RNN processes sequential information, it performs the same operations on every element of the input sequence. Its output, at each time step, depends on previous inputs and past computations. This allows the network to develop a memory of previous events, which is implicitly encoded in its hidden state variables. This is certainly different from traditional feedforward neural networks, where it is assumed that all inputs (and outputs) are independent of each other. Theoretically, RNNs can remember arbitrarily long sequences. However, their memory is in practice limited by their finite size and, more critically, by the suboptimal training of their parameters. To overcome memory limitations, recent research efforts have led to the design of novel RNN architectures, which are equipped with an external, permanent memory capable of storing information for indefinitely long amount of time [48, 49].

Contrarily to other linear models adopted for prediction, RNNs can learn functions of arbitrary complexity and they can deal with time series data possessing properties such as saturation or exponential effects and nonlinear interactions between latent variables. However, if the temporal dependencies of data are prevalently

contained in a finite and small time interval, the use of RNNs can be unnecessary. In these cases performances, both in terms of computational resources required and accuracy, are generally lower than the ones of time-window approaches, like ARIMA, SVM, Multi-Layer Perceptron and ANFIS. On the other hand, in many load forecasting problems the time series to be predicted are characterized by long temporal dependencies, whose extent may vary in time or be unknown in advance. In all these situations, the use of RNNs may turn out to be the best solution.

Despite the STLF problem has been one of the most important applications for both early RNNs models [50] and most recent ones [51], an up-to-date and comprehensive analysis of the modern RNN architectures applied to the STLF problem is still lacking. In several recent works on STFL, NARX networks (see Sec. 4.1) or Echo State Networks (see Sec. 4.2) are adopted for time series prediction and their performance is usually compared with standard static models, rather than with other RNN architectures. With this paper, we aim to fill these gaps by performing a comparative study on the problem of STLF using different classes of state-of-the-art RNNs. We provide an introduction to the RNN framework, presenting the most important architectures and their properties. We also furnish the guidelines for configuring and training the different RNN models to predict real-valued time series. In practice, we formulate the STLF problem as the prediction of a real-valued univariate time series, given its past values as input. In some cases, beside the time series of past target values, additional "context" time series are fed to the network in order to provide exogenous information related to the environment in which the system to be modeled operates.

The paper is structured as follows.

In Sec. 2 we provide a general overview of a standard RNN architecture and we discuss its general properties. We also discuss the main issues encountered in the training phase, the most common methodologies for learning the model parameters and common ways of defining the loss function to be optimized during the training.

In Sec. 3, we present the most basic architecture, called Elman RNN, and then we analyze two important variants, namely the Long-Short Term Memory and Gated Recurrent Units networks. Despite the recent popularity of these architectures [52], their application to prediction of real-valued time series has been limited so far [53]. For each RNN, we provide a brief review, explaining its main features, the approaches followed in the training stage and a short list of the main works concerning time series prediction in which the specific network has been applied.

Successively, in Sec. 4 we illustrate two particular RNN architectures, which differ from the previous ones, mainly due to their training procedure. In particular, we analyze the Nonlinear AutoRegressive with eXogenous inputs (NARX) neural network and the Echo State Network (ESN). These architectures have been successfully applied in the literature of time series prediction and they provide important advantages with respect to traditional models, due to their easy applicability and fast training procedures.

In Sec. 5 we describe three synthetic datasets, used to test and to compare the computational capabilities of the five RNN architectures in a controlled environment.

In Sec. 6, we present three real-world datasets of time series relative to the load profile in energy distribution and telecommunication networks. For each dataset, we perform a series of analysis with the purpose of choosing a suitable preprocessing for the data.

Sec. 7 is dedicated to the experiments and to the discussion of the performance of the RNN models. The first part of the experimental section focuses on the benchmark tests, while in the second part we employ the RNNs to solve STLF tasks on real-world time series.

Finally, in Sec. 8 we discuss our conclusions.

2. Properties and Training in Recurrent Neural Networks

RNNs are learning machines that recursively compute new states by applying transfer functions to previous states and inputs. Typical transfer functions are composed by an affine transformation followed by a nonlinear function, which are chosen depending on the nature of the particular problem at hand. It has been shown by Maass et al. [54] that RNNs possess the so-called universal approximation property, that is, they are capable of approximating arbitrary nonlinear dynamical systems (under loose regularity conditions) with

arbitrary precision, by realizing complex mappings from input sequences to output sequences [55]. However, the particular architecture of an RNN determines how information flows between different neurons and its correct design is crucial in the realization of a robust learning system. In the context of prediction, an RNN is trained on input temporal data $\mathbf{x}(t)$ in order to reproduce a desired temporal output $\mathbf{y}(t)$. $\mathbf{y}(t)$ can be any time series related to the input and even a temporal shift of $\mathbf{x}(t)$ itself. The most common training procedures are gradient-based, but other techniques have been proposed, based on derivative-free approaches or convex optimization [56, 57]. The objective function to be minimized is a loss function, which depends on the error between the estimated output $\hat{\mathbf{y}}(t)$ and the actual output of the network $\mathbf{y}(t)$. An interesting aspect of RNNs is that, upon suitable training, they can also be executed in generative mode, as they are capable of reproducing temporal patterns similar to those they have been trained on [46].

The architecture of a simple RNN is depicted in Fig. 1. In its most general form an RNN can be seen as

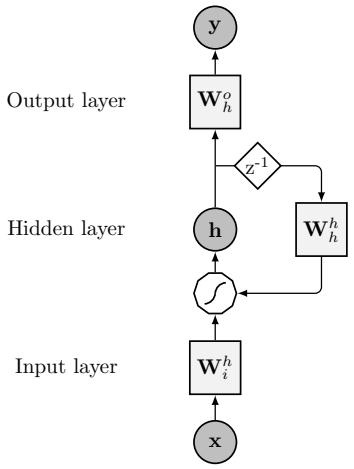


Figure 1: Schematic depiction of a simple RNN architecture. The circles represent input $\mathbf{x}$, hidden, $\mathbf{h}$, and output nodes, $\mathbf{y}$, respectively. The solid squares $\mathbf{W}_i^h$, $\mathbf{W}_h^h$ and $\mathbf{W}_h^o$ are the matrices which represent input, hidden and output weights respectively. Their values are commonly tuned in the training phase through gradient descent. The polygon represents the non-linear transformation performed by neurons and z^{-1} is the unit delay operator.

a weighted, directed and cyclic graph that contains three different kinds of nodes, namely the input, hidden and output nodes [58]. Input nodes do not have incoming connections, output nodes do not have outgoing connections, hidden nodes have both. An edge can connect two different nodes which are at the same or at different time instants. In this paper, we adopt the time-shift operator z^n to represent a time delay of n time steps between a source and a destination node. Usually $n = -1$, but also lower values are admitted and they represent the so called skip connections [59]. Self-connecting edges always implement a lag operator with $|n| \geq 1$. In some particular cases, the argument of the time-shift operator is positive and it represents a forward-shift in time [60]. This means that a node receives as input the content of a source node in a future time interval. Networks with those kind of connections are called bidirectional RNNs and are based on the idea that the output at a given time may not only depend on the previous elements in the sequence, but also on future ones [61]. These architectures, however, are not reviewed in this work as we only focus on RNNs with $n = -1$.

While, in theory, an RNN architecture can model any given dynamical system, practical problems arise during the training procedure, when model parameters must be learned from data in order to solve a target task. Part of the difficulty is due to a lack of well established methodologies for training different types of models. This is also because a general theory that might guide designer decisions has lagged behind the feverish pace of novel architecture designs [62, 63]. A large variety of novel strategies and heuristics have arisen from the literature in the past the years [64, 65] and, in many cases, they may require a considerable amount of expertise from the user to be correctly applied. While the standard learning procedure is based on gradient optimization, in some RNN architectures the weights are trained following different approaches [66, 67], such as real-time recurrent learning [68], extended Kalman filters [69] or evolutionary algorithms [70], and in some cases they are not learned at all [71].

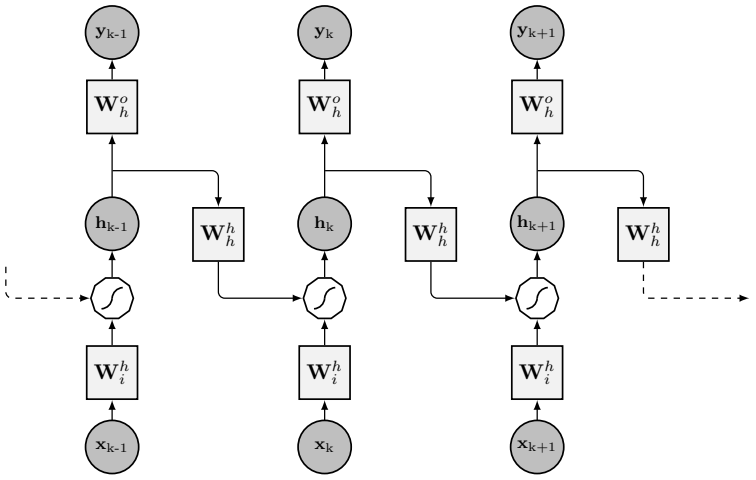


Figure 2: The diagram depicts the RNN from Fig. 1, being unfolded (or unrolled) into a FFNN. As we can see from the image, each input $\mathbf{x}_t$ and output $\mathbf{y}_t$ are relative to different time intervals. Unlike a traditional deep FFNN, which uses different parameters in each layer, an unfolded RNN shares the same weights across every time step. In fact, the input weights matrix $\mathbf{W}_i^h$, the hidden weights matrix $\mathbf{W}_h^h$ and the output weights matrix $\mathbf{W}_h^o$ are constrained to keep the same values in each time interval.

2.1. Backpropagation Through Time

Gradient-based learning requires a closed-form relation between the model parameters and the loss function. This relation allows to propagate the gradient information calculated on the loss function back to the model parameters, in order to modify them accordingly. While this operation is straightforward in models represented by a directed acyclic graph, such as a FeedForward Neural Network (FFNN), some caution must be taken when this reasoning is applied to RNNs, whose corresponding graph is cyclic. Indeed, in order to find a direct relation between the loss function and the network weights, the RNN has to be represented as an equivalent infinite, acyclic and directed graph. The procedure is called *unfolding* and consists in replicating the network's hidden layer structure for each time interval, obtaining a particular kind of FFNN. The key difference of an unfolded RNN with respect to a standard FFNN is that the weight matrices are constrained to assume the same values in all replicas of the layers, since they represent the recursive application of the same operation.

Fig. 2 depicts the unfolding of the RNN, previously reported in Fig. 1. Through this transformation the network can be trained with standard learning algorithms, originally conceived for feedforward architectures. This learning procedure is called Back Propagation Through Time (BPTT) [72] and is one of the most successful techniques adopted for training RNNs. However, while the network structure could in principle be replicated an infinite number of times, in practice the unfolding is always truncated after a finite number of time instants. This maintains the complexity (depth) of the network treatable and limits the issue of the vanishing gradient (as discussed later). In this learning procedure called Truncated BPPT [73], the folded architecture is repeated up to a given number of steps τ_b, with τ_b upperbounded by the time series length T. The size of the truncation depends on the available computational resources, as the network grows deeper by repeating the unfolding, and on the expected maximum extent of time dependencies in data. For example, in a periodic time series with period t it may be unnecessary, or even detrimental, to set $\tau_b > t$.

Another variable we consider is the frequency τ_f at which the BPTT calculates the backpropagated gradients. In particular, let us define with BPTT(τ_b, τ_f) the truncated backpropagation that processes the sequence one time step at a time, and every τ_f time steps, it runs BPTT for τ_b time steps [74]. Very often the term τ_f is omitted in the literature, as it is assumed equal to 1, and only the value for τ_b is specified. We refer to the case $\tau_f = 1$ and $\tau_b = n$ as *true* BPTT, or BPTT$(n, 1)$.

In order to improve the computational efficiency of the BPTT, the ratio τ_b/τ_f can be decremented, effectively reducing the frequency of gradients evaluation. An example, is the so-called *epochwise* BPTT or

BPTT(n, n), where $\tau_b = \tau_f$ [75]. In this case, the ratio $\tau_b/\tau_f = 1$. However, the learning procedure is in general much less accurate than BPTT$(n, 1)$, since the gradient is truncated too early for many values on the boundary of the backpropagation window.

A better approximation of the true BPTT is reached by taking a large difference $\tau_b - \tau_f$, since no error in the gradient is injected for the earliest $\tau_b - \tau_f$ time steps in the buffer. A good trade-off between accuracy and performance is BPTT$(2n, n)$, which keeps the ratio $\tau_b/\tau_f = 2$ sufficiently close to 1 and the difference $\tau_b - \tau_f = n$ is large as in the true BPTT [73]. Through preliminary experiments, we observed that BPTT$(2n, n)$ achieves comparable performance to BPTT$(n, 1)$, in a significantly reduced training time. Therefore, we followed this procedure in all our experiments.

2.2. Gradient descent and loss function

Training a neural network commonly consists in modifying its parameters through a gradient descent optimization, which minimizes a given loss function that quantifies the accuracy of the network in performing the desired task. The gradient descent procedure consists in repeating two basic steps until convergence is reached. First, the loss function L_k is evaluated on the RNN configured with weights $\mathbf{W}_k$, when a set of input data $\mathcal{X}_k$ are processed (forward pass). Note that with $\mathbf{W}_k$ we refer to *all* network parameters, while the index k identifies their values at epoch k, as they are updated during the optimization procedure. In the second step, the gradient $\partial L_k/\partial \mathbf{W}_k$ is back-propagated through the network in order to update its parameters (backward pass).

In a time series prediction problem, the loss function evaluates the dissimilarity between the predicted values and the actual future values of the time series, which is the ground truth. The loss function can be defined as

$$L_k = E\left(\mathcal{X}_k, \mathcal{Y}_k^*; \mathbf{W}_k\right) + R_\lambda\left(\mathbf{W}_k\right), \tag{1}$$

where E is a function that evaluates the prediction error of the network when it is fed with inputs in $\mathcal{X}_k$, in respect to a desired response $\mathcal{Y}_k^*$. R_λ is a regularization function that depends on a hyperparameter λ, which weights the contribution of the regularization in the total loss.

The error function E that we adopt in this work is Mean Square Error (MSE). It is defined as

$$\mathrm{MSE}(\mathcal{Y}_k, \mathcal{Y}_k^*) = \frac{1}{|\mathcal{X}_k|} \sum_{\mathbf{x} \in \mathcal{X}_k} \left(\mathbf{y}_\mathbf{x} - \mathbf{y}_\mathbf{x}^*\right)^2, \tag{2}$$

where $\mathbf{y}_\mathbf{x} \in \mathcal{Y}_k$ is the output of the RNN (configured with parameters $\mathbf{W}_k$) when the input $\mathbf{x} \in \mathcal{X}_k$ is processed and $\mathbf{y}_\mathbf{x}^* \in \mathcal{Y}_k^*$ is the ground-truth value that the network must learn to reproduce.

The regularization term R_λ introduces a bias that improves the generalization capabilities of the RNN, by reducing overfitting on the training data. In this work, we consider four types of regularization:

1. L_1: the regularization term in Eq. 1 has the form $R_\lambda(\mathbf{W}_k) = \lambda_1 \|\mathbf{W}_k\|_1$. L_1 regularization enforces sparsity in the network parameters, is robust to noisy outliers and it can possibly deliver multiple optimal solutions. However, this regularization can produce unstable results, in the sense that a small variation in the training data can yield very different outcomes.
2. L_2: in this case, $R_\lambda(\mathbf{W}_k) = \lambda_2 \|\mathbf{W}_\mathbf{k}\|_2$. This function penalizes large magnitudes in the parameters, favouring dense weight matrices with low values. This procedure is more sensitive to outliers, but is more stable than L_1. Usually, if one is not concerned with explicit features selection, the use of L_2 is preferred.
3. *Elastic net penalty:* combines the two regularizations above, by joining both L_1 and L_2 terms as $R_\lambda(\mathbf{W}_k) = \lambda_1 \|\mathbf{W}_k\|_1 + \lambda_2 \|\mathbf{W}_k\|_2$. This regularization method overcomes the shortcomings of the L_1 regularization, which selects a limited number of variables before it saturates and, in case of highly correlated variables, tends to pick only one and ignore the others. Elastic net penalty generalizes the L_1 and L_2 regularization, which can be obtained by setting $\lambda_2 = 0$ and $\lambda_1 = 0$, respectively.

4. *Dropout:* rather than defining an explicit regularization function $R_\lambda(\cdot)$, dropout is implemented by keeping a neuron active during each forward pass in the training phase with some probability. Specifically, one applies a randomly generated mask to the output of the neurons in the hidden layer. The probability of each mask element to be 0 or 1 is defined by a hyperparameter p_{drop}. Once the training is over, the activations are scaled by p_{drop} in order to maintain the same expected output. Contrarily to feedforward architectures, a naive dropout in recurrent layers generally produces bad performance and, therefore, it has usually been applied only to input and output layers of the RNN [76]. However, in a recent work, Gal and Ghahramani [77] shown that this shortcoming can be circumvented by dropping the same network units in each epoch of the gradient descent. Even if this formulation yields a slightly reduced regularization, nowadays this approach is becoming popular [78, 79] and is the one followed in this paper.

Beside the ones discussed above, several other kinds of regularization procedures have been proposed in the literature. Examples are the stochastic noise injection [80] and the max-norm constraint [81], which, however, are not considered in our experiments.

2.3. Parameters update strategies

Rather than evaluating the loss function over the entire training set to perform a single update of the network parameters, a very common approach consists in computing the gradient over mini-batches $\mathcal{X}_k$ of the training data. The size of the batch is usually set by following rules of thumb [82].

This gradient-update method is called Stochastic Gradient Descent (SGD) and, in presence of a non-convex function, its convergence to a local minimum is guaranteed (under some mild assumptions) if the learning rate is sufficiently small [83]. The update equation reads

$$\mathbf{W}_{k+1} = \mathbf{W}_k + \eta \nabla L_k(\mathbf{W}_k), \tag{3}$$

where η is the *learning rate*, an important hyperparameter that must be carefully tuned to achieve an effective training [84]. In fact, a large learning rate provides a high amount of kinetic energy in the gradient descent, which causes the parameter vector to bounce, preventing the access to narrow area of the search space, where the loss function is lower. On the other hand, a strong decay can excessively slow the training procedure, resulting in a waste of computational time.

Several solutions have been proposed over the years, to improve the convergence to the optimal solution [85]. During the training phase it is usually helpful to anneal η over time or when the performance stops increasing. A method called *step decay* reduces the learning rate by a factor α, if after a given number of epochs the loss has not decreased. The *exponential decay* and the *fractional decay* instead, have mathematical forms $\eta = \eta_0 e^{-\alpha k}$ and $\eta = \frac{\eta_0}{(1+\alpha k)}$, respectively. Here α and η_0 are hyperparameters, while k is the current optimization epoch. In our experiments, we opted for the step decay annealing, when we train the networks with SGD.

Even if SGD usually represents a safe optimization procedure, its rate of convergence is slow and the gradient descent is likely to get stuck in a saddle point of the loss function landscape [86]. Those issues have been addressed by several alternative strategies proposed in the literature for updating the network parameters. In the following we describe the most commonly used ones.

Momentum. In this first-order method, the weights $\mathbf{W}_k$ are updated according to a linear combination of the current gradient $\nabla L_k(\mathbf{W}_k)$ and the previous update $\mathbf{V}_{k-1}$, which is scaled by a hyperparameter μ:

$$\begin{aligned} \mathbf{V}_k &= \mu \mathbf{V}_{k-1} - \eta \nabla L_k(\mathbf{W}_k), \\ \mathbf{W}_{k+1} &= \mathbf{W}_k + \mathbf{V}_k. \end{aligned} \tag{4}$$

With this approach, the updates will build up velocity toward a direction that shows a consistent gradient [74]. A common choice is to set $\mu = 0.9$.

A variant of the original formulation is the *Nesterov momentum*, which often achieves a better convergence rate, especially for smoother loss functions [87]. Contrarily to the original momentum, the gradient is

evaluated at an approximated future location, rather than at the current position. The update equations are

$$\begin{aligned} \mathbf{V}_k &= \mu \mathbf{V}_{k-1} - \eta \nabla L_k(\mathbf{W}_k + \mu \mathbf{V}_{k-1}), \\ \mathbf{W}_{k+1} &= \mathbf{W}_k + \mathbf{V}_k. \end{aligned} \tag{5}$$

Adaptive learning rate. The first adaptive learning rate method, proposed by Duchi et al. [88], is Adagrad. Unlike the previously discussed approaches, Adagrad maintains a different learning rate for each parameter. Given the update information from all previous iterations $\nabla L_k(\mathbf{W}_j)$, with $j \in \{0, 1, \cdots, k\}$, a different update is specified for each parameter i of the weight matrix:

$$\mathbf{W}_{k+1}^{(i)} = \mathbf{W}_k^{(i)} - \eta \frac{\nabla L_k\left(\mathbf{W}_k^{(i)}\right)}{\sqrt{\sum_{j=0}^{k} \nabla L_k\left(\mathbf{W}_j^{(i)}\right)^2 + \epsilon}}, \tag{6}$$

where ϵ is a small term used to avoid division by 0. A major drawback with Adagrad is the unconstrained growth of the accumulated gradients over time. This can cause diminishing learning rates that may stop the gradient descent prematurely.

