POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Quasi-stateful RNNs with truncated back-propagation through time

Laurea Magistrale in High-Performance Computing Engineering - Ingegneria del Calcolo ad Alte Prestazioni

Author: Valentina Moretti

Advisor: Prof. Manuel Roveri, Prof. Cesare Alippi

Co-advisor: Ivan Marisca, Andrea Cini

Academic year: 2023-2024

1. Introduction

In the context of long-term time series forecasting, it is necessary to address the computational complexity of training models and the memory-related issues that arise when dealing with long input histories. Among the machine learning models, recurrent neural networks (RNNs) are particularly advantageous for time series forecasting since they have a recurrent state that allows them to remember the information of previous steps of the time sequence and to capture temporal dependencies. RNNs, with their recurrent connections, are designed to process sequences in order, retaining the hidden state from one batch to the next. This type of training on long time series is computationally expensive and prevents the use of mini-batches leading to memory capacity and scalability issues. This study addresses the challenge of efficiently training RNNs on long time series by proposing a novel methodology to initialize the state of the network from past observations. The primary objective is to improve the computational efficiency and scalability of the training procedure by computing gradients only for a predefined, fixed-size window interval rather than the entire time series while still modeling longer-range dependencies. We propose a new approach, named reservoir recurrent neural network (ReRNN), composed of a reservoir that encodes the past sequences without weighing on the training of the model, followed by an RNN which initializes its hidden state with an encoding of the reservoir state and that is trained only on the most recent data. In this way, the input can be divided into batches, of a size that fits the hardware, and that can be processed in parallel, without preventing the RNN from taking into account the history of the time series preceding its input window.

2. Problem statement

2.1. Data structure

We work with a set of K time series, each consisting of W vectors with N_x features. A vector at time step t in the i-th time series is denoted as $\boldsymbol{x}_t^i \in \mathbb{R}^{N_x}$, where t ranges from 1 to W and i ranges from 1 to K. The entire set of time series is represented as $\mathbf{X} \in \mathbb{R}^{W \times K \times N_x}$. $\mathbf{X}_t \in \mathbb{R}^{K \times N_x}$ represents the observations at time step t across all K time series. $\mathbf{X}_{t+T} \in \mathbb{R}^{T \times K \times N_x}$ denotes the observations from time t to $t+T-1$. $\mathbf{X}_{<t}$ includes all observations up to but not including time t. We assume that the time series are

generated by a time-invariant stochastic process where each observation $\boldsymbol{x}_t^i$ depends on past observations $\boldsymbol{x}_{<t}^i$ according to an unknown probability distribution p^i:

$$\boldsymbol{x}_t^i \sim p^i(\boldsymbol{x}_t^i \mid \boldsymbol{x}_{<t}^i) \quad \forall i = 1, \dots, K.$$

2.2. Task description and objective function

Our task is to perform multi-step ahead time series forecasting. Given a time step t, we aim to predict the horizon for H steps in the future using all the past input information $\boldsymbol{X}_{<t}$. We define the target $\mathbf{X}_{t:t+H}$. We aim to train a model $p_{\boldsymbol{\theta}}$ to find the best set of parameters $\boldsymbol{\theta}$ to approximate the stochastic process p^i in which past values influence future ones:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+h}^i \mid \boldsymbol{x}_{<t}^i) \approx p^i(\boldsymbol{x}_{t+h}^i \mid \boldsymbol{x}_{<t}^i) \tag{1}$$

$\forall h \in [0, H), \forall i = 1, \dots, K.$

We train the models' family $\mathcal{F}(\cdot; \boldsymbol{\theta})$ that generates the predictions over the horizon $\widehat{\boldsymbol{X}}_{t:t+H}$ that should approximate the expected value of the target value $\boldsymbol{X}_{t:t+H}$. More formally:

$$\widehat{\boldsymbol{x}}_{t:t+H}^i = \mathcal{F}\left(\boldsymbol{x}_{<t}^i; \boldsymbol{\theta}\right) \text{ s.t. } \widehat{\boldsymbol{x}}_{t:t+H}^i \approx \mathbb{E}_p\left[\boldsymbol{x}_{t:t+H}^i\right]$$

