International Journal of Forecasting 37 (2021) 388–427

Contents lists available at ScienceDirect

International Journal of Forecasting

journal homepage: www.elsevier.com/locate/ijforecast

Recurrent Neural Networks for Time Series Forecasting: Current status and future directions

Hansika Hewamalage *, Christoph Bergmeir, Kasun Bandara

Faculty of Information Technology, Monash University, Melbourne, Australia

ARTICLE INFO

Keywords:
Big data
Forecasting
Best practices
Framework

ABSTRACT

Recurrent Neural Networks (RNNs) have become competitive forecasting methods, as most notably shown in the winning method of the recent M4 competition. However, established statistical models such as exponential smoothing (ETS) and the autoregressive integrated moving average (ARIMA) gain their popularity not only from their high accuracy, but also because they are suitable for non-expert users in that they are robust, efficient, and automatic. In these areas, RNNs have still a long way to go. We present an extensive empirical study and an open-source software framework of existing RNN architectures for forecasting, and we develop guidelines and best practices for their use. For example, we conclude that RNNs are capable of modelling seasonality directly if the series in the dataset possess homogeneous seasonal patterns; otherwise, we recommend a deseasonalisation step. Comparisons against ETS and ARIMA demonstrate that (semi-) automatic RNN models are not silver bullets, but they are nevertheless competitive alternatives in many situations.

1. Introduction

The forecasting field in the past has been characterised, on the one hand, by practitioners who discard neural networks (NNs) as not being competitive, and on the other hand, by NN enthusiasts presenting many complex novel NN architectures, mostly without convincing empirical evaluations against simpler univariate statistical methods. In particular, this notion was supported by many time series forecasting competitions, such as the M3, NN3, and NN5 competitions (Crone, 2008; Crone et al., 2011; Makridakis et al., 2018b). Consequently, NNs were labelled as unsuitable for forecasting (Hyndman, 2018).

There are a number of possible reasons for the underperformance of NNs in the past, one being that individual time series themselves usually are too short to be modelled using complex approaches. Another possibility may be that the characteristics of time series have changed over time, so that even long time series may not contain enough relevant data to fit a complex model. Thus, to model sequences by complex approaches, it is essential that they have adequate length and that they are generated from a comparatively stable system. Also, NNs are further criticised for their black-box nature (Makridakis et al., 2018b). Thus, forecasting practitioners have traditionally opted for more straightforward statistical techniques.

However, we are now living in the era of big data. Companies have gathered a plethora of data over the years, which contain important information about their business patterns. Big data in the context of time series does not necessarily mean that the individual time series contain lots of data. Rather, it typically means that there are many related time series from the same domain. In such a context, univariate forecasting techniques that consider individual time series in isolation may fail to produce reliable forecasts. They become inappropriate in the context of big data, where a single model can learn from many similar time series simultaneously. By

* Correspondence to: Faculty of Information Technology, P.O. Box 63 Monash University, Victoria 3800, Australia.
E-mail address: hansika.hewamalage@monash.edu (H. Hewamalage).

https://doi.org/10.1016/j.ijforecast.2020.06.008
0169-2070/

contrast, more complex models such as NNs benefit most from the availability of massive amounts of data.

Therefore, researchers are now looking into the possibility of applying NNs as substitutes to many other machine learning and statistical techniques. Most notably, in the recent M4 competition, a recurrent neural network (RNN) was able to achieve impressive performance and win the competition (Smyl, 2020). Other examples for successful new developments in the field include novel architectures such as the DeepAR model, multi-quantile RNNs, spline quantile function RNNs, and deep state space models for probabilistic forecasting (Gasthaus et al., 2019; Rangapuram et al., 2018; Salinas et al., 2019; Wen et al., 2017). Alhough there is an abundance of machine learning and NN forecasting methods in the literature, as (Makridakis et al., 2018b) point out, these methods are not typically evaluated rigorously against statistical benchmarks and would usually perform worse than those. This finding of Makridakis et al. (2018b) was arguably one of the main drivers for organising the M4 competition (Makridakis et al., 2018a). Furthermore, often the datasets used or the code implementations for NN-related forecasting research are not publicly available, which poses an issue for the reproducibility of the claimed performance (Makridakis et al., 2018b). A lack of released code implementations also makes it difficult for the forecasting community to adapt such research work to their practical forecasting objectives.

By contrast, popular statistical models such as exponential smoothing (ETS) and the autoregressive integrated moving average (ARIMA) that have traditionally supported forecasting in a univariate context gain their popularity not only from their high accuracy. They also have the advantages of being relatively simple, robust, efficient, and automatic, so that they can be used by non-expert users. For example, the `forecast` (Hyndman & Khandakar, 2008) package in the `R` programming language (R Core Team, 2014) implements a number of statistical techniques related to forecasting, such as ARIMA, ETS, and seasonal and trend decomposition using Loess (STL decomposition), in a single cohesive software package. This package still outshines many other forecasting packages developed later, mainly due to its simplicity, accuracy, robustness, and ease of use.

Consequently, many users of traditional univariate techniques will not have the expertise to develop and adapt complex RNN models. They will want to apply competitive yet easy to use models that can replace the univariate models they currently use in production. Therefore, regardless of the recent successes of RNNs in forecasting, they may still be reluctant to try RNNs as an alternative since they may not have the expert knowledge to use the RNNs to achieve satisfactory accuracy. This is also directly related to the recently emerged dispute in the forecasting community around whether so-called off-the-shelf deep learning techniques are able to outperform classical benchmarks. Furthermore, besides the above-mentioned intuitions around short isolated series versus large time series databases, no established guidelines exist as to when traditional statistical methods will outperform RNNs, and which particular RNN architecture should be used over another or how their parameters should be tuned to fit a practical forecasting context. Although the results from the M4 forecasting competition have clearly shown the potential of RNNs, it remains unclear how competitive RNNs can be in practice in an automated standard approach, without the extensive expert input that competitions provide. Therefore, it is evident that the forecasting community would benefit from standard software implementations, as well as from guidelines and extensive experimental comparisons of the performance of traditional forecasting methods and the different RNN architectures available.

A few recent works have addressed the development of standard software in this area. A package was developed by Tensorflow in Python, for structural time series modelling using the Tensorflow Probability library (Dillon et al., 2017). This package provides support for generating probabilistic forecasts by modelling a time series as a sum of several structural components such as seasonality, local linear trends, and external variables. GluonTS, a Python-based open-source forecasting library recently introduced by Alexandrov et al. (2019), is specifically designed for the purpose of supporting forecasting researchers with easy experimentation using deep neural networks.

We also present a standard software framework, but our work focuses on RNNs and is supported by a review of the literature and implemented techniques, as well as an extensive empirical study. In particular, our study has four main contributions. First, we offer a systematic overview and categorization of the relevant literature. Second, we perform a rigorous empirical evaluation of a number of the most popular RNN architectures for forecasting on several publicly available datasets for pure univariate forecasting problems (i.e., without exogenous variables). The implemented models are standard RNN architectures with adequate preprocessing, without considering special and sophisticated network types such as in Smyl (2020)'s work on RNNs. This is mainly because our motivation in this study is to evaluate the competitiveness of off-the-shelf RNN techniques for forecasting when set against traditional univariate techniques. Our work thus provides insights to non-expert forecasting practitioners who are starting to use RNNs. We compare the performance of the involved models against two state-of-the-art statistical forecasting benchmarks, namely, the implementations of the ETS and ARIMA models from the `forecast` (Hyndman & Khandakar, 2008) package. We stick to single seasonality forecasting problems for comparisons with those two benchmarks. Although the state of the art in forecasting can now be seen as the methods of the M4 competition winners, most notably Montero-Manso et al. (2020) and Smyl (2020), we do not present comparisons against those methods, as they are not automated in a way that they would be straightforwardly applicable to other datasets. Moreover, the idea of our research is to tune RNNs to replace the fully automatic univariate benchmarks that are being used heavily by practitioners for everyday forecasting activities outside of competition environments. Third, based on the experiments, we offer conclusions as a best practices guideline to tune the networks in general. We introduce guidelines

at every step of RNN modelling, from the initial preprocessing of the data to tuning the hyperparameters of the models. The methodologies are generic in that they are not tied to any specific domain. Thus, the main objective of our study is to make this work reproducible by other practitioners for practical forecasting tasks. Finally, all implementations are publicly available as a cohesive open-source software framework.[1]

The rest of the paper is structured as follows. In Section 2, we give a comprehensive background study of all related concepts, including the traditional univariate forecasting techniques and different NN architectures for forecasting mentioned in the literature. Section 3 presents the details of the methodology employed, including the RNN architectures implemented and the corresponding data preprocessing techniques. In Section 4, we explain the experimental framework used for the study with a description of the datasets, training, validation, and testing specifics, and the benchmarks used for the comparison. In Section 5, we present a critical analysis of the results, followed by conclusions in Section 6 and future directions in Section 7.

2. Background study

This section details the literature related to univariate forecasting, traditional univariate forecasting techniques, artificial neural networks (ANNs), and leveraging cross-series information when using ANNs.

2.1. Univariate forecasting

A purely univariate forecasting problem refers to predicting future values of a time series based on its own past values. That is, there is only one time-dependent variable. Given the target series $X = \{x_1, x_2, x_3, \ldots, x_t, \ldots, x_T\}$ the problem of univariate forecasting can be formulated as follows:

$$\{x_{T+1}, \ldots, x_{T+H}\} = F(x_1, x_2, \ldots, x_T) + \epsilon \quad (1)$$

Here, F is the function approximated by the model developed for the problem with the variable X. The function predicts the values of the series for the future time steps from $T+1$ to $T+H$, where H is the intended forecasting horizon. ϵ denotes the error associated with the function approximation F.

2.2. Traditional univariate forecasting techniques

Time series forecasting has traditionally been a research topic in statistics and econometrics, from simple methods such as seasonal naïve and simple exponential smoothing, to more complex ones such as ETS (Hyndman et al., 2008) and ARIMA (Box et al., 1994). In general, traditional univariate methods dominated other computational intelligence methods at many forecasting competitions, including NN3, NN5, and M3 (Crone et al., 2011; Makridakis & Hibon, 2000). The benefit of traditional univariate methods is that they work well when the volume of the data is minimal (Bandara et al., 2020). The number of parameters that must be determined with these techniques is quite low compared to other complex machine learning techniques. However, such traditional univariate techniques lack a few key requirements involved with complex forecasting tasks. Since one model is built for each series, frequent retraining is required, which is computationally intensive, especially in the case of massive time series databases. Also, these univariate techniques are not meant for exploiting cross-series information using global models, since they take into account only the features and patterns inherent in one time series at a time. This is not an issue if the individual time series are long enough and have many data points, such that the model is capable of capturing the sequential patterns. However, in practice, this is usually not the case. On the other hand, learning from many time series can be effectively used to address such problems of limited data availability in a single series.

2.3. Artificial neural networks

With the ever-increasing availability of data, ANNs have become a dominant and popular technique for machine learning tasks. A feed-forward neural network (FFNN) is the most basic type of ANN. It has only forward connections in between the neurons, unlike RNNs, which have feedback loops. There are a number of works where ANNs are used for forecasting. Zhang et al. (1998) provided a comprehensive summary of this research. According to them, ANNs possess various appealing attributes that make them good candidates for forecasting, compared to the aforementioned statistical techniques. First, ANNs can model any form of unknown relationship in the data with few a priori assumptions. Second, ANNs can generalise and transfer the learned relationships to unseen data. Third, ANNs are universal approximators, meaning that they are capable of modelling any form of relationship in the data, especially non-linear relationships (Hornik et al., 1989). The range of functions modelled by an ANN is much higher than the range covered by the statistical techniques.

Tang et al. (1991) compared ANNs to the Box–Jenkins methodology for forecasting and showed that ANNs are comparatively better for forecasting problems with long forecasting horizons. Claveria and Torra (2014) derived the same observations with respect to a tourism-demand forecasting problem. However, determining the best network structure and training procedure for a given problem is crucial to tune ANNs for maximal accuracy. An equally critical decision is the selection of the input variables for modelling (Zhang & Kline, 2007). ANNs have been applied to forecasting in a multitude of domains over the years. Mandal et al. (2006) used an ANN for electric load forecasting, along with a Euclidean norm applied to weather information in both the training data as well as the forecasting duration to derive the similarity between data points in the training data and the forecasts. This technique is called the similar days approach. Efforts

[1] Available at: https://github.com/HansikaPH/time-series-forecasting .

have also been made to minimise manual interventions in the NN modelling process to make it automated. To this end, Yan (2012) proposed a new form of ANN, called the generalised regression neural network (GRNN), which is a special form of a radial basis function (RBF) network. The GRNN requires the estimation of just one design parameter, the spread factor, which decides the width of the RBF and consequently how much the training samples contribute to the output. A hybrid approach to forecasting was proposed by Zhang (2003), who combined an ARIMA model with an ANN to leverage the strengths of both models. The ARIMA model models the linear component of the time series, while the ANN models the residual that corresponds to the non-linear part. This approach is closely related to boosting, commonly used as an ensemble technique to reduce bias in predictions. The idea was inspired from fact that a combination of several models often outperforms any individual model in isolation. Rahman et al. (2016) and Zhang and Berardi (2001) also used ANNs for forecasting, specifically with ensembles.

The most common way to feed time series data into an ANN, specifically an FFNN, is to break the whole sequence into consecutive input windows and then have the FFNN predict the window or the single data point immediately following the input window. Yet, FFNNs ignore the temporal order within the input windows and every new input is considered in isolation (Bianchi et al., 2017). No state is carried forward from the previous inputs to the future time steps. This is where RNNs come into play. RNNs are specialised NNs developed for modelling data with a time dimension.

2.3.1. Recurrent neural networks for forecasting

RNNs are the most commonly used NN architecture for sequence prediction problems. They have gained particular popularity in the domain of natural language processing. Similar to ANNs, RNNs are universal approximators (Schäfer & Zimmermann, 2006). However, unlike ANNs, the feedback loops of the recurrent cells inherently address the temporal order, as well as the temporal dependencies of the sequences (Schäfer & Zimmermann, 2006).

Every RNN is a combination of a number of RNN units. The most popular RNN units used for sequence modelling tasks are the Elman RNN (ERNN) cell, the long short-term memory (LSTM) cell, and the gated recurrent unit (GRU) cell (Cho et al., 2014; Elman, 1990; Hochreiter & Schmidhuber, 1997). Apart from these, other variants have been introduced, such as depth-gated LSTM, clockwork RNNs, stochastic recurrent networks, and bidirectional RNNs (Bayer & Osendorfer, 2014; Koutník et al., 2014; Schuster & Paliwal, 1997; Yao et al., 2015). Nevertheless, these latter RNN units are rarely used for forecasting. They were mostly designed with language modelling tasks in mind.

Jozefowicz et al. (2015) performed an empirical evaluation of different RNN cells. In particular, they aimed to find an RNN unit that performs better than the LSTM and GRU in three defined tasks: arithmetic computations; XML modelling, where the network has to predict the next character in a sequence of XML data; and a language modelling task using the Penn TreeBank dataset. They did not cover time series forecasting in particular. The work by Bianchi et al. (2017) specifically targeted the problem of short-term load forecasting. Their experiments systematically tested the performance of all three popular recurrent cells: the ERNN cell, LSTM cell, and GRU cell. They compared these to the echo state network (ESN) and the non-linear autoregressive with exogenous inputs (NARX) network. The experiments were performed on both synthetic and real-world time series. They found that LSTM and GRU performed similarly with the selected datasets. In general, it is difficult to differentiate which one is better in which scenario. Additionally, the ERNN showed comparable performance to the gated RNN units, and it was faster to train. The authors argued that gated RNN units can potentially outperform ERNNs in language modelling tasks where the temporal dependencies are highly non-linear and abrupt. Furthermore, the authors found that gradient-based RNN units (ERNN, LSTM, and GRU) were relatively slow in terms of training time, due to the time-consuming process of backpropagation.

A recurrent unit can constitute an RNN in various types of architectures. A number of different RNN architectures for forecasting are found in the literature. Although mostly used for natural language processing tasks, these architectures are used in different time series forecasting tasks as well. The stacked architecture is the most commonly used architecture for forecasting with RNNs. Bandara et al. (2020) employed a stacked model in their work, which involved using a clustering approach to group related time series for forecasting. Due to the vanishing gradient problem with vanilla RNN cells, LSTM cells (with peephole connections) were used instead. Their method included several essential data preprocessing steps and a clustering phase of the related time series to exploit cross-series information. They demonstrated significant results on both the CIF 2016 and the NN5 forecasting competition datasets. In fact, theirs was the same architecture used by Smyl (2016) to win the CIF 2016 forecasting competition. In the competition, Smyl developed two global models, one for time series with forecasting horizon 12, and the other one for time series with horizon 6. The work by Smyl and Kuber (2016) closely followed the aforementioned stacked architecture, albeit with a slight modification, known as a skip connection. Skip connections allow layers far below in the stack to directly pass information to layers well above them, and thus minimise the vanishing gradient effect. When skip connections are added, the architecture is called a ResNet architecture. The ResNet architecture, originally used for image recognition, was adapted from the work of He et al. (2016).

Another RNN architecture popular in neural machine translation tasks is the sequence-to-sequence (S2S) architecture introduced by Sutskever et al. (2014). The overall model has two components, the encoder and the decoder, which both act as two RNN networks on their own. Peng et al. (2018) applied a S2S architecture for host load prediction, using GRU cells as the RNN unit in the network. The authors tested their approach using two datasets, the Google clusters dataset, and the Dinda

dataset from traditional Unix systems. The approach was compared to two state-of-the-art RNN models, an LSTM-based network, and the ESN. According to the results, the GRU-based S2S model outperformed the other models in both datasets. The DeepAR model for probabilistic forecasting, developed by Salinas et al. (2019) for Amazon, also uses a S2S architecture for prediction. The authors of that work used the same architecture for both the encoder and the decoder components, although in practice the two components usually differ. Therefore, the weights of both the encoder and the decoder were the same. Wen et al. (2017) developed a probabilistic forecasting model using a slightly modified version of the S2S network. Teacher-signal enforcing of the decoder using autoregressive connections generally leads to the accumulation of errors throughout the prediction horizon. Therefore, the authors used a combination of two multilayer perceptrons (MLPs) for the decoder: a global MLP that encapsulated the encoder outputs and the inputs of the future time steps, and a local MLP that acted upon each specific time step of the prediction horizon to generate the quantiles for that point. Since the decoder does not use any autoregressive connections, the technique is called the direct multi horizon strategy.

More recently, S2S models (also known as autoencoders) were used in forecasting to extract time series features followed by another step to generate the actual predictions. Zhu and Laptev (2017) used an autoencoder to address the issue of uncertainty, specifically in the form of model misspecification. Models that are fitted using training data may not be the optimal ones for the test data when the training data patterns are different from the test data. The autoencoder can alleviate this by extracting the time series features during training, and calculating the difference between these features and the features of the series encountered during testing. This gives the model the intuition of how different the training and test data partitions are. Laptev et al. (2017) incorporated an autoencoder to train a global model across many heterogeneous time series. The autoencoder in this context extracts features of each series that are later concatenated to the input window for forecasting with an LSTM model. Likewise, apart from direct prediction, the S2S model is used in intermediate feature extraction steps prior to the actual forecasting.

Bahdanau et al. (2015) and Luong et al. (2015) presented two different variants of the S2S model, with attention mechanisms. Attention mechanisms were used to overcome a problem with S2S models, namely that they encode all the information in the whole time series to just a single vector and then decode this vector to generate outputs. Embedding all the information in a fixed-size vector can result in information loss (Qin et al., 2017). Hence, the attention model can overcome this by identifying important points in the time series to consider. All the points in the time series are assigned weights that are computed for each output time step. The more important points are assigned higher weights than the less important ones. This method has been empirically proven to be successful in various neural machine translation tasks where the translation of each word from one language to another requires specific attention to particular words in the source language sentence.

Since time series possess seasonality components, attention weights can also be used in a forecasting context. For instance, if a particular monthly time series has yearly seasonality, predicting the value for the next immediate month benefits more from the value of the exact same month from the previous year. This is analogous to assigning more weight to the value of a sequence exactly 12 months prior. The work of Suilin (2017) for the Kaggle challenge of Wikipedia Web Traffic Forecasting encompassed this idea (Google, 2017). However, the author did not use the traditional techniques of Bahdanau et al. (2015) or Luong et al. (2015) since they require re-computation of the attention weights for the whole sequence at each time step, which is computationally expensive. There are two variants to the attention mechanism, proposed by Suilin (2017). In the first variant, the encoder outputs that correspond to important points identified before are directly fed as inputs to the respective time steps in the decoder. In the second variant, the important points concept is relaxed to take a weighted average of those important points along with their two neighbouring points. This scheme helps to account for noise in the time series and to cater to the different lengths of the months and to leap years. This latter variant was further extended to form a type of hierarchical attention using an interconnected hierarchy of 1D convolution layers corresponding to weighted averaging of the neighbouring points and max pooling layers in between. Despite using the same weights for all the forecasting steps in the decoder, Suilin (2017) claimed that this reduced the error significantly.

Cinar et al. (2017) used a more complex attention scheme to treat the periods in the time series. They argued that the classical attention mechanism proposed by Bahdanau et al. (2015) is intended for machine translation, and therefore it does not attend to the seasonal periods in time series in a forecasting condition. Hence, they proposed an extension to Bahdanau-style attention called the position-based content attention mechanism. An extra term ($\pi^{(1)}$) is used in the attention equations to embed the importance of each time step in the time series to calculate the outputs for the forecast horizon. Based on whether or not a particular time step in the history corresponds to a pseudo-period of the output step, the modified equation can be used to either augment or diminish the effect of the hidden state. The vector $\pi^{(1)}$ is trained along with the other parameters of the network. The authors performed experiments using their proposed technique over six datasets, both univariate and multivariate. They further compared the resulting error with the traditional attention mechanism, as well as ARIMA and random forests. The results suggested that the proposed variant of the attention mechanism was promising, since it was able to surpass the other models in terms of accuracy for five of the tested six datasets. Furthermore, the plots of the attention weights of the classical attention scheme and the proposed variant indicated that by introducing this variant, the pseudo-periods were assigned more weight, whereas with naïve attention, the weights

increased towards the time steps closer to the forecast origin.

Qin et al. (2017) proposed a dual-stage attention mechanism (DA-RNN) for multivariate forecasting problems. In the first stage of the attention, which is the input attention, different weights are assigned to different driving series based on their significance for contributing to forecasting at each time step. This is done prior to feeding the input to the encoder component of the S2S network. The input received by each time step of the encoder is a set of values from different exogenous driving series whose influence is sufficiently increased or decreased. The weights pertaining to each driving series' values are derived using another MLP, which is trained alongside the S2S network. In the second stage of attention, which is temporal attention, a usual Bahdanau-style attention scheme is employed to assign weights to the encoder outputs at each prediction step. The model was tested using two datasets. An extensive comparison was performed among many models, including ARIMA, NARX RNN, the encoder–decoder, attention RNN, an input attention RNN, and the DA-RNN. The DA-RNN model outperformed all the other models with both datasets. Further experiments using noisy time series revealed that the DA-RNN is comparatively robust to noise.

More recently, Liang et al. (2018) developed a multi-level attention network, named GeoMAN, for time series forecasting of geo-sensory data. According to the authors, the significance of the model in comparison to the DA-RNN model is that the GeoMAN model explicitly handles the special characteristics inherent in geo-sensory data, such as the spatiotemporal correlations. While the DA-RNN has a single input attention mechanism to differentiate between different driving series, GeoMAN has two levels of spatial attention: local spatial attention, which highlights different local time series produced by the same sensor based on their importance to the target series; and global spatial attention, which does the same for the time series produced by different surrounding sensors globally. The concatenation of the context vectors produced by these two attention mechanisms is then fed to a third temporal attention mechanism used on the decoder. The authors emphasised the importance of the two-fold spatial attention mechanism as opposed to a single input attention, as in the work by Qin et al. (2017). The latter treats both local and global time series as equal in its attention technique. The underlying attention scheme used for all three attention steps follows a Bahdanau-style additive scheme. Empirical evidence on two geo-sensory datasets showed that the GeoMAN model outperformed other state-of-the-art techniques such as S2S models, ARIMA, LSTM, and DA-RNN.