A procedure called RMSprop [89] attempts to solve this issue by using an exponential decaying average of square gradients, which discourages an excessive shrinkage of the learning rates:

$$\begin{aligned} v_k^{(i)} &= \begin{cases} (1-\delta) \cdot v_{k-1}^{(i)} + \delta \nabla L_k\left(\mathbf{W}_k^{(i)}\right)^2 & \text{if } \nabla L_k\left(\mathbf{W}_k^{(i)}\right) > 0 \\ (1-\delta) \cdot v_{k-1}^{(i)} & \text{otherwise} \end{cases} \\ \mathbf{W}_{k+1}^{(i)} &= \mathbf{W}_k^{(i)} - \eta v_k^{(i)}. \end{aligned} \tag{7}$$

According to the update formula, if there are oscillation in gradient updates, the learning rate is reduced by $1 - \delta$, otherwise it is increased by δ. Usually the decay rate is set to $\delta = 0.01$.

Another approach called Adam and proposed by Kingma and Ba [90], combines the principles of Adagrad and momentum update strategies. Usually, Adam is the adaptive learning method that yields better results and, therefore, is the gradient descent strategy most used in practice. Like RMSprop, Adam stores an exponentially decaying average of gradients squared, but it also keeps an exponentially decaying average of the moments of the gradients. The update difference equations of Adam are

$$\begin{aligned} m_k &= \beta_1 m_{k-1} + (1-\beta_1) \nabla L_k\left(\mathbf{W}_k^{(i)}\right), \\ v_k &= \beta_2 v_{k-1} + (1-\beta_2) \nabla L_k\left(\mathbf{W}_k^{(i)}\right)^2, \\ \hat{m}_k &= \frac{m_k}{1-\beta_1^k}, \;\; \hat{v}_k = \frac{v_k}{1-\beta_2^k}, \\ \mathbf{W}_{k+1} &= \mathbf{W}_k + \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}} \hat{m}_k \; . \end{aligned} \tag{8}$$

m corresponds to the first moment and v is the second moment. However, since both m and v are initialized as zero-vectors, they are biased towards 0 during the first epochs. To avoid this effect, the two terms are corrected as $\hat{m}_t$ and $\hat{v}_t$. Default values of the hyperparameters are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$.

Second-order methods. The methods discussed so far only consider first-order derivatives of the loss function. Due to this approximation, the landscape of the loss function locally looks and behaves like a plane. Ignoring the curvature of the surface may lead the optimization astray and it could cause the training to progress very slowly. However, second-order methods involve the computation of the Hessian, which is expensive and usually untreatable even in networks of medium size. A Hessian-Free (HF) method that considers derivatives of the second order, without explicitly computing the Hessian, has been proposed by Martens [91]. This latter, unlike other existing HF methods, makes use of the positive semi-definite Gauss-Newton curvature

matrix and it introduces a damping factor based on the Levenberg-Marquardt heuristic, which permits to train networks more effectively. However, Sutskever et al. [92] showed that HF obtains similar performance to SGD with Nesterov momentum. Despite being a first-order approach, Nestrov momentum is capable of accelerating directions of low-curvature just like a HF method and, therefore, is preferred due to its lower computational complexity.

2.4. Vanishing and exploding gradient

Increasing the depth in an RNN, in general, improves the memory capacity of the network and its modeling capabilities [93]. For example, stacked RNNs do outperform shallow ones with the same hidden size on problems where it is necessary to store more information throughout the hidden states between the input and output layer [94]. One of the principal drawback of early RNN architectures was their limited memory capacity, caused by the *vanishing* or *exploding gradient* problem [95], which becomes evident when the information contained in past inputs must be retrieved after a long time interval [96]. To illustrate the issue of vanishing gradient, one can consider the influence of the loss function L_t (that depends on the network inputs and on its parameters) on the network parameters $\mathbf{W}_t$, when its gradient is backpropagated through the unfolded The network Jacobian reads as

$$\frac{\partial L[t]}{\partial W} = \sum_{\tau} \frac{\partial L[t]}{\partial h[t]} \frac{\partial h[t]}{\partial h[\tau]} \frac{\partial h[\tau]}{\partial W}. \tag{9}$$

In the previous equation, the partial derivatives of the states with respect to their previous values can be factorized as

$$\frac{\partial h[t]}{\partial h[\tau]} = \frac{\partial h[t]}{\partial h[t-1]} \cdots \frac{\partial h[\tau+1]}{\partial h[\tau]} = f_t^{'} \cdots f_{\tau+1}^{'}. \tag{10}$$

To ensure local stability, the network must operate in a ordered regime [97], a property ensured by the condition $|f_t^{'}| < 1$. However, in this case the product expanded Eq. 10 rapidly (exponentially) converges to 0, when $t - \tau$ increases. Consequently, the sum in Eq. 9 becomes dominated by terms corresponding to short-term dependencies and the vanishing gradient effect occurs. As principal side effect, the weights are less and less updated as the gradient flows backward through the layers of the network. On the other hand, the phenomenon of exploding gradient appears when $|f_t^{'}| > 1$ and the network becomes locally unstable. Even if global stability can still be obtained under certain conditions, in general the network enters into a chaotic regime, where its computational capability is hindered [98].

Models with large recurrent depths exacerbate these gradient-related issues, since they posses more nonlinearities and the gradients are more likely to explode or vanish. A common way to handle the exploding gradient problem, is to clip the norm of the gradient if it grows above a certain threshold. This procedure relies on the assumption that exploding gradients only occur in contained regions of the parameters space. Therefore, clipping avoids extreme parameter changes without overturning the general descent direction [99].

On the other hand, different solutions have been proposed to tackle the vanishing gradient issue. A simple, yet effective approach consists in initializing the weights to maintain the same variance withing the activations and back-propagated gradients, as one moves along the network depth. This is obtained with a random initialization that guarantees the variance of the components of the weight matrix in layer l to be $\mathrm{Var}(\mathbf{W}_l) = 2/(N_{l-1} + N_{l+1})$, N_{l-1} and N_{l+1} being the number of units in the previous and the next layer respectively [100]. He et al. [101] proposed to initialize the network weights by sampling them from an uniform distribution in $[0, 1]$ and then rescaling their values by $1/\sqrt{N_h}$, N_h being the total number of hidden neurons in the network. Another option, popular in deep FFNN, consists in using ReLU [102] as activation function, whose derivative is 0 or 1, and it does not cause the gradient to vanish or explode. Regularization, besides preventing unwanted overfitting in the training phase, proved to be useful in dealing with exploding gradients. In particular, L_1 and L_2 regularizations constrain the growth of the components of the weight matrices and consequently limit the values assumed by the propagated gradient [38]. Another popular solution is adopting gated architectures, like Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU), which have been specifically designed to deal with vanishing gradients and allow the network

to learn much longer-range dependencies. Srivastava et al. [103] proposed an architecture called *Highway Network*, which allows information to flow across several layers without attenuation. Each layer can smoothly vary its behavior between that of a plain layer, implementing an affine transform followed by a non-linear activation, and that of a layer which simply passes its input through. Optimization in highway networks is virtually independent of depth, as information can be routed (unchanged) through the layers. The Highway architecture, initially applied to deep FFNN [104], has recently been extended to RNN where it dealt with several modeling and optimization issues [78].

Finally, gradient-related problems can be avoided by repeatedly selecting new weight parameters using random guess or evolutionary approaches [70, 105]; in this way the network is less likely to get stuck in local minima. However, convergence time of these procedures is time-consuming and can be impractical in many real-world applications. A solution proposed by Schmidhuber et al. [56], consists in evolving only the weights of non-linear hidden units, while linear mappings from hidden to output units are tuned using fast algorithms for convex problem optimization.

3. Recurrent Neural Networks Architectures

In this section, we present three different RNN architectures trainable through the BPPT procedure, which we employ to predict real-valued time series. First, in Sec. 3.1 we present the most basic version of RNN, called Elman RNN. In Sec. 3.2 and 3.3 we discuss two gated architectures, which are LSTM and GRU. For each RNN model, we provide a quick overview of the main applications in time series forecasting and we discuss its principal features.

3.1. Elman Recurrent Neural Network

The Elman Recurrent Neural Network (ERNN), also known as *Simple RNN* or *Vanillan RNN*, is depicted in Fig. 1 and is usually considered to be the most basic version of RNN. Most of the more complex RNN architectures, such as LSTM and GRU, can be interpreted as a variation or as an extension of ERNNs.

ERNN have been applied in many different contexts. In natural language processing applications, ERNN demonstrated to be capable of learning grammar using a training set of unannotated sentences to predict successive words in the sentence [106, 107]. Mori and Ogasawara [108] studied ERNN performance in short-term load forecasting and proposed a learning method, called "diffusion learning" (a sort of momentum-based gradient descent), to avoid local minima during the optimization procedure. Cai et al. [109] trained a ERNN with a hybrid algorithm that combines particle swarm optimization and evolutionary computation to overcome the local minima issues of gradient-based methods. Furthermore, ERNNs have been employed by Cho [110] in tourist arrival forecasting and by Mandal et al. [111] to predict electric load time series. Due to the critical dependence of electric power usage on the day of the week or month of the year, a preprocessing step is performed to cluster similar days according to their load profile characteristics. Chitsaz et al. [112] proposes a variant of ERNN called Self-Recurrent Wavelet Neural Network, where the ordinary nonlinear activation functions of the hidden layer are replaced with wavelet functions. This leads to a sparser representation of the load profile, which demonstrated to be helpful for tackling the forecast task through smaller and more easily trainable networks.

The layers in a RNN can be divided in *input layers*, *hidden layers* and the *output layers* (see Fig. 1). While input and output layers are characterized by feedforward connections, the hidden layers contain recurrent ones. At each time step t, the input layer process the component $\mathbf{x}[t] \in \mathbb{R}^{N_i}$ of a serial input $\mathbf{x}$. The time series $\mathbf{x}$ has length T and it can contain real values, discrete values, one-hot vectors, and so on. In the input layer, each component $\mathbf{x}[t]$ is summed with a bias vector $\mathbf{b}_i \in \mathbb{R}^{N_h}$ (N_h is the number of nodes in the hidden layer) and then is multiplied with the input weight matrix $\mathbf{W}_i^h \in \mathbb{R}^{N_i \times N_h}$. Analogously, the internal state of the network $\mathbf{h}[t-1] \in \mathbb{R}^{N_h}$ from the previous time interval is first summed with a bias vector $\mathbf{b}_h \in \mathbb{R}^{N_h}$ and then multiplied by the weight matrix $\mathbf{W}_h^h \in \mathbb{R}^{N_h \times N_h}$ of the recurrent connections. The transformed current input and past network state are then combined and processed by the neurons in the hidden layers, which apply a non-linear transformation. The difference equations for the update of the internal state and

the output of the network at a time step t are:

$$\begin{aligned} \mathbf{h}[t] &= f\left(\mathbf{W}_i^h\left(\mathbf{x}[t]+\mathbf{b}_i\right)+\mathbf{W}_h^h\left(\mathbf{h}[t-1]+\mathbf{b}_h\right)\right), \\ \mathbf{y}[t] &= g\left(\mathbf{W}_h^o\left(\mathbf{h}[t]+\mathbf{b}_o\right)\right), \end{aligned} \tag{11}$$

where $f(\cdot)$ is the activation function of the neurons, usually implemented by a sigmoid or by a hyperbolic tangent. The hidden state $\mathbf{h}[t]$ conveys the content of the memory of the network at time step t, is typically initialized with a vector of zeros and it depends on past inputs and network states. The output $\mathbf{y}[t] \in \mathbb{R}^{N_o}$ is computed through a transformation $g(\cdot)$, usually linear, on the matrix of the output weights $\mathbf{W}_h^o \in \mathbb{R}^{N_r \times N_o}$ applied to the sum of the current state $\mathbf{h}[t]$ and the bias vector $\mathbf{b}_o \in \mathbb{R}^{N_o}$. All the weight matrices and biases can be trained through gradient descent, according to the BPPT procedure. Unless differently specified, in the following to compact the notation we omit the bias terms by assuming $\mathbf{x} = [\mathbf{x}; 1]$, $\mathbf{h} = [\mathbf{h}; 1]$, $\mathbf{y} = [\mathbf{y}; 1]$ and by augmenting $\mathbf{W}_i^h$, $\mathbf{W}_h^h$, $\mathbf{W}_h^o$ with an additional column.

3.2. Long Short-Term Memory

The Long Short-Term Memory (LSTM) architecture was originally proposed by Hochreiter and Schmidhuber [47] and is widely used nowadays due to its superior performance in accurately modeling both short and long term dependencies in data. LSTM tries to solve the vanishing gradient problem by not imposing any bias towards recent observations, but it keeps constant error flowing back through time. LSTM works essentially in the same way as the ERNN architecture, with the difference that it implements a more elaborated internal processing unit called *cell*.

LSTM has been employed in numerous sequence learning applications, especially in the field of natural language processing. Outstanding results with LSTM have been reached by Graves and Schmidhuber [44] in unsegmented connected handwriting recognition, by Graves et al. [113] in automatic speech recognition, by Eck and Schmidhuber [114] in music composition and by Gers and Schmidhuber [115] in grammar learning. Further successful results have been achieved in the context of image tagging, where LSTM have been paired with convolutional neural network, to provide annotations on images automatically [116].

However, few works exist where LSTM has been applied to prediction of real-valued time series. Ma et al. [117] evaluated the performances of several kinds of RNNs in short-term traffic speed prediction and compared them with other common methods like SVMs, ARIMA, and Kalman filters, finding that LSTM networks are nearly always the best approach. Pawlowski and Kurach [118] utilized ensembles of LSTM and feedforward architectures to classify the danger from concentration level of methane in a coal mine, by predicting future concentration values. By following a hybrid approach, Felder et al. [119] trains a LSTM network to output the parameter of a Gaussian mixture model that best fits a wind power temporal profile.

While an ERNN neuron implements a single nonlinearity $f(\cdot)$ (see Eq. 11), a LSTM cell is composed of 5 different nonlinear components, interacting with each other in a particular way. The internal state of a cell is modified by the LSTM only through linear interactions. This permits information to backpropagate smoothly across time, with a consequent enhancement of the memory capacity of the cell. LSTM protects and controls the information in the cell through three gates, which are implemented by a sigmoid and a pointwise multiplication. To control the behavior of each gate, a set of parameters are trained with gradient descent, in order to solve a target task.

Since its initial definition [47], several variants of the original LSTM unit have been proposed in the literature. In the following, we refer to the commonly used architecture proposed by Graves and Schmidhuber [120]. A schema of the LSTM cell is depicted in Fig. 3.

The difference equations that define the forward pass to update the cell state and to compute the output

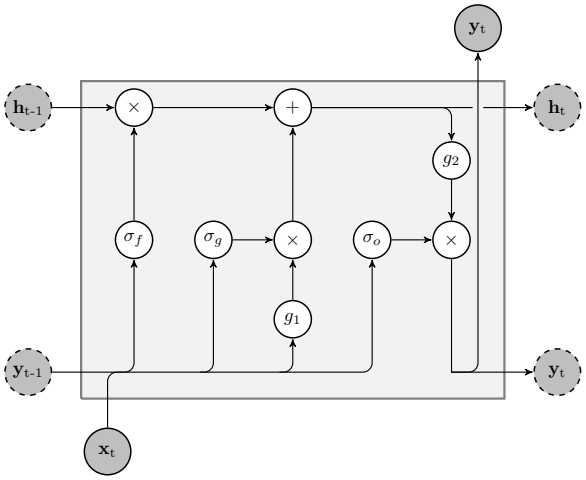


Figure 3: Illustration of a cell in the LSTM architecture. Dark gray circles with a solid line are the variables whose content is exchanged with the input and output of the cell. Dark gray circles with a dashed line represent the internal state variables, whose content is exchanged between the cells of the hidden layer. Operators g_1 and g_2 are the non-linear transformation, usually implemented as a hyperbolic tangent. White circles with $+$ and $\times$ represent linear operations, while σ_f, σ_u and σ_o are the sigmoids used in the forget, update and output gates respectively.

are listed below.

$$\begin{aligned}
\text{forget gate}: &\ \sigma_f[t] = \sigma\left(\mathbf{W}_f\mathbf{x}[t] + \mathbf{R}_f\mathbf{y}[t-1] + \mathbf{b}_f\right), \\
\text{candidate state}: &\ \tilde{\mathbf{h}}[t] = g_1\left(\mathbf{W}_h\mathbf{x}[t] + \mathbf{R}_h\mathbf{y}[t-1] + \mathbf{b}_h\right), \\
\text{update gate}: &\ \sigma_u[t] = \sigma\left(\mathbf{W}_u\mathbf{x}[t] + \mathbf{R}_u\mathbf{y}[t-1] + \mathbf{b}_u\right), \\
\text{cell state}: &\ \mathbf{h}[t] = \sigma_u[t] \odot \tilde{\mathbf{h}}[t] + \sigma_f[t] \odot \mathbf{h}[t-1], \\
\text{output gate}: &\ \sigma_o[t] = \sigma\left(\mathbf{W}_o\mathbf{x}[t] + \mathbf{R}_o\mathbf{y}[t-1] + \mathbf{b}_o\right), \\
\text{output}: &\ \mathbf{y}[t] = \sigma_o[t] \odot g_2(\mathbf{h}[t]).
\end{aligned} \tag{12}$$

$\mathbf{x}[t]$ is the input vector at time t. $\mathbf{W}_f$, $\mathbf{W}_h$, $\mathbf{W}_u$, and $\mathbf{W}_o$ are rectangular weight matrices, that are applied to the input of the LSTM cell. $\mathbf{R}_f$, $\mathbf{R}_h$, $\mathbf{R}_u$, and $\mathbf{R}_o$ are square matrices that define the weights of the recurrent connections, while $\mathbf{b}_f$, $\mathbf{b}_h$, $\mathbf{b}_u$, and $\mathbf{b}_o$ are bias vectors. The function $\sigma(\cdot)$ is a sigmoid [2], while $g_1(\cdot)$ and $g_2(\cdot)$ are pointwise non-linear activation functions, usually implemented as hyperbolic tangents that squash the values in $[-1, 1]$. Finally, $\odot$ is the entrywise multiplication between two vectors (Hadamard product).

Each gate in the cell has a specific and unique functionality. The *forget gate* σ_f decides what information should be discarded from the previous cell state $\mathbf{h}[t-1]$. The *input gate* σ_u operates on the previous state $\mathbf{h}[t-1]$, after having been modified by the forget gate, and it decides how much the new state $\mathbf{h}[t]$ should be updated with a new candidate $\tilde{\mathbf{h}}[t]$. To produce the output $\mathbf{y}[t]$, first the cell filters its current state with a nonlinearity $g_2(\cdot)$. Then, the *output gate* σ_o selects the part of the state to be returned as output. Each gate depends on the current external input $\mathbf{x}[t]$ and the previous cells output $\mathbf{y}[t-1]$.

As we can see from the Fig. 3 and from the forward-step equations, when $\sigma_f = \mathbf{1}$ and $\sigma_u = \mathbf{0}$, the current state of a cell is transferred to the next time interval exactly as it is. By referring back to Eq. 10, it is possible to observe that in LSTM the issue of vanishing gradient does not occur, due to the absence of nonlinear transfer functions applied to the cell state. Since in this case the transfer function $f(\cdot)$ in Eq. 10 applied to the internal states is an identity function, the contribution from past states remains unchanged over time. However, in practice, the update and forget gates are never completely open or closed due to the functional form of the sigmoid, which saturates only for infinitely large values. As a result, even if long term memory in LSTM is greatly enhanced with respect to ERNN architectures, the content of the cell cannot be kept completely unchanged over time.

