

University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Marta Nowakowska

Student no. 385914

Time series prediction using self-attention models

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr hab. Marek Cygan, prof. UW
University of Warsaw

Co-supervisor:
dr hab. Piotr Miłoś
Institute of Mathematics, Polish Academy of Sciences

Warsaw, September 2022

Abstract

The Transformer model has excelled at various sequence-processing problems. However, the past works dedicated to time series forecasting with neural networks commonly simplified this task to predicting parameters of simple distributions, i.e. the Gaussian distribution. This thesis explores the idea of applying the Transformer to forecasting parameters of a granular categorical distribution for each predicted data point. This model is compared against the Gaussian distribution-based Transformer in the presented experiments.

Keywords

Time series, Neural network, Transformer, Self-attention model

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

I. Computing Methodologies
I.2. Artificial Intelligence
I.2.6. Learning
Connectionism and neural nets

Tytuł pracy w języku polskim

Prognostowanie szeregów czasowych przy użyciu modeli z uwagą

Contents

Introduction	5
1. Background	7
1.1. Time series and notation	7
1.2. Transformer model architecture	7
1.2.1. Overview	7
1.3. Attention	8
1.3.1. Scaled Dot-Product Attention	8
1.3.2. Convolutional Attention	9
1.3.3. Multi-headed attention	9
1.4. Training	10
1.4.1. Additional input	10
1.4.2. Scaling and weighted sampling	10
1.5. Metrics	10
1.5.1. Quantile loss	10
1.5.2. Risk metric	11
2. Approximating a continuous distribution with the Transformer	13
2.1. Serial Predictor	13
2.1.1. Discretization	14
2.1.2. Architecture's summary	14
3. Experiments	15
3.1. Datasets	15
3.2. Differences from previous works	15
3.3. Experimental setup	16
3.4. Results	16
4. Conclusion	19
4.1. Summary	19
4.2. Further work	19
Bibliography	21

Introduction

Time series and their importance

Forecasting future values and their probability distribution on the basis of historical data points is one of the most prevalent problems in both industry and research. For example, time series analysis can be used for predicting the number of ride-hailing trips completed during special events like holidays ([10]), but also for estimating stock market behavior ([11]) and solving resource allocation tasks ([12]).

This wide usage brought time series a lot of attention in the past. Some of the most known methods for their forecasting are based on the Box-Jenkins method like the ARIMA (autoregressive integrated moving average) [8] model, but there also exist ones using the exponential smoothing techniques and state space models [9]. Despite their popularity, they come with some fundamental disadvantages. For example, they need to be fitted separately for every similar sequence ([1], [7]) and cannot be trained to recognize these similarities which makes using them time-inefficient in the long term. Furthermore, these models typically use parameters that have to be handpicked by the user ([1], [7]), which makes them more suitable for field experts with knowledge of both parameter selection procedure and preferably time series' behavior.

The aforementioned issues are not a problem for deep neural networks, since they can be trained to recognize a group of similar time series and their hyper-parameters have to be only selected once for the training and not at the prediction time. What is more, once trained neural networks can be adapted to new data using transfer learning.

On top of that, neural networks proved to be very successful in other fields, including natural language translation, which is a problem with similar properties to time series forecasting. In particular, the translation can be defined as a mapping of one sequence to a second one, similarly to how one can map historical values to future ones. Thus, the achievements of the natural language processing (NLP) field, including RNN models have almost directly transferred to time series forecasting. However, RNNs and their successors (LSTM [2], GRU [3]) have difficulty with long sequences [4], causing them to struggle with time series that can contain long-term dependencies.

The next important step in the development of deep neural network adaptations for time series forecasting was the Transformer [5]. Instead of using recurrent networks, it relies almost entirely on the self-attention mechanism, which can be trained to estimate dependencies in arbitrarily long sequences, making it possible for the model to deal with larger-scale forecasting. However, there are problems with integrating it directly for time series forecasting usage. One of them is that the scaled dot-product attention can only capture dependencies between particular data points, but it cannot fully use the time series' local shape, which makes it struggle with capturing its seasonality and other patterns or anomalies ([1]). One solution to that problem was proposed in [1] by introducing convolutional self-attention.

Problem statement

Previously neural networks including the Transformer were commonly used for predicting parameters of a simple distribution like Gaussian per each forecasted point in time ([7], [1]). This work expands that idea to modeling any continuous distribution by approximating it with a categorical one. We present a comparison of both methods on commonly used datasets and metrics.