Apart from using individual RNN models, several researchers have also experimented using ensembles of RNN models for forecasting. Ensembles are meant for combining multiple weak learners together to generate a more robust prediction and thus reduce the individual training time of each base RNN. Smyl (2017) segmented the problem into two parts: developing a group of specialised RNN models, and ensembling them to derive a combined prediction. The author's methodology follows the idea that general clustering based on a standard metric for developing ensembles does not offer the best performance in the context of forecasting. Therefore, the approach randomly assigns time series of the dataset to one of the RNNs in the pool for training at the first epoch. Once all networks are trained, every time series in the dataset is assigned to a specific N number of best networks that give the minimum error after the training in that epoch. RNNs are trained in this manner iteratively using the newly allocated series until the validation error grows or the number of epochs finishes. Ensembling includes using another network to determine which RNN in the pool should make the forecasts for a given series, feeding the forecasts of each network to another network to make the final predictions, and using another network to determine the weights to be assigned to the forecasts of each RNN in the pool. Despite the complexity associated with these techniques, that author claimed that none of them work well. Simple techniques work better, such as the average, the weighted average (weights based on the training loss of each network or the rate of becoming a best network) of all the networks, and the weighted average of the N best networks. This methodology had encouraging results with a monthly series of the M3 competition dataset. The RNNs used for the pool comprised a stacked LSTM with skip connections. This technique was also used by the winning solution of Smyl (2020) at the M4 competition, which proves that the approach works well.

Krstanovic and Paulheim (2017) suggested that the higher the diversity of the base learners, the better the accuracy of the final ensemble. The authors developed an ensemble using the method known as stacking, where a number of base LSTM learners are combined using a meta-learner whose inputs are the outputs of base learners. The outputs of the meta-learner correspond to the final forecasts expected. The dissimilarity of the base learners is enforced by using a range of values for the hyperparameters, such as the dropout rate, number of hidden layers, number of nodes for the layers, and learning rate. For the meta-learner, the authors used ridge regression, extreme gradient boosting (XGBoost) and random forests. The method was compared to other popular techniques, such as LSTM, ensemble via mean forecast, the moving average, ARIMA, and XGBoost on four different datasets. The suggested ensemble with stacking performed best in general.

A modified boosting algorithm for RNNs in the context of time series forecasting was proposed by Assaad et al. (2008) based on the AdaBoost algorithm. The novelty of this approach is that the whole training set is considered for model training at each iteration by using a specialised parameter k, which controls the weight exerted on each training series based on its error in the previous iteration. Therefore, in every iteration, series that received a high error in the previous iteration are assigned more weight. A value of 0 for k assigns equal weights for all the time series in the training set. Furthermore, the base models in this approach are merged together using a weighted median that is more robust when encountered with outliers as opposed to a weighted mean. Experiments using two time series datasets showed promising

performance with this ensemble model on both single-step-ahead and multi-step-ahead forecasting problems, compared to other models in the literature, such as MLP and the threshold autoregressive (TAR) model.

2.4. Leveraging cross-series information

Exploiting cross-series information in forecasting is an idea that has received increased attention recently, especially in the aftermath of the M4 competition. The idea is that, instead of developing one model for each time series in a dataset, a model is developed by exploiting information from many time series simultaneously. In the literature, such models are often referred to as global models, whereas univariate models, which build a model for every series, are known as local models (Januschowski et al., 2020). However, the application of global models to a set of time series does not indicate any interdependence between them with respect to the forecasts. Rather, it means that the parameters are estimated globally for all the time series available (Januschowski et al., 2020). This is basically the scenario of multivariate forecasting, where the value of one time series is driven by other external time-varying variables. Such relationships are directly modelled in the forecast equations of the method. Yet, a global model trained across series usually works independently on the individual series in a univariate manner when producing forecasts.

In modern forecasting problems, the requirement is often to produce forecasts for many time series that may have similar patterns, as opposed to forecasting just one time series. One common example, in the domain of retail forecasts, is to produce forecasts for many similar products. In such scenarios, global models can demonstrate their true potential by learning across series to incorporate more information, in comparison to local methods. Trapero et al. (2015) used a similar idea in their work for demand forecasting of stock-keeping units (SKUs). In particular, for SKUs with limited or no promotional history associated with them, the coefficients of the regression model are calculated by pooling across many SKUs. However, the regression model used can capture only linear relationships and it does not maintain any internal state for each time series.

In recent literature, researchers have used the idea of developing global models in the context of deep neural networks. For RNNs, this means that the weights are calculated globally, yet the state is maintained for each time series. The winning solution by Smyl (2020) at the M4 forecasting competition used the global model concept with local parameters as well, to cater to the individual requirements of different series. Bandara et al. (2020) did this by clustering groups of related time series and developing a global model for each cluster. Salinas et al. (2019) also applied the idea of cross-series information in their DeepAR model for probabilistic forecasting. More recently, Oreshkin et al. (2019), Rangapuram et al. (2018), Wang et al. (2019) and Wen et al. (2017) employed a cross-series learning concept in their work using deep neural networks for forecasting. Bandara et al. (2019) developed global models for forecasting in an E-commerce environment by considering the sales demand patterns of similar products.

3. Methodology

In this section, we describe the details of the methodology employed in our comprehensive experimental study. We present the different recurrent unit types, the RNN architectures, and the learning algorithms that we implemented and compared in our work.

3.1. Recurrent neural networks

We implemented a number of RNN architectures along with different RNN units. In what follows, these are explained in detail.

3.1.1. Recurrent units

Among the different RNN units mentioned in the literature, we selected the following three types of recurrent units to constitute the layers of the RNNs in our experiments.

- Elman recurrent unit
- Gated recurrent unit
- Long short-term memory with peephole connections

The base recurrent unit was introduced by Elman (1990). The structure of the basic ERNN cell is shown in Fig. 1.

$$h_t = \sigma(W_i \cdot h_{t-1} + V_i \cdot x_t + b_i) \tag{2a}$$

$$z_t = tanh(W_o \cdot h_t + b_o) \tag{2b}$$

In Eqs. (2a) and (2b), $h_t \in \mathbb{R}^d$ denotes the hidden state of the RNN cell (where d is the cell dimension). This is the only form of memory in the ERNN cell. $x_t \in \mathbb{R}^m$ (where m is the size of the input) and $z_t \in \mathbb{R}^d$ respectively denote the input and output of the cell at time step t. $W_i \in \mathbb{R}^{d\times d}$ and $V_i \in \mathbb{R}^{d\times m}$ denote the weight matrices, whereas $b_i \in \mathbb{R}^d$ denotes the bias vector for the hidden state. Likewise, $W_o \in \mathbb{R}^{d\times d}$ and $b_o \in \mathbb{R}^d$ respectively signify the weight matrix and the bias vector of the cell output. The current hidden state depends on the hidden state of the previous time step as well as the current input. This is supported with the feedback loops in the RNN cell connecting its current state to the next state. These connections are of extreme importance in order to consider past information when updating the current cell state. In the experiments, we used the sigmoid function (indicated by σ) as the activation of the hidden state, and the hyperbolic tangent function (indicated by tanh) as the activation of the output.

The ERNN cell suffers from the well known vanishing gradient and exploding gradient problems over very long sequences. This implies that the simple RNN cells are not capable of carrying long-term dependencies to the future. When the sequences are quite long, the backpropagated gradients tend to diminish (vanish), and consequently the weights are not updated adequately. On the other hand, when the gradients are huge, they can burst (explode) over long sequences resulting in unstable weight matrices. Both these issues ensue from the gradients being intractable, and they hinder the ability of RNN cells to capture long-term dependencies.

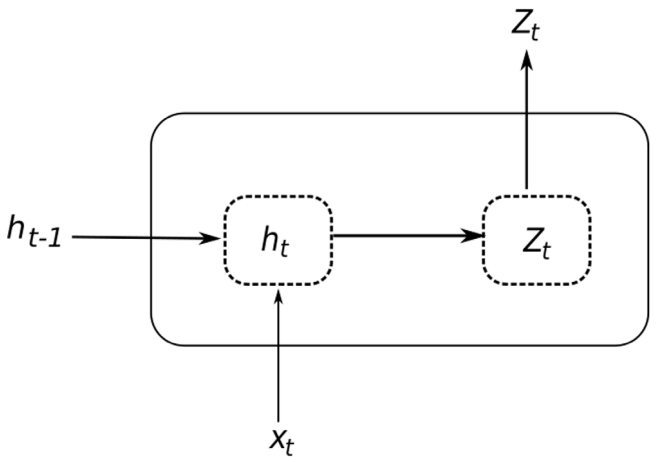

Fig. 1. Elman recurrent unit.

Over the years, several variations have been introduced to this base recurrent unit to address its shortcomings. The LSTM cell introduced by Hochreiter and Schmidhuber (1997) is perhaps the most popular cell for natural language processing tasks because it can capture long-term dependencies in the sequence while alleviating gradient vanishing issues. The structure of the LSTM cell is illustrated in Fig. 2

$$i_t = \sigma(W_i \cdot h_{t-1} + V_i \cdot x_t + b_i) \tag{3a}$$
$$o_t = \sigma(W_o \cdot h_{t-1} + V_o \cdot x_t + b_o) \tag{3b}$$
$$f_t = \sigma(W_f \cdot h_{t-1} + V_f \cdot x_t + b_f) \tag{3c}$$
$$\tilde{C}_t = tanh(W_c \cdot h_{t-1} + V_c \cdot x_t + b_c) \tag{3d}$$
$$C_t = i_t \odot \tilde{C}_t + f_t \odot C_{t-1} \tag{3e}$$
$$h_t = o_t \odot tanh(C_t) \tag{3f}$$
$$z_t = h_t \tag{3g}$$

Compared to the basic RNN cell, the LSTM cell has two components to its state: the hidden state, and the internal cell state. The hidden state corresponds to the short-term memory component, and the cell state corresponds to the long-term memory. With its constant error carrousel (CEC) capability supported by the internal state of the cells, LSTM avoids the vanishing and exploding gradient issues. Moreover, a gating mechanism is introduced which comprises the input, forget, and output gates. In Eqs. (3a)–(3g), $h_t \in \mathbb{R}^d$ is a vector that denotes the hidden state of the cell, where d is the cell dimension. Similarly, $C_t \in \mathbb{R}^d$ is the cell state and $\tilde{C}_t \in \mathbb{R}^d$ is the candidate cell state at time step t, which captures important information that will persist through to the future. $x_t \in \mathbb{R}^m$ and $z_t \in \mathbb{R}^d$ are the same as for the basic RNN cell. $W_i, W_o, W_f, W_c \in \mathbb{R}^{d\times d}$ denote the weight matrices of the input gate, output gate, forget gate, and cell state, respectively. Likewise, $V_i, V_o, V_f, V_c \in \mathbb{R}^{d\times m}$ and $b_i, b_o, b_f, b_c \in \mathbb{R}^d$ denote the weight matrices corresponding to the current input and the bias vectors, respectively. $i_t, o_t, f_t \in \mathbb{R}^d$ are the input, output, and forget gate vectors.

The activation function σ of the gates denotes the sigmoid function, which outputs values in the range [0, 1]. In Eq. (3e), the input and the forget gates together determine how much of the past information to retain in the current cell state and how much of the current context to propagate forward to the future time steps. $\odot$ denotes the element-wise multiplication, which is known as the Hudmard product. A value of 0 in forget gate f_t denotes that nothing should be carried forward from the previous cell state. In other words, the previous cell state should be completely forgotten in the current cell state. Following this argument, a value of 1 implies that the previous cell state should be completely retained. The same notion holds for the other two gates, i_t and o_t. A value in between the two extremes of 0 and 1 for both the input and forget gates can carefully control the value of the current cell state using only the information that is important from both the previous cell state and the current candidate cell state. For the candidate cell state, the activation function is a hyperbolic tangent function that outputs values in the range $[-1, 1]$. An important difference between the LSTM cell and a simple RNN cell is that the output z_t of the former is equal to the hidden state h_t.

In this review, we use a variant of the vanilla LSTM cell known as the LSTM cell with peephole connections. In this LSTM unit, the states are updated as follows:

$$i_t = \sigma(W_i \cdot h_{t-1} + V_i \cdot x_t + P_i \cdot C_{t-1} + b_i) \tag{4a}$$
$$o_t = \sigma(W_o \cdot h_{t-1} + V_o \cdot x_t + P_o \cdot C_t + b_o) \tag{4b}$$
$$f_t = \sigma(W_f \cdot h_{t-1} + V_f \cdot x_t + P_f \cdot C_{t-1} + b_f) \tag{4c}$$
$$\tilde{C}_t = tanh(W_c \cdot h_{t-1} + V_c \cdot x_t + b_c) \tag{4d}$$
$$C_t = i_t \odot \tilde{C}_t + f_t \odot C_{t-1} \tag{4e}$$
$$h_t = o_t \odot tanh(C_t) \tag{4f}$$
$$z_t = h_t \tag{4g}$$

The difference is that the peephole connections let the forget gate and the input gate of the cell look at the previous cell state C_{t-1} before updating it. In Eqs. (4a) to (4g), $P_i, P_o, P_f \in \mathbb{R}^{d\times d}$ represent the weight matrices of the input, output, and forget gates, respectively. As for the output gate, it can now inspect the current cell state for generating the output. The implementation used in

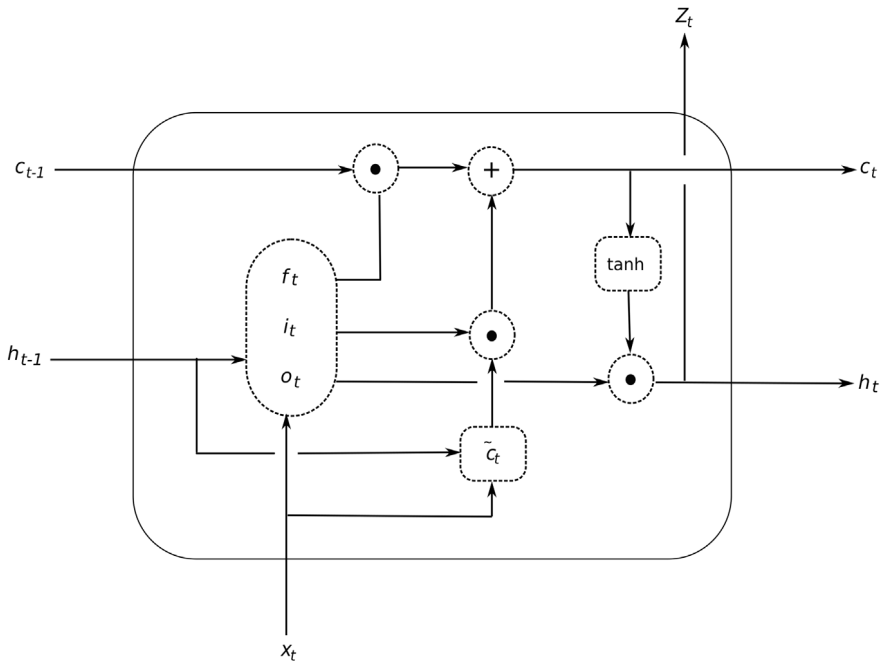

Fig. 2. Basic long short-term memory Unit.

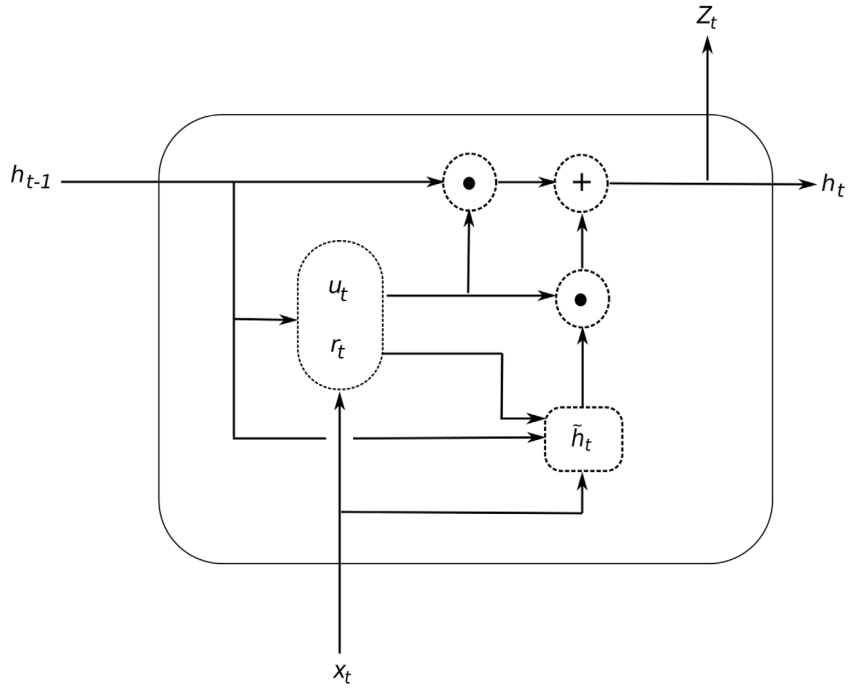

Fig. 3. Gated Recurrent Unit.

this research is the default implementation of peephole connections in the Tensorflow framework.

The GRU is another variant, introduced by Cho et al. (2014), that is comparatively simpler than the LSTM unit, as well as faster in computations. This is due to the LSTM unit having three gates within the internal gating mechanism, whereas the GRU has only two (viz. the update gate and the reset gate). The update gate in this unit plays the combined role of the forget gate and the input gate. Furthermore, similar to the vanilla RNN cell, the GRU cell has only one component to the state, i.e. the hidden state. Fig. 3 and the following equations display

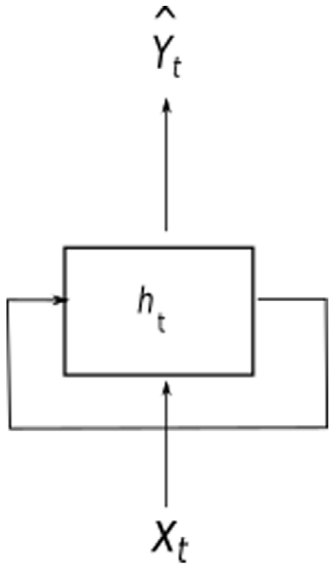

Fig. 4. Folded version of RNN.

the functionality of the GRU cell:

$$u_t = \sigma(W_u \cdot h_{t-1} + V_u \cdot x_t + b_u) \tag{5a}$$

$$r_t = \sigma(W_r \cdot h_{t-1} + V_r \cdot x_t + b_r) \tag{5b}$$

$$\tilde{h_t} = tanh(W_h \cdot r_t \cdot h_{t-1} + V_h \cdot x_t + b_h) \tag{5c}$$

$$h_t = u_t \odot \tilde{h_t} + (1 - u_t) \odot h_{t-1} \tag{5d}$$

$$z_t = h_t \tag{5e}$$

where $u_t, r_t \in \mathbb{R}^d$ denote the update and reset gates, respectively. $\tilde{h_t} \in \mathbb{R}^d$ indicates the candidate hidden state, and $h_t \in \mathbb{R}^d$ indicates the current hidden state at time step t. The weights and biases follow the same notation as mentioned above. The reset gate decides how much of the previous hidden state contributes to the candidate state of the current step. Since the update gate functions alone without a forget gate, $(1 - u_t)$ is used as an alternative. The GRU is popular, owing to its simplicity (i.e. it has fewer parameters than the LSTM cell) and because of its efficiency in training.

3.1.2. Recurrent neural network architectures

We used the following RNN architectures for our study.

3.1.2.1. Stacked architecture. The stacked architecture used in this study closely relates to the architecture mentioned in the work of Bandara et al. (2020). It is illustrated in Fig. 5.

Fig. 4 shows the folded version of the RNN, and Fig. 5 demonstrates the unfolded version through time. The idea is that the same RNN unit repeats for every time step, sharing the same weights and biases for each of them. The feedback loop of the cell helps the network to propagate the state h_t to the future time steps. For the purpose of generalisation, only the state h_t is shown in the diagram. However, for an LSTM cell, h_t should be accompanied by the cell state C_t. The notion of stacking means that multiple LSTM layers can be stacked on top of one another. In the most basic setup, the model has only one LSTM layer. Fig. 5 shows this basic case, whereas Fig. 6 shows a multi-layered scenario. As seen in Fig. 6, for many hidden layers, the same structure as shown in Fig. 5 is repeated multiple times stacked on top of each other. The output from every layer is directly fed as input to the next layer immediately above, and the final forecasts are retrieved from the last layer.

As seen in Fig. 5, X_t denotes the input to the cell at time step t and $\hat{Y}_t$ corresponds to the output. X_t and $\hat{Y}_t$ used for the stacking architecture are vectors instead of single data points. This is done according to the moving window scheme explained in Section 4.2.5, below. At every time step, the cell functions use their existing weights and produce the output Z_t, which corresponds to the next immediate output window of the time series. Likewise, the output of the RNN cell instance of the final time step $\hat{Y}_T$ corresponds to the expected forecasts for the particular time series. However, the output of the cell does not conform to the expected dimension, which is the forecasting horizon (H). Since the cell dimension (d) is an externally tuned hyperparameter, it may take on any appropriate value. Therefore, to project the output of the cell to the expected forecasting horizon, an affine neural layer is connected on top of every recurrent cell, whose weights are trained altogether with the recurrent network itself. In Fig. 5, this fully connected layer is not shown explicitly, and the output $\hat{Y}_t \in \mathbb{R}^H$ corresponds to the output of the combined RNN cell and the dense layer.

During the model training process, the error is calculated for each time step and accumulated until the end of the time series. Let the error for each time step t be e_t. Then,

$$e_t = Y_t - \hat{Y}_t \tag{6}$$

where Y_t is the actual output vector at time step t. This is preprocessed according to the moving window strategy. For all the time steps, the accumulated error E can be defined as follows.

$$E = \sum_{t=1}^{T} e_t \tag{7}$$

At the end of every time series, the accumulated error E is used for backpropagation through time (BPTT) once for the whole sequence. This BPTT then updates the weights and biases of the RNN cells according to the optimiser algorithm used.

3.1.2.2. Sequence-to-sequence architecture. The S2S architecture used in this study is illustrated in Fig. 7. There are a few major differences between the S2S network and the stacked architecture. The first is the input format. The input x_t fed to each cell instance of this network is a single data point instead of a vector. Stated differently, this network does not use the moving window scheme for data preprocessing. The RNN cells keep getting input at each time step and consequently build the state of the network. This component is known as the encoder. However, in contrast to the stacked architecture, the output is not considered for each time step; rather, only the forecasts produced after the last input point of the

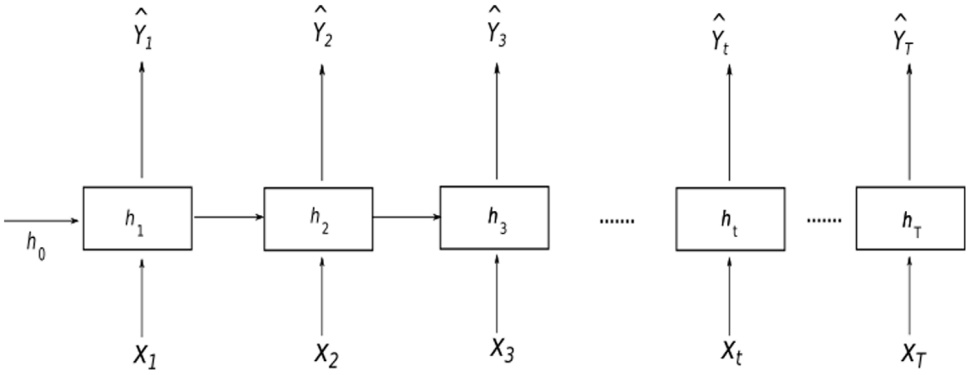

Fig. 5. Stacked architecture.

