CHALMERS
UNIVERSITY OF TECHNOLOGY

Time Series Forecasting With Neural Networks

Minimizing Food Waste By Forecasting Demand in Retail Sales

Master's thesis in Complex Adaptive Systems

ARIANIT ZEQIRI,
MORAD MAHMOUDYAN

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Time Series Forecasting Using Neural Networks

Minimizing Food Waste By Forecasting Demand in Retail Sales

ARIANIT ZEQIRI
MORAD MAHMOUDYAN

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2021

Time Series Forecasting Using Neural Networks
Minimizing Food Waste By Forecasting Demand in Retail Sales
ARIANIT ZEQIRI
MORAD MAHMOUDYAN

© ARIANIT ZEQIRI & MORAD MAHMOUDYAN, 2021.

Supervisors: Anton Graf, Sina Torabi
Examiner: Mats Granath, Department of Physics

Master's Thesis 2021
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Grocery bags containing different food articles.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Time Series Forecasting Using Neural Networks
Minimmizing Food Waste by Forecasting Demand in Retail Sales
ARIANIT ZEQIRI
MORAD MAHMOUDYAN
Department of Physics
Chalmers University of Technology

Abstract

A third of the food produced for human consumption is wasted annually, amounting to 1.3 billion tons of food waste per year [10]. Minimizing these enormous quantities of waste would not only be beneficial for the planet but also help feed an ever increasing human population. One way of minimizing this waste is by helping retail sellers better plan their logistical operations by accurately predicting the demand of goods using forecasting models. The procedure in time series forecasting has traditionally been to use statistical models such as autoregressive integrated moving average (ARIMA) and exponential smoothing methods. These methods have been shown to be limited in their predictive capabilities as the sizes of data sets and the number of variables increases. In the last decade new machine learning algorithms have been used extensively in various fields and has opened up the door for utilization of models based on neural networks in time series forecasting. A subset of these new machine learning algorithms, such as transformer based models and recurrent neural networks, have been proven to be especially suitable for temporal data. In this thesis we investigate two models, Temporal Fusion Transformers (TFTs) and Deep Temporal Convolutional Networks (DeepTCNs), showcasing their abilities to generate accurate forecasts of retail sales on a real-world data set. We demonstrate, using several metrics, their ability to outperform baseline models.

Keywords: forecasting, machine learning, neural networks, food waste, retail sales, dilated convolution, temporal fusion transformer, temporal convolutional network

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
1.1 Aim 2
1.2 Delimitation 2
1.3 Thesis layout 3

2 Theory **5**
2.1 Traditional time series forecasting 5
2.1.1 Multivaraite time series forecasting 7
2.2 Neural Networks 7
2.2.1 Dilated convolution 9
2.2.2 Recurrent Neural Networks 10
2.2.3 Long Short-Term Memory Networks(LSTMs) 12
2.2.4 Sequence to sequence models 13
2.2.5 Self-attention 13
2.2.6 Transformers 14
2.3 Quantile Loss 15
2.4 Metrics 16

3 Methods **19**
3.1 The Data set 19
3.1.1 Data Splitting 23
3.2 Deep temporal convolutional network (DeepTCN) 24
3.3 Temporal fusion transformer (TFT) 26
3.3.1 Gated Residual Network (GRN) 26
3.3.2 Variable Selection Network 27
3.3.3 Static covariate enconders 28
3.3.4 Sequence to sequence module 28
3.3.5 Temporal Fusion Decoder 29
3.3.6 Modifications 29

4 Results **31**

5 Discussion **37**

5.1 Challenges 37
5.2 Future Work 38
5.3 Conclusion 40

Bibliography **44**

List of Figures

2.1 A graph over a perceptron model. The left most layer represents the inputs to the model. Each input has a weighted connection to the neuron and the these connections are then summed by the neuron. The summed neuron value is then fed to an activation function g. . . 8
2.2 Stacked dilated Convolutional layers for an input sequence of 16 and a kernel size of 2. At each layer the dilation is increased with a power of 2. 10
2.3 To the left: a simple recurrent neural network consisting of one input neuron, one hidden neuron and one output neuron. To the right: the same RNN unrolled in time. The unrolling does not affect the weights, they remain the same for each time step. 11
2.4 High-level architecture of a transformer. The left part of the figure depicts the decoder structure consisting of multi-head attention and a feed forward network. To the right part we have the decoder structure which has one more, masked, attention layer than the encoder. Output probabilities are produced by applying a softmax activation function on the output of the decoder. 15

3.1 Frequency distribution of the percentage of date range covered by all articles in our data set. 20
3.2 A heatmap of the density of points where each point represents one article. The x value denotes the article's date range and the y value denotes the number of days that article has been sold. There is a large concentration of articles around the origin. 21
3.3 Heatmap of the results produced by repeatedly applying the filter date range function for various date ranges and date fractions. A possible region of interest is also displayed in the heat map. The color bar denotes the fraction of articles remaining after applying the filter function. 22
3.4 A box and whisker plot over daily sales for the entire data set grouped by day of week. 23
3.5 Each of the eight graphs represent the distribution of the number of articles sold every day for a specific store. 23
3.6 A Diagram of a residual block with two dilated convolutions. d and k refer to dilation rate and kernel size respectively 25

3.7 A Diagram over the complete DeepTCN architecture, with multiple stacked residual blocks and a decoder module. Furthermore a VSN module has been added at the onset. 26
3.8 Architecture of a VSN model. 28
3.9 High-level architecture of TFT. **S** denotes static input, $\{\chi_i\}_{i=t-k}^{t}$ denotes historical temporal data and $\{\chi_i\}_{i=t+1}^{t+\tau}$ denotes future temporal data. Skip connections are indicated by dashed lines. Context vectors produced by the static covariate encoder are fed to various modules, including VSNs, Encoders and GRNs. 30

4.1 The training and validation loss for both DeepTCN models 33
4.2 Loss curves for four different TFT networks with $h = 8$, $h = 16$, $h = 32$ and $h = 64$. Red lines indicate training loss and black lines depict validation loss. 33
4.3 Forecasts made using the SES baseline model 34
4.4 Time series forecasting done by TFT with $h = 32$ on six random articles from two stores, store 173 and store 3998. See figure 4.5 for a detailed description of the layout. 34
4.5 Forecasting done by DeepTCN with k=2 for a some random samples in the test data. The transparent red area defines the models quantile forecasts - the upper boundary is the 90th quantile while the lower boundary is 10th quantile. The grey line represents the historical data that the model has access too. The data points marked by black dots are the actual predictions. 35

List of Tables

4.1 The following table shows the performance of the models on four chosen metrics (Normalized Quantile Loss, MAE, MSLE and MAAPE). The performance of the models where measured on the test data. All models presented in this table used the data splitting method described in algorithm 1 in section 3.1.1. 32

1

Introduction

A third of the food produced for human consumption is wasted annually, amounting to 1.3 billion tons of food waste per year[10]. All resources that go into producing these huge amounts of food are in the end wasted and, furthermore, the greenhouse gas emissions that are generated at the agricultural, processing and transportation stages are unnecessary and can potentially be avoided [10]. Food wastage occurs throughout the life cycle of food systems, from production to consumption, but the amount wasted at each stage differs depending on the economic status of the nation in question [10]. A significant part of the wastage occurs for example at the consumption stage in the industrialized world while the same effect is not seen in low-income nations [10][15]. With an ever increasing population food production must clearly increase to meet the future demand of an increasing and affluent population. Decreasing food waste is therefore a step in the right direction for improving global food security as well as mitigating the effects of climate change. Decreasing the food loss at the consumption stage can be achieved by implementing forecasting models that can forecast the demand for individual food articles and help retail business owners better plan the logistics of their operations.

Besides the environmental benefits there are economical benefits of generating accurate forecasts of retail demand. Accurate forecasts lead to decreased uncertainty for retailers since they can replenish articles at a timely manner which in turn increases sales by means of product availability. Improving the supply chain and inventory management has therefore economical benefits apart from the environmental benefits.

Time series forecasting is the task of predicting future values of a variable by means of analyzing a range of features that may influence the variable of interest. Examples of features that may affect the future values are historical patterns, future time dependent factors, business decisions and external factors. The ability to make accurate forecasts is essential in various businesses and can be used to optimize business processes as well as enable data driven decision making. Predicting future demand of items in retail [9], future load on power grids [25] and the general use of forecasting in economic models [5] are a couple of real-world applications where the problem of time series forecasting is important.

Statistical methods such as moving average, exponential smoothing [11] and autoregressive integrated moving average (ARIMA) [33] have traditionally been used

extensively in time series forecasting but as the number of variables and the sizes of data sets have increased the manual tuning of parameters associated with statistical methods has become unfeasible. New deep learning algorithms in areas such as image detection [28], bioinformatics [7], automatic speech recognition [14] and natural language processing [12] have proven to be able to model complex systems in real life by automating the tuning of parameters in complex functions and, in the end, generating black-box models that in all instances outperform classical interpretable models. The rise of these new algorithms the last decade has brought about new opportunities for creating neural network based forecasting models for big data sets.

Several neural network based forecasting models have been proposed for time series forecasting. Most of these models are designed to act on univariate data such as the autoregressive recurrent forecaster (DeepAR) model[29] or the Multi-Horizon Quantile Recurrent Forecaster[36] model that was tested on univariate sales data from Amazon. Multivariate models have not been explored as much but there are some examples of them such as the Temporal Fusion Transformer model (TFT)[23], a particularly complex architecture with multiple modules designed to act on certain types of data. A more popular approach for creating multivariate models is to use various Temporal Convolutional Network (TCN) architectures. DeepTCN[6], SeriesNet[27] and M-TCN[35] are a few TCN based models that have been proposed for time series forecasting.

In this report we will demonstrate two neural network based forecasting models that are trained using a real-world data set consisting of retail sales data. The trained models will be used to forecast food purchases in order to help retailers better plan their logistic operations and in the end decrease food wastage.

