International Journal of Forecasting 36 (2020) 75–85

Contents lists available at ScienceDirect

International Journal of Forecasting

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

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

Slawek Smyl

Uber Technologies, 555 Market St, 94104, San Francisco, CA, USA

ARTICLE INFO

Keywords:
Forecasting competitions
M4
Dynamic computational graphs
Automatic differentiation
Long short term memory (LSTM) networks
Exponential smoothing

ABSTRACT

This paper presents the winning submission of the M4 forecasting competition. The submission utilizes a dynamic computational graph neural network system that enables a standard exponential smoothing model to be mixed with advanced long short term memory networks into a common framework. The result is a hybrid and hierarchical forecasting method.

1. Introduction

Over recent decades, neural networks (NNs) and other machine learning (ML) algorithms have achieved remarkable success in various areas, including image and speech recognition, natural language processing (NLP), autonomous vehicles and games (Makridakis, 2017), among others. The key to their success is the fact that, given a large representative dataset, ML algorithms can learn to identify complex non-linear patterns and explore unstructured relationships without hypothesizing them a priori. Thus, ML algorithms are not limited by assumptions or pre-defined data generating processes, which allows the data to speak for itself.

However, the superiority of ML is not apparent when it comes to forecasting. While ML algorithms have been successful (Weron, 2014) in some applications like energy forecasting (Dimoulkas, Mazidi, & Herre, 2019), where the series being extrapolated are often numerous, long, and accompanied by explanatory variables, the performance of ML algorithms in more typical time series forecasting, where the data availability is often limited and regressors are not available, tends to be below expectations (Makridakis, Spiliotis, & Assimakopoulos, 2018b).

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

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

The preprocessing parameters were actually those from (slightly simplified) updating formulas of some models from the exponential smoothing family. Thus, 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 in a common framework. The ES equations enable the method to capture the main components of the individual series, such as seasonality and level, effectively, while the LSTM networks allow non-linear trends and cross-learning. In this regard, the data are exploited in a

E-mail address: slaweks@hotmail.co.uk.

https://doi.org/10.1016/j.ijforecast.2019.03.017
0169-2070/

hierarchical manner, meaning that both local and global components are utilized in order to extract and combine information at either a series or a dataset level, thus enhancing the forecasting accuracy.

The rest of the paper is organized as follows. Section 2 introduces the method and describes it in a general sense, while Section 3 gives more implementation details. Section 4 concludes by sketching some general modelling possibilities that are provided by recent NN systems and probabilistic programming languages, and, in this context, traces back the 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 in so doing, provides forecasts that are more accurate than those generated by either pure statistical or ML approaches, thus exploiting their advantages while avoiding their drawbacks. This hybrid forecasting approach has 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. The initial seasonality components (e.g. four for the quarterly series) and smoothing coefficients (two of them in the 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 the values of the series allows the seasonality components and levels to be calculated, and these are used for deseasonalization and normalization. The deseasonalization of seasonal series was very important in the M4 Competition, given that the series were provided as numeric vectors without any time stamp, so that there was no way of incorporating calendar features like the day of the week or the month number. Also, the series came from many sources, so their seasonality patterns varied.

The second element is a NN that operates 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 produce forecasts. The NN is global, learning across many time series.

The final element of the method is the ensembling of the 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 enhances the robustness of the method further, mitigating the model and parameter uncertainty (Petropoulos, Hyndman, & Bergmeir, 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 the sense that statistical modeling (ES models) is combined concurrently with ML algorithms (LSTM networks).
- It is hierarchical in nature, in the sense that both global (applicable to large subsets of all series) and local (applied to each series individually) parameters are utilized in order 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

The M4 time series, even within the same frequency subset, e.g. monthly, come from many different sources and exhibit a range of seasonality patterns. In addition, the starting dates of the series are not provided. In such circumstances, the NNs are unable to learn how to deal with seasonality effectively. A standard remedy is to apply a deseasonalization at preprocessing time. This solution is adequate but not ideal, as it separates the preprocessing from the forecasting completely, and the quality of the decomposition is likely to be worst near the end of the series, where it counts most for the forecast. One can also observe that the deseasonalization algorithms, however sophisticated and robust, were not designed to be good preprocessors for NNs. Classic statistical models, such as those from the exponential smoothing family, show a better way: the forecasting model has integral parts that deal with the seasonality.