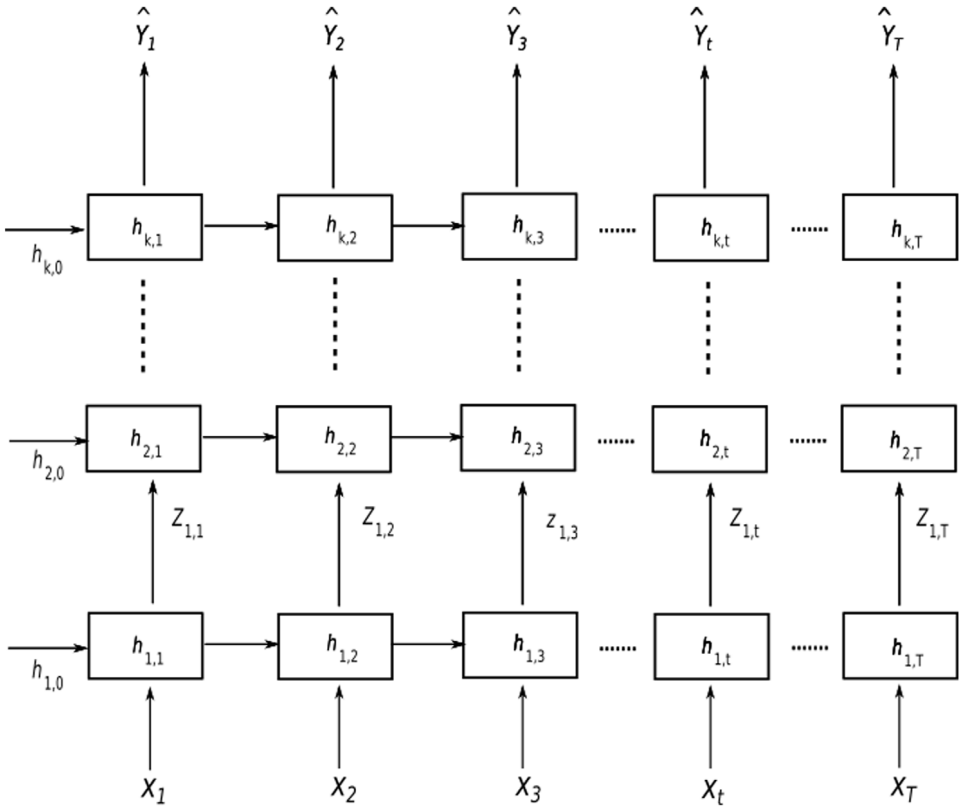

Fig. 6. Multi-layer stacked architecture.

encoder are considered. Here, every y_t corresponds to a single forecasted data point in the forecast horizon. The component that produces the outputs in this manner is called the decoder.

The decoder comprises a set of RNN cell instances as well, one for each step of the forecast horizon. The initial state of the decoder is the final state built from the encoder, which is also known as the context vector. A distinct feature of the decoder is that it contains autoregressive connections from the output of the previous time step into the input of the cell instance of the next time step. During training, these connections are disregarded, and the externally fed actual output of each previous time step is used as a means of teacher forcing. This teaches the decoder how inaccurate it is at predicting the previous output, and how much it should be corrected. During testing, since the actual targets are not available, the autoregressive connections are used instead as a substitute for the generated forecasts. This is known as scheduled sampling, where a decision is made to sample from either the outputs or the external inputs at the decoder. The decoder can also accept external inputs to support exogenous variables whose values are known for the future time steps. Since the hidden state size may not be equal to 1, which is the expected output size of each cell of the decoder, an affine neural layer is applied on top of

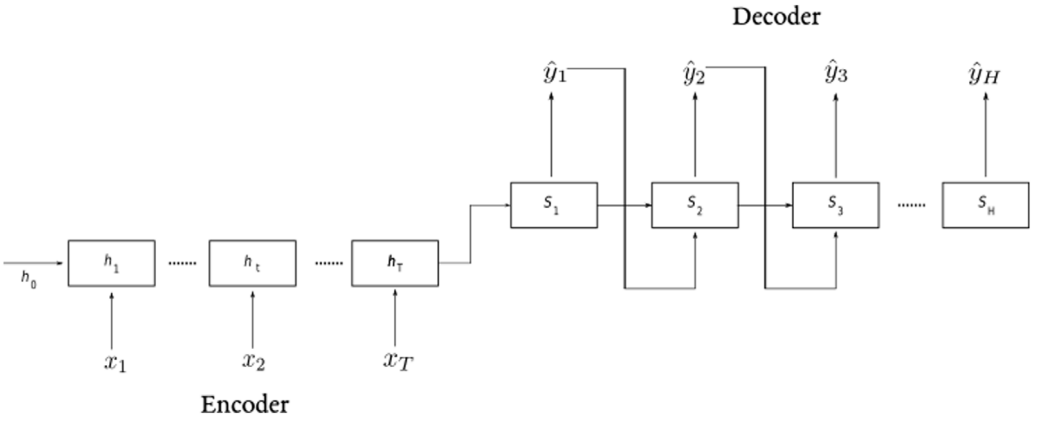

Fig. 7. Sequence-to-sequence with decoder architecture.

every cell instance of the decoder, similar to the stacked architecture. The error computed for backpropagation in the S2S architecture differs from that of the stacked architecture, since no error accumulation happens over the time steps of the encoder. Only the error at the decoder is considered for the loss function of the optimiser:

$$E = \sum_{t=1}^{H} y_t - \hat{y}_t \tag{8}$$

There are basically two types of components for the output in a S2S network. The most common is the decoder, as mentioned above. Inspired by the work of Wen et al. (2017), a dense layer can also be used in place of the decoder. We implemented both these possibilities in our experiments. However, since we did not feed any future information into the decoder, we did not use the concept of a local MLP, as in Wen et al. (2017). Rather, we used only one global MLP for the forecast horizon. This technique was expected to obviate the error propagation issue in the decoder with autoregressive connections. This model is illustrated in Fig. 8.

As seen in Fig. 8, the network no longer contains a decoder. Only the encoder exists to take the input for each time step. However, the format of the input in this model can be twofold: either with a moving window scheme or without it. Both of these were tested in our study. In the scheme with the moving window, each encoder cell receives the vector of inputs X_t, whereas without the moving window scheme, the input x_t corresponds to a single scalar input, as explained above regarding the network with the decoder. The forecast for both of these is simply the output of the last encoder time step projected to the desired forecast horizon using a dense layer without bias (i.e. a fully connected layer). Therefore, the output of the last encoder step is a vector either with or without a moving window. The error considered for the backpropagation is the error produced by this forecast.

$$E = Y_T - \hat{Y}_T \tag{9}$$

The stacked architecture, which is also fed with a moving window input, only differs from the S2S architecture with the dense layer and the moving window input format in that the former calculates the error for each time step and the latter calculates the error only for the last time step.

Table 1 shows the set of models selected for implementation by considering all the aforementioned architectures and input formats. All the models were implemented using the three RNN units (viz. the ERNN cell, LSTM cell, and GRU cell) and tested across the five datasets described in Section 4.1.

3.2. Learning algorithms

Three learning algorithms were tested in our framework: the Adam optimiser, the Adagrad optimiser, and the continuous coin betting (COCOB) optimiser (Duchi et al., 2011; Kingma & Ba, 2015; Orabona & Tommasi, 2017). Both Adam and Adagrad have built-in Tensorflow implementations, while the COCOB optimiser has an open-source implementation that uses Tensorflow (Orabona, 2017). The Adam optimiser and the Adagrad optimiser both require a hyperparameter learning rate, which, if poorly tuned, can perturb the whole learning process. The Adagrad optimiser introduces an adaptive learning rate, whereby a separate learning rate is kept for each variable of the function to be optimised. Consequently, the different weights are updated using separate equations. However, the Adagrad optimiser is prone to shrinking learning rates over time, as its equations for updating the learning rate have accumulating gradients in the denominator. This slows down the learning process of the optimiser over time. The Adam optimiser, similar to Adagrad, keeps one learning rate for each parameter. However, to address the issue of vanishing learning rates, the Adam optimiser uses both the exponentially decaying average of gradient moments and the gradients squared in its update equations. The Adam optimiser is generally expected to perform better than the other optimisers, as demonstrated empirically by Kingma and Ba (2015).

The COCOB optimiser attempts to minimise the loss function by self-tuning its learning rate. Therefore, this is one step closer to fully automating the NN modelling process, since the user is relieved from the burden of defining the initial learning rate ranges for the hyperparameter tuning algorithm. Learning algorithms are

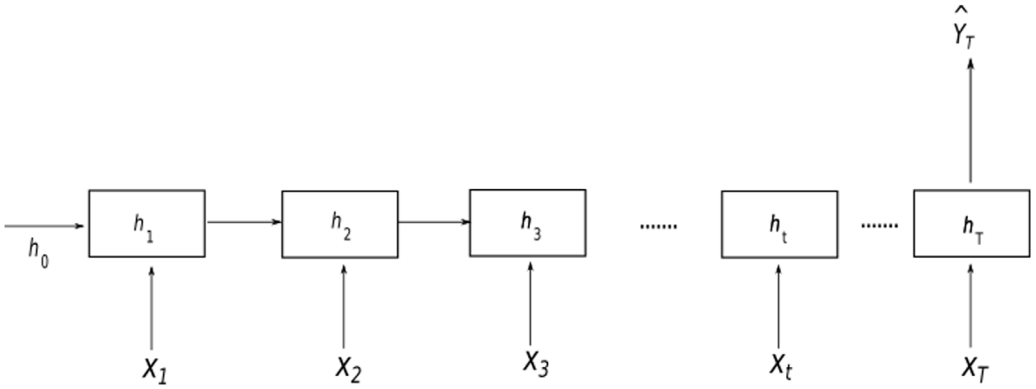

Fig. 8. Sequence-to-sequence with dense layer architecture.

Table 1
RNN architecture information.

Architecture	Output component	Input format	Error computation
Stacked	Dense layer	Moving window	Accumulated error
Sequence-to-sequence	Decoder	Without moving window	Last step error
Sequence-to-sequence	Dense layer	Without moving window	Last step error
Sequence-to-sequence	Dense layer	Moving window	Last step error

extremely sensitive to their learning rate. Therefore, finding the optimal learning rate is crucial for model performance. The COCOB algorithm is based on a coin betting scheme where, during each iteration, an amount of money is bet on the outcome of a coin toss such that the total wealth in possession is maximised. Orabona and Tommasi (2017) applied the same idea to function optimisations, where the bet corresponds to the size of the step taken along the axis of the independent variable. The total wealth and the outcome of the coin flip correspond to the optimum point of the function and the negative subgradient of the function at the bet point, respectively. During each round, a fraction of the current total wealth (optimum point) is bet. The betting strategy is designed such that the total wealth does not become negative at any point and the fraction of the money bet in each round increases until the outcome of the coin toss remains constant. In our context, this means that as long as the signs of the negative subgradient evaluations remain the same, the algorithm keeps making bigger steps along the same direction. This makes the convergence faster than other gradient descent algorithms that have a constant learning rate or a decaying learning rate, where the convergence becomes slower close to the optimum. Like the Adagrad optimiser, COCOB maintains separate coins (i.e. learning rates) for each parameter.

4. Experimental framework

To implement the models, we used version 1.12.0 of the Tensorflow open-source deep learning framework introduced by Abadi et al. (2015). This section describes the different datasets used for the experiments, their associated preprocessing steps, the model training and testing procedures, and the benchmarks used for comparison.

4.1. Datasets

The datasets used for the experiments were taken from the following forecasting competitions, held over the past few years.

- CIF 2016 Forecasting Competition Dataset
- NN5 Forecasting Competition Dataset
- M3 Forecasting Competition Dataset
- M4 Forecasting Competition Dataset
- Wikipedia Web Traffic Time Series Forecasting Competition Dataset
- Tourism Forecasting Competition Dataset

As mentioned above, our study was limited to using RNN architectures on univariate, multi-step-ahead forecasting with only single seasonality. This was done for the sake of a straightforward comparison with automatic standard benchmark methods. For the NN5 dataset and the Wikipedia Web Traffic dataset, since they contained daily data with less than two years of data, we considered only weekly seasonality. For the M3 and M4 datasets, we used only monthly time series that contained single-year seasonality. The NN5, Wikipedia Web Traffic, and Tourism datasets contained non-negative series, meaning that they also had 0 values. Table 2 gives further details about these datasets.

The CIF 2016 competition dataset had 72 monthly time series. Of these, 57 had a prediction horizon of 12, and the remaining 15 had a prediction horizon of 6 from the original competition. Some of the series with a prediction horizon of 6 were shorter than two full periods and were thus considered as having no seasonality. Out of all the time series, 48 series were artificially generated and the rest were real-time series originating from the banking domain (Štěpnička & Burda, 2017). The NN5 competition was held in 2008. This dataset had 111 daily time series,

Table 2
Dataset information.

Dataset name	No. of time series	Forecasting horizon	Frequency	Max. length	Min. length
CIF 2016	72	6, 12	Monthly	108	22
NN5	111	56	Daily	735	735
M3	1428	18	Monthly	126	48
M4	48,000	18	Monthly	2794	42
Wikipedia Web Traffic	997	59	Daily	550	550
Tourism	366	24	Monthly	309	67

which represent close to two years of daily cash withdrawal data from ATM machines in the UK (Ben Taieb et al., 2012). The forecasting horizon for all time series was 56. The NN5 dataset also contained missing values. The methods used to compensate for these issues are detailed in Section 4.2.2.

Two of the datasets were selected from the M competition series held over the years. From both the M3 and M4 datasets, we selected only the monthly category, which contained single-year seasonality. The yearly data contained no seasonality and the series were relatively shorter. In the quarterly data too, although they contained yearly seasonality, the series were shorter and contained fewer series compared to the monthly category. The M3 competition, held in 2000, comprised monthly time series with a prediction horizon of 18. In particular, this dataset consisted of time series from a number of different categories: micro, macro, industry, finance, demography, and other. The total number of time series in the monthly category was 1428 (Makridakis & Hibon, 2000). The M4 competition, held in 2018, had a dataset with a similar format to that of the M3 competition. The dataset had the same six categories as stated above, and the participants were required to make 18-months-ahead predictions for the monthly series. Compared to the previous competitions, one of the key objectives of the M4 was to increase the number of time series available for forecasting (Makridakis et al., 2018a). Consequently, the whole dataset consisted of 100,000 series and the monthly subcategory alone had 48,000 series.

The other two datasets that we selected for the experiments were both taken from Kaggle challenges: the Web Traffic Time Series Forecasting Competition for Wikipedia Articles, and the Tourism Dataset Competition (Athanasopoulos et al., 2010; Google, 2017). The task of the first competition was to predict the future web traffic (number of hits) of a given set of Wikipedia pages (145,000 articles), given their history of traffic over approximately two years. The dataset includes some metadata as well to denote whether the traffic came from a desktop, mobile device, spider, or all these sources. For this study, this problem was reduced to the history of the first 997 articles for the period of July 1, 2015 to December 31, 2016. The expected forecast horizon was from January 1, 2017 to February 28, 2017, covering 59 days. One important distinction in this dataset compared to the others is that all the values are integers. Since the NN outputs continuous values that are both positive and negative, to obtain the final forecasts, the values need to be rounded to the closest non-negative integer. The tourism dataset contained monthly, yearly, and quarterly time series. Again, for our experiments we selected the monthly category, which had 366 series in total. The requirement in this competition for the monthly category was to predict the next 24 months of the given series. However, the nature of the data was generically termed 'tourism-related' and no specific details were given about what values the individual time series held. The idea of the competition was to encourage the public to come up with new models that could beat the results initially published by Athanasopoulos et al. (2011) using the same tourism dataset.

Table 2 also gives details of the lengths of different time series of the datasets. The CIF 2016, M3, and Tourism datasets have relatively short time series (a maximum length of 108, 126, and 309, respectively). The NN5 and Wikipedia Web Traffic dataset time series are longer. In the M4 monthly dataset, the lengths of the series vary considerably, from 42 to 2794.

Fig. 9 shows violin plots of the seasonality strength of the different datasets. To extract the seasonality strength, we used the `tsfeatures` R package (Hyndman et al., 2019).

4.2. Data preprocessing

We applied a number of preprocessing steps in our work that we detail in this section. Most of these steps are closely related to the ideas presented in Bandara et al. (2020).

4.2.1. Dataset split

The training and validation datasets were separated in manner similar to that described by Bandara et al. (2020) and Suilin (2017). From each time series, we reserved a part from the end for validation with a length equal to the forecast horizon. This was for finding the optimal values of the hyperparameters using the automated techniques explained in Section 4.3.1. The remaining time series constituted the training data. This approach is illustrated in Fig. 10. We used a fixed origin mechanism for validation instead of a rolling origin scheme, because we wanted to replicate a usual competition setup. Also, given the fact that we implemented 36 RNN models and tested them across six datasets with thousands of time series for both training and evaluation, we deemed our results representative, even with a fixed origin evaluation. It would be extremely computationally challenging to perform a rolling origin evaluation.

However, as stated by Suilin (2017), this kind of split is problematic, since the last part of the sequence is not considered for training the model. The further away the test predictions are from the training set, the worse, because the underlying patterns may change during this last

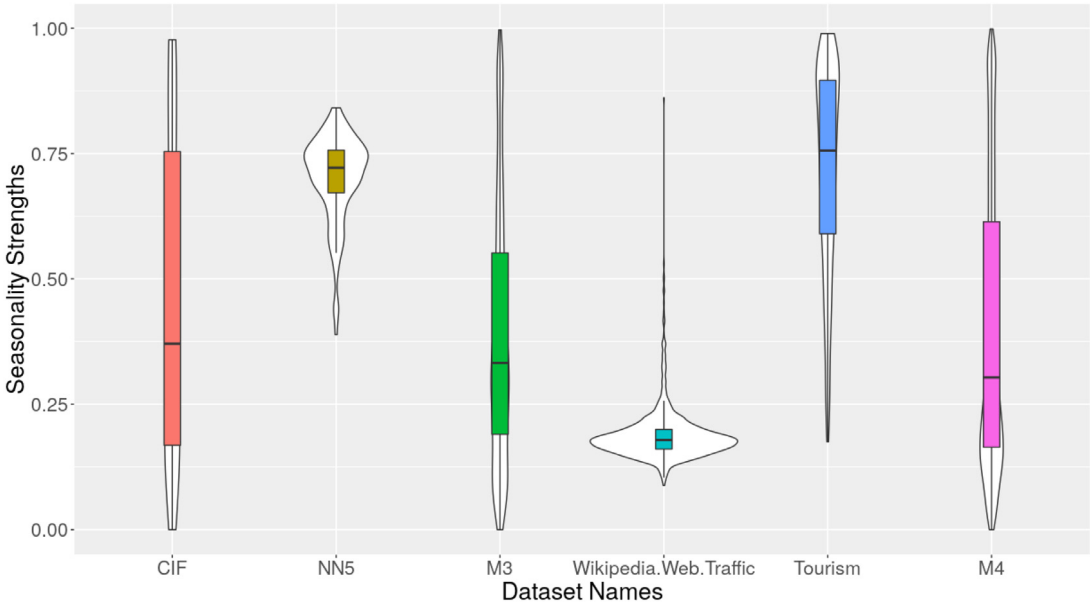

Fig. 9. Violin plots of seasonality strength. The NN5 and Tourism datasets have higher seasonality strength compared to the others. The inter-quantile ranges of the two respective violin plots are located comparatively higher along the *y* axis. Among these two, the seasonality strength of the individual NN5 series varies less from each other since the inter-quantile range is quite narrow. For the Wikipedia Web Traffic dataset, the inter-quantile range is even more narrow and located much lower along the *y* axis. This indicates that the time series of the Wikipedia Web Traffic dataset carry minimal seasonality compared to the other datasets. For the CIF, M3 monthly, and M4 monthly datasets, the seasonality strength of the different time series is spread across a wide spectrum.

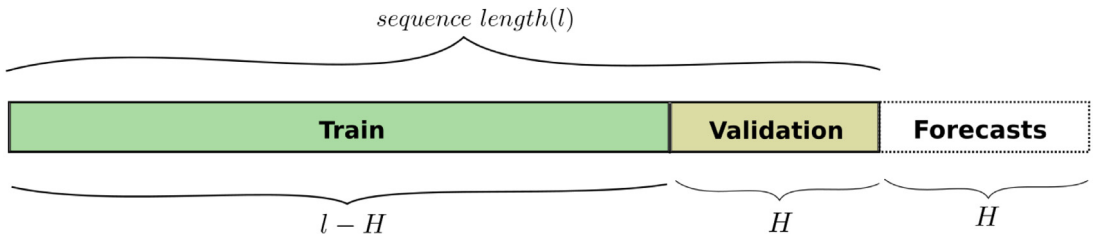

Fig. 10. Training validation set split.

part of the sequence. Therefore, in this work, the aforementioned split was used only for the validation phase. For testing, the model was re-trained using the whole sequence without any data split. For each dataset, the models were trained using all the time series available, for the purpose of developing a global model. Different RNN architectures were fed data in different formats.

4.2.2. Addressing missing values

Many machine learning methods cannot handle missing values. These need to be properly replaced by other appropriate substitutes. There are different techniques available to fill in missing values: e.g. mean substitution, median substitution, linear interpolation, and replacement by zero. Among the datasets selected for our experiments, the NN5 dataset and the Kaggle Web Traffic dataset contained missing values. For the NN5 dataset, we used a median substitution method. Since the NN5 dataset included daily data, a missing value on a particular day was replaced by the median across all the same days of the week along the whole series. For example, if a missing value was encountered on a Tuesday, the median of all values on Tuesdays from that time series was taken as the substitute. This approach seemed superior to taking the median across all the available data points, since the NN5 series had strong weekly seasonality. Compared to the NN5 dataset, the Kaggle Web Traffic dataset had many more missing values. In addition to that, this dataset did not differentiate between missing values and zero values. Therefore, for the Kaggle Web Traffic dataset, a simple substitution by zeros was carried out for all the missing values.

4.2.3. Modelling seasonality

With regard to modelling seasonality using NNs, there have been mixed notions among researchers over the years. Some of the early works inferred that NNs are capable of modelling seasonality accurately (Sharda & Patil, 1992; Tang et al., 1991). However, more recent experiments have suggested that deseasonalisation prior to feeding data to the NNs is essential, since NNs are weak at modelling seasonality. Particularly, Claveria et al.

(2017) empirically showed for a tourism-demand forecasting problem that seasonally adjusted data can boost the performance of NNs, especially in the case of long forecasting horizons. Zhang and Qi (2005) concluded that using both de-trending and deseasonalisation can improve the forecasting accuracy of NNs. Similar observations were recorded in Nelson et al. (1999) and Zhang and Kline (2007). More recently, the winning solution by Smyl (2020) at the M4 forecasting competition likewise proceeded on the argument that NNs are weak at modelling seasonality. A core part of our research was to investigate whether NNs actually struggle to model seasonality on their own. Therefore, we ran experiments with and without removing the seasonality. In the following, we describe the procedure we used when we removed the seasonality.

In theory, deterministic seasonality does not change, and is therewith known ahead of time. Thus, the NN can be relieved from the burden of modelling, easing its task by letting it predict only the non-deterministic parts of the time series. Following this general consensus, we ran a set of experiments with prior deseasonalisation of the time series data. We used STL decomposition, introduced by Cleveland et al. (1990), to decompose the time series into their seasonal, trend, and remainder components. Loess is the underlying method for estimating non-linear relationships in the data. By the application of a sequence of Loess smoothers, STL decomposition can efficiently separate the seasonality, trend, and remainder components of time series. However, this technique can only be used with additive trend and seasonality components. Therefore, in order to convert all multiplicative time series components to additive format, STL decomposition was immediately preceded by a variance stabilisation technique, as discussed in Section 4.2.4. STL decomposition is potentially able to allow the seasonality to change over time. However, we used STL in a deterministic way, where we assumed that the seasonality of all the time series was fixed along the whole time span.

In particular, we used the implementation of STL decomposition available in the forecast R package introduced by Hyndman and Khandakar (2008). By specifically setting the s.window parameter in the stl method to "periodic", we rendered the seasonality deterministic. Hence, we removed only the deterministic seasonality component from the time series. Any stochastic seasonality components remained. The NN was expected to model such stochastic seasonality by itself. STL decomposition was applied in this manner to all the time series, regardless of whether they actually showed seasonal behaviour. Nevertheless, the technique required at least two full periods of time series data to determine its seasonality component. In extreme cases where the full length of the series was less than two periods, the technique considered such sequences as having no seasonality, and returned a value of zero for the seasonality component.