$\forall i = 1, \dots, K.$
To learn the parameters $\boldsymbol{\theta}$, it is minimized the cost function $\ell(\cdot)$ on training data such as

$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \frac{1}{K} \sum_{i=1}^{K} \ell\left(\widehat{\boldsymbol{x}}_{t:t+H}^i, \boldsymbol{x}_{t:t+H}^i\right). \tag{2}$$

We use as cost function the mean absolute error (MAE):

$$\ell\left(\widehat{\boldsymbol{x}}_{t:t+H}^i, \boldsymbol{x}_{t:t+H}^i\right) = \frac{1}{H} \sum_{h=0}^{H-1} \left\| \widehat{\boldsymbol{x}}_{t+h}^i - \boldsymbol{x}_{t+h}^i \right\|_1$$

$\forall i = 1, \dots, K.$

3. Baselines

To validate our findings, our approach is compared against relevant baselines having a training procedure that enables subsampling in batches. The first baseline is the linear model which is composed of a single linear layer and is not expected to perform well in the task of time series forecasting since it does not keep track of the dependencies across time steps. We introduce two types of RNNs, namely persistent state recurrent neural network (persistent state RNN) and nonpersistent state recurrent neural network (nonpersistent state RNN). The former type of RNN processes the whole input sequence in order and maintains a linked internal state throughout the entire time sequence without resetting it at any point. The sequential processing allows the model to use the hidden state from previous time steps but makes the training less efficient and scalable and prevents subsampling. In our experiments, we implement a version of persistent state RNN that enables subsampling in the training. In this implementation, the RNN receives batches with a larger number of time steps than the nonpersistent state RNN, so that it is allowed to initially process a portion of the batch without considering any initial state and detaching the gradient from the computation graph. Subsequently, the same RNN processes the remaining portion of the batch having as initial state the one calculated using the previous portion.
The latter type of RNN, instead, is called *nonpersistent state RNN* since the internal state is reset after the processing of each batch. This method is scalable and computationally efficient and it allows the batches to be shuffled and processed in parallel. However, since the state is reset at the beginning of each batch, it only accounts for the history within the time steps of the batch, breaking and losing the dependencies that exist across the different batches. In our approach, we aim to keep the velocity and computational efficiency of nonpersistent state RNNs while having the ability to capture long-term dependencies of the persistent state RNNs. Lastly, echo state networks (ESNs) are taken as baselines since they are computationally efficient and can capture long-term dependencies in the data. The reservoir is made by a high number of recurrent hidden layers that are left untrained after being initialized ([2], [4]). The reservoir is initialized with random weights in the range $[-1, 1]$ and scaled by the spectral radius, which is the maximum absolute value of the eigenvalues of the reservoir matrix. The recurrent weights can be pruned to create sparse connectivity.

The reservoir implements the following equation:

$$\boldsymbol{h}(t) = (1-\alpha)\boldsymbol{h}(t-1)+\alpha f(\boldsymbol{W}_r^r \boldsymbol{u}(t)+\boldsymbol{W}_i^r \boldsymbol{h}(t-1))$$