In most NN variants, including LSTMs, the update size of weight w_{ij} is proportional to the final error, but also to the absolute value of the strength of the associated signal (coming from neuron i in the current layer to neuron j in the next layer). Thus, the NNs behave like analogue devices, even if implemented on a digital computer: small-valued inputs will have small impacts during the learning process. Normalizing each series globally to an interval like [0–1] is also problematic, as the values to be forecast may lie outside this range, and, more importantly, for series that change a lot over their lifespan, the parts of the series with small values will be ignored. Finally, information on the strength of trend is lost: two series of the same lengths and similar shapes, but one growing from 100 to 110 and another from 100 to 200, will look very similar after the [0–1] normalization. Thus, while normalization is necessary, it 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 all have positive values, the models of Holt (Gardner, 2006) and Holt and Winters (Hyndman, Koehler, Ord, & Snyder, 2008) with multiplicative seasonality were chosen. However, these were simplified by the removal of the linear trend: the NN was tasked to produce a trend that was most likely to be non-linear. Moreover, non-seasonal (yearly and daily data), single-seasonal (monthly, quarterly, and weekly data) or double-seasonal (hourly data) models (Taylor, 2003) were used, depending on the frequency of the data. The updating formulas for each case are as follows:

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 level, seasonality, and second seasonality components, respectively; K is the number of observations per seasonal period, i.e., four for quarterly, 12 for monthly and 52 for weekly; and, finally, L is the 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 zero and one. These restrictions can be implemented easily 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 the level and seasonality components to be calculated for all points of each series. These components are then 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 the resulting levels and seasonality components were different during each epoch.

The standard approach of constant size, rolling input and output windows was applied, as is shown in Fig. 1 for the 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, for seasonal series, it should cover at least a full seasonal period (e.g., being equal to or larger than four in the case of quarterly series), while 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, unlike in many other recurrent neural network (RNN) based sequence processing systems, the input size is larger than one. This works better because it allows the NN to be exposed to the immediate history of the series directly.

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

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

It is generally worthwhile to increase the size of the input window and extract more sophisticated features, like the strength of the seasonality or the variability, 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 summarize the characteristics of the series effectively, 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, Wang, & Laptev, 2015; Kang, Hyndman, & Smith-Miles, 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:

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

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 is the forecasting horizon. All operations are elementwise. The above is summarized in Fig. 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 that 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, evolving over time. For instance, the level and seasonal components, as well as a recurrent NN state, are local states.