4.2.4. Stabilising the variance in the data

Variance stabilisation is necessary if an additive seasonality decomposition technique such as the STL decomposition is used. For time series forecasting, variance stabilisation can be done in many ways, and applying a logarithmic transformation is arguably the most straightforward way to do so. However, a shortcoming of the logarithm is that it is undefined for negative and zero-valued inputs. All values need to be on the positive scale for the logarithm. For non-negative data, this can be easily resolved by defining a log transformation, as in Eq. (10), as a slightly altered version of a pure logarithm transformation:

$$w_t = \begin{cases} log(y_t) & \text{if } min(y) > \epsilon, \\ log(y_t + 1) & \text{if } min(y) \leq \epsilon \end{cases} \tag{10}$$

where y denotes the whole time series. For count data, ϵ can be defined as equal to 0, whereas for real-valued data, ϵ can be selected as a small positive value close to 0. The logarithm is a very strong transformation and may not always be adequate. There are some other similar transformations that attempt to overcome this shortcoming. One such transformation is the Box–Cox transformation. It is defined as follows:

$$w_t = \begin{cases} log(y_t) & \text{if } \lambda = 0, \\ (y_t^{\lambda} - 1)/\lambda & \text{if } \lambda \neq 0 \end{cases} \tag{11}$$

As indicated by Eq. (11), the Box–Cox transformation is a combination of a log transformation (when $\lambda = 0$) and a power transformation (when $\lambda \neq 1$), denoted by y_t^{λ}. The parameter λ needs to be carefully chosen. In the case when $\lambda = 1$, the Box–Cox transformation results in $y_t - 1$. Thus, the shape of the data is unchanged but the series is simply shifted down by 1 (Rob J Hyndman, 2018). Another similar form of power transformation is the Yeo–Johnson transformation (Yeo & Johnson, 2000), as shown in Eq. (12):

$$w_t = \begin{cases} ((y_t + 1)^{\mu} - 1)/\mu & \text{if } \mu \neq 0, y_t \geq 0, \\ log(y_t + 1) & \text{if } \mu = 0, y_t \geq 0, \\ -[(-y_t + 1)^{(2-\mu)} - 1]/(2 - \mu) & \text{if } \mu \neq 2, y_t < 0, \\ -log(-y_t + 1) & \text{if } \mu = 2, y_t < 0 \end{cases} \tag{12}$$

The parameter μ in the Yeo–Johnson transformation is confined in the range $[0, 2]$ while $\mu = 1$ gives the identity transformation. Values of y_t can be either positive, negative, or zero. Even though this is an advantage over the logarithm, the choice of the optimal value of the parameter μ is again not trivial.

Due to the complexities associated with selecting the optimal values of the parameters λ and μ in the Box–Cox and Yeo–Johnson transformations, and given that all data used in our experiments were non-negative, we used a log transformation in our experiments.

4.2.5. Multiple output strategy

Forecasting problems are typically multi-step-ahead forecasting problems, as they were in our experiments. Ben Taieb et al. (2012) performed extensive research involving five different techniques for multi-step-ahead forecasting. The recursive strategy, which is a sequence of one-step-ahead forecasts, involves feeding the prediction from the

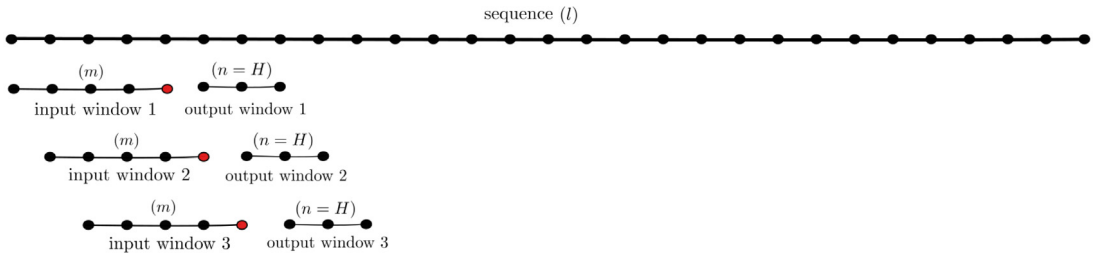

Fig. 11. Moving window scheme.

last time step as input for the next prediction. The direct strategy involves different models, one for each time step of the forecasting horizons. The DiRec strategy combines the direct and recursive strategies and develops multiple models, one for each forecasting step, where the prediction from each model is fed as input to the next consecutive model. In all these techniques, the model basically outputs a scalar value corresponding to one forecasting step. The other two techniques tested by Ben Taieb et al. (2012) use a direct multi-step-ahead forecast, where a vector of outputs corresponding to the whole forecasting horizon is produced by the model directly. The first technique under this category is known as the multi-input multi-output (MIMO) strategy. The advantage of the MIMO strategy comes from producing the forecasts for the whole output window at once, and thus incorporating the inter-dependencies between each time step, rather than forecasting each time step in isolation. The DIRMO strategy synthesises the ideas of the direct strategy and the MIMO strategy, where each model produces forecasts for windows of size s ($s \in \{1, \ldots, H\}$, where H is the forecast horizon). In the extreme cases where $s = 1$ and $s = H$, the technique narrows down to the direct strategy and the MIMO strategy, respectively. The experimental analysis by Ben Taieb et al. (2012) with the NN5 competition dataset demonstrated that the multiple output strategies are the best overall. The DIRMO strategy requires careful selection of the output window size s (Ben Taieb et al., 2012). Wen et al. (2017) also found that the MIMO strategy performs reliably, since it avoids error accumulation over the prediction time steps.

Following these early findings, we used a multiple output strategy for all the RNN architectures. However, the inputs and outputs of each recurrent cell differed between the different architectures. For the S2S model, each recurrent cell of the encoder was fed a single scalar input. For the stacking model and the S2S model with the dense layer output, a moving window scheme was used to feed the inputs and create the outputs, as in the work of Bandara et al. (2020). According to this approach, every recurrent cell accepts a window of inputs and produces a window of outputs corresponding to the time steps, which immediately follow the fed inputs. The recurrent cell instances of the next consecutive time step accept an input window of the same size as the previous cell, shifted forward by one. This method can also be regarded as an effective data augmentation mechanism (Smyl & Kuber, 2016). Fig. 11 illustrates the scheme.

Let the length of the whole sequence be l, the size of the input window be m, and the size of the output window be n. As mentioned in Section 4.2.1, during the training phase, the last n-sized piece from this sequence is left out for validation. The rest of the sequence (of length $l - n$) is broken down into blocks of size $m + n$ forming the input–output combination for each recurrent cell instance. Likewise in total, there are $l - n * 2 - m$ and $l - n - m$ such blocks for the training and the validation stages, respectively. Even though the validation stage does not involve explicitly training the model, the created blocks should still be fed in sequence to build up the state. The output window size is set to be equal to the size of the forecasting horizon H ($n = H$). The selection of the input window size is a careful task. The objective of using an input window as opposed to a single input is to relax the duty of the recurrent unit to remember the whole history of the sequence. Although theoretically, the RNN unit is expected to memorize all the information from the whole sequence it has seen, it is not practically competent at doing so (Smyl & Kuber, 2016). The importance of information from distant time steps tends to fade as the model continues to see new inputs. However, in terms of the trend, it is intelligible that only the last few time steps contribute the most to forecasting.

Putting these ideas together, we can select the input window size m in two ways. One is to make the input window size slightly bigger than the output window size ($m = 1.25 * output\_window\_size$). The other is to make the input window size slightly bigger than the seasonality period ($m = 1.25 * seasonality\_period$). For instance, if the time series has weekly seasonality ($seasonality\_period = 7$) with an expected forecasting horizon of 56, with the first option, we set the input window size to 70 ($1.25*56$), whereas with the second option, it is set to be 9 ($1.25*7$). The constant 1.25 is selected purely as a heuristic. The idea is to ascertain that every recurrent cell instance at each time step gets to see at least its last periodic cycle, such that it can model any remaining stochastic seasonality. In case the total length of the time series is too short, m is selected to be either of the two feasible, or some other possible value if none of them work. For instance, the forecasting horizon 6 subgroup of the CIF 2016 dataset comprises such short series, and m is selected to be 7 (slightly bigger than the output window size) even though the *seasonality_period* is equal to 12.

4.2.6. Trend normalisation

The activation functions used in RNN cells, such as the sigmoid and the hyperbolic tangent function, have a saturation area after which the outputs are constant.

Hence, when using RNN cells it should be assured that the inputs fed are normalised adequately, such that the outputs do not lie in the saturated range (Smyl & Kuber, 2016). To this end, we performed a per-window local normalisation step for the RNN architectures that used a moving window scheme. From each deseasonalised input and corresponding output window pair, the trend value of the last time step of the input window (found by applying STL decomposition) is deducted. This is carried out for all the input and output window pairs of the moving window strategy. The red coloured time steps in Fig. 11 correspond to those last time steps of all the input windows. This technique was inspired by the batch normalisation scheme, and it helps address the trend in time series (Ioffe & Szegedy, 2015). As for the models that did not use the moving window scheme, per-sequence normalisation was carried out, where the trend value of the last point of the whole training sequence was used for normalisation instead.

4.2.7. Mean normalisation

For the set of experiments that did not use STL decomposition, it was not possible to perform trend normalisation. For those experiments, we performed a mean normalisation of the time series before applying the log transformation described in Section 4.2.4. For mean normalisation, every time series was divided by the mean of that particular time series. The division instead of subtraction further helped to scale all the time series to a similar range, which helped the RNN learning process. Log transformation was then carried out on the resulting data.

4.3. Training & validation scheme

The methodology used for training the models is detailed in this section. We used the idea of global models, as described in Section 2.4, and developed one model for all the available time series. However, it is important to understand that the implemented techniques work well only in the case of related (similar/homogeneous) time series. If the individual time series are from heterogeneous domains, modelling such time series using a single model may not render the best results.

Due to the complexity and the total number of models associated with the study, we ran the experiments on three computing resources in parallel. The details of these machines are given in Table 3. Due to the scale of the M4 monthly dataset, GPU machines were used to run the experiments. Resource 2 and Resource 3 denote the resources allocated from the MASSIVE cluster (eResearch Centre, 2019). Resource 2 was used for GPU-only operations, whereas Resource 3 was used for CPU-only operations. Resource 1 was used for both CPU and GPU operations.

4.3.1. Hyper-parameter tuning

The experiments required properly tuning the following hyperparameters associated with model training:

1. Minibatch size
2. Number of epochs
3. Epoch size
4. Learning rate
5. Standard deviation of the Gaussian noise
6. L2 regularisation parameter
7. RNN cell dimension
8. Number of hidden layers
9. Standard deviation of the random normal initialiser

The minibatch size denotes the number of time series considered for each full backpropagation in the RNN. This is a more limited version than using all the available time series at once to perform one backpropagation, which poses a significant memory requirement. On the other hand, this could also be regarded as a more generalised version of the extreme case of using only a single time series for each full backpropagation, also known as stochastic gradient descent. Apart from the explicit regularisation schemes used in the RNN models, as discussed in Section 4.3.2, below, minibatch gradient descent also introduces some implicit regularisation to the deep learning models in specific problem scenarios such as classification (Soudry et al., 2018). An epoch denotes one full forward and backward pass through the whole dataset. Therefore, the number of epochs denotes how many such passes across the dataset were required for the optimal training of the RNN. Even within each epoch, the dataset is traversed a number of times, denoted by the epoch size. This is especially useful for datasets with a limited number of time series, in order to increase the number of datapoints available for training. This is because NNs are supposed to be trained best when the amount of data available is higher.

Of the three optimisers used in the experiments, the Adam optimiser and the Adagrad optimiser both required the appropriate tuning of the learning rate to converge to the optimal state of the network parameters quickly. To reduce the effect of model overfitting, two steps were taken: adding Gaussian noise to the input, and using L2 regularisation for the loss function (explained in Section 4.3.2, below). Both of these techniques required tuning hyperparameters: the standard deviation of the Gaussian noise distribution, and the L2 regularisation parameter. Furthermore, the weights of the RNN units were initialised using random samples drawn from a normal distribution whose standard deviation was tuned as another hyperparameter.

There are two other hyperparameters that directly related to the RNN network architecture: the number of hidden layers, and the cell dimension of each RNN cell. For simplicity, both the encoder and the decoder components of the S2S network were composed of the same number of hidden layers. Also, the same cell dimension was used for the RNN cells in both the encoder and the decoder. However, as explained in Section 3.1.1, above, the different recurrent unit types that we employed in this study—namely, the LSTM cell, ERNN cell, and GRU cell—have different numbers of trainable parameters for the same cell dimension: the LSTM has the most and the ERNN has the least number of parameters. Therefore, for a fair comparison, in other research communities such as natural language processing, it is common practice to consider the total number of trainable parameters in the different

Table 3
Hardware specifications.

Specification	Resource 1	Resource 2	Resource 3
CPU model name	Intel(R) Core(TM) i7–8700	Intel Xeon Gold 6150	Intel Xeon CPU E5-2680 v3
CPU architecture	x86_64	x86_64	x86_64
CPU cores	12	1	4
Memory (GB)	64	50	5
GPU model name	GP102 [GeForce GTX 1080 Ti]	nVidia Tesla V100	–
No. of GPUs	2	2	–

models compared (Collins et al., 2016; Jagannatha & Yu, 2016; Ji et al., 2016). This is to ensure that the capacities of all the compared models are the same, to clearly distinguish the performance gains achieved purely through the novelty introduced by the models. As our main aim was to compare software frameworks, we used the cell dimension, which is the natural hyperparameter to be tuned by practitioners, and additionally performed experiments using the number of trainable hyperparameters instead. For this, we selected the best model combination identified from the preceding experiments involving the cell dimension as a hyperparameter and ran it again, along with all three recurrent unit types, by letting the hyperparameter tuning algorithm choose the optimal number of trainable parameters as a hyperparameter. Hence, for this experiment, instead of setting the cell dimension directly, we set the same initial range for the number of trainable parameters across all the compared models with the three RNN units. The number of trainable parameters is directly proportional to the cell dimension, and the deep learning framework we used allowed us to set the cell dimension exclusively. Thus, we calculated the corresponding cell dimension from the number of parameters, and set this parameter accordingly.

For tuning hyperparameters, there are many different techniques available. The most naïve and fundamental approach is hand tuning, which requires intensive manual experimentation to find the best possible hyperparameters. However, our approach is more targeted towards a fully automated framework. To this end, a grid Search and random search are two of the most basic methods for automated hyperparameter tuning (Bergstra & Bengio, 2012). A grid search, as its name suggests, explores the space of a grid of different hyperparameter value combinations. For example, if there are two hyperparameters Γ and Θ, a set of values is specified for each hyperparameter, as in $(\gamma_1, \gamma_2, \gamma_3, \ldots)$ for Γ and $(\theta_1, \theta_2, \theta_3, \ldots)$ for Θ. Then, the grid search algorithm goes through each pair of hyperparameter values (γ, θ) that lie in the grid of Γ and Θ and uses them in the model to calculate the error on a held-out validation set. The hyperparameter combination that gives the minimum error is chosen as the optimal set of hyperparameter values. This is an exhaustive search through a grid of all possible hyperparameter values. In the random search algorithm, instead of the exact values, distributions for the hyperparameters are provided, from which the algorithm randomly samples values for each model evaluation. A maximum number of iterations is provided for the algorithm until the model evaluations are performed on the validation set and the optimal combination thus far is selected. There are other more sophisticated hyperparameter tuning techniques as well.

4.3.1.1. Bayesian optimisation. The methods discussed above evaluate an objective function (validation error) point-wise. This means retraining machine learning models from scratch, which is computationally intensive. By contrast, Bayesian optimisation involves limiting the number of such expensive objective function evaluations. The technique models the objective function using a Gaussian prior over all possible functions (Snoek et al., 2012). This probabilistic model embeds all prior assumptions regarding the objective function to be optimised. During every iteration, another instance of the hyperparameter space is selected to evaluate the objective function value. The decision of which point to select next depends on the minimisation/maximisation of a separate acquisition function. This is much cheaper to evaluate than the objective function itself. The acquisition function can take many forms, of which the implementation of Snoek et al. (2012) uses the expected improvement. The optimisation of the acquisition function considers all the previous objective function evaluations to decide the next evaluation point. Therefore, the Bayesian optimisation process makes smarter decisions for function evaluations compared to the aforementioned grid search and random search. Given an initial range of values for every hyperparameter and a defined number of iterations, the Bayesian optimisation method uses a set of prior known parameter configurations and thus attempts to find the optimal values for those hyperparameters for the given problem. There are many practical implementations and variants of the basic Bayesian optimisation technique, such as hyperopt, spearmint, and bayesian-optimization (Bergstra, 2012; Fernando, 2012; Snoek, 2012).

4.3.1.2. Sequential model-based algorithm configuration (SMAC). SMAC for hyperparameter tuning is a variant of Bayesian optimisation that was proposed by Hutter et al. (2011). This implementation is based on sequential model-based optimisation (SMBO). SMBO is a technique founded on the ideas of the Bayesian optimisation but uses a tree-structured Parzen estimator (TPE) for modelling the objective function, instead of the Gaussian prior. The TPE algorithm produces a set of candidate optimal values for the hyperparameters in each iteration, as opposed to just one candidate in the usual Bayesian optimisation. The tree structure lets the user define conditional hyperparameters that depend on one another. The process is carried out until a given time bound or a number of iterations is reached. The work by Hutter et al. (2011) further enhanced the basic SMBO algorithm by adding the capability to work with categorical hyperparameters. The Python implementation of SMAC used in our study is available as a Python package (Lindauer et al., 2017).

We used version 0.8.0 for our experiments. The initial hyperparameter ranges used for the NN models across the different datasets are listed in Table 4. Additionally, for the experiment that involved the number of trainable parameters instead of the cell dimension as a hyperparameter, we set an initial range of 2000–25,000 across all the compared models in all of the selected datasets. This range was selected based on initial experiments involving the cell dimension.

4.3.2. Dealing with model overfitting

Overfitting in a machine learning model is when the performance on the validation dataset is much worse than the performance on the training dataset. This means that the model has fitted quite well onto the training data to the extent that it has lost its generalisation capability to model other unseen data. In NN models, this mostly happens due to the complexity ensued from the large number of parameters. In order to make the models more versatile and capture unseen patterns, we used two techniques in our framework. One method involved adding noise to the input to distort it partially. The noise distribution applied was a Gaussian distribution with zero mean and with standard deviation treated as another hyperparameter. The other technique involved L2 weight regularisation, which uses a ridge penalty in the loss function. Weight regularisation avoids the weights of the network from growing excessively by incorporating them in the loss function and thus penalizing the network for increasing its complexity. Given the L2 regularisation parameter ψ, the loss function for the NN can be defined as follows:

$$L = E + \underbrace{\psi \sum_{i=1}^{p} w_i^2}_{\text{L2 regularization}} \tag{13}$$

Applied to our context, E in Eq. (13) denotes the mean absolute error from the network outputs, and w_i represents the trainable parameters of the network, where p is the number of all such parameters. Hence, as seen in Eq. (13), L2 regularisation adds the squared magnitudes of the weights multiplied by the regularisation parameter ψ. The selection of ψ is paramount, since too large a value results in underfitting, and very small values allow the model to become overfitted. In our study, ψ was also optimised as a hyperparameter.

4.4. Model testing

Once an optimal hyperparameter combination was found, those values were used to train the model and get the final forecasts. During testing, the model was trained using all the data available and the final forecasts were written to files. We addressed parameter uncertainty by training all the models on 10 different Tensorflow graph seeds using the same initially tuned hyperparameters. We did not re-run hyperparameter tuning, as this would be very computationally expensive. The different seeds provided 10 different initialisations to the networks. Once the RNN forecasts were obtained from every model for the different seeds, they were ensembled by taking the median across the seeds.

4.4.1. Data post-processing

To calculate the final error metrics, the effects of the prior preprocessing were properly reversed on the generated forecasts. For the experiments that used STL decomposition, this post-processing was carried out as follows.

1. Reverse the local normalisation by adding the trend value of the last input point.
2. Reverse deseasonalisation by adding back the seasonality components.
3. Reverse the log transformation by taking the exponential.
4. Subtract by one if the data contain zeros.
5. For integer data, round the forecasts to the closest integer.
6. Clip all negative values at zero (to allow for only positive values in the forecasts).

For those experiments that did not use STL Decomposition, the forecasts were post-processed as follows.

1. Reverse the log transformation by taking the exponential.
2. Subtract by one if the data contain zeros.
3. Multiply the forecasts of every series by its corresponding mean to reverse the effect of mean scaling.
4. For integer data, round the forecasts to the closest integer.
5. Clip all negative values at zero (to allow for only positive values in the forecasts).

4.4.2. Performance measures

We measured the performance of the models in terms of a number of metrics. The symmetric mean absolute percentage error (SMAPE) is the most common performance measure used in many forecasting competitions:

$$SMAPE = \frac{100\%}{H} \sum_{k=1}^{H} \frac{|F_k - Y_k|}{(|Y_k| + |F_k|)/2} \tag{14}$$

where H, F_k, and Y_k indicate the size of the horizon, the forecast of the NN, and the actual forecast, respectively. As seen in Eq. (14), the SMAPE is a metric based on percentage errors. However, according to Hyndman and Koehler (2006), SMAPE is susceptible to instability with values close to 0. The forecasts of the Wikipedia Web Traffic, NN5, and Tourism datasets were heavily affected by this, since they all had zero-values. Therefore, to overcome this, we used another variant of the SMAPE, proposed by Suilin (2017). In this metric, the denominator of the above SMAPE was changed as follows:

$$max(|Y_k| + |F_k| + \epsilon, 0.5 + \epsilon) \tag{15}$$

where the parameter ϵ was set to 0.1 following Suilin (2017). The above metric avoids division by values close to zero by switching to an alternate positive constant for the denominator in SMAPE when the forecast values are too small. Yet, the SMAPE error metric has several pitfalls, such as its lack of interpretability and high skewness (Hyndman & Koehler, 2006). Based on these issues, we used another metric, proposed by Hyndman

Table 4
Initial hyperparameter ranges.

Dataset	Batch size	Epochs	Epoch size	Std. noise	L2 reg.	Cell dim.	Layers	Std. initializer	Learning rate	
									Adam	Adagrad
CIF (12)	10–30	3–25	5–20	0.01–0.08	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
CIF (6)	2–5	3–30	5–15	0.0001–0.0008	0.0001–0.0008	20–50	1–5	0.0001–0.0008	0.001–0.1	0.01–0.9
NN5	5–15	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–25	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M3(Mic)	40–100	3–30	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M3(Mac)	30–70	3–30	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M3(Ind)	30–70	3–30	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M3(Dem)	20–60	3–30	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M3(Fin)	20–60	3–30	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M3(Oth)	10–30	3–30	5–20	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M4(Mic)	1000–1500	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M4(Mac)	1000–1500	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M4(Ind)	1000–1500	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M4(Dem)	850–1000	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M4(Fin)	1000–1500	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–50	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
M4(Oth)	50–60	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–25	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
Wikipedia	200–700	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–25	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9
Tourism	10–90	3–25	2–10	0.0001–0.0008	0.0001–0.0008	20–25	1–2	0.0001–0.0008	0.001–0.1	0.01–0.9

and Koehler (2006), known as the mean absolute scaled error (MASE). The MASE is defined as follows:

$$MASE = \frac{\frac{1}{H}\sum_{k=1}^{H}|F_k - Y_k|}{\frac{1}{T-M}\sum_{k=M+1}^{T}|Y_k - Y_{k-M}|} \tag{16}$$

The MASE is a scale-independent measure, where the numerator is the same as in SMAPE, but normalised by the average in-sample one-step naïve forecast error or the seasonal naïve forecast error in the case of seasonal data. A MASE value greater than 1 indicates that the performance of the tested model is worse on average than the naïve benchmark, and a value less than 1 denotes the opposite. Therefore, this error metric provides a direct indication of the performance of the model relative to the naïve benchmark.

The model evaluation of this study is presented using six metrics: the mean SMAPE, median SMAPE, mean MASE, median MASE, rank SMAPE, and rank MASE of the time series of each dataset. The rank measures compare the performance of every model to every other model with respect to every time series in the dataset. Apart from them, when considering all the datasets together, ranks of the models within each dataset with respect to both mean SMAPE and mean MASE metrics are plotted, since the original SMAPE and MASE values lie in very different ranges for the different datasets. These metrics are referred to as the mean SMAPE ranks and the mean MASE ranks, respectively.

