A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting

Article *in* International Journal of Forecasting · July 2019

DOI: 10.1016/j.ijforecast.2019.03.017

Slawek Smyl
Uber

A hybrid method of Exponential Smoothing and Recurrent Neural Networks for time series forecasting

Slawek Smyl[a,*]

[a]*Uber Technologies, 555 Market St, 94104, San Francisco, CA, USA*

Abstract

This paper presents the winning submission of the M4 forecasting competition. The submission utilizes a Dynamic Computational Graph Neural Network system that enables mixing of a standard Exponential Smoothing model with advanced Long Short Term Memory networks into a common framework. The result is a hybrid and hierarchical forecasting method.

Keywords: Forecasting competitions, M4, Dynamic Computational Graphs, Automatic Differentiation, Long Short Term Memory (LSTM) networks, Exponential Smoothing

1. Introduction

During the last decades Neural Networks (NNs), as well as other Machine Learning (ML) algorithms, have achieved remarkable success in various areas, including, among others image and speech recognition, Natural Language Processing (NLP), autonomous vehicles and games (Makridakis, 2017). The key behind their success is that, provided a large representative dataset, ML algorithms can learn to identify complex non-linear patterns and explore unstructured relations without hypothesizing them a priori. Thus, ML algorithms are not limited by assumptions or pre-defined data generating processes, allowing data to speak for itself.

However, when it comes to forecasting, the superiority of ML is not apparent. While in some applications like energy forecasting (Dimoulkas et al., 2018), where the series being extrapolated are often numerous, long, and accompanied by explanatory variables, ML

*Corresponding author
Email address: slaweks@hotmail.co.uk (Slawek Smyl)

algorithms were successful (Weron, 2014), in more typical time series forecasting, where data availability is often limited and regressors are not available, the performance of ML algorithms tends to be below expectations (Makridakis et al., 2018c).

None of the popular ML algorithms have been created for time series forecasting. In order to use them for forecasting time series data needs to be preprocessed. The strength, and actually the requirement, for a successful use of ML algorithms, is cross-learning, i.e. using many series to train a single model. This is unlike the standard statistical time series algorithms where a separate model is developed for each series. But in order to learn across many time series, the preprocessing needs to be well thought over. NNs are particularly sensitive in this area, this topic which will be expanded later.

There are many good rules on preprocessing, but it remains an experiment-intensive art. One of the most important ingredient of the success of the method in M4 Competition was the on the fly preprocessing that was an inherent part of the training process. Crucially, parameters of this preprocessing were being updated by the same overall optimization procedure (Stochasting Gradient Descent) as weights of the NNs, with the overarching goal of accurate forecasting (minimizing forecasting errors).

The preprocessing parameters were actually ones of (a bit simplified) updating formulas of some models from Exponential Smoothing family. So what is presented here is a hybrid forecasting method that mixes an Exponential Smoothing (ES) model with advanced Long Short Term Memory (LSTM) neural networks into a common framework. The ES equations enable the method to effectively capture the main components of the individual series, such as seasonality and level, while the LSTM networks allow nonlinear trend and cross-learning. In this regard, data are exploited in a hierarchical manner, meaning that both local and global components are utilized to extract and combine information coming either from a series or a dataset level and, therefore, enhance forecasting accuracy.

The rest of the paper is organized as follows. In Section 2 the method is introduced and described in a general sense, while Section 3 gives more implementation details. Section 4 concludes by sketching some general modelling possibilities provided by recent NN systems and Probabilistic Programming languages, and in this context traces back development of the models described in this paper.

2. Methodology

2.1. Intuition and overview of the hybrid method

The method effectively mixes ES models with LSTM networks and by doing so it provides more accurate forecasts than ones generated by either pure statistical or ML approaches, exploiting that way their advantages while avoiding their drawbacks. This hybrid forecasting approach has the following three main elements: (i) Deseasonalization and adaptive normalization, (ii) Generation of forecasts and (iii) Ensembling.

The first element is implemented with state-space ES-style formulas. Initial seasonality components (e.g. 4 for the quarterly series) and smoothing coefficients (two of them in case of a single seasonality system) are per series parameters and were fitted, together with global NN weights, by Stochastic Gradient Descent (SGD). Knowing these parameters and values of the series allows to calculate all the seasonality components and levels, and these are used for deseasonalization and normalization. Deseasonalization of seasonal series was very important in M4 Competition, given that the series were provided as numeric vectors without any time stamp, so there was no way to incorporate calendar features like day of week or month number. Also series came from many sources, so their seasonality patters were different.

The second element was a NN that operated on deseasonalized and normalized data, providing the horizon-steps ahead (e.g. 18 points in case of monthly series) outputs that were subsequently re-normalized and re-seasonalized to provide forecasts. The NN is global, learning across many time series.

The final element of the method is the ensembling of forecasts made in the previous step. This includes ensembling the forecasts produced by the individual models from several independent runs, sometimes, produced by a subset of concurrently trained models, and also averaging those generated by the most recent training epochs. This process further enhances the robustness of the method, mitigating model and parameter uncertainty (Petropoulos et al., 2018) while also exploiting the beneficial effects of combining (Chan & Pauwels, 2018).

Based on the above, it can be said that the method has the following two special characteristics:

- It is hybrid, in a sense that statistical modeling (ES models) is combined concurrently with ML algorithms (LSTM networks).
- It is of a hierarchical nature, in a sense that both global (applicable to large subsets of all series) and local (applied to each series individually) parameters are utilized to enable cross-learning while also emphasizing the particularities of the time series being extrapolated.

2.2. Method description

2.2.1. Deseasonalization and normalization

M4 time series, even within same frequency subset, e.g. monthly, came from many different sources and exhibited various seasonality patterns. Additionally, starting dates of the series were not provided. In such circumstances the NNs are unable to learn to deal effectively with seasonality. A standard remedy is to apply a deseasonalization at preprocessing time. This is an adequate solution, but not the best one, as it completely separates the preprocessing from forecasting, and the quality of the decomposition near the end of the series, where it counts most for the forecast, is likely to be the worse. One can also observe that the deseasonalization algorithms, however sophisticated and robust, were not designed to be good preprocessors for NNs. Classical, statistical models like those from Exponential Smoothing family show a better way: the forecasting model has integral parts dealing with seasonality.

