Applying Deep Neural Networks to Financial Time Series Forecasting

Allison Koenecke

Abstract For any financial organization, forecasting economic and financial variables is a critical operation. As the granularity at which forecasts are needed increases, traditional statistical time series models may not scale well; on the other hand, it is easy to incorrectly overfit machine learning models. In this chapter, we will describe the basics of traditional time series analyses, discuss how neural networks work, show how to implement time series forecasting using neural networks, and finally present an example with real data from Microsoft. In particular, Microsoft successfully approached revenue forecasting using deep neural networks conjointly with curriculum learning, which achieved a higher level of precision than is possible with traditional techniques.

1 Introduction to Time Series Analysis

Time series are simply series of data points ordered by time. We first discuss the most commonly-used traditional (non-neural network) models, and then comment on pitfalls to avoid when formulating these models.

1.1 Common Methods for Modeling

1.1.1 Stationary Time Series

Time series analyses can be classified as parametric or non-parametric, where the former assumes a certain underlying stationary stochastic process, whereas the latter does not and implicitly estimates covariance of the process. We mostly focus on

Institute for Computational & Mathematical Engineering, Stanford, California, USA, e-mail: koenecke@stanford.edu

parametric modeling within this chapter. Note that there are trade-offs in dealing with parametric modeling. Specifically, significant data cleaning is often necessary to transform non-stationary time series into stationary data that can be used with parametric models; tuning parameters is also often a difficult and costly process. Many other machine learning methods exist, such as running a basic linear regression or random forest using time series features (e.g., lags of the given data, times of day, etc.).

Stationary time series have constant mean and variance (that is, statistical properties do not change over time). The Dickey-Fuller test [1] is used to test for stationarity; specifically, it tests the null hypothesis of a unit root being present in an autoregressive model.

Autoregressive models are regressions on the time series itself, lagged by a certain number of timesteps. An AR(1) model, with a time lag of one, is defined as $y_t = \rho y_{t-1} + u_t$, for y_t being the time series with time index t, ρ being the coefficient, and u_t being the error term.

If $\rho = 1$ in the autoregressive model above, then a unit root is present, in which case the data are non-stationary. Suppose we find that this is the case; how can we then convert the time series to be stationary in order to use a parametric model? There are several tricks to doing this.

If high autocorrelation is found, one can instead perform analyses on the first difference of the data, which is $y_t - y_{t-1}$. If it appears that the autocorrelation is seasonal (i.e., there are periodic fluctuations over days, months, quarters, years, etc.), one can perform de-seasonalization. This is often done with the STL method (Seasonal and Trend Decomposition using Loess) [2], which decomposes a time series into its seasonal, trend, and residual parts; these three parts can either additively or multiplicatively form the original time series. The STL method is especially robust because the seasonal component can change over time, and outliers will not affect the seasonal or trend components (as they will mostly affect the residual component). Once the time series is decomposed into the three parts, one can remove the seasonality component, run the model, and post hoc re-incorporate seasonality. While we focus on STL throughout this chapter, it is worth mentioning that other common decomposition methods in economics and finance include TRAMO-SEATS [3], X11 [4], and Hodrick-Prescott [5].

Recall that stationarity also requires constant variance over time; the lack thereof is referred to as heteroskedasticity. To resolve this issue, a common suggestion is to instead study $log(y_t)$, which has lower variance.

1.1.2 Common Models

Moving average calculation simply takes an average of values from time t_{i-k} through t_i, for each time index $i > k$. This is done to smooth the original time series and make trends more apparent. A larger k results in a smoother trend.

Exponential smoothing non-parametrically assigns exponentially decreasing weights for historical observations. In this way, new data have higher weights for forecasting. Exponential smoothing (ETS) is explicitly defined as

$$s_t = \alpha x_t + (1-\alpha)s_{t-1}, t > 0 \tag{1}$$

where the smoothing algorithm output is s_t, which is initialized to $s_0 = x_0$, and x_t is the original time series sequence. The smoothing parameter $0 \leq \alpha \leq 1$ allows one to set how quickly weights decrease over historical observations. In fact, we can perform exponential smoothing recursively; if twice, then we introduce an additional parameter β as the trend smoothing factor in "double exponential smoothing." If thrice (as in "triple exponential smoothing"), we introduce a third parameter γ as the seasonal smoothing factor for a specified season length. One downside, though, is that historical data are forgotten by the model relatively quickly.

ARIMA is the Autoregressive Integrated Moving Average [6], one of the most common parametric models. We have already defined autoregression and moving average, which respectively take parameters $AR(p)$ where p is the maximum lag, and $MA(q)$ where q is the error lag. If we combine just these two concepts, we get ARMA, where α are parameters of the autoregression, θ are parameters of the moving average, and ε are error terms assumed to be independent and identically distributed (from a normal distribution centered at zero). ARMA is as follows:

$$Y_t - \alpha_1 Y_{t-1} - ... - \alpha_p Y_{t-p} = \varepsilon_t + \theta_1 \varepsilon_{t-1} + ... + \theta_q \varepsilon_{t-q} \tag{2}$$

ARMA can equivalently be written as follows, with lag operator L:

$$(1 - \sum_{i=1}^{p} \alpha_i L^i) Y_t = (1 + \sum_{i=1}^{q} \theta_i L^i) \varepsilon_t \tag{3}$$

From this, we can proceed to ARIMA, which includes integration. Specifically, it regards the order of integration: that is, the number of differences to be taken for a series to be rendered stationary. Integration $I(d)$ takes parameter d, the unit root multiplicity. Hence, ARIMA is formulated as follows:

$$(1 - \sum_{i=1}^{p-d} \alpha_i L^i)(1-L)^d Y_t = (1 + \sum_{i=1}^{q} \theta_i L^i) \varepsilon_t \tag{4}$$

1.1.3 Forecasting Evaluation Metrics

We often perform the above modeling so that we can forecast time series into the future. But, how do we compare whether a model is better or worse at forecasting? We first designate a set of training data, and then evaluate predictions on a separately held-out set of test data. Then, we compare our predictions to the test data and calculate an evaluation metric.