1.1 Aim

The aim of this thesis is to examine various neural network models on a retail sales data set and determine whether neural network based models are suitable for retail sales forecasting. A second objective is to examine forecasting models with the aim that such models should help alleviate the indirect strains put on the environment by inefficient management of food resources at the consumption stage, causing unnecessary food wastage.

1.2 Delimitation

The focus of this paper is on machine learning approaches, more specifically on multivariate models with multi-head prediction capabilities. The derived models should be able to handle time dependent variables as well as static variables. A fully interpretable model cannot be guaranteed and is outside the scope of this thesis. Moreover, we will confine the project to testing already existing models on the retail data set that has been made available to us, and we will refrain from creating a novel model specifically for this task.

1.3 Thesis layout

The thesis is divided into four main sections: theory, methods, results and discussion. In the theory section of this paper an overview of basic theory in time series forecasting, statistical methods as well as neural networks is provided. In the method section our approach and the models used are explained in detail. The method section is followed by the presentation of our results and in the end a final section containing discussions and conclusion is presented.

well written intro!

2

Theory

In the following sections theory about time series forecasting is explained in detail, starting with time series predictions using statistical models, continuing with time series predictions using neural networks and ending with descriptions of quantile losses and different metrics.

2.1 Traditional time series forecasting

Time series forecasting entails using observations of historical values of a variable y, made at discrete time steps, to predict the future value of said variable, $\hat{y}$. The aim is therefore to find a function that can approximate the future value y_{t+1} based on historical observations of y: $\hat{y}_{t+1} = f(y_1, y_2, y_3, ..., y_t)$. Traditionally one has used different statistical methods for deriving these approximations.

Two popular statistical linear models are the Autoregressive (AR) and the Moving Averages (MA) models. In an autoregressive model the predictions for the variable of interest are made using linear combinations of its past values. The forecasting value $\hat{y}_t$ in an autoregressive model with order p is therefore the linear combination of the p past values plus some noise $\epsilon_t \sim N(0, \sigma^2)$ [18][1] as well as constant value c, as seen in equation 2.1. L is a lag operator, defined as $L^k X_t = X_{t-k}$, which acts on elements of a time series by shifting them k time steps backwards.

$$\hat{y}_{t+1} = c + \sum_{i=0}^{p} \phi_i y_{t-i} + \epsilon_t = c + \sum_{i=0}^{p} \phi_i L^i y_t + \epsilon_t. \tag{2.1}$$

In contrast to the AR model whose predictions are based on past values the Moving Average model generates predictions based on past forecasting errors. A MA model with an order of q is therefore defined as the linear combination of the past q forecasting errors

$$\hat{y}_{t+1} = \mu + \sum_{j=0}^{q} \theta_t \epsilon_{t-j} + \epsilon_t = \mu + (1 + \sum_{i=0} \theta_i L^i)\epsilon_t \tag{2.2}$$

where $\theta_t \in \mathbb{R}$ are model parameters, $\epsilon_t \sim N(0, \sigma^2)$ are forecasting errors modeled as

how is it initiated?
need a forecast to
get the error?

white noise and μ is the average value of the time series.

For many statistical time series forecasting models such as AR the concept of stationarity is an important precondition. This implies that the statistical mean μ and variance σ^2 of a time series should be independent of time. In many cases this precondition cannot be satisfied, in which case differencing, the processes of removing temporal dependencies from the data set, can be applied in order to impose the stationarity precondition.

The AR and MA models are often times combined to form more complex models such as Autoregressive Moving Average (ARMA)[18], Autoregressive Integrated Moving Average (ARIMA)[33] and Auto Regressive Integrated Moving Average with Exogeneous Input (ARIMAX)[37]. ARMA, which is the simplest model of the three, is a combination of $MA(q)$ and $AR(p)$

$$ARMA(p,q) = AR(p) + MA(q) = c + \epsilon_t \sum_{i=0}^{p} \phi_i y_{t-i} + \sum_{j=1}^{q} \theta_t \epsilon_{t-j}. \tag{2.3}$$

An alternative approach to AR and MA is to utilize an Exponential Smoothing (ES) mechanism where the weights are not weighted equally as in the case of the AR and MA models. The weights in an ES model are instead decreased exponentially over time. One of the simplest ES models that exists is the Simple Exponential Smoothing (SES) model [21][18] which is utilized when a time series has no observed trend. When a time series exhibits a trend then other models are more appropriate such as a Double Exponential Smoothing (DES) model. In SES the prediction $\hat{y}$ for y at time $t+1$ with a smoothing parameter $\alpha \in [0,1]$ is defined as:

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha) y_{t-1} + (1-\alpha)^2 y_{t-2} \tag{2.4}$$

From a statistical point of view a forecasting task can be regarded as the modeling of the conditional distribution $P(y_{t+1}|y_1, y_2, ..., y_t)$. However, on many occasions there is a need of being able to forecast multiple time steps into the future. A distinction must therefore be made between the two tasks. In multi-horizon forecasting multiple predictions are made at τ future time steps, $(y_{t+1}, y_{t+2}, \ldots, y_{t+\tau})$. For all models presented so far multi-horizon forecasting can be achieved by means of recursion. This implies that at each time step the forecasted value is taken to be the ground truth. This generative approach approximates the future values by factorizing the joint probability of future values, given past values, as the product of conditional probabilities[6]

$$P(y_{t+1}, y_{t+2}, ..., y_{t+\tau}|y_1, y_2, ..., y_t) = \prod_{\pi=1}^{\tau} p(y_{t+\pi}|y_1, y_2, ..., y_{t+\pi-1}). \tag{2.5}$$

2.1.1 Multivaraite time series forecasting

So far we have only considered univariate time series where forecasting is done by only taking into consideration the values of the variable of interest. In real life applications a multivariate setting is more suitable, with the expectation that models that utilize multiple variables when generating forecasts will be more precise. The multivariate setting requires models that are more complicated than the simple univariate models we have so far presented. While univariate time series forecasting may be viable in a theoretical framework they do not perform as well as multivariate models since univariate models do not take into consideration the relationship between multiple interconnected and interdependent features [19].

In a multivariate setting the data is divided into multiple categories with the primary divisions being: y_t, the variable that is to be forecasted, and time dependent input variables $\boldsymbol{\chi}_t$. The time series can also have accompanying static covariates variables $\mathbf{s}$ which are variables that do not change over time. The time dependent variables can be further subdivided into observed inputs $\boldsymbol{z}_t$, that cannot be determined prior to their observation and known inputs $\boldsymbol{x}_t$ that can be predetermined. In the above notation the index t represents the time at which an observation of the variable was made.

So far we have only considered single entity time series but in many cases one has multiple time series that are related to one another such as the power load for different stations or the number of items sold of a specific article at different stores. Assume that we have I entities, where each entity i represents a time series with an associated static covariate $\mathbf{s}^{(i)} \in \mathbb{R}^{m_s}$, scalar target $y_t^{(i)} \in \mathbb{R}$ and input $\boldsymbol{\chi}_t^{(i)} \in \mathbb{R}^{m_x}$. Under these assumptions the task becomes finding a model that approximates the conditional distribution:

$$P(\boldsymbol{y}^{(j)}_{t+1:t+\tau} | \boldsymbol{y}^{(j)}_{0:t}, \boldsymbol{\chi}^{(j)}_{0:t}) = \prod_{\pi=1}^{\tau} p\Big(y^{(j)}_{t+\pi} | \boldsymbol{y}^{(j)}_{0:t}, \boldsymbol{\chi}^{(i)}_{0:t}, i = 1, ..., I\Big). \tag{2.6}$$

The inclusion of multivariate data and multiple related time series is too complex for traditional statistical methods to generate reliable forecasts and one needs to instead use machine learning approaches due to their ability to generalize for any type of data.

2.2 Neural Networks

Machine learning algorithms are in essence applied statistics with emphasis on function approximation using powerful computers. Neural networks are machine learning models that define a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$ [13] that approximates the parameter values θ through a processes of iterative parameter improvement by minimizing an error function involving the model predictions $\hat{y}$ and the targets y. The underlying building block in a neural network model is the perceptron which is essentially a binary linear classifier, see figure 2.1 and equation 2.7.

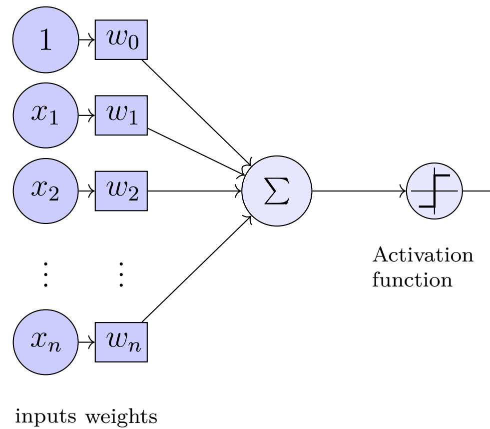


Figure 2.1: A graph over a perceptron model. The left most layer represents the inputs to the model. Each input has a weighted connection to the neuron and the these connections are then summed by the neuron. The summed neuron value is then fed to an activation function g.

$$f(\mathbf{x};\mathbf{w}) = g(w_0 + \sum_{j=1}^{n} x_j w_j) = g(\mathbf{x}^T\mathbf{w}) \tag{2.7}$$

Equation 2.7 shows the mathematical operations performed by a perceptron, where $\mathbf{w} \in \mathbb{R}^{n+1}$ is the weight vector, $\mathbf{x} \in \mathbb{R}^{n+1}$ is the input vector and $g(\cdot)$ is the activation function whose primary function is to introduce non-linearities to the model but also bound the output. Multiple perceptrons can be stacked on top of each other forming a layer with multiple outputs. These layers can then be connected to each other horizontally such that the output from each layer acts as the input to the next layer of the network. For a multi-perceptron neural network with multiple layers and n inputs, the value of ith neuron in layer k is defined as:

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=0}^{n} g(z_{k-1,j}) w_{j,i}^{(k)}. \tag{2.8}$$

Here $g(z_{k-1,j})$ is the output from the jth neuron in the previous layer. The inputs always flow in one direction, there are no connections between inputs within a layer nor are there any backward connections. This ensures convergence of the algorithm during training. The network is trained iteratively where at each iteration a batch of data is fed to the network and the weights and thresholds of the network are updated such that the output error is minimized [24]. The output error is measured by comparing the output with the targets using a loss function:

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n} \sum_{i}^{n} \mathcal{L}(f(x^{(i)};\mathbf{W}), y^{(i)}). \tag{2.9}$$

The network learns by updating the weights and thresholds using a gradient descent algorithm which is an iterative optimization algorithm, that ensures convergence:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}. \tag{2.10}$$

2.2.1 Dilated convolution

Convolutional networks are a type of neural network architecture that has revolutionized image classification and is particularly useful when the input is spatial and there is an order between the various nodes in the input layer. It is also a more efficient architecture compared to fully connected layers on account of the former having substantially fewer trainable parameters [2]. In a fully connected network the outputs from the previous layer are connected to each node in the subsequent layer resulting in a large amount of trainable parameters: a $32 \times 32 \times 3$ image connected to a fully connected layer with 1024 neurons has 3 145 728 trainable parameters while a convolutional layer with similar number of neurons has only 75 trainable parameters[2].