where $\boldsymbol{u}(t)$ and $\boldsymbol{h}(t)$ are the input and state of the reservoir at time t, $\boldsymbol{W}_r^r$ and $\boldsymbol{W}_i^r$ are the recurrent and input weights of the reservoir, and α is the leaking rate which controls the amount of information from the previous hidden state that is retained in the new hidden state.
The reservoir processes each time step of the series, producing a high-dimensional dataset with the same time steps as the original one, but with a higher number of features, given by the hidden size of the reservoir [4]. The output is derived from stacking the reservoir states and it represents at each time step the input history up to that point in time. The last reservoir state is then mapped to predictions using a linear regression solved with the ordinary least squares method. Therefore, the reservoir pre-computes ahead of training, in an unsupervised manner, the encoding of the history enabling the system to learn the historical patterns of the input time series efficiently before entering the training phase. However, mapping the representation created by the reservoir to the target using a linear layer might be difficult due to the complexity of the task of long-range time series forecasting. Moreover, while there are benefits in not having to train the system, relying on the architectural bias only requires a significantly larger hidden state to cover a broader range of patterns and features and hyperparameters become a critical factor [2].
In the experiments, we decided to include also the comparison with the full backpropagation through time recurrent neural network (FullBPTT RNN), which is an RNN trained on a bigger window than the one considered for the other models. It receives in input the same number of time steps as the persistent state RNN, but it performs full backpropagation through time (BPTT), while the persistent state RNN performs truncated BPTT. Since the FullBPTT RNN model is trained on the full window, it could be able to achieve lower loss compared to the ReRNN model. However, it has drawbacks in decreased speed and increased memory consumption, together with the loss of the possibility of using mini-batch training that we applied consistently in all the other models. Indeed, in case of memory or computational constraints, the use of FullBPTT RNN could be impractical.

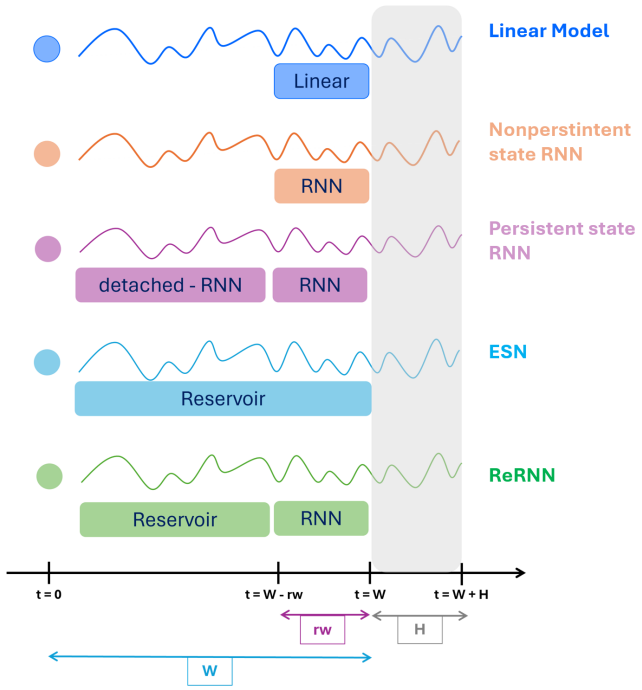

Figure 1: Overview of the models and what part of the time series they process.

4. Proposed solution

To address the problem of training RNNs on long time series, we propose ReRNN, a new approach that combines the benefits of reservoir computing to process long sequences of data with the RNNs' capacity to better adapt to the dynamics of the process generating the data on which it is trained. We follow a similar approach to [1] which overcomes the problem of scalability of spatiotemporal graph neural networks (STGNNs) and enables the parallelization of the training procedure by employing a reservoir ahead of training and by sampling in batches the input data. We propose a relatively simple modification to the RNN architecture by modifying only the way in which the history of observation is managed while keeping the rest of the architecture unchanged. In order to make the model effective for temporal dynamics, our idea is to initialize the hidden state of the RNN using an encoding of the dynamics of the reservoir instead of using the RNN itself.
Hence, the solution we propose is to train a

model $p_{\boldsymbol{\theta}}$ that, rather than using the whole time steps of the input $\boldsymbol{x}^i_{<t}$ (as in Equation 1), takes only a recent window of data $\boldsymbol{x}^i_{t-W:t}$ and an encoding of the past history $\boldsymbol{z}^i_{<t-W}$. Therefore, we define the model to train $p_{\boldsymbol{\theta}}$ as:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}^i_{t+h} \mid \boldsymbol{x}^i_{t-W:t}, \boldsymbol{z}^i_{t-W}) \approx p^i(\boldsymbol{x}^i_{t+h} \mid \boldsymbol{x}^i_{<t}) \quad (3)$$

$$\forall h \in [0, H) \quad \forall i = 1, \ldots, K$$

where $\boldsymbol{z}^i_{t-W}$ is an encoding of the history of the system obtained as $g(\boldsymbol{x}^i_{<t-W})$ where g is a function that encodes the whole past in a vector.
We aim to learn the parameters $\boldsymbol{\theta}$, by minimizing the cost function $\ell(\cdot)$ on training data (see Equation 2) using as cost function the MAE.
The encoding $\boldsymbol{z}^i_{<t-W}$ is the result of an encoding of the past history given by a reservoir followed by a multilayer perceptron (MLP) (or a linear layer).