Hold-out sets It is important to perform cross-validation properly such that the model only sees training data up to a certain point in time, and does not get to "cheat" by looking into the future; this mistake is referred to as "data leakage." Because we may only have a single time series to work with, and our predictions are temporal, we cannot simply arbitrarily split some share of the data into a training set, and another share into validation and test sets. Instead, we use a walk-forward split [7] wherein validation sets are a certain number of timesteps forward in time from the training sets. If we train a model on the time series from time t_0 to t_i, then we make predictions for times t_{i+1} through t_{i+k} (where $k = 1$ in Figure 1, and prediction on t_{i+4} only is shown in Figure 2) for some chosen k. We calculate an error on the predictions, and then enlarge the training set and iteratively calculate another error, and so on, continuously walking forward in time until the end of the available data. In general, this idea is called the "rolling window process," and allows us to do multiple-fold validation. The question that remains now is how to calculate prediction error.

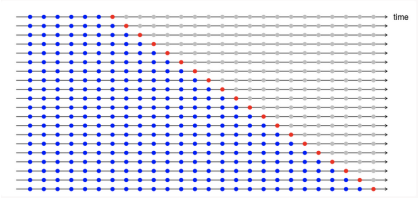

Fig. 1 Cross validation with 1-step-ahead forecasts; training data are in blue, and test data are in red. Specifically, forecasts are only made one timestep further than the end of the training data [8]

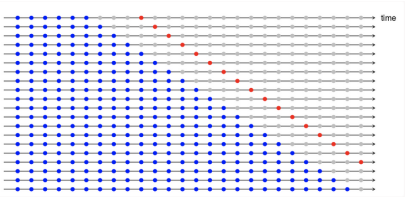

Fig. 2 Cross validation with 4-step-ahead forecasts. Forecasting can be done for any specific timestep ahead, or range of timesteps, so long as the training and test split are preserved as such [8]

Evaluation metrics We can use several different evaluation metrics to calculate error on each of the validation folds formed as described above. Let y_i be the actual test value, and $\hat{y}_i$ be the prediction for the i^{th} fold over n folds. The below common metrics can be aggregated across folds to yield one final metric:

1. Mean Absolute Error: $\sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{n}$
2. Mean Absolute Percentage Error: $\frac{1}{n}\sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|$
3. Mean Squared Error: $\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
4. R squared: $1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \frac{1}{n}\sum_{i=1}^{n} y_i)^2}$

1.2 Common Pitfalls

While there are many ways for time series analyses to go wrong, there are four common pitfalls that should be considered: using parametric models on non-stationary data, data leakage, overfitting, and lack of data overall. These pitfalls extend to the data cleaning steps that will be used with neural networks, which are described in Section 2. Below we recap what can be done to ameliorate these issues.

Non-stationary data Ensure that data is transformed prior to modeling. We suggest several methods in Section 1.1, such as order differences, seasonality removal, and logarithmic transformations. Further to this, trends can also be removed (e.g., by subtracting the overall mean of a time series), smoothing can be done by replacing the time series with a moving average, and other transformations may be useful as well (e.g., standardization or Box-Cox). In general, it is also good practice to clean data by removing outliers.

Data leakage Confirm that cross-validation is done correctly, so that the model does not see any future data relative to the timestep it is predicting. In particular, if de-meaning the data to make it stationary, ensure that means are calculated only over each training data batch, rather than across the entire duration of the time series.

Overfitting Certain training data points may predict overly well, so the model may overfit to those rather than generalizing, resulting in lower accuracy in the test set. For regression-based analyses, regularizers (such as Lasso and Ridge) or feature selection can help. When running ARIMA, check the Akaike information criterion (AIC) to determine which parameters are optimal to avoid overfitting.

Lack of data This goes hand in hand with overfitting, as too few data for the number of parameters poses a similar concern to having too many parameters for the number of observations. On the data side, one can use rolling windows to allow for more training data, since shorter lengths of time series are incorporated. Keep in mind that if the time series is too short, STL often cannot be trained. If multiple time series exist, a subset of which have some historical data missing that cannot be imputed, transfer learning can help. If the time series express roughly similar temporal trajectories, try only training on the time series with full historical data.

2 Introduction to Deep Learning

Deep learning has become popularized in many aspects of data science. Why would we want to use deep learning rather than traditional time series methods? Traditional models struggle with high-dimensional multivariate problems, non-linear relationships, and incomplete datasets. These traits commonly occur in real-world data, and are much more easily resolved by using deep neural networks. That said, a drawback is that neural networks are designed as black boxes which can be difficult for statistical interpretability.

2.1 Deep Learning Basics

Deep Learning Algorithms use neural networks, which associate inputs and outputs using intermediate layers to model non-linear relationships. Each unit in a layer uses a particular representation of the data; for time series data, for example, the input layer may correspond to a vector of numerical values, or a matrix containing auxiliary data. For categorical data, a "one-hot encoding" can be used as input to a neural network; it creates a binary vector with size equal to the number of categories, where all vector values are 0 barring the vector position corresponding to the category, which takes on a value of 1.

A canonical example using neural networks involves predicting the price of a house: Layer 1 (the input layer) contains attributes such as the size, number of bedrooms, zip code, etc. Layer 2 (assuming only one hidden layer) intuits some attributes from the input layer that are not explicitly hard-coded; e.g., size and number of bedrooms indicate a "family size" metric, and zip code indicates "school quality." Finally, in Layer 3 (the output layer), we aggregate the hidden layers and the model returns the price of the home.

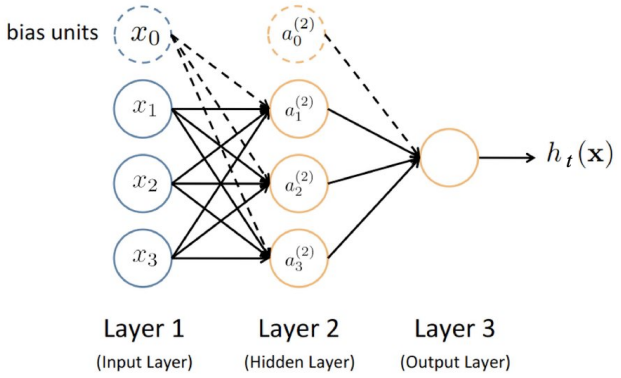

Fig. 3 A basic neural network with three layers [9]

Given a certain neural network architecture, we want to find the correct weights that connect the different neurons (also referred to as nodes, cells, or units), where the neurons are depicted as circles in Figure 3. There are many ways to train a neural network, such as backpropagation, genetic algorithms, or random search; all of these aim to find optimal hyperparameters for the neural network. Note that statistical properties can be well-defined using "traditional" econometric time series methods, and also for shallow networks (having a single hidden layer); however, the statistical aspects become more complicated for deep neural nets.