The parameter reduction in a convolutional layer is achieved by replacing a fully connected network with local connections only, where a kernel, also called filter, is applied to the input data by means of a convolutional operation. For a univariate time series $y \in \mathbb{R}^T$ with T entries and a filter $f \in \mathbb{R}^K$ with size K the convolutional operation is defined as

$$(y * f)[t] = \sum_{\tau=0}^{K-1} y[t-\tau] f[\tau]. \tag{2.11}$$

By associating the weights with the kernel a reduction in the number of parameters is achieved. A convolutional layer with $5 \times 5 \times 5$ filters has only 125 trainable parameters regardless of the shape of the input data. One of the main selling points of CNN is that different filters learn to recognize different features present in the input: a certain filter can learn to recognize a nose while another one can learn to recognize something much simpler such as a line[2].

A modified version of a convolutional model is the dilated causal convolution model which has shown great promise in data with long-range temporal dependencies such as retail forecasting [6], generating raw audio [26], weather prediction [38] as well as action segmentation and detection [22]. The model is said to be causal if the prediction $p(y_{t+1}|y_1, y_2, ...y_t)$ generated at time t only depends on data points prior to time t, which ensures that the temporal order of the data is not violated. Causality is obtained by asymmetrically padding the beginning of the input vector. Dilation is an efficient way of increasing the receptive field of the model, allowing it to learn about correlations between data points that are far away. Intuitively the dilation operation can be thought of as augmenting the kernel with zeros; when the kernel is convoluted over the input vector it equates to periodically skipping input values with a certain step size, also called dilation rate, d. For a kernel f with size K and input vector y the result of a dilated convolution[39][26] is defined as

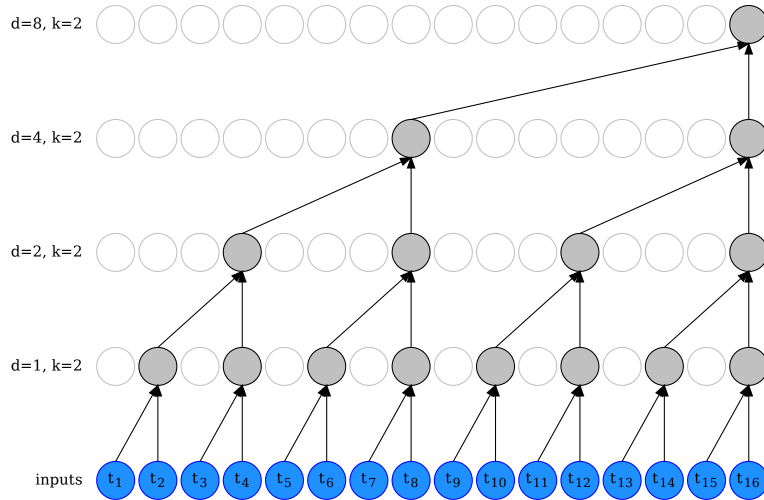


Figure 2.2: Stacked dilated Convolutional layers for an input sequence of 16 and a kernel size of 2. At each layer the dilation is increased with a power of 2.

$$(y *_d f)[t] = \sum_{\tau=0}^{K-1} y[t - d \cdot \tau] f[\tau]. \tag{2.12}$$

The $*_d$ operator is referred to as the d-dilated convolution. Dilated convolutions can be stacked with an exponentially growing dilation value, increasing the dilation factor d results in a increased receptive field $r = K2^{L-1}$, here L is number of stacked layers, for figure 2.2 the receptive field is therefore 16.

2.2.2 Recurrent Neural Networks

Recurrent neural networks are a type of network that include feedback connections[24]. The aim with these types of networks is to be able to handle inputs of varying sizes and capture dependencies within an input sequence [31]. The manner in which this is done is via a hidden state neuron that keeps track of what has happened in the past and the network can for each output node include this hidden state vector. This means that the same input can produce different outputs if previous inputs in the sequence are different. For example if the current input is “eat” then the output can be “mice” or “fish” depending on if the previous sequence of inputs was “A lot of cats” or if the sequence was “A lot of sharks”.

The general architecture of an RNN is similar to the multi-layer perceptron introduced in section 2.2 with the addition of a feedback connection in the form of a hidden node. This indicates that data can be fed laterally to the network as well as forwardly, unlike a perceptron where data is only fed forwardly. These lateral connections allow the network to maintain a memory state, enabling it to remember the past. Figure 2.3 shows the general structure of an RNN consisting of one input neuron, one hidden neuron and one output neuron. This recurrent network has been

unfolded in time on the right side of figure 2.3, which increases the number of input and output neurons from 1 to T and the number of hidden neurons from 1 to $T+1$.

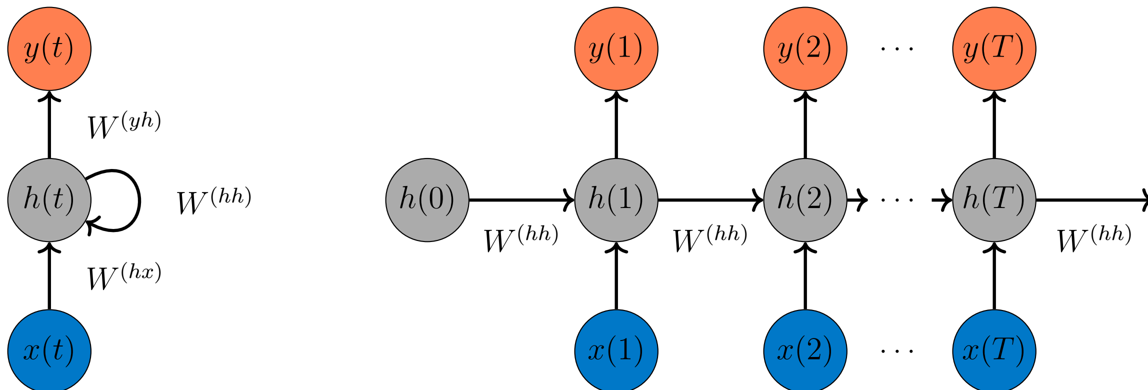


Figure 2.3: To the left: a simple recurrent neural network consisting of one input neuron, one hidden neuron and one output neuron. To the right: the same RNN unrolled in time. The unrolling does not affect the weights, they remain the same for each time step.

The output y_t is derived using equation 2.13

$$\begin{aligned} h_t &= g\left(W^{hx}x_t + W^{hh}h_{t-1}\right) \\ y_t &= W^{yh}h_t \end{aligned} \tag{2.13}$$

where $h_t \in \mathbb{R}$ is the value of the hidden neuron, x_t is the value of the input neuron and $g(\cdot)$ is the activation function.

The unrolling in time of a recurrent network, as seen in figure 2.3, is what enables one to train an RNN by slightly modifying the backpropagation algorithm [24]. This allows one to transform the cyclic graphs associated with RNNs to acyclic ones, allowing one to train the network in a similar fashion as to the training procedure described in section 2.2. The drawback of unrolling a network is that each time step has a neuron that is associated with it which can result in very big networks [24].

One of the main problems with RNNs is the vanishing gradient problem. This is the problem of having an ever decreasing gradient as it is updated for each hidden node by means of backpropagation. This causes the network to forget what it has learned about early inputs. One way of alleviating this is through truncated backpropagation through time. This process involves having a cut-off so that the network only backpropagates for a set time $t - T$ instead of having a full backpropagation. The downside with this approach is that long-term correlations are more difficult or downright impossible to learn [24].

2.2.3 Long Short-Term Memory Networks(LSTMs)

Long Short-Term Memory was created as another solution to the vanishing gradient problem present in RNN models [16]. LSTMs are able to reduce the vanishing gradient problem without affecting the possibility to learn long-term correlations, unlike truncated backpropagation through time which does affect this. This is accomplished through the introduction of memory modules which are modified perceptrons with the added complexity of multiple gating mechanisms and a cell state c_t. Similarly to an RNN a hidden state h_t is included as well. The main difference between the hidden state h_t and cell state c_t is that the hidden state is designed to act on short term dependencies while the cell state is used to catch long term dependencies.

The first gating mechanism in the memory module is in the form of a forget gate which is designed to determine how much of the previous cell state c_{t-1} should be discarded. The output from the forget gate is a vector of values between 0 and 1 which are multiplied by the value of the cell state. A value of 0 discards everything from the cell state while a value of 1 keeps it completely intact. The forget gate is defined as $f_t(x_t, h_{t-1}) = \sigma(\mathbf{W}_f x_t + \mathbf{U}_f h_{t-1} + b_f)$ where $x_t \in \mathbb{R}^d$ is the input to the module at time step t and $h_{t-1} \in \mathbb{R}^h$ denotes the previous hidden state. $\mathbf{W}_f \in \mathbb{R}^{h \times d}$ and $\mathbf{U}_f \in \mathbb{R}^{h \times h}$ are the weights associated with the forget gate, superscript h is the number of LSTM units and superscript d denotes the number of features in the input data. The sigmoid activation function ensures that the forget gate is constrained to the interval $[0, 1]$.