In most NNs variants, including LSTMs, the update size of weight w_{ij} is proportional to the final error, but also the absolute value of the strength of the associated signal (coming from neuron i in current layer to neuron j in next layer). Therefore the NNs, even if implemented on a digital computer, behave like analogue devices: small-valued inputs will have small impact during the learning process. Normalizing each series globally to an interval like [0-1] is also problematic as the values to be forecasted may lie outside this range, and, more importantly, for series that change a lot during their lifespan, the parts of the series with small values will be also ignored. Finally, information on the strength of trend is lost: two series of the same length and similar shape, one growing from 100 to 110 and another from 100 to 200, will look very similar after the [0-1] normalization. Therefore, normalization is necessary, but should be adaptive and local, where the "normalizer" follows the series values.

2.2.2. Exponential Smoothing formulas

Keeping in mind that the M4 series are all of positive values, the models of Holt (Gardner, 2006) and Holt-Winters (Hyndman et al., 2008) with multiplicative seasonality were chosen. However, these were simplified by removal of the linear trend: the NN was tasked to produce a, most likely non-linear, trend. Moreover, depending on the frequency of the data, either non-seasonal (yearly and daily data), single-seasonal (monthly, quarterly, and weekly data) or double-seasonal (hourly data) models (Taylor, 2003) were used. The updating formulas for each case are listed below.

Non-seasonal models:

$$l_t = \alpha y_t + (1-\alpha) l_{t-1} \tag{1}$$

Single seasonality models:

$$\begin{aligned} l_t &= \alpha y_t / s_t + (1-\alpha) l_{t-1} \\ s_{t+K} &= \beta y_t / l_t + (1-\beta) s_t \end{aligned} \tag{2}$$

Double seasonality models:

$$\begin{aligned} l_t &= \alpha y_t / (s_t u_t) + (1-\alpha) l_{t-1} \\ s_{t+K} &= \beta y_t / (l_t u_t) + (1-\beta) s_t \\ u_{t+L} &= \gamma y_t / (l_t s_t) + (1-\gamma) u_t \end{aligned} \tag{3}$$

where y_t is the value of the series at point t, l_t, s_t, and u_t are the components of level, seasonality, and the second seasonality respectively, K is the number of observations per seasonal period, i.e, 12 for monthly, 4 for quarterly, 52 for weekly, and finally L is number of observations per second seasonal period (for hourly data, 168). Note that s_t and u_t are always positive, while the smoothing coefficients α, β and γ take a value between 0 and 1. These restrictions are easily implemented by applying exp() to the underlying parameters of the initial seasonality components and sigmoid() to the underlying parameters of the smoothing coefficients.

2.2.3. On the fly preprocessing

The above formulas allow to calculate level and seasonality components for all points of each series. Then these components are used for deseasonalization and adaptive normalization during the on the fly preprocessing. This step is a crucial part of the method and is described in this section.

Each series was preprocessed anew for each training epoch, because the parameters (initial seasonality components and smoothing coefficients) and resulting levels and seasonality components, were different during each epoch.

The standard approach of constant size, rolling input and output windows was applied, as shown in Figure 1 for a case of a monthly series. The size of the output window was always equal to the forecasting horizon (e.g., 13 for the weekly series), while the size of the input window was determined by a rule that it should cover at least a full seasonal period (e.g., being equal or larger than 4 in case of quarterly series), and for non-seasonal series the size of the input window should be close to the forecasting horizon. However, the exact size was defined after conducting experimentation (backtesting). Please note that in contradistinction to many other Recurrent Neural Network (RNN) -based sequence processing systems, the input size is larger than 1. This works better as it allows the NN to be directly exposed to the immediate history of the series.

Preprocessing was rather simple: at each step the values in input and output windows were normalized by dividing them by the last value of level in the input window (the thick blue dot on Figure 1), and then, in case of seasonal time series, further divided by the relevant seasonality components. That resulted in input and output values to be close to 1, irrespective of the original amplitude of the series and its history. Finally, a squashing function, log() was applied. The squashing function prevented outliers to have an unduly large and disturbing effect on the learning.

Additionally, the domain of the time series (e.g. Finance or Macro) was one-hot encoded as a 6-long vector and appended to the time series-derived features. The domain information was the only meta information available and I considered prudent to expose the NN to it.

In general, increasing the size of the input window and extracting more sophisticated features, like strength of seasonality or variability, are worth-trying ideas when preprocessing for NNs, but such approaches were not adopted here for several reasons. The most important one was that many series were too short to afford a large input window, meaning that they could not be used for backtesting. Another reason was that creating features that effectively summarize the characteristics of the series irrespective of their length is not straightforward. It was only after the end of the competition that a promising R package called tsfeatures came to my attention (Hyndman et al., 2015; Kang et al., 2017).

2.2.4. Forecast by NNs

As explained above, the NNs operated on deseasonalized, adaptively normalized, and squashed values. Their output needed to be "unwound", in following way:

For non-seasonal models:

$$\hat{y}_{t+1..t+h} = exp(NN(x)) * l_t \tag{4}$$

For single seasonality models:

$$\hat{y}_{t+1..t+h} = exp(NN(x)) * l_t * s_{t+1:..t+h} \tag{5}$$

For dual seasonality models:

$$\hat{y}_{t+1..t+h} = exp(NN(x)) * l_t * s_{t+1:..t+h} * u_{t+1:..t+h} \tag{6}$$