Backpropagation is a key component in a basic but powerful neural net training method. On a batch of training data, after randomly initializing weight matrices:

1. Calculate loss using feed-forward propagation;
2. Generate loss gradients using backpropagation;
3. Update neural network weights using gradient descent.

We will describe loss in greater detail below; for now, it can be thought of as how incorrect a model's prediction is; a perfect model has a loss of zero, more prediction errors mean a greater loss. Machine learning methods generally aim to minimize the loss function.

For layer number i and unit j (within each layer), the standard notation used for generating weightings in neural networks is as follows. Let z be the output, w be the weight, and b be the bias. Here, we define the raw output function of each unit:

$$z_j^{[i]} = w_j^{[i]^T} x + b_j^{[i]} \tag{5}$$

The forward propagation process goes from left to right of the neural network as drawn in Figure 3. The raw output value is computed for each unit per the above equation, interpreted as the weighted sum of values feeding into the unit. Then, an activation function is usually applied to non-linearly transform the raw output value (in all layers except the input layer); this can be thought of as thresholding the unit, such that the transformation of $z_j^{[i]}$ values are "activated" at a certain level. Common activation functions include the sigmoid ($g(z) = \frac{1}{1+e^{-z}}$), tanh ($g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$), and ReLU ($g(z) = max(0,z)$). Let us define activation $a_j^{[i]} = g\left(z_j^{[i]}\right)$.

After one full pass through forward propagation, we have our activation a which functions as our predicted output $\hat{y}$, as compared to our final output $h_t(x)$ at time iteration t, which functions as our general output y. From this, we calculate a single scalar value: loss. There are many different loss functions to choose from; for regression problems, mean squared loss is commonly used. The Mean Squared Error (MSE) loss function is defined on N samples as $L(y,\hat{y}) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2$. We aim to minimize this loss using gradient descent.

Next, in the backpropagation step, gradients are calculated backwards, on layers from right to left per Figure 3. For intuition, we work backwards so that we can take our previous loss calculation, and determine which mis-steps throughout the neural network led to the lower accuracy that we found in the loss. Each derivative with respect to weight w is calculated using the chain rule as follows:

$$\frac{\partial L(w)}{\partial w} = \frac{\partial L(w)}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w} \tag{6}$$

Finally, the neural network weights w are updated using gradient descent, defined for learning rate α by:

$$w \leftarrow w - \alpha \frac{\partial L(w)}{\partial w} \tag{7}$$

We repeat and update weights until convergence. There are myriad ways to improve from this basic outline. For example, the dropout technique can be used, which ameliorates overfitting data by dropping units with a pre-defined probability. The

learning rate α can be fixed or iteratively changed by methods such as Adam, Adagrad, momentum, etc. Instead of random initialization of weights, Xavier [10] or He [11] initialization can be used. Additionally, L2 regularization can be used in the gradient descent update step for w [12].

Various types of neural network architectures exist for different tasks. We discuss two major categories: Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

Convolutional Neural Networks are primarily used in computer vision, where inputs are often matrices with entries corresponding to pixel values. They are commonly used due to relatively easy feature engineering, which do not need to learn directly from, e.g., lagged time series observations. CNNs are good at learning representations of large inputs. As shown in Figure 4, CNNs have four types of layers, from left to right: the input layer, convolutions of the input (CONV), pooling of the convolutions (POOL), and finally a fully connected (FC) layer [13].

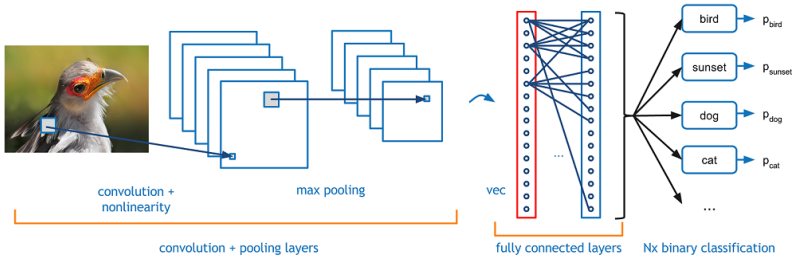

Fig. 4 An example of a CNN to perform classification (among Nx classes) on a 2-D image [13]

The CONV layer performs the convolution operator as the input is taken in, and outputs a "feature map." For input volume W, filter size F (yielding a $F \times F$ square receptive field within which the convolution is taken), zero padding P (number of zeroes added surrounding input boundaries so that edge data are seen by filters), and stride S (number of pixels that the filter moves after each operation), the convolutional layer must have N units, as specified below:

$$N = \frac{W - F + 2P}{S} + 1 \tag{8}$$

The POOL layer downsamples the layer that is fed into it. This can be done with, e.g., max pooling (taking the maximum value over a filter) or average pooling (taking the average value over a filter). Lastly, the FC layer takes as input a flattened layer wherein all units are mutually connected.

Recurrent Neural Networks are primarily used in natural language processing, with the canonical example being translation. They share many positive aspects with CNNs when dealing with sequences, with the additional bonus of applying a recurrence at each time step when processing sequences. This recurrence is shown

in Figure 5. Further, model size does not increase with the size of the input (which can be any length). RNNs allow layers to have hidden states when being input to the next layer, so that historical values can play a role in prediction (though accessing very early data in, e.g., an input sequence is relatively difficult).

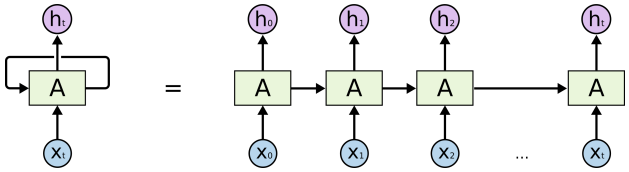

Fig. 5 An unrolled RNN, showing recurrence occurring at each timestep [14]

2.2 Deep Learning for Time Series

Sequence-to-sequence modeling for time series has been fairly popular for the past several years, not just in industry, but also broadly from classrooms [15] to Kaggle [16]. These methods range from vanilla models to advanced industry competitors. Sequence-to-sequence models specifically refer to neural networks where the input is a sequence (e.g., a time series), and the output is similarly a sequence. In order to allow for sequence manipulation, the model needs to keep track of historical dependencies of the sequence, remember time index ordering, and allow for parameter tuning over the time duration of the sequence. We introduce some examples of specific sequence-to-sequence models below.