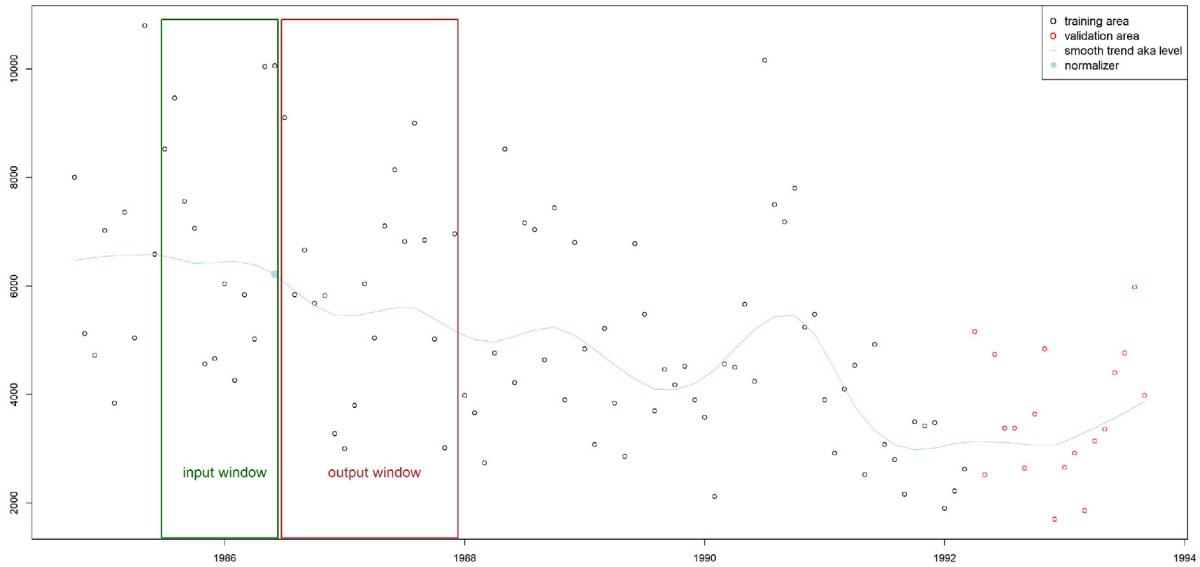

Fig. 1. An example showing how rolling windows are used for preprocessing a random monthly series. The last step is used for validation.

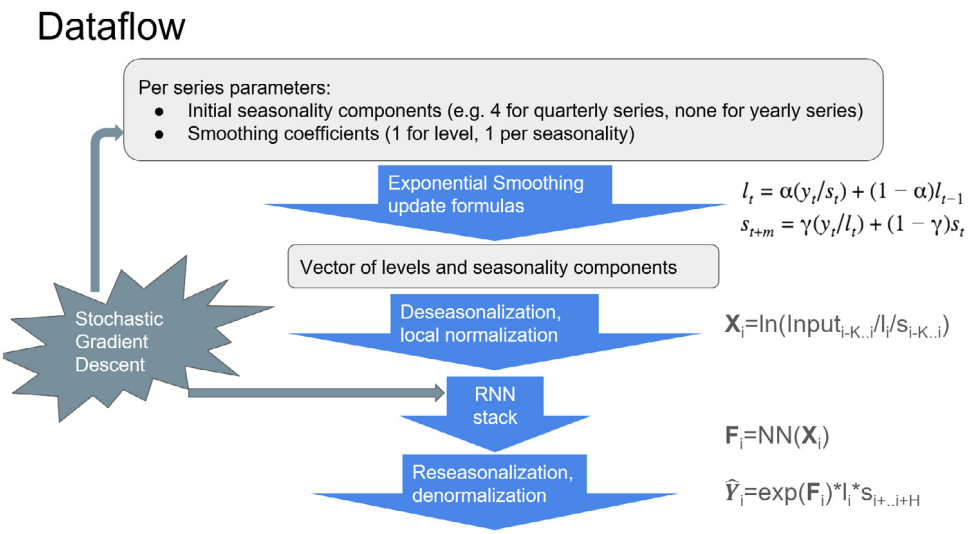

Fig. 2. Data flow and system architecture for the single seasonality case. X_i is the normalized, deseasonalized, and squashed input to the NN. F_i is the NN output. $\hat{Y}_i$ is the forecast, covering outputs $i+1 \ldots i+H$, where H is the forecasting horizon. l_i is scalar, the last level in the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

- **Global constants:** These parameters reflect the patterns learned across large sets of series and are constant; they do not change 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 on individual 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 the use of dynamic computation graph (DCG) systems, such as DyNet (Neubig, Dyer, Goldberg, Matthews, Ammar, Anastasopoulos, et al., 2017), PyTorch (Paszke, Gross, Chintala, Chanan, Yang, DeVito, et al., 2017) and TensorFlow in "eager mode" (Abadi, Agarwal, Barham, Brevdo, Chen, Citro, et al., 2015). The difference between static and dynamic computational graph systems is that the latter have the ability to recreate the computational graph (built behind the scenes by the NN system) for each sample, here, for each time series. Thus, each series may have a 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, Zhang, Han, Yu, Guo, Tan, et al., 2017), sometimes followed by a non-linear layer and always followed by a linear "adapter" layer, the objective of which is to adapt the size of the state of the last layer to the size of the output layer (the forecasting horizon, or twice the forecasting horizon in case of prediction interval (PI) models). The LSTM stacks are composed of a number of blocks (here 1–2). In case of two (and theoretically more) blocks, the output of a block is added to the next block's output using Resnet-style shortcuts (He, Zhang, Ren, & Sun, 2015). Each block is a sequence of one to four layers, belonging to one of the three types of dilated LSTMs: standard (Chang et al., 2017), with an attention mechanism (Qin, Song, Chen, Cheng, Jiang, & Cottrell, 2017) and a special residual version (Kim, El-Khamy, & Lee, 2017).