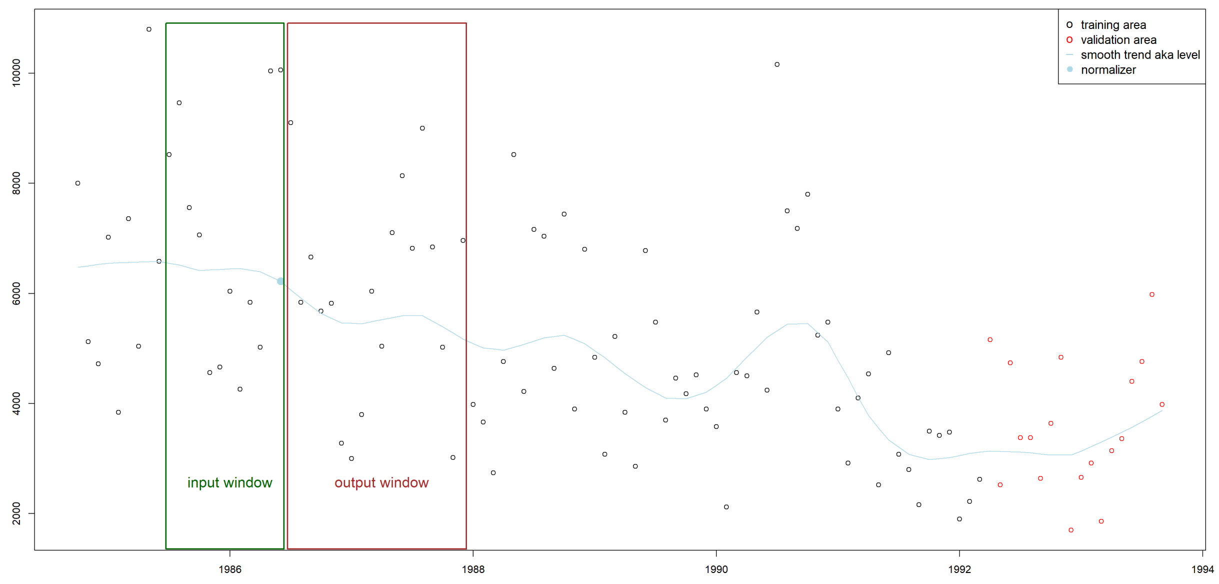

Figure 1: An example displaying how rolling windows are used for preprocessing a random monthly series. The last step is used for validation.

where x is the pre-processed input (a vector), NN(x) is an NN output (a vector), l_t is the value of level at time t (last known data point) and h the forecasting horizon. All operations are elementwise. The above is summarized in Figure 2

Note that the final forecast is actually an ensemble of many such forecasts, a procedure which is explained later in the paper.

2.2.5. Architectures of Neural Networks

In order to better understand the implementation, it is useful to classify the parameters of forecasting systems into the following three groups:

- **Local constants:** These parameters reflect the behavior of a single series. They do not change as we step through the series. For example, the smoothing coefficients of the ES model, as well as the initial seasonal components, are the local non-changing (constant) parameters.

- **Local states:** These parameters change as we step through a series, they evolve in

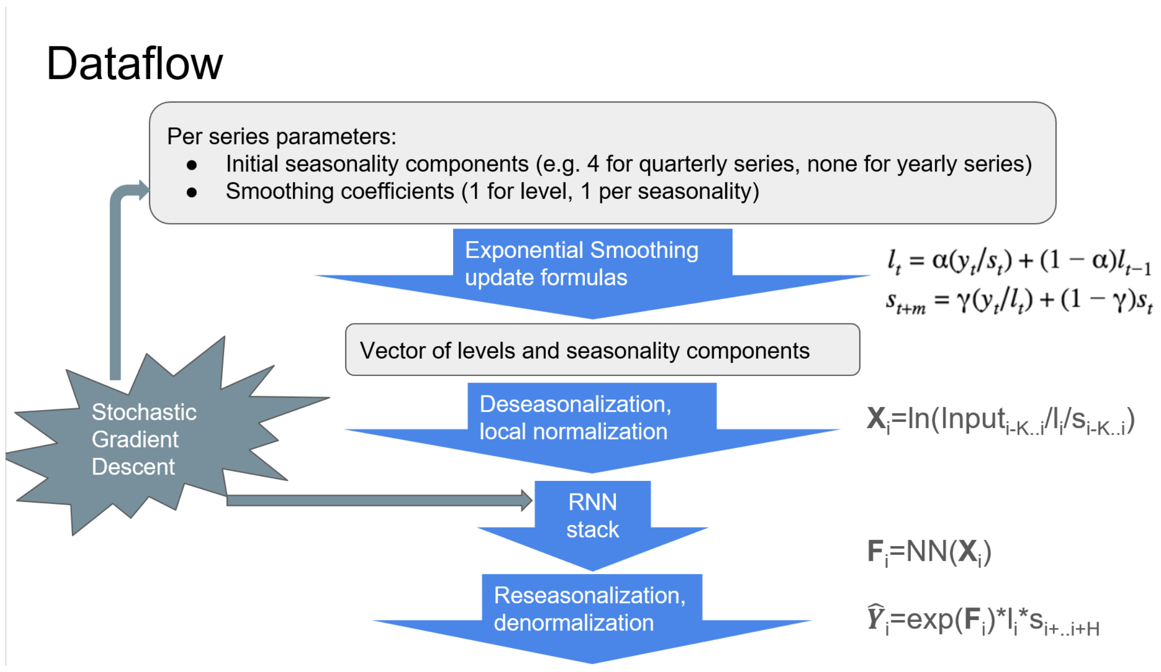
Dataflow
Per series parameters:
● Initial seasonality components (e.g. 4 for quarterly series, none for yearly series)
● Smoothing coefficients (1 for level, 1 per seasonality)
Exponential Smoothing update formulas
$l_t = \alpha(y_t/s_t) + (1-\alpha)l_{t-1}$
$s_{t+m} = \gamma(y_t/l_t) + (1-\gamma)s_t$
Vector of levels and seasonality components
Stochastic Gradient Descent
Deseasonalization, local normalization
$\mathbf{X}_i=\ln(\mathrm{Input}_{i-K..i}/l_i/s_{i-K..i})$
RNN stack
$\mathbf{F}_i=\mathrm{NN}(\mathbf{X}_i)$
Reseasonalization, denormalization
$\hat{Y}_i=\exp(\mathbf{F}_i)*l_i*s_{i+..i+H}$

Figure 2: Data flow and system architecture for the single seasonality case. X_i is the normalized, deseasonalized, and squashed input to the NN. F_i is the NN output. $\hat{Y}_i$ is the forecast, covering outputs i+1..i+H, where H is the forecasting horizon. l_i is scalar, the last level in the input window. X_i, $\hat{Y}_i$, F_i are vectors. Ensembling not shown.

time. For instance, the level and the seasonal components, as well as a Recurrent NN state, are local states.

