Hierarchical temporal memory and recurrent neural networks for time series prediction : an empirical validation and reduction to multilayer perceptions

Hierarchical Temporal Memory and Recurrent Neural Networks for Time Series Prediction: An Empirical Validation and Reduction to Multilayer Perceptrons

Jakob Struye, Steven Latré

University of Antwerp - imec, IDLab
Middelheimlaan 1, 2020 Antwerp, Belgium

Abstract

Recurrent Neural Networks such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) are often deployed as neural network-based predictors for time series data. Recently, Hierarchical Temporal Memory (HTM), a machine learning technology attempting to simulate the human brain's neocortex, has been proposed as another approach to time series data prediction. While HTM has gained a lot of attention, little is known about the actual performance compared to the more common RNNs. The only performance comparison between the two, performed at the company behind HTM, shows they perform similarly. In this article, we present a more in-depth performance comparison, involving more extensive hyperparameter tuning and evaluation on more scenarios. Surprisingly, our results show that both LSTM and GRUs can outperform HTM by over 30 % at lower runtime. Furthermore, we show that HTM requires explicitly timestamped data to recognize daily and weekly patterns, while LSTM only needs the raw sequential data to predict such time series accurately. Finally, our experiments indicate that the temporally aware components of all considered predictors contribute nothing to the prediction accuracy. We further strengthen this claim by presenting equally or better performing Multilayer Perceptrons conceptually similar to the HTM and LSTM, disregarding their temporal aspects.

Keywords: HTM, LSTM, GRU, Time series prediction

1. Introduction

Time series forecasting is a vital activity in many different fields. It can help predict the weather [1], foresee demand for different products or resources [2] and help governments in taking measures against imminent traffic congestion [3]. In its most essential form, a time series is a set of observations V, with each $v_t \in V$ observed at time t [4]. The goal of time series forecasting is then, given V up to time n, to predict future data point v_{n+h} where h is the *horizon* or how far into the future predictions are to be made. Inputs X and outputs Y of a predictive system are often defined as $X = \{x_t : x_t = v_t, t \in [0\,..\,N]\}$ and $Y = \{y_t : y_t = v_{t+h}, t \in [0\,..\,N]\}$ where $N = |V| - h - 1$. Among the many predictive systems available, Recurrent Neural Networks (RNNs), which feature *memory* retaining aspects of previous inputs, and specifically their subtypes Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) have gained significant traction in time series prediction. Another predictive system attaining more popularity is Hierarchical Temporal Memory (HTM), based on current insights into the human brain's workings. Current understanding of its performance in this field is however limited, as the only published evaluation of its predictive capabilities is in a comparative study performed by Numenta, the company behind HTM, in Y. Cui et al. [5, 6]. Using one dataset, they show comparable performance between HTM and LSTM for time series forecasting. In this article, we investigate these performance claims and attempt to improve the performance of both predictors.

The **main contribution** of this article is twofold. First, we provide an elaborate and in-depth qualitative and quantitative performance comparison of HTM, LSTM and GRU. Compared to the state of the art, this performance comparison differs in three ways to make it much more realistic and relevant:

1. Extensive hyperparameter tuning optimizes performance
2. Several new scenarios and input data configurations are considered
3. We also evaluate a GRU predictor

Second, we propose and evaluate two new Multilayer Perceptron (MLP) predictors. They are inspired by the LSTM and HTM predictors, while disregarding their temporally aware components, leading to a significantly simpler design. We investigate whether these simplifications retain the predictive capabilities of their originals.

The remainder of this article is structured as follows. Section 2 explores related work on time series prediction. In Section 3, we provide an overview of MLP, RNN and HTM theory and how they apply to time series prediction. Section 4 covers our experimental setup, with results provided in Section 5. In Section 6, we investigate how to reduce both LSTM and HTM predictors to MLPs. In Section 7, we evaluate whether the best performing predictors are robust to a global trend in the time series. Finally, Section 8 concludes this article.

2. Related Work

Time series prediction has been an active field of research for decades. In this section, we provide a brief overview, focusing on more recent advances. Historically common forecasting methods are linear regression [7], Autoregressive Moving Average (ARMA) [4, 8] and the Kalman filter [9–11]. In addition to these linear models, a wide array of machine learning tools is often applied to predict more complex time series. This includes Support Vector Machines (SVMs) [12–14] and numerous Artificial Neural Networks (ANNs) such as MLPs [15], Deep Belief Networks (DBNs) [16, 17], Echo-State Networks (ESNs) [18–20] and Extreme Learning Machines (ELMs) [21–23]. To cope with uncertain and vague datasets, ANN-based models have recently been extended for fuzzy time series [24, 25]. Apart from fuzzy input data, fuzzy set theory has also been applied to neural network weights, improving noise tolerance and overall performance [26, 27]. Other recently proposed predictors include those capable of dealing with irregularly spaced time series, based on either Convolutional Neural Networks (CNNs) [28] or RNNs [29]. Lately, these RNNs, and specifically LSTM and GRUs, have been gaining considerable traction in time series prediction [30–33]. Historically, concerns about overparameterization and overfitting of ANN predictors have often been raised [34–36] and simpler models have been shown to outperform neural networks [37, 38]. More recent literature on neural network-based forecasting rarely mentions these worries as potential deal-breakers however. We theorize that recent advances in neural network regularization, with dropout [39, 40] at the forefront, have largely addressed these concerns.
One final forecasting method that is gaining more attention is based on HTM, a neuroscience-based machine learning approach developed at machine intelligence company Numenta [41]. While Numenta has shown how HTM can be

used as a pure time series predictor, research has so far been focused more on the related fields of anomaly detection [42–44] and pattern recognition [45]. In those fields, HTM is shown to be a competent tool. We cover the detailed workings of RNNs and HTM and how they can be applied to time series prediction in Sections 3.2 and 3.3.

3. Machine Learning for Prediction

In this article, we focus on machine learning-based approaches to time series prediction. Specifically, one contribution of this article is to provide an evaluation of MLPs [46], RNNs including LSTM [47] and GRUs [48], and HTM [49] in the domain of time series prediction. Therefore, this section provides an overview of these technologies and of how they are applied to time series prediction. We focus on the core concepts of the technologies and refer the reader to the works referenced above for more details.

3.1. Multilayer Perceptrons

An MLP is a basic ANN consisting of multiple sequential fully connected layers, each containing a number of neurons, all connected to every neuron in the previous layer. Given a matrix X representing the values in the previous layer, the fully connected layer's values are computed as $WX + b$, with W a weight matrix and b a bias vector. We will omit this bias vector in the remainder of this article for simplicity. During its learning process, the MLP is fed inputs and their expected outputs, and continuously tweaks its weight matrices to reduce the gap between the expected outputs and the final layer's actual outputs. How to tweak the weights is determined through *backpropagation*, a process where, starting from the final layer, each layer's weights' derivatives are computed. These derivatives indicate in which direction the weights should be tweaked to optimally improve the outputs' accuracy. MLPs have been shown to be effective in time series prediction [50, 51].

3.2. Recurrent Neural Networks

RNNs are commonly used as an architecture for modeling, learning and predicting temporal data. An RNN considers inputs within a context, determined by previously received inputs [52]. These networks thus need memory to retain (aspects of) previously received inputs. The classical RNN achieves this by maintaining a state s, updated at every time step using the formula

$$s_t = f(W \begin{bmatrix} x_t \\ s_{t-1} \end{bmatrix}) \tag{1}$$

where $W = \begin{bmatrix} W_{in} & W_{rec} \end{bmatrix}$ is a weight matrix consisting of input weights W_{in} and recurrent weights W_{rec}, and f is a nonlinear function. These weights W must be trained to attain a well-performing network. As the gradient of a time step is dependent on future time steps, regular backpropagation cannot be applied at every time step. An intuitive solution is to logically *unroll* the recurrent layer into a separate layer for each time step, all sharing their weights, and apply backpropagation to this unrolled representation. This is called Backpropagation Through Time (BPTT). A major issue with BPTT for classical RNNs is the phenomenon of vanishing gradients, meaning gradients approach zero, effectively halting learning [53]. Repeated multiplication with the same recurrent weight matrix W_{rec} in the unrolled network lies at the root of this issue. Avoiding it therefore requires modifications to the RNN structure. The following sections describe two such solutions.