The second gating mechanism, the input gate, has the same structure as the forget gate. It acts on the input data at time step t in addition to the previous hidden state h_{t-1} and is defined as $i_t(x_t, h_{t-1}) = \sigma(\mathbf{W}_i x_t + \mathbf{U}_i h_{t-1} + b_i)$. The role of the input gate is to decide what new information is going be added to the cell state. This is done by combining the input gate with a hyperbolic tangent function that outputs a list of candidate values: $\tilde{c} = \tanh(\mathbf{W}_c x_t + \mathbf{U}_c h_{t-1} + b_c)$.

The result from the two previous gates allows us to now update the cell state $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$.

The final gating mechanism, the output gate $o_t \in \mathbb{R}^h$, is defined similarly to the previous two gates with the exception that a hyperbolic tangent function is used as an activation function instead of a sigmoid function $o_t(x_t, h_{t-1}) = \tanh(\mathbf{W}_o x_t + \mathbf{U}_o h_{t-1} + b_o)$. This allows us to finally decide what the output from the memory module should be by multiplying the output from the output gate with the updated cell state, $h_t = o_t \circ \tanh(c_t)$.

In summary, the outputs c_t and $h_t \in \mathbb{R}^h$ from the LSTM module are defined as:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{2.14}$$

$$h_t = o_t \circ \tanh(c_t). \tag{2.15}$$

2.2.4 Sequence to sequence models

There are various types of sequence modeling problems in neural networks, the difference being in the dimensions of the inputs given to the model and outputs generated by the model. The simplest example is having a one dimensional input and output such as image classification. Another, more complex, example is having an input sequence that produces a single output such as models that summarize movie reviews and produce a score for the movie in question. A final, and most important example in relation to this thesis, is having an input and an output both consisting of sequences. This last type of problem is handled by sequence to sequence models.

Sequence to sequence models are neural network based architectures that consist of an encoder that converts a given input sequence to a vector and a decoder that takes the output from the encoder and converts it into another sequence[31] that corresponds to the desired output. By splitting up the processing of an input sequence into two parts, an encoder and a decoder, one is given the possibility to utilize separate networks for the encoding and decoding. Because of the sequential nature of both the inputs and outputs the network architecture of both the encoder and decoder are usually of a recurrent type: one can use RNNs, LSTMs, gated recurrent units (GRUs) or a combination of these.

The primary purpose of the encoder is to take a sequence of inputs and create a context vector consisting of information about the hidden states of the network. This context vector is supplied by the last hidden state of the encoder which is then used as an input to the decoder. The hidden states in the decoder are initialized with the help of the context vector supplied by the encoder. The decoder can then predict what the next output should be in the output sequence.

2.2.5 Self-attention

One major drawback with sequence to sequence models is that they rely heavily on the context vector supplied by the encoder which is in itself limited to learning short-term dependencies. This is remedied by the introduction of self-attention [34][4]. An attention function can be described as the mapping of a query and key-value pairs to outputs and performs calculations of weighted averages. The mapping is derived by relating different positions in a sequence to one another and producing a computed representation of the sequence [34]. The attention mechanism allows a sequence to sequence model to, for each time step, take in an input from a sequence, encode it, and to incorporate previous inputs by computing weighted averages of them. The decoder is then fed the encoded input and the weights from the attention mechanism, allowing it to learn long-term correlations.

Equation 2.16 shows the general layout of the attention function where queries, keys and values have been grouped into the matrices $\mathbf{Q}$, $\mathbf{K} \in \mathbb{R}^{n \times d_k}$ and $\mathbf{V} \in \mathbb{R}^{n \times d_v}$, respectively, where n is the length of the sequence. The dimensions of one query vector and one key vector are denoted by d_k while the dimension of one value vector is denoted by d_v. $A(\mathbf{Q}, \mathbf{K})$ is a normalization function and one often chooses the

scaled dot-product attention defined as $A(\mathbf{Q}, \mathbf{K}) = \text{Softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)$.

$$\text{Attention}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = A(\mathbf{Q}, \mathbf{K})\mathbf{V} = \text{Softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.16}$$

Instead of performing a single attention function $A(\mathbf{Q}, \mathbf{K})$ with queries, keys and values of dimension d one can project these three components h times with different projections onto the dimensions d_k, d_k and d_v, respectively. The attention functions are subsequently computed in parallel for each of these projections, resulting in d_v-dimensional outputs. All projected versions of the output are concatenated into one output and projected once again, as seen in equation 2.17.

$$\text{MultiHead}\left(\mathbf{Q}, \mathbf{K}, \mathbf{V}\right) = \text{Concat}\left(\mathbf{head}_1, \ldots, \mathbf{head}_h\right)\mathbf{W}^O \tag{2.17}$$

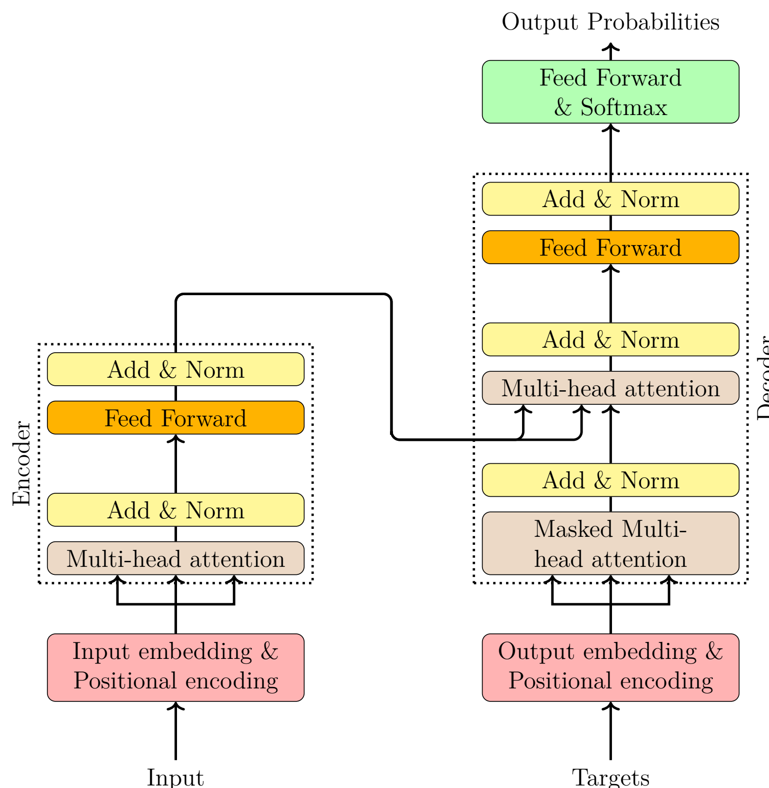
where $\mathbf{head}_i = \text{Attention}\left(\mathbf{QW}_i^Q, \mathbf{KW}_i^K, \mathbf{VW}_i^V\right)$ and $\mathbf{W}_i^Q \in \mathbb{R}^{d \times d_k}$, $\mathbf{W}_i^K \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_i^V \in \mathbb{R}^{d \times d_v}$ denote the projected parameter matrices.

The benefit of multi-head attention is that it allows a model to attend to different representaion spaces at different positions in the sequence, in parallel [34].

2.2.6 Transformers

Transformers were introduced as a remedy to one of the main problems with classical sequence to sequence models and attention-based models [34]. In both types of models one has the problem of not being able to run computational tasks in parallel due to the use of recurrent networks to process sequences. In order to produce a hidden vector h_t one must have fed in input x_t as well as the hidden vector from previous time step h_{t-1} which in turn depends on h_{t-2}. It is this iterative process that prevents the models to perform computations of of a certain sequence in parallel[34]. In transformers recurrent connections are omitted in both the encoder and decoder, requiring them to rely solely on attention mechanisms to produce connections between inputs and outputs [34].

Figure 2.4 shows a high-level architecture of a transformer. The transformer consists of two modules, an encoder and a decoder, that can be stacked on top of each other to produce transformers with varying size. Both modules consist of multi-head attention layers and feed forward networks. Similarly to the encoder in a sequence to sequence model the encoder in a transformer produces hidden vectors that are used to initialize a specific part of the decoder, see the black arrows originating from the encoder in figure 2.4. The output from the decoder is fed to a linear layer with a softmax activation function producing a vector of probabilities. This vector represent what the next part of the sequence is most likely to be. The removal of recurrent networks removes the model's ability to keep track of what the position of an input is in the sequence. By using positional encodings for both the input to the encoder and the decoder one can resolve the inability to keep track of positions.



Is it your image?

Figure 2.4: High-level architecture of a transformer. The left part of the figure depicts the decoder structure consisting of multi-head attention and a feed forward network. To the right part we have the decoder structure which has one more, masked, attention layer than the encoder. Output probabilities are produced by applying a softmax activation function on the output of the decoder.

2.3 Quantile Loss

Traditional loss functions such as *Mean Squared Loss Error* (MSLE) and *Mean Absolute Error* (MAE) are suitable loss functions for point forecasting models since they predict the conditional mean $\mathbb{E}(\boldsymbol{y}_{t+1:t+\tau}|\boldsymbol{y}_{1:t})$. In the case of multi-quantile forecasting, however, multiple values are produced that correspond to a specific quantile for each horizon step [36]. The output therefore corresponds to the full conditional distribution $P(\boldsymbol{y}_{t+1:t+\tau}|\boldsymbol{y}_{1:t})$, resulting in traditional loss functions such as MSLE and MAE being inadequate. This inadequacy is primarily because each output must learn to approximate the conditional quantile $\mathbb{P}(y_{t+\pi} \leq y^{(q)}_{t+\pi}|y_1, y_2, ..., y_{t+\pi-1}) = q$, which cannot be guaranteed when utilizing MSLE and MAE. Alternative loss functions such as quantile regression loss need to therefore be used.