4.1. Components of the ReRNN

The ReRNN is composed of a randomized RNN (a reservoir) that processes the whole time series ahead of training. The reservoir projects the dynamics of the time series in a high-dimension representation where the important dynamics and dependencies among the time steps of the time series are embedded in a rich feature space [4]. Its hidden state is processed with an encoder (eg. an MLP) to create an appropriate initialization for an RNN that will process the last time steps of the time series having, in this way, access to the history preceding its window. Since the reservoir is training-free, it can process the whole time series ahead of training and a RNN can be trained in an efficient way on a short window of the time series while accounting for the history of the time series. We choose to use GRU (Gated Recurrent Unit) as the RNN for our experiments.

4.2. Detailed rewiew of the ReRNN

Given an input sequence $\mathbf{X} \in \mathbb{R}^{W \times K \times N_x}$, where:

- W is the number of time steps of the series in input and rw is the window on which we train the RNN, with $rw < W$,
- K is the number of time series,
- N_x is the number of features per time series.

The reservoir processes the input $\mathbf{X}_{0:W-rw}$ from time step 0 to time step $W-rw-1$ and produces an output sequence $\mathbf{Z}$ with the same number of time series and of the same length $(W-rw)$ but on a richer feature space:

$$\boldsymbol{z}^i_{0:(W-rw)} = \text{RESERVOIR}\left(\boldsymbol{x}^i_{0:W-rw}\right)$$

$\forall i = 1, \ldots, K$. The number of features in each time series produced by the reservoir corresponds to the hidden state size N_r of the reservoir itself, hence the output is $\mathbf{Z}_{0:(W-rw)} \in \mathbb{R}^{(W-rw)\times K \times N_r}$.
Only the last reservoir's state after processing the input sequence $\mathbf{Z}_T \in \mathbb{R}^{K \times N_r}$ $(T = W - rw - 1)$ is kept and passed through an MLP that learn how to create an encoding suitable for initializing the GRU. Hence, the MLP outputs a representation with features dimension equal to the dimension of the hidden state of the GRU (N_h), i.e.,

$$\boldsymbol{h}^{i\prime}_T = \text{MLP}\left(\boldsymbol{z}^i_T\right) \in \mathbb{R}^{N_h} \quad \forall i = 1, \ldots, K.$$

The GRU is initialized with $\boldsymbol{h}^{i\prime}_T \in \mathbb{R}^{N_h}$ and it processes the remaining part of the input sequence $\boldsymbol{x}^i_{\text{W-rw:}W} \in \mathbb{R}^{rw \times N_h}$:

$$\boldsymbol{h}^{i\prime}_W = \text{GRU}\left(\boldsymbol{x}^i_{\text{W-rw:W}}, \boldsymbol{h}^{i\prime}_T\right) \in \mathbb{R}^{N_h} \forall i = 1, \ldots, K.$$

The GRU outputs a state $\boldsymbol{h}^{i\prime}_W \in \mathbb{R}^{N_h}$ which is then mapped to prediction through a linear layer (READOUT):

$$\hat{\boldsymbol{x}}^i_{W:W+H} = \text{READOUT}(\boldsymbol{h}^{i\prime}_W) \in \mathbb{R}^{H \times N_x}$$

$\forall i = 1, \ldots, K$.
$\hat{\boldsymbol{x}}^i_{W:W+H} \in \mathbb{R}^{H \times N_x}$ is the predicted output sequence, where H is the horizon length (see Figure 1).
In the case of sliding window approach, the processing remains analogous to the previous case. However, the entire output of the reservoir is kept and associated as a covariate to the batches. Specifically, each batch associated with time steps from t to $t+rw$, with t being any time step in the input $0 \leq t < (W - rw)$, uses a covariate vector $\boldsymbol{z}^i_{t-1} \in \mathbb{R}^{N_r}$ that is the reservoir's representation $\boldsymbol{z}^i$ at the time step preceding t.