3.2.1. Long Short-Term Memory

The most well-known RNN modification is LSTM. Originally proposed in 1997 by Hochreiter & Schmidhuber [54], the LSTM architecture explicitly addresses the issue of vanishing gradients through a gating mechanism [55, 56]. Figure 1 shows the LSTM structure, based on the following formulas:

$$i_t = \sigma(W_i \cdot (x_t; h_{t-1})) \quad \text{(input gate)} \quad (2)$$

$$f_t = \sigma(W_f \cdot (x_t; h_{t-1})) \quad \text{(forget gate)} \quad (3)$$

$$o_t = \sigma(W_o \cdot (x_t; h_{t-1})) \quad \text{(output gate)} \quad (4)$$

$$g_t = tanh(W_g \cdot (x_t; h_{t-1})) \quad \text{(candidate values)} \quad (5)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad \text{(cell state)} \quad (6)$$

$$h_t = o_t \odot tanh(c_t) \quad \text{(hidden state = output)} \quad (7)$$

The gating mechanism controls how much of the state flows through the system through pointwise multiplication. The input gate controls which information of the new input is added to the cell state, the forget gate controls which parts of the cell state to retain, and the output gate controls which parts of the new cell state to use as output. A combination of fully opened forget gates and fully closed input gates can maintain an unchanged state indefinitely. When new info does get incorporated into the state, this is done through a *constant error carousel* [57], in Equation 6. New information is simply added to the state, avoiding the repeating multiplications that lead to vanishing gradients.

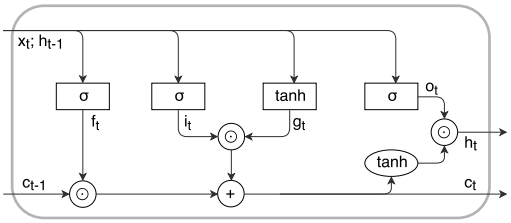


Figure 1: The inner workings of the LSTM cell given input x_t and previous hidden and cell states c_{t-1} and h_{t-1}. Rectangles are neural layers, applying backpropagation to their weights. Ovals are simple activation functions without learning. Circles represent pointwise operations. Merging lines concatenate, while splitting lines copy.

3.2.2. Gated Recurrent Unit

The LSTM structure is rather complicated, and features a large number of trainable weights, meaning training is a fairly slow process. Multiple modifications leading to simpler models with fewer weights have been proposed, the most notable being GRU [48]. GRU no longer differentiates between cell state and hidden state and instead considers the full cell state as output. This eliminates the need for an output gate, leading to 25 % fewer trainable weights. Additionally, the input and forget gates are merged into an *update gate*, while a new *reset gate* is introduced to control which information from the previous state is incorporated into the candidate values. The following formulas, visualized in Figure 2, define a GRU:

$$z_t = \sigma(W_z \cdot (x_t; h_{t-1})) \quad \text{(update gate)} \quad (8)$$

$$r_t = \sigma(W_r \cdot (x_t; h_{t-1})) \quad \text{(reset gate)} \quad (9)$$

$$g_t = tanh(W_g \cdot (x_t; (r_t \odot h_{t-1}))) \quad \text{(candidate values)} \quad (10)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot g_t \quad \text{(state = output)} \quad (11)$$

GRU still avoids vanishing gradients through its use of gates and the constant error carousel in Equation 11. GRU has been shown to perform comparably to the LSTM, with neither consistently outperforming the other [58].

3.3. Hierarchical Temporal Memory

HTM is a technology based on a theory of the working of the biological neocortex [59]. The neocortex is a part of the brain unique to mammals, involved in higher functions such as sensory perception, conscious movement and thought, and language. It accounts for over 75 % of the human brain's mass [60]. The principles of biological neurons, synapses and dendrites lie at

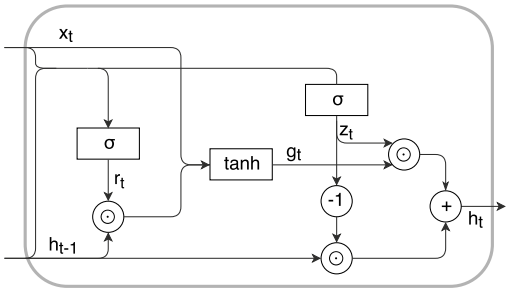


Figure 2: The inner workings of the GRU cell given input x_t and previous state h_{t-1}, using the same symbols as Figure 1.

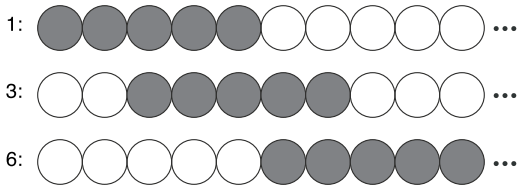


Figure 3: (Partial) SDRs assigned by a scalar encoder with $m = 5$ for the integers 1, 3 and 6, showing overlap between 1 and 3 and between 3 and 6 but not between 1 and 6

the core of HTM [41]. Furthermore, all data in HTM theory is represented using Sparse Distributed Representations (SDRs), a binary data type with relatively few active bits (i.e., with value 1) [61]. This follows the principle that in the brain, only a small percentage of the vast amount of neurons are active at any time.

HTM can function as a time series predictor. The HTM predictor consists of four components, of which the two middle ones comprise the core HTM system. This section provides an overview of all four components.

3.3.1. Encoders

HTM requires inputs in the form of fixed-length SDRs. As most data is by default not represented in this way, we need encoders to convert the data. A well-designed encoder assigns an SDR to each data point, such that two data points' similarity is proportional to the number of overlapping active bits in their SDR representations. Many simple encoding schemes have been proposed [62], mostly for data types where the concept of similarity between data points is clearly defined, such as numbers and locations. For example, integers between 0 and 100 could be encoded through a sliding window of m active bits over an array of length n (with $n \gg m$), such that adjacent

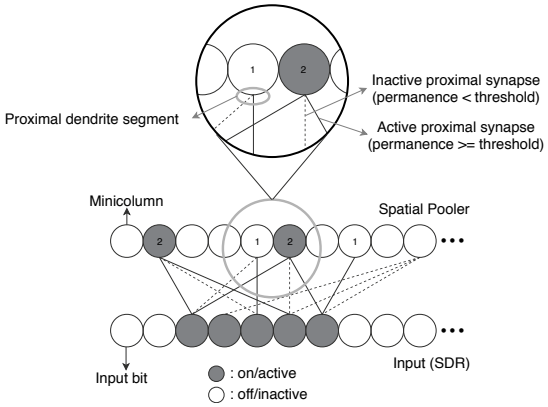


Figure 4: The input SDR connects to the spatial pooler's minicolumns through proximal synapses. Only the active synapses, whose permanence values exceed some threshold, can carry a signal. Based on this input, with 5 on bits shown, 4 of the minicolumns receive an input signal. All minicolumns are ranked on number of inputs received and the top $k\%$ become active. For simplicity, some bits, minicolumns and synapses are omitted.

integers are assigned partially overlapping windows. Figure 3 illustrates such a *scalar encoder*. More complex encoders, such as for words and by extension text, have also been proposed [63]. For other types of data, such as audio, no general encoders exist as of yet, to the best of our knowledge.

3.3.2. Spatial Pooler

After being encoded as SDRs, the inputs reach the spatial pooler. This component again outputs SDRs, now of a more strictly defined format. Most importantly, the spatial pooler guarantees a fixed sparsity of all SDRs [64]. This sparsity k is often set to 2 %, while encoders' outputs may exhibit sparsities of up to 35 % [49]. The spatial pooler achieves this as follows. It consists of a number (by default 2048) of *minicolumns*, each connected to a randomly chosen fraction (by default 0.5) of bits in the input SDR. Such connections are called *proximal synapses*, and the set of a minicolumn's proximal synapses form its *proximal dendrite segment*. Each of these synapses carries a randomly initialized *permanence* value between 0 and 1, and only those synapses with a permanence over some threshold (by default 0.5) are enabled, meaning they can carry a signal. Given an input, each minicolumn is assigned a score equal to the number of active bits in the input connected through enabled synapses. The top k percent (by default 2 %) of minicolumns ranked by descending score are activated, achieving fixed sparsity. Activated

minicolumns are then tuned to recognize similar inputs through classical Hebbian learning [65]: permanences of synapses to active bits are increased and the other permanences are decreased. Figure 4 summarizes the spatial pooler's behavior through an example.
Overall, the spatial pooler converts input SDRs to fixed-sparsity SDRs, while adapting to changing patterns in the input without losing information on similarity between data points.