Quantile regression aims to penalize underpredictions, $\hat{y} < y$, and overpredictions, $\hat{y} > y$, disproportionately depending on the quantile. For small quantiles overpredictions should be penalized whilst the opposite holds true for large quantiles. For a specific quantile $q \in [0, 1]$ the loss is calculated as

$$QL(y_t, \hat{y}_t^{(q)}, q) = q \cdot \max\left(0, y_t - \hat{y}_t^{(q)}\right) + (1-q) \cdot \max\left(0, \hat{y}_t^{(q)} - y_t\right). \tag{2.18}$$

The total loss is then calculated as a summation over the set of all quantiles $\mathcal{Q}$ and horizons τ [23][36]:

$$\mathcal{L}(\Omega, \mathbf{W}) = \frac{1}{M} \sum_{y_\pi \in \Omega} \sum_{q \in \mathcal{Q}} \sum_{\pi=t}^{t+\tau} QL(y_\pi, \hat{y}_\pi^q, q) \tag{2.19}$$

Here Ω is the set of all training samples consisting of M samples, $\mathbf{W}$ is the set of model specific trainable parameters, τ is number of forecasting horizons and $y_\pi, \hat{y}_\pi^{(q)} \in \mathbb{R}$ are the ground truth and predictions respectively for quantile q at time π. For $q = 0.5$ the quantile loss is simply the mean absolute error.

2.4 Metrics

In order to judge the performance of different models during training and testing one needs to use performance metrics. Metrics are similar to loss functions with the exception that they do not need to be differentiable. One of the most basic metrics is Mean Absolute Error (MAE) which can be interpreted as the arithmetic average of the absolute errors. MAE is a scale-dependent metric and can therefore not be used to make comparisons between time series with different units[17]. The MAE of a prediction $\hat{y}_\pi$ and target y_π is defined as

$$MAE = \frac{1}{M \cdot \tau} \sum_{y, \hat{y} \in \Omega} \sum_{\pi=t}^{t+\tau} |y_\pi - \hat{y}_\pi|. \tag{2.20}$$

A possible approach to making MAE scale-independent is to use a normalization factor which is what is used when calculating a Mean Average Percentage Error (MAPE). MAPE is a scale-independent MAE based metric for time series forecasting but is unfortunately not suitable for the task at hand on the grounds that the metric is undefined for zero-valued observations which are included in our data set. Mean Squared Logarithmic Error (MSLE) [32] is an alternative metric that can handle zero-valued targets and involves the logarithmic quotient between the target and forecast value. For non negative data, division by zero is avoided by augmenting the numerator as well as the denominator by one:

$$MSLE = \frac{1}{M \cdot \tau} \sum_{y, \hat{y} \in \Omega} \sum_{\pi=t}^{t+\tau} \ln \left[\frac{y_\pi + 1}{\hat{y}_\pi + 1} \right]^2 \tag{2.21}$$

A similar metric to MSLE is *Mean Average Arctangent Percentage Error* (MAAPE)[20] that was proposed to solve the problem of division by zero as well as other shortcomings of MAPE, such as the problem of very big values when the ground truth

is close to zero: $MAPE \xrightarrow[y \to 0]{} \infty$. MAPE also places heavier penalties on positive values than for negative values meaning that it is not symmetrical. Alternative variants of MAPE exist that present a solution to the asymmetrical property of MAPE such as Symmetrical Mean Percentage Error (SMAPE), unfortunately the problem of division by zero still persist. MAAPE on the other hand has an additional benefit of being bounded to the range $[0, \frac{\pi}{2}]$, since for large values $tan^{-1}(x) \xrightarrow[x \to \infty]{} \frac{\pi}{2}$. By multiplying the metric with the fraction $\frac{200}{\pi}$ the bounded range can be transformed to $[0, 100]$.

$$MAPE = \frac{100}{M \cdot \tau} \sum_{y \in \Omega}^{M} \sum_{\pi = t}^{t+\tau} \Big| \frac{y_\pi - \hat{y}_\pi}{\hat{y}_\pi} \Big| \tag{2.22}$$

$$MAAPE = \frac{2 \cdot 100}{\pi \cdot M \cdot \tau} \sum_{y, \hat{y} \in \Omega}^{M} \sum_{\pi = t}^{t+\tau} tan^{-1} \Big(\Big| \frac{y_\pi - \hat{y}_\pi}{y_\pi} \Big| \Big) \tag{2.23}$$

The metrics defined so far are designed with a point forecasting model in mind, thus they only take into consideration one quantile, the 50th, and not the 10th and the 90th quantiles. Normalized quantile loss takes into consideration all quantile outputs and can therefore be used as a performance metric that measures the performance of all three quantiles.

$$\frac{\sum_{y, \hat{y} \in \Omega} \sum_{q \in \mathcal{Q}} \sum_{\pi = t}^{t+\tau} QL(y_\pi, \hat{y}_\pi^q, q)}{\sum_{y, \hat{y} \in \Omega} \sum_{\pi = t}^{t+\tau} |y_\pi|} \tag{2.24}$$

3

Methods

A successful time series engine depends heavily on finding an appropriate model and fitting it to the underlying system that is driving the time series. In this chapter we will describe the two main models used in our report as well as the two base models that are used as reference models. We will, however, start by describing the data set and the various filtering and transformation operations that were applied to the data set.

3.1 The Data set

The data set consists of retail sales from multiple stores and was initially unprocessed and incomplete in the sense of missing dates. The missing dates were a result of either the stores being closed on that particular day or because nothing was sold. The proposed models, which will be defined in sections 3.2 and 3.3, require that the data set has no missing data for any dates in the time interval of interest. The data set was therefore padded with zeros for all missing dates. A zero value in the processed data set indicates therefore that there were no sales on that particular date.

As is evident from figure 3.1 many items are sold very sparsely, the majority of them covering less than 40% of their date range. This implies that in a given week the majority of articles are sold for only $7 \cdot 0.4 = 2.8$ days out of the seven days in a week. A decision was therefore made to filter out products that had too few sales during the relevant time period, with the motivation that filtering out items that are sparsely sold would improve the predictive capabilities of the models. If one does not remove these sparsely sold articles then the models would most likely start predicting zero sales too often, even for products that are sold frequently.

The filtration was accomplished by first deriving the percentage of days that have missing data using the ratio $d_{fraction} = \frac{\text{\# of missing dates}}{\text{date range}}$. Date range denotes the temporal span in which data has been recorded, i.e. the number of days the article has existed in a store. Simply filtering out items by this ratio can be misleading since a substantial amount of articles in the data set have a low date range and few missing dates - see figure 3.2 - which will go undetected by the filter. An article that has existed for 3 days and been sold for all these days would have a 100% date range coverage and would therefore go undetected. Hence, prior to applying the filter, all

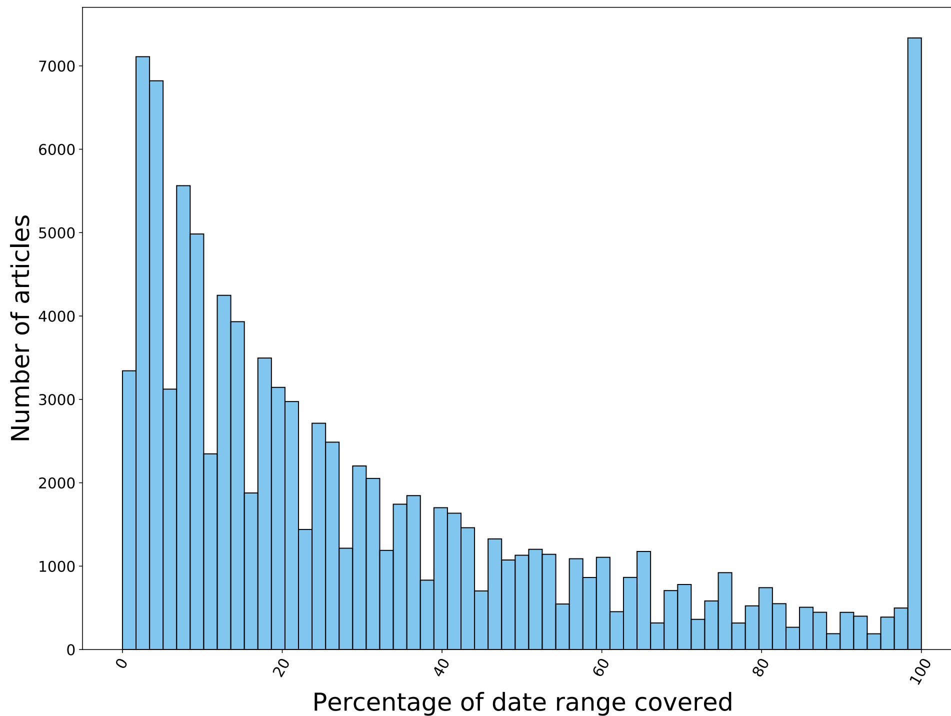


Figure 3.1: Frequency distribution of the percentage of date range covered by all articles in our data set.

items with a date range less than a specified threshold are removed.

The thresholds for the filtering process were decided upon by repeatedly applying the filter described in the paragraph above for different date range and date fraction values and observing the number of articles that were filtered away. Figure 3.3 shows a heatmap of the results from this process and a region of interest has also been added to the same plot. This region of interest shows value combinations that would result in a reasonable amount of items being left untouched by the process. The filtering process can be summarized as:

1. Remove all items with a date range less than d_{range}.
2. Remove all items with a date fraction less than $d_{fraction}$
3. Padd missing dates with zeros

After the filtering process the data was augmented with additional features, of which all can be categorized as categorical or time dependent, future known, variables. An example of one of the columns that were added is a Boolean column indicating whether a specific day is a holiday or not.