5. Experiments

To validate our findings, we benchmark the proposed model and the baseline models on the task of long-range time series forecasting on synthetic

datasets and real-world data from traffic and energy analytics. We collected the test metrics of the baselines and ReRNN models and the performances in terms of GPU utilization, memory consumption, and training speed. Then we analyze how different model behaves at different forecasting horizons and assess the contribution in terms of performance and efficiency of the proposed components.

5.1. Dataset

In our experiments, we used three datasets:

- A synthetic dataset specifically created to access the ability to identify long-range dependencies in the data. The dataset consists of 15000 time series, each 2400 time steps long, generated by summing sine waves with varying frequencies, amplitudes, and phases. Long-period sine waves have periods between $\lfloor \frac{T}{4} \rfloor$ and $\lfloor \frac{T}{2} \rfloor$, while short-period waves have periods between $\lfloor \frac{T}{32} \rfloor$ and $\lfloor \frac{T}{8} \rfloor$. Amplitudes range from 1 to 2, and phases are randomly assigned. Each time series is normalized by the range of its values. The horizon is 400 timesteps long.
- MetrLa is a real-world large-scale traffic dataset collecting traffic speed measurements along the highway of Los Angeles County by 207 sensors during four months. The traffic speed is aggregated into 5-minute windows, and the length of the time series is 34272 time steps. The horizon is 96 timesteps long (corresponding to eight hours).
- The CEREn-SME dataset is a real-world electricity consumption dataset collected by the Commission for Energy Regulation over 5000 Irish homes and small/medium businesses between 2009 and 2010 with 30 minutes granularity. It is composed of 485 time series of 25727 time steps each. The horizon is 96 timesteps long (corresponding to two days).

On the real-world datasets, the sliding-window approach is used.

5.2. Results

Results are averaged over three runs with different seeds. The best model configurations were found through hyperparameter search using the validation data, on equal set of hyperparameters across models for fairness.

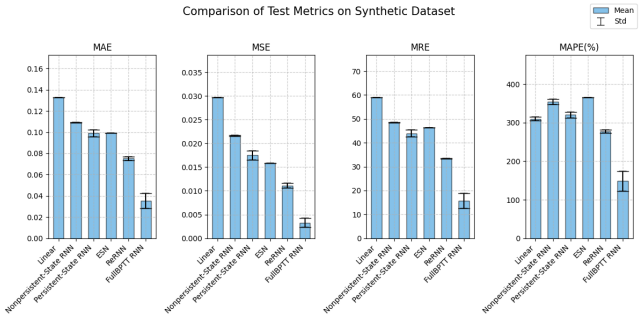

Figure 2: Metrics on the synthetic dataset

The performances of the models on the three datasets in terms of MAE, Mean square error (MSE), Mean absolute percentage error (MAPE) and Mean relative error (MRE) are shown in Figure 2–3–4. Since MAPE and MRE can be problematic on datasets with small or mixed-sign values, for the synthetic and CEREn-SME datasets, which have such issues, we focus on MAE and MSE for evaluating model performance.

As can be seen from the plots, considering equal window sizes, the ReRNN model consistently outperforms all the other baselines when using a training procedure that enables subsampling in batches and employs the same window size.

For the synthetic dataset, the FullBPTT RNN model outperforms the ReRNN model in forecasting accuracy, showing a potential margin for improvement in the ReRNN model. This suggests that the ReRNN model might require a larger representation or more extensive hyperparameter tuning.

In the case of the Metr-La dataset, the ReRNN model outperforms the FullBPTT RNN model, showing that in this setting, our approach outperforms a fully-trainable RNN with a much bigger window as input. One possible explanation might be that FullBPTT RNN, being trained on a larger window, is more difficult to train and may overfit the training data.