Dilated LSTMs use as part of their input the hidden state from previous, but not necessarily the latest, steps. In standard LSTMs and related cells, part of the input at a time t is the hidden state from step $t-1$. In a cell that is k-dilated, e.g. three, the hidden state is taken from step $t-k$, so here $t-3$. This improves long-term memory performance. As 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 slightly worse. Even bigger drops in performance would have happened if the recurrent NNs had been replaced with non-recurrent ones, indicating that the RNN state is useful for dealing with the time series and sequences more generally.

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

Fig. 3 shows three examples of configurations; the first one generates point forecasts (PFs) for the quarterly series, the second one PFs for the monthly series, and the third one prediction intervals (PIs) for the yearly series.

(a) The NN consists of two blocks, each one involving two dilated LSTMs, that are 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 the hidden output from the fourth layer (the one with dilation = 8), usually 30–40 long, into the expected output size (here eight).

(b) The NN consists of a single block composed of four dilated LSTMs, with residual connections as per (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 consisting of two dilated LSTMs with the 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 the calculations considerably, but occasionally appeared best.

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

3. Implementation details

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

3.1. Loss function

3.1.1. Point forecasts

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, Spiliotis, & Assimakopoulos, 2018a). The two metrics are quite similar in nature, in the sense that both are normalized absolute differences between the predicted and actual values of the series. Recalling that the inputs to the NN in this system are already deseasonalized and normalized, I postulated that the training loss function does not need to include normalization: it could be just a simple absolute difference between the target values and the predicted ones. However, it became apparent during backtesting that the models tend to have a positive bias, probably as a 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 this, 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:

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

Thus, the pinball function is asymmetric, penalizing actual values that are above and below a quantile differently so as to allow the method to deal with the bias. It is an important loss function on its own; minimizing

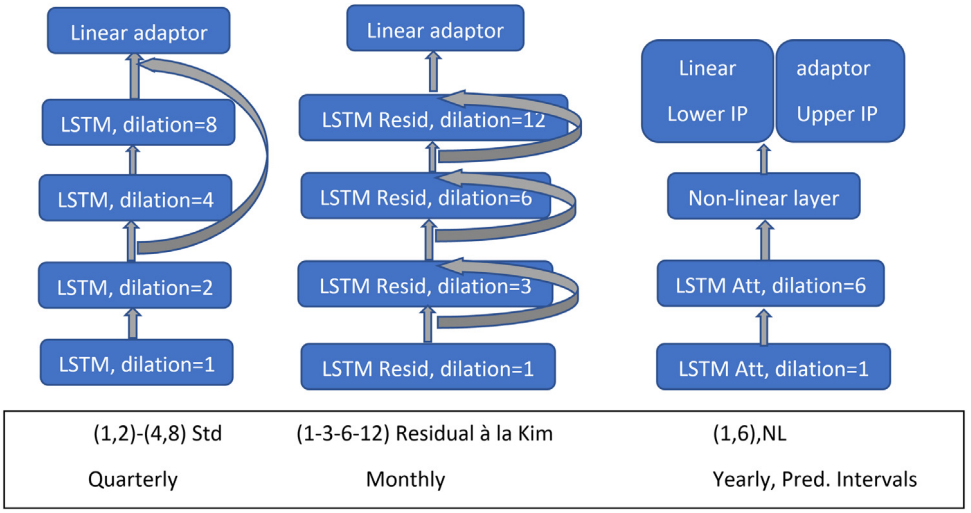

Fig. 3. NN architectures used for generating some of the PFs and PIs.

it produces quantile regression (Takeuchi, Le, Sears, & Smola, 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 have tried to forecast 2.5% and 97.5% intervals. However, the competition metric for PIs was not based on separate upper and lower pinball losses; instead, it was a single formula called the mean scaled interval score (MSIS) (Makridakis et al., 2018a). Once again, the denominator of the MSIS was omitted, since the input to the NNs was already deseasonalized and normalized. It should be noted that, although the method provided the most precise PIs among those of all methods submitted, the positive bias mentioned above can still be observed 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 NN-based systems: the ease of creating a loss function that is aligned with business/scientific objectives. 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 expect this to be of secondary importance, and more of an aesthetic-level requirement. However, it turns out that the smoothness of the level influenced the forecasting accuracy substantially. 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 absorbed the seasonality properly. In functional data analysis, an average of squares of second derivatives is a popular penalty against the wiggliness of a curve (Ramsay & Silverman, 2002). However, 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, which was 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.