[2] the logistic sigmoid is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$

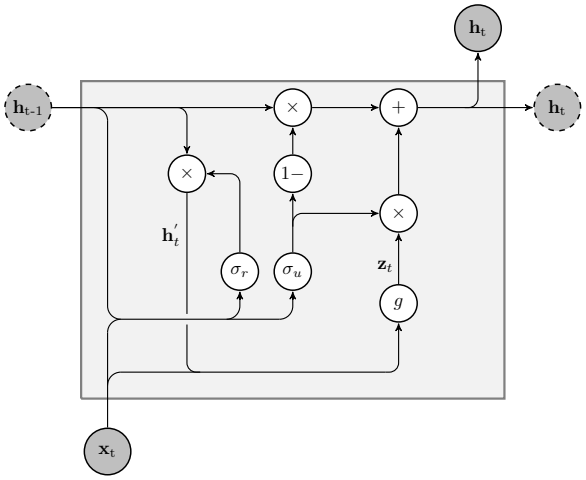


Figure 4: Illustration of a recurrent unit in the GRU architecture. Dark gray circles with a solid line are the variables whose content is exchanged with the input and output of the network. Dark gray circles with a dashed line represent the internal state variables, whose content is exchanged within the cells of the hidden layer. The operator g is a non-linear transformation, usually implemented as a hyperbolic tangent. White circles with '+', '−1' and '×' represent linear operations, while σ_r and σ_u are the sigmoids used in the reset and update gates respectively.

3.3. Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is another notorious gated architecture, originally proposed by Cho et al. [121], which adaptively captures dependencies at different time scales. In GRU, forget and input gates are combined into a single update gate, which adaptively controls how much each hidden unit can remember or forget. The internal state in GRU is always fully exposed in output, due to the lack of a control mechanism, like the output gate in LSTM.

GRU were firstly tested by Cho et al. [121] on a statistical machine translation task and reported mixed results. In an empirical comparison of GRU and LSTM, configured with the same amount of parameters, Chung et al. [122] concluded that on some datasets GRU can outperform LSTM, both in terms of generalization capabilities and in terms of time required to reach convergence and to update parameters. In an extended experimental evaluation, Zaremba [123] employed GRU to (i) compute the digits of the sum or difference of two input numbers, (ii) predict the next character in a synthetic XML dataset and in the large words dataset Penn TreeBank, (iii) predict polyphonic music. The results showed that the GRU outperformed the LSTM on nearly all tasks except language modeling when using a naive initialization. Bianchi et al. [34] compared GRU with other recurrent networks on the prediction of superimposed oscillators. However, to the best of author's knowledge, at the moment there are no researches where the standard GRU architecture has been applied in STLF problems.

A schematic depiction of the GRU cell is reported in Fig. 4. GRU makes use of two gates. The first is the *update gate*, which controls how much the current content of the cell should be updated with the new candidate state. The second is the *reset gate* that, if closed (value near to 0), can effectively reset the memory of the cell and make the unit act as if the next processed input was the first in the sequence. The state equations of the GRU are the following:

$$\begin{aligned}
\text{reset gate}: \quad & \mathbf{r}[t] = \sigma\left(\mathbf{W}_r \mathbf{h}[t-1] + \mathbf{R}_r \mathbf{x}[t] + \mathbf{b}_r\right), \\
\text{current state}: \quad & \mathbf{h}'[t] = \mathbf{h}[t-1] \odot \mathbf{r}[t], \\
\text{candidate state}: \quad & \mathbf{z}[t] = g\left(\mathbf{W}_z \mathbf{h}'[t-1] + \mathbf{R}_z \mathbf{x}[t] + \mathbf{b}_z\right), \\
\text{update gate}: \quad & \mathbf{u}[t] = \sigma\left(\mathbf{W}_u \mathbf{h}[t-1] + \mathbf{R}_u \mathbf{x}[t] + \mathbf{b}_u\right), \\
\text{new state}: \quad & \mathbf{h}[t] = (1 - \mathbf{u}[t]) \odot \mathbf{h}[t-1] + \mathbf{u}[t] \odot \mathbf{z}[t].
\end{aligned} \tag{13}$$

Here, $g(\cdot)$ is a non-linear function usually implemented by a hyperbolic tangent.

In a GRU cell, the number of parameters is larger than in the an ERNN unit, but smaller than in a LSTM cell. The parameters to be learned are the rectangular matrices $\mathbf{W}_r$, $\mathbf{W}_z$, $\mathbf{W}_u$, the square matrices $\mathbf{R}_r$, $\mathbf{R}_z$, $\mathbf{R}_u$, and the bias vectors $\mathbf{b}_r$, $\mathbf{b}_z$, $\mathbf{b}_u$.

4. Other Recurrent Neural Networks Models

In this section we describe two different types of RNNs, which are the Nonlinear AutoRegressive eXogenous inputs neural network (NARX) and the Echo State Network (ESN). Both of them have been largely employed in STLF. These two RNNs differ from the models described in Sec. 3, both in terms of their architecture and in the training procedure, which is not implemented as a BPPT. Therefore, some of the properties and training approaches discussed in Sec. 2 do not hold for these models.

4.1. NARX Network

NARX networks are recurrent dynamic architectures with several hidden layers and they are inspired by discrete-time nonlinear models called Nonlinear AutoRegressive with eXogenous inputs [124]. Differently from other RNNs, the recurrence in the NARX network is given only by the feedback on the output, rather than from the whole internal state.

NARX networks have been employed in many different applicative contexts, to forecast future values of the input signal [125, 126]. Menezes and Barreto [127] showed that NARX networks perform better on predictions involving long-term dependencies. Xie et al. [128] used NARX in conjunction with an input embedded according to Takens method, to predict highly non-linear time series. NARX are also employed as a nonlinear filter, whose target output is trained by using the noise-free version of the input signal [129]. NARX networks have also been adopted by Plett [130] in a gray-box approach for nonlinear system identification.

A NARX network can be implemented with a MultiLayer Perceptron (MLP), where the next value of the output signal $\mathbf{y}[t] \in \mathbb{R}^{N_y}$ is regressed on d_y previous values of the output signal and on d_x previous values of an independent, exogenous input signal $\mathbf{x}[t] \in \mathbb{R}^{N_x}$ [131]. The output equation reads

$$\mathbf{y}[t] = \phi\left(\mathbf{x}[t-d_x], \ldots, \mathbf{x}[t-1], \mathbf{x}[t], \mathbf{y}[t-d_y], \ldots, \mathbf{y}[t-1], \Theta\right), \tag{14}$$

where $\phi(\cdot)$ is the nonlinear mapping function performed by the MLP, Θ are the trainable network parameters, d_x and d_y are the input and the output time delays. Even if the numbers of delays d_x and d_y is a finite (often small) number, it has been proven that NARX networks are at least as powerful as Turing machines, and thus they are universal computation devices [132].

The input $\mathbf{i}[t]$ of the NARX network has $d_x N_x + d_y N_y$ components, which correspond to a set of two Tapped-Delay Lines (TDLs), and it reads

$$\mathbf{i}[t] = \begin{bmatrix} (\mathbf{x}[t-d_x], \ldots, \mathbf{x}[t-1])^T \\ (\mathbf{y}[t-d_y], \ldots, \mathbf{y}[t-1])^T \end{bmatrix}^T. \tag{15}$$

The structure of a MLP network consists of a set of source nodes forming the input layer, $N_l \geq 1$ layers of hidden nodes, and an output layer of nodes. The output of the network is governed by the following difference equations

$$\mathbf{h}_1[t] = f\left(\mathbf{i}[t], \theta_i\right), \tag{16}$$

$$\mathbf{h}_l[t] = f\left(\mathbf{h}_{l-1}[t-1], \theta_{h_l}\right), \tag{17}$$

$$\mathbf{y}[t] = g\left(\mathbf{h}_{N_l}[t-1], \theta_o\right), \tag{18}$$

where $\mathbf{h}_l[t] \in \mathbb{R}^{N_{h_l}}$ is the output of the l^{th} hidden layer at time t, $g(\cdot)$ is a linear function and $f(\cdot)$ is the transfer function of the neuron, usually implemented as a sigmoid or *tanh* function.

The weights of the neurons connections are defined by the parameters $\Theta = \left\{\theta_i, \theta_o, \theta_{h_1}, \ldots, \theta_{h_{N_l}}\right\}$. In particular, $\theta_i = \left\{\mathbf{W}_i^{h_1} \in \mathbb{R}^{d_x N_x + d_y N_y \times N_{h_1}}, \mathbf{b}_{h_1} \in \mathbb{R}^{N_{h_1}}\right\}$ are the parameters that determine the weights in the input layer, $\theta_o = \left\{\mathbf{W}_{h_{N_l}}^o \in \mathbb{R}^{N_{N_l} \times N_y}, \mathbf{b}_o \in \mathbb{R}^{N_y}\right\}$ are the parameters of the output layer and $\theta_{h_l} =$

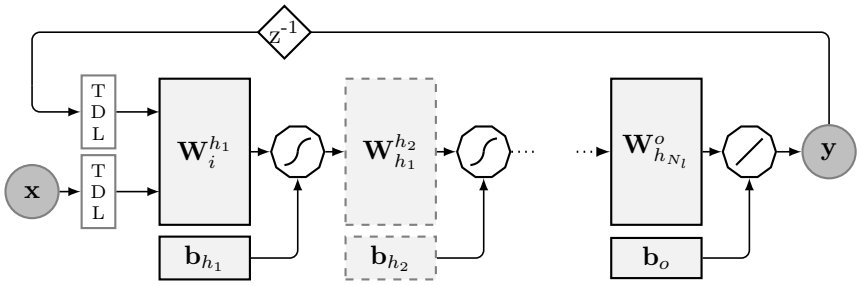
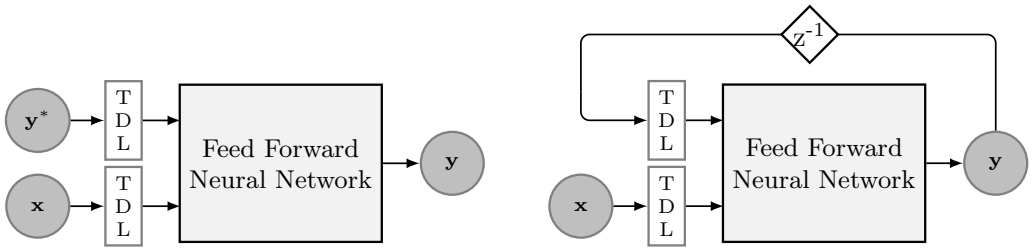


Figure 5: Architecture of the NARX network. Circles represent input $\mathbf{x}$ and output $\mathbf{y}$, respectively. The two TDL blocks are the tapped-delay lines. The solid squares $\mathbf{W}_i^{h_1}$, $\mathbf{W}_{h_{N_l}}^{o}$, $\mathbf{b}_i$, and $\mathbf{b}_o$ are the weight matrices and the bias relative to the input and the output respectively. The dashed squares are the weight matrices and the biases relative to the N_l hidden layers – in the figure, we report $\mathbf{W}_{h_1}^{h_2}$ and $\mathbf{b}_{h_2}$, relative to the first hidden layer. The polygon with the sigmoid symbol represents the nonlinear transfer function of the neurons and the one with the oblique line is a linear operation. Finally, z^{-1} is the backshift/lag operator.

$\left\{\mathbf{W}_{h_{l-1}}^{h_l} \in \mathbb{R}^{N_{h_{l-1}} \times N_{h_l}}, \mathbf{b}_{h_l} \in \mathbb{R}^{N_{h_l}}\right\}$ are the parameters of the l^{th} hidden layer. A schematic depiction of a NARX network is reported in Fig. 5.

Due to the architecture of the network, it is possible to exploit a particular strategy to learn the parameters Θ. Specifically, during the training phase the time series relative to the desired output $\mathbf{y}^*$ is fed into the network along with the input time series $\mathbf{x}$. At this stage, the output feedback is disconnected and the network has a purely feed-forward architecture, whose parameters can be trained with one of the several, well-established standard backpropagation techniques. Notice that this operation is not possible in other recurrent networks such as ERNN, since the state of the hidden layer depends on the previous hidden state, whose ideal value is not retrievable from the training set. Once the training stage is over, the teacher signal of the desired output is disconnected and is replaced with the feedback of the predicted output $\mathbf{y}$ computed by the network. The procedure is depicted in Fig. 6.



(a) Training mode (b) Operational mode

Figure 6: During the training, the desired input $\mathbf{y}^*$ is fed directly to the network. Once the network parameters have been optimized, the teacher signal is removed and the output $\mathbf{y}$ produced by the network is connected to the input with a feedback loop.

Similar to what discussed in Sec. 2.2 for the previous RNN architectures, the loss function employed in the gradient descent is defined as

$$L(\mathbf{x}, \mathbf{y}^*; \Theta) = \text{MSE}(\mathbf{y}, \mathbf{y}^*) + \lambda_2 \|\Theta\|_2, \tag{19}$$

where MSE is the error term defined in Eq. 2 and λ_2 is the hyperparameter that weights the importance of the L_2 regularization term in the loss function. Due to the initial transient phase of the network, when the estimated output $\mathbf{y}$ is initially fed back as network input, the first initial outputs are discarded.

Even if it reduces to a feed-forward network in the training phase, NARX network is not immune to the problem of vanishing and exploding gradients. This can be seen by looking at the Jacobian $\mathbf{J_h}(t, n)$ of the state-space map at time t expanded for n time step. In order to guarantee network stability, the Jacobian must have all of its eigenvalues inside the unit circle at each time step. However, this results in $\lim_{n\to\infty} \mathbf{J_h}(t, n) = 0$, which implies that NARX networks suffer from vanishing gradients, like the other RNNs [133].

4.2. Echo State Network

While most hard computing approaches and ANNs demand long training procedures to tune the parameters through an optimization algorithm [134], recently proposed architectures such as Extreme Learning Machines [135, 136] and ESNs are characterized by a very fast learning procedure, which usually consists in solving a convex optimization problem. ESNs, along with Liquid State Machines [137], belong to the class of computational dynamical systems implemented according to the so-called *reservoir computing* framework [71].

ESN have been applied in a variety of different contexts, such as static classification [138], speech recognition [139], intrusion detection [140], adaptive control [141], detrending of nonstationary time series [142], harmonic distortion measurements [143] and, in general, for modeling of various kinds of non-linear dynamical systems [144].

ESNs have been extensively employed to forecast real valued time series. Niu et al. [145] trained an ESN to perform multivariate time series prediction by applying a Bayesian regularization technique to the reservoir and by pruning redundant connections from the reservoir to avoid overfitting. Superior prediction capabilities have been achieved by projecting the high-dimensional output of the ESN recurrent layer into a suitable subspace of reduced dimension [146]. An important context of application with real valued time series is the prediction of telephonic or electricity load, usually performed 1-hour and a 24-hours ahead [5, 10, 9, 147, 8]. Deihimi et al. [10] and Peng et al. [6] decomposed the time series in wavelet components, which are predicted separately using distinct ESN and ARIMA models, whose outputs are combined to produce the final result. Important results have been achieved in the prediction of chaotic time series by Li et al. [148]. They proposed an alternative to the Bayesian regression for estimating the regularization parameter and a Laplacian likelihood function, more robust to noise and outliers than a Gaussian likelihood. Jaeger and Haas [149] applied an ESN-based predictor on both benchmark and real dataset, highlighting the capability of these networks to learn amazingly accurate models to forecast a chaotic process from almost noise-free training data.

An ESN consists of a large, sparsely connected, untrained recurrent layer of nonlinear units and a linear, memory-less read-out layer, which is trained according to the task that the ESN is demanded to solve. A visual representation of an ESN is shown in Fig. 7

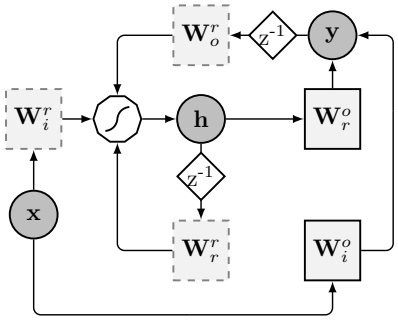


Figure 7: Schematic depiction of the ESN architecture. The circles represent input $\mathbf{x}$, state $\mathbf{h}$, and output $\mathbf{y}$, respectively. Solid squares $\mathbf{W}_r^o$ and $\mathbf{W}_i^o$, are the trainable matrices of the read-out, while dashed squares, $\mathbf{W}_r^r$, $\mathbf{W}_o^r$, and $\mathbf{W}_i^r$, are randomly initialized matrices. The polygon represents the non-linear transformation performed by neurons and z^{-1} is the unit delay operator.

The difference equations describing the ESN state-update and output are, respectively, defined as follows:

$$\mathbf{h}[t] = f(\mathbf{W}_r^r \mathbf{h}[t-1] + \mathbf{W}_i^r \mathbf{x}[t] + \mathbf{W}_o^r \mathbf{y}[t-1] + \epsilon), \tag{20}$$

$$\mathbf{y}[t] = g(\mathbf{W}_i^o \mathbf{x}[t] + \mathbf{W}_r^o \mathbf{h}[t]), \tag{21}$$

where ϵ is a small noise term. The reservoir contains N_h neurons whose transfer/activation function $f(\cdot)$ is typically implemented by a hyperbolic tangent. The readout instead, is implemented usually by a linear function $g(\cdot)$. At time instant t, the network is driven by the input signal $\mathbf{x}[t] \in \mathbb{R}^{N_i}$ and produces the output $\mathbf{y}[k] \in \mathbb{R}^{N_o}$, N_i and N_o being the dimensionality of input and output, respectively. The vector $\mathbf{h}[t]$ has N_h components and it describes the ESN (instantaneous) state. The weight matrices $\mathbf{W}_r^r \in \mathbb{R}^{N_r \times N_r}$ (reservoir connections), $\mathbf{W}_i^r \in \mathbb{R}^{N_i \times N_r}$ (input-to-reservoir), and $\mathbf{W}_o^r \in \mathbb{R}^{N_o \times N_r}$ (output-to-reservoir feedback) contain real values in the $[-1, 1]$ interval drawn from a uniform distribution and are left untrained. Alternative options have been explored recently by Rodan and Tiňo [150] and Appeltant et al. [151] to generate the connection weights. The sparsity of the reservoir is controlled by a hyperparameter R_c, which determines the number of nonzero elements in $\mathbf{W}_r^r$. According to the ESN theory, the reservoir $\mathbf{W}_r^r$ must satisfy the so-called "echo state property" (ESP) [71]. This means that the effect of a given input on the state of the reservoir must vanish in a finite number of time-instants. A widely used rule-of-thumb to obtain this property suggests to rescale the matrix $\mathbf{W}_r^r$ in order to have $\rho(\mathbf{W}_r^r) < 1$, where $\rho(\cdot)$ denotes the spectral radius. However, several theoretical approaches have been proposed in the literature to tune ρ more accurately, depending on the problem at hand [152, 97, 153, 154].

On the other hand, the weight matrices $\mathbf{W}_i^o$ and $\mathbf{W}_r^o$ are optimized for the target task. To determine them, let us consider the training sequence of T_{tr} desired input-outputs pairs given by:

$$(\mathbf{x}[1], y^*[1]) \ldots, (\mathbf{x}[T_{\text{tr}}], y[T_{\text{tr}}]), \tag{22}$$

where T_{tr} is the length of the training sequence. In the initial phase of training, called *state harvesting*, the inputs are fed to the reservoir, producing a sequence of internal states $\mathbf{h}[1], \ldots, \mathbf{h}[T_{\text{tr}}]$, as defined in Eq. (20). The states are stacked in a matrix $\mathbf{S} \in \mathbb{R}^{T_{\text{tr}} \times N_i + N_r}$ and the desired outputs in a vector $\mathbf{y}^* \in \mathbb{R}^{T_{\text{tr}}}$:

$$\mathbf{S} = \begin{bmatrix} \mathbf{x}^T[1],\ \mathbf{h}^T[1] \\ \vdots \\ \mathbf{x}^T[T_{\text{tr}}],\ \mathbf{h}^T[T_{\text{tr}}] \end{bmatrix}, \tag{23}$$

$$\mathbf{y}^* = \begin{bmatrix} \mathbf{y}^*[1] \\ \vdots \\ \mathbf{y}^*[T_{\text{tr}}] \end{bmatrix}. \tag{24}$$

The initial rows in $\mathbf{S}$ (and $\mathbf{y}^*$) are discarded, since they refer to a transient phase in the ESN's behavior.

The training of the readout consists in learning the weights in $\mathbf{W}_i^o$ and $\mathbf{W}_r^o$ so that the output of the ESN matches the desired output $\mathbf{y}^*$. This procedure is termed *teacher forcing* and can be accomplished by solving a convex optimization problem, for which several closed form solution exist in the literature. The standard approach, originally proposed by Jaeger [57], consists in applying a least-square regression, defined by the following regularized least-square problem:

$$\boldsymbol{W}_{\text{ls}}^* = \underset{\boldsymbol{W} \in \mathbb{R}^{N_i + N_h}}{\arg\min} \frac{1}{2} \|\mathbf{S}\boldsymbol{W} - \mathbf{y}^*\|^2 + \frac{\lambda_2}{2} \|\boldsymbol{W}\|_2^2, \tag{25}$$

where $\boldsymbol{W} = [\mathbf{W}_i^o, \mathbf{W}_r^o]^T$ and $\lambda_2 \in \mathbb{R}^+$ is the L_2 regularization factor.