Contributions

The work described in this thesis has been done via a collaboration with scientists from the Institute of Mathematics of the Polish Academy of Sciences.

My most important contributions are:

1. Gaussian distribution-based Transformer
2. Experiments with the Gaussian distribution-based Transformer
3. Base implementation of the dataset feeder
4. Time features as in [1] (on the data feeding side)

Chapter 1

Background

1.1. Time series and notation

We define time series as a sequence of real numbers measured across time with a constant **frequency**. Frequency is defined in time units, e.g. weeks, days, or hours. For multiple time series represented as a matrix \mathbf{y} , where each row is a separate time series, $y_t^{(i)}$ denotes t time step of a series number i .

One can extend the definition above to a sequence of any data type, but it is not necessary for the datasets that we use for this thesis.

1.2. Transformer model architecture

Transformer [5] is a deep neural network model that relies on a scaled dot-product attention mechanism introduced in the same paper. There exist some modifications to the model presented in [5], but we follow the "decoder-only" [6] architecture type.

1.2.1. Overview

Our implementation of the Transformer model consists mainly of a causal convolution layer, positional encoding layer, N identical decoder layers, layer normalization, and a fully-connected layer. Most of the layers' outputs have d_{model} dimension. I present the general architecture in Figure 1.1.

Positional encoding allows us to feed the model embedded information about how the data points are ordered in the inputted time series. We use a trainable embedding for the positional encoding and add it to the input before it reaches the first decoder layer.

A single **decoder** layer consists of a causal attention sub-layer and a feed-forward sub-layer:

- The feed-forward sub-layer contains a simple feed-forward neural network with two dense layers and a ReLU activation in-between.
- The causal attention sub-layer contains the attention mechanism that I describe later on.

Both sub-layers are residual and wrapped with a layer normalization and a dropout layer.

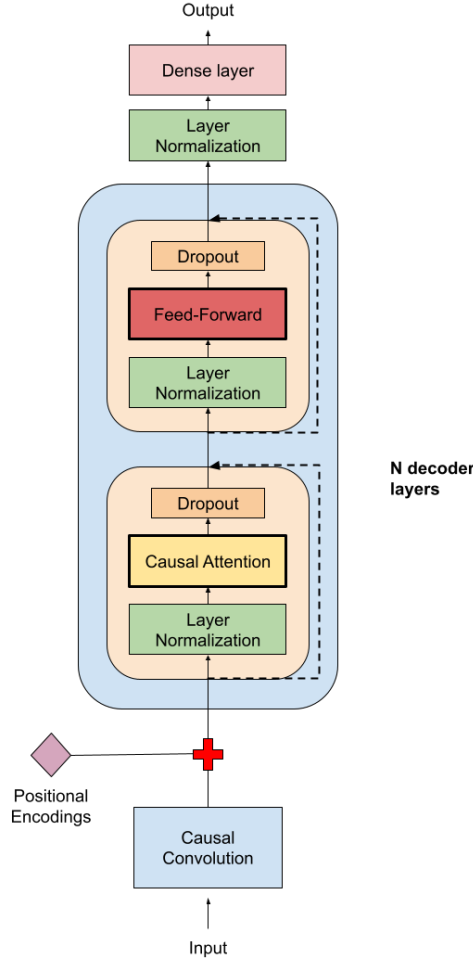


Figure 1.1: Transformer’s architecture. The dotted lines represent residual connections.

1.3. Attention

1.3.1. Scaled Dot-Product Attention

Attention is the key to explaining how the Transformer works. It enables us to train the neural network to encode the relations between different parts of the input (e.g., a time series’ data points).

Scaled dot-product attention introduced along with the Transformer architecture in [5] is perhaps the most known kind of attention function. One can informally see its calculation as operations on a neural network-approximated dictionary with vector keys and corresponding values. We use a dot operation between the query being matched and the dictionary’s keys as a measure of similarity. When they are orthogonal, the dot product will be equal to 0. On the other hand, for vectors of fixed length, the dot product is maximal when they have the same direction.

To calculate the attention score vector between one data point and every that comes after, one first needs to provide a query vector for the first data point and a set of vector pairs consisting of keys and values for all the other data points. We match our query against all of the keys by a dot product and use the softmax function to normalize the result. When the result of softmax is large between two data points, we can informally say that the attention

relates them. Finally, by multiplying the softmax results with the corresponding values we get a set of weighted values vectors. After summing them we get the attention score vector for one data point.