- **Global constants:** These parameters reflect the patterns learned across large sets of series and are constant, not changing as we step through a series. For example, the weights used for the NN systems are global constants.

Typical statistical time series methods are trained per series, meaning that they involve only local constant and local state parameters. On the other hand, standard ML methods are usually trained on large datasets, involving only global parameters. The hybrid method described here uses all three types of parameters, being partly global and partly time series specific. This type of modeling becomes possible through Dynamic Computation Graph (DCG) systems, such as Dynet (Neubig et al., 2017), Pytorch (Paszke et al., 2017) and Tensorflow in "eager mode" (Abadi et al., 2015). The difference between static and dynamic computational graph systems is the latter ability to recreate the computational graph (built behind the scenes by the NN system) for each sample, here, for each time series. Therefore each series may have partially unique and partially shared model.

The architecture deployed was different for each frequency and output type (point forecast or prediction intervals).

At a high level the NNs of the model are Dilated LSTM-based stacks (Chang et al., 2017), sometimes followed by a non-linear layer and always followed by a linear "adapter" layer whose objective is to adapt the size of the state of the last layer to the size of the output layer (forecasting horizon or twice the forecasting horizons in case of prediction intervals (PI) models). The LSTM stacks are composed of a number (here 1-2) of blocks. In case of two (and theoretically more) blocks, output of a block is added using Resnet-style shortcuts (He et al., 2015) to the next block's output. Each block is a sequence of 1 to 4 layers, belonging to one of the three types of Dilated LSTMs: standard (Chang et al., 2017), with Attention mechanism (Qin et al., 2017) and a special Residual version (Kim et al., 2017).

Dilated LSTMs use, as part of input, the hidden state from previous, but not necessarily latest, steps. In standard LSTMs and related cells, at a time t, part of the input is the hidden state from step t-1. In a cell that is k-dilated, e.g. 3, the hidden state is taken from step t-k, so here t-3. This improves long-term memory performance. As it is customary for Dilated LSTMs (Chang et al., 2017), they were deployed in stacks of cells with increasing dilations. Similar blocks of standard, non-dilated LSTMs performed a bit worse. Even bigger drop of performance would have happened if the recurrent NNs had been replaced with non-recurrent ones, indicating that the RNN state is useful while dealing the time series and sequences more generally.

The general idea of Recurrent NN Attention mechanism is that instead of using the previous hidden state as in standard LSTMs, or the delayed state as in case of Dilated LSTMs, one calculates weights that are applied to a number of past hidden states to create an artificial weighted average state. This allows the system to "concentrate" or "attend" dynamically to a particular single or a group of past states. My implementation is an extension of Dilated LSTM, so the maximum look-behind horizon is equal the dilation. In case of weekly series, the network was composed of single block with two layers, encoded as Attentive (1,52). The first layer dilation is equal 1, so it is a standard LSTM, but the second layer does calculate weights over past 52 hidden states (when they become available, so at the point 53 or later while stepping through a series). The weights are calculated with a separate, but embedded into the LSTM, standard 2-layer NN; its inputs being concatenations of the LSTM input and last hidden state, and its weights adjusted by the same Gradient Descent mechanism operating on all other parameters.

Figure 3 shows three examples of configurations, the first one generating point forecasts (PFs) for the quarterly series, the second one PFs for the monthly series, while the third one prediction intervals (PIs) for the yearly series:

(a) The NN consists of two blocks, each one involving two Dilated LSTMs, connected by a shortcut around the second block. The final element is the "adapter layer" - it is just a standard linear layer (the transfer function equals identity) that adapts hidden output from the fourth layer (the one with dilation=8), usually 30-40 long, into expected output size, e.g. here 8.

(b) The NN consists of a single block composed of 4 dilated LSTMs, with residual connections as by Kim et al. (2017). Please note that the shortcut arrows point correctly into the inside of the Residual LSTM cell - this is a non-standard residual shortcut.

(c) The NN consists of a single block composed of two dilated LSTMs with Attention mechanism, followed by a dense non-linear layer (with tanh() activation), then by a linear adapter layer of the double size of the output, so that forecasts of both lower and upper bounds are generated simultaneously. The attention mechanism (Qin et al., 2017) slows calculations considerably, but occasionally appeared best.

Later, I provide a table that lists architectures and hyperparameters for all, not just these three, cases. Please keep in mind that the graph shows just the global parts of the models, but equally important are per-series parts.

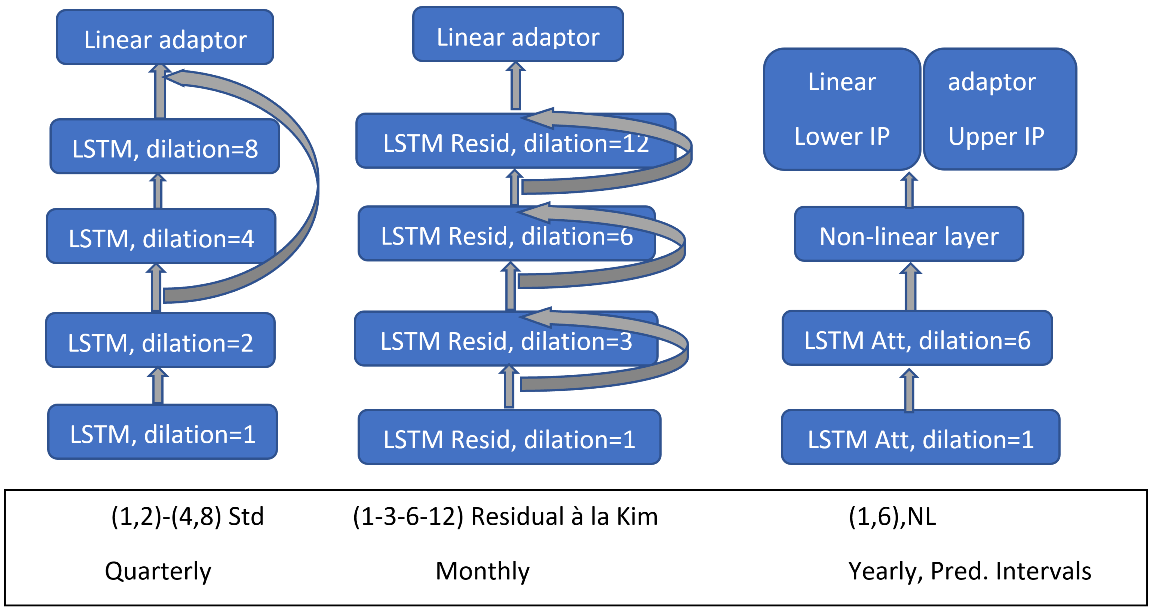

Figure 3: NN architectures used for generating some of the PFs and PIs.