3.3.3. Temporal Memory

The SDRs generated by the spatial pooler do not contain any temporal information; an input's SDR does not depend directly on the specific sequence of inputs preceding it. The next HTM component, the temporal memory, aims to reveal exactly these temporal patterns. It acts as an extension to the spatial pooler, subdividing each of its minicolumns into a fixed number of *neurons*. A minicolumn being active is then equivalent to one or more of its neurons being active. Similar to proximal synapses between minicolumns and input, *distal synapses* grow between neurons. These synapses are bundled into *distal dendrite segments*. As opposed to the proximal dendrite segment, of which exactly one exists per minicolumn, multiple distal dendrite segments may exist per neuron. When a neuron activates upon an input, it propagates a signal across all distal synapses on all of its segments. If the number of such signals a neuron receives on one segment exceeds some configurable threshold, it enters a *predictive* state. This indicates that the neuron expects to become active upon the next input. Each time a minicolumn activates, only its neurons in a predictive state activate. If there are none, all neurons activate instead —the minicolumn *bursts*. When a predictive neuron receives a new input, the synapses on its segments are reinforced through Hebbian learning, analogous to the proximal synapses. For any such segments with few synapses, new synapses grow to a randomly chosen subset of neurons that were active in the previous step. Overall, the activation of neurons within a minicolumn indicates in which context the minicolumn was activated.

3.3.4. SDR Classifier

Finally, the SDR classifier receives the set of activated neurons from the temporal memory and predicts the likelihood of each possible input occurring the next step. This classifier is simply a single fully connected neural network layer with a softmax activation function and cross-entropy loss function. The softmax activation function squashes its inputs I so that they sum to 1.

	RNN	**HTM**
Learning alg.	Backprop. (through time)	Hebbian
Network size	Problem-dependent	Usually fixed
Hyperparameters	Optimizer, learning rate, loss function, network size/architecture, etc.	Encoder design, Hebbian learning speed, boosting, etc.
Online learning	Initial training phase, occasional retraining	Learns continuously while predicting
Data types	Numeric, categorical, text, images, sound, etc.	SDR (requires encoder, distance metric)
Temporal aspect	Explicit state value	Predictive neurons in temporal memory

Table 1: General comparison of RNN and HTM

With cross-entropy loss, the difference between the predicted value Y_j and expected value T_j for every output contributes to the loss. The following functions define the activation function and loss:

$$\text{softmax}(I_i) = \frac{e^{I_i}}{\sum_{k=1}^{|I|} e^{I_k}} \tag{12}$$

$$\text{loss}_{\text{ce}}(Y) = \sum_{j=1}^{|Y|} -Y_j \log(T_j) \tag{13}$$

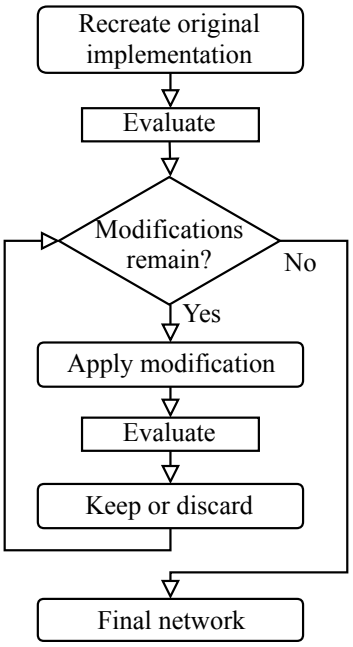
As the classifier outputs class probabilities, it cannot directly generate a prediction. The usual approach is to instead manually divide the output space into disjoint *buckets* and to have the classifier predict which bucket the future value will belong in.

3.3.5. Comparison to Neural Networks

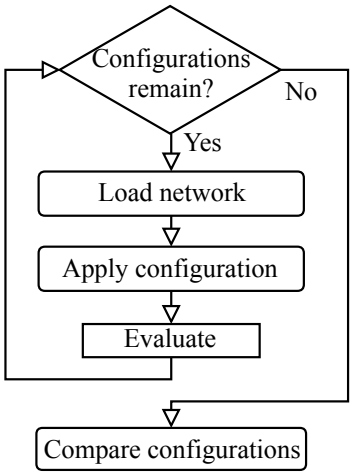
As shown by the descriptions above, RNNs and HTM are two vastly different technologies. We provide a brief general comparison in Table 1 and a comparison focusing on time series prediction in Table 2

	RNN	HTM
№ of inputs	One or more (explicit)	One or more (as 1 SDR)
Preprocessing	Normalization strategy	Partitioning into buckets
Output format	Exact value	Classification into bucket

Table 2: Comparison of RNN and HTM for time series prediction



(a) Main workflow



(b) Additional LSTM workflow

Figure 5: Workflow diagrams for time series predictor experiments. For each rectangular "Evaluate" step, the results are reported in Section 5. The left diagram concerns modifications to the networks, while the right diagram pertains to configurations of how to format the input data and present it to the network.

4. Experimental Setup

The original study by Cui et al. compared the predictive performance of HTM to other well-known systems, including LSTM. It used the Trip Record Data dataset on taxi rides published by the New York City Taxi and Limousine Commission[1], reduced to the number of taxi rides per 30 minutes over the period of one year, starting from July 2014. For each data point, the passenger count five steps into the future (i.e., 2.5 hours) was predicted.
This original study provided only an initial comparison and lacked a proper LSTM tuning and configuration. As such we evaluate several variations and extensions on the original experiments, including extensive LSTM hyperparameter tuning. In addition, we also consider a GRU predictor. Both the original code and our experiments are publicly available[2] [3]. In this section, we summarize the original experiments along with our extensions. Figure 5 outlines the main workflow for the experiments.

4.1. Hierarchical Temporal Memory

The HTM predictor was configured as follows. Each data point consisted of the number of passengers along with the day of the week ($\{d : d \in [0\,..\,6]\}$) and time of day in minutes ($\{30m : m \in [0\,..\,47]\}$), each encoded by a separate encoder. The passenger count, day and time encoders created SDRs of sizes 109, 100 and 600, respectively, each with 29 on bits. We assume the encoders were tuned manually for maximum performance. The concatenated encoders' outputs were then fed to the HTM with both the spatial pooler and temporal memory enabled. For the SDR classifier, a separate encoder provided target samples to the softmax layer. This additional encoder generated SDRs of only size 50, meaning it divided passenger counts into 22 equally sized buckets. For each of the 22 buckets, a moving average of the actual passenger counts of data points classified in that bucket was maintained. Of all buckets, the one with the highest probability was selected as the prediction. Before forecasting any results, a pretraining step fed the first 5000 inputs to the spatial pooler for 5 iterations, to start predicting with already reinforced spatial pooler synapses.

[1] http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

[2] https://github.com/numenta/htmresearch/tree/master/projects/sequence_prediction/continuous_sequence

[3] https://github.com/JakobStruye/timeseries-comparison

From these original experiments, it is unclear how significantly each component of the predictor contributes to the overall performance. We investigate whether the system can still generate accurate predictions when either or both of the main HTM components are disabled. This will also provide insight into how much of the overall performance is contributed by the classifier alone. Next we experiment with extending the pretraining step in an attempt to improve overall performance.

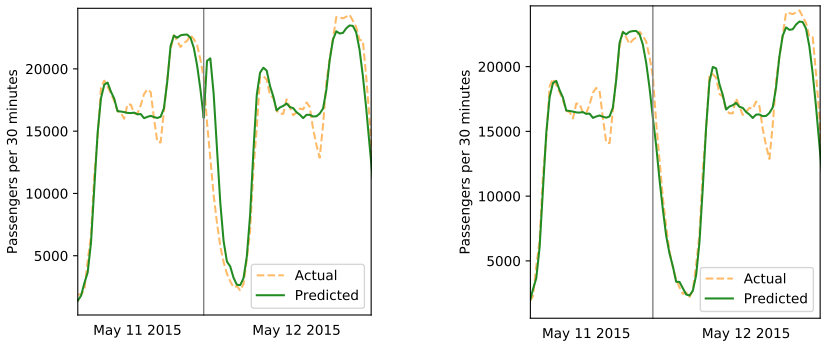
4.2. Long Short-Term Memory

The performance of the HTM predictor was compared to that of other sequence learning algorithms, of which LSTM performed best. This predictor consisted of a simple neural network with one LSTM layer of 20 units followed by a fully connected layer with linear activation function, providing the final prediction as its single output. Input data consisted of the passenger counts along with day of week and time of day data, each normalized to have a mean of 0 and a standard deviation of 1. The model was retrained after every week of data (i.e., 336 data points) with the 1000, 3000 or 6000 most recent data points for 100 epochs. In the remainder of this section, we propose modifications to this configuration and consider different ways of feeding the input to the network.