In reality, we do not need to calculate the attention sequentially for every data point, but rather use matrix operations. One needs to provide three matrices to the attention layer: $Q, K \in \mathbb{R}^{d_I \times d_k}, V \in \mathbb{R}^{d_I \times d_v}$ (respectively called queries, keys and values). They are created by multiplying the input matrix $I \in \mathbb{R}^{d_I \times d_{model}}$ with three distinct trainable weight matrices $W_Q \in \mathbb{R}^{d_{model} \times d_k}, W_K \in \mathbb{R}^{d_{model} \times d_k}, W_V \in \mathbb{R}^{d_{model} \times d_v}$.

For given Q, K, V we compute the attention with the following formula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}} - M)V \text{ (from [1])}$$

where d_k stands for the dimension of Q and K . Dividing by its square root is supposed to balance out the growth of Q and K^T dot product that occurs with larger values of d_k .

M is a **mask matrix** that sets all the elements corresponding to connections between future and past (upper triangular) to $-\infty$. It prevents the model from attempting to relate past events to future ones.

While calculating the softmax score for any two data points in a time series, the scaled dot-product attention does not use any information about other data points. We call this property **context-blindness**. Sometimes this might lead the model to assign a strong attention score between events that have similar values, even if their contexts (neighboring data points) are very different. That might cause the model to incorrectly recognize patterns in the time series ([1]).

1.3.2. Convolutional Attention

Causal convolution is a convolution used on input padded with $k - 1$ zeros at the start, where k denotes the size of the kernel.

This type of convolution is particularly useful for time series since it enables us to make predictions with only partial input, essentially generating a completely new time series.

The convolutional attention [1] is calculated in almost the same way as the scaled dot-product attention. It only differs in the way it computes queries and keys. Instead of applying a matrix multiplication with weight matrices, it uses a causal convolution with a stride of 1. When the kernel's size is equal to 1 it is equal to the matrix multiplication.

The usage of convolution is supposed to offset the dot-product attention's context blindness. Since the convolution layer learns to recognize the input's context, a stronger attention score should be assigned to similar events.

1.3.3. Multi-headed attention

Each attention block might contain more than one parallel attention layer, aka the attention "head". In theory, this allows for different layers to learn various kinds of relationships in the input and is a very successful tool e.g. in natural language processing [5].

The outputs of all h attention heads are concatenated and multiplied by a weight matrix $W^O \in \mathbb{R}^{hd_v \times d_{model}}$ to squeeze them back into $d_I \times d_{model}$ shape before passing them to the next layer.

To offset the additional computational cost related with multiple attention heads, we set d_k and d_v to d_{model}/h .

1.4. Training

1.4.1. Additional input

Following [1], we use additional variables as part of the model’s input aside from the time series itself. There are three kinds of additional input we use:

1. Numerical information about the time the input was recorded (year, month, day-of-the-week, hour-of-the-day, minute-of-the-hour).
2. Identifier of the time series.
3. Input’s distance from the sequence’s start.

We embed all of the additional variables to the same dimension, sum those embeddings and normalize them. Finally, we add the result to our model’s (embedded) input and normalize it as well.

1.4.2. Scaling and weighted sampling

The time series in a single dataset can vary in their ranges, which might obstruct the model’s learning process. Due to the network’s non-linearities, it needs to size down the input in its first layer and then reverse this scaling on the output ([7]). Therefore, following [1] and [7] we scale the input and output by respectively dividing and multiplying them by a scale factor v_i :

$$v_i = r + \frac{\sum_{t=1}^T y_t^{(i)}}{T},$$

where T is $y^{(i)}$ ’s length and r is a regularization term.

Furthermore, we adopt a weighted sampling technique in our training, similarly to [1] and [7]. Its main purpose is to counteract the model’s tendency to underfit large-scale data since it occurs more rarely in the datasets. We calculate the weights by normalizing the scale factor v_i .

1.5. Metrics

1.5.1. Quantile loss

Quantile loss is a metric designed to evaluate and/or train models designed with quantile prediction in mind. We define ρ -quantile loss as follows:

$$QL_\rho(\hat{y}, y) = \max(\rho(y - \hat{y}), (\rho - 1)(y - \hat{y})),$$

where \hat{y} denotes the model’s prediction for the ρ quantile and y is the ground truth value.