This penalty, multiplied by a constant parameter in the range of 50–100, called the level variability penalty (LVP), was added to both PFs and PIs loss function. The level wiggliness penalty affected the performance of the method significantly, and it is conceivable that this 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 six 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 a full training of the models as well as the generation of forecasts for all series in the subset. However, each run for a given series produces a slightly different forecast, since the parameter initializations are random. Ensembling models constructed from different runs can mitigate the effect of randomness and decrease the uncertainty. Backtesting indicated that increasing the number of runs above the 6–9 range did not improve the forecasting accuracy, and 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 turned out to be the case for all except the monthly and quarterly series, several concurrently-trained models, learning from different subset of series, were used rather than training

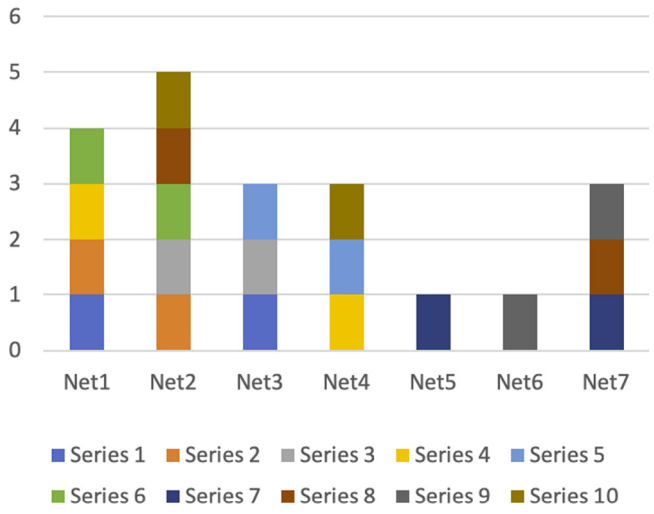

Fig. 4. An example allocation performed by the ensemble of specialists algorithm for a set of ten series to seven models, using the top two models per series.

a single model. This approach, called "ensemble of specialists", was proposed originally 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 if one used a separate model for each group instead of a single one for the whole dataset. However, there is no straightforward way of performing the grouping task, as series from disparate sources may look and behave similarly. Moreover, clustering the series using generic metrics may not be useful for improving the forecasting accuracy.

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

1. Create a pool of models (e.g. seven 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 over 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. two) best models.
4. Repeat steps 2 and 3 until the average error in the validation area starts growing.

Thus, 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 in the out-of-sample part of the series as well. The architecture and the inputs used for the individual models remain the same. What differs, and is manipulated actively, between epochs is the composition of the training data set for each model. Fig. 4 shows an example allocation of ten series among seven models altogether and two top models.

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

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

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; thus, in the case of 20 epochs for example, 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 the 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, the system was tested on the validation (removed) area after every epoch, and the results guided the architectural and hyperparameter choices. In practice, there was a very strong correlation

Table 1
Details of the 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 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}

between the validation results when testing on the last horizon-number of points and the penultimate horizon-number of points, with the former approach typically being used because 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, for example representing over 300 years of monthly data. The usefulness of the early parts of such series for the accuracy of the forecast was not obvious, while they involved an obvious computational demand. Thus, the long series were shortened, keeping only the most recent "maximum length" points. The maximum length hyperparameter was tested and increased from a relatively small value until no further meaningful improvement in accuracy was observed in backtesting. It is listed in Table 1 along with the rest of the hyperparameters.

3.4. Hyperparameters

All hyperparameters were chosen using some combination of reasoning, intuition, and backtesting. The main tool used for preventing over-fitting was early stopping: during training, the average accuracy on the validation area (typically the last output horizon number of points,

see Fig. 1) was calculated after every training epoch. The epoch with the lowest validation error was noted and used as the maximum number of epochs when doing the final (using all of the data) learning and forecasting. The learning rate schedule was also decided by observing the validation errors after every epoch.