A solution to problem (25) can be expressed in closed form as

$$\boldsymbol{W}_{\text{ls}}^* = \left(\mathbf{S}^T\mathbf{S} + \lambda_2 \mathbf{I}\right)^{-1} \mathbf{S}^T \mathbf{y}^*, \tag{26}$$

which can be solved by computing the Moore-Penrose pseudo-inverse. Whenever $N_h + N_i > T_{\text{tr}}$, Eq. (26) can be computed more efficiently by rewriting it as

$$\boldsymbol{W}_{\text{ls}}^* = \mathbf{S}^T \left(\mathbf{S}\mathbf{S}^T + \lambda_2 \mathbf{I}\right)^{-1} \mathbf{y}^*. \tag{27}$$

5. Synthetic time series

We consider three different synthetically generated time series in order to provide controlled and easily replicable benchmarks for the architectures under analysis. The three forecasting exercises that we study have a different level of difficulty, given by the nature of the signal and the complexity of the task to be solved by the RNN. In order to obtain a prediction problem that is not too simple, it is reasonable to select as forecast horizon a time interval t_f that guarantees the measurements in the time series to become decorrelated. Hence, we consider the first zero of the autocorrelation function of the time series. Alternatively, the first minimum of the average mutual information [155] or of the correlation sum [156] could be chosen to select a t_f where the signal shows a more-general form of independence. All the time series introduced in the following consist of 15.000 time steps. We use the first 60% of the time series as training set, to learn the parameters of the RNN models. The next 20% of the data are used as validation set and the prediction accuracy achieved by the RNNs on this second dataset is used to tune the hyperparameters of the models. The final model performance is evaluated on a test set, corresponding to the last 20% of the values in the time series.

Mackey-Glass time series. The Mackey-Glass (MG) system is commonly used as benchmark for prediction of chaotic time series. The input signal is generated from the MG time-delay differential system, described by the following equation:

$$\frac{dx}{dt} = \frac{\alpha x(t-\tau_{\mathrm{MG}})}{1+x(t-\tau_{\mathrm{MG}})^{10}} - \beta x(t). \tag{28}$$

For this prediction task, we set $\tau_{\mathrm{MG}} = 17, \alpha = 0.2, \beta = 0.1$, initial condition $x(0) = 1.2$, 0.1 as integration step for (28) and the forecast horizon $t_f = 12$.

NARMA signal. The Non-Linear Auto-Regressive Moving Average (NARMA) task, originally proposed by Jaeger [157], consists in modeling the output of the following r-order system:

$$y(t+1) = 0.3y(t) + 0.05y(t)\left[\sum_{i=0}^{r} y(t-i)\right] + 1.5x(t-r)x(t) + 0.1. \tag{29}$$

The input to the system $x(t)$ is uniform random noise in [0, 1], and the model is trained to reproduce $y(t+1)$. The NARMA task is known to require a memory of at least r past time-steps, since the output is determined by input and outputs from the last r time-steps. For this prediction task we set $r = 10$ and the forecast step $t_f = 1$ in our experiments.

Multiple superimposed oscillator. The prediction of a sinusoidal signal is a relatively simple task, which demands a minimum amount of memory to determine the next network output. However, superimposed sine waves with incommensurable frequencies are extremely difficult to predict, since the periodicity of the resulting signal is extremely long. The time series we consider is the Multiple Superimposed Oscillator (MSO) introduced by Jaeger and Haas [149], and it is defined as

$$y(t) = sin(0.2t) + sin(0.311t) + sin(0.42t) + sin(0.51t). \tag{30}$$

This academic, yet important task, is particularly useful to test the memory capacity of a recurrent neural network and has been studied in detail by Xue et al. [158] in a dedicated work. Indeed, to accurately predict the unseen values of the time series, the network requires a large amount of memory to simultaneously implement multiple decoupled internal dynamics [159]. For this last prediction task, we chose a forecast step $t_f = 10$.

6. Real-world load time series

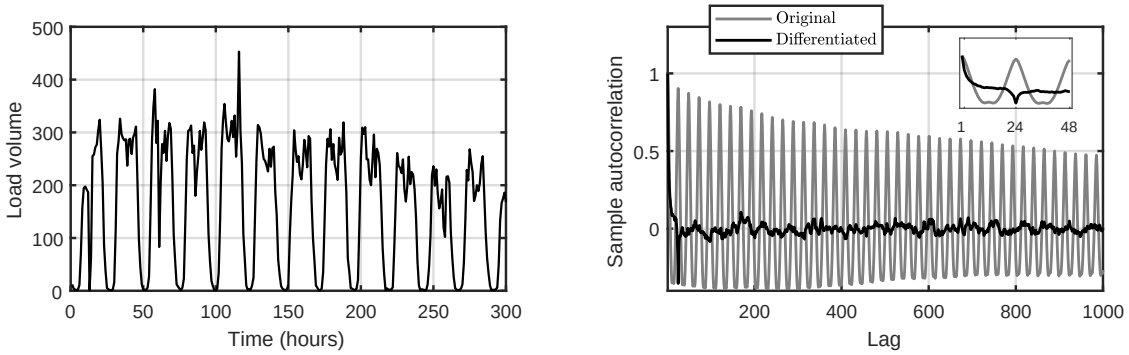
In this section, we present three different real-world dataset, where the time series to be predicted contain measurements of electricity and telephonic activity load. Two of the dataset contain exogenous variables, which are used to provide additional context information to support the prediction task. For each dataset, we perform a pre-analysis to study the nature of the time series and to find the most suitable data preprocessing. In fact, forecast accuracy in several prediction models, among which neural networks, can be considerably improved by applying a meaningful preprocessing [160].

6.1. Orange dataset – telephonic activity load

The first real-world dataset that we analyze is relative to the load of phone calls registered over a mobile network. Data come from the Orange telephone dataset [161], published in the Data for Development (D4D) challenge [162]. D4D is a collection of call data records, containing anonymized events of Orange's mobile phone users in Ivory Coast, in a period spanning from December 1, 2011 to April 28, 2012. More detailed information on the data are available in Ref. [163]. The time series we consider are relative to antenna-to-antenna traffic. In particular, we selected a specific antenna, retrieved all the records in the dataset relative to the telephone activity issued each hour in the area covered by the antenna and generated 6 time series:

- `ts1`: number of incoming calls in the area covered by the antenna;
- `ts2`: volume in minutes of the incoming calls in the area covered by the antenna;
- `ts3`: number of outgoing calls in the area covered by the antenna;
- `ts4`: volume in minutes of the outgoing calls in the area covered by the antenna;
- `ts5`: hour when the telephonic activity is registered;
- `ts6`: day when the telephonic activity is registered.

In this work, we focus on predicting the volume (in minutes) of the incoming calls in `ts1` of the next day. Due to the hourly resolution of the data, the STFL problem consists of a 24 step-ahead prediction. The profile of `ts1` for 300 hours is depicted in Fig. 8(a). The remaining time series are treated as exogenous



(a) Load profile (b) Autocorrelation functions

Figure 8: In (a), the load profile of `ts1`, the incoming calls volume, for 300 time intervals (hours). In (b), the autocorrelation functions of the time series `ts1` before (gray line) and after (black line) a seasonal differentiation. The original time series shows a strong seasonal pattern at lag 24, while after seasonal differencing, the time series does not show any strong correlation or trend.

variables and, according to a common practice in time series forecasting [164], they are fed into the network to provide the model with additional information for improving the prediction of the target time series. Each

time series contain 3336 measurements, hourly sampled. We used the first 70% as training set, the successive 15% as validation set and the remaining 15% as test set. The accuracy of each RNN model is evaluated on this last set.

In each time series there is a (small) fraction of missing values. In fact, if in a given hour no activities are registered in the area covered by the considered antenna, the relative entries do not appear in the database. As we require the target time series and the exogenous ones to have same lengths and to contain a value in each time interval, we inserted an entry with value "0" in the dataset to fill the missing values. Another issue is the presence of corrupted data, marked by a "-1" in the dataset, which are relative to periods when the telephone activity is not registered correctly. To address this problem, we followed the procedure described by Shen and Huang [165] and we replaced the corrupted entries with the average value of the corresponding periods (same weekday and hour) from the two adjacent weeks. Contrarily to some other works on STLF [166, 7, 167], we decided to not discard outliers, such as holidays or days with an anomalous number of calls, nor we modeled them as separate variables.

As next step in our pre-analysis, we identify the main seasonality in the data. We analyze `ts1`, but similar considerations hold also for the remaining time series. Through frequency analysis and by inspecting the autocorrelation function, depicted as a gray line in Fig. 8(b), it emerges a strong seasonal pattern every 24 hours. As expected, data experience regular and predictable daily changes, due to the nature of the telephonic traffic. This cycle represents the main seasonality and we filter it out by applying a seasonal differencing with lag 24. In this way, the RNNs focus on learning to predict the series of changes in each seasonal cycle. The practice of removing the seasonal effect from the time series, demonstrated to improve the prediction accuracy of models based on neural networks [168, 169]. The black line in Fig. 8(b) depicts the autocorrelation of the time series after seasonal differentiation. Except from the high anticorrelation at lag 24, introduced by the differentiation, the time series appears to be uncorrelated elsewhere and, therefore, we can exclude the presence of a second, less obvious seasonality.

Due to the nature of the seasonality in the data, we expect a strong relationship between the time series of the loads (`ts1` - `ts4`) and `ts5`, which is relative to the hour of the day. On the other hand, we envisage a lower dependency of the loads with `ts6`, the time series of the week days, since we did not notice the presence of a second seasonal cycle after the differentiation at lag 24. To confirm our hypothesis, we computed the mutual information between the time series, which are reported in the Hinton diagram in Fig. 9. The size

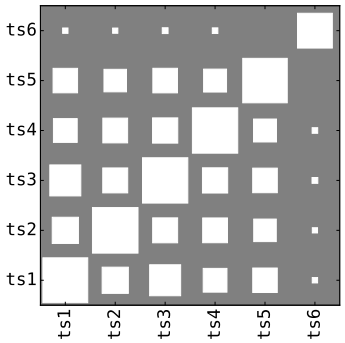
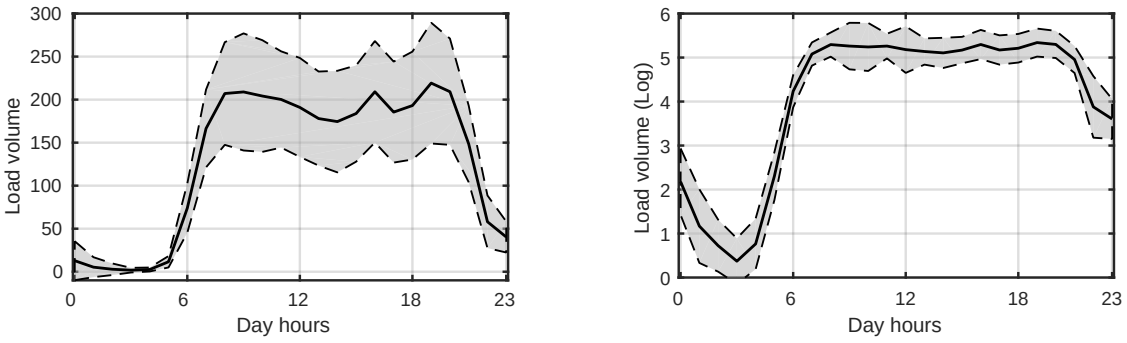


Figure 9: Hinton diagram of the mutual information between the time series in the Orange dataset. The size of each block is proportional to the degree of mutual information among the time series. The measurements indicates a strong relationship between the load time series and the daily hours (`ts5`), while the dependency with the day of the week (`ts6`) is low.

of the blocks is proportional to the degree of mutual information among the time series. Due to absence of strong relationships, we decided to discard `ts6` to reduce the complexity of the model by excluding a variable with potentially low impact in the prediction task. We also discarded `ts5` because the presence of the cyclic daily pattern is already accounted by doing the seasonal differencing at lag 24. Therefore, there is not need to provide daily hours as an additional exogenous input.

Beside differentiation, a common practice in STLF is to apply some form of normalization to the data. We applied a standardization (z-score), but rescaling into the interval $[-1, 1]$ or $[0, 1]$ are other viable options. Additionally, a nonlinear transformation of the data by means of a non-linear function (e.g., square-root or logarithm) can remove some kinds of trend and stabilize the variance in the data, without altering too

much their underlying structure [170, 7, 166]. In particular, a log-transform is suitable for a set of random variables characterized by a high variability in their statistical dispersion (heteroscedasticity), or for a process whose fluctuation of the variance is larger than the fluctuation of the mean (overdispersion). To check those properties, we analyze the mean and the variance of the telephonic traffic within the main seasonal cycle across the whole dataset. The solid black line in Fig. 10(a), represents the mean load of `ts1`, while the



(a) Raw data (b) Log-transformed data

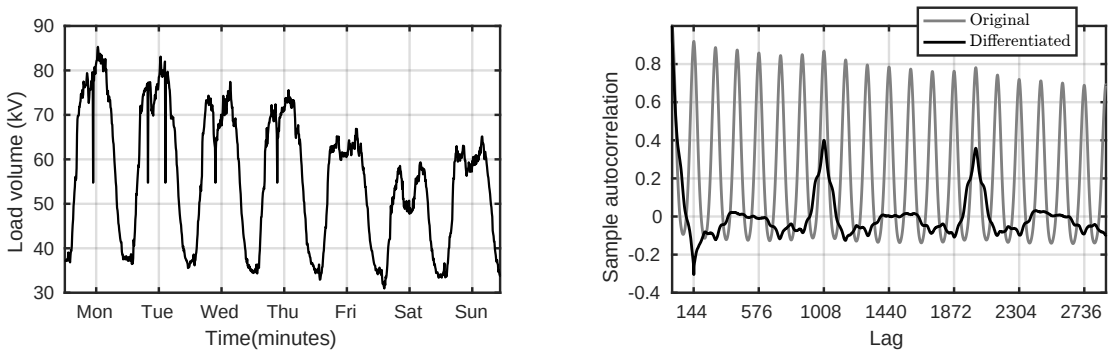
Figure 10: Average weekly load (solid black line) and the standard deviation (shaded gray area) of the telephonic activity in the whole dataset.

shaded gray area illustrates the variance. As we can see, the data are not characterized by overdispersion, since the fluctuations of the mean are greater than the ones of the variance. However, we notice the presence of heteroscedasticity, since the amount of variance changes in different hours of the day. In fact, the central hours where the amount of telephonic activity is higher, are characterized by a greater standard deviation in the load. In Fig. 10(b), we observe that by applying a log-transform we significantly reduce the amount of variance in the periods characterized by a larger traffic load. However, after the log-transformation the mean value of the load become more flattened and the variance relative to periods with lower telephonic activity is enhanced. This could cause issues during the training of the RNN, hence in the experiments we evaluate the prediction accuracy both with and without applying the log-transformation to the data.

Preprocessing transformations are applied in this order: (i) log-transform, (ii) seasonal differencing at lag 24, (iii) standardization. Each preprocessing operation is successively reversed to evaluate the forecast produced by each RNN.

6.2. ACEA dataset – electricity load

The second time series we analyze is relative to the electricity consumption registered by ACEA (Azienda Comunale Energia e Ambiente), the company which provides the electricity to Rome and some neighbouring regions. The ACEA power grid in Rome consists of 10.490 km of medium voltage lines, while the low voltage section covers 11.120 km. The distribution network is constituted of backbones of uniform section, exerting radially and with the possibility of counter-supply if a branch is out of order. Each backbone is fed by two distinct primary stations and each half-line is protected against faults through the breakers. Additional details can be found in Ref. [171]. The time series we consider concerns the amount of supplied electricity, measured on a medium voltage feeder from the distribution network of Rome. Data are collected every 10 minutes for 954 days of activity (almost 3 years), spanning from 2009 to 2011, for a total of 137444 measurements. Also in this case, we train the RNNs to predict the electricity load 24h ahead, which corresponds to 144 time step ahead prediction. For this forecast task we do not provide any exogenous time series to the RNNs. In the hyperparameter optimization, we use the load relative to the first 3 months as training set and the load of the 4^{th} month as validation set. Once the best hyperparameter configuration is identified, we fine-tune each RNN on the first 4 months and we use the 5^{th} month as test set to evaluate and to compare the accuracy of each network.



(a) Load profile

(b) Autocorrelation functions

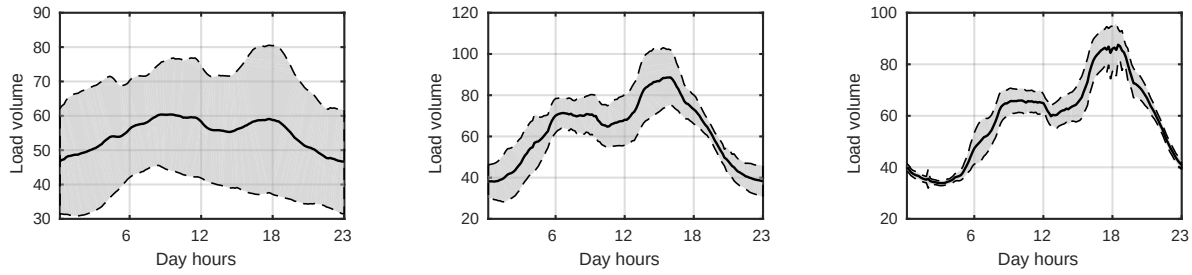
Figure 11: In (a), the load profile in kiloVolts (kV) of the electricity consumption registered over one week. The sampling time is 10 minutes. In (b), the autocorrelation functions of the ACEA time series before (gray line) and after (black line) a seasonal differentiation at lag 144. The original time series shows a strong seasonal pattern at lag 144, which corresponds to a daily cycle. After seasonal differencing, a previously hidden pattern is revealed at lag 1008, which corresponds to a weekly cycle.

A profile of the electric consumption over one week (1008 measurements), is depicted in Fig. 11(a).

In the ACEA time series there are no missing values, but 742 measurements (which represent 0.54% of the whole dataset) are corrupted. The consumption profile is more irregular in this time series, with respect to the telephonic data from the Orange dataset. Therefore, rather than replacing the corrupted values with an average load, we used a form of imputation with a less strong bias. Specifically, we first fit a cubic spline to the whole dataset and then we replaced the corrupted entries with the corresponding values from the fitted spline. In this way, the imputation better accounts for the local variations of the load.

Also in this case, we perform a preemptive analysis in order to understand the nature of the seasonality, to detect the presence of hidden cyclic patterns, and to evaluate the amount of variance in the time series. By computing the autocorrelation function up to a sufficient number of lags, depicted as a gray line in Fig. 11(b), it emerges a strong seasonality pattern every 144 time intervals. As expected, this corresponds exactly to the number of measurements in one day. By differencing the time series at lag 144, we remove the main seasonal pattern and the trend. Also in this case, the negative peak at lag 144 is introduced by the differentiation. If we observe the autocorrelation plot of the time series after seasonal differencing (black line in 11(b)), a second strong correlation appears each 1008 lags. This second seasonal pattern represents a weekly cycle, that was not clearly visible before the differentiation. Due to the long periodicity of the time cycle, to account this second seasonality a predictive model would require a large amount of memory to store information for a longer time interval. While a second differentiation can remove this second seasonal pattern, we would have to discard the values relative to the last week of measurements. Most importantly, the models we train could not learn the similarities in consecutive days at a particular time, since they would be trained on the residuals of the load at the same time and day in two consecutive weeks. Therefore, we decided to apply only the seasonal differentiation at lag 144.

To study the variance in the time series, we consider the average daily load over the main seasonal cycle of 144 time intervals. As we can see from Fig. 12(a), data appear to be affected by overdispersion, as the standard deviation (gray shaded areas) fluctuates more than the mean. Furthermore, the mean load value (black solid line) seems to not change much across the different hours, while it is reasonable to expect significant differences in the load between night and day. However, we remind that the Acea time series spans a long time lapse (almost 3 years) and that the electric consumption is highly related to external factors such as temperature, daylight saving time, holidays and other seasonal events that change over time. Therefore, in different periods the load profile may vary significantly. For example, in Fig. 12(b) we report the load profile relative to the month of January, when temperatures are lower and there is a high consumption of electricity, also in the evening, due to the usage of heating. In June instead (Fig. 12(c)), the overall electricity consumption is lower and mainly concentrated on the central hours of the day. Also,



(a) Whole dataset (b) January (c) June

Figure 12: In (a) we report the mean load (black line) and the standard deviation (gray area) of the electricity consumption in a week, accounting the measurements from all the dataset. In (b) and (c), the measurements are relative only to one month of activity, which are January and June respectively.

it is possible to notice that the load profile is shifted due to the daylight saving time. As we can see, the daily averages within a single month are characterized by a much lower standard deviation (especially in the summer months, with lower overall load consumption) and the mean consumption is less flat. Henceforth, a non-linear transformation for stabilizing the variance is not required and, also in this case, standardization is suitable for normalizing the values in the time series. Since we focus on a short term forecast, having a high variance in loads relative to very distant periods is not an issue, since the model prediction will depends mostly on the most recently seen values.