3. Implementation details

This section provides more implementation details about the hybrid method. This includes information about the loss function, hyperparameters of the models, as well as the ensembling procedures.

3.1. Loss function

3.1.1. Point Forecast

The error measure used in the M4 Competition for the case of the PFs was a combination of the symmetric Mean Absolute Error (sMAPE) and the Mean Scaled Error (MASE) (Makridakis et al., 2018b). The two metrics are quite similar in nature, in a sense that both are normalized absolute differences between the predicted and the actual values of the series. Recalling that the inputs to the NN are already deseasonalized and normalized in this system, I postulated that the training loss function does not need to include normalization, it could be just a simple absolute difference of the target values and the predicted ones. However, during backtesting it became apparent that the models tend to have a positive bias. This was most likely the result of applying a squashing function, $log()$, to time series-derived inputs and outputs to the NN. The system learned in the log space, but the final forecast errors are calculated back in the linear space. To counter it, a pinball loss with a τ value a bit smaller than 0.5 (typically 0.45-0.49) was used. The pinball loss is defined as follows:

$$\begin{aligned} L_t &= (y_t - \hat{y}_t)\tau, \text{if } y_t \geq \hat{y}_t \\ &= (\hat{y}_t - y_t)(1-\tau), \text{if } \hat{y}_t > y_t \end{aligned} \tag{7}$$

So, the pinball function is asymmetric, penalizing differently for an actual being above and below a quantile, allowing the method to deal with the bias. It is an important loss function on its own - minimizing on it produces quantile regression (Takeuchi et al., 2006).

3.1.2. Prediction Intervals

The pinball loss function could have been adopted for generating the PIs as well. The requested coverage was 95%, so one could try to forecast 2.5% and 97.5% intervals. However, the competition metric for PIs was not based on separate upper and lower pinball losses, it was a single formula called Mean Scaled Interval Score (MSIS) (Makridakis et al., 2018b). Once again, the denominator of the MSIS was omitted given that the input to the NNs was already deseasonalized and normalized. It is noted that, although the method provided the most precise PIs among the submitted methods, the positive bias mentioned above is also observable for the case of the PIs, with the upper interval being exceeded less frequently than the lower one.

At this point I would like to draw the reader's attention to the great practical feature of the NN-based systems: ease of creating a loss function that is aligned with business/scientific objective. For this application, the loss functions were aligned with the accuracy metrics used in the M4 Competition.

3.1.3. Level wiggliness penalty

Intuitively, the level should be a smooth version of the time series, with no seasonality patterns. One would think that this is of secondary importance and more like an aestheticlevel requirement. However, it turned out that the smoothness of level helped substantially the forecasting accuracy. It appears that when the input to the NN was smooth, the NN concentrated on predicting the trend, instead of over fitting on some spurious, seasonality-related patterns. A smooth level also means that the seasonality components properly absorbed the seasonality. In Functional Data Analysis, an average of squares of second derivative is a popular penalty against wiggliness of a curve (Ramsay & Silverman, 2002). But such a penalty may be too strict and not robust enough when applied to time series with occasional large shifts. In this regard, a modified version of this penalty was applied, calculated as follows:

- Calculate log differences, i.e. $d_t = log(y_{t+1}/y_t)$ where y_t is point t of the series
- Calculate differences of the above: $e_t = d_{t+1} - d_t$
- Square and average them for each series.

The penalty, multiplied by a constant parameter in range of 50-100, called Level Variability Penalty (LVP), was added to both PFs and PIs loss function. The level wiggliness penalty significantly affected the performance of the method and it conceivable that the submission would not have won the M4 Competition without it.

3.2. Ensembling and data subsetting

Two models (one for PFs and one for PIs) were built for each of the 6 single-frequency subsets (daily, weekly etc.). Each of the models was actually an ensemble at several levels, which are presented below.

3.2.1. Independent runs

A single run involves the full training of the models as well as the generation of forecasts for all the series of the subset. Yet, given that parameter initializations are random, each run produces a bit different forecast. Ensembling models constructed from different runs can mitigate the effect of randomness and decrease uncertainty. Backtesting indicated that increasing the number of runs above 6-9 range did not improve forecasting accuracy. As a result, the number of independent runs was limited accordingly.

3.2.2. Ensemble of Specialists or Simple Ensemble

When it was computationally feasible, as it turned out in all cases except monthly and quarterly series, instead of training a single model, several concurrently trained models, learning from different subset of series were used. This approach, called Ensemble of Specialists, was originally proposed by Smyl (2017) and is summarized below.

The main idea is that, when a dataset contains a large number of series from unknown sources, it is reasonable to assume that these could possibly be grouped in subsets, such that the overall forecasting accuracy would improve by using a separate model per group instead of a single one for the whole dataset. However, there is no straightforward way to perform the grouping task, as series from disparate sources may look and behave similarly. Moreover, clustering the series through generic metrics may not be useful for improving the forecasting accuracy.

In this regard, the Ensemble of Specialists algorithm trains concurrently a number of models (NNs and per-series parameters) and forces them to specialize in a subset of series. The algorithm is summarized as follows:

1. Create a pool of models, e.g. 7 models, and randomly allocate a part, e.g. half of the time series, to each model
2. For each model:
   (a) Execute a single training on the allocated subset
   (b) Record the performance for the whole training set (in-sample, average per all points of the training part of a series)
3. Rank the models for each series and then allocate each series to the top N, e.g. 2, best models
4. Repeat steps 2 and 3 until the average error in the validation area starts growing.