Dilated Convolutional Neural Networks (DCNN) are a specific kind of CNN that allow for a longer reach back to historical information by connecting an exponential number of input values to the output (using dilated convolutional layers of varied dimensions), as shown in Figure 6 [17]. This allows for better performance on long time series with strong historical dependencies.

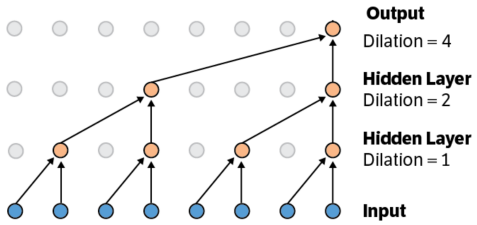

Fig. 6 A three-layer DCNN [17]

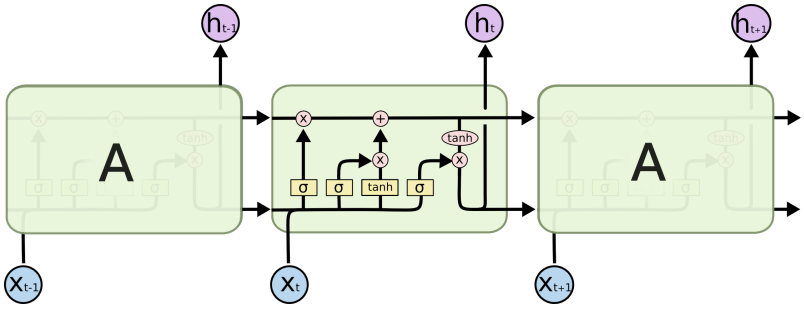

Fig. 7 Three LSTM modules, each with four layers [14]

Long Short-Term Memory (LSTM) is a type of RNN that similarly learns a mapping from input to output over time, but the mapping is no longer fixed as in the standard RNN. In backpropagation, one problem that arises is vanishing and exploding gradients (which result in poor results stemming from divide-by-zero errors). To resolve this, LSTMs use "computational gates" to control information flow over time, allowing them to forget less useful historical information. Hence, LSTMs do better with long-term dependencies.

An LSTM gate takes as input the previous hidden state and current input [14], and returns a new state. These gates are of the form $G_t = g(W_G \cdot [h_{t-1}, x_t] + b_G)$, where activation function g will be either tanh or a sigmoid function σ. The input gate decides which values to update, the forget gate decides which values to forget, and the output gate decides which values to output [14].

First, the LSTM uses the forget gate f_t by passing the previous hidden state and current input through a sigmoid function, which tells the LSTM which information to keep in the cell state C_t. Next, an input gate layer i_t (another sigmoid layer) decides the values to be updated. The new values to be updated in the cell state are found with a tanh layer C_t. Then, we update cell state so that $C_t \leftarrow f_t * C_{t-1} + i_t * C_t$. Finally, we filter outputs using the output gate o_t (a sigmoid layer), and finally return $h_t = o_t * \tanh(C_t)$. These gates are portrayed in the middle module of Figure 7.

Note that LSTMs can be interpreted as a general form of another sub-type of RNN, the Gated Recurrent Unit (GRU). The GRU only has an input gate and a reset gate, the latter of which similarly selectively forgets historical information [14].

For sequence-to-sequence modelings, usually one RNN layer functions as an "encoder" while another functions as a "decoder." The "encoder" processes input and creates an internal state; the internal state is then fed into the "decoder," which predicts the output sequence. In Figure 8, we see that LSTMs can function as encoders and decoders to translate linguistic sequences; character sequences can easily be substituted for numerical values [18].

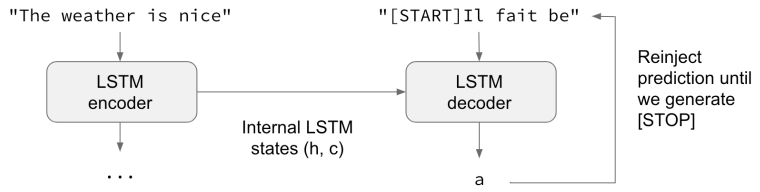

Fig. 8 This LSTM-based sequence-to-sequence architecture will eventually translate "The weather is nice" in English to "Il fait beau" in French; with inference, the process is repeated until a [STOP] character is generated [18]

3 Case Study: Microsoft Revenue Forecasting

We now introduce a real-life case study, excerpted from research conducted at Microsoft [19]. The specific approaches were developed to forecast Microsoft revenue data corresponding to Enterprise, and Small, Medium & Corporate (SMC) products, spanning approximately 60 regions across the globe for 8 different business segments, and totaling tens of billions of USD. In particular, this time series forecasting is done by borrowing deep learning techniques from Natural Language Processing (Encoder-Decoder LSTMs) and Computer Vision (Dilated CNNs). For both techniques, a curriculum learning training regime was introduced, which trains on increasingly difficult-to-forecast trend lines. The resulting forecasts yield a 30% improvement in overall accuracy on test data relative to Microsoft's former baseline forecasting model using traditional techniques.

3.1 Background

This case study is differentiated from prior work in two major ways: firstly, by using curriculum learning (defined below), and secondly, by using both Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) on medium-sized data—that is, data in the thousands of rows with limited historical data—without overfitting.

Curriculum learning is a technique that changes the order of inputs to a model to improve results. The intuition from Natural Language Processing (NLP) regarding this method is that shorter sentences are easier to learn than longer sentences; so, without initialization, one can bootstrap via iterated learning in order of increasing sentence length. The relevant literature, including specifically the described Baby Steps algorithm [20, 21], has been applied to LSTMs for parsing a Wall Street Journal corpus [21], n-gram language modeling [22], and for performing digit sums using LSTMs [23]. However, there has been no application of this work to real numerical time series data.

It is fairly straightforward to use neural networks on large datasets, but more difficult to apply these techniques to smaller-sized data due to the risk of overfitting. Many companies have adopted the use of LSTMs for time series modeling; an advanced public methodology comes from Uber, which won the 2018 M4 Forecasting Competition using a hybrid Exponential Smoothing and RNN model [24]. Another proven neural network method for financial forecasting is the Dilated CNN [17], wherein the underlying architecture comes from DeepMind's WaveNet project [25]. Both instances use similar methods to this case study, but the Microsoft revenue data is an order of magnitude smaller, and has more rigid hierarchical elements.