To summarize, as preprocessing operation we apply: (i) seasonal differencing at lag 144, (ii) standardization. As before, the transformations are reverted to estimate the forecast.

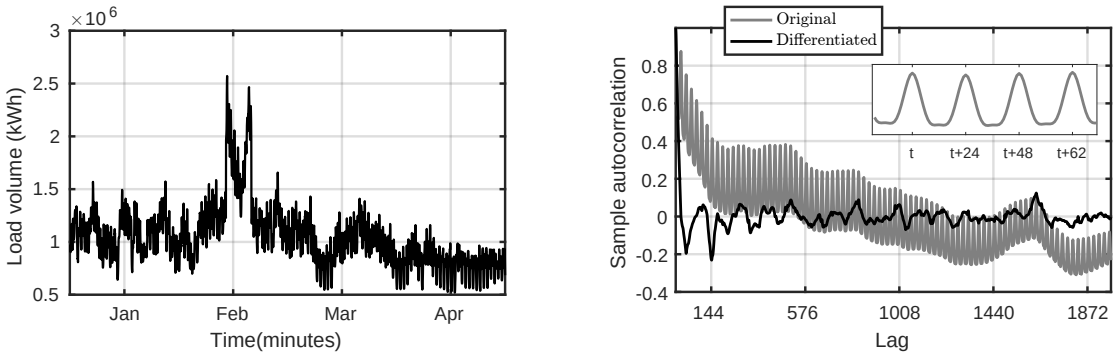
6.3. GEFCom2012 dataset – electricity load

The last real world dataset that we study is the time series of electricity consumption from the Global Energy Forecasting Competition 2012 (GEF-Com2012) [172]. The GEFCom 2012 dataset consists of 4 years (2004 – 2007) of hourly electricity load collected from a US energy supplier. The dataset comprehends time series of consumption measurements, from 20 different feeders in the same geographical area. The values in each time series represent the average hourly load, which varies from 10.000kWh to 200.000kWh. The dataset also includes time series of the temperatures registered in the area where the electricity consumption is measured.

The forecast task that we tackle is the 24 hours ahead prediction of the aggregated electricity consumption, which is the sum of the 20 different load time series in year 2006. The measurements relative to the to first 10 months of the 2006 are used as training set, while the 11^{th} month is used as validation set for guiding the hyperparameters optimization. The time series of the temperature in the area is also provided to the RNNs as an exogenous input. The prediction accuracy of the optimized RNNs is then evaluated on the last month of the 2006. A depiction of the load profile of the aggregated load time series is reported in Fig. 13(a). We can observe a trend in the time series, which indicates a decrement in the energy demand over time. This can be related to climate conditions since, as the temperature becomes warmer during the year, the electricity consumption for the heating decreases.

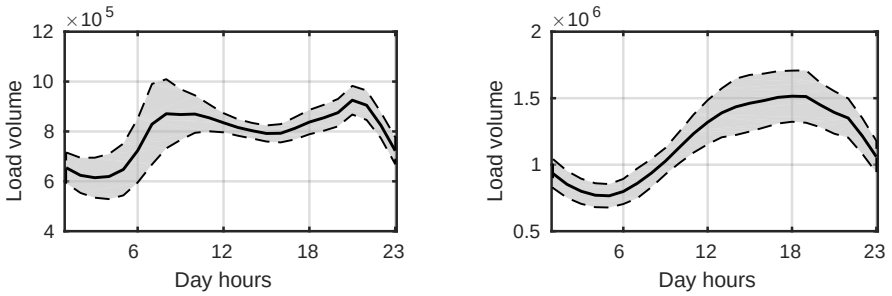
To study the seasonality in the aggregated time series, we evaluate the autocorrelation function, which is depicted as the gray line in Fig. 13(b). From the small subplot in top-right part of the figure, relative to a small segment of the time series, it emerges a strong seasonal pattern every 24 hours. By applying a seasonal differentiation with lag 24 the main seasonal pattern is removed, as we can see from the autocorrelation function of the differentiated time series, depicted as a black line in the figure. After differentiation, the autocorrelation becomes close to zero after the first lags and, therefore, we can exclude the presence of a second, strong seasonal pattern (e.g. a weekly pattern).

Similarly to what we did previously, we analyze the average load of the electricity consumption during one week. As for the ACEA dataset, rather than considering the whole dataset, we analyze separately the



(a) Load profile (b) Autocorrelation functions

Figure 13: In (a), the load profile in kilowatt-hour (kWh) of the aggregated electricity consumption registered in the first 4 months of activity in 2006, from the GEFCom dataset. The sampling time in the time series is 1 hour. In (b), the autocorrelation functions of the GEFCom time series before (gray line) and after (black line) a seasonal differentiation at lag 24. The small subplot on the top-right part of the figure reports a magnified version of the autocorrelation function before differentiation at lag $t = 200$.



(a) January (b) May

Figure 14: In (a) the average load (solid black line) and the standard deviation (shaded gray area) of the electricity consumption during one week, in the month of January. In (b), we report the measurements relative to the month of June.

load in one month of winter and one month in summer. In Fig. 14(a), we report the mean load (black line) and standard deviation (gray area) in January. Fig. 14(b) instead, depicts the measurements for May. It is possible to notice a decrement of the load during the spring period, due to the reduced usage of heating. It is also possible to observe a shift in the consumption profile to later hours in the day, due to the time change. By analyzing the amount of variance and the fluctuations of the mean load, we can exclude the presence of overdispersion and heteroscedasticity phenomena in the data.

To improve the forecasting accuracy of the electricity consumption, a common practice is to provide to the prediction system the time series of the temperature as an exogenous variable. In general, the load and the temperature are highly related, since both in the coldest and in the warmest months electricity demand increases, due to the usage of heating and air conditioning, respectively. However, the relationship between temperature and load cannot be captured by the linear correlation, since the consumption increases both when temperatures are too low or too high. Indeed, the estimated correlation between the aggregated load time series of interest and the time series of the temperature in the area yields only a value of 0.2. However, their relationship is evidenced by computing a 2-dimensional histogram of the two variables, proportional to their estimated joint distribution, which is reported in Fig 15. The V-shape, denotes an increment of the electricity consumption for low and high temperatures with respect to a mean value of about 22° C.

The preprocessing operations we apply on the GEFCom dataset are: (i) seasonal differencing at lag 24,

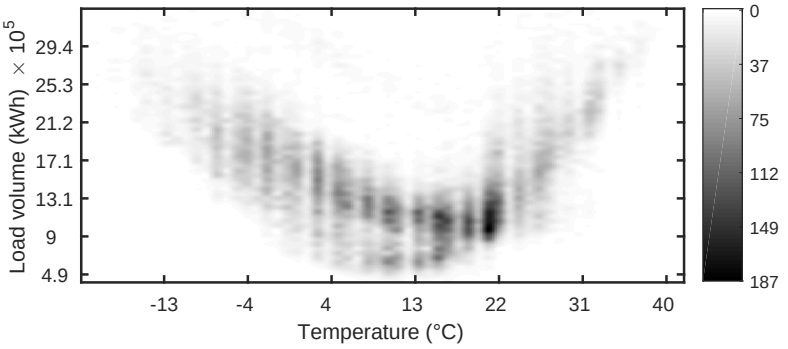


Figure 15: 2-dimensional histogram of the aggregated electricity load and temperature in GEFCom dataset. Darker areas represent more populated bins. The bar on the right indicates the number of elements in each bin. The characteristic V-shape of the resulting pattern is because of the increased use of heating and cooling devices in presence of hot and cold temperatures.

(ii) standardization. Also in this case, these transformations are reverted to estimate the forecast.

7. Experiments

In this section we compare the prediction performance achieved by the network architectures presented in Sec. 3 and 4 on different time series. For each architecture, we describe the validation procedure we follow to tune the hyperparameters and to find an optimal learning strategy for training the weights. During the validation phase, different configurations are randomly selected from admissible intervals and, once the training is over, their optimality is evaluated as the prediction accuracy achieved on the validation set. We opted for a random search as it can find more accurate results than a grid search, when the same number of configurations are evaluated [173]. Once the (sub)optimal configuration is identified, we train each model 10 times on the training and validation data, using random and independent initializations of the network parameters, and we report the highest prediction accuracy obtained on the unseen values of the test set.

To compare the forecast capability of each model, we evaluate the prediction accuracy ψ as $\psi = 1 -$ NRMSE. NRMSE is the Normalized Root Mean Squared Error that reads

$$\mathrm{NRMSE}\left(\mathcal{Y}, \mathcal{Y}^*\right) = \sqrt{\frac{\langle \|\mathcal{Y} - \mathcal{Y}^*\|^2 \rangle}{\langle \|\mathcal{Y} - \langle \mathcal{Y}^* \rangle\|^2 \rangle}}, \tag{31}$$

where $\langle \cdot \rangle$ computes the mean, $\mathcal{Y}$ are the RNN outputs and $\mathcal{Y}^*$ are the ground-truth values.

In the following, we present two types of experiments. The first experiment consists in the prediction of the synthetic time series presented in Sec. 5, commonly considered as benchmarks in forecast applications, and the results are discussed in Sec. 7.2. In the second experiment we forecast the real-world telephonic and electricity load time series, presented in Sec. 6. The results of this second experiment are discussed in Sec. 7.3.

7.1. Experimental settings

ERNN, LSTM and GRU. The three RNNs described in Sec. 3 have been implemented in Python, using Keras library with Theano [174] as backend[3].

To identify an optimal configuration for the specific task at hand, we evaluate for each RNN different values of the hyperparameters and training procedures. The configurations are selected randomly and their

[3]Keras library is available at `https://github.com/fchollet/keras`. Theano library is available at `http://deeplearning.net/software/theano/`

performances are evaluated on the validation set, after having trained the network for 400 epochs. To get rid of the initial transient phase, we drop the first 50 outputs of the network. A total of 500 random configurations for each RNN are evaluated and, once the optimal configuration is found, we compute the prediction accuracy on the test set. In the test phase, each network is trained for 2000 epochs.

The optimization is performed by assigning to each hyperparameter a value uniformly sampled from a given interval, which can be continuous or discrete. The gradient descent strategies are selected from a set of possible alternatives, which are SGD, Nesterov momentum and Adam. For SGD and Nesterov, we anneal the learning rate with a step decay of 10^{-6} in each epoch. The learning rate η is sampled from different intervals, depending on the strategy selected. Specifically, for SGD we set $\eta = 10^c$, with c uniformly sampled in $[-3, -1]$. For Nesterov and Adam, since they benefit from a smaller initial value of the learning rate, we sample c uniformly in $[-4, -2]$. The remaining hyperparameters used in the optimization strategies are kept fixed to their default values (see Sec. 2.3). Regarding the number N_h of hidden units in the recurrent hidden layer, we randomly chose for each architecture four possible configurations that yield an amount of trainable parameters approximately equal to 1800, 3900, 6800, and 10000. This corresponds to $N_h = \{40, 60, 80, 100\}$ in ERNN, $N_h = \{20, 30, 40, 50\}$ in LSTM and $N_h = \{23, 35, 46, 58\}$ in GRU. For each RNNs, N_h is randomly selected from these sets. To deal with the problem of vanishing gradient discussed in Sec. 2.4, we initialize the RNN weights by sampling them from an uniform distribution in $[0, 1]$ and then rescaling their values by $1/\sqrt{N_h}$. For the L_1 and L_2 regularization terms, we sample independently λ_1 and λ_2 from $[0, 0.1]$, an interval containing values commonly assigned to these hyperparameters in RNNs [175]. We apply the same regularization to input, recurrent and output weights. As suggested by Gal and Ghahramani [77], we drop the same input and recurrent connections at each time step in the BPTT, with a dropout probability p_{drop} drawn from $\{0, 0.1, 0.2, 0.3, 0.5\}$, which are commonly used values [176]. If $p_{\text{drop}} \neq 0$, we also apply a L_2 regularization. This combination usually yields a lowest generalization error than dropout alone [177]. Note that another possible approach combines dropout with the max-norm constraint, where the L_2 norm of the weights is clipped whenever it grows beyond a given constant, which, however, introduces another hyperparameter.

For the training we consider the backpropagation through time procedure BPTT(τ_b, τ_f) with $\tau_b = 2\tau_f$. The parameter τ_f is randomly selected from the set $\{10, 15, 20, 25, 30\}$. As we discussed in Sec. 2.1, this procedure differs from both the *true* BPTT and the *epochwise* BPTT [75], which is implemented as default by popular deep learning libraries such as TensorFlow [178].

NARX. This RNN is implemented using the Matlab Neural Network toolbox[4]. We configured NARX network with an equal number of input and output lags on the TDLs ($d_x = d_y$) and with the same number of neurons N_h in each one of the N_l hidden layers. Parameters relative to weight matrices and bias values $\Theta = \left\{\theta, \theta_o, \theta_{h_1}, \ldots, \theta_{h_{N_l}}\right\}$ are trained with a variant of the quasi Newton search, called Levenberg-Marquardt optimization algorithm. This is an algorithm for error backpropagation that provides a good tradeoff between the speed of the Newton algorithm and the stability of the steepest descent method [179]. The loss function to be minimized is defined in Eq. 19.

NARX requires the specification of 5 hyperparameters, which are uniformly drawn from different intervals. Specifically, TDL lags are drawn from $\{2, 3, \ldots, 10\}$; the number of hidden layers N_l is drawn from $\{1, 2, \ldots, 5\}$; the number of neurons N_h in each layer is drawn from $\{5, 6, \ldots, 20\}$; the regularization hyperparameter λ_2 in the loss function is randomly selected from $\{2^{-1}, 2^{-2}, \ldots, 2^{-10}\}$; the initial value η of learning rate is randomly selected from $\{2^{-5}, 2^{-6}, \ldots, 2^{-25}\}$.

A total of 500 random configurations for NARX are evaluated and, for each hyperparameters setting, the network is trained for 1000 epochs in the validation. In the test phase, the network configured with the optimal hyperparameters is trained for 2000 epochs. Also in this case, we discard the first 50 network outputs to get rid of the initial transient phase of the network.

[4] `https://se.mathworks.com/help/nnet/ref/narxnet.html`

ESN. For the ESN, we used a modified version of the Python implementation[5], provided by Løkse et al. [146]. Learning in ESN is fast, as the readout is trained by means of a linear regression. However, the training does not influence the internal dynamics of the random reservoir, which can be controlled only through the ESN hyperparameters. This means that a more accurate (and computationally intensive) search of the optimal hyperparametyers is required with respect to the other RNN architectures. In RNNs, the precise, yet slow gradient-based training procedure is mainly responsible for learning the necessary dynamics and it can compensate a suboptimal choice of the hyperparameters.

Therefore, in the ESN validation phase we evaluate a larger number of configurations (5000), by uniformly drawing 8 different hyperparameters from specific intervals. In particular, the number of neurons in the reservoir, N_h, is drawn from $\{400, 450, \ldots, 900\}$; the reservoir spectral radius, ρ, is drawn in the interval $[0.5, 1.8]$; the reservoir connectivity R_c is drawn from $[0.15, 0.45]$; the noise term ξ in Eq. (20) comes from a Gaussian distribution with zero mean and variance drawn from $[0, 0.1]$; scaling of input signal ω_i and desired response ω_o are drawn from $[0.1, 1]$; scaling of output feedback ω_f is drawn from $[0, 0.5]$; the linear regression regularization parameter λ_2 is drawn from $[0.001, 0.4]$. Also in this case, we discarded the first 50 ESN outputs relative to the initial transient phase.

7.2. Results on synthetic dataset

In Fig. 16 we report the prediction accuracy obtained by the RNNs on the test set of the three synthetic problems. The best configurations of the architectures identified for each task through random search are reported in Tab. 1.

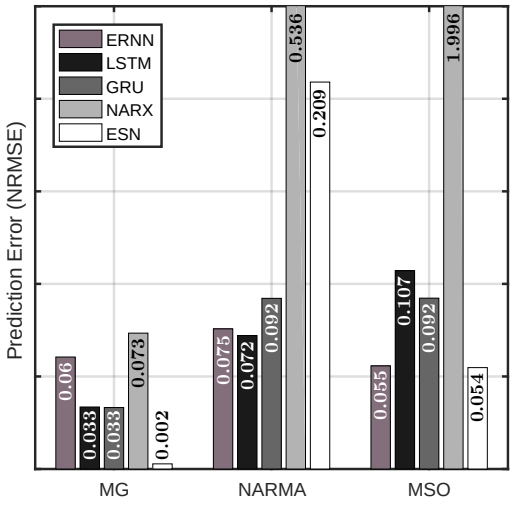


Figure 16: NRMSE values achieved on the test sets by each RNN architecture on the three synthetic prediction tasks.

First of all, we observe that the best performing RNN is different in each task. In the MG task, ESN outperforms the other networks. This result confirms the excellent and well-known capability of the ESN in predicting chaotic time series [180, 149]. In particular, ESN demonstrated to be the most accurate architecture for the prediction of the MG system [181]. The ESN achieves the best results also in the MSO task, immediately followed by ERNN. On the NARMA task instead, ESN performs poorly, while the LSTM is the RNN that predicts the target signal with the highest accuracy.

[5] `https://github.com/siloekse/PythonESN`

Table 1: Optimal RNNs configurations for solving the three synthetic prediction tasks, MG, NARMA and MSO. The acronyms in the table are: N_h – number of nodes in the hidden layer; N_l – number of hidden layers; TDL – number of lags on the tapped delay lines; η – learning rate; λ_1 – L_1 regularization parameter; λ_2 – L_2 regularization parameter; OPT – gradient descent strategy; τ_f – number of new time steps processed before computing the BPTT; τ_b – number of time step the gradient is propagated back in BPTT; p_{drop} – dropout probability; ρ – spectral radius of ESN reservoir; R_c – percentage of sparsity in ESN reservoir; ξ – noise in ESN state update; ω_i, ω_o, ω_f – scaling of input, teacher and feedback weights.

Network	**Task**	**RNN Configuration**							
		N_h		N_l		**TDL**		η	λ_2
Narx	MG	15		2		6		3.8E-6	0.0209
	NARMA	17		2		10		2.4E-4	0.4367
	MSO	12		5		2		0.002	0.446
		τ_b	τ_f	N_h	**OPT**	η	p_{drop}	λ_1	λ_2
ERNN	MG	20	10	80	Adam	0.00026	0	0	0.00037
	NARMA	50	25	80	Nesterov	0.00056	0	0	1E-5
	MSO	50	25	60	Adam	0.00041	0	0	0.00258
		τ_b	τ_f	N_h	**OPT**	η	p_{drop}	λ_1	λ_2
LSTM	MG	50	25	40	Adam	0.00051	0	0	0.00065
	NARMA	40	20	40	Adam	0.00719	0	0	0.00087
	MSO	50	25	20	Adam	0.00091	0	0	0.0012
		τ_b	τ_f	N_h	**OPT**	η	p_{drop}	λ_1	λ_2
GRU	MG	40	20	46	SGD	0.02253	0	0	6.88E-6
	NARMA	40	20	46	Adam	0.00025	0	0	0.00378
	MSO	50	25	35	Adam	0.00333	0	0	0.00126
		N_h	ρ	R_c	ξ	ω_i	ω_o	ω_f	λ_2
ESN	MG	800	1.334	0.234	0.001	0.597	0.969	0.260	0.066
	NARMA	700	0.932	0.322	0.013	0.464	0.115	0.045	0.343
	MSO	600	1.061	0.231	0.002	0.112	0.720	0.002	0.177

In each test, NARX struggles in reaching performance comparable with the other architectures. In particular, in NARMA and MSO task the NRMSE prediction error of NARX is 0.53 and 1.99, respectively (note that we cut the y-axis to better show the remaining bars). Note that, since the NRMSE is normalized by the variance of the target signal, an error greater than 1 means that the performance is worse than a constant predictor, with value equal to the mean of the target signal.

It is also interesting to notice that in MSO, ERNN achieves a prediction accuracy higher than GRU and LSTM. Despite the fact that the MSO task demands a large amount of memory, due to the extremely long periodicity of the target signal, the two gated architectures (LSTM and GRU) are not able to outperform the ERNN. We can also notice that for MSO the optimal number of hidden nodes (N_h) is lower than in the other tasks. A network with a limited complexity is less prone to overfit on the training data, but it is also characterized by an inferior modeling capability. Such a high modeling capability is not needed to solve the MSO task, given that the network manages to learn correctly the frequencies of the superimposed sinusoidal signals.