So, the final forecast for a particular series is the average of the forecasts produced by the top N models. The main assumption here is continuity: if a particular model is good at forecasting the in-sample part of the series, it will hopefully display accurate results at the out-of-sample part of the series as well. The architecture and the inputs used for the individual models remain the same. What is different, and actively manipulated between epochs, is the composition of the training data set for each model. Figure 4 shows an example allocation of 10 series among 7 models altogether and 2 top models.

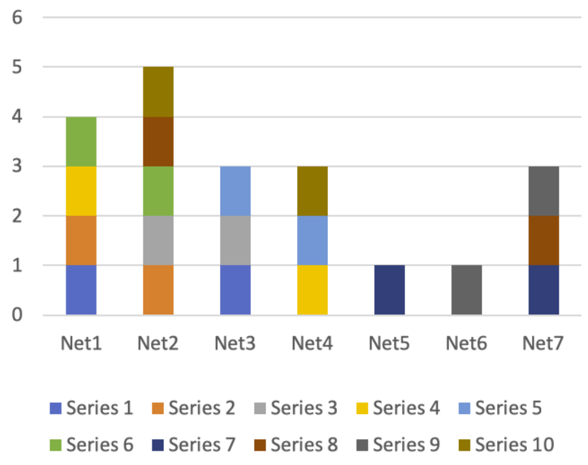

Figure 4: An example allocation performed by the Ensemble of Specialists algorithm for a set of 10 series to 7 models, using top 2 models per series.

A simpler approach, called here Simple Ensemble, was used instead of Ensemble of Specialists for monthly and quarterly data. In this case, at the beginning of each run, the data was split into two non-overlapping sets and then models were trained and forecasts made for each one of the two halves. This was a kind of bagging and worked well too.

It is worth mentioning that in the work of Smyl (2017), the Ensemble of Specialists improved forecasting accuracy by around 3% on M3 monthly data set. However the difference reported is not stable, depending on the data and the quality of the models used. Therefore, more work is needed to clearly delineate areas of superiority of each method.

3.2.3. Stage of training

The forecasts generated by a few (e.g 4-5) of the most recent training epochs were ensembled to provide a single forecast. The whole training typically used 10-30 epochs, so e.g. in case of 20 epochs, the final forecast was actually an average of the forecasts produced at epochs 16, 17, 18, 19 and 20.

3.3. Backtesting

Backtesting was implemented by removing typically one, but sometimes two, of last horizon-number of points from each series, e.g. 18 or 36 for monthly data, and training the system on a set with such shortened series. However, while the training steps were never exposed to the removed values, after every epoch the system was tested on the validation (removed) area, and the results guided the architectural and hyperparameter choices. In practice, there was a very strong correlation of the validation results when testing on the last horizon-number of points and penultimate horizon-number of points and the former approach was typically used, as it also admitted a larger number of series (many were too short to be used for backtesting if more than one horizon-number of points was removed).

While many series were short, there were also many that were very long, representing e.g. over 300 years of monthly data. Usefulness of the early parts of such series for the accuracy of the forecast was not obvious, while there was an obvious computational demand caused by them. In this respect, the long series were shortened, keeping just the most recent "Maximum Length" points. The Maxiumum Length hyperparameter was tested and increased from a relatively small value until no meaningful improvement in accuracy was observed in backtesting. It is listed, along with the rest of the other hyperparameters, in Table 1.

3.4. Hyper-parameters

All hyperparameters were chosen using some reasoning, intuition, and backtesting. The main tool to prevent over-fitting was the early stopping: during training, after every training epoch, average accuracy on the validation area (typically the last output horizon number of points, see Figure 1) was calculated. The epoch of lowest validation error was noted and used as the maximum number of epochs when doing the final (using all the data) learning and forecasting. Learning rate schedule was also decided by observing validation errors after every epoch.

Table 1 lists all NN architectures and hyperparameters used. If a PIs model used the same values as the PF model, the values are not repeated. A few comments about each:

- Ensembling

  Either Simple Ensemble or Ensemble of Specialists. In the latter case it is detailed as topN/numberOfAllModels, e.g. 4/5, see 3.2.2. In case of yearly data both ensembling methods were tried, but in cases of daily, weekly, and hourly data, the Ensemble of Specialists was chosen without experimentation, on belief that it should provide better results.

- NN architecture

  It is encoded as a sequence of blocks, in brackets, see 2.2.5. The residual shortcuts, around the blocks, or in the special case of LSTMs la (Kim et al., 2017) around the layers, are marked with dashes. Let me quickly describe the architecture of each type of the series:

  Monthly series used a single block of the special residual layers.

  Quarterly, daily, and hourly series used, what should be perhaps a standard architecture (as it appears to work well also in other contexts, outside of the M4 Competition): two blocks of two dilated LSTM layers.

  Point forecast models of yearly series used a single block of Dilated LSTMs with Attention, encoded as Attentive (1,6), while models for prediction intervals added a standard dense layer with tanh() activation, which I call Nonlinear Layer (NL), and therefore this is encoded as Attentive (1,6), NL.

  The architecture for weekly series, as described above, used Attentive LSTMs.

  As in other cases in the table, the architecture chosen was result of some reasoning/beliefs and experimentation. I believed that in case of seasonal series, at least one of the dilations should be equal seasonality, and another one be in the range of the prediction horizon. It is likely that the architecture was over-fitted to the backtesting results. E.g. more standard architectures (1,3)-(6,12) or (1,3,12) would almost certainly work also well for monthly series (without the special residual architecture).

- LVP

  LVP stands for Level Variability Penalty and is the multiplier applied to the level wiggliness penalty. It is applicable only to the seasonal models. The value is not very sensitive, changing it even by 50% would not make a big difference. Nonetheless it is important.

- Number of epochs

  The number of training epochs for the final training and forecasting runs was chosen experimentally as one that minimized error on the validation area. There was a clear interplay between learning rates and number of epochs: higher learning rates needed smaller number of epochs. Another factor influencing both of them was computational requirement of a subset: a larger number of series in a subset was forcing a preference for a smaller number of epochs (so higher learning rates).