A sliding window was subsequently applied over the data set that transformed each

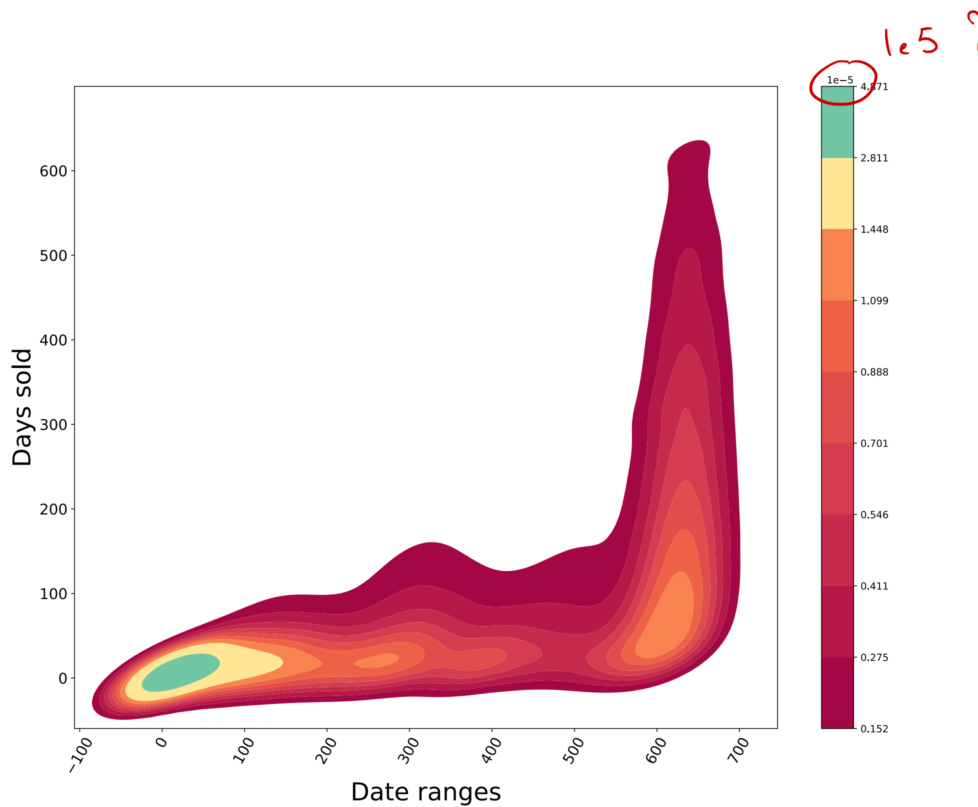


Figure 3.2: A heatmap of the density of points where each point represents one article. The x value denotes the article's date range and the y value denotes the number of days that article has been sold. There is a large concentration of articles around the origin.

time series from a 2-dimensional array to a 3-dimensional tensor where each window is a consecutive time series with a fixed size. The size was set to 97 time steps, where the first 90 time steps corresponded to historical data and the last 7 time steps corresponded to the forecasting horizon. In order to exemplify how a sliding window functions lets assume that the model requires the input to be a 32×12 array and that the time series that we want to feed into the model is of shape 45×12. The first dimension represents number of time steps and the second dimension represents number of features. Before being able to feed the time series into the model we need to apply a sliding window so that it is transformed into a $14 \times 32 \times 12$ tensor, where the new dimension is number of windows. In essence the same data is fed to the network multiple times albeit time shifted. A striding mechanism is also incorporated into the sliding window, resulting in a fewer number of sliding windows.

The data set contains a great deal of variability and different scales. Figure 3.4 shows the existence of some outliers in the data set, especially for the days Monday, Wednesday and Sunday. The amount of outliers is substantially greater when analyzing the daily sales data, for example 100 items of one article might be sold every day while only 1 item a week might be sold for another obscure article. The

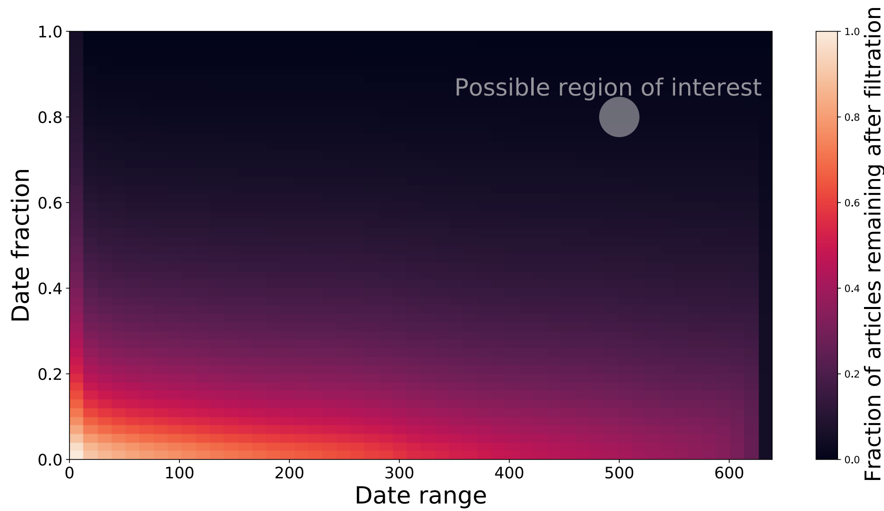


Figure 3.3: Heatmap of the results produced by repeatedly applying the filter date range function for various date ranges and date fractions. A possible region of interest is also displayed in the heat map. The color bar denotes the fraction of articles remaining after applying the filter function.

presence of outliers will affect the performance of the models negatively. Often times outliers are simply removed but that is not possible in a time series forecasting task that requires that all time steps are present. The alternative is to replace them with a more adequate value by means of applying filters such as a hampel filter. We decided, however, to not apply any filters with the with the justification that the outliers are not the result of errors in the data gathering process but rather a side effect of the various scales that exist in the data set, as seen in figure 3.5. The figure shows the number of articles sold per day for each store and implies that the data set is far from balanced.

The final preprocessing step entailed standardizing the data by either using z-score normalization or robust scaling. Z-score normalization is sensitive to the presence of outliers in the data set since they affect the calculation of the empirical mean $\bar{x}$ and standard deviation σ. These derived values would undoubtedly skew the standardization process negatively. Robust scalers, on the other hand, are not affected by outliers to the same degree as z-score normalization.

$$\textbf{Robust} = \frac{x - median(x)}{\text{Inter quantile range}}, \qquad \textbf{z-score} = \frac{x - \bar{x}}{\sigma}.$$

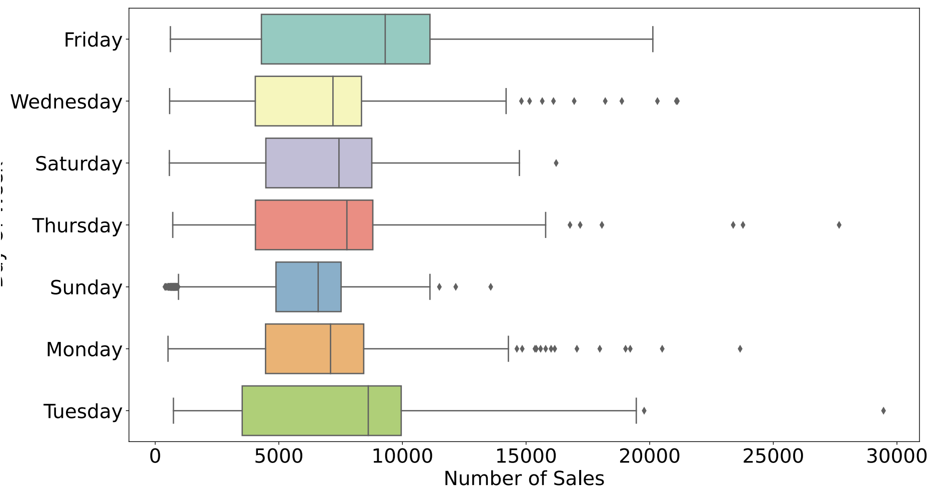


Figure 3.4: A box and whisker plot over daily sales for the entire data set grouped by day of week.

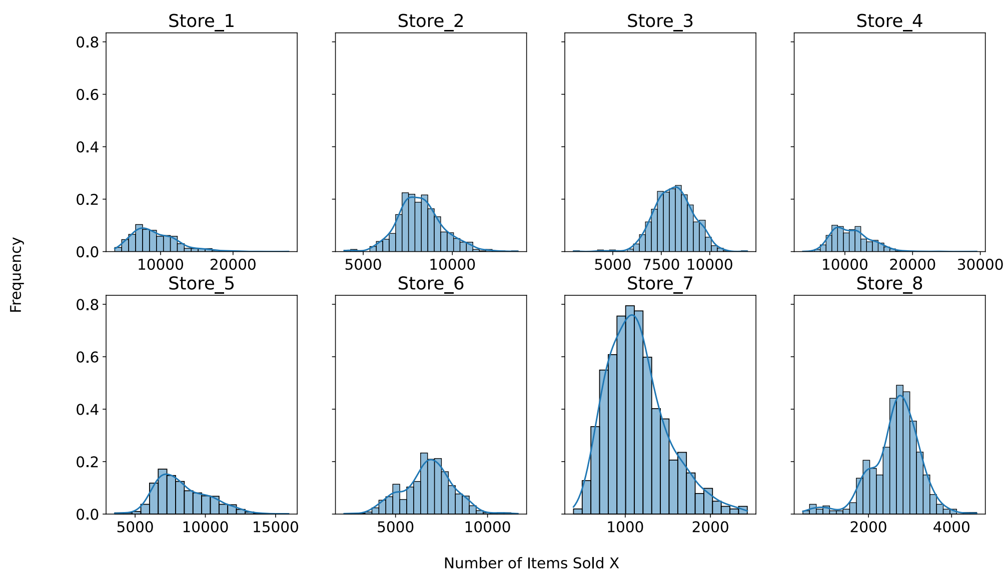


Figure 3.5: Each of the eight graphs represent the distribution of the number of articles sold every day for a specific store.