Finally, we observe that LSTM and GRU performs similarly on the each task, but there is not a clear winner. This finding is in agreement with previous studies, which, after several empirical evaluations, concluded that it is difficult to choose in advance the most suitable gated RNN to solve a specific problem [122].

Regarding the gradient descent strategies used to train the parameters in RNN, LSTM and GRU, we observe in Tab. 1 that Adam is often identified as the optimal strategy. The standard SGD is selected only for GRU in the MG task. This is probably a consequence of the lower convergence rate of the SGD minimization, which struggles to discover a configuration that achieves a good prediction accuracy on the validation set in the limited amount (400) of training epochs. Also, the Nesterov approach seldom results to be as the optimal strategy and a possible explanation is its high sensitivity to the (randomly selected)

learning rate. In fact, if the latter is too high, the gradient may build up too much momentum and bring the weights into a configuration where the loss function is very large. This results in even greater gradient updates, which leads to rough oscillations of the weights that can reach very large values.

From the optimal configurations in Tab. 1, another striking behavior about the optimal regularization procedures emerges. In fact, we observe that in each RNN and for each task, only the L_2 norm of the weights is the optimal regularizer. On the other hand, the parameters λ_1 and p_{drop} relative to the L_1 norm and the dropout are always zero. This indicates that, to successfully solve the synthetic prediction tasks, it is sufficient to train the networks with small weights in order to prevent the overfitting.

Finally, we notice that the best results are often found using network with a high level of complexity, in terms of number of neurons and long windows in BPTT or TDL, for Narx. In fact, in most cases the validation procedure identifies the optimal values for these variables to be close to the upper limit of their admissible intervals. This is somehow expected, since a more complex model can achieve higher modeling capabilities, if equipped with a suitable regularization procedure to prevent overfitting during training. However, the tradeoff in terms of computational resources for training more complex models is often very high and small increments in the performance are obtained at the cost of much longer training times.

7.3. Results on real-world dataset

The highest prediction accuracies obtained by the RNNs on the test set (unseen data) of the real-world load time series, are reported in Fig. 17. As before, in Tab. 2 we report the optimal configuration of each RNN for the different tasks.

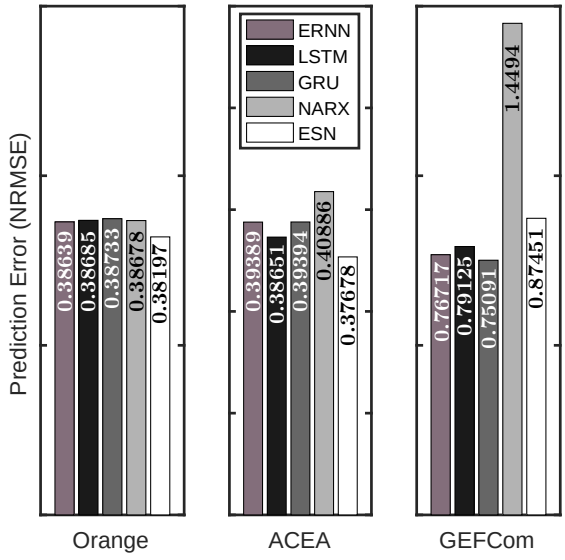


Figure 17: NRMSE values achieved on the test sets by each RNN architecture on the three real-world STLF problems. Note that scales are different for each dataset

Orange. All the RNNs achieve very similar prediction accuracy on this dataset, as it is possible to see from the first bar plot in Fig. 17. In Fig. 18 we report the residuals, depicted as black areas, between the target time series and the forecasts of each RNN. The figure gives immediately a visual quantification of the accuracy, as the larger the black areas, the greater the prediction error in that parts of the time series. In particular, we observe that the values which the RNNs fail to predict are often relative to the same interval. Those values represent fluctuations that are particularly hard to forecast, since they correspond to

Table 2: Optimal RNNs configurations adopted in the three real-world STLF problems. Refer to Tab. 1 for the definition of the acronyms in this table.

Network	**Task**	**RNN Configuration**							
		N_h		N_l		**TDL**		η	λ_2
Narx	Orange	11		4		2		$1.9E-6$	0.082
	ACEA	11		3		2		$1.9E-6$	0.0327
	GEFCom	18		4		9		$6.1E-5$	0.3136
		τ_b	τ_f	N_h	**OPT**	η	p_{drop}	λ_1	λ_2
ERNN	Orange	30	15	100	SGD	0.011	0	0	0.0081
	ACEA	60	30	80	Nesterov	0.00036	0	0	0.0015
	GEFCom	50	25	60	Adam	0.0002	0	0	0.0023
		τ_b	τ_f	N_h	**OPT**	η	p_{drop}	λ_1	λ_2
LSTM	Orange	40	20	50	Adam	0.0013	0	0	0.0036
	ACEA	50	25	40	Adam	0.0010	0.1	0	0.0012
	GEFCom	50	25	20	SGD	0.0881	0	0	0.0017
		τ_b	τ_f	N_h	**OPT**	η	p_{drop}	λ_1	λ_2
GRU	Orange	40	20	46	SGD	0.0783	0	0.0133	0.0004
	ACEA	40	20	35	Adam	0.0033	0	0	0.0013
	GEFCom	60	30	23	Adam	0.0005	0	0	0.0043
		N_h	ρ	R_c	ξ	ω_i	ω_o	ω_f	λ_2
ESN	Orange	400	0.5006	0.3596	0.0261	0.2022	0.4787	0.1328	0.3240
	ACEA	800	0.7901	0.4099	0.0025	0.1447	0.5306	0.0604	0.1297
	GEFCom	500	1.7787	0.4283	0.0489	0.7974	0.9932	0.0033	0.2721

unusual increments (or decrements) of load, which differ significantly from the trend observed in the past. For example, the error increases when the load suddenly grows in the last seasonal cycle in Fig. 18.

In the Orange experiment we evaluate the results with or without applying a log transform to the data. We observed sometime log-transform yields slightly worse result ($\sim 0.1\%$), but in most cases the results are equal.

For ERNN SGD is found as optimal, which is a slower yet more precise update strategy and is more suitable for gradient descent if the problem is difficult. ERNN takes into account a limited amount of past information, as the window in the BPTT procedure is set to a relatively small value.

Like ERNN, also for GRU the validation procedure identified SGD as the optimal gradient descent strategy. Interestingly, L_1 regularization is used, while in all the other cases it is not considered. On the other hand, the L_2 regularization parameter is much smaller.

In the optimal NARX configuration, TDL is set to a very small value. In particular, since the regression is performed only on the last 2 time intervals, the current output depends only on the most recent inputs and estimated outputs. From the number of hidden nodes and layers, we observe that the optimal size of the network is relatively small.

Relatively to the ESN configuration, we notice a very small spectral radius. This means that, also in this case, the network is configured with a small amount of memory. This results in reservoir dynamics that are more responsive and, consequently, in outputs that mostly depend on the recent inputs. As a consequence, the value of input scaling is small, since there is no necessity of quickly saturating the neurons activation.

ACEA. The time series of the electricity load is quite regular except for few, erratic fluctuations. As for the Orange dataset, RNN predictions are inaccurate mainly in correspondence of such fluctuations, while they output a correct prediction elsewhere. This behavior is outlined by the plots in Fig. 19, where we observe that the residuals are small and, in each RNN prediction, they are mostly localized in common time intervals. From the NRMSE values in Fig. 17, we see that ESN performs better than the other networks. The worst performance is obtained by NARX, while the gradient-based RNNs yield better results, which are very similar to each other.

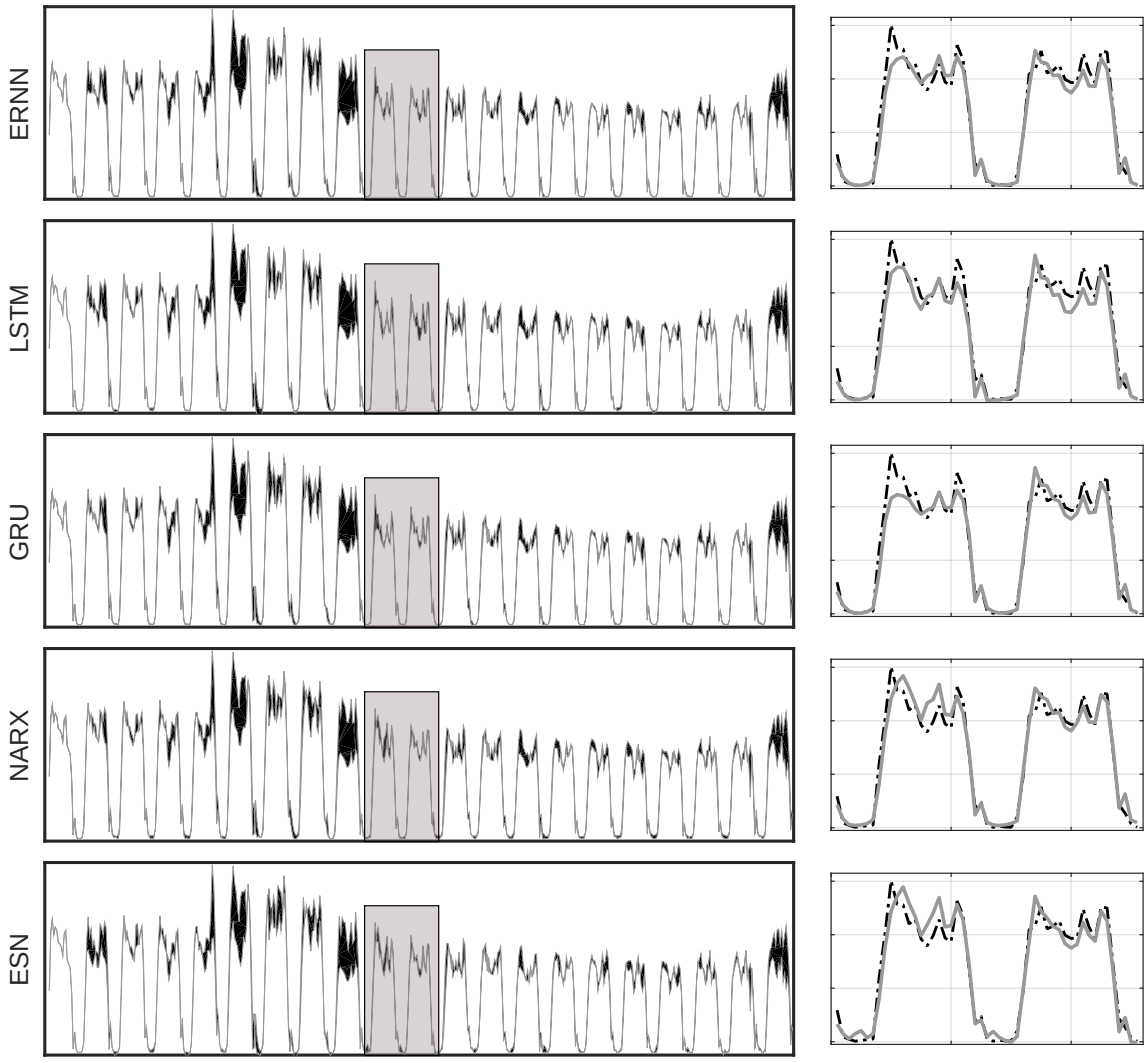


Figure 18: Orange dataset – The plots on the left show the residuals of predictions of each RNN with respect to the ground truth; black areas indicate the errors in the predictions. The plots on right depict a magnification of the area in the gray boxes from the left graphics; the dashed black line is the ground truth, the solid gray line is the prediction of each RNN.

In ERNN and GRU, the optimal regularization found is the L_2 norm, whose coefficient assumes a small value. In LSTM instead, beside the L_2 regularization term, the optimal configuration includes also a dropout regularization with a small probability. The BPTT windows have comparable size in all the gradient-based networks.

The optimal NARX configuration for ACEA is very similar to the one identified for Orange and is characterized by a low complexity in terms of number of hidden nodes and layers. Also in this case the TDLs are very short.

Similarly to the optimal configuration for Orange, the ESN spectral radius assumes a small value, meaning that the network is equipped with a short-term memory and it captures only short temporal correlations in the data. The reservoir is configured with a high connectivity, which yields more homogeneous internal dynamics.

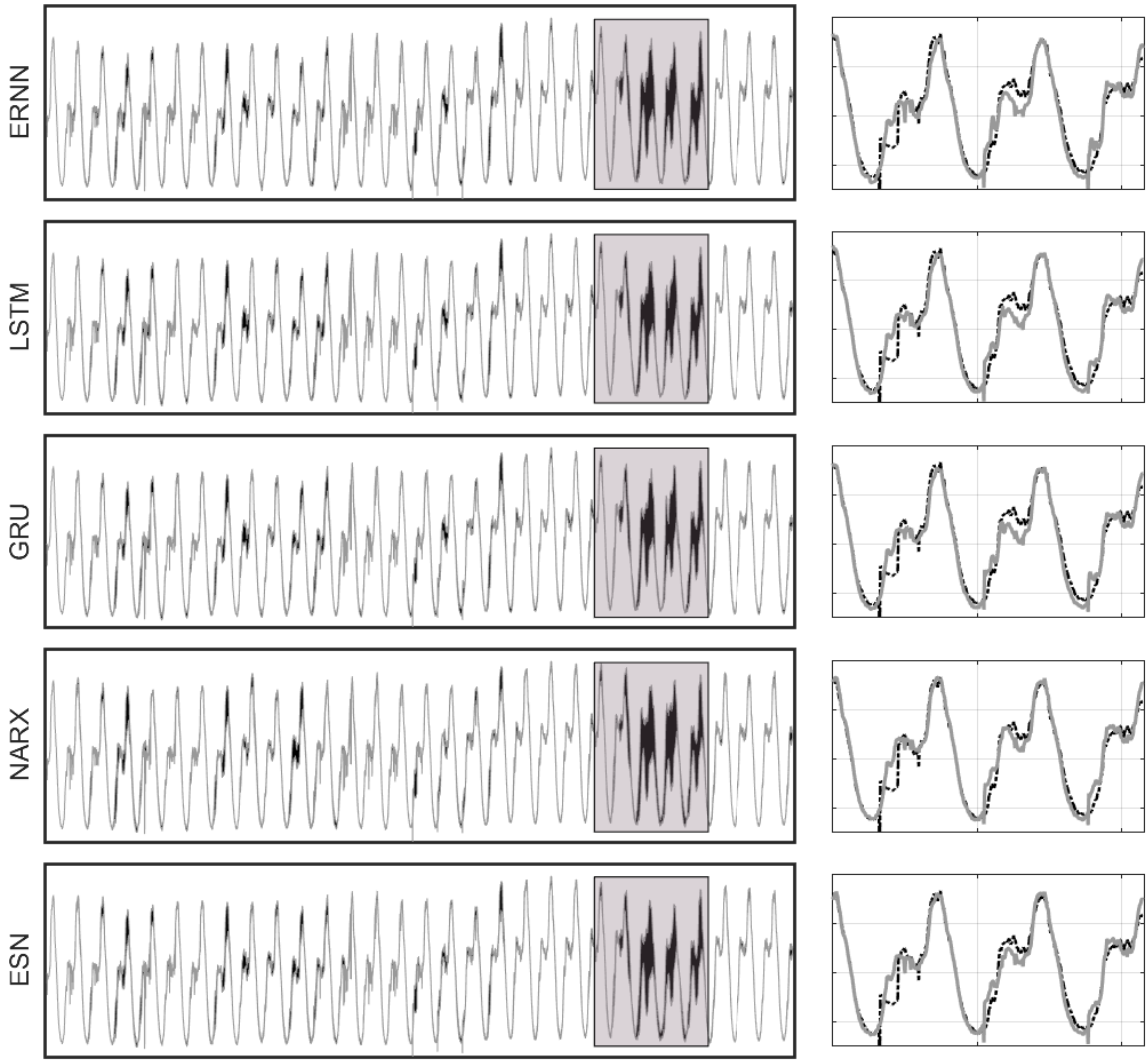


Figure 19: ACEA dataset – The plots on the left show the residuals of predictions of each RNN with respect to the ground truth; black areas indicate the errors in the predictions. The plots on right depict a magnification of the area in the gray boxes from the left graphics; the dashed black line is the ground truth, the solid gray line is the prediction of each RNN.

GEFCom. This time series is more irregular than the previous ones, as it shows a more noisy behavior that is harder to be predicted. From Fig. 20 we see that the extent of the black areas of the residual is much larger than in the other datasets, denoting a higher prediction error. From the third panel in Fig. 17 we observe larger differences in the results with respect to the previous cases. In this dataset, the exogenous time series of temperature plays a key role in the prediction, as it conveys information that are particularly helpful to yield a high accuracy. The main reason of the discrepancy in the results for the different networks may be in their capability of correctly leveraging this exogenous information for building an accurate forecast model.

From the results, we observe that the gradient-based RNNs yield the best prediction accuracy. In particular, ERNN and GRU generate a prediction with the lowest NRMSE with respect to the target signal. ESN, instead, obtains considerably lower performance. Like for the syntethic datasets NARMA and MSO, NARX produces a very inaccurate prediction, scoring a NRMSE which is above 1.

The optimal ERNN configuration consists of only 60 nodes.

For LSTM, the optimal configuration includes only 20 hidden units, which is the lowest amount admitted in the validation search and SGD is the best as optimizer.

The optimal configuration for GRU is characterized by a large BPTT window, which assumes the maximum value allowed. This means that the network benefits from considering a large amount of past values to compute the prediction. As in LSTM, the number of processing units is very low. The best optimizer is Adam initialized with a particularly small learning rate, which yields a slower but more precise gradient update.

The optimal configuration of NARX network is characterized by a quite large number of hidden nodes and layers, which denote a network of higher complexity with respect to the ones identified in the other tasks. This can be related to the TDL larger values, which require to be processed by a network with greater modeling capabilities.

For ESN, we notice an extremely large spectral radius, close to the maximum value admitted in the random search. Consequently, also the value of the input scaling is set to a high number, to increase the amount of nonlinerarity in the processing units. The output scaling is set close to 1, meaning that the teacher signal is almost unchanged when fed into the training procedure. A feedback scaling close to zero means that the feedback is almost disabled and it is not used by the ESN to update its internal state.

8. Conclusions

In this paper we studied the application of recurrent neural networks to time series prediction, focusing on the problem of short term load forecasting. We reviewed five different architectures, ERNN, LSTM, GRU, NARX, and ESN, explaining their internal mechanisms, discussing their properties and the procedures for the training. We performed a comparative analysis of the prediction performance obtained by the different networks on several time series, considering both synthetic benchmarks and real-world short term forecast problems. For each network, we outlined the scheme we followed for the optimization of its hyperparameters. Relative to the real-world problems, we discussed how to preprocess the data according to a detailed analysis of the time series. We completed our analysis by comparing the performance of the RNNs on each task and discussing their optimal configurations.

From our experiments we can draw the following important conclusions.

There is not a specific RNN model that outperforms the others in every prediction problem. The choice of the most suitable architecture depends on the specific task at hand and it is important to consider more training strategies and configurations for each RNN. On average, the NARX network achieved the lowest performance, especially on synthetic problems NARMA and MSO, and on the GEFCom dataset.

The training of gradient-based networks (ERNN, LSTM and GRU) is slower and in general more complex, due to the unfolding and backpropagation through time procedure. However, while some precautions need to be taken in the design of these networks, satisfactory results can be obtained with minimal fine-tuning and by selecting default hyperparameters. This implies that a strong expertise on the data domain is not always necessary.

The results obtained by the ESN are competitive in most tasks and the simplicity of its implementation makes it an appealing instrument for time series prediction. ESN is characterized by a faster training procedure, but the performance heavily depends on the hyperparameters. Therefore, to identify the optimal configuration in the validation phase, ESN requires a search procedure of the hyperparameters that is more accurate than in gradient-based models.

Another important aspect highlighted by our results is that the gated RNNs (LSTM and GRU) did not perform particularly better than an ERNN, whose architecture is much simpler, as well as its training. While LSTM and GRU achieve outstanding results in many sequence learning problems, the additional complexity of the complicated gated mechanisms seems to be unnecessary in many time series predictions tasks.