- Learning rates

  The first number is the initial learning rate, often reduced during training, e.g. in case of the model for yearly PIs, it started with 1e-4, it was reduced at epoch 17 to 3e-5 and then again at epoch 22 to 1e-5. The schedule was result of observing behavior of validation errors after each training epochs. When they plateaued, for around 2 epochs, the learning rate was reduced by 3-10 factor.

- Max length

  This parameter lists maximum length of series used, see 3.3. In case of hourly series, there was no chopping, all series were used in their original length.

- Training percentile

  See 3.1.1

- State size of LSTMs

  LSTM cells maintain a vector of numbers, called state, their memory. The size of the state was not a sensitive parameter, values above 30 worked well. Larger values slow down calculations, but reduce a bit the number of epochs needed. There was no benefit for accuracy in using larger states.

Table 1: Details of architecture and parameters used.

Frequency	**PF**	**PIs**

Monthly	Simple ensemble
	Residual (1-3-6-12)
	LVP=50
	Epochs=10
	LR=5e-4
	Max length=272
	Training percentile=49
	State size of LSTMs=50
		Epochs=14
		LR=1e-3, {8,3e-4},{13,1e-4}
		Max length=512

Quarterly	Simple ensemble
	(1,2)-(4,8)
	LVP=80
	Epochs=15
	LR=1e-3, {10,1e-4}
	Max length=174
	Training percentile=45
	State size of LSTMs=40
		Epochs=16
		LR=1e-3, {7,3e-4},{11,1e-4}

Yearly	Ensemble of Specialists 4/5
	Attentive (1,6)
	LVP=0
	Epochs=12
	LR=1e-4, {15,1e-5}
	Max length=72
	Training percentile=50
	State size of LSTMs=30
		Attentive (1,6),NL
		Epochs=29
		LR=1e-4, {17,3e-5},{22,1e-5}

Daily	Ensemble of Specialists 4/5
	(1,3)-(7,14)
	Seasonality=7
	LVP=100	

Continued on next page

Table 1 – continued from previous page

Frequency	PF	PIs

	Epochs=13
	LR=3e-4, {9,1e-4}
	Max length=112
	Training percentile=49
	State size of LSTMs=40
		Epochs=21
		LR=3e-4, {13,1e-4}

Weekly	Ensemble of Specialists 3/5
	Attentive(1,52)
	Seasonality=52
	LVP=100
	Epochs=23
	LR=1e-3, {11,3e-4},{17,1e-4}
	Max length=335
	Training percentile=47
	State size of LSTMs=40
		Ensemble of Specialists 4/5
		
		Epochs=31
		LR=1e-3, {15,3e-4}

Hourly	Ensemble of Specialists 4/5
	(1,4)-(24,168)
	Seasonality=24,168
	LVP=10
	Epochs=27
	LR=1e-2,{7,5e-3},{18,1e-3},{22,3e-4}
	Max length=NA
	Training percentile=49
	State size of LSTMs=40
		
		Epochs=37
		LR=1e-2, {20,1e-3}

3.5. Implementation

The method was implemented through four programs: two using the Ensemble of Specialists and two using Simple Ensembling, as described earlier. Each group of two was composed of one program generating the PFs and another one for estimating the PIs. If the Competition were to happen today, probably just two programs would be needed, one using Ensemble of Specialists and another using the Simple Ensemble - both PIs and PFs can be generated from a single program by modifying the loss function and the architecture. The method was written in C++ relying on the Dynet library (Neubig et al., 2017). It can be compiled and run on Windows, Linux or Mac and can optionally write to a relational database, such as and SQL Server or MySQL, to facilitate the analysis of the backtesting results, very useful in practice. The programs are using CPU, not GPU, and are meant to be run in parallel. The code is publicly available at the M4 GitHub repository (https://github.com/M4Competition/M4-methods) to facilitate replicability and support future research (Makridakis et al., 2018a). The code is well commented and is the ultimate description of the method.

3.6. What did not work well and recent changes

The method generated accurate forecasts for most of the frequencies and especially for the monthly, yearly and quarterly ones. However, the accuracy was sub-optimal for the case of the daily and weekly data. This is partly explainable by the author's concentration on the "three big" subsets: monthly, yearly and quarterly, as they covered 95% of the data, and performing well on them was key to success in the Competition. However, subsequent work on daily and weekly data confirmed that the under-performance is a real problem.

Since the Competition ended, several improvements were attempted. Following is a description of one that made noticeable improvements in accuracy on daily and weekly data, bringing the performance to the level of best benchmarks: When analyzing the values of the smoothing coefficients, as they changed with passing training epochs, it became clear that they did not seemed to plateau in late epochs, the Gradient Descent did not seem to push them strongly enough. Therefore a separate, larger learning rate, e.g. being a multiple of 3 of the main learning rate, was assigned to them and that had the required effects. The smoothing coefficients changed quickly and eventually plateaued in late epochs.

4. Hybrid, hierarchical, and understandable ML models

In this section, the main features of the model are summarized and then generalizations and broader implications of its techniques and approaches are outlined. Also in this context, I retrace the steps that lead to formulation of the models described in this paper.

The winning solution was a hybrid forecasting method which mixes Exponential Smoothing-inspired formulas, used for deseasonalizing and normalizing the series, with advanced neural networks, exploited for extrapolating the series. Equally important was the hierarchical structure of the method, combining a global part learned across many time series (weights of the NN) with time series-specific part (smoothing coefficients and initial seasonality components). The third main component of the method was broad usage of ensembling, at multiple levels. The first two features were made possible by the great functionalities offered by the modern NN systems: Automatic Differentiation and Dynamic Computational Graphs (Paszke et al., 2017).

Automatic Differentiation allows to build models utilizing expressions made up of two sets: a quite broad list of basic functions, like sin(), exp(), etc., and a list of operators like matrix and elementwise multiplications, additions, reciprocal, etc. Neural Networks that use matrix operations and some nonlinear functions are just examples of the allowed expressions. The Gradient Descent machinery fits parameters of all of these expressions. In the model described here, there was both a NN and the non-NN part (Exponential Smoothing-inspired formulas). It is quite feasible to build models that encode complicated technical or business knowledge.