This metric is characterized by a stronger penalization of over-predictions for ρ smaller than 0.5, and under-predictions for ρ greater than 0.5.

1.5.2. Risk metric

$Risk_\rho$ is a type of normalized quantile loss metric defined as follows for $\rho \in (0, 1)$:

$$Risk_\rho(\hat{\mathbf{y}}, \mathbf{y}) = 2 \frac{\sum_{i,t} QL_\rho(\hat{y}_t^{(i)}, y_t^{(i)})}{\sum_{i,t} |y_t^{(i)}|}$$

It is very popular in time series prediction papers that use the Transformer (e.g. in [1], [7]), which leads us to choose it as the metric we use in the model's evaluation. Furthermore, quantile loss allows us to assess how well our model estimates the series' extreme values. It is useful for applications like electricity consumption risk estimation, where one might ask how likely it is in the given time frame that the power consumption reaches a certain level.

Chapter 2

Approximating a continuous distribution with the Transformer

While the Transformer has already proven to be a very efficient tool for time series prediction, we believe it has a lot more potential that could be used without changing its architecture. In the past, works like [1] were centered around the idea of every point in a time series being drawn from some simple distribution, e.g. Gaussian. Thus, the model's goal was to predict the parameters of these distributions. In contrast, one could apply a network to directly forecast series values, which seems like a much simpler approach. However, one of the main benefits of predicting distributions is how it opens the door to risk modeling. This is especially useful when asking how likely is the time series to exceed a certain value, e.g. while modeling high-risk markets.

This thesis explores whether the Transformer's performance on time series prediction can be improved by training it to predict a categorical distribution for each point in the sequence instead of a Gaussian one. Along with the discretization of the continuous space, this should enable our Transformer to model a broad spectrum of distributions. One should also take into consideration that Transformer is very successful in tasks requiring the prediction of thousands of categories like language translation ([5]). This could potentially indicate that the previous usages of the Transformer for the time series prediction were underutilizing its abilities.

We further hypothesize that this ability to model a wide range of distributions could be very useful for especially difficult or non-standard datasets that simpler models would struggle with. For an instance, it would be inefficient to train a Gaussian distribution-based Transformer on how to model a multimodal distribution.

2.1. Serial Predictor

The main contribution of our work is a time series forecasting model named Serial Predictor that uses the Transformer at its core and does not modify its architecture. The only introduced changes are to the input preprocessing, as we use a discretization process on it. It enables us to turn the input into a series of discrete tokens from the $[0, 1]$ interval. We hypothesize that discretizing and embedding the time series before feeding it to a Transformer can make it easier for the model to train since it helps to normalize the input.

2.1.1. Discretization

We call discretization the process of bucketing each value in a real-valued series, i.e. mapping it to a discrete categorical value from a finite number of categories.

Our discretization algorithm requires two parameters: the maximum value we expect in the datasets, minimal (usually equal to 0 in the datasets we consider), and the exact number of categories (*vocabulary_size*). To serialize a series, we first assign intervals of real values uniformly starting with the minimal value to *vocabulary_size* categories. Afterward, we bucket every value in the series to its appropriate interval/category.

Deserialization is the reverse of this process in which we output the minimal value assigned to a specific bucket.

Since the number of categories we use is finite, serialization is bound to lose the precise value of some data points in a series. With enough buckets used, we hope this drop in precision should be negligible in comparison to our gains in the prediction’s reliability.

2.1.2. Architecture’s summary

We use the Transformer’s architecture as it was described in Section 1.2. Our input consists of serialized and embedded time series. The final decoder layer outputs *vocabulary_size* logits, each of them corresponding to a single bucket. During the evaluation process, we sample the buckets from a categorical distribution given by the values of output logits. Finally, we deserialize the sampled bucket.

We use a category cross-entropy loss to train our model.

Chapter 3

Experiments

In our experiments, we decided to compare our Serial Predictor with a time series forecasting model that we call Gaussian Predictor. This model uses the same Transformer body as the Serial Predictor, but it does not use serialization on its input and outputs parameters of a Gaussian distribution. We use the log-likelihood loss for training the Gaussian Predictor.

3.1. Datasets

I chose two datasets for presenting our results in this thesis: *electricity* and *traffic*. They are both based on real-world data and are easily accessible through the GluonTS ([13]) package. Furthermore, they are used in many of the papers we base our work on ([1], [7]), which makes them a good reference point.