3.2 Data

3.2.1 Data Structure

World-wide revenues for Enterprise and SMC groups are partitioned into 8 business segments; each segment is partitioned into approximately 60 regions, and each region's revenue is partitioned further into 20 different products. Overall, there are approximately 6,000 datarows (since not all products are sold in all regions), with each datarow corresponding to a time series used for our forecasting problem. Given historical quarterly revenue data, our goal is to forecast quarterly revenue for these products per combination of product, segment, and region; we then generate the aggregated segment-level forecasts as well as world-wide aggregates. Note that we focus on segment-level (rather than subregion or product-level) forecasts for comparison's sake, since this level has historically been used by the business. All revenue numbers are adjusted to be in USD currency using a constant exchange rate. Sample datarow structure is presented in Figure 9.

	Group	SubRegionName	CustomSubsegment	CustomSRSD	2009-01-01 00:00:00	2009-04-01 00:00:00	2009-07-01 00:00:00	2009-10-01 00:00:00	2010-01-01 00:00:00
0	Argentina . Commercial Enterprise . Developer ...	Argentina	Commercial Enterprise	Developer Tools					
1	Argentina . Commercial Enterprise . Dynamics O...	Argentina	Commercial Enterprise	Dynamics OnPrem					

Fig. 9 Sample datarow structure excluding financial values

We use the quarterly revenue available (post-data-processing) for each datarow over fiscal quarters from January 2009 through July 2018 (totaling 39 timesteps). Broadly speaking, we train on the first 35 timesteps of all datarows, and test on the final 4 timesteps. For all DNN models, we train specifically on a subset of the datarows which has good enough history to fit a reasonable model (approximately 84% of all datarows). Post-training, we apply this model to forecast revenue both for

datarows on which it was trained, and also on out-of-sample datarows that were not seen by the model at the time of training due to insufficient historical information.

3.2.2 Microsoft Baseline

Circa 2015, most of the revenue forecasting in Microsoft's Finance division was driven by human judgement. In order to explore more efficient, accurate and unbiased revenue forecasting, machine learning methodology was explored along with statistical time series models [7]. The methodology described in [7]—which roughly ensembles ARIMA, ETS, STL, Random Forest, and ElasticNet models—was used to compute forecasts in 13 worldwide regions. In [26], this approach was further extended to use product level data available within each region, and to generate forecasts for each product within region (allowing for aggregation to region-level and world-wide forecasts). This is the approach that is currently adopted by Microsoft's Finance team; models based on this approach are running in a production environment to generate quarterly revenue forecasts to be used by Finance team members. The results obtained from this method are referred to as the Microsoft baseline, and this case study explores whether the proposed DNN-based models can outperform the current baseline model in production. Because the ensuing revenue predictions are sensitive, we cannot report these numbers directly, but rather report them in terms of improvement over the Microsoft baseline.

3.3 Methods

This work on time series is mostly inspired by non-financial applications. Specifically, Encoder-Decoder LSTMs (Section 3.3.1) are used in NLP, and Dilated CNNs (Section 3.3.2) are applied in Computer Vision and Speech Recognition.

3.3.1 RNN Model: Encoder-Decoder LSTM

We present four variants, each cumulatively building upon the previous variant, of our RNN model to show increasing reduction in error. In all variants, we use a walk-forward split [7] wherein validation sets are four steps forward into time from training sets, ensuring no data leakage. We do this iteratively for windows of size 15 timesteps within the data, continuously walking forward in time until the end of the training data (i.e., until July 2017); this is referred to as the rolling window process as described in Section 1.2. The window size of 15 timesteps was chosen empirically through experiments not described here. As we move from one window to the next, we use weights obtained from the model trained on data corresponding to the previous window for initialization. An example loss function when using the rolling window process is shown in Figure 10; notice that gradual loss is attained

as we step through consecutive windows because the model uses prior weights to warm-start rather than fitting from scratch.

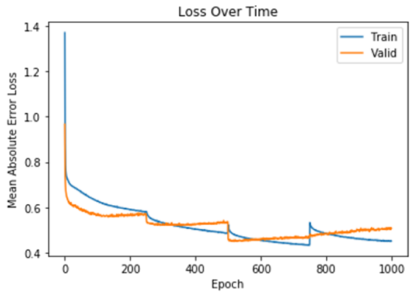

Fig. 10 Example Mean Absolute Error loss for rolling window method on LSTM

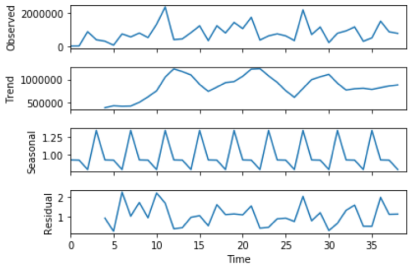

Fig. 11 Seasonal decomposition example on one financial datarow

Basic LSTM The first model we discuss is our basic RNN model. All training, validation, and test data are historical financial values that have been smoothed using a logarithmic transformation and de-meaned on training data only. These preprocessing methods are used throughout due to better experimental results relative to other smoothing transformations. A single-layer sequence-to-sequence model is fed into a dense layer, using the Adam optimizer on mean absolute error. The sequence-to-sequence [27] architecture involves a LSTM encoder (to process revenue and return internal state), and an LSTM decoder (to use the previous time step's actual data and internal LSTM encoder states to generate the next output). Teacher forcing is used only during training; for inference, we feed in the predicted values for the next timestep as input to the decoder instead of the actual value as would be the case in the teacher forcing method. Next, we apply the inverse smoothing transformation on the decoder's output for last four timesteps (i.e., revenue for the last four quarters) to calculate test error.

LSTM with Categorical Indicators The second model we examine is simply the basic model with additional indicator covariates (i.e., one-hot categorical variables are incorporated in the model). Specifically, for our three categorical variables (segment, region, and product), we include one-hot encodings so that the hierarchical product information is reflected in the model.

LSTM with Seasonality The third model incorporates seasonal effects in the second model. Specifically, we use multiplicative Seasonal Trend decomposition using Loess (STL) [2] to calculate trend, seasonal, and residual components. A sample datarow decomposition is shown in Figure 11. We extract the seasonal component from the relevant datarows, and we use only the product of trend and residual effects (in each quarter, and for each datarow) as inputs to be smoothed and fed to the neural network model. De-seasonalizing the input data along with the other transformations (logarithmic and de-meaning) helps to make the data more stationary.