4.2.1. Modifications

As Cui et al. used an outdated LSTM implementation, we updated this in several ways to be compatible with the current state of the art. Most importantly, we rely on the Adam optimizer [66], instead of the outdated Rprop- algorithm. Additionally, we propose the following improvements to the predictor:

- **Train size**: We lower the number of samples used per training step to 5000, as HTM pretraining also used the first 5000 samples.
- **Re-feed on retrain**: Manual inspection of the original results showed that the error spikes considerably following every retrain. This revealed a major issue with the original implementation: on each retrain, the model is fully reinitialized, while prediction does not re-feed previous samples. This means that, after a retrain at sample i, the predictor is immediately fed sample i while still having a zero state. This eliminates the state's contribution to the prediction, resulting in underpredictions



(a) Prediction without re-feeding (b) Prediction with re-feeding

Figure 6: In the original experiment (left), predictions were clearly erratic directly following retraining (vertical line). By re-feeding all previous inputs after retraining (right), this behavior disappears completely.

(or, once denormalized, predictions close to the mean). We can avoid this easily by first re-feeding all previous samples after every retrain step. Figure 6 illustrates this effect and its solution.

- **MAE loss**: we replace the Mean Squared Error (MSE) loss function with a Mean Absolute Error (MAE) loss function, as the accuracy measure is MAE-based
- **No reinitialization**: We do not reinitialize the network before retraining. This means retraining starts from weights based on similar data, instead of on random, small weights. As patterns stay relatively consistent across the entire dataset, retraining starts at a lower loss.
- **Retrain interval**: We retrain less frequently, which lowers runtime and may have an acceptable or even positive effect on accuracy.

4.2.2. Data Configurations

Next to the modifications proposed above, we also propose three different *configurations* of how to present the input data to the predictor. First, we feed each sample one at a time and make a prediction at every step. This is the only configuration considered in the original study. This classic implementation of BPTT is extremely computationally expensive: with a training set of size 5000, it requires feeding 5000 samples and updating the state 5000

times, then backpropagating the error over 5000 unrolled layers for a single weight update. As a second configuration, we apply Truncated Backpropagation Through Time (TBPTT) [67], which greatly reduces the computational requirements. After every k samples, the algorithm unrolls for l steps and updates the weights [68]. Generally, this leads to both more numerous and computationally cheaper weight updates per epoch. We expand each data point to now contain a *sliding lookback window* of the previous l time steps of each feature. We then apply TBPTT(k, l) for $k = 1$ and a configurable l. As a third configuration, we again apply the TBPTT principle, but now consider the different time steps of one feature as separate, parallel features. While this removes the explicit temporal nature of the data by eliminating the LSTM state entirely, skipping the l state updates per prediction should improve runtime considerably. Depending on the impact on performance, this could be a viable approach.

4.2.3. Hyperparameter Tuning

For each of these three configurations, we perform extensive hyperparameter tuning for the number of hidden units, learning rate and, where applicable, lookback window and batch size. We first evaluate 1000 to 10 000 randomly chosen hyperparameter combinations, then continue searching in the vicinity of the best performing combination for another 100 to 1000 steps. Each random combination is run with a fixed number of epochs. To reduce runtime, the accuracy is calculated on samples 5500 up to 7500 only and retraining is disabled. Note that random search is generally considered an adequate choice for hyperparameter tuning at this scale [69, 70]. We select a perturbed version of the best performing combination to avoid overly optimistic results due to using a hyperparameter combination performing exceptionally well only for one fixed random seed.

4.3. Accuracy Measures

The performance of the predictors was evaluated using the Mean Absolute Percent Error (MAPE) measure. Instead of using the regular MAPE formula

$$MAPE = \frac{100}{n} \sum_{t=0}^{n-1} \left| \frac{T_t - Y_t}{T_t} \right| \tag{14}$$

all MAPEs listed in the original article are calculated using the alternative formula

$$MAPE = 100 * \frac{\sum_{t=0}^{n-1} |T_t - Y_t|}{\sum_{t=0}^{n-1} |T_t|} \tag{15}$$

which is simply the MAE divided by the average target value. The regular MAPE is known to penalize overestimating the target value more harshly than underestimating it [71]. In addition, exceptionally small data points lead to an exceptionally high Absolute Percent Error (APE) for those points. Y. Cui et al. likely adopted the alternative formula to avoid these effects. In the remainder of this article, all references to MAPE concern this alternative formula.

To avoid confusion with the common MAPE formula, we chose to introduce another accuracy measure: the Mean Absolute Scaled Error (MASE) [72]. As this measure is directly proportional to the modified MAPE, our results can still be compared directly to the original paper's. The MASE expresses the MAE of the predictor under analysis as a fraction of the MAE of a simple, naive predictor. For a highly seasonal dataset such as the one considered here, an obvious value to predict naively is the known value at the same point in the previous iteration of the seasonal cycle. Specifically, we take the measured value 24 hours (i.e., 48 data points) prior to the value to predict as the naive prediction. We modify the range of the dataset such that points for each $t \in [-48 \,..\, n-1]$ are known, and define the MASE as

$$MASE = \frac{\sum_{t=0}^{n-1} |T_t - Y_t|}{\sum_{t=0}^{n-1} |T_t - T_{t-s}|} \tag{16}$$

with s the length of the season (i.e., 48). A MASE of 1 means the analyzed predictor performs on average equally well as the naive predictor, while a MASE of 0.5 indicates its predictions are on average twice as close to the target value as with the naive predictor. In addition to the $\text{MASE}_{10\,000}$ (i.e., the MASE starting from the 10 000th sample) as reported in the original paper, we also report the MASE_{5500}, as initial (pre)training only uses the first 5000 samples. We foresee a buffer of 500 samples to accommodate for MASE calculation and any changes in training set size due to mini-batch learning. The MASE can easily be converted to the MAPE by multiplying it with the naive predictor's MAPE_{5500} of 18.2 % or $\text{MAPE}_{10\,000}$ of 17.8 %. We

also convert any accuracy measures taken from the original paper to MASE using this formula.

5. Results

In this section, we investigate the impact of our proposed modifications to the HTM and LSTM predictors, and evaluate the performance of a GRU predictor. Finally, we measure the impact of removing day and time info from the input.

5.1. Hierarchical Temporal Memory

Before starting our HTM experiments, we first note and fix a case of *information bleeding* in the original implementation. The classifier is always in learning mode, meaning it learns from every sample it receives. As such, the classifier will have learned info from sample $k-1$ when predicting sample k, while no info about samples past $k-5$ (with default horizon 5) should be known at that point. We address this by delaying classifier learning for 5 samples. The impact remains limited; the $\text{MASE}_{10\,000}$ increases from the original 0.455 to 0.460. For the remainder of this article, we worked with this fixed version of the system.

In these experiments, we repeat the HTM experiments using the original approach by Cui et al., but disable parts of the network. As a baseline, we run the experiment without any modifications. We then experiment with different configurations of the predictor. Table 3 provides an overview of the attained MASE. Unless specified otherwise, we only consider the MASE_{5500} in our results below. In addition, we report the runtime of all training and forecasting combined. Our experiments use the same hyperparameters as the original experiment. Attempts at hyperparameter tuning did not improve performance, and HTM is claimed to not require hyperparameter tuning due to insensitivity to its hyperparameter settings [49, 64].