3.1.1 Data Splitting

The data set was split into three sets, one for training, one for validation and one for testing. The splitting was done on an article by article basis where each article was split in such a way that all data points except for the last $4 \cdot$ (forecasting horizon) were placed in the training set. This method maximizes the amount of data in the training set by creating overlaps between the three sets without running into the problem of using a label twice. The overlap is therefore only between the historic

data for each window, implying that the data that is used as ground truth in the validation and test sets does not overlap. Algorithm 1 gives a detailed description of how the data was split.

Algorithm 1 Splitting data

let Θ be the set of all articles
let F be the size of the forecasting horizon & L the size of the look back window
for θ in Θ **do**
  The training set θ_{train} is all datapoints between $0 \rightarrow len(\theta) - 4 * F$
  The validation set $\theta_{val}$is defined as $len(\theta_{train}) - L \rightarrow len(\theta_{train}) + 2F)$
  The test set θ_{test} is all points between $len(\theta_{train}) - L + 2F \rightarrow len(\theta_{train}) + 2F$
end for

A secondary approach that was used extensively, prior to switching to the above data splitting method is to use hard boundaries without any overlaps. Thus, the training set would be all data in the set $[0 : len(\theta) - 2(L + F)]$, whilst the validation set would be $[len(\theta) - 2(L + F) : len(\theta) - (L + F)]$ and the test set would constitute the remaining data points. This method results in fewer windows, in particular for the validation data set which will only have one window per article. This was the main reason for switching data splitting method, since it increased the number of windows per article, especially for the validation set.

3.2 Deep temporal convolutional network (DeepTCN)

Deep Temporal Convolutional Network is a multi-step horizon forecasting model which generates forecasts based on both observed historical inputs as well as known future inputs. It consists of an encoder-decoder architecture with a dilated convolutional network as an encoder and two fully connected layers functioning as a decoder. The encoder acts on historical inputs, while the decoder acts solely on the future known data. The output from the two modules are subsequently combined and fed to a final fully connected layer of size $s = h \cdot Q$ where h is the forecasting horizon and Q is the number of quantiles. DeepTCN is a small and shallow network with very few trainable parameters and layers, relying mainly on the vast receptive field of dilated convolutional layers for accurate forecasts. The encoder consists of 5-7 stacked blocks, denoted as residual blocks, but more layers can be added as long as the receptive field is smaller than the size of the temporal dimension. A residual block consists of two dilated convolutional layers with accompanying batch normalization layers and activation functions, see figure 3.6. There is also a residual connection where an alternative path for the data exists such that the non-linear processing can be skipped if needed. An *Exponential Linear Unit* (ELU) is utilized as an activation function[8].

The decoder in the original DeepTCN architecture consisted of a residual block called resnet-v followed by a feed forward layer with the purpose of mapping the output from the resnet-v block to the desired multi-horizon output. The output from the encoder was therefore used to initialize the decoder. The module was designed

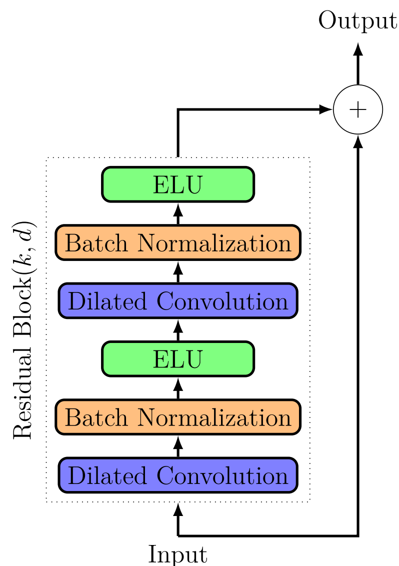


Figure 3.6: A Diagram of a residual block with two dilated convolutions. d and k refer to dilation rate and kernel size respectively

to act on two types of inputs: future-known variables and the historical information that is generated by the encoder module [6]. Resnet-v consists of two feed forward layers with an intermediary batch normalization layers as well as a Rectified Linear Unit (ReLU) activation function. There is also a batch normalization layer after the second feed forward layer. The above part of the model acts solely on the future known variables which is later aggregated with the output from the encoder that is then fed to the final output layer [6].

Our implementation differs substantially from the architecture presented by the authors. The primary difference is the inclusion of a variable selection network (VSN), see section 3.3.2 for a description of VSN. As a consequence, in our modified version of DeepTCN, the historical data is first fed to the VSN prior to feeding it to the encoder module, see figure 3.7. The expectation with this addition is that the VSN module will weigh the various features based on their significance for the forecasting task at hand. Other, smaller, modifications include the incorporation of two convolutional layers with kernel size $k = 1$ and number of filters $f = 1$ and placing the last activation before the residual connection.

Each feature in the input tensor was encoded using either Tensorflows embedding layer in the case of categorical features or a dense layer for numerical features. The input data was therefore transformed from a 3-dimensional tensor with dimensions $B \times T \times K$ to a 4-dimensional tensor $B \times T \times E \times K$, where B is the batch size. T is number of time steps, K is number of features in the input data and E is the embedding size. Transforming the input to a 4D tensor is necessary when using VSN, as is explained in section 3.3.2.

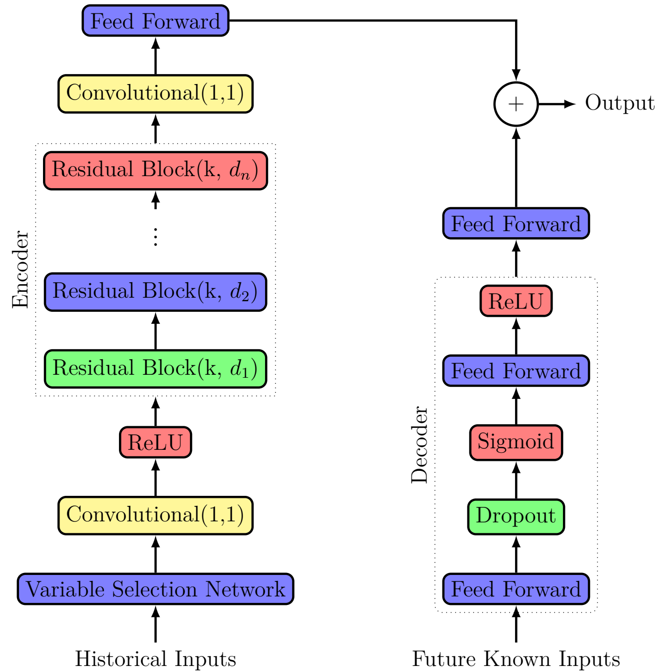


Figure 3.7: A Diagram over the complete DeepTCN architecture, with multiple stacked residual blocks and a decoder module. Furthermore a VSN module has been added at the onset.

3.3 Temporal fusion transformer (TFT)

Temporal fusion transformer is a complex time series forecasting model made up of a range of different neural network based modules[23]. It is capable of handling different types of data such as static data, time dependent future known data and time dependent future unknown data. TFT is able to produce multi-horizon quantile forecasts on top of point-forecasts and also has the ability to capture both short-distance temporal correlations as well as long-distance temporal correlations by using sequence to sequence modules and transformer based attention modules. Gating mechanisms are used to control what type of information is important in different modules and whether to apply non-linear processing.

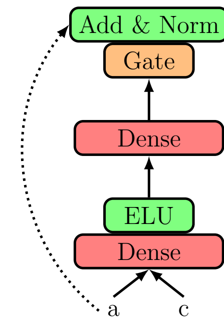
The different modules used in TFT are gating mechanisms in the form of gated linear units (GLUs) or in the form of gated residual networks (GRNs), variable selection networks (VSNs), static covariate encoders, sequence to sequence modules with LSTM encoders and decoders and a temporal fusion decoder. Each respective module will be described in detail in the following sections.

3.3.1 Gated Residual Network (GRN)

A gated residual network consists of two feed forward layers and a gating mechanism in the form of a GLU, see figure 3.3.1. A GLU is the Hadamard-product between a sigmoid function of the input γ and the input itself, see equation 3.2 for a mathematical description. $\mathbf{W}_{1,\omega}$ and $\mathbf{W}_{1,\omega}$ are the weights for the GLU layer that act on

the input a.

$$\mathrm{GRN}_\omega(\mathbf{a}, \mathbf{c}) = \mathrm{LayerNorm}(\mathbf{a} + \mathrm{GLU}_\omega(\eta_1)) \tag{3.1}$$
$$\mathrm{GLU}_\omega(a) = \sigma\left(\mathbf{W}_{1,\omega}\mathbf{a} + \mathbf{b}_{1,\omega}\right) \odot \left(\mathbf{W}_{2,\omega}\mathbf{a} + \mathbf{b}_{2,\omega}\right) \tag{3.2}$$



why not 3.8 ?

In figure 3.3.1 we observe that time dependent input a and static input c are fed into the first feed forward layer. The output from the first layer is given as input to the second feed forward layer after an exponential linear unit (ELU) has first been applied. The output from this second feed forward layer is finally fed into the gating mechanism of the GRN in the form of a GLU which decides what to output. In essence the purpose of a GRN is to allow the network to decide whether or not to apply non-linear processing. If the value outputted by the GLU is close to zero then we have essentially skipped the non-linear processing of the input and it is simply given as a normal input to the next layer.

3.3.2 Variable Selection Network

It is often unknown beforehand the relevance of the various features that are fed to the network. In most real-world application one can expect the data to contain insignificant variables in relation to the forecasting task. TFT employs a variable selection network (VSN) which is applied to both static and time dependent features. The idea is that the VSN should be able to improve the models predictive capabilities as well as filter out noisy data. See figure 3.8 for an overview of the architecture of a VSN. Each input variable is transformed into a (d_{model})-dimensional vector, i.e. $\boldsymbol{\xi}_t^{(i)} = f(x_t^{(i)}) \in \mathbb{R}^{(d_{model})}$, where x_t^i is the ith variable at time step t, using either entity embeddings for categorical variables or linear transformation for continuous variables. In practical terms this amounts to transforming the input data into a 4-dimensional tensor consisting of the dimensions (Batch $\times$ Time $\times$ Embedding $\times$ Features). For each feature we feed the 3-dimensional vector (Batch $\times$ Time $\times$ Embedding) into a GRN where a TimeDistributed layer is used to ensure that the same weights are applied at each time step.