Lastly, the FullBPTT RNN model outperforms the ReRNN model on the CEREn-SME dataset, but the performances are so close that choosing among the two models becomes a matter of choice between the resources needed to train the model and the need for a more accurate prediction.

In Figure 5 are shown the performances of the

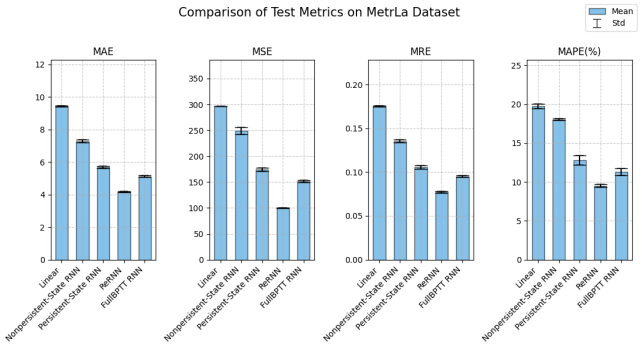

Figure 3: Metrics on the MetrLa dataset.

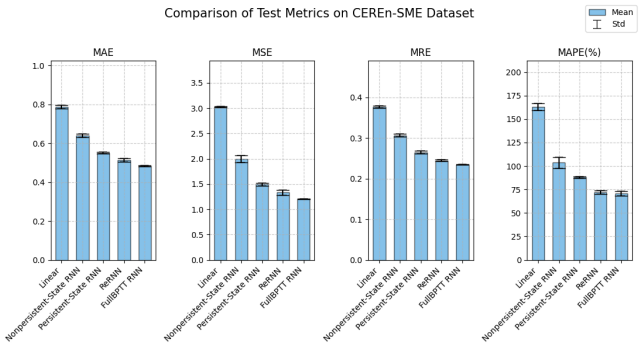

Figure 4: Metrics on the CEREn-SME dataset

models on the synthetic dataset in terms of GPU memory consumption, GPU usage and the number of batches per second in the training phase. The plot shows that the ReRNN significantly outperforms in terms of velocity, GPU usage and GPU memory consumption the persistent state RNN and that persistent state RNN and FullBPTT RNN consistently have the worst performances.

Moreover, we performed a study to assess the

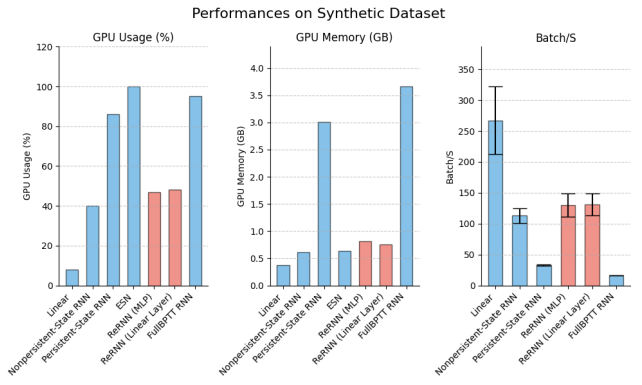

Figure 5: Plot of the performances of the models on the synthetic dataset with a focus on the ablation study (in red).

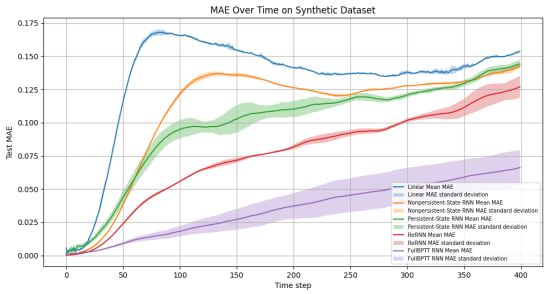

Figure 6: MAE over the horizon for the synthetic dataset

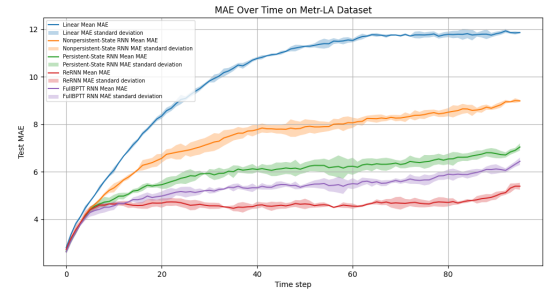

Figure 7: MAE over the horizon for the MetrLa dataset

models' different behaviors and their ability to predict further into the future by plotting the test MAE across all time steps of the horizon. The results on real-world datasets (see Figure 7–8) show that FullBPTT RNN and ReRNN can predict time steps into the future with almost the same metric performance as the time steps near the present.