Researchers have introduced several other performance measures for forecasting, such as the geometric mean relative absolute error (GMRAE) (Hyndman & Koehler, 2006). The GMRAE is also a relative error measure, similar to MASE, and thus measures performance with respect to some benchmark method, although the GMRAE uses the geometric mean instead of the mean. However, as Chen et al. (2017) state in their work, due to the usage of the geometric mean, values from the GMRAE can be quite small close to zero, especially in situations like ours, where there are many time series and the expected forecasting horizon is very long, as it is in the NN5 and the Wikipedia Web Traffic datasets.

4.5. Benchmarks

As for the benchmarks for comparison, we selected two strong, well established traditional univariate techniques: `ets` and `auto.arima`, with their default parameters from the `forecast` (Hyndman & Khandakar, 2008) package in R. We performed experiments with these two techniques on all the aforementioned datasets. We measured the performance in terms of the same metrics described in Section 4.4.2, above. Furthermore, we performed experiments with pooled regression, similar to the work of Trapero et al. (2015). As mentioned in Section 2.4, the pooled regression model differs from the RNN in that it does not maintain a state for every series, and it models a linear relationship between the lags and the target variable. However, similar to the RNNs, pooled regression models calculate their weights globally by considering cross-series information. That is, a pooled regression model works as a global AR model. Therefore, we used them in this study as a benchmark against RNNs to differentiate between the performance gains in RNNs purely from building a global model rather than due to the choice of RNN architecture. We also trained unpooled versions of regression models on all the datasets. As such, models were built for every series, similar to `ets` and `arima`. The unpooled regression models were used to observe the accuracy gains from training regression models as global models rather than usual univariate models.

To implement the regression models, we used the linear regression implementation from the `glmnet` package in the R programming language (Friedman et al., 2010). Before feeding the series into the pooled regression models, we normalised them by dividing every series by its mean value. For the number of lags, we experimented with two options. The `auto.arima` model from the `forecast` package selects by default up to a maximum of five lags for both the AR and MA components combined. An ARMA model can be approximated by a pure AR model with a higher number of lags (Rob J Hyndman, 2018). Since the pooled regression model is a pure AR model, to be approximately compatible with the complexity of the `auto.arima` model, we used 10 lags (as

opposed to five lags) as the first option for the number of lags in the pooled regression models. By contrast, the moving window scheme, described in Section 4.2.5, already imposes a number of lags on the input window of the relevant RNN architectures. Therefore, we also built versions of the regression models with this number of lagged values. Furthermore, as mentioned in Section 4.3.2, the RNN models were regularised using a ridge penalty. Hence, for a fair comparison, we developed regression models with and without L2 regularisation. To find the L2 regularisation parameter, we used two methods. We used the built-in grid search technique offered in the `glmnet` package using 10-fold cross-validation with a mean squared error as the validation error metric. However, to be more comparable to RNNs, we also used Bayesian optimisation hyperparameter tuning along with the SMAPE as the validation error metric to find the L2 regularisation parameter in the regression models. For this, we used a simple 70%–30% split of the training and validation sets, respectively. For the pooled regression models, the range for the L2 regularisation parameter was set to the 0–1 interval with 50 iterations in the tuning process. For the unpooled versions of the regression models, this initial range was set to the 0–200 interval. This is because, for one series there is a smaller amount of data available and the regularisation parameter may need a larger value to avoid overfitting. For Bayesian optimisation we used the `rBayesianOptimization` package in the R programming language (Yan, 2016). During testing, a recursive strategy was carried out to produce forecasts up to the forecasting horizon where one point was forecasted at every step, taking the last forecast as the most recent lag.

4.6. Statistical tests of the results

To evaluate the statistical significance of the differences in the performance of these techniques, we performed a non-parametric Friedman rank-sum test. This test determined whether the performance differences were statistically significant. To further explore these differences with respect to a control technique—specifically, the best performing technique—we then performed Hochberg's post hoc procedure (García et al., 2010). The significance level used was $\alpha = 0.05$. For tests comparing only two techniques, we used a non-parametric paired Wilcoxon signed-rank test with Bonferroni correction to measure the statistical significance of the differences. For this, we applied the `wilcox.test` function from the `stats` package of the R core libraries (R Core Team, 2014). The Bonferroni procedure is used to adjust the significance level used for comparison at each step throughout a number of successive comparisons (García et al., 2010). In particular, the procedure divides the significance level by the total number of comparisons for use in every individual comparison. The mean and the median of the SMAPE measure were used where relevant for all the statistical testing.

5. Analysis of results

This section provides a comprehensive analysis of the results obtained from the experiments. The datasets that we used for our experiments differed from each other in terms of the seasonality strength, number of time series, and length of individual time series. Thus, it seems reasonable to assume that the selected datasets covered a sufficient number of different time series characteristics to arrive at generalised conclusions. The results of all the implemented models in terms of the mean SMAPE metric are shown in Table 5. Results in terms of all the other error metrics are available in an Online Appendix.[2] In the tables presenting the results, we use the abbreviations 'NSTL', 'MW', and 'NMW' to denote 'without using STL decomposition', 'moving windows', and 'non-moving windows', respectively. Furthermore, 'IW+' indicates an increased input window size.

In Table 5, the best model in each dataset is indicated in boldface. Furthermore, as mentioned in Section 4.6, since we selected `auto.arima` and `ets` as the two established benchmarks, we performed paired Wilcoxon signed-rank tests with Bonferroni correction for every model against the two benchmarks. In Table 5, † and * denote models that performed significantly worse than `auto.arima` and `ets`, respectively. Similarly, ‡ and ** denote models that performed significantly better than `auto.arima` and `ets`, respectively. We see that on the CIF2016, M3, Tourism, and M4 datasets, no model was significantly better than the two benchmarks. On the NN5 dataset, the RNN models managed to perform significantly better than `auto.arima`, but not `ets`. Moreover, across all the datasets it can be seen that the RNN models that performed significantly better than the benchmarks were mostly variants of the stacked architecture. Also, on the Wikipedia Web Traffic dataset exclusively, the pooled versions of the regression models performed significantly better than the benchmark `auto.arima`.

In general, increasing the number of lags from 10 to the size of the input window of the RNNs improved the accuracy of the regression models across all datasets. On the Wikipedia Web Traffic, M3, and NN5 datasets, the inclusion of cross-series information alone had an effect, since the pooled regression models with an increased number of lags performed better than the unpooled versions of regression. Furthermore, on the Wikipedia Web Traffic and NN5 datasets, the pooled regression models with an increased number of lags outperformed the `auto.arima` model, which acts upon every series independently. Although the pooled models performed worse than the unpooled models on the CIF, Tourism, and M4 datasets, the RNN models mostly outperformed both the unpooled and pooled regression models on these datasets. This indicates that even though the series in those datasets were not sufficiently homogeneous to build global models, the use of RNN architectures had a clear effect on the final accuracy. In fact, on the CIF dataset, the effect from the choice of RNN architecture managed to outperform the two statistical benchmarks, `ets` and `auto.arima`. On the Wikipedia Web Traffic, M3, and NN5 datasets, moreover, the RNNs outperformed the statistical benchmarks and both versions of the regression models. This implies that, on these datasets, the cross-series inclusion together with the choice of the RNN led to better accuracy.

[2] https://doi.org/10.1016/j.ijforecast.2020.06.008.

Table 5
Mean SMAPE results

Model Name	CIF	Kaggle	M3	NN5	Tourism	M4
auto.arima	11.7	47.96	14.25	25.91	19.74	**13.08**
ets	11.88	53.5	14.14	21.57	**19.02**	13.53
Pooled Regression Lags 10	*14.67	*122.43†	*14.73†	*30.21†	*31.29†	*14.18†
Pooled Regression Lags 10 Bayesian Regularized	*14.67	*122.43†	*14.85†	*30.21†	*31.29†	*14.18†
Pooled Regression Lags 10 Regularized	*14.52	*123.94†	*14.75†	*30.21†	*31.29†	*14.19†
Pooled Regression Lags Window	*12.89	**47.47	*14.36†	*22.41	*21.1†	*13.75†
Pooled Regression Lags Window Bayesian Regularized	*12.89	**47.55	*14.42†	*22.41	*21.1†	*13.75†
Pooled Regression Lags Window Regularized	*12.89	**47.46	*14.38†	*22.41	*21.1†	*13.75†
Unpooled Regression Lags 10	15.35	*75.39†	14.81†	*30.06†	*27.39†	*13.38†
Unpooled Regression Lags 10 Bayesian Regularized	12.85	*84.36†	*15.99†	*30.21†	*30.41†	*14.2†
Unpooled Regression Lags 10 Regularized	*12.74	*94.28†	15.32†	*30.06†	*27.61†	*14†
Unpooled Regression Lags Window	14.34	*56.75†	14.93	*22.63	*20.71†	*13.46†
Unpooled Regression Lags Window Bayesian Regularized	12.46	*60.66†	*15.72†	*23.15	*21.58†	*14.15†
Unpooled Regression Lags Window Regularized	11.61	54.61†	14.72	*22.67	*21.13†	*13.65†
S2S GRU adagrad	11.57	51.77†	*15.91†	28.33	*20.57†	*13.53†
S2S GRU adam	11.66	49.9†	14.53†	*28.61	*20.62†	*14.52†
S2S GRU cocob	10.79	**46.89	*14.74†	28.36	*20.35†	*13.61†
S2S LSTM adagrad	11.23	51.77†	14.29†	28.34	*20.9†	*14.13†
S2S LSTM adam	10.71	**49.34‡	14.77†	*28.04	*20.5†	*13.99†
S2S LSTM cocob	11.04	52.46†	14.6	24.77	*20.51†	*14.02†
S2S ERNN adagrad	10.47	51.45†	*14.7†	*28.99	*20.44†	*14.24†
S2S ERNN adam	11.19	50.7†	*14.66†	*28.32	*20.63†	*13.86†
S2S ERNN cocob	10.48	**48.95†	*14.56†	*26.35	*20.77†	*13.8†
S2SD GRU MW adagrad	11.6	51.77†	**14.11**	28.73	19.7	*13.38†
S2SD GRU MW adam	10.66	51.76†	14.78†	*25.17	19.5	*13.79†
S2SD GRU MW cocob	10.3	48.8†	14.38†	*27.92	19.67	*13.57†
S2SD GRU NMW adagrad	10.65	*52.29†	*14.27†	28.73	*20.99†	*13.51†
S2SD GRU NMW adam	10.41	**46.9	14.35†	*25.42	*20.5†	*13.92†
S2SD GRU NMW cocob	10.48	**47.33	14.35†	24.8	*20.8†	*13.75†
S2SD LSTM MW adagrad	11.58	51.76†	14.19	28.73	*20.36†	*13.39†
S2SD LSTM MW adam	10.28	51.09†	14.68	*26	*20.18	*13.76†
S2SD LSTM MW cocob	11.23	51.77†	14.45	*24.29	19.76	*13.59†
S2SD LSTM NMW adagrad	10.81	*52.28†	*14.39†	28.73	*20.22†	*13.5†
S2SD LSTM NMW adam	10.34	**47.37	14.43†	25.36	*20.33†	*13.62†
S2SD LSTM NMW cocob	**10.23**	49.74†	14.31†	*27.82	*20.29†	*13.66†
S2SD ERNN MW adagrad	11.34	51.58†	14.2	28.58	19.58	*13.42†
S2SD ERNN MW adam	10.73	49.12†	14.75†	*27.49	19.69	*13.83†
S2SD ERNN MW cocob	10.39	**47.96†	14.73†	*25.58	19.44	*14.01†
S2SD ERNN NMW adagrad	10.31	49.72†	*14.86†	28.65	*21.27†	*13.52†
S2SD ERNN NMW adam	10.3	**47.76†	*14.79†	*29.1	20.04	*14.32†
S2SD ERNN NMW cocob	10.28	**47.94†	*14.94†	*26.37	19.68	*13.57†
Stacked GRU adagrad	10.54	**47.23	14.64	21.84‡	*21.56†	*14.15†
Stacked GRU adam	10.55	**47.11	14.76	21.93‡	*21.92†	*14.29†
Stacked GRU cocob	10.54	**46.73	14.67	22.18‡	*21.66†	*14.57†
Stacked LSTM adagrad	10.51	**47.12	14.34	22.12‡	*20.61†	*13.97†
Stacked LSTM adam	10.51	**47.13	14.39	**21.53**‡	*22.28†	*14.23†
Stacked LSTM cocob	10.4	**47.14	14.44	21.61‡	*21.12†	*14.13†
Stacked ERNN adagrad	10.53	**47.79	14.73	23.83‡	*21.24	*14.3†
Stacked ERNN adam	10.51	**46.7	14.71	23.53‡	*22.22†	*14.51†
Stacked ERNN cocob	10.57	**47.38	14.93	23.59‡	*21.72†	*14.24†
Stacked GRU adagrad(IW+)	-	**47.09	-	23.43	-	-
Stacked GRU adam(IW+)	-	**46.35	-	24.96	-	-

Stacked GRU cocob(IW+)	-	**46.46	-	21.92‡	-	-
Stacked LSTM adagrad(IW+)	-	**46.41	-	22.97	-	-
Stacked LSTM adam(IW+)	-	**46.52	-	21.59‡	-	-
Stacked LSTM cocob(IW+)	-	**46.54	-	21.54‡	-	-
Stacked ERNN adagrad(IW+)	-	**47.08	-	22.96	-	-
Stacked ERNN adam(IW+)	-	**46.76	-	22.5‡	-	-
Stacked ERNN cocob(IW+)	-	**47.59	-	23.13	-	-
NSTL S2S GRU adagrad	*20.02†	*68.6†	*18.32†	*30.98†	*199.97†	-
NSTL S2S GRU adam	*16.98†	50.48†	*18.52†	*28.09†	*48.2†	-
NSTL S2S GRU cocob	*17.96†	50.44†	*17.56†	*26.01	*95.19†	-
NSTL S2S LSTM adagrad	*17.54†	*71.25†	*17.75†	*29.52†	*41.41†	-
NSTL S2S LSTM adam	*16.11†	**47.8†	*17.37†	*36.33†	*32.69†	-
NSTL S2S LSTM cocob	*17.8†	*68.36†	*18.42†	*30.29†	*48.44†	-
NSTL S2S ERNN adagrad	*19.98†	51.4†	*19.73†	*26.33†	*49.83†	-
NSTL S2S ERNN adam	*18.12†	**49.84†	*19.06†	*29.81†	*51.24†	-
NSTL S2S ERNN cocob	*17.97†	51.69†	*18.85†	*31.77†	*43.68†	-
NSTL S2SD GRU MW adagrad	*16.12†	51.08†	*20.87†	*24.45	*199.97†	*19.47†
NSTL S2SD GRU MW adam	*13.74	51.2†	*18.09†	*24.08	*36.02†	-
NSTL S2SD GRU MW cocob	*17.77†	51.71†	*19.48†	*24.31	*37.91†	-
NSTL S2SD GRU NMW adagrad	*14.42	**46.57	*15.8†	*25.43	*199.97†	-
NSTL S2SD GRU NMW adam	*15.71†	**46.08	*16.17†	*23.85	*38.63†	-
NSTL S2SD GRU NMW cocob	*15.05†	**47.69†	*16.14†	*24.85	*31.57†	-
NSTL S2SD LSTM MW adagrad	*27.06†	51.24†	*21.29†	*40.96†	*40.27†	-
NSTL S2SD LSTM MW adam	*15.23	51.11†	*19.33†	*25.7	*38.52†	-
NSTL S2SD LSTM MW cocob	*18.61†	49.83†	*23.05†	*24.14	*24.34†	-
NSTL S2SD LSTM NMW adagrad	*14.81†	49.77†	*15.9†	*25.03	*25.47†	-
NSTL S2SD LSTM NMW adam	*14.84	**46.07	*17.05†	*24.55	*23.91†	-
NSTL S2SD LSTM NMW cocob	*22.74†	**47.72†	*16.35†	*25.31	*31.52†	-
NSTL S2SD ERNN MW adagrad	*19.19†	50.71†	*19.22†	*27.35†	*22.29†	-
NSTL S2SD ERNN MW adam	*13.92	49.69†	*18.55†	*24.52	*30.79†	-
NSTL S2SD ERNN MW cocob	*14.88†	50.82†	*16.42†	*24.87	*26.16†	-
NSTL S2SD ERNN NMW adagrad	*15.78†	**47.41	*17.22†	*27.47†	*38.19†	-
NSTL S2SD ERNN NMW adam	*15.02	**46.37	*16.34†	*22.81	*28.55†	-
NSTL S2SD ERNN NMW cocob	*14.24	48.27†	*15.26†	*23.22	*25.82†	-
NSTL Stacked GRU adagrad	*16.77†	**45.68‡	*16.39†	*24.39	*199.97†	-
NSTL Stacked GRU adam	*16.34†	****45.62**‡	*16.76†	*23.47	*38.66†	-
NSTL Stacked GRU cocob	*17.64†	**46.35	*16.23†	*23.55	*41.52†	-
NSTL Stacked LSTM adagrad	*17.08†	**45.88‡	*16.69†	*25.1	*23.46†	-
NSTL Stacked LSTM adam	*15.83†	**46.12	*17.54†	*24.44	*21.73†	-
NSTL Stacked LSTM cocob	*15.27†	**45.9	*17.97†	*26.28†	*25.24†	*17.71†
NSTL Stacked ERNN adagrad	*13.25	**46	*16.78†	*24.22	*25.36†	-
NSTL Stacked ERNN adam	*16.05†	**45.77‡	*16.43†	*23.04	*20.67	-
NSTL Stacked ERNN cocob	*15.74†	**46.02	*17.32†	*23.55	*28.71†	-
NSTL Stacked GRU adagrad(IW+)	-	**45.9	-	*23.53	-	-
NSTL Stacked GRU adam(IW+)	-	**45.69‡	-	22.2	-	-
NSTL Stacked GRU cocob(IW+)	-	**46.01	-	*22.78	-	-
NSTL Stacked LSTM adagrad(IW+)	-	**45.68‡	-	*23.15	-	-
NSTL Stacked LSTM adam(IW+)	-	**46.32‡	-	22.45	-	-
NSTL Stacked LSTM cocob(IW+)	-	**46.12	-	22.66	-	-
NSTL Stacked ERNN adagrad(IW+)	-	**45.69‡	-	*22.79	-	-
NSTL Stacked ERNN adam(IW+)	-	**45.92‡	-	22.13	-	-
NSTL Stacked ERNN cocob(IW+)	-	**45.79‡	-	*22.8	-	-

Another significant observation is that, on many datasets, both Bayesian optimisation and hyperparameter tuning with 10-fold cross validation for the L2 regularisation parameter of the pooled regression models selected values close to 0, such that the models had effectively no L2 regularisation. This is why, in many datasets, the mean SMAPE values are the same both with and without regularisation. Therefore, for the pooled regression models, we cannot conclude that L2 weight regularisation improves model performance. The model without any L2 regularisation was already an appropriate fit for the data. For the unpooled regression models, we obtained mixed results. As discussed by Bergmeir et al. (2018), although applicable to the detection of overfitting, a generic k-fold cross-validation may lead to an underestimation of the cross-validation errors for models that underfit. However, our pooled regression models did not show this behaviour, as they had very small L2 regularisation parameters. As for the unpooled models, although for some series the L2 regularisation parameters were high, the same case obtained for both 10-fold cross-validation and the normal cross-validation.

5.1. Relative performance of RNN architectures

We compared the relative performance of the different RNN architectures against each other across all datasets and in terms of all error metrics. The violin plots in Fig. 12 illustrate these results.

We see that the best architecture differed based on the error metric. In terms of the mean SMAPE, the S2S architecture with a dense layer (S2SD) and without moving windows performed the best. However, on all the other error metrics, the stacked architecture performed the best. With respect to the ranked error plots, the stacked architecture produced the best results for most time series. Thus, the results of the mean SMAPE indicate that the stacked architecture produced higher errors for certain outlying time series. The two versions of the S2SD architectures (with and without moving windows) performed comparably well, although not as well as the stacked architecture in most cases. Therefore, it is difficult to determine which of the two versions of the S2SD architecture was better. The S2S architecture (with the decoder) performed the worst overall.

With respect to the mean SMAPE values, the Friedman test of statistical significance gave an overall p-value of 9.3568×10^{-3}, implying that the differences were statistically significant. The Hochberg's post-hoc procedure was performed by using the S2SD MW architecture as the control method, which performed the best. The stacked and S2SD non-moving windows architectures did not perform significantly worse, with an adjusted p-value of 0.602. However, the S2S architecture performed significantly worse, with an adjusted p-value of 5.24×10^{-3}. In terms of the median SMAPE, the Friedman test of statistical significance gave an overall p-value of 0.349, implying that the differences in the models were not statistically significant.

5.2. Performance of recurrent units

The violin plots in Fig. 13 compare the performance of the different RNN units in terms of mean SMAPE ranks and mean MASE ranks. The Friedman test of statistical significance with respect to the mean SMAPE values produced an overall p-value of 0.101. Although this p-value did not indicate statistical significance, from the plots, we can infer that the LSTM cell with peephole connections performed the best. The ERNN cell performed the worst, and the GRU exhibited relatively moderate performance.

5.3. Performance of optimisers

The violin plots in Fig. 14 compare the performance of the different optimisers, in terms of both the mean SMAPE ranks and the mean MASE ranks. From the plots we see that the Adagrad optimiser performed the worst, consistent with the findings in the literature stated in Section 3.2. Even though the Adam optimiser is the best optimiser thus far, our results indicated that the COCOB optimiser performed the best out of the three. However, the Adam optimiser also showed competitive performance in between these two.

The Friedman test of statistical significance with respect to the mean SMAPE values gave an overall p-value of 0.115. Although there was no strong statistical evidence in terms of significance, we conclude that the Cocob optimiser is preferable to the other two, since it does not require setting an initial learning rate. This supports fully automating the forecasting approach, since it eliminates the tuning of one more external hyperparameter.

5.4. Performance of the output components for the sequence to sequence architecture

Fig. 15 shows the relative performance of the two output components for the S2S architecture. The dense layer performed better than the decoder, owing to the error accumulation issue associated with teacher forcing by the decoder, as mentioned in Section 2.3.1. With teacher forcing, the autoregressive connections in the decoder tend to carry forward the errors generated from each forecasting step, resulting in even more forecast uncertainty along the prediction horizon. The output from the paired Wilcoxon signed-rank test with respect to the mean SMAPE values gave an overall p-value of 2.064×10^{-4}, which indicates that the decoder performed significantly worse than the dense layer.