We maintain use of the indicator covariates introduced in the second model. The only difference now is in the inference step: in addition to decoding and using an inverse smoothing transformation, we must also multiply our predictions obtained from the decoder by the seasonal values calculated for each quarter (timestep) in the previous year.

LSTM with Curriculum Learning The fourth model applies curriculum learning to the third model. We use the pre-calculated seasonal decomposition to determine a useful batch ordering method to feed into our neural net, and then apply the Baby Steps curriculum algorithm [23, 21] defined in Figure 12.

1: **procedure** BS-CURRICULUM(M,$\mathcal{D}$, $\mathcal{C}$)
2: $\mathcal{D}'$ = sort($\mathcal{D}$, $\mathcal{C}$)
3: $\{\mathcal{D}^1, \mathcal{D}^2, ..., \mathcal{D}^k\} = \mathcal{D}'$ where $\mathcal{C}(d_a) < \mathcal{C}(d_b)$ $d_a \in D^i$, $d_b \in D^j, \forall i < j$
4: $\mathcal{D}^{train} = \emptyset$
5: **for** s = 1...k **do**
6: $\mathcal{D}^{train} = \mathcal{D}^{train} \cup \mathcal{D}^s$
7: **while** not converged for p epochs **do**
8: train(M, $\mathcal{D}^{train}$)
9: **end while**
10: **end for**
11: **end procedure**

Fig. 12 Baby Steps Curriculum Learning Algorithm [23, 21]

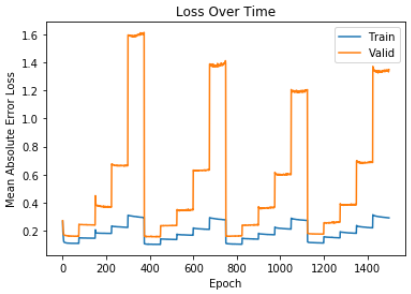

Fig. 13 Example Mean Absolute Error loss for curriculum learning with rolling window method on LSTM (ordered from easiest to hardest estimated prediction)

Let us define $D' = sort(D,C)$ for training data D and curriculum sort metric C [23], where our sort metric is specified as the residual trend weighted by segment revenue for each datarow. That is, we will sort our training data on this new variable we have created, which exists for each datarow. Then, we order our batches such that $\{D^1, D^2, ..., D^k\} = D'$ where $C(d_a) < C(d_b)$ for $d_a \in D^i, d_b \in D^j, \forall i < j$. In words, we train in increasing order of the residual error calculated from the STL decomposition mentioned previously. Once the k batches are defined, we shuffle the datarows within the batch during training. Within each iteration of the rolling window process, we continue the warm-start process by iteratively adding one batch at a time to the training data, running each rolling window iteration p times, where p is the number of epochs chosen such that convergence can be reached. In summary, for each of the $s = 1, ..., k$ batches, we run $D^{train} = D^{train} \cup D^s$ until the end of p epochs of each rolling window iteration. A sample loss function including curriculum learning is shown in Figure 13, where we have experimentally chosen $p = 75$.

We note that there are several ways to form the batching described above. In our results, we present batches formed directly from the datarow sort metric calculated as described above, i.e., the datarow-level residual error (using 5 batches, as determined experimentally). However, we have also found good results when batching by segment (with each batch corresponding to one Microsoft segment), where batches

are sorted by revenue-weighted segment-level residual error. In all cases, we shuffle datarows within each batch when training. The idea of curriculum learning is to train on easier examples to learn harder ones. As such, we find that running curriculum learning with batches sorted in order of best-fit to least-fit segments yields similar results to those that we find from the datarow-level (uniform-weighting) method we use in this paper. However, we also experimented with curriculum learning using batches sorted in reverse order: least-fit to best-fit segments. This resulted in far better accuracies for certain (smaller-revenue) segments, but worse accuracies for all other segments. Hence, it remains a reasonable option to sort in various ways, and then ensemble results, to tease out best results for different segments.

3.3.2 CNN Model: Dilated CNN

We present three versions of our CNN model to show increased reduction in error across iterations. We note that we do not explore separating seasonal effects from our DCNN model as a pre-processing step. While found to be useful in the LSTM model, results are not significantly better for the DCNN when performing seasonal decomposition prior to training the model. This is likely because the exponential nature of the DCNN layers allows us to capture the seasonality over a long time series, since we are using each relevant datarow's full 35-quarter financial history as input to the model (rather than the rolling window method applied for LSTMs).

Basic Dilated CNN In our first Dilated CNN model, we use 1D convolutions in each of 10 dilated convolutional layers (with 6 filters of width 2 per layer). This connects to an exponential number (2^{10}) of input values for the output. Two fully connected layers are used to obtain a final output: a dense layer of size 128 with ReLU activation, and a dense layer of size 1. We apply an Adam optimizer on the Mean Absolute Error. Teacher forcing is done during training only, and predicts the four test quarters of data iteratively, appending each prediction to history for the next timestep's prediction. Similar to the LSTM model, all historical financial values passed into the DCNN model have been smoothed using a logarithmic transformation and de-meaned on training data only.

Dilated CNN with Categorical Indicators The second DCNN model we examine is simply the above basic model with additional indicator covariates. Specifically, for our three categorical variables (segment, region, and product), we include one-hot encodings so that the hierarchical product information is reflected in the model.

Dilated CNN with Curriculum Learning We apply the same mechanism for curriculum learning as explained for the LSTM model, using the residual from seasonal decompositions as a proxy for the difficulty of forecasting each datarow. We batch by the datarow-level residual error (using 8 batches, as determined experimentally). The curriculum learning is performed based on the second DCNN model, i.e., including categorical variables, but not using seasonal decomposition for anything aside from the sort order for curriculum learning.

3.3.3 Evaluation

For evaluation purposes, we use the four quarters of data from October 2017 to July 2018 as our test dataset. For certain products, there are only null values available in recent quarters (e.g., if the product is being discontinued) and hence we do not include these products in the test dataset. We use the Mean Absolute Percentage Error (MAPE) as our error evaluation metric. To take into account the inherent randomness involved from weight initialization when training the DNNs, and considering that our data is medium-sized, we run each experiment 30 times to obtain a more robust estimate of the errors. For each datarow, we take the average of the forecasts across runs and across quarters as the final forecast and compare this predicted revenue to the actual observed revenue. The segment-level forecast is the sum of all (subregion-level and product-level) forecasts falling into that segment. The world-wide forecast is the sum of forecasts for all datarows. Due to privacy concerns, actual test errors are not displayed. We instead report relative percentage improvement over the Microsoft baseline in production.