We show that disabling the spatial pooler is not a viable approach: not only does performance decrease considerably, the runtime also increases. While removing the spatial pooler reduces total input size from 2048 to 800, the number of active bits more than doubles: the spatial pooler guarantees 2 % (≈ 41) active bits while encoder output has 87. This indicates that the temporal memory's runtime increases linearly with the number of *activated*

SP	TM	Pretrain	MASE_{5500}	$\text{MASE}_{10\,000}$	Runtime (s)
✓	✓	5x, SP	0.515	0.460	276
✗	✓	5x, SP	0.777	0.719	511
✓	✗	5x, SP	0.478	0.417	72
✗	✗	5x, SP	0.695	0.660	24
✓	✓	10x, SP	0.530	0.475	315
✓	✓	5x, SP+TM	0.478	0.439	816
✓	✓	10x, SP+TM	0.483	0.446	1270

Table 3: Performance of the HTM predictor when disabling components in the network, prolonging the pretraining step or also involving the Temporal Memory (TM) along with the Spatial Pooler (SP) in it. The runtime was measured on 4 2.5 GHz cores

bits. Having both components disabled also performs poorly, while only disabling the temporal memory actually improves performance and reduces runtime significantly. Excluding pretraining, the runtime is almost 7 times as low. This shows that, while the majority of the runtime is dedicated to the temporal memory, it fails to contribute anything to the final result and even worsens it. While this does not necessarily discredit the temporal memory's capabilities for other applications, we at least pose that a rather simple temporal dependency such as this time series does not require a complicated mechanism such as temporal memory to uncover it.

Next we modify the pretraining step, where by default the first 5000 data points were fed to the spatial pooler 5 times. Incorporating the temporal memory into pretraining provides a boost in performance at the cost of additional runtime. This still fails to overtake the order-of-magnitude faster spatial pooler-only configuration in accuracy. Increasing the number of epochs in the pretrain step failed to further improve accuracy. Overall, we were able to improve the HTM predictor's performance by 7.2 % while reducing its runtime by almost a factor 4.

5.2. Recurrent Neural Networks

In this section, we evaluate the effects of the modifications to the LSTM predictor proposed in Section 4.2.1 and of the different configurations covered in Section 4.2.2. We then compare the best performing LSTM predictor to a similarly configured GRU predictor. As RNN performance appeared sensitive to the initial random seed of the implementation, we perform each

experiment 10 times with different seeds, and report averaged results along with standard deviations.

5.2.1. Modifications

Before modifying and tuning the LSTM implementation, we first recreate the original configuration as closely as possible, using the original hyperparameters. As we replaced the outdated Rprop- optimizer, which features a dynamic learning rate, with the Adam optimizer, we had to determine an appropriate learning rate. Manual experimentation showed that a learning rate of 0.1 lead to performance similar to the original optimizer. This obtained an average $\text{MASE}_{10\,000}$ of 0.467, very close to the original 0.461.
Before attempting to improve the LSTM, we point out a case of information bleeding here as well. The dataset is normalized to have a mean of 0 and a standard deviation of 1 for each feature. As this is calculated across the entire dataset, a data point's normalized value may depend on future values. We fix this by instead normalizing using the mean and standard deviation of the first 5000 data points. This slightly increases the $\text{MASE}_{10\,000}$ to 0.475. We apply this normalization in all future experiments.

We now apply the modifications consecutively, and present the resulting accuracies in Table 4. These experiments show that all proposed improvements are effective. Increasing the retrain interval to 1000 even had a positive effect on accuracy, while increasing it further to 2500 only had a slight negative impact. Considering the $\text{MASE}_{10\,000}$, we reduced the error by 24.4 % through simple improvements only.

5.2.2. Data Configurations

Next we evaluate three different configurations of how to feed the data to the predictor and perform hyperparameter tuning for each of them. We perform tuning with fixed but manually determined epoch counts. Tables 5 and 6 provide overviews of respectively the tuned hyperparameters and their attained performance.

We first investigate the *continuous* configuration, which picks up where we left off in Section 5.2.1, retraining every 1000 samples. Hyperparameter tuning resulted in another improvement of 1.6 %.
The *lookback as timesteps* or *lb-ts* configuration with hyperparameter *lb* sim-

Modification	$MASE_{5500}$	$MASE_{10\,000}$
Base	-	0.475 (± 0.0100)
Train size 5000	0.499 (± 0.00944)	0.464 (± 0.0124)
Re-feed on retrain	0.459 (± 0.00830)	0.422 (± 0.0113)
MAE loss	0.451 (± 0.00972)	0.413 (± 0.00718)
No reinitialization	0.386 (± 0.00886)	0.373 (± 0.0111)
Retrain interval 1000	0.383 (± 0.00820)	0.359 (± 0.00834)
Retrain interval 2500	0.393 (± 0.00791)	0.365 (± 0.00686)

Table 4: Average error measures (± standard deviation) for all proposed modifications to the LSTM predictor. The modifications are applied consecutively.

Configuration	LR	Units	Batch	lb	Epochs (LSTM)	Epochs (GRU)
Continuous	0.02	130	1	-	200;100	300;100
Lb-ts	0.003	200	128	50	200;60	300;90
Lb-ft (optimal)	0.0015	180	512	75	300;100	400;125
Lb-ft (fast)	0.003	40	1024	50	120;25	-

Table 5: The tuned hyperparameters for the GRU and LSTM predictors. We use different epoch values for the GRU predictor as it learned slightly slower. The initial epoch value and retrain epoch value are set separately, as retraining starts from an already trained network instead of a randomly initialized network and thus requires fewer epochs to converge. We consider feeding data continuously while applying BPTT, feeding data in lookback windows while applying TBPTT, and a variation on this where the values in the lookback window are considered as different features instead of different timesteps of the same feature. For this final configuration we add a separate runtime-focused tuning for LSTM, as the optimal tuning was rather slow.

Configuration	$MASE_{5500}$	$MASE_{10\,000}$	Runtime (s)
Continuous	0.377 (± 0.00489)	0.350 (± 0.00709)	2679
Lb-ts	0.376 (± 0.00709)	0.365 (± 0.00533)	17323
Lb-ft (optimal)	0.324 (± 0.00133)	0.304 (± 0.00146)	483
Lb-ft (fast)	0.331 (± 0.00386)	0.312 (± 0.00548)	50

Table 6: Average accuracy measures (± standard deviation) for all tuned configurations using the LSTM predictor, along with their runtime in seconds.

Configuration	MASE_{5500}	$\text{MASE}_{10\,000}$	Runtime (s)
Continuous	0.377 (± 0.00685)	0.357 (± 0.00615)	3112
Lb-ts	0.381 (± 0.00936)	0.365 (± 0.0114)	19081
Lb-ft	0.331 (± 0.00153)	0.309 (± 0.00160)	601

Table 7: Average accuracy measures (± standard deviation) with the GRU predictor.

ulates TBPTT(1,*lb*), i.e., backpropagating for *lb* steps on every new sample. It has no effect on accuracy compared to the previous configuration, but mainly sees a drastic increase in runtime. Our implementation feeds each input *lb* times. A more efficient implementation could reduce runtime, however this result shows that the accuracy would not benefit from it.
Finally, in the *lookback as features* or *lb-ft* configuration we consider the entries in the lookback window as separate features instead of different time steps of the same features. This performs exceptionally well, outperforming even the best HTM predictor by 32.2 %. The HTM predictor however still wins out in terms of runtime. We therefore repeat hyperparameter tuning with intervals leading to low runtimes, finding a tuning that only sacrifices a fraction of the accuracy in exchange for reducing the runtime to 50 seconds. As such this configuration beats any HTM predictor in both runtime and accuracy. We were able to reduce the runtime by an order of magnitude, resulting in only a 2.2 % error increase.

Table 7 shows the performance of the GRU predictor. Except for the epoch values, this uses the same hyperparameters as the LSTM predictor. We did perform separate hyperparameter tuning for GRU but did not observe any improvement in performance. GRU performance is very close to LSTM performance, although runtime is slightly higher. The additional epochs required for GRU to converge did not outweigh the lower per-epoch runtime due to the lower weight count. There is no clear winner between the two RNN implementations in this experiment.

5.3. Ignoring Day and Time

Next to the passenger count, the day of week and time of day serve as additional input features in all experiments. As the main patterns in the data are daily, these additional features essentially provide a hint to the predictor on where to find these patterns. In this section, we investigate how

well the optimal configurations of the different predictors perform when this information is withheld. A predictor that still performs well in the absence of this information is more versatile. We first attempt this on the spatial pooler-only HTM predictor. It performs very poorly in this scenario, with the MASE increasing from 0.478 to 1.93, meaning this predictor's error is on average almost twice as high as that of a naive predictor. Clearly, the HTM predictor mainly relies on previous samples with similar day and time values to form predictions and is largely unable to extract patterns from just the passenger count feature.

Running the LSTM predictor using the passenger count as the single input feature leads to an average MASE of 0.364. While this is a noticeable decrease in performance from the 0.324 attained with all three features, it still better than any HTM predictor's performance with date and time info. This shows that the LSTM predictor is still able to detect seasonal patterns without any indications to the pattern's period, making it a more generally applicable predictor than the HTM predictor.