The *electricity* dataset is the electric consumption data collected from 370 customers recorded with a frequency of 1 hour ([1], [7]). The *traffic* dataset consists of the hourly occupancy rates of 963 lanes in San Francisco ([1], [7]). Since *traffic* displays higher differences between patterns observed on weekends and weekdays, it is considered a more demanding dataset and is useful for finding out how well the model can capture both long-term and short-term dependencies ([1]).

3.2. Differences from previous works

During the late stages of our work, we contacted the authors of [1] and received their code. We discovered some differences in their evaluation processes. Mainly, their autoregressive prediction was based not on sampled data points, but rather on the predicted Gaussian means.

Additionally, the datasets provided by GluonTS ([13]) are not identical to the ones used in the experiments from [1] or [7]. Even though the data's source is the same, its processing was different¹.

Furthermore, we did not include the sparse attention ([1]) in our experiments. This difference should not have contributed to the gap between Gaussian and Serial Predictors, though, since both operate with the same attention mechanism.

¹Source: <https://github.com/awsmlabs/gluon-ts/issues/830>.

3.3. Experimental setup

We used the Adam optimizer for all our experiments along with an rsqrt decay learning rate scheduler. Evaluations were carried out autoregressively over a prediction length of 24 hours. During the evaluation process, we always use the series' suffix of *series_length* = 768. For the training, we use random series samples of the same length.

Every experiment was run as follows:

- We used a hard limit of $n_steps = 1000000$ training steps with early stopping *patience* = 10. There were 1000 linear warm-up steps (*warmup_steps* = 1000) and *batch_size* = 16.
- The evaluation was run every *eval_every* = 1000 steps over $n_eval_batches = 128$ samples.

We ran a randomized parameter grid search over the following parameters:

- Convolutional attention's kernel size: *conv_attention_kernel_width* $\in \{1, 2, 3, 6, 9\}$
- Number of heads in the multi-headed attention: *n_heads* $\in \{1, 2, 4, 8\}$
- Dropout rate (same in various places): *dropout* $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9\}$
- Most of the layers outputs' size: *d_model* $\in \{64, 128, 256, 512\}$
- Number of decoder layers: *n_layers* $\in \{2, 4, 6, 8\}$
- Learning rate's scheduler starting constant: *constant* $\in \{0.001, 0.003, 0.01, 0.03, 0.1\}$. It is used to recalculate the current learning rate based on the warmup and rsqrt decay.
- Usage of weighted sampling (see Section 1.4.2): *weighted_sampling* $\in \{True, False\}$
- (Serial Predictor only, see Section 2.1.1) Maximum value we can predict: *high* $\in \{2, 5, 10\}$
- (Serial Predictor only, see Section 2.1.1) Number of buckets: *vocab_size* $\in \{1024, 2048, 4096\}$

We ran every experiment with a 16GB of RAM limitation on a cluster containing various GPUs: GTX 1080 Ti, Titan X, Titan V, RTX 2080 Ti, and RTX A6000. There were respectively 120 and 125 experiments with the Serial Predictor and the Gaussian Predictor. This slight difference is due to some of the experiments not being able to fit in the memory limit.

3.4. Results

Best performing models' comparison

We have not managed to reproduce the results from [1] with our Gaussian Predictor, which we attribute to many differences between the works that we investigated as I described in Section 3.2. There is, however, a large gap between the performance of our Gaussian Predictor and the Serial one in every evaluated metric and dataset, with the Serial Predictor scoring almost a third of the $R_{0.5}$ metric over the *traffic* dataset.

	Gaussian Predictor		Serial Predictor		Enhancing Locality ([1])	
	$R_{0.5}$	$R_{0.9}$	$R_{0.5}$	$R_{0.9}$	$R_{0.5}$	$R_{0.9}$
electricity	0.127	0.087	0.069	0.041	0.059	0.034
traffic	0.402	0.224	0.123	0.093	0.122	0.081

Table 3.1: Results of our best performing models’ and the ones cited from [1]. The best results for each dataset and metric are marked in bold. R_p refers to the risk metric described in Section 1.5.2.

Serial Predictor’s performance analysis

Finally, we present how a different number of discretization buckets (*vocab_size*) influences the Serial Predictor’s performance. In general, one should expect that the higher *vocab_size* we use, the better performance we can achieve, since it allows us to have more precision (granularity) in our predictions, and the Transformer model is successful in tasks requiring thousands of categories.