We hypothesize as a possible explanation that in sequence learning problems, such as the ones of Natural Language Processing [182], the temporal dependencies are more irregular than in the dynamical systems underlying the load time series. In natural language for example, the dependency from a past word can persist for a long time period and then terminate abruptly when a sentence ends. Moreover, there could

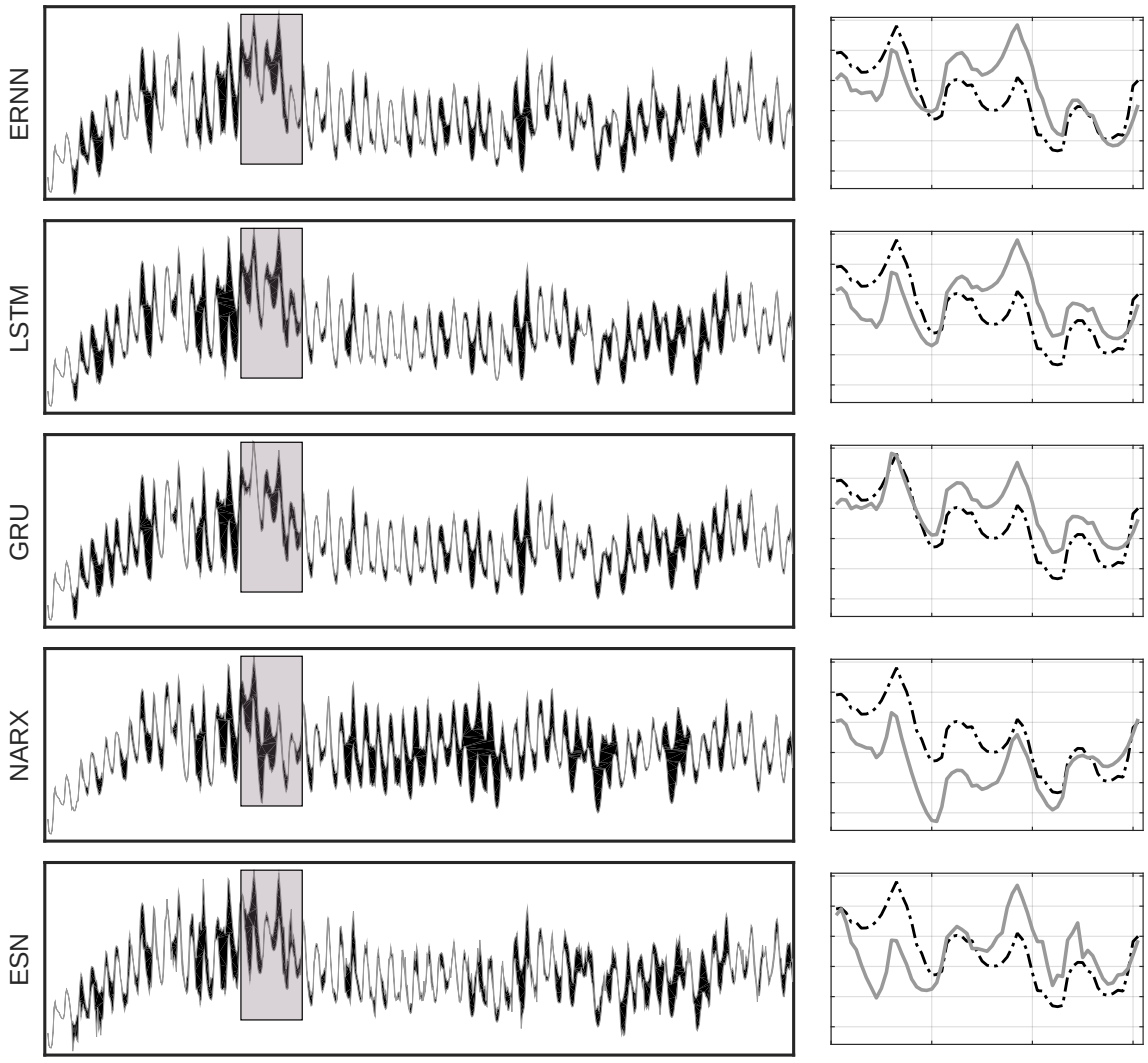


Figure 20: GEFCom dataset – The plots on the left show the residuals of predictions of each RNN with respect to the ground truth; black areas indicate the errors in the predictions. The plots on right depict a magnification of the area in the gray boxes from the left graphics; the dashed black line is the ground truth, the solid gray line is the prediction of each RNN.

exist relations between very localized chunks of the sequence. In this case, the RNN should focus on a specific temporal segment.

LSTM and GRU can efficiently model these highly nonlinear statistical dependencies, since their gating mechanisms allow to quickly modify the memory content of the cells and the internal dynamics. On the other hand, traditional RNNs implement smoother transfer functions and they would require a much larger complexity (number of units) to approximate such nonlinearities. However, in dynamical systems with dependencies that decay smoothly over time, the features of the gates may not be necessary and a simple RNN could be more suitable for the task.

Therefore, we conclude by arguing that ERNN and ESN may represent the most convenient choice in time series prediction problems, both in terms of performance and simplicity of their implementation and training.

References

[1] J. G. D. Gooijer, R. J. Hyndman, 25 years of time series forecasting, International Journal of Forecasting 22 (3) (2006) 443 – 473, ISSN 0169-2070, DOI: http://doi.org/10.1016/j.ijforecast.2006.01.001, twenty five years of forecasting.

[2] D. Simchi-Levi, E. Simchi-Levi, P. Kaminsky, Designing and managing the supply chain: Concepts, strategies, and cases, McGraw-Hill New York, 1999.

[3] D. W. Bunn, Forecasting loads and prices in competitive power markets, Proceedings of the IEEE 88 (2).

[4] P. A. Ruiz, G. Gross, Short-term resource adequacy in electricity market design, IEEE Transactions on Power Systems 23 (3) (2008) 916–926.

[5] A. Deihimi, H. Showkati, Application of echo state networks in short-term electric load forecasting, Energy 39 (1) (2012) 327–340.

[6] Y. Peng, M. Lei, J.-B. Li, X.-Y. Peng, A novel hybridization of echo state networks and multiplicative seasonal ARIMA model for mobile communication traffic series forecasting, Neural Computing and Applications 24 (3-4) (2014) 883–890.

[7] H. Shen, J. Z. Huang, Interday forecasting and intraday updating of call center arrivals, Manufacturing & Service Operations Management 10 (3) (2008) 391–410.

[8] F. M. Bianchi, S. Scardapane, A. Uncini, A. Rizzi, A. Sadeghian, Prediction of telephone calls load using Echo State Network with exogenous variables, Neural Networks 71 (2015) 204–213, DOI: 10.1016/j.neunet.2015.08.010.

[9] F. M. Bianchi, E. De Santis, A. Rizzi, A. Sadeghian, Short-term electric load forecasting using echo state networks and PCA decomposition, IEEE Access 3 (2015) 1931–1943, ISSN 2169-3536, DOI: 10.1109/ACCESS.2015.2485943.

[10] A. Deihimi, O. Orang, H. Showkati, Short-term electric load and temperature forecasting using wavelet echo state networks with neural reconstruction, Energy 57 (2013) 382–401.

[11] G. Jan van Oldenborgh, M. A. Balmaseda, L. Ferranti, T. N. Stockdale, D. L. Anderson, Did the ECMWF seasonal forecast model outperform statistical ENSO forecast models over the last 15 years?, Journal of climate 18 (16) (2005) 3240–3249.

[12] T.-H. Dang-Ha, F. M. Bianchi, R. Olsson, Local Short Term Electricity Load Forecasting: Automatic Approaches, ArXiv e-prints .

[13] R. Hyndman, A. B. Koehler, J. K. Ord, R. D. Snyder, Forecasting with exponential smoothing: the state space approach, Springer Science & Business Media, ISBN 9783540719182, 2008.

[14] G. E. Box, G. M. Jenkins, G. C. Reinsel, Time series analysis: forecasting and control, vol. 734, John Wiley & Sons, 2011.

[15] G. E. Box, D. R. Cox, An analysis of transformations, Journal of the Royal Statistical Society. Series B (Methodological) (1964) 211–252.

[16] J. W. Taylor, A comparison of univariate time series methods for forecasting intraday arrivals at a call center, Management Science 54 (2) (2008) 253–265.

[17] F. Takens, Detecting strange attractors in turbulence, Springer, 1981.

[18] N. I. Sapankevych, R. Sankar, Time series prediction using support vector machines: a survey, Computational Intelligence Magazine, IEEE 4 (2) (2009) 24–38.

[19] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (5) (1989) 359 – 366, ISSN 0893-6080, DOI: http://dx.doi.org/10.1016/0893-6080(89)90020-8.

[20] J.-S. R. Jang, ANFIS: adaptive-network-based fuzzy inference system, Systems, Man and Cybernetics, IEEE Transactions on 23 (3) (1993) 665–685.

[21] G. Zhang, B. E. Patuwo, M. Y. Hu, Forecasting with artificial neural networks:: The state of the art, International Journal of Forecasting 14 (1) (1998) 35 – 62, ISSN 0169-2070, DOI: http://doi.org/10.1016/S0169-2070(97)00044-7.

[22] H. Hippert, C. Pedreira, R. Souza, Neural networks for short-term load forecasting: a review and evaluation, IEEE Transactions on Power Systems 16 (1) (2001) 44–55, ISSN 08858950, DOI: 10.1109/59.910780.

[23] R. Law, Back-propagation learning in improving the accuracy of neural network-based tourism demand forecasting, Tourism Management 21 (4) (2000) 331 – 340, ISSN 0261-5177, DOI: http://doi.org/10.1016/S0261-5177(99)00067-9.

[24] S.-H. Tsaur, Y.-C. Chiu, C.-H. Huang, Determinants of guest loyalty to international tourist hotels—a neural network approach, Tourism Management 23 (4) (2002) 397 – 405, ISSN 0261-5177, DOI: http://doi.org/10.1016/S0261-5177(01)00097-8.

[25] S. C. Kon, L. W. Turner, Neural network forecasting of tourism demand, Tourism Economics 11 (3) (2005) 301–328, DOI: http://dx.doi.org/10.5367/000000005774353006.

[26] A. Palmer, J. J. Montaño, A. Sesé, Designing an artificial neural network for forecasting tourism time series, Tourism Management 27 (5) (2006) 781 – 790, ISSN 0261-5177, DOI: http://doi.org/10.1016/j.tourman.2005.05.006.

[27] O. Claveria, S. Torra, Forecasting tourism demand to Catalonia: Neural networks vs. time series models, Economic Modelling 36 (2014) 220 – 228, ISSN 0264-9993, DOI: http://doi.org/10.1016/j.econmod.2013.09.024.

[28] N. Kourentzes, Intermittent demand forecasts with neural networks, International Journal of Production Economics 143 (1) (2013) 198 – 206, ISSN 0925-5273, DOI: http://doi.org/10.1016/j.ijpe.2013.01.009.

[29] L. A. Díaz-Robles, J. C. Ortega, J. S. Fu, G. D. Reed, J. C. Chow, J. G. Watson, J. A. Moncada-Herrera, A hybrid {ARIMA} and artificial neural networks model to forecast particulate matter in urban areas: The case of Temuco, Chile, Atmospheric Environment 42 (35) (2008) 8331 – 8340, ISSN 1352-2310, DOI: http://doi.org/10.1016/j.atmosenv.2008.07.020.

[30] E. Plummer, Time series forecasting with feed-forward neural networks: guidelines and limitations, Neural Networks 1 (2000) 1.
[31] J. P. Teixeira, P. O. Fernandes, Tourism Time Series Forecast -Different ANN Architectures with Time Index Input, Procedia Technology 5 (2012) 445 – 454, ISSN 2212-0173, DOI: http://dx.doi.org/10.1016/j.protcy.2012.09.049.
[32] O. Claveria, E. Monte, S. Torra, Tourism Demand Forecasting with Neural Network Models: Different Ways of Treating Information, International Journal of Tourism Research 17 (5) (2015) 492–500, ISSN 1522-1970, DOI: 10.1002/jtr.2016, jTR-13-0416.R2.
[33] A. M. Schäfer, H.-G. Zimmermann, Recurrent Neural Networks are Universal Approximators, International Journal of Neural Systems 17 (04) (2007) 253–263, DOI: 10.1142/S0129065707001111.
[34] F. M. Bianchi, M. Kampffmeyer, E. Maiorino, R. Jenssen, Temporal Overdrive Recurrent Neural Network, arXiv preprint arXiv:1701.05159 .
[35] A. M. Schäfer, H.-G. Zimmermann, Recurrent Neural Networks are universal approximators., International journal of neural systems 17 (4) (2007) 253–263, ISSN 0129-0657, DOI: 10.1142/S0129065707001111.
[36] A. Graves, Sequence transduction with recurrent neural networks, arXiv preprint arXiv:1211.3711 .
[37] A. Graves, Generating sequences with recurrent neural networks, arXiv preprint arXiv:1308.0850 (2013) 1–43.
[38] R. Pascanu, T. Mikolov, Y. Bengio, On the Difficulty of Training Recurrent Neural Networks, in: Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28, ICML'13, JMLR.org, III–1310–III–1318, DOI: http://dl.acm.org/citation.cfm?id=3042817.3043083, 2013.
[39] T. Mikolov, Statistical language models based on neural networks, Ph.D. thesis, PhD thesis, Brno University of Technology. 2012.[PDF], 2012.
[40] I. Sutskever, J. Martens, G. E. Hinton, Generating text with recurrent neural networks, in: Proceedings of the 28th International Conference on Machine Learning (ICML-11), 1017–1024, 2011.
[41] A. Graves, Practical variational inference for neural networks, in: Advances in Neural Information Processing Systems, 2348–2356, 2011.
[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Advances in neural information processing systems, 3111–3119, 2013.
[43] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu, Wavenet: A generative model for raw audio, arXiv preprint arXiv:1609.03499 .
[44] A. Graves, J. Schmidhuber, Offline handwriting recognition with multidimensional recurrent neural networks, in: Advances in neural information processing systems, 545–552, 2009.
[45] A. Graves, M. Liwicki, H. Bunke, J. Schmidhuber, S. Fernández, Unconstrained on-line handwriting recognition with recurrent neural networks, in: Advances in Neural Information Processing Systems, 577–584, 2008.
[46] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, D. Wierstra, DRAW: A recurrent neural network for image generation, arXiv preprint arXiv:1502.04623 .
[47] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural computation 9 (8) (1997) 1735–1780.
[48] J. Weston, S. Chopra, A. Bordes, Memory Networks, CoRR abs/1410.3916.
[49] A. Graves, G. Wayne, I. Danihelka, Neural Turing Machines, CoRR abs/1410.5401, DOI: http://arxiv.org/abs/1410.5401.
[50] F. A. Gers, D. Eck, J. Schmidhuber, Applying LSTM to Time Series Predictable through Time-Window Approaches, in: G. Dorffner, H. Bischof, K. Hornik (Eds.), Artificial Neural Networks — ICANN 2001: International Conference Vienna, Austria, August 21–25, 2001 Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-44668-2, 669–676, DOI: 10.1007/3-540-44668-0_93, 2001.
[51] V. Flunkert, D. Salinas, J. Gasthaus, DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks, ArXiv e-prints .
[52] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, J. Schmidhuber, LSTM: A search space odyssey, arXiv preprint arXiv:1503.04069 .
[53] P. Malhotra, L. Vig, G. Shroff, P. Agarwal, Long short term memory networks for anomaly detection in time series, in: Proceedings, Presses universitaires de Louvain, 89, 2015.
[54] W. Maass, P. Joshi, E. D. Sontag, Computational aspects of feedback in neural circuits, PLoS Computational Biology 3 (1) (2007) e165, DOI: 10.1371/journal.pcbi.0020165.eor.
[55] H. T. Siegelmann, E. D. Sontag, Turing computability with neural nets, Applied Mathematics Letters 4 (6) (1991) 77–80.
[56] J. Schmidhuber, D. Wierstra, M. Gagliolo, F. Gomez, Training recurrent networks by evolino, Neural computation 19 (3) (2007) 757–779.
[57] H. Jaeger, The "echo state" approach to analysing and training recurrent neural networks-with an erratum note, Bonn, Germany: German National Research Center for Information Technology GMD Technical Report 148 (2001) 34.
[58] S. Zhang, Y. Wu, T. Che, Z. Lin, R. Memisevic, R. R. Salakhutdinov, Y. Bengio, Architectural Complexity Measures of Recurrent Neural Networks, in: D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, R. Garnett (Eds.), Advances in Neural Information Processing Systems 29, Curran Associates, Inc., 1822–1830, 2016.
[59] J. Koutník, K. Greff, F. J. Gomez, J. Schmidhuber, A Clockwork RNN, CoRR abs/1402.3511.
[60] I. Sutskever, G. Hinton, Temporal-kernel recurrent neural networks, Neural Networks 23 (2) (2010) 239–243.
[61] M. Schuster, K. K. Paliwal, Bidirectional recurrent neural networks, Signal Processing, IEEE Transactions on 45 (11) (1997) 2673–2681.
[62] S. S. Schoenholz, J. Gilmer, S. Ganguli, J. Sohl-Dickstein, Deep Information Propagation, ArXiv e-prints .

[63] Z. C. Lipton, A Critical Review of Recurrent Neural Networks for Sequence Learning, CoRR abs/1506.00019, DOI: http://arxiv.org/abs/1506.00019.
[64] G. Montavon, G. Orr, K.-R. Müller, Neural networks-tricks of the trade second edition, Springer, DOI: 10.1007/978-3-642-35289-8, 2012.
[65] S. Scardapane, D. Comminiello, A. Hussain, A. Uncini, Group sparse regularization for deep neural networks, Neurocomputing 241 (2017) 81 – 89, ISSN 0925-2312, DOI: https://doi.org/10.1016/j.neucom.2017.02.029.
[66] S. Scardapane, D. Wang, Randomness in neural networks: an overview, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 7 (2) (2017) e1200–n/a, ISSN 1942-4795, DOI: 10.1002/widm.1200, e1200.
[67] H. Jaeger, Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach, vol. 5, GMD-Forschungszentrum Informationstechnik, 2002.
[68] R. J. Williams, D. Zipser, A learning algorithm for continually running fully recurrent neural networks, Neural computation 1 (2) (1989) 270–280.
[69] S. S. Haykin, S. S. Haykin, S. S. Haykin, Kalman filtering and neural networks, Wiley Online Library, 2001.
[70] H. John, Holland, Adaptation in natural and artificial systems, 1992.
[71] M. Lukoševičius, H. Jaeger, Reservoir computing approaches to recurrent neural network training, Computer Science Review 3 (3) (2009) 127–149, DOI: 10.1016/j.cosrev.2009.03.005.
[72] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning internal representations by error propagation, Tech. Rep., DTIC Document, 1985.
[73] R. J. Williams, J. Peng, An efficient gradient-based algorithm for on-line training of recurrent network trajectories, Neural computation 2 (4) (1990) 490–501, DOI: 10.1162/neco.1990.2.4.490.
[74] I. Sutskever, Training recurrent neural networks, Ph.D. thesis, University of Toronto, 2013.
[75] R. J. Williams, D. Zipser, Gradient-based learning algorithms for recurrent networks and their computational complexity, Backpropagation: Theory, architectures, and applications 1 (1995) 433–486.
[76] V. Pham, T. Bluche, C. Kermorvant, J. Louradour, Dropout improves recurrent neural networks for handwriting recognition, in: Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on, IEEE, 285–290, 2014.
[77] Y. Gal, Z. Ghahramani, A Theoretically Grounded Application of Dropout in Recurrent Neural Networks, ArXiv e-prints .
[78] J. G. Zilly, R. K. Srivastava, J. Koutník, J. Schmidhuber, Recurrent Highway Networks, ArXiv e-prints .
[79] Z. Che, S. Purushotham, K. Cho, D. Sontag, Y. Liu, Recurrent Neural Networks for Multivariate Time Series with Missing Values, CoRR abs/1606.01865.
[80] A. Neelakantan, L. Vilnis, Q. V. Le, I. Sutskever, L. Kaiser, K. Kurach, J. Martens, Adding Gradient Noise Improves Learning for Very Deep Networks, arXiv preprint arXiv:1511.06807 .
[81] J. D. Lee, B. Recht, N. Srebro, J. Tropp, R. R. Salakhutdinov, Practical large-scale optimization for max-norm regularization, in: Advances in Neural Information Processing Systems, 1297–1305, 2010.
[82] Y. Bengio, Practical Recommendations for Gradient-Based Training of Deep Architectures, in: G. Montavon, G. B. Orr, K.-R. Müller (Eds.), Neural Networks: Tricks of the Trade: Second Edition, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-642-35289-8, 437–478, DOI: 10.1007/978-3-642-35289-8_26, 2012.
[83] L. Bottou, Stochastic Learning, in: O. Bousquet, U. von Luxburg (Eds.), Advanced Lectures on Machine Learning, Lecture Notes in Artificial Intelligence, LNAI 3176, Springer Verlag, Berlin, 146–168, DOI: http://leon.bottou.org/papers/bottou-mlss-2004, 2004.
[84] L. Bottou, Stochastic gradient descent tricks, in: Neural Networks: Tricks of the Trade, Springer, 421–436, 2012.
[85] L. Bottou, Stochastic Gradient Descent Tricks, in: G. Montavon, G. B. Orr, K.-R. Müller (Eds.), Neural Networks: Tricks of the Trade: Second Edition, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-642-35289-8, 421–436, DOI: 10.1007/978-3-642-35289-8_25, 2012.
[86] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, Y. Bengio, Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, in: Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, K. Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 27, Curran Associates, Inc., 2933–2941, 2014.
[87] Y. Nesterov, A method of solving a convex programming problem with convergence rate O(1/sqrt(k)), Soviet Mathematics Doklady 27 (1983) 372–376.
[88] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, The Journal of Machine Learning Research 12 (2011) 2121–2159.
[89] T. Tieleman, G. Hinton, Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, COURSERA: Neural Networks for Machine Learning 4 (2012) 2.
[90] D. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 .
[91] J. Martens, Deep learning via Hessian-free optimization, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), 735–742, 2010.
[92] I. Sutskever, J. Martens, G. E. Dahl, G. E. Hinton, On the importance of initialization and momentum in deep learning., ICML (3) 28 (2013) 1139–1147.
[93] R. Pascanu, Ç. Gülçehre, K. Cho, Y. Bengio, How to Construct Deep Recurrent Neural Networks, CoRR abs/1312.6026.
[94] I. Sutskever, O. Vinyals, Q. V. Le, Sequence to sequence learning with neural networks, in: Advances in neural information processing systems, 3104–3112, 2014.