6. Reduction to Multilayer Perceptrons

In Section 5.2 we concluded that the LSTM performed best in the *lookback as features* configuration, in which the recurrent state is wiped after every step and therefore zero at the start of every step. In this section, we simplify the LSTM to eliminate memory from its structure entirely. The resulting network is an MLP. We then show that this idea is also applicable to the HTM predictor, by proposing an MLP conceptually equivalent to the spatial pooler-only HTM predictor. We implement and evaluate both MLP predictors. By demonstrating these MLPs we further corroborate our claim regarding the minimal contribution of the predictors' temporal aspects. In addition, the resulting MLPs may be viable predictors on their own given their accuracy, runtime and architectural simplicity.

6.1. Reducing the LSTM

We simplify the LSTM formulas in Section 3.2.1 knowing the previous state c_{t-1} will always be zero, meaning the forget gate f_t can be eliminated. Furthermore, the hidden state h_{t-1} is always zero as well, meaning each $W \cdot (x_t; h_{t-1})$ simplifies to $W_{in} \cdot x_t$, where W_{in} is the non-recurrent component

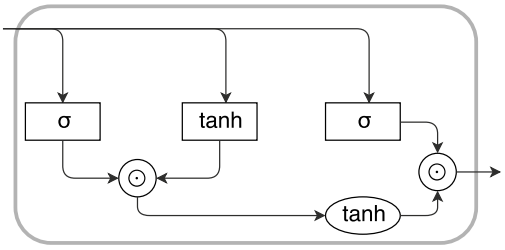


Figure 7: MLP equivalent of LSTM with all state eliminated. All elements are now implemented using separate layers.

of W. As such the expanded calculation for the output becomes:

$$h_t = \sigma(W_{o,in} \cdot x_t) \odot tanh(\sigma(W_{i,in} \cdot x_t) \odot tanh(W_{c,in} \cdot x_t)) \tag{17}$$

This simplified formula contains no references to the previous state, and in fact characterizes the simple MLP shown in Figure 7.

6.2. Reducing the HTM

Next we investigate whether we can design an MLP that is in concept similar to the spatial pooler-only HTM. Its classifier, which was simply a fully connected layer with softmax activation function, can simply be copied over to the MLP model. Next we observe that the encoders and spatial pooler together (to which we will refer as the *HTM encoder*) serve to convert the 3 input values to an SDR of size 2048 with 41 active bits. We can instead express this SDR as the 41 indices of the active bits. We can thus replace the HTM encoder with a fully connected *encoding layer* with 41 outputs corresponding to the indices of the active bits. This would however lose the periodicity of the day and time encoders: 30 minutes to midnight on a Sunday would no longer be considered close to 30 minutes past midnight on a Monday. We replace each of the periodic features by both a sine wave and a cosine wave with periods equal to the feature's period p. Using either wave ensures that points near the beginning and the end of the period are encoded similarly, while using both additionally ensures that no two points are unintentionally encoded the same way:

$$\sin(a) = \sin(b) \wedge \cos(a) = \cos(b) \iff \exists i \in \mathbb{Z} : b = a + ip \tag{18}$$

Figure 8 illustrates this point: only using the cosine wave would encode 6 A.M. ($0.25p$) and 6 P.M. ($0.75p$) equally. We adopted this technique from a

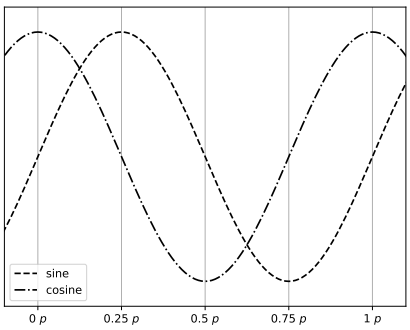


Figure 8: Encoding periodicity with sine and cosine waves

blog post by Ian London [73]. We can then feed these four waves along with the (normalized) passenger counts to our encoding layer. To further level the playing field between this layer and the HTM encoder, we limit each of the 41 output values to 2048 possible values. We achieve this by using an activation function with a known range and quantizing its output to one of 2048 possible values during the forward pass. During backpropagation we do use the non-quantized values. This method is inspired by recent work in binarized and quantized neural networks [74, 75]. Experimentation showed that quantized *tanh* performed best in this case.

6.3. Evaluation

We implemented and evaluated both MLP predictors. For the LSTM-inspired predictor, we evaluated the prediction accuracy using the same optimal hyperparameters of the *lookback as features* LSTM experiment. Averaged across 10 repetitions, the MLP reaches a MASE_{5500} of 0.324, and $\text{MASE}_{10\,000}$ of 0.305, nearly identical to its source model's 0.324 and 0.304, indicating the best performing LSTM can indeed be reduced to an MLP. This confirms that, at least for these simple seasonal time series, RNN memory does not increase predictive capabilities. In addition, this MLP runs 37 % faster.

Next we evaluate the HTM-inspired MLP predictor. After hyperparameter tuning as in Section 4.2.3, we attained a MASE_{5500} of 0.444 and a $\text{MASE}_{10\,000}$ of 0.399. Compared to the best HTM results of 0.478 and 0.418, this shows that our fully connected encoder significantly outperforms the HTM encoder. While with a runtime of 515 seconds, this MLP predictor is 7 times slower,

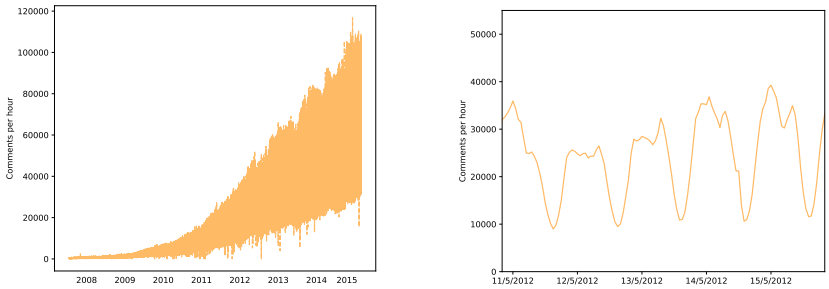


Figure 9: The Reddit dataset contains both a daily seasonal pattern and a global trend.

careful hyperparameter tuning aimed at reducing runtime would likely result in an MLP outperforming the HTM with lower runtime. Such tuning reduced a similar runtime by an order of magnitude in Section 5.2.2.

7. Robustness to Trend Information

The only dataset considered so far is *trend-stationary*, meaning its mean remains stable throughout the entire sequence. Some time series do however contain a clear global trend, leading to a changing mean and standard deviation over time. We apply our best performing predictors to such a time series. This dataset contains the hourly number of publicly visible comments posted on Reddit between October 15 2007 and May 31 2015. In this period, the average number of comments per hour increased from a few hundred to over fifty thousand. Figure 9 shows the data. This dataset, parsed from publicly available data, has not been used before in the literature.

We first apply the *lookback-as-features* LSTM predictor and its MLP equivalent to this dataset. We use the same hyperparameters as in Table 5. These predictors perform poorly on this dataset, with a MASE_{5500} of 0.948 and 0.925. The normalization, configured using only the first 5000 samples, no longer ensures an overall mean of 0 and a standard deviation of 1 due to the global trend. We instead apply Adaptive Normalization (AN) [76], an approach which divides the dataset into Disjoint Sliding Windows (DSWs) and normalizes each DSW separately. With this normalization, the LSTM and MLP predictors perform equally well, with a MASE_{5500} of 0.311 and 0.310.

We then apply the HTM predictor and its MLP-based alternative to this

dataset. As the range of values is three times as wide as with the previous dataset, we triple the number of buckets the predictor uses to 66. Without any other modifications, the HTM attains a MASE_{5500} of 0.541. The HTM's performance does not suffer greatly from the global trend, as it constantly adapts to the changing patterns in the input. The MLP variant did not perform as well on this dataset, as it struggled to adapt to the changing trend. Even when retraining every 200 steps, it only attained a MASE_{5500} of 0.632. It does however still outperform the regular LSTM without AN by 33.3 %, indicating it does not fail entirely at adapting to the global trend.