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

- Ensembling
  Either the simple ensemble or the ensemble of specialists. In the latter case it is detailed as top$N$/numberOfAllModels, e.g. 4/5, see Section 3.2.2. In the case of yearly data, both ensembling methods were tried, but in the cases of daily, weekly, and hourly data, the ensemble of specialists was chosen without experimentation, under the belief that it should provide better results.
- NN architecture
  It is encoded as a sequence of blocks, in brackets, see Section 2.2.5. The residual shortcuts around the blocks, or, in the special case of LSTMs as per Kim et al. (2017), around the layers, are marked with dashes. Let me quickly describe the architecture of each type of series:
  - Monthly series used a single block of the special residual layers.
  - Quarterly, daily, and hourly series used what perhaps should be a standard architecture (as it appears to work well in other contexts, outside of the M4 Competition, too): 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 the 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 to the seasonality, while another should be in the range of the prediction horizon. It is likely that the architecture was over-fitted to the backtesting results; for example, the more standard architectures (1,3)–(6,12) or (1,3,12) would almost certainly work well for monthly series too (without the special residual architecture).
- LVP
  LVP stands for level variability penalty, and is the multiplier that is applied to the level wiggliness penalty. It applies only to the seasonal models. The value is not very sensitive, as changing it even by 50% would not make a big difference. However, it is still important.
- Number of epochs
  The number of training epochs for the final training and forecasting runs was chosen experimentally as the one that minimized the error on the validation area. There was a clear interplay between the learning rate and the number of epochs: higher learning rates needed smaller numbers of epochs. Another factor that influenced them both was the computational requirement of a subset: a larger number of series in a subset forced a preference for a smaller number of epochs (and thus higher learning rates).
- Learning rates
  The first number is the initial learning rate, which was often reduced during training; for example, in case of the model for yearly PIs, it started at 1e−4, but was reduced to 3e−5 at epoch 17 and again to 1e−5 at epoch 22. The schedule was result of observing the behaviors of the validation errors after each training epoch. When they plateaued for around two epochs, the learning rate was reduced by a factor of 3–10.
- Max length
  This parameter lists maximum length of series used, see Section 3.3. In the case of hourly series, there was no chopping, all series were used in their original length.
- Training percentile
  See Section 3.1.1.
- State size of LSTMs
  LSTM cells maintain a vector of numbers, called the state, which is their memory. The size of the state was not a sensitive parameter, with values above 30 working well. Larger values slow down the calculations, but reduce the number of epochs needed slightly. There was no benefit in accuracy of using larger states.

3.5. Implementation

The method was implemented through four programs: two using the ensemble of specialists and two using simple ensembling, as was described earlier. Each pair consisted of one program for generating the PFs and another for estimating the PIs. If the competition were to happen today, probably only two programs would be needed, one using the ensemble of specialists and another using the simple ensemble, as 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 SQL Server or MySQL, to facilitate the analysis of the backtesting results, which is very useful in practice. The programs use CPU, not GPU, and are meant to be run in parallel. The code is available publicly at the M4 GitHub repository (https://github.com/M4Competition/M4-methods) to facilitate replicability and support future research (Makridakis, Assimakopoulos, & Spiliotis, 2018). 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, but especially the monthly, yearly and quarterly ones. However, the accuracy was sub-optimal for the cases of the daily and weekly data. This can be explained in part 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 under-performance on these frequencies is a real problem.

Since the competition ended, several improvements have been attempted. One such attempt that achieved noticeable improvements in accuracy on the daily and weekly data, bringing the performance to the level of the best benchmarks, is as follows. When analyzing the values of the smoothing coefficients, as they changed with passing training epochs, it became clear that they did not seem to plateau in late epochs, as the gradient descent did not seem to push them strongly enough. Thus, a separate, larger learning rate, which was a multiple of three of the main learning rate, was assigned to them, and this had the required effect. The smoothing coefficients changed quickly and eventually plateaued in late epochs.

4. Hybrid, hierarchical, and understandable ML models

This section begins by summarizing the main features of the model, then outlines generalizations and broader implications of its techniques and approaches. Also in this context, I retrace the steps that lead to the formulation of the models described in this paper.