[95] S. El Hihi, Y. Bengio, Hierarchical Recurrent Neural Networks for Long-term Dependencies, in: Proceedings of the 8th International Conference on Neural Information Processing Systems, NIPS'95, MIT Press, Cambridge, MA, USA, 493–499, DOI: http://dl.acm.org/citation.cfm?id=2998828.2998898, 1995.

[96] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[97] F. M. Bianchi, L. Livi, C. Alippi, Investigating Echo-State Networks Dynamics by Means of Recurrence Analysis, IEEE Transactions on Neural Networks and Learning Systems PP (99) (2016) 1–13, ISSN 2162-237X, DOI: 10.1109/TNNLS.2016.2630802.

[98] L. Livi, F. M. Bianchi, C. Alippi, Determination of the Edge of Criticality in Echo State Networks Through Fisher Information Maximization, IEEE Transactions on Neural Networks and Learning Systems PP (99) (2017) 1–12, ISSN 2162-237X, DOI: 10.1109/TNNLS.2016.2644268.

[99] R. Pascanu, T. Mikolov, Y. Bengio, Understanding the exploding gradient problem, Computing Research Repository (CoRR) abs/1211.5063 .

[100] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: International conference on artificial intelligence and statistics, 249–256, 2010.

[101] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE International Conference on Computer Vision, 1026–1034, 2015.

[102] V. Nair, G. E. Hinton, Rectified Linear Units Improve Restricted Boltzmann Machines, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel, 807–814, 2010.

[103] R. K. Srivastava, K. Greff, J. Schmidhuber, Training Very Deep Networks, in: C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems 28, Curran Associates, Inc., 2377–2385, 2015.

[104] K. He, X. Zhang, S. Ren, J. Sun, Deep Residual Learning for Image Recognition, ArXiv e-prints .

[105] F. J. Gomez, R. Miikkulainen, Robust non-linear control through neuroevolution, Computer Science Department, University of Texas at Austin, 2003.

[106] J. L. Elman, Language as a dynamical system, Mind as motion: Explorations in the dynamics of cognition (1995) 195–223.

[107] T. Ogata, M. Murase, J. Tani, K. Komatani, H. G. Okuno, Two-way translation of compound sentences and arm motions by recurrent neural networks, in: Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on, IEEE, 1858–1863, 2007.

[108] H. M. H. Mori, T. O. T. Ogasawara, A recurrent neural network for short-term load forecasting, 1993 Proceedings of the Second International Forum on Applications of Neural Networks to Power Systems 31 (1993) 276–281, DOI: 10.1109/ANN.1993.264315.

[109] X. Cai, N. Zhang, G. K. Venayagamoorthy, D. C. Wunsch, Time series prediction with recurrent neural networks trained by a hybrid PSO-EA algorithm, Neurocomputing 70 (13-15) (2007) 2342–2353, ISSN 09252312, DOI: 10.1016/j.neucom.2005.12.138.

[110] V. Cho, A comparison of three different approaches to tourist arrival forecasting, Tourism Management 24 (3) (2003) 323 – 330, ISSN 0261-5177, DOI: http://doi.org/10.1016/S0261-5177(02)00068-7.

[111] P. Mandal, T. Senjyu, N. Urasaki, T. Funabashi, A neural network based several-hour-ahead electric load forecasting using similar days approach, International Journal of Electrical Power and Energy Systems 28 (6) (2006) 367–373, ISSN 01420615, DOI: 10.1016/j.ijepes.2005.12.007.

[112] H. Chitsaz, H. Shaker, H. Zareipour, D. Wood, N. Amjady, Short-term electricity load forecasting of buildings in microgrids, Energy and Buildings 99 (2015) 50–60, ISSN 03787788, DOI: 10.1016/j.enbuild.2015.04.011.

[113] A. Graves, A.-r. Mohamed, G. Hinton, Speech recognition with deep recurrent neural networks, in: Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, IEEE, 6645–6649, 2013.

[114] D. Eck, J. Schmidhuber, Finding temporal structure in music: Blues improvisation with LSTM recurrent networks, in: Neural Networks for Signal Processing, 2002. Proceedings of the 2002 12th IEEE Workshop on, IEEE, 747–756, 2002.

[115] F. A. Gers, J. Schmidhuber, LSTM recurrent networks learn simple context-free and context-sensitive languages, Neural Networks, IEEE Transactions on 12 (6) (2001) 1333–1340.

[116] O. Vinyals, A. Toshev, S. Bengio, D. Erhan, Show and Tell: Lessons Learned from the 2015 MSCOCO Image Captioning Challenge, IEEE Trans. Pattern Anal. Mach. Intell. 39 (4) (2017) 652–663, DOI: 10.1109/TPAMI.2016.2587640.

[117] X. Ma, Z. Tao, Y. Wang, H. Yu, Y. Wang, Long short-term memory neural network for traffic speed prediction using remote microwave sensor data, Transportation Research Part C: Emerging Technologies 54 (2015) 187–197, ISSN 0968090X, DOI: 10.1016/j.trc.2015.03.014.

[118] K. Pawlowski, K. Kurach, Detecting Methane Outbreaks from Time Series Data with Deep Neural Networks, in: Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing - 15th International Conference, RSFDGrC 2015, Tianjin, China, November 20-23, 2015, Proceedings, vol. 9437, ISBN 978-3-319-25782-2, ISSN 03029743, 475–484, DOI: 10.1007/978-3-319-25783-9_42, 2015.

[119] M. Felder, A. Kaifel, A. Graves, Wind Power Prediction using Mixture Density Recurrent Neural Networks, in: Poster P0.153, 1–7, 2010.

[120] A. Graves, J. Schmidhuber, Framewise phoneme classification with bidirectional LSTM and other neural network architectures, Neural Networks 18 (5–6) (2005) 602 – 610, ISSN 0893-6080, DOI: 10.1016/j.neunet.2005.06.042, iJCNN 2005.

[121] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, arXiv preprint arXiv:1406.1078 .

[122] J. Chung, Ç. Gülçehre, K. Cho, Y. Bengio, Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, CoRR abs/1412.3555.

[123] W. Zaremba, An empirical exploration of recurrent network architectures, Proceedings of the 32nd International Conference on Machine Learning, Lille, France .

[124] I. Leontaritis, S. A. Billings, Input-output parametric models for non-linear systems Part I: deterministic non-linear systems, International journal of control 41 (2) (1985) 303–328.

[125] E. Diaconescu, The use of NARX neural networks to predict chaotic time series, Wseas Transactions on computer research 3 (3) (2008) 182–191.

[126] T.-N. Lin, C. L. Giles, B. G. Horne, S.-Y. Kung, A delay damage model selection algorithm for NARX neural networks, Signal Processing, IEEE Transactions on 45 (11) (1997) 2719–2730.

[127] J. M. P. Menezes, G. A. Barreto, Long-term time series prediction with the NARX network: an empirical evaluation, Neurocomputing 71 (16) (2008) 3335–3343.

[128] H. Xie, H. Tang, Y.-H. Liao, Time series prediction based on NARX neural networks: An advanced approach, in: Machine Learning and Cybernetics, 2009 International Conference on, vol. 3, 1275–1279, DOI: 10.1109/ICMLC.2009.5212326, 2009.

[129] R. Napoli, L. Piroddi, Nonlinear active noise control with NARX models, Audio, Speech, and Language Processing, IEEE Transactions on 18 (2) (2010) 286–295.

[130] G. L. Plett, Adaptive inverse control of linear and nonlinear systems using dynamic neural networks, Neural Networks, IEEE Transactions on 14 (2) (2003) 360–376.

[131] S. A. Billings, Nonlinear system identification: NARMAX methods in the time, frequency, and spatio-temporal domains, John Wiley & Sons, 2013.

[132] H. T. Siegelmann, B. G. Horne, C. L. Giles, Computational capabilities of recurrent NARX neural networks, Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 27 (2) (1997) 208–215.

[133] T. Lin, B. G. Horne, P. Tiňo, C. L. Giles, Learning long-term dependencies in NARX recurrent neural networks, Neural Networks, IEEE Transactions on 7 (6) (1996) 1329–1338.

[134] C.-M. Huang, C.-J. Huang, M.-L. Wang, A particle swarm optimization to identifying the ARMAX model for short-term load forecasting, Power Systems, IEEE Transactions on 20 (2) (2005) 1126–1133.

[135] E. Cambria, G.-B. Huang, L. L. C. Kasun, H. Zhou, C. M. Vong, J. Lin, J. Yin, Z. Cai, Q. Liu, K. Li, et al., Extreme learning machines [trends & controversies], Intelligent Systems, IEEE 28 (6) (2013) 30–59.

[136] S. Scardapane, D. Comminiello, M. Scarpiniti, A. Uncini, Online Sequential Extreme Learning Machine With Kernels, IEEE Transactions on Neural Networks and Learning Systems 26 (9) (2015) 2214–2220, ISSN 2162-237X, DOI: 10.1109/TNNLS.2014.2382094.

[137] W. Maass, T. Natschläger, H. Markram, Real-time computing without stable states: A new framework for neural computation based on perturbations, Neural computation 14 (11) (2002) 2531–2560, DOI: 10.1162/089976602760407955.

[138] L. A. Alexandre, M. J. Embrechts, J. Linton, Benchmarking reservoir computing on time-independent classification tasks, in: Neural Networks, 2009. IJCNN 2009. International Joint Conference on, IEEE, 89–93, 2009.

[139] M. D. Skowronski, J. G. Harris, Automatic speech recognition using a predictive echo state network classifier, Neural networks 20 (3) (2007) 414–423.

[140] D. Hai-yan, P. Wen-jiang, H. Zhen-ya, A multiple objective optimization based echo state network tree and application to intrusion detection, in: VLSI Design and Video Technology, 2005. Proceedings of 2005 IEEE International Workshop on, 443–446, DOI: 10.1109/IWVDVT.2005.1504645, 2005.

[141] S. Han, J. Lee, Fuzzy Echo State Neural Networks and Funnel Dynamic Surface Control for Prescribed Performance of a Nonlinear Dynamic System, Industrial Electronics, IEEE Transactions on 61 (2) (2014) 1099–1112, ISSN 0278-0046, DOI: 10.1109/TIE.2013.2253072.

[142] E. Maiorino, F. Bianchi, L. Livi, A. Rizzi, A. Sadeghian, Data-driven detrending of nonstationary fractal time series with echo state networks, Information Sciences 382-383 (2017) 359–373, DOI: 10.1016/j.ins.2016.12.015.

[143] J. Mazumdar, R. Harley, Utilization of Echo State Networks for Differentiating Source and Nonlinear Load Harmonics in the Utility Network, Power Electronics, IEEE Transactions on 23 (6) (2008) 2738–2745, ISSN 0885-8993, DOI: 10.1109/TPEL.2008.2005097.

[144] S. I. Han, J. M. Lee, Fuzzy echo state neural networks and funnel dynamic surface control for prescribed performance of a nonlinear dynamic system, Industrial Electronics, IEEE Transactions on 61 (2) (2014) 1099–1112.

[145] D. Niu, L. Ji, M. Xing, J. Wang, Multi-variable Echo State Network Optimized by Bayesian Regulation for Daily Peak Load Forecasting, Journal of Networks 7 (11) (2012) 1790–1795.

[146] S. Løkse, F. M. Bianchi, R. Jenssen, Training Echo State Networks with Regularization Through Dimensionality Reduction, Cognitive Computation (2017) 1–15ISSN 1866-9964, DOI: 10.1007/s12559-017-9450-z.

[147] S. Varshney, T. Verma, Half Hourly Electricity Load Prediction using Echo State Network, International Journal of Science and Research 3 (6) (2014) 885–888.

[148] D. Li, M. Han, J. Wang, Chaotic time series prediction based on a novel robust echo state network, IEEE Transactions on Neural Networks and Learning Systems 23 (5) (2012) 787–799.

[149] H. Jaeger, H. Haas, Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication, Science 304 (5667) (2004) 78–80, ISSN 0036-8075, DOI: 10.1126/science.1091277.

[150] A. Rodan, P. Tiňo, Minimum complexity echo state network, IEEE Transactions on Neural Networks 22 (1) (2011) 131–144, DOI: 10.1109/TNN.2010.2089641.
[151] L. Appeltant, M. C. Soriano, G. Van der Sande, J. Danckaert, S. Massar, J. Dambre, B. Schrauwen, C. R. Mirasso, I. Fischer, Information processing using a single dynamical node as complex system, Nature Communications 2 (2011) 468, DOI: 10.1038/ncomms1476.
[152] J. Boedecker, O. Obst, J. T. Lizier, N. M. Mayer, M. Asada, Information processing in echo state networks at the edge of chaos, Theory in Biosciences 131 (3) (2012) 205–213.
[153] D. Verstraeten, B. Schrauwen, On the Quantification of Dynamics in Reservoir Computing, in: C. Alippi, M. Polycarpou, C. Panayiotou, G. Ellinas (Eds.), Artificial Neural Networks – ICANN 2009, vol. 5768, Springer Berlin, Heidelberg, ISBN 978-3-642-04273-7, 985–994, DOI: 10.1007/978-3-642-04274-4_101, 2009.
[154] Bianchi Filippo Maria, Livi Lorenzo, Alippi Cesare, Jenssen Robert, Multiplex visibility graphs to investigate recurrent neural network dynamics, Scientific Reports 7 (2017) 44037, DOI: http://dx.doi.org/10.1038/srep4403710.1038/srep44037.
[155] A. M. Fraser, H. L. Swinney, Independent coordinates for strange attractors from mutual information, Physical review A 33 (2) (1986) 1134.
[156] W. Liebert, H. Schuster, Proper choice of the time delay for the analysis of chaotic time series, Physics Letters A 142 (2-3) (1989) 107–111.
[157] H. Jaeger, Adaptive nonlinear system identification with echo state networks, in: Advances in neural information processing systems, 593–600, 2002.
[158] Y. Xue, L. Yang, S. Haykin, Decoupled echo state networks with lateral inhibition, Neural Networks 20 (3) (2007) 365 – 376, ISSN 0893-6080, DOI: http://dx.doi.org/10.1016/j.neunet.2007.04.014, echo State Networks and Liquid State Machines.
[159] D. Wierstra, F. J. Gomez, J. Schmidhuber, Modeling systems with internal state using evolino, in: Proceedings of the 7th annual conference on Genetic and evolutionary computation, ACM, 1795–1802, 2005.
[160] G. Zhang, M. Qi, Neural network forecasting for seasonal and trend time series, European Journal of Operational Research 160 (2) (2005) 501 – 514, ISSN 0377-2217, DOI: http://doi.org/10.1016/j.ejor.2003.08.037, decision Support Systems in the Internet Age.
[161] Orange, D4D challenge, http://http://www.d4d.orange.com/en/Accueil, accessed: 2016-09-22, 2013.
[162] V. D. Blondel, M. Esch, C. Chan, F. Clérot, P. Deville, E. Huens, F. Morlot, Z. Smoreda, C. Ziemlicki, Data for Development: the D4D Challenge on Mobile Phone Data, ArXiv preprint arXiv:1210.0137 .
[163] F. M. Bianchi, A. Rizzi, A. Sadeghian, C. Moiso, Identifying user habits through data mining on call data records, Engineering Applications of Artificial Intelligence 54 (2016) 49 – 61, ISSN 0952-1976, DOI: http://dx.doi.org/10.1016/j.engappai.2016.05.007.
[164] P. H. Franses, Seasonality, non-stationarity and the forecasting of monthly time series, International Journal of Forecasting 7 (2) (1991) 199–208.
[165] H. Shen, J. Z. Huang, Analysis of call centre arrival data using singular value decomposition, Applied Stochastic Models in Business and Industry 21 (3) (2005) 251–263.
[166] R. Ibrahim, P. L'Ecuyer, Forecasting call center arrivals: Fixed-effects, mixed-effects, and bivariate models, Manufacturing & Service Operations Management 15 (1) (2013) 72–85.
[167] B. H. Andrews, S. M. Cunningham, LL Bean improves call-center forecasting, Interfaces 25 (6) (1995) 1–13.
[168] G. P. Zhang, D. M. Kline, Quarterly Time-Series Forecasting With Neural Networks, IEEE Transactions on Neural Networks 18 (6) (2007) 1800–1814, ISSN 1045-9227, DOI: 10.1109/TNN.2007.896859.
[169] O. Claveria, E. Monte, S. Torra, Data pre-processing for neural network-based forecasting: does it really matter?, Technological and Economic Development of Economy 0 (0) (0) 1–17, DOI: 10.3846/20294913.2015.1070772.
[170] J. Weinberg, L. D. Brown, J. R. Stroud, Bayesian forecasting of an inhomogeneous Poisson process with applications to call center data, Journal of the American Statistical Association 102 (480) (2007) 1185–1198.
[171] E. D. Santis, L. Livi, A. Sadeghian, A. Rizzi, Modeling and recognition of smart grid faults by a combined approach of dissimilarity learning and one-class classification, Neurocomputing 170 (2015) 368 – 383, ISSN 0925-2312, DOI: http://dx.doi.org/10.1016/j.neucom.2015.05.112, advances on Biological Rhythmic Pattern Generation: Experiments, Algorithms and ApplicationsSelected Papers from the 2013 International Conference on Intelligence Science and Big Data Engineering (IScIDE 2013)Computational Energy Management in Smart Grids.
[172] Kaggle, GEFCom 2012 Global Energy Forecasting Competition 2012, https://www.kaggle.com/c/global-energy-forecasting-competition-2012-load-forecasting, accessed: 2017-04-26, 2012.
[173] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, The Journal of Machine Learning Research 13 (1) (2012) 281–305.
[174] Theano Development Team, Theano: A Python framework for fast computation of mathematical expressions, arXiv e-prints abs/1605.02688.
[175] A. Zeyer, P. Doetsch, P. Voigtlaender, R. Schlüter, H. Ney, A Comprehensive Study of Deep Bidirectional LSTM RNNs for Acoustic Modeling in Speech Recognition, CoRR abs/1606.06871.
[176] V. Pham, T. Bluche, C. Kermorvant, J. Louradour, Dropout Improves Recurrent Neural Networks for Handwriting Recognition, in: 2014 14th International Conference on Frontiers in Handwriting Recognition, ISSN 2167-6445, 285–290, DOI: 10.1109/ICFHR.2014.55, 2014.

[177] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting, The Journal of Machine Learning Research 15 (1) (2014) 1929–1958.

[178] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, D. S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, DOI: `http://tensorflow.org/`, software available from tensorflow.org, 2015.

[179] R. Battiti, First-and second-order methods for learning: between steepest descent and Newton's method, Neural computation 4 (2) (1992) 141–166.

[180] D. Li, M. Han, J. Wang, Chaotic Time Series Prediction Based on a Novel Robust Echo State Network, IEEE Transactions on Neural Networks and Learning Systems 23 (5) (2012) 787–799, ISSN 2162-237X, DOI: `10.1109/TNNLS.2012.2188414`.

[181] Z. Shi, M. Han, Support vector echo-state machine for chaotic time-series prediction, Neural Networks, IEEE Transactions on 18 (2) (2007) 359–372.

[182] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, R. Socher, Ask Me Anything: Dynamic Memory Networks for Natural Language Processing, in: M. F. Balcan, K. Q. Weinberger (Eds.), Proceedings of The 33rd International Conference on Machine Learning, vol. 48 of *Proceedings of Machine Learning Research*, PMLR, New York, New York, USA, 1378–1387, 2016.