For the *traffic* dataset we used the best performing model’s hyperparameters and fixed them, changing only the *vocab_size*. For the *electricity* dataset we used the second best performing model’s parameters, as the first one could only be trained with our memory available with the lowest *vocab_size*.

Our model works best with *vocab_size* = 4096 on the *traffic* dataset (the highest one), but in the *electricity* dataset its performance seems slightly hindered with the highest granularity. This suggests it might be still necessary to tweak the *vocab_size* to every dataset.

	<i>vocab_size</i> = 1024		<i>vocab_size</i> = 2048		<i>vocab_size</i> = 4096	
	$R_{0.5}$	$R_{0.9}$	$R_{0.5}$	$R_{0.9}$	$R_{0.5}$	$R_{0.9}$
electricity	0.074	0.046	0.069	0.044	0.069	0.047
traffic	0.133	0.106	0.135	0.99	0.123	0.093

Table 3.2: Comparison of Serial Predictor models with different *vocab_size*.

Chapter 4

Conclusion

4.1. Summary

This thesis compares the accuracy of a Transformer model architecture used for the time series prediction in two ways. The first one is based on important past time series works and involves using the Transformer to predict the parameters of a Gaussian distribution for each forecasted time point. The second way is our Serial Predictor which approximates a continuous distribution with a categorical one. Although the results of this direct comparison are promising, we did not manage to reproduce ones similar enough to [1].

4.2. Further work

While the work presented in this thesis uses its fair baseline model, further experiments should focus on reproducing or readjusting the results from [1] and crafting a consistent comparison with the past time series research.

Furthermore, it is important to attempt to find datasets that might benefit more from our approach in comparison to established Transformer-based time series models. Since the Serial Predictor can theoretically learn almost any distribution, we hope that it might perform better on data that has low determinism qualities like the stock or cryptocurrency markets.

This work also suggests that it might be beneficial for any further research involving the Transformer model to leverage its ability to successfully predict many categories.

Bibliography

- [1] Shiyang Li, Xiaoyong Jin, Yao Xuan, Xiyu Zhou, Wenhui Chen, Yu-Xiang Wang and Xifeng Yan. *Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting*. arXiv:1907.00235, 2019.
- [2] Sepp Hochreiter and Jürgen Schmidhuber. *Long short-term memory*. Neural computation, 9(8):1735–1780, 1997.
- [3] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. *On the properties of neural machine translation: Encoder-decoder approaches*. arXiv preprint arXiv:1409.1259, 2014.
- [4] Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. *Sharp nearby, fuzzy far away: How neural language models use context*. arXiv preprint arXiv:1805.04623, 2018.
- [5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. *Attention is all you need*. NIPS 2017.
- [6] Peter J Liu, Mohammad Saleh, Etienne Pot, Ben Goodrich, Ryan Sepassi, Łukasz Kaiser, and Noam Shazeer. *Generating wikipedia by summarizing long sequences*. arXiv preprint arXiv:1801.10198, 2018.
- [7] Valentin Flunkert, David Salinas, and Jan Gasthaus. *Deepar: Probabilistic forecasting with autoregressive recurrent networks*. arXiv preprint arXiv:1704.04110, 2017.
- [8] George EP Box and Gwilym M Jenkins. *Some recent advances in forecasting and control*. Journal of the Royal Statistical Society. Series C (Applied Statistics), 17(2):91–109, 1968.
- [9] R. Hyndman, A. B. Koehler, J. K. Ord, and R. D. Snyder. *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Series in Statistics. Springer, 2008. ISBN 9783540719182
- [10] Nikolay Laptev, Jason Yosinski, Li Erran Li, and Slawek Smyl. *Time-series extreme event forecasting with neural networks at Uber*. International Conference on Machine Learning.
- [11] J. L. Ticknor. *A bayesian regularized artificial neural network for stock market forecasting*. Expert Systems with Applications, 40(14):5501–5506, 2013.
- [12] Hongjie Chen, Ryan A. Rossi, Kanak Mahadik, Sungchul Kim and Hoda Eldardiry. *Graph Deep Factors for Forecasting*. arXiv:2010.07373.
- [13] Alexander Alexandrov, Konstantinos Benidis, Michael Bohlke-Schneider, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Danielle C. Maddix, Syama Rangapuram, David Salinas, Jasper Schulz, Lorenzo Stella, Ali Caner Türkmen and Yuyang Wang. *GluonTS: Probabilistic and Neural Time Series Modeling in Python*. Journal of Machine Learning Research.