5.5. Comparison of input window sizes for the stacked architecture

Fig. 16 compares the two input window size options for the stacked architecture on the two daily datasets, NN5 and Wikipedia Web Traffic. The results are plotted both with and without STL decomposition. Here, 'Large' denotes an input window size slightly larger than the expected prediction horizon, and 'Small' denotes an input window size slightly larger than the seasonality period, which is 7 (for the daily data). From the plots, we can see

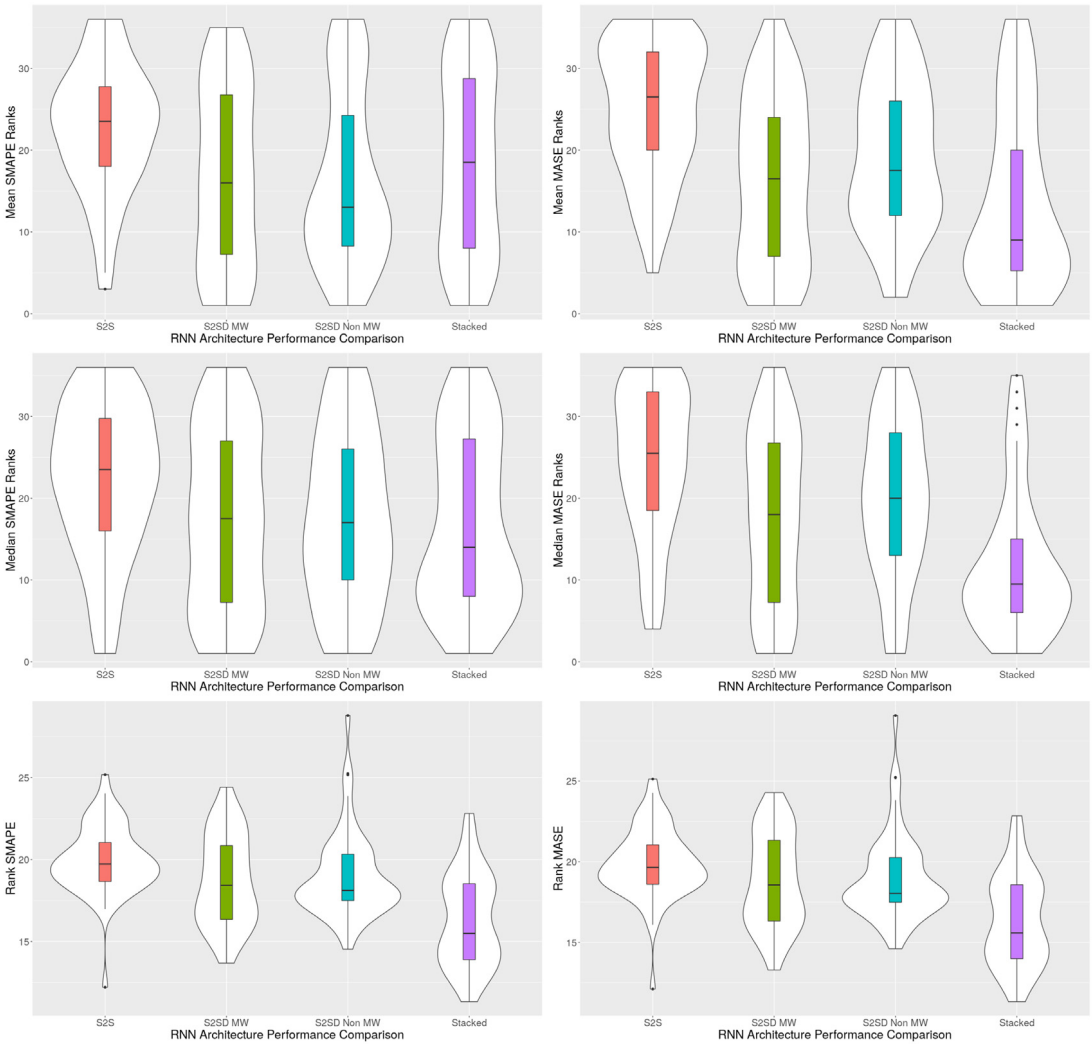

Fig. 12. Relative performance of different RNN architectures.

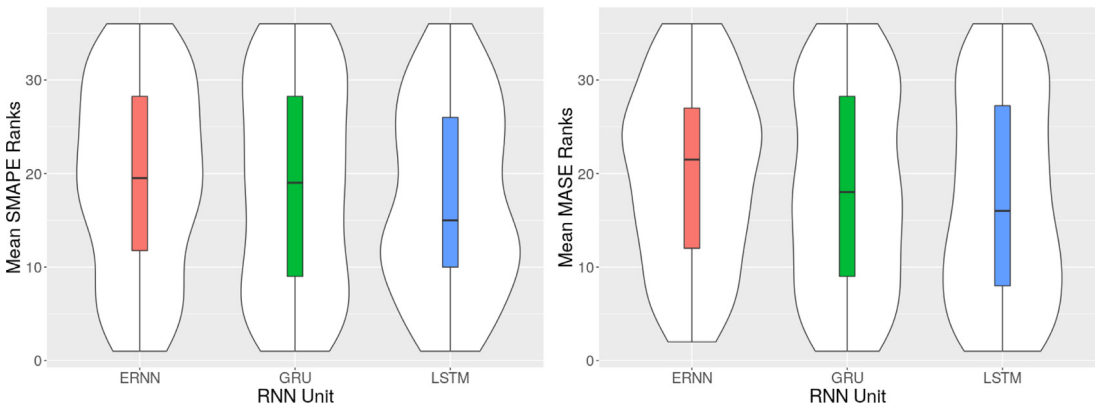

Fig. 13. Relative performance of different recurrent cell types.

that large input window sizes helped the stacked architecture both when STL decomposition was used and when it was not. However, with mean MASE ranks, small input window sizes performed comparably. For the case when the seasonality was not removed, larger input window sizes improved the accuracy by a considerable margin.

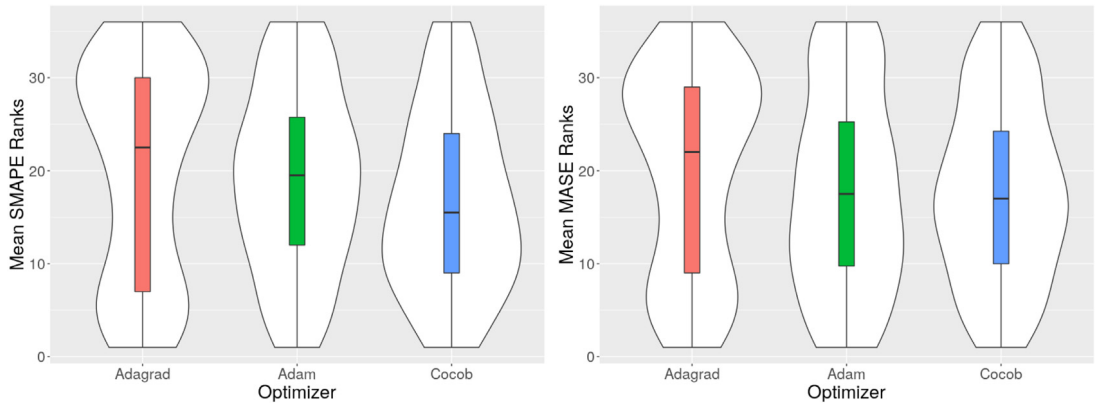

Fig. 14. Relative performance of different optimisers.

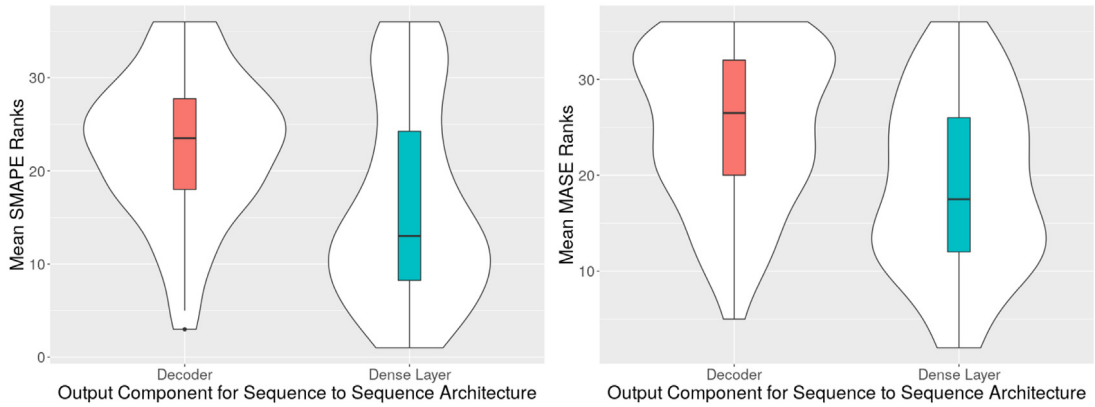

Fig. 15. Comparison of the output component for the sequence-to-sequence architecture with a dense layer.

Thus, large input windows make it easier for the stacked architecture to learn the underlying seasonal patterns in the time series. The output from the paired Wilcoxon signed-rank test with respect to the mean SMAPE values gave an overall p-value of 2.206×10^{-3}. This indicates that smaller input window sizes perform significantly worse than larger ones with and without STL decomposition.

5.6. Analysis of seasonality modelling

The results from comparing the models with and without STL decomposition are illustrated in Figs. 17 and 18, for the mean SMAPE metric and the mean MASE metric, respectively.

Due to the scale of the M4 monthly dataset, we did not run all the models without removing seasonality on that dataset. Rather, only the best few models were selected, based on the results from the first stage with removed seasonality. Consequently, we do not plot these results here. The plots indicated that on the CIF, M3, and Tourism datasets, removing seasonality worked better than modelling seasonality with the NN itself. Likewise, on the M4 monthly dataset, the same observation holds when looking at the results in Table 5. However, on the Wikipedia Web Traffic dataset, except for a few outliers, modelling seasonality using the NN itself worked almost as well as removing the seasonality beforehand. This can be attributed to the fact that in the Wikipedia Web Traffic dataset, all the series have minimal seasonality according to the violin plots in Fig. 9. As such, the data were similar whether or not the seasonality was removed. On the NN5 dataset as well, the NNs were able to model seasonality on their own, except for a few outliers.

To further explain this result, Fig. 19 plots the seasonal patterns of the different datasets. For the convenience of illustration, in every category of the M4 monthly dataset and the Wikipedia Web Traffic dataset, only the first 400 series are plotted. Furthermore, from every series of every dataset, only the first 50 time steps are plotted. Additional seasonal pattern plots for the different categories of the CIF, M3 monthly, and M4 monthly datasets are available in the Online Appendix.[3]

In the NN5 dataset, almost all the series had the same seasonal patterns as in the 4th plot of Fig. 19, with all of them having the same length as indicated in Table 2. All of the series in the NN5 dataset started and ended on the same dates (Crone, 2008). Also, according to Fig. 9, the NN5 dataset had higher seasonality, with the seasonality strengths and patterns showing very little variation among the individual series. By contrast, for the Tourism

[3] https://doi.org/10.1016/j.ijforecast.2020.06.008.

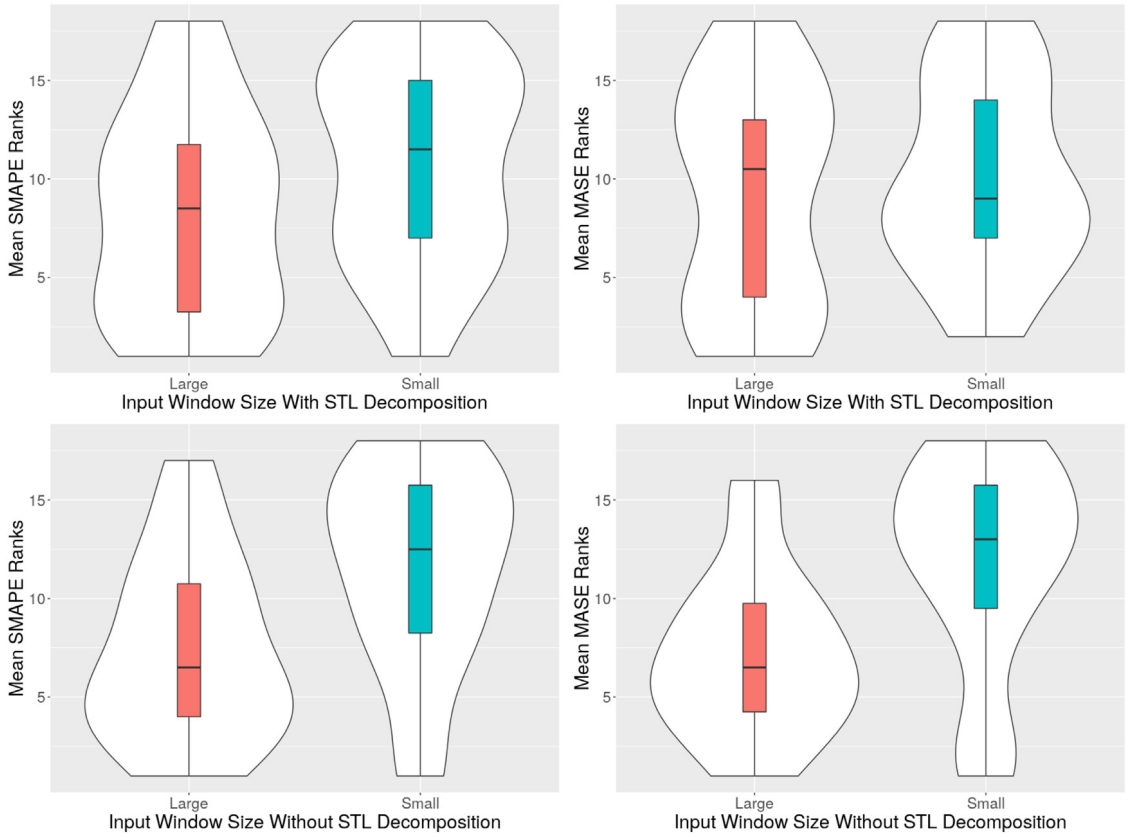

Fig. 16. Comparison of input window sizes for the stacked architecture.

dataset, although it contained series with higher seasonality, it showed high variation in the individual seasonality strength and patterns as well as in the lengths of the series. The starting and ending dates differed among the series. Looking at the 5th plot of Fig. 19, it is evident that the series had very different seasonal patterns. For the rest of the datasets as well, the lengths and the seasonal patterns of the individual series varied considerably. Therefore, from these observations we can conclude that NNs are capable of modelling seasonality on their own, when all the series in the dataset have similar seasonal patterns and when the lengths of the time series are equal, with the start and the end dates coinciding.

Table 6 shows rankings and the overall *p*-values obtained from the paired Wilcoxon signed-rank tests for comparing the cases with and without STL decomposition in every dataset. The overall *p*-values were calculated for each dataset separately by comparing cases with and without STL decomposition for all the available models.

5.7. Performance of RNN models vs. traditional univariate benchmarks

Figs. 20 and 21 show the relative performance of the model types in terms of the mean SMAPE metric and median SMAPE metric, respectively. More comparison plots in terms of the mean MASE and median MASE metrics are available in the Online Appendix.[4] As a representative suggested model combination based on our results, we identified the stacked architecture with LSTM cells and peephole connections optimised with COCOB. Therefore, this are indicated in every plot as 'Stacked_LSTM_COCOB' (with small and large input window sizes). In cases where the best RNN on each dataset differs from the 'Stacked_LSTM_COCOB', these best models are also shown in the plots. All the other RNNs are indicated as 'RNN'.

The performance of the RNN architectures compared to traditional univariate benchmarks depended on the performance metric used. An RNN architecture was able to outperform the benchmark techniques on all the datasets except for the M4 monthly dataset, in terms of both the SMAPE and MASE error metrics. On the M4 monthly dataset, some RNNs outperformed ETS, but ARIMA performed better than all of the RNNs. However, since we built models for each category of the M4 monthly dataset, we further plotted the performance in terms of the different categories, as shown in Fig. 22. From these plots, we can see that the RNNs outperformed the traditional univariate benchmarks only in the Micro category. Further plots in terms of the other error metrics are shown in the Online Appendix.[5]

[4] https://doi.org/10.1016/j.ijforecast.2020.06.008.
[5] https://doi.org/10.1016/j.ijforecast.2020.06.008.

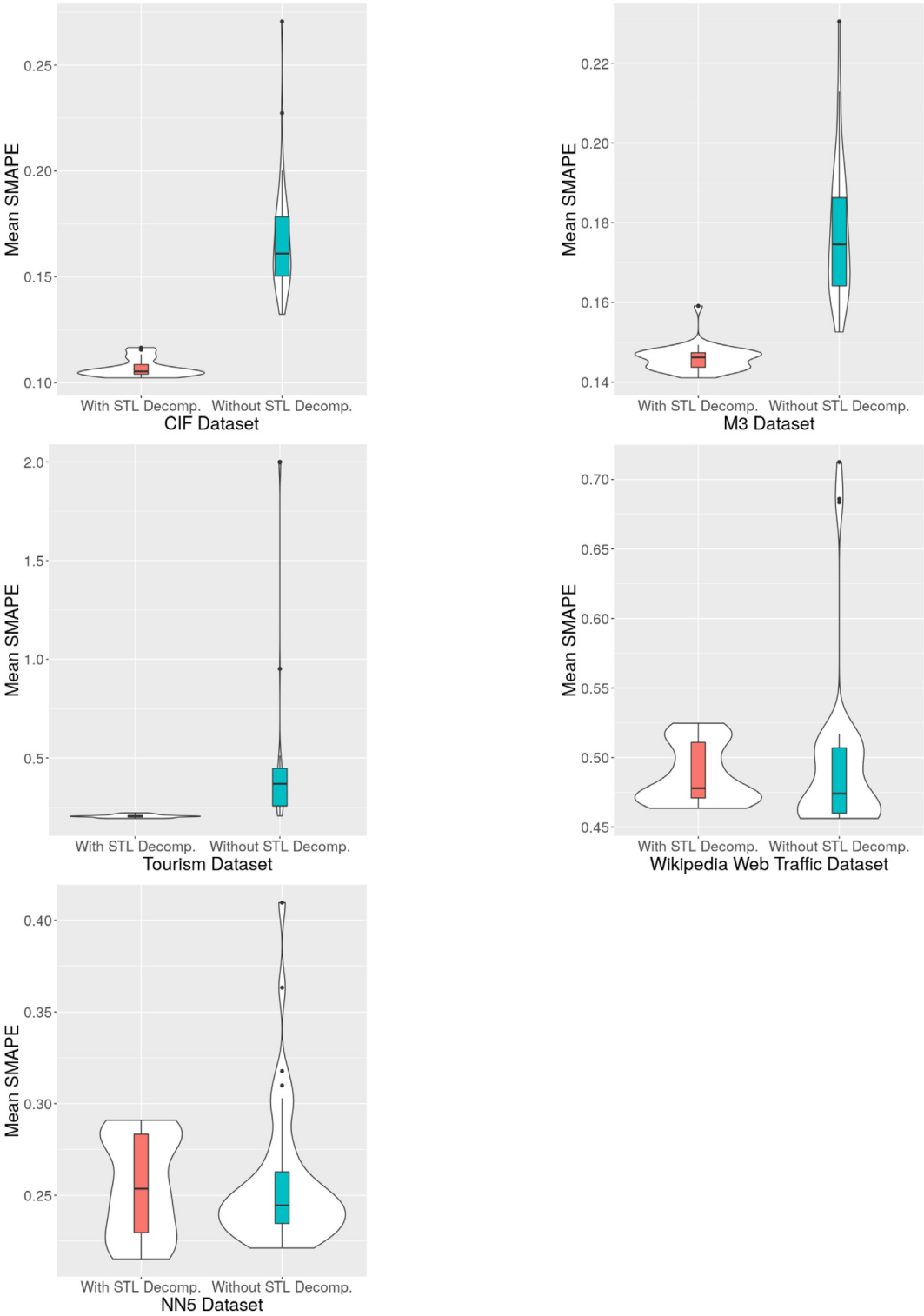

Fig. 17. Comparison of performance with and without STL decomposition: mean SMAPE.

We also observed that mean error metrics were dominated by some outlier errors for certain time series. This was because, on the Tourism dataset, the RNNs outperformed ETS and ARIMA with respect to the median SMAPE, but not with respect to the mean SMAPE. Furthermore, the 'Stacked_LSTM_COCOB' model combination

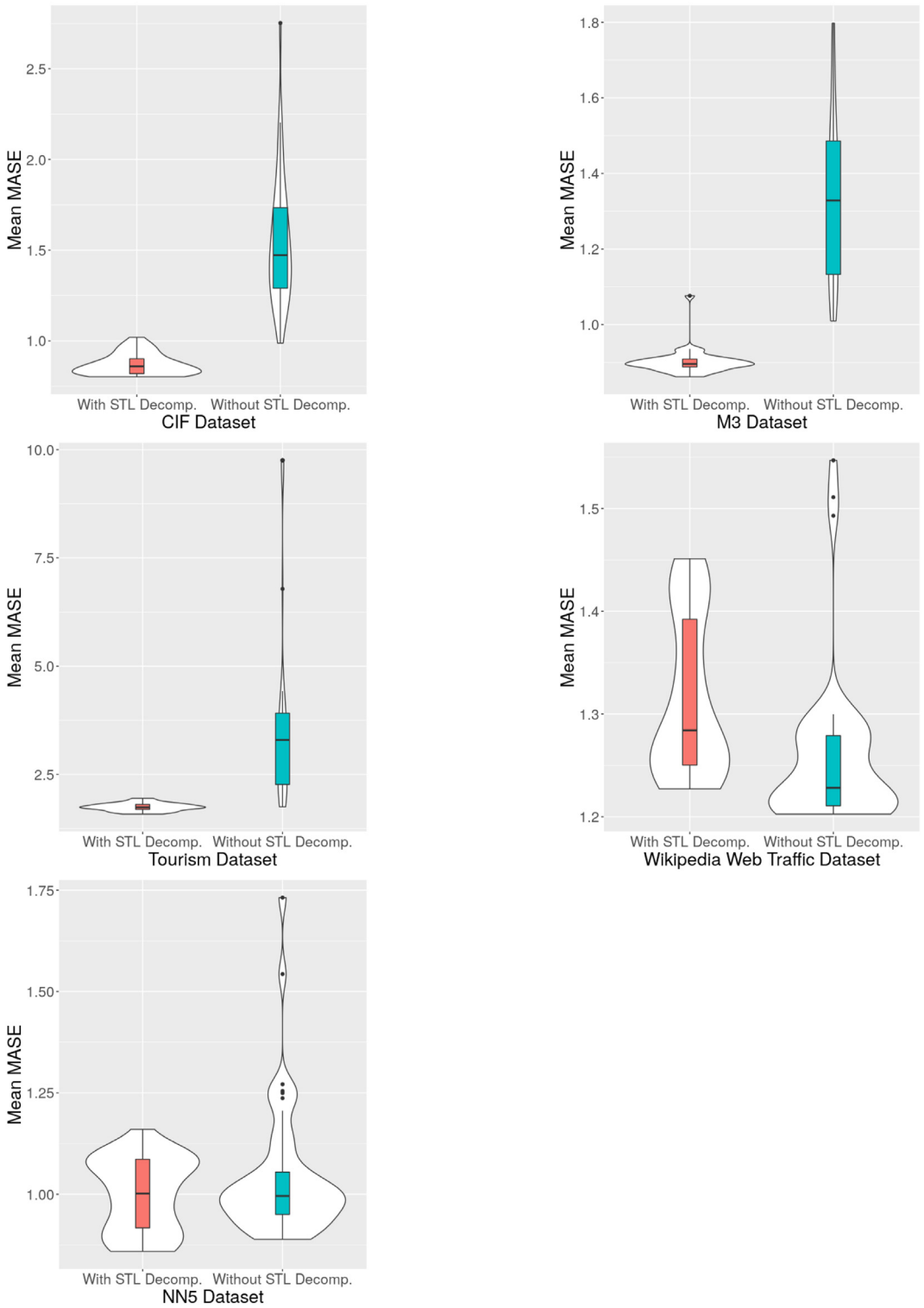

Fig. 18. Comparison of performance with and without STL decomposition: mean MASE.

generally performed competitively on most datasets. It outperformed the two univariate benchmarks on the CIF 2016, NN5, and Wikipedia Web Traffic datasets. Therefore, we can conclude that stacked model combined with LSTM cells and peephole connections optimised with COCOB was a competitive model combination for forecasting.