3.4 Results

We find that both LSTM and DCNN models with curriculum learning out-perform the respective models without curricum learning. In particular, the Encoder-Decoder LSTM with curriculum learning (including categorical indicators and seasonality effects) yields the lowest error rates, showing a world-wide improvement of 27%, and a revenue-weighted segment-based improvement of 30%, over Microsoft's production baseline. We further find that curriculum learning models can yield either lower bias or variance for various segments.

3.4.1 World-wide Error Rates

World-wide MAPEs for all models are compared in Table 1. Both LSTM and DCNN models with curriculum learning outperform all variants without curriculum learning by over 10 percentage points. It is worth noting that even the baseline LSTM model (without curriculum learning) improves upon the Microsoft baseline in production. It is worth commenting on the decrease in world-wide accuracy upon adding seasonality to the LSTM model. While the world-wide error (MAPE) is higher for this model variant, we see in Table 2 that the revenue-weighted segment-level average yields an improvement of 21% from seasonality over the previous LSTM model variants. The interpretation here is that seasonality can be more accurately inferred for the few product segments having the largest revenues, and hence the segment-level benefits are outweighed world-wide by the many smaller-revenue datarows that are less accurate (due to more fluctuation in seasonal effects on smaller products). We suggest that seasonal trend decomposition be used only after careful

consideration of the durability of financial seasonality. In our application, we only present LSTM results including seasonality since we find it beneficial conjointly with curriculum learning; experimentally, our displayed results fare better than the alternative of curriculum learning sans seasonality.

Table 1 World-wide test error reduction percentages of DNN models over previous Microsoft production baseline.

Model	**Percent MAPE Improvement**
Basic LSTM	1.9%
LSTM with Categorical Indicators	18.2%
LSTM with Seasonality	-5.1%
LSTM with Curriculum Learning	**27.0%**
Basic DCNN	-0.7%
DCNN with Categorical Indicators	12.1%
DCNN with Curriculum Learning	**22.6%**

Recall that we ran each model 30 times and took outcome averages to reduce variance. Density plots are shown for world-wide results in Figures 14 and 15, which reflect the distribution of calculated (non-absolute) percentage error for each of the LSTM and DCNN models tested, respectively. These figures allow us to examine the extent to which curriculum learning models are less biased: we see that the curriculum learning variant (red curve for LSTM, and green curve for DCNN, respectively) is both closest to zero-centered and is one of the models with lowest variance. [1]

Table 2 LSTM Model Segment-level MAPE reduction percentages (%) over previous Microsoft production baseline (positive % corresponds to error reduction).

Segment	**Basic LSTM** ***(Model (a))***	**Model (a) + Categorical Indicators** ***(Model (b))***	**Model(b) + Seasonality** ***(Model (c))***	**Model(c) + Curriculum Learning** ***(Model(d))***
1	25.5	22.0	53.4	70.0
2	-47.9	-34.3	-23.0	-0.8
3	7.65	-5.8	26.0	20.3
4	14.2	30.3	12.0	27.4
5	-15.4	-13.2	-11.8	-25.9
6	-79.2	-60.3	-110.1	-12.4
7	34.7	30.1	31.0	11.5
8	17.9	15.5	57.2	61.4
Revenue-weighted Average	**10.3**	**10.3**	**21.3**	**30.0**

[1] For both density plots, the y-axis denoting density is fully presented. However, note that the x-axis (expressing percent error) values aside from 0 are excluded for Microsoft privacy reasons. For both LSTM and DCNN models, we can disclose that the spread of error is bounded by a range of approximately ± 10 percentage points.

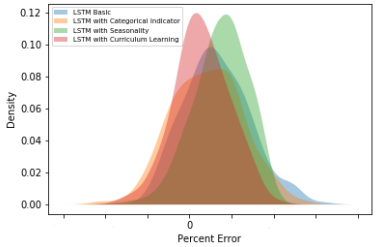

Fig. 14 LSTM World-wide Error Density

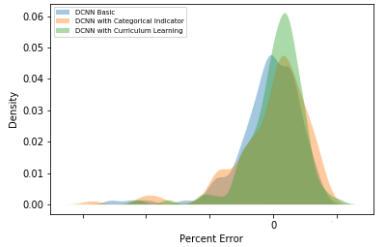

Fig. 15 DCNN World-wide Error Density

3.4.2 Segment-Level MAPEs

In Tables 2 and 3, we share the segment-level incremental improvement percentages obtained from the Encoder-Decoder LSTM and DCNN model MAPEs, respectively, as compared to Microsoft's previously-implemented baseline. Due to privacy concerns, the actual names of the segments are not shared since their revenues are considered High Business Impact (HBI) data. Overall, the revenue-weighted LSTM segment-level MAPEs show a drastic 30% improvement in MAPE relative to the Microsoft baseline currently used in production (see Table 2); curriculum learning yields a 19% improvement in MAPE using the DCNN model (see Table 3).

While we cannot discuss segment-specific results in terms of absolute numbers, we comment that it is clear that some segments see significant gains relative to the Microsoft production baseline, whereas others only see modest gains (or slight decreases). In particular, the addition of curriculum learning with our uniformly-weighted ordering improves results for larger-revenue segments. However, when using batches based on segment, and using a reverse ordering of segments from hardest to easiest to predict (based on seasonality as we have been doing; results for this variant are not disclosed in this paper), the most improvement is seen in smaller-revenue segments, which otherwise would have been overshadowed by model weights contributing towards larger-revenue segments. Thus, ensembling these two different sorts is a promising future step.

Segment-specific density plots, which can be found in [19], yield similar findings to the world-wide density plots. When looking at both segments with smaller and larger revenue (the latter more than four times the size of the smaller segment), we see that curriculum learning lessens bias and/or across-run variance. Specifically, for the segments wherein curriculum learning improves accuracy, some will see lessened bias (with little change in variance from non-curriculum learning model variants), some will see lessened variance (with little change in bias from non-curriculum learning model variants), and some instances yield better bias and variance. Hence, we conclude that curriculum learning models on our financial time series for specific segments not only yield more accurate forecasts, but also can achieve relatively low variance and bias.

Table 3 DCNN Model Segment-level MAPE reduction percentages (%) over previous Microsoft production baseline (positive % corresponds to error reduction).