8. Conclusions

In this article, we provide the first in-depth and independent study of time series prediction performance of HTM, LSTM and GRU. In contrast to previously published work [5], we show that, through hyperparameter tuning and careful formatting of the data, the LSTM predictor outperforms the HTM predictor by over 30 % at lower runtime. The GRU predictor is slightly slower than its LSTM counterpart but performs similarly; their accuracy measures differ at by at most 1.3 %. In addition, we evaluate the effects of different traits of time series on the predictors' performance. For instance, the HTM is unable to predict a seasonal pattern in the absence of features indicating the pattern's period (e.g., timestamps), while the LSTM still performs reasonably well in this case. This indicates that HTM mainly bases its predictions on these secondary features, rather than actually uncovering underlying patterns in the time series. Next we show that the HTM predictor still performs reasonably well on a non-stationary time series, despite a noticeable decrease in performance. An unmodified LSTM fails to produce any reasonable predictions in this scenario. Modifying only the normalization algorithm leads to performance similar to the stationary case.

A more detailed study of these algorithms has also led to the design of conceptually similar but architecturally much simpler alternatives to both HTM and LSTM using MLP predictors, performing at least as well on trend-stationary data. As a result, we do not only disprove the claim that HTM slightly outperforms LSTM in this application, but also show that both may be unnecessarily intricate for this type of time series prediction. Overall, we were unable to discover a case where using HTM for time series prediction would be preferable to neural network-based approaches. Requiring encoders for all

input also severely restricts HTM's applicability. Finally, we note that there may be other fields where HTM is a more competitive tool. More research, such as very recent work on its applicability to anomaly detection [42, 43] is required to further investigate potential use cases of HTM.

Acknowledgements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

[1] S.-M. Chen, J.-R. Hwang, Temperature prediction using fuzzy time series, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 30 (2) (2000) 263–275. doi:10.1109/3477.836375.

[2] F. J. Nogales, J. Contreras, A. J. Conejo, R. Espinola, Forecasting next-day electricity prices by time series models, IEEE Transactions on Power Systems 17 (2) (2002) 342–348. doi:10.1109/TPWRS.2002.1007902.

[3] M. V. D. Voort, M. Dougherty, S. Watson, Combining kohonen maps with arima time series models to forecast traffic flow, Transportation Research Part C: Emerging Technologies 4 (5) (1996) 307 – 318. doi:10.1016/S0968-090X(97)82903-8.

[4] P. J. Brockwell, R. A. Davis, Introduction to Time Series and Forecasting, 2nd Edition, Springer, 2002.

[5] Y. Cui, C. Surpur, S. Ahmad, J. Hawkins, A comparative study of htm and other neural network models for online sequence learning with streaming data, in: International Joint Conference on Neural Networks (IJCNN), 2016, pp. 1530–1538. doi:10.1109/IJCNN.2016.7727380.

[6] Y. Cui, S. Ahmad, J. Hawkins, Continuous online sequence learning with an unsupervised neural network model, Neural Computation 28 (11) (2016) 2474–2504. doi:10.1162/neco_a_00893.

[7] J. D. Hamilton, Time series analysis, Vol. 2, Princeton University Press, 1994.

[8] H. Yang, Z. Pan, Q. Tao, J. Qiu, Online learning for vector autoregressive moving-average time series prediction, Neurocomputing doi:10.1016/j.neucom.2018.04.011.

[9] R. E. Kalman, A new approach to linear filtering and prediction problems, Journal of Basic Engineering 82 (1) (1960) 35–45. doi:10.1115/1.3662552.

[10] S. Särkkä, A. Vehtari, J. Lampinen, Cats benchmark time series prediction by kalman smoother with cross-validated noise density, Neurocomputing 70 (13) (2007) 2331 – 2341. doi:10.1016/j.neucom.2005.12.132.

[11] R. Faragher, Understanding the basis of the kalman filter via a simple and intuitive derivation [lecture notes], IEEE Signal Processing Magazine 29 (5) (2012) 128–132. doi:10.1109/MSP.2012.2203621.

[12] K. R. Müller, A. J. Smola, G. Rätsch, B. Schölkopf, J. Kohlmorgen, V. Vapnik, Predicting time series with support vector machines, in: Artificial Neural Networks — ICANN'97, Springer Berlin Heidelberg, 1997, pp. 999–1004. doi:10.1007/BFb0020283.

[13] K. Kim, Financial time series forecasting using support vector machines, Neurocomputing 55 (1) (2003) 307 – 319. doi:10.1016/S0925-2312(03)00372-2.

[14] Y. Bao, T. Xiong, Z. Hu, Multi-step-ahead time series prediction using multiple-output support vector regression, Neurocomputing 129 (2014) 482 – 493. doi:10.1016/j.neucom.2013.09.010.

[15] G. Zhang, B. Patuwo, M. Y. Hu, A simulation study of artificial neural networks for nonlinear time-series forecasting, Computers & Operations Research 28 (4) (2001) 381 – 396. doi:10.1016/S0305-0548(99)00123-9.

[16] T. Kuremoto, S. Kimura, K. Kobayashi, M. Obayashi, Time series forecasting using a deep belief network with restricted boltzmann machines, Neurocomputing 137 (2014) 47 – 56. doi:10.1016/j.neucom.2013.03.047.

[17] F. Shen, J. Chao, J. Zhao, Forecasting exchange rate using deep belief networks and conjugate gradient method, Neurocomputing 167 (2015) 243 – 253. doi:10.1016/j.neucom.2015.04.071.

[18] C. Sheng, J. Zhao, Y. Liu, W. Wang, Prediction for noisy nonlinear time series by echo state network based on dual estimation, Neurocomputing 82 (2012) 186 – 195. doi:10.1016/j.neucom.2011.11.021.

[19] C. Yang, J. Qiao, H. Han, L. Wang, Design of polynomial echo state networks for time series prediction, Neurocomputing 290 (2018) 148 – 160. doi:10.1016/j.neucom.2018.02.036.

[20] S. Zhong, X. Xie, L. Lin, F. Wang, Genetic algorithm optimized double-reservoir echo state network for multi-regime time series prediction, Neurocomputing 238 (2017) 191 – 204. doi:10.1016/j.neucom.2017.01.053.

[21] Y. Lan, Y. C. Soh, G.-B. Huang, Ensemble of online sequential extreme learning machine, Neurocomputing 72 (13) (2009) 3391 – 3395. doi:10.1016/j.neucom.2009.02.013.

[22] J. Xue, S. Zhou, Q. Liu, X. Liu, J. Yin, Financial time series prediction using $\ell_{2,1}$rf-elm, Neurocomputing 277 (2018) 176 – 186. doi:10.1016/j.neucom.2017.04.076.

[23] X. Wang, M. Han, Online sequential extreme learning machine with kernels for nonstationary time series prediction, Neurocomputing 145 (2014) 90 – 97. doi:10.1016/j.neucom.2014.05.068.

[24] P. Singh, Rainfall and financial forecasting using fuzzy time series and neural networks based model, International Journal of Machine Learning and Cybernetics 9 (3) (2018) 491–506. doi:10.1007/s13042-016-0548-5.

[25] O. C. Yolcu, H.-K. Lam, A combined robust fuzzy time series method for prediction of time series, Neurocomputing 247 (2017) 87 – 101. doi:10.1016/j.neucom.2017.03.037.

[26] F. Gaxiola, P. Melin, F. Valdez, O. Castillo, Interval type-2 fuzzy weight adjustment for backpropagation neural networks with application in time series prediction, Information Sciences 260 (2014) 1 – 14. doi:10.1016/j.ins.2013.11.006.

[27] F. Gaxiola, P. Melin, F. Valdez, O. Castillo, Generalized type-2 fuzzy weight adjustment for backpropagation neural networks in time series

prediction, Information Sciences 325 (2015) 159 – 174. doi:10.1016/j.ins.2015.07.020.

[28] M. Binkowski, G. Marti, P. Donnat, Autoregressive convolutional neural networks for asynchronous time series, in: Proceedings of the 35th International Conference on Machine Learning, Vol. 80, PMLR, 2018, pp. 579–588.

[29] Z. Che, S. Purushotham, G. Li, B. Jiang, Y. Liu, Hierarchical deep generative models for multi-rate multivariate time series, in: Proceedings of the 35th International Conference on Machine Learning, Vol. 80, PMLR, 2018, pp. 783–792.

[30] Z. Zhao, W. Chen, X. Wu, P. C. Y. Chen, J. Liu, Lstm network: a deep learning approach for short-term traffic forecast, IET Intelligent Transport Systems 11 (2) (2017) 68–75. doi:10.1049/iet-its.2016.0208.