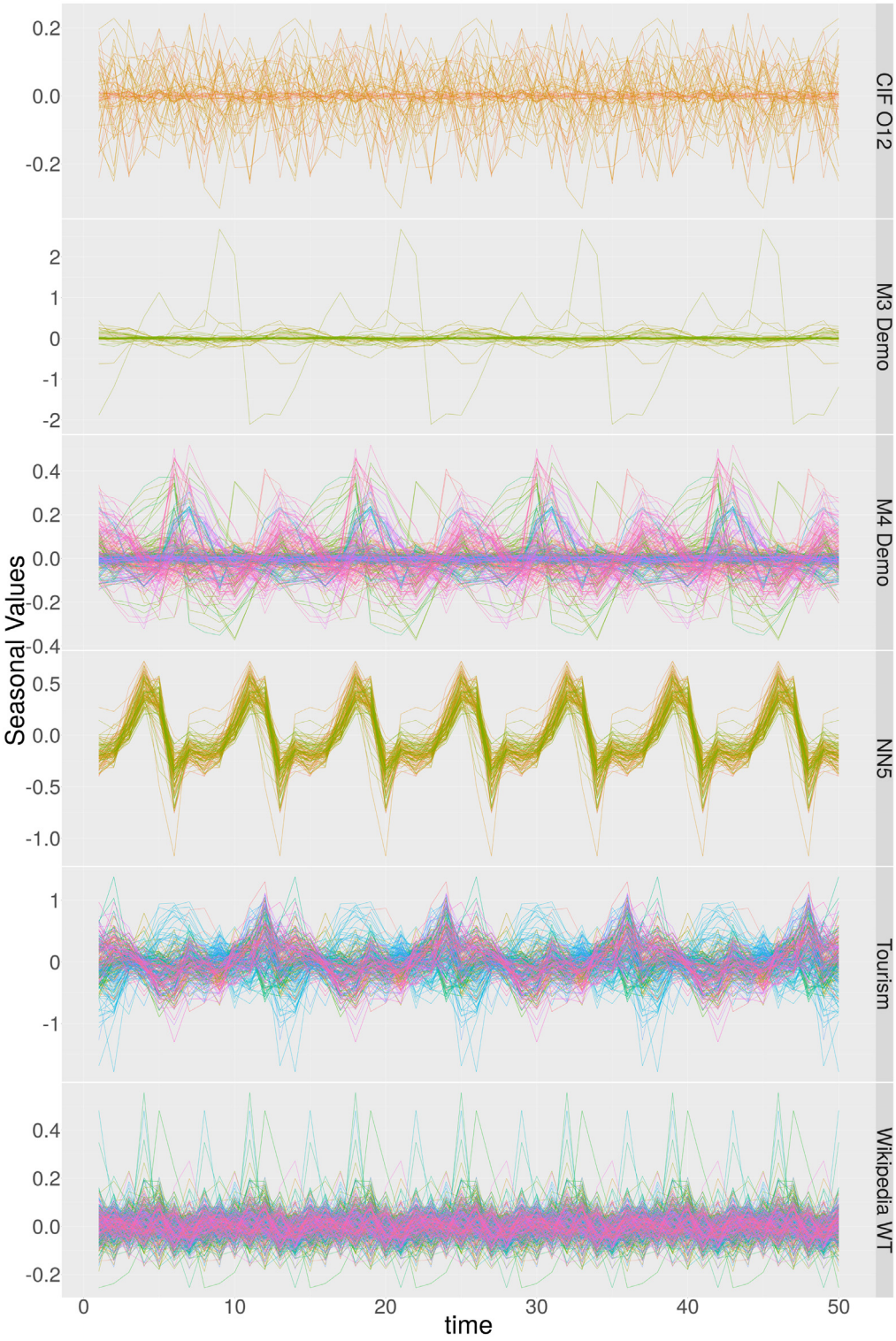

Fig. 19. Seasonal patterns of all datasets.

Table 6
Average rankings and results of the statistical testing for seasonality modelling with respect to the mean SMAPE across all datasets. On the CIF, M3, and Tourism datasets, using STL decomposition had the best ranking. On the Wikipedia Web Traffic and the NN5 datasets, not using STL decomposition had the best ranking. The calculated overall *p*-values obtained from the paired Wilcoxon signed-rank tests are shown for the different datasets in the 'Overall *p*-value' column. On the CIF, M3, and Tourism datasets, eliminating STL decomposition resulted in significantly worse performance than applying STL decomposition. However, on the Wikipedia Web Traffic dataset, applying STL decomposition resulted in a significantly worse performance than not applying it. On the NN5 dataset, there was no significant difference between using and not using STL decomposition.

Dataset	Ranking		Overall *p*-value
	With STL decomp.	Without STL decomp.	
CIF	**1.0**	2.0	2.91×10^{-11}
M3	**1.0**	2.0	2.91×10^{-11}
Tourism	**1.06**	1.94	1.455×10^{-10}
Wikipedia Web Traffic	1.73	**1.27**	0.028
NN5	1.56	**1.44**	0.911

Table 7
Mean SMAPE results with the number of trainable parameters as a hyperparameter.

Model name	CIF	Kaggle	M3	NN5
Stacked GRU cocob	10.69	**45.77**	14.67	**22.46**
Stacked LSTM cocob	**10.54**	45.93	**14.44**	24.04
Stacked ERNN cocob	10.66	129.23	14.99	24.67

5.8. Experiments involving the total number of trainable parameters

To compare the relative performance of the recurrent units presented in Section 5.2, we considered the cell dimension as a tuneable hyperparameter for the RNNs. However, as mentioned in Section 4.3.1, in other research communities it is also common to tune the total number of trainable parameters as a hyperparameter, instead of the cell dimension. Therefore, we also performed experiments by tuning the total number of trainable parameters as a hyperparameter to compare the relative performance of the different recurrent unit types. For this experiment, we selected the best configurations identified based on the results. Consequently, we selected the stacked architecture along with the COCOB optimiser to run on all three recurrent unit types. We performed this experiment on the four datasets: CIF, Wikipedia Web Traffic, M3, and NN5, in cases where the RNNs outperformed the statistical benchmarks. Particularly, on the NN5 and Wikipedia Web Traffic datasets, we ran the version of the stacked architecture without applying STL decomposition and with increased input window sizes. Table 7 shows these results in terms of the mean SMAPE values.

The violin plots in Fig. 23 indicate the relative performance of the three RNN units in terms of both the mean SMAPE and mean MASE ranks. In accordance with the results obtained in Section 5.2, Fig. 23 indicates that the LSTM cell with peephole connections performed the best, the ERNN cell performed the worst, and the performance of the GRU was in between. Again, the Friedman test of statistical significance in terms of the mean SMAPE values produced an overall *p*-value of 0.174, which means that this difference was not statistically significant.

Overall, we see that the results obtained after tuning the total number of trainable parameters are consistent with the results obtained by tuning the cell dimension instead. As mentioned in Section 4.3.1, there is a direct connection between the number of trainable parameters and the cell dimension. Following Smyl (2020), the cell dimension is not a very sensitive hyperparameter, which means that changes in the cell dimension, and, consequently, changes in the number of trainable parameters, do not have significant effects on the RNN performance. Therefore, we assume that the conclusions derived by tuning the cell dimension as a hyperparameter are valid even after correcting for the total number of trainable parameters in the different recurrent units. Nevertheless, since tuning the cell dimension is the only viable approach due to practical limitations, we concluded that it was sufficient to tune the cell dimension across a similar range to produce a suitable number of trainable parameters in the different recurrent unit types.

5.9. Comparison of the computational costs of the RNN models vs. the benchmarks

In addition to the comparisons of the prediction performance presented thus far, we compared the computational times of the RNN models with respect to the selected standard benchmarks on the four datasets: CIF2016, Wikipedia Web Traffic, M3, and NN5, in cases where an RNN model outperformed the benchmarks.

The computational times presented in Table 8 are for the best performing RNN models in each one of those datasets. For the purpose of comparison, both the benchmarks and the RNN models were allocated 25 CPU cores for the execution. The overall process of the RNN models had different stages in its pipeline, such as preprocessing the data, tuning the hyperparameters, and final model training with the optimal configuration for testing. Therefore, Table 8 shows a breakdown of the computational costs for these individual stages, as well as the total time.

From Table 8 we see that, compared to the standard benchmarks, the RNN models required a considerable amount of computational time for the overall process. For instance, on the horizon 12 category of the CIF dataset, the total time for the RNN was 3749.5 s (1.0 hr), whereas `auto.arima` and `ets` required only 85.1 s (1.4 min) and 41.6 s, respectively. Similarly, on the NN5 dataset, the RNN model required a total of 51490.7 s (14.3 hrs), whereas `auto.arima` and `ets` models required only

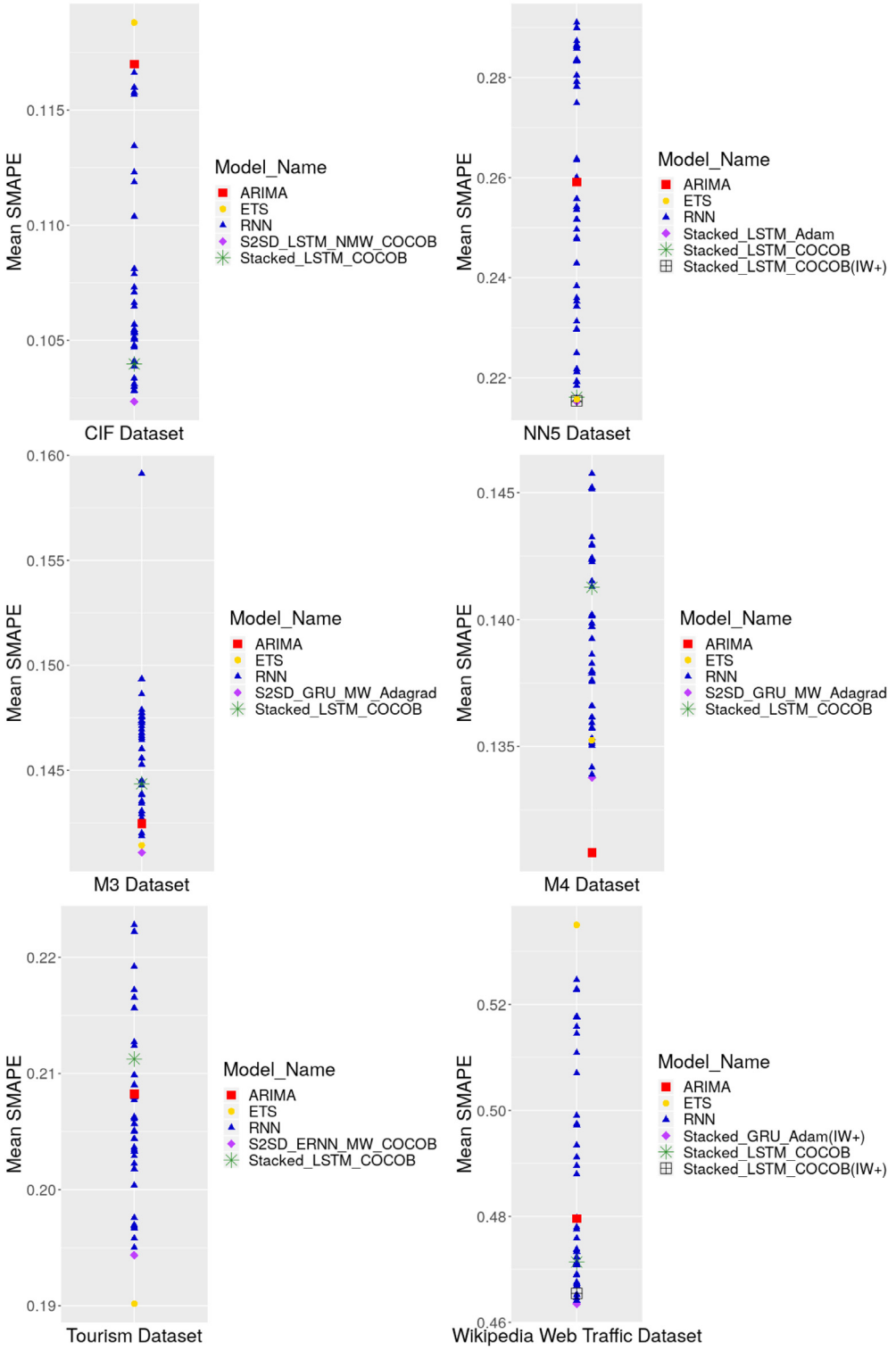

Fig. 20. Performance of RNNs compared to traditional univariate techniques: mean SMAPE.

1067.7 s (17.8 min) and 53.3 s, respectively. However, it is also evident from Table 8 that most of the computational time in the RNNs was devoted to hyperparameter tuning using SMAC with 50 iterations. For example, in the Micro category of the M3 dataset, although the whole pipeline took 2523.0 s (42.1 min), training the final model

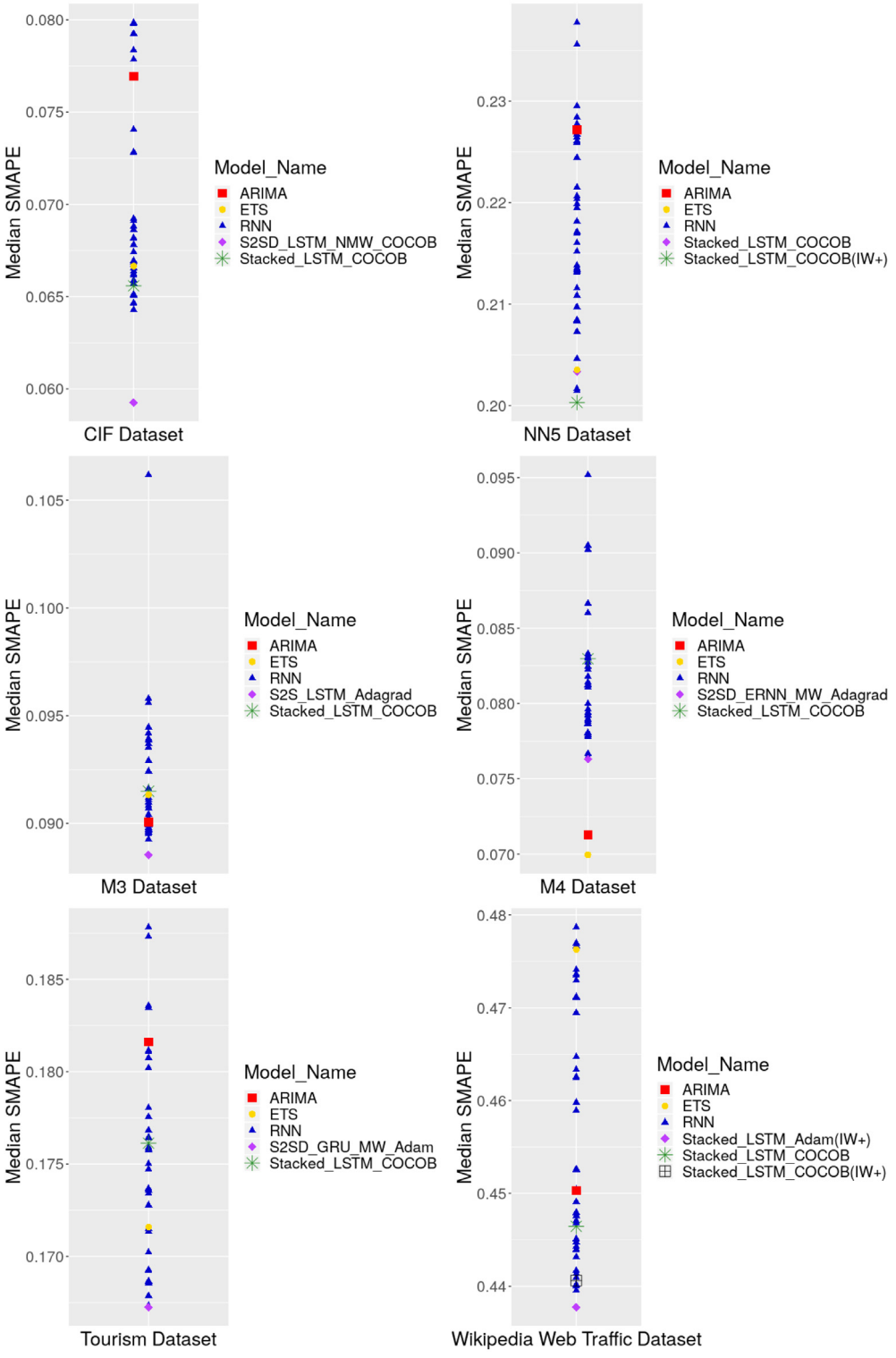

Fig. 21. Performance of RNNs compared to traditional univariate techniques: median SMAPE.

with the optimal configuration and testing accounted for only 565.6 s (9.4 min). On the other hand, the total running time for `auto.arima` and `ets` on the same dataset was 849.8 s (14.2 min) and 73.0 s (1.2 min), respectively. Hence, even though the overall process of RNNs was computationally costly, the training and testing of one

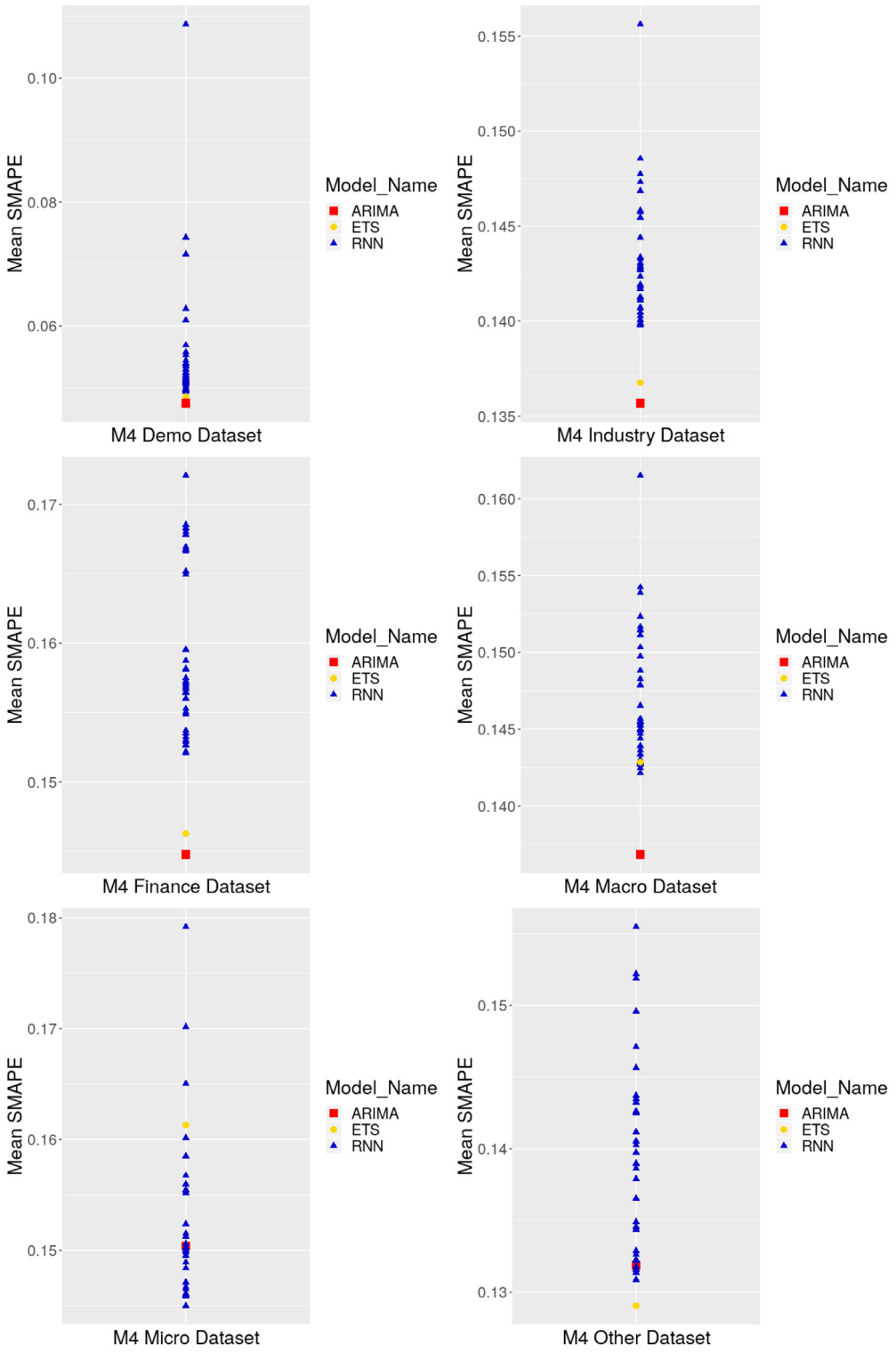

Fig. 22. Performance of RNNs compared to traditional univariate techniques in different M4 categories: mean SMAPE.

model was comparable to the computational costs of the statistical benchmarks. Moreover, compared to the benchmarks, which built one model for every series, the final trained models from the RNNs were less complex, with fewer parameters than the univariate techniques on a global scale.

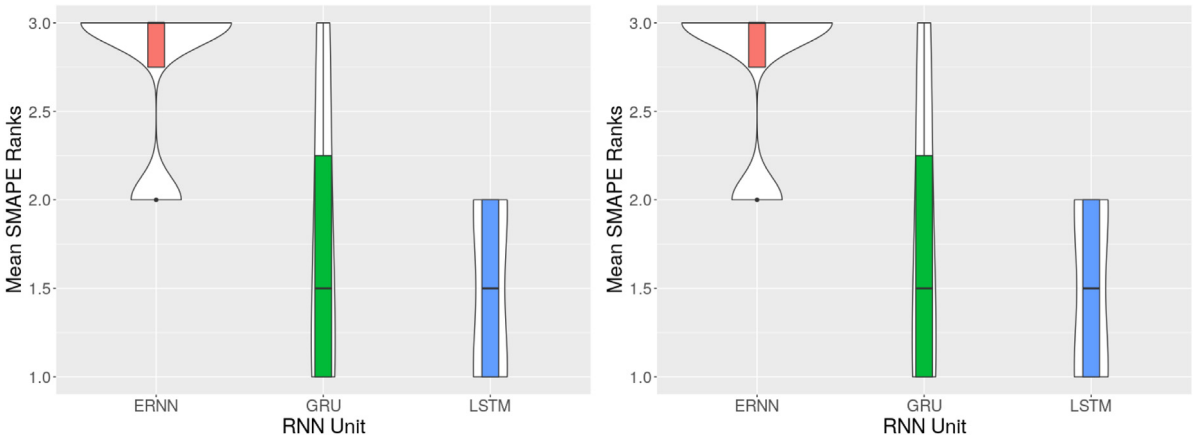

Fig. 23. Relative performance of different recurrent cell types under the same number of total trainable parameters.

From this comparison, we can see that RNNs are typically computationally more expensive models than traditional univariate techniques. However, regardless of the computational cost, they were capable of performing better than the traditional univariate benchmarks in all the cases listed in Table 8. This finding is consistent with another aspect to recent changes in the forecasting community, which now acknowledges that complex methods have merits over simpler statistical benchmarks (Makridakis et al., 2018b). Yet, our study shows how RNNs can be trained in such a way that they can achieve these improvements. With the current availability of massive amounts of computational resources, the improved accuracy brought forward by RNNs for forecasting is certainly beneficial to forecasting practitioners.

5.10. Hyperparameter configurations

Usually, when searching a larger initial hyperparameter space, there should be more iterations to the automated hyperparameter tuning technique. This depends on the number of hyperparameters as well as the initial hyperparameter range. We used 50 iterations of the SMAC algorithm for hyperparameter tuning, in order to be suitable for all the datasets. For our experiments, we selected roughly the same range across all the datasets, as shown in Table 4, except for the minibatch size. The minibatch size needed to be chosen proportional to the size of the dataset. Usually, for the lower bound of the initial hyperparameter range of the minibatch size, around a tenth of the size of the dataset is appropriate. However, we varied the upper bound in different orders for the different datasets. For small datasets, such as the CIF, NN5, and Tourism datasets and for the different categories of the M3 dataset, the upper bound differed from the lower bound in the order of tens. For bigger datasets, such as the Wikipedia Web Traffic dataset and the different categories of the M4 dataset, we set the upper bound to be larger than the lower bound in the order of hundreds. For the number of hidden layers in the RNN, many recent studies have suggested that a low value usually performs better (Bandara et al., 2020; Salinas et al., 2019; Smyl & Kuber, 2016; Wang et al., 2019). Following this convention, we selected between one and two layers and we observed that the RNNs performed well with such low values. On the one hand, this is due to the overfitting effects resulting from the increased number of parameters with the added layers. On the other hand, the number of layers also directly relates to the computational complexity. As for the learning rates, we found that the convergence of the Adagrad optimiser usually required a higher learning rate, in the range of 0.01–0.9. For the Adam optimiser, the identified range was smaller (0.001–0.1). On the other hand, variations in the cell dimension and the standard deviation of the random normal initialiser barely affected the performance of the models. It is also important not to set large values for the standard deviation of the regularisation parameters for Gaussian noise and L2 weight regularisation, since too high a value makes the model almost completely underfit the data and eliminates the NN's effect entirely when producing the final forecasts.

6. Conclusions

The motivation for this study was to address some of the key issues with using RNNs for forecasting and to evaluate whether and how they can be used effectively by forecasting practitioners with limited knowledge of these techniques. Through a number of systematic experiments across datasets with diverse characteristics, we derived conclusions regarding general data preprocessing best practices, hyperparameter configurations, and the best RNN architectures, recurrent units, and optimisers. All of the models we considered exploit cross-series information in the form of global models, and thus leverage the existence of massive time series databases with many related time series.

From our experiments, we conclude that a stacked architecture combined with LSTM cells with peephole connections and the COCOB optimiser, fed with deseasonalised data in a moving window format, is a generally competitive model across many datasets. In particular, though not statistically significant, the LSTM is the best unit type, even when correcting for the number of trainable parameters. We further conclude that when all the

Table 8
Computational times comparison of the RNNs with the benchmarks (in seconds).