Finally, we conducted an ablation study to assess the contribution of the various ReRNN components. Replacing the MLP with a linear layer resulted in a very little drop in both metric and efficiency performance that does not alter the

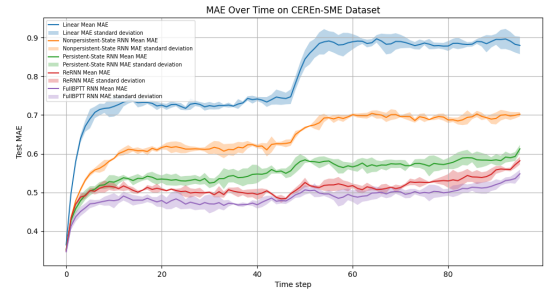

Figure 8: MAE over the horizon for the CEREn-SME dataset

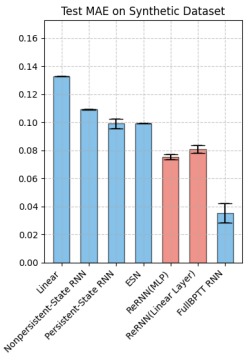

Figure 9: Ablation study: test MA on the synthetic dataset.

previous findings (see Figure 5, 9). This result is in line with our expectations and suggests that the ReRNN's superiority may be due to the reservoir-based initialization, and not to the added complexity of the MLP layer.

6. Conclusions

Reported results show that our ReRNN outperforms state-of-the-art approaches in terms of forecasting accuracy while keeping memory consumption and GPU utilization low and higher speed in the training phase compared to a FullBPTT RNN. Hence, our work introduces a new framework that combines scalable and efficient training, which is crucial in some contexts like distributed sensor networks with hardware constraints. Furthermore, our hybrid architecture can be simpler to train resulting in improved performance accuracy in some scenarios.

6.1. Limitations

Our approach enables subsampling of the input data during training by encoding historical data in fixed-size representations in a pre-processing step. However, the pre-processing step may require a significant amount of memory if the reservoir is large.
Moreover, the transfer of the reservoir's hidden state on the GPU can be time-consuming, as the hidden state can be large and may require significant memory and storage.
Nevertheless, our approach remains scalable and computationally efficient, since it enables subsampling in mini-batches which is particularly necessary in the context of long time series.

6.2. Future works

The most suitable application of this work is in the context of graph neural networks (GNNs). Indeed, GNNs can be used to account for dependencies among time series and this approach could be helpful also in this context. Additionally, it addresses hardware constraints when dealing with distributed sensor networks by sampling the node representations in mini-batches of a size proportional to the hardware capacity.
The work can be extended with the integration of techniques that enhance the stability and improve the memory of the reservoir over long time steps, in addition to the ones already employed (as the Euler state networks [3]). For instance, incorporating delayed hidden states in the reservoir layers that make use of older updates of the hidden state instead of the most recent ones. This technique, and analogous ones, could enhance the ReRNN's performances in terms of metrics and achieve or outperform the FullBPTT RNN.

References

[1] Andrea Cini, Ivan Marisca, Filippo Maria Bianchi, and Cesare Alippi. Scalable spatiotemporal graph neural networks, 2023.

[2] Claudio Gallicchio. Tutorial on Deep Reservoir Neural Networks. `https://sites.google.com/site/cgallicch/resources/tutorial_DRNN`, 2017.

[3] Claudio Gallicchio. Euler state networks: Non-dissipative reservoir computing, 2023.

[4] Claudio Gallicchio and Simone Scardapane. Deep randomized neural networks, 2021.