Dynamic Computational Graphs allow to build hierarchical models, with global and local (here, per time series) expressions and parameters. There could also be per-group parts. The models can be quite general e.g. in a classical, statistical vein:

Student performance = School impact + Teacher Impact + Individual impact

Note that each of the component can be a separate NN, an inscrutable black box. However, we can observe and quantify the impact of each of the black boxes, generally, and in each case. Therefore, we are getting a partially understandable ML model.

Automatic Differentiation is also a fundamental feature of Stan - a Probabilistic Programming Language (Carpenter et al., 2015). It fits models primarily with Hamiltonian Markov Chain Monte Carlo, so the optimization is different, but the underlying auto differentiation feels very similar. This similarity in modeling capabilities between Stan and Dynet led to the formulation of the proposed model, as described in more detail below.

By middle of 2016 I and my collaborators have successfully created extensions and generalizations of Holt and Holt-Winters models in Stan (I called this family of models LGT - Local and Global Trend models (Smyl & Zhang, 2015), (Smyl et al., 2019)) and experimented with using them together with NN models (Smyl & Kuber, 2016). Later, I have also experimented a lot building NN models for M3 Competition data set (Smyl, 2017). I was able to beat classical statistical algorithms on yearly (so non-seasonal) subset, but could not do it on the monthly subset. For seasonal series, I used STL decomposition as part of preprocessing, so clearly it did not work well. Also, in every category of M3 data, my LGT models were more accurate than my NN models. So, when I realized that Dynet, like Stan, allows to code freely a broad range of models, I decided to apply LGT ideas, like dealing with seasonality, into a NN model. And this is how the M4 winning solution was born.

Acknowledgement

I would like to thank Professor Spyros Makridakis and his colleagues for organizing the M4 Competition. I believe that forecasting competitions have been the main driver for advancing Deep Learning during its early years. Competitions enable comparison of various forecasting methods, exchanging of ideas and sharing the code. The competition dataset often becomes a valuable resource for years to come.
Finally, I would like to thank Evangelos Spiliotis for his help in editing this paper.

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., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., & Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., & Betancourt, M. (2015). The stan math library: Reverse-mode automatic differentiation in c++. *CoRR*, *abs/1509.07164*.

Chan, F., & Pauwels, L. L. (2018). Some theoretical results on forecast combinations. *International Journal of Forecasting*, *34*, 64 – 74.

Chang, S., Zhang, Y., Han, W., Yu, M., Guo, X., Tan, W., Cui, X., Witbrock, M., Hasegawa-Johnson, M., & Huang, T. S. (2017). Dilated Recurrent Neural Networks. *arXiv e-prints*, (p. arXiv:1710.02224).

Dimoulkas, I., Mazidi, P., & Herre, L. (2018). Neural networks for GEFCom2017 probabilistic load forecasting. *International Journal of Forecasting*, .

Gardner, E. S. (2006). Exponential smoothing: The state of the art-Part II. *International Journal of Forecasting*, *22*, 637–666.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *arXiv e-prints*, (p. arXiv:1512.03385).

Hyndman, R., Wang, E., & Laptev, N. (2015). Large-scale unusual time series detection. In *IEEE International Conference on Data Mining, 14-17 November 2015. Atlantic City, NJ, USA*.

Hyndman, R. J., Koehler, A. B., Ord, A. B., & Snyder, R. D. (2008). Forecasting with exponential smoothing: the state space approach. *Berlin: Springer Verlag*, .

Kang, Y., Hyndman, R. J., & Smith-Miles, K. (2017). Visualising forecasting algorithm performance using time series instance spaces. *International Journal of Forecasting*, *33*, 345–358.

Kim, J., El-Khamy, M., & Lee, J. (2017). Residual LSTM: Design of a Deep Recurrent Architecture for Distant Speech Recognition. *arXiv e-prints*, (p. arXiv:1701.03360).

Makridakis, S. (2017). The forthcoming artificial intelligence (AI) revolution: Its impact on society and firms. *Futures*, *90*, 46–60.

Makridakis, S., Assimakopoulos, V., & Spiliotis, E. (2018a). Objectivity, reproducibility and replicability in forecasting research. *International Journal of Forecasting*, *34*, 835–838.

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

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018c). Statistical and machine learning forecasting methods: Concerns and ways forward. *PLOS ONE*, *13*, 1–26.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., & Yin, P. (2017). Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, .

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in pytorch. In *NIPS-W*.

Petropoulos, F., Hyndman, R. J., & Bergmeir, C. (2018). Exploring the sources of uncertainty: Why does bagging for time series forecasting work? *European Journal of Operational Research*, *268*, 545 – 554.

Qin, Y., Song, D., Chen, H., Cheng, W., Jiang, G., & Cottrell, G. (2017). A Dual-Stage Attention-Based Recurrent Neural Network for Time Series Prediction. *arXiv e-prints*, (p. arXiv:1704.02971).

Ramsay, J., & Silverman, B. (2002). *Functional Data Analysis*.

Smyl, S. (2017). Ensemble of Specialized Neural Networks for Time Series Forecasting. In *37th International Symposium on Forecasting, 25-28 June 2017, Cairns, Australia*.

Smyl, S., Bergmeir, C., Wibowo, E., & Ng, T. W. (2019). *Rlgt: Bayesian Exponential Smoothing Models with Trend Modifications*. R package version 0.1-2.

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, June 2016, Santander, Spain*.

Smyl, S., & Zhang, Q. (2015). Fitting and Extending Exponential Smoothing Models with Stan. In *35th International Symposium on Forecasting, June 2015, Riverside, USA*.

Takeuchi, I., Le, Q. V., Sears, T. D., & Smola, A. J. (2006). Nonparametric quantile estimation. *Journal of machine learning research*, *7*, 1231–1264.

Taylor, J. W. (2003). Short-term electricity demand forecasting using double seasonal exponential smoothing. *The Journal of the Operational Research Society*, *54*, 799–805.

Weron, R. (2014). Electricity price forecasting: A review of the state-of-the-art with a look into the future. *International Journal of Forecasting*, *30*, 1030–1081.