Dataset	Model	Preprocessing	Hyperparameter tuning	Model training & Testing	Total
CIF(12)	S2SD LSTM NMW cocob	1.9	2531.8	1215.7	3749.5
CIF(12)	auto.arima	-	-	-	85.1
CIF(12)	ets	-	-	-	41.6
CIF(6)	S2SD LSTM NMW cocob	0.5	2535.3	304.9	2840.3
CIF(6)	auto.arima	-	-	-	4.3
CIF(6)	ets	-	-	-	8.3
Kaggle	NSTL Stacked GRU adam	159.7	12803.2	5350.2	18313.0
Kaggle	auto.arima	-	-	-	834.4
Kaggle	ets	-	-	-	481.7
M3(Mic)	S2SD GRU MW adagrad	30.2	1927.2	565.6	2523.0
M3(Mic)	auto.arima	-	-	-	849.8
M3(Mic)	ets	-	-	-	73.0
M3(Mac)	S2SD GRU MW adagrad	28.2	5359.9	1863.0	7251.0
M3(Mac)	auto.arima	-	-	-	719.5
M3(Mac)	ets	-	-	-	72.7
M3(Ind)	S2SD GRU MW adagrad	30.5	4084.0	1054.6	5169.1
M3(Ind)	auto.arima	-	-	-	1462.3
M3(Ind)	ets	-	-	-	218.8
M3(Dem)	S2SD GRU MW adagrad	5.8	1875.9	422.6	2304.4
M3(Dem)	auto.arima	-	-	-	273.3
M3(Dem)	ets	-	-	-	73.1
M3(Fin)	S2SD GRU MW adagrad	12.1	1848.9	493.0	2354.0
M3(Fin)	auto.arima	-	-	-	352.2
M3(Fin)	ets	-	-	-	90.3
M3(Oth)	S2SD GRU MW adagrad	4.1	1319.8	147.5	1471.5
M3(Oth)	auto.arima	-	-	-	273.3
M3(Oth)	ets	-	-	-	31.3
NN5	Stacked LSTM adam	42.2	37660.2	13788.3	51490.7
NN5	auto.arima	-	-	-	1067.7
NN5	ets	-	-	-	53.3

series in a dataset follow homogeneous seasonal patterns, with all of them covering the same duration in time with sufficient lengths, RNNs are capable of capturing seasonality without prior deseasonalisation. Otherwise, RNNs are weak when modelling seasonality on their own, and a deseasonalisation step should be employed. From the experiments involving the pooled and unpooled versions of the regression models, we can conclude that cross-series information helps with certain types of datasets. However, even on those datasets that involve many heterogeneous series, the strong modelling capabilities of RNNs can drive them to perform competitively in terms of forecasting accuracy. Therefore, we can conclude that leveraging cross-series information has its own benefits on sets of time series, while the predictive capability provided by RNNs can further improve forecasting accuracy. We also observed that RNNs incur relatively higher computational cost compared to statistical benchmarks. However, with the cloud infrastructure that is now commonly available to companies, such processing times are feasible. Moreover, our study empirically demonstrated that RNNs are good candidates for forecasting, and in many cases outperform statistical benchmarks that are currently the state of the art in the community. Thus, based on the extensive experiments involved with this study, we can confirm that complex methods now have benefits over simpler statistical benchmarks in many forecasting situations.

Our procedure was (semi-) automatic with regard to the initial hyperparameter range of the SMAC algorithm. We selected approximately the same range across all the datasets, except for the minibatch size, which we selected depending on the size of each dataset. Thus, although fitting RNNs is still not as straightforward and automatic as it is with the two state-of-the-art univariate forecasting benchmarks (viz., `ets` and `auto.arima`) our study and released code framework are important steps in this direction. Finally, we conclude that RNNs are now a good option for forecasting practitioners. They can be used to obtain reliable forecasts that outperform the benchmarks.

7. Future directions

The results of our study are limited to point forecasts in a univariate context. Nevertheless, modelling the uncertainty of NN predictions through probabilistic forecasting has received growing attention recently in the community. Further, when considering a complex forecasting scenario, such as a retail sales forecast, the sales of different products may be interdependent. Therefore, a sales forecasting task in such a context requires multivariate forecasting, rather than univariate forecasting. Furthermore, this study considered only single seasonality forecasting. In terms of higher frequency data with sufficient length, it will be beneficial to model multiple seasonalities in a big data context.

As seen in our study, global NN models often suffer from outlier errors for certain time series. This is probably because the average weights of NNs found by fitting global models may be unsuitable for the individual requirements of certain time series. Consequently, novel models are needed that incorporate both global parameters and local parameters for individual time series, in the form of hierarchical models, potentially combined with ensembling, where single models are trained in different ways with the existing dataset (e.g., on different subsets). The work by Bandara et al. (2020), Smyl (2020), and Sen et al. (2019) are steps in this direction, but this topic largely remains an open research question.

In general, deep learning is a fast-paced research field, where new architectures are rapidly introduced and discussed. However, it often remains unclear to practitioners which situations the techniques are most useful in and how difficult it is to adapt them to a given application. Although initially intended for image processing, CNNs have recently become increasingly popular for time series forecasting. The work carried out by Lai et al. (2018) and Shih et al. (2019) followed the argument that typical RNN-based attention schemes are not good at modelling seasonality. Therefore, they used a combination of CNN filters to capture local dependencies and a custom attention score function to model long-term dependencies. Lai et al. (2018) also experimented with recurrent skip connections to capture seasonality patterns. By contrast, dilated causal convolutions are specifically designed to capture long-range dependencies effectively along the temporal dimension (van den Oord et al., 2016). Such layers stacked on top of each other can build massive hierarchical attention networks, attending to points that are far back in the history. These have been recently used along with CNNs for time series forecasting problems. More advanced CNNs have also been introduced, such as temporal convolution networks (TCNs), which combine both dilated convolutions and residual skip connections. TCNs have also been used for forecasting in recent literature (Borovykh et al., 2018). Recent studies suggest that TCNs are promising NN architectures for sequence modelling tasks, in addition to being efficient at training (Bai et al., 2018). Therefore, forecasting practitioners using CNNs instead of RNNs may begin to gain a competitive advantage.

Acknowledgment

This research was supported by the Australian Research Council under grant DE190100045, a research award from Facebook Statistics for Improving Insights and Decisions, USA, Monash University, Australia Graduate Research funding, and the MASSIVE high-performance computing facility, Australia.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.ijforecast.2020.06.008.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., ... Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. https://www.tensorflow.org/.

Alexandrov, A., Benidis, K., Bohlke-Schneider, M., Flunkert, V., Gasthaus, J., Januschowski, T., Maddix, D. C., Rangapuram, S. S., Salinas, D., Schulz, J., Stella, L., Türkmen, A. C., & Wang, Y. (2019). Gluonts: Probabilistic time series models in python. *CoRR, abs/1906.05264*, http://arxiv.org/abs/1906.05264.

Assaad, M., Boné, R., & Cardot, H. (2008). A new boosting algorithm for improved time-series forecasting with recurrent neural networks. *Information Fusion*, *9*, 41–55.

Athanasopoulos, G., Hyndman, R., Song, H., & Wu, D. (2011). The tourism forecasting competition. *International Journal of Forecasting*, *27*, 822–844. http://dx.doi.org/10.1016/j.ijforecast.2010.04.009.

Athanasopoulos, G., J.H.yndman, R., Song, H., & Wu, D. (2010). Tourism forecasting part two. https://www.kaggle.com/c/tourism2/data.

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In Y. Bengio & Y. LeCun (Eds.), *3rd international conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, conference track proceedings*. http://arxiv.org/abs/1409.0473.

Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR, abs/1803.01271*, http://arxiv.org/abs/1803.01271.

Bandara, K., Bergmeir, C., & Smyl, S. (2020). Forecasting across time series databases using recurrent neural networks on groups of similar series: A clustering approach. *Expert Systems with Applications*, *140*, Article 112896.

Bandara, K., Shi, P., Bergmeir, C., Hewamalage, H., Tran, Q., & Seaman, B. (2019). Sales demand forecast in e-commerce using a long short-term memory neural network methodology. In T. Gedeon, K. W. Wong, & M. Lee (Eds.), *Neural information processing* (pp. 462–474). Cham: Springer International Publishing.

Bayer, J., & Osendorfer, C. (2014). Learning stochastic recurrent networks. https://arxiv.org/abs/1411.7610.

Ben Taieb, S., Bontempi, G., Atiya, A., & Sorjamaa, A. (2012). A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition. *Expert Systems with Applications*, *39*, 7067–7083. http://dx.doi.org/10.1016/j.eswa.2012.01.039.

Bergmeir, C., Hyndman, R. J., & Koo, B. (2018). A note on the validity of cross-validation for evaluating autoregressive time series prediction. *Computational Statistics & Data Analysis*, *120*, 70–83. http://dx.doi.org/10.1016/j.csda.2017.11.003.

Bergstra, J. (2012). Hyperopt: Distributed asynchronous hyper-parameter optimization. https://github.com/hyperopt/hyperopt.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*, 281–305.

Bianchi, F. M., Maiorino, E., Kampffmeyer, M. C., Rizzi, A., & Jenssen, R. (2017). An overview and comparative analysis of recurrent neural networks for short term load forecasting. *CoRR, abs/1705.04378*, http://arxiv.org/abs/1705.04378.

Borovykh, A., Bohte, S., & Oosterlee, C. W. (2018). Conditional time series forecasting with convolutional neural networks. arXiv preprint arXiv:1703.04691, https://arxiv.org/abs/1703.04691.

Box, G., Jenkins, G., & Reinsel, G. (1994). *Forecasting and control series, Time series analysis: forecasting and control*. Prentice Hall.

eResearch Centre, M. (2019). M3 user guide. https://docs.massive.org.au/index.html.

Chen, C., Twycross, J., & Garibaldi, J. M. (2017). A new accuracy measure based on bounded relative error for time series forecasting. *PLoS One*, *12*, Article e0174202.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN Encoder–Decoder for statistical machine translation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1724–1734). Stroudsburg, PA, USA: Association for Computational Linguistics.

Cinar, Y. G., Mirisaee, H., Goswami, P., Gaussier, E., Aït-Bachir, A., & Strijov, V. (2017). Position-based content attention for time series forecasting with sequence-to-sequence RNNs. In D. Liu, S. Xie, Y. Li, D. Zhao, & E.-S. M. El-Alfy (Eds.), *Neural information processing* (pp. 533–544). Cham: Springer International Publishing.

Claveria, O., Monte, E., & Torra, S. (2017). Data pre-processing for neural network-based forecasting: does it really matter? *Technological and Economic Development of Economy*, *23*, 709–725.

Claveria, O., & Torra, S. (2014). Forecasting tourism demand to Catalonia: Neural networks vs. time series models. *Economic Modelling*, *36*, 220–228.

Cleveland, R. B., Cleveland, W. S., McRae, J. E., & Terpenning, I. (1990). STL: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics*, *6*, 3–33.

Collins, J., Sohl-Dickstein, J., & Sussillo, D. (2016). Capacity and trainability in recurrent neural networks. In *International conference on learning representations 2016*.

Crone, S. F. (2008). NN5 Competition. http://www.neural-forecasting-competition.com/NN5/.

Crone, S. F., Hibon, M., & Nikolopoulos, K. (2011). Advances in forecasting with neural networks? empirical evidence from the NN3 competition on time series prediction. *International Journal of Forecasting*, *27*, 635–660.

Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M. D., & Saurous, R. A. (2017). Tensorflow distributions. *CoRR*, *abs/1711.10604*, http://arxiv.org/abs/1711.10604.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, 2121–2159.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*, 179–211.

Fernando (2012). Bayesian optimization. https://github.com/fmfn/BayesianOptimization.

Friedman, J., Hastie, T., & Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, *33*, 1–22.

García, S., Fernández, A., Luengo, J., & Herrera, F. (2010). Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences*, *180*, 2044–2064.

Gasthaus, J., Benidis, K., Wang, Y., Rangapuram, S. S., Salinas, D., Flunkert, V., & Januschowski, T. (2019). Probabilistic forecasting with spline quantile function RNNs. In K. Chaudhuri, & M. Sugiyama (Eds.), *Proceedings of machine learning research: 89*, *Proceedings of machine learning research* (pp. 1901–1910). PMLR.

Google (2017). Web traffic time series forecasting. https://www.kaggle.com/c/web-traffic-time-series-forecasting.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE conference on computer vision and pattern recognition* (pp. 770–778). http://dx.doi.org/10.1109/CVPR.2016.90.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*, 1735–1780.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*, 359–366.

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In C. A. C. Coello (Ed.), *Learning and intelligent optimization* (pp. 507–523). Berlin, Heidelberg: Springer Berlin Heidelberg.

Hyndman, R. (2018). A brief history of time series forecasting competitions. https://robjhyndman.com/hyndsight/forecasting-competitions/.

Hyndman, R., Kang, Y., Talagala, T., Wang, E., & Yang, Y. (2019). Tsfeatures: Time series feature extraction. R package version 1.0.0. https://pkg.robjhyndman.com/tsfeatures/.

Hyndman, R., & Khandakar, Y. (2008). Automatic time series forecasting: The forecast package for R. *Journal of Statistical Software, Articles*, *27*, 1–22.

Hyndman, R. J., & Koehler, A. B. (2006). Another look at measures of forecast accuracy. *International Journal of Forecasting*, *22*, 679–688.

Hyndman, R., Koehler, A., Ord, K., & D. Snyder, R. (2008). *Forecasting with exponential smoothing. The state space approach*. Springer Berlin Heidelberg.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd international conference on machine learning: Vol. 37* (pp. 448–456). JMLR.org.

Jagannatha, A. N., & Yu, H. (2016). Bidirectional RNN for medical event detection in electronic health records. In *Proceedings of the conference. Association for Computational Linguistics. North American chapter. Meeting* (pp. 473–482).

Januschowski, T., Gasthaus, J., Wang, Y., Salinas, D., Flunkert, V., Bohlke-Schneider, M., & Callot, L. (2020). Criteria for classifying forecasting methods. *International Journal of Forecasting*, *36*, 167–177. http://dx.doi.org/10.1016/j.ijforecast.2019.05.008, M4 Competition.

Ji, Y., Haffari, G., & Eisenstein, J. (2016). A latent variable recurrent neural network for discourse-driven language models. In *Proceedings of the 2016 conference of the North American chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 332–342). San Diego, California: Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/N16-1037.

Jozefowicz, R., Zaremba, W., & Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd international conference on machine learning: Vol. 37* (pp. 2342–2350). JMLR.org.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd international conference for learning representations*.

Koutník, J., Greff, K., Gomez, F., & Schmidhuber, J. (2014). A clockwork RNN. In *Proceedings of the 31st international conference on machine learning: Vol. 32* (pp. II–1863–II–1871). JMLR.org.

Krstanovic, S., & Paulheim, H. (2017). Ensembles of recurrent neural networks for robust time series forecasting. In *Artificial intelligence XXXIV* (pp. 34–46). Springer International Publishing.

Lai, G., Chang, W.-C., Yang, Y., & Liu, H. (2018). Modeling long- and short-term temporal patterns with deep neural networks. In *The 41st International ACM SIGIR conference on research & development in information retrieval* (pp. 95–104). ACM.

Laptev, N., Yosinski, J., Li, L. E., & Smyl, S. (2017). Time-series extreme event forecasting with neural networks at uber. In *International conference on machine learning: Vol. 34* (pp. 1–5).

Liang, Y., Ke, S., Zhang, J., Yi, X., & Zheng, Y. (2018). Geoman: Multi-level attention networks for geo-sensory time series prediction. In *Proceedings of the twenty-seventh international joint conference on artificial intelligence* (pp. 3428–3434). International Joint Conferences on Artificial Intelligence Organization, http://dx.doi.org/10.24963/ijcai.2018/476.

Lindauer, M., Eggensperger, K., Feurer, M., Falkner, S., Biedenkapp, A., & Hutter, F. (2017). SMAC V3: Algorithm configuration in python. https://github.com/automl/SMAC3.

Luong, T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 conference on empirical methods in natural language processing* (pp. 1412–1421). Stroudsburg, PA, USA: Association for Computational Linguistics.

Makridakis, S., & Hibon, M. (2000). The M3-Competition: results, conclusions and implications. *International Journal of Forecasting*, *16*, 451–476.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018a). The M4 competition: Results, findings, conclusion and way forward. *International Journal of Forecasting*, *34*, 802–808.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018b). Statistical and machine learning forecasting methods: Concerns and ways forward. *PLOS ONE*, *13*, Article e0194889. http://dx.doi.org/10.1371/journal.pone.0194889.

Mandal, P., Senjyu, T., Urasaki, N., & Funabashi, T. (2006). A neural network based several-hour-ahead electric load forecasting using similar days approach. *International Journal of Electrical Power & Energy Systems*, *28*, 367–373.

Montero-Manso, P., Athanasopoulos, G., Hyndman, R. J., & Talagala, T. S. (2020). FFORMA: Feature-based forecast model averaging. *International Journal of Forecasting*, *36*, 86–92, M4 Competition.

Nelson, M., Hill, T., Remus, W., & O'Connor, M. (1999). Time series forecasting using neural networks: Should the data be deseasonalized first? *Journal of Forecasting*, *18*, 359–367.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. W., & Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. In *The 9th ISCA speech synthesis workshop* (p. 125). ISCA.

Orabona, F. (2017). Cocob. https://github.com/bremen79/cocob.

Orabona, F., & Tommasi, T. (2017). Training deep networks without learning rates through coin betting. In *Proceedings of the 31st international conference on neural information processing systems* (pp. 2157–2167). USA: Curran Associates Inc..

Oreshkin, B. N., Carpov, D., Chapados, N., & Bengio, Y. (2019). N-BEATS: neural basis expansion analysis for interpretable time series forecasting. *CoRR, abs/1905.10437*, http://arxiv.org/abs/1905.10437.

Peng, C., Li, Y., Yu, Y., Zhou, Y., & Du, S. (2018). Multi-step-ahead host load prediction with GRU based encoder-decoder in cloud computing. In *2018 10th international conference on knowledge and smart technology* (pp. 186–191).

Qin, Y., Song, D., Cheng, H., Cheng, W., Jiang, G., & Cottrell, G. W. (2017). A dual-stage attention-based recurrent neural network for time series prediction. In *Proceedings of the 26th international joint conference on artificial intelligence* (pp. 2627–2633). AAAI Press.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, http://www.R-project.org/.

Rahman, M. M., Islam, M. M., Murase, K., & Yao, X. (2016). Layered ensemble architecture for time series forecasting. *IEEE Transactions on Cybernetics*, *46*, 270–283.

Rangapuram, S. S., Seeger, M., Gasthaus, J., Stella, L., Wang, Y., & Januschowski, T. (2018). Deep state space models for time series forecasting. In *Proceedings of the 32nd international conference on neural information processing systems* (pp. 7796–7805). USA: Curran Associates Inc..

Rob J Hyndman, G. A. (2018). *Forecasting: Principles and practice* (2nd ed.). OTexts, https://otexts.com/fpp2/.

Salinas, D., Flunkert, V., Gasthaus, J., & Januschowski, T. (2019). Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*.

Schäfer, A. M., & Zimmermann, H. G. (2006). Recurrent neural networks are universal approximators. In *Proceedings of the 16th international conference on artificial neural networks: Vol. Part I* (pp. 632–640). Berlin, Heidelberg: Springer-Verlag, http://dx.doi.org/10.1007/11840817_66.

Schuster, M., & Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*, 2673–2681.

Sen, R., Yu, H.-F., & Dhillon, I. (2019). Think globally, act locally: A deep neural network approach to high-dimensional time series forecasting. https://arxiv.org/abs/1905.03806.

Sharda, R., & Patil, R. B. (1992). Connectionist approach to time series prediction: an empirical test. *Journal of Intelligent Manufacturing*, *3*, 317–323.

Shih, S.-Y., Sun, F.-K., & Lee, H.-y. (2019). Temporal pattern attention for multivariate time series forecasting. *Machine Learning*, *108*, 1421–1441. http://dx.doi.org/10.1007/s10994-019-05815-0.

Smyl, S. (2016). Forecasting short time series with LSTM neural networks. https://gallery.azure.ai/Tutorial/Forecasting-Short-Time-Series-with-LSTM-Neural-Networks-2. (Accessed 30 October 2018).

Smyl, S. (2017). Ensemble of specialized neural networks for time series forecasting. In *37th international symposium on forecasting*.

Smyl, S. (2020). A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, *36*, 75–85. http://dx.doi.org/10.1016/j.ijforecast.2019.03.017, M4 Competition.

Smyl, S., & Kuber, K. (2016). Data preprocessing and augmentation for multiple short time series forecasting with recurrent neural networks. In *36th international symposium on forecasting*.

Snoek, J. (2012). Spearmint. https://github.com/JasperSnoek/spearmint.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Proceedings of the 25th international conference on neural information processing systems: Vol. 2* (pp. 2951–2959). USA: Curran Associates Inc..

Soudry, D., Hoffer, E., Nacson, M. S., Gunasekar, S., & Srebro, N. (2018). The implicit bias of gradient descent on separable data. *Journal of Machine Learning Research*, *19*, 2822–2878.

Štěpnička, M., & Burda, M. (2017). On the results and observations of the time series forecasting competition CIF 2016. In *2017 IEEE international conference on fuzzy systems* (pp. 1–6).

Suilin, A. (2017). Kaggle-web-traffic. https://github.com/Arturus/kaggle-web-traffic/. (Accessed 19 November 2018).

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th international conference on neural information processing systems: Vol. 2* (pp. 3104–3112). Cambridge, MA, USA: MIT Press.

Tang, Z., de Almeida, C., & Fishwick, P. A. (1991). Time series forecasting using neural networks vs. Box–Jenkins methodology. *Simulation*, *57*, 303–310.

Trapero, J. R., Kourentzes, N., & Fildes, R. (2015). On the identification of sales forecasting models in the presence of promotions. *The Journal of the Operational Research Society*, *66*, 299–307. http://dx.doi.org/10.1057/jors.2013.174.

Wang, Y., Smola, A., Maddix, D., Gasthaus, J., Foster, D., & Januschowski, T. (2019). Deep factors for forecasting. In K. Chaudhuri, & R. Salakhutdinov (Eds.), *Proceedings of the 36th International Conference on Machine Learning: Vol. 97* (pp. 6607–6617). Long Beach, California, USA: PMLR.

Wen, R., Torkkola, K., Narayanaswamy, B., & Madeka, D. (2017). A multi-horizon quantile recurrent forecaster. In *31st conference on neural information processing systems (NIPS 2017), time series workshop*.

Yan, W. (2012). Toward automatic time-series forecasting using neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, *23*, 1028–1039.

Yan, Y. (2016). RBayesianoptimization: Bayesian optimization of hyperparameters. R package version 1.1.0. https://CRAN.R-project.org/package=rBayesianOptimization.

Yao, K., Cohn, T., Vylomova, K., Duh, K., & Dyer, C. Depth-gated LSTM. In *20th Jelinek summer workshop on speech and language technology 2015*.

Yeo, I.-K., & Johnson, R. A. (2000). A new family of power transformations to improve normality or symmetry. *Biometrika*, *87*, 954–959.

Zhang, G. (2003). Time series forecasting using a hybrid ARIMA and neural network model. *Neurocomputing*, *50*, 159–175.

Zhang, G., & Berardi, V. (2001). Time series forecasting with neural network ensembles: an application for exchange rate prediction. *Journal of the Operational Research Society*, *52*, 652–664.

Zhang, G., Eddy Patuwo, B., & Hu, M. Y. (1998). Forecasting with artificial neural networks: The state of the art. *International Journal of Forecasting*, *14*, 35–62.

Zhang, G., & Kline, D. (2007). Quarterly time-series forecasting with neural networks. *IEEE Transactions on Neural Networks*, *18*, 1800–1814.

Zhang, G., & Qi, M. (2005). Neural network forecasting for seasonal and trend time series. *European Journal of Operational Research*, *160*, 501–514.

Zhu, L., & Laptev, N. (2017). Deep and confident prediction for time series at uber. In *2017 IEEE international conference on data mining workshops* (pp. 103–110). IEEE.