Segment	**Basic DCNN** ***(Model (a))***	**Model (a) + Categorical Indicators** ***(Model (b))***	**Model(b) + Curriculum Learning** ***(Model (c))***
1	24.8	44.0	34.2
2	-0.2	-19.5	-19.5
3	-8.7	28.9	39.9
4	35.5	35.4	22.6
5	45.4	58.4	26.8
6	-258.2	-263.2	-80.5
7	27.0	28.7	29.4
8	33.8	35.5	24.9
Revenue-weighted Average	**-3.1**	**4.5**	**16.2**

3.5 Discussion

It is clear from these results that there is value in using DNNs on Microsoft's financial time series, and further that curriculum learning is an indispensable tool to improve accuracy of forecasts—not just within NLP settings. These curriculum learning results are robust both world-wide and at the segment-level. Further, we see from Figures 14 and 15 that curriculum learning allows for less bias in errors (robust across both LSTM and DCNN methods), and in certain instances less variance in error at the segment level. Further, we assert that the financial data used in our models need not be extraordinarily large to successfully use neural networks for forecasting. DNN methods are far more efficient than the Microsoft production baseline that involves ensembling traditional statistical and machine learning methods; it takes a fraction of the time spent to run each DNN model. While curriculum learning involves sorting, and hence may be unwieldy for very large datasets, it does not significantly impact runtime on the "medium-sized" Microsoft data, and we are able to create models that do not overfit the data.

There has been much work expanding upon the neural networks seen here, from meta-learning across time series [28] to robustness of out-of-sample prediction testing at different timesteps [29], with many other methods that are beyond the scope of this chapter [30]. We hope to see greater use of deep neural networks as applied to financial time series forecasting.

Acknowledgements We are grateful to Amita Gajewar for her support in co-authoring the research conducted at Microsoft. We acknowledge Kuba Karpierz for insightful conversations and copy-editing. The work of A.K. is jointly supported by Microsoft and the National Science Foundation Graduate Research Fellowship under Grant No. DGE – 1656518. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. D. A. Dickey and W. A. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *Journal of the American Statistical Association*, vol. 74, no. 366, p. 427, Jun. 1979. [Online]. Available: https://doi.org/10.2307/2286348
2. R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning, "Stl: A seasonal-trend decomposition procedure based on loess (with discussion)," *Journal of Official Statistics*, vol. 6, pp. 3–73, 1990.
3. V. Gómez and A. Maravall Herrero, *Programs TRAMO and SEATS: instructions for the user (beta version: September 1996)*. Banco de España. Servicio de Estudios, 1995.
4. E. B. Dagum, "The x-11-arima seasonal adjustment method, statistics canada," 1980.
5. R. J. Hodrick and E. C. Prescott, "Postwar u.s. business cycles: An empirical investigation," *Journal of Money, Credit and Banking*, vol. 29, no. 1, p. 1, Feb. 1997. [Online]. Available: https://doi.org/10.2307/2953682
6. G. Box, *Time series analysis : forecasting and control*. Hoboken, N.J: John Wiley, 2008.
7. K. G. G. B. M. C. Jocelyn Barker, Amita Gajewar, "Secure and automated enterprise revenue forecasting," in *The Thirtieth AAAI Conference on Innovative Applications of Artificial Intelligence*, 2018, pp. 7567–7664.
8. R. Hyndman, "Cross-validation for time series," Dec 2016. [Online]. Available: https://robjhyndman.com/hyndsight/tscv/
9. A. Ng, "Neural network." [Online]. Available: https://slideplayer.com/slide/12620833/76/images/9/Neural+Network+12+Slide+by+Andrew+Ng.jpg
10. X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html
11. K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Dec. 2015. [Online]. Available: https://doi.org/10.1109/iccv.2015.123
12. A. Ng, "Deep learning." [Online]. Available: https://cs229.stanford.edu/notes2019fall/cs229-notes-deep_learning.pdf
13. A. Deshpande, "A beginner's guide to understanding convolutional neural networks." [Online]. Available: https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/
14. C. Olah, "Understanding lstm networks." [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/
15. J. Brownlee, "Time series prediction with lstm recurrent neural networks in python with keras," Aug 2019. [Online]. Available: https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras
16. J. Eddy, "Timeseries seq2seq." [Online]. Available: https://github.com/JEddy92/TimeSeries_Seq2Seq
17. A. Borovykh, S. Bohte, and C. W. Oosterlee, "Conditional time series forecasting with convolutional neural networks," 2017.
18. F. Chollet, "The keras blog." [Online]. Available: https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html
19. A. Koenecke and A. Gajewar, "Curriculum learning in deep neural networks for financial forecasting," in *Mining Data for Financial Applications*. Springer International Publishing, 2020, pp. 16–31. [Online]. Available: https://doi.org/10.1007/978-3-030-37720-5_2
20. Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th Annual International Conference on Machine Learning - ICML 09*. ACM Press, 2009. [Online]. Available: https://doi.org/10.1145/1553374.1553380

21. V. I. Spitkovsky, H. Alshawi, and D. Jurafsky, "From baby steps to leapfrog: How "less is more" in unsupervised dependency parsing," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2010, pp. 751–759.
22. A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu, "Automated curriculum learning for neural networks," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 1311–1320. [Online]. Available: http://proceedings.mlr.press/v70/graves17a.html
23. V. Cirik, E. Hovy, and L.-P. Morency, "Visualizing and understanding curriculum learning for long short-term memory networks," 2016.
24. N. Laptev, J. Yosinski, L. E. Li, and S. Smyl, "Time-series extreme event forecasting with neural networks at uber," in *International Conference on Machine Learning*, vol. 34, 2017, pp. 1–5.
25. A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 2016.
26. A. Gajewar, "Improving regional revenue forecasts using product hierarchy," in *39th International Symposium on Forecasting*, 2018.
27. I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.
28. T. S. Talagala, R. J. Hyndman, G. Athanasopoulos *et al.*, "Meta-learning how to forecast time series," *Monash Econometrics and Business Statistics Working Papers*, vol. 6, p. 18, 2018.
29. J. Sirignano and R. Cont, "Universal features of price formation in financial markets: perspectives from deep learning," 2018.
30. O. B. Sezer, M. U. Gudelek, and A. M. Ozbayoglu, "Financial time series forecasting with deep learning : A systematic literature review: 2005-2019," 2019.