The winning solution was a hybrid forecasting method which mixed 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, which combined a global part learned across many time series (weights of the NN) with a time series specific part (smoothing coefficients and initial seasonality components). The third main component of the method was a broad usage of ensembling, at multiple levels. The first two features were made possible by the great functionalities offered by the modern NN systems of automatic differentiation and dynamic computational graphs (Paszke et al., 2017).

Automatic differentiation allows the building of models that utilize 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 element-wise 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 a 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 the building of 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:

$$\text{Student performance} = \text{School impact} + \text{Teacher impact} + \text{Individual impact}.$$

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

Automatic differentiation is also a fundamental feature of Stan, a probabilistic programming language (Carpenter, Hoffman, Brubaker, Lee, Li, & Betancourt, 2015). It fits models primarily using 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 is described in more detail below.

By the middle of 2016, I and my collaborators had successfully created extensions and generalizations of the Holt and Holt-Winters models in Stan (I called this family of models LGT: local and global trend models; see Smyl & Zhang, 2015, and Smyl, Bergmeir, Wibowo, & Ng, 2019), and experimented with using them along with NN models (Smyl & Kuber, 2016). Later, I also experimented a lot with building NN models for the M3 Competition data set (Smyl, 2017). I was able to beat classical statistical algorithms on the yearly (and therefore non-seasonal) subset, but could not do it on the monthly subset. For seasonal series, I used STL decomposition as part of the preprocessing, so clearly it did not work well. Also, my LGT models were more accurate than my NN models in every category of the M3 data. Thus, when I realized that DyNet, like Stan, allows a broad range of models to be coded freely, I decided to apply LGT ideas, such as dealing with seasonality, to a NN model. That is how the M4 winning solution was born.

Acknowledgments

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 the advancement of deep learning during its early years. Competitions enable the comparison of various forecasting methods, the exchange of ideas and the sharing of code. In addition, 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., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. URL https://www.tensorflow.org/, 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*(1), 64–74.

Chang, S., Zhang, Y., Han, W., Yu, M., Guo, X., Tan, W., et al. (2017). Dilated recurrent neural networks. arXiv e-prints, arXiv:1710.02224.

Dimoulkas, I., Mazidi, P., & Herre, L. (2019). Neural networks for GEFCom2017 probabilistic load forecasting. *International Journal of Forecasting*, (in press).

Gardner, E. S. (2006). Exponential smoothing: The state of the art — Part II. *International Journal of Forecasting*, *22*(4), 637–666.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. arXiv e-prints, arXiv:1512.03385.

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

Hyndman, R., Wang, E., & Laptev, N. (2015). Large-scale unusual time series detection. In *IEEE international conference on data mining*.

Kang, Y., Hyndman, R. J., & Smith-Miles, K. (2017). Visualising forecasting algorithm performance using time series instance spaces. *International Journal of Forecasting*, *33*(2), 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, 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. (2018). Objectivity, reproducibility and replicability in forecasting research. *International Journal of Forecasting*, *34*(4), 835–838.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018a). The M4 Competition: Results, findings, conclusion and way forward. *International Journal of Forecasting*, *34*(4), 802–808.

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018b). Statistical and machine learning forecasting methods: Concerns and ways forward. *PLoS One*, *13*(3), 1–26.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., et al. (2017). DyNet: The dynamic neural network toolkit. arXiv preprint, arXiv:1701.03980.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., et al. (2017). Automatic differentiation in PyTorch. In NIPS 2017 autodiff workshop.

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*(2), 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, arXiv:1704.02971.

Ramsay, J., & Silverman, B. (2002). *Functional data analysis*. New York: Springer-Verlag.

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

Smyl, S., Bergmeir, C., Wibowo, E., & Ng, T. W. (2019). Rlgt: Bayesian exponential smoothing models with trend modifications. R package version 01-2.

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*.

Smyl, S., & Zhang, Q. (2015). Fitting and extending exponential smoothing models with Stan. In *35th international symposium on forecasting*.

Takeuchi, I., Le, Q. V., Sears, T. D., & Smola, A. J. (2006). Nonparametric quantile estimation. *Journal of Machine Learning Research (JMLR)*, *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*(8), 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*(4), 1030–1081.