[31] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, S. Savarese, Social lstm: Human trajectory prediction in crowded spaces, in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 961–971. doi:10.1109/CVPR.2016.110.

[32] B. Liu, C. Fu, A. Bielefield, Y. Q. Liu, Forecasting of chinese primary energy consumption in 2021 with gru artificial neural network, Energies 10 (10). doi:10.3390/en10101453.

[33] L. Kuan, Z. Yan, W. Xin, C. Yan, P. Xiangkun, S. Wenxue, J. Zhe, Z. Yong, X. Nan, Z. Xin, Short-term electricity load forecasting method based on multilayered self-normalizing gru network, in: IEEE Conference on Energy Internet and Energy System Integration (EI2), 2017, pp. 1–5. doi:10.1109/EI2.2017.8245330.

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

[35] I. V. Tetko, D. J. Livingstone, A. I. Luik, Neural network studies. 1. comparison of overfitting and overtraining, Journal of Chemical Information and Computer Sciences 35 (5) (1995) 826–833. doi:10.1021/ci00027a006.

[36] S. Lawrence, C. L. Giles, Overfitting and neural networks: conjugate gradient and backpropagation, in: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, Vol. 1, 2000, pp. 114–119. doi:10.1109/IJCNN.2000.857823.

[37] A. J. Conejo, J. Contreras, R. Espínola, M. A. Plazas, Forecasting electricity prices for a day-ahead pool-based electric energy market, International Journal of Forecasting 21 (3) (2005) 435 – 462. doi:10.1016/j.ijforecast.2004.12.005.

[38] G. Tkacz, Neural network forecasting of canadian gdp growth, International Journal of Forecasting 17 (1) (2001) 57 – 69. doi:10.1016/S0169-2070(00)00063-7.

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

[40] V. Pham, T. Bluche, C. Kermorvant, J. Louradour, Dropout improves recurrent neural networks for handwriting recognition, in: 14th International Conference on Frontiers in Handwriting Recognition, 2014, pp. 285–290. doi:10.1109/ICFHR.2014.55.

[41] J. Hawkins, S. Ahmad, Why neurons have thousands of synapses, a theory of sequence memory in neocortex, Frontiers in Neural Circuits 10 (2016) 23. doi:10.3389/fncir.2016.00023.

[42] M. A. Rodriguez, R. Kotagiri, R. Buyya, Detecting performance anomalies in scientific workflows using hierarchical temporal memory, Future Generation Computer Systemsdoi:10.1016/j.future.2018.05.014.

[43] J. Wu, W. Zeng, F. Yan, Hierarchical temporal memory method for time-series-based anomaly detection, Neurocomputing 273 (2018) 535 – 546. doi:10.1016/j.neucom.2017.08.026.

[44] Z. Hasani, Robust anomaly detection algorithms for real-time big data: Comparison of algorithms, in: 6th Mediterranean Conference on Embedded Computing (MECO), 2017, pp. 1–6. doi:10.1109/MECO.2017.7977130.

[45] O. Krestinskaya, T. Ibrayev, A. P. James, Hierarchical temporal memory features with memristor logic circuits for pattern recognition, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (6) (2018) 1143–1156. doi:10.1109/TCAD.2017.2748024.

[46] M. Gardner, S. Dorling, Artificial neural networks (the multilayer perceptron) —a review of applications in the atmospheric sciences, Atmospheric Environment 32 (14) (1998) 2627 – 2636. doi:10.1016/S1352-2310(97)00447-0.

[47] A. Graves, Generating sequences with recurrent neural networks, CoRR abs/1308.0850. arXiv:1308.0850.

[48] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using rnn encoder–decoder for statistical machine translation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, 2014, pp. 1724–1734. doi:10.3115/v1/D14-1179.

[49] J. Hawkins, S. Ahmad, S. Purdy, A. Lavin, Biological and machine intelligence (bami), initial online release 0.4 (2016).
URL https://numenta.com/resources/biological-and-machine-intelligence/

[50] I. Aizenberg, L. Sheremetov, L. Villa-Vargas, J. Martinez-Muñoz, Multilayer neural network with multi-valued neurons in time series forecasting of oil production, Neurocomputing 175 (2016) 980 – 989. doi:10.1016/j.neucom.2015.06.092.

[51] S. Galeshchuk, Neural networks performance in exchange rate prediction, Neurocomputing 172 (2016) 446 – 452. doi:10.1016/j.neucom.2015.03.100.

[52] A. Graves, A. r. Mohamed, G. Hinton, Speech recognition with deep recurrent neural networks, in: IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 6645–6649. doi:10.1109/ICASSP.2013.6638947.

[53] R. Pascanu, T. Mikolov, Y. Bengio, Understanding the exploding gradient problem, CoRR abs/1211.5063.

[54] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8) (1997) 1735–1780. doi:10.1162/neco.1997.9.8.1735.

[55] R. Jozefowicz, W. Zaremba, I. Sutskever, An empirical exploration of recurrent network architectures, in: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, 2015, pp. 2342–2350.

[56] Y. Bengio, N. Boulanger-Lewandowski, R. Pascanu, Advances in optimizing recurrent networks, in: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 8624–8628. doi:10.1109/ICASSP.2013.6639349.

[57] F. A. Gers, J. A. Schmidhuber, F. A. Cummins, Learning to forget: Continual prediction with lstm, Neural Comput. 12 (10) (2000) 2451–2471. doi:10.1162/089976600300015015.

[58] J. Chung, Ç. Gülçehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, CoRR abs/1412.3555. arXiv:1412.3555.

[59] J. Hawkins, S. Blakeslee, On intelligence: How a new understanding of the brain will lead to the creation of truly intelligent machines, Macmillan, 2007.

[60] M. S. Gazzaniga, Human: The science behind what makes us unique, Ecco, 2008.

[61] S. Ahmad, J. Hawkins, How do neurons operate on sparse distributed representations? a mathematical theory of sparsity, neurons and active dendrites, arXiv preprint arXiv:1601.00720.

[62] S. Purdy, Encoding data for HTM systems, CoRR abs/1602.05925. arXiv:1602.05925.

[63] F. D. S. Webber, Semantic folding theory and its application in semantic fingerprinting, CoRR abs/1511.08855. arXiv:1511.08855.

[64] Y. Cui, S. Ahmad, J. Hawkins, The htm spatial pooler—a neocortical algorithm for online sparse distributed coding, Frontiers in Computational Neuroscience 11 (2017) 111. doi:10.3389/fncom.2017.00111.

[65] D. O. Hebb, The organization of behavior: A neuropsychological theory, Wiley, 1949.

[66] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, CoRR arXiv:1412.6980.

[67] R. J. Williams, J. Peng, An efficient gradient-based algorithm for online training of recurrent network trajectories, Neural Computation 2 (4) (1990) 490–501. doi:10.1162/neco.1990.2.4.490.

[68] I. Sutskever, Training recurrent neural networks, University of Toronto, Toronto, Ont., Canada.

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

[70] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, J. Schmidhuber, Lstm: A search space odyssey, IEEE Transactions on Neural Networks and Learning Systems 28 (10) (2017) 2222–2232. doi:10.1109/TNNLS.2016.2582924.

[71] S. Makridakis, Accuracy measures: theoretical and practical concerns, International Journal of Forecasting 9 (4) (1993) 527 – 529. doi:10.1016/0169-2070(93)90079-3.

[72] R. J. Hyndman, A. B. Koehler, Another look at measures of forecast accuracy, International Journal of Forecasting 22 (4) (2006) 679 – 688. doi:10.1016/j.ijforecast.2006.03.001.

[73] I. London, Encoding cyclical continuous features - 24-hour time, http://ianlondon.github.io/blog/encoding-cyclical-features-24hour-time/, [Acccessed: 31 March 2018] (2016).

[74] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks, in: Advances in Neural Information Processing Systems 29, Curran Associates, Inc., 2016, pp. 4107–4115.

[75] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Quantized neural networks: Training neural networks with low precision weights and activations, CoRR arXiv:1609.07061.

[76] E. Ogasawara, L. C. Martinez, D. de Oliveira, G. Zimbrão, G. L. Pappa, M. Mattoso, Adaptive normalization: A novel data normalization approach for non-stationary time series, in: The 2010 International Joint Conference on Neural Networks (IJCNN), 2010, pp. 1–8. doi:10.1109/IJCNN.2010.5596746.