In addition to the transformed input variables $\boldsymbol{\xi}_t^{(i)}$ that are fed to one GRN per variable we flatten them all and feed them to a separate GRN layer with an accompanying softmax activation function. The flattened vector is denoted as $\boldsymbol{\Xi}_t = \left[\boldsymbol{\xi}_t^{(1)}, \boldsymbol{\xi}_t^{(2)}, \boldsymbol{\xi}_t^{(3)}, ..., \boldsymbol{\xi}_t^{(m_x)}\right]^T$ in figure 3.8. These two operations result in two new tensors, $\boldsymbol{v}_{\chi_t}$ and $\tilde{\boldsymbol{\xi}}_t^{(i)}$, as seen in equation 3.3.

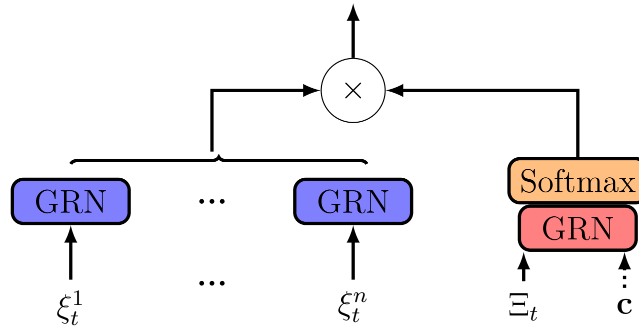


Figure 3.8: Architecture of a VSN model.

$$\begin{aligned} \boldsymbol{v}_{\chi_t} &= \text{softmax}\Big(\text{GRN}_{\chi_t}(\boldsymbol{\Xi}, \mathbf{c_s})\Big) \\ \tilde{\boldsymbol{\xi}}_t^{(i)} &= \text{GRN}_i\Big(\boldsymbol{\xi}_t^{(i)}\Big) \end{aligned} \tag{3.3}$$

There is also the possibility of feeding a context vector $\mathbf{c_s}$ to the model-wise GRN layer. The context vector $\mathbf{c_s}$ is one of several context vectors generated by a static covariate encoder and they are all used in various parts of TFT to better integrate static meta data into the model. $\boldsymbol{v}_{\chi_t} \in \mathbb{R}^{(m_x)}$ is the output from the model-wise GRN module where the last layer in GRN has a layer size equal to number of non-static input variables. In the end the two components are multiplied together producing the output

$$\tilde{\boldsymbol{\xi}}_t = \sum_{j=1}^{m_x} = v_{\chi_t}^{(j)} \tilde{\boldsymbol{\xi}}_t^j. \tag{3.4}$$

3.3.3 Static covariate enconders

As mentioned in the previous section TFT makes use of enconders to produce four different context vectors for static covariates that are then fed into the network at various points. A static covariate encoder takes the output of a VSN and applies a GRN to it. This produces the desired context vectors which are then used throughout TFT to better incorporate static metadata into the model, see figure 3.9 for a schematic look of where the static vectors are used.

3.3.4 Sequence to sequence module

After having decided what inputs are important by using variable selection networks we feed the outputs $\tilde{\boldsymbol{\xi}}_t$ into a sequence to sequence module consisting of LSTM encoders and decoders which process the input to find local correlations in the time series data. $\tilde{\boldsymbol{\xi}}_{t-k:t}$ are fed to the encoder while $\tilde{\boldsymbol{\xi}}_{t+1:t+\tau}$ are fed to the decoder. The sequence to sequence module produces a set of uniform temporal outputs $\boldsymbol{\phi}(t,n) \in \{\boldsymbol{\phi}(t,-k), \ldots, \boldsymbol{\phi}(t,\tau)\}$ where n is a position index. Similarly to other modules in

TFT we let static meta-data influence the output in this module by initializing the cell state c_t and the hidden state h_t in the LSTM modules with two context vectors $\boldsymbol{c}_c$ and $\boldsymbol{c}_h$.

A gated skip connection is the final processing step in the sequence to sequence module: $\tilde{\boldsymbol{\phi}}(t,n) = \tilde{\boldsymbol{\xi}}_t + \text{LayerNorm}\left(\text{GLU}_{\tilde{\phi}}(\boldsymbol{\phi}(t,n))\right)$. These temporal outputs are then given as an input to the temporal fusion decoder.

3.3.5 Temporal Fusion Decoder

Temporal fusion decoder is one of the more important modules in TFT that has the task of capturing long-term relationships within a time series sequence, see figure 3.9 for a schematic look of what is included in a temporal fusion decoder.

A static enrichment layer in the form of a GRN network is used to enhance the temporal data $\tilde{\boldsymbol{\phi}}(t,n)$ outputted by the sequence to sequence module. This is done by applying the GRN over the temporal data as well as a context vector $\boldsymbol{c}_e$ generated by the static covariate encoder: $\boldsymbol{\theta}(t,n) = \text{GRN}_\theta(\tilde{\boldsymbol{\phi}}(t,n), \boldsymbol{c}_e)$. The static-enriched temporal data is then combined and fed into a multi-head self-attention module to capture long-term dependencies for each time step. The result from the multi-head self-attention module is $\boldsymbol{\delta}(t,n) = \text{LayerNorm}(\boldsymbol{\theta}(t,n) + \text{GLU}_\delta(\boldsymbol{\beta}(t,n)))$.

A second GRN is used to give the possibility of further non-linear processing of the output from the self-attention module. A gated residual connection is also applied so that one can skip the static enrichment layer as well as the multi-head self-attention module if the model believes that all that non-linear processing is not needed to model the data at hand.

Finally we output prediction intervals in terms of quantiles by applying a feed forward layer to the output of the temporal fusion decoder. The size of the feed forward layer is therefore the number of quantiles times the number of time steps to forecast.

3.3.6 Modifications

TFT was modified in various ways to suit the needs of the forecasting task at hand. The first modification made is related to the static covariate encoders. By making some slight modifications to the code that defines a static covariate encoder we were able to add the flexibility of being able to train on different types of data sets: those that contain static data as well as those that do not contain it.

A second modification that we made is connected to the multi-head attention mechanism. Long-term correlations within a time series are captured by this attention mechanism and is similar to the one described in section 2.2.5 with the added feature of interpretability. In our thesis we have removed the ability to make interpretable predictions, as mentioned in section 1.2. The rest of the multihead self-attention mechanism is, however, used as is to better capture long-term correlations.

The third modification made is connected to Tensorflow. The source code provided

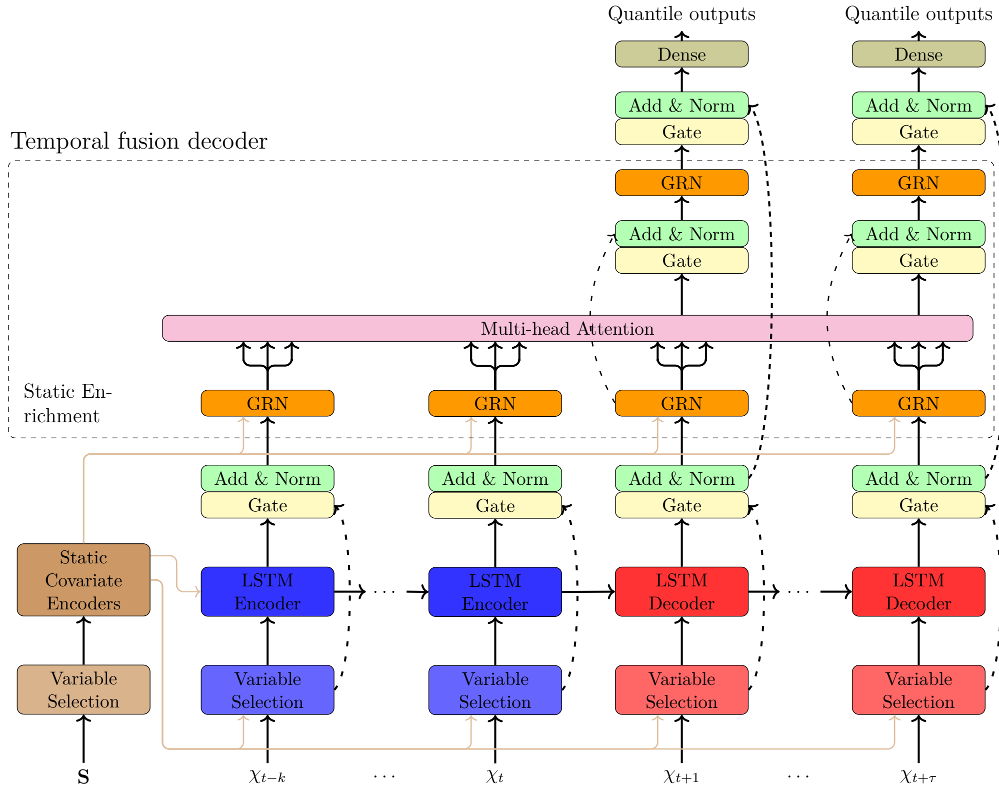


Figure 3.9: High-level architecture of TFT. **S** denotes static input, $\{\chi_i\}_{i=t-k}^{t}$ denotes historical temporal data and $\{\chi_i\}_{i=t+1}^{t+\tau}$ denotes future temporal data. Skip connections are indicated by dashed lines. Context vectors produced by the static covariate encoder are fed to various modules, including VSNs, Encoders and GRNs.

in the original paper was written in Tensorflow v.1 but we modified it so that Tensorflow v.2 can be used instead. The main differences between the versions are that one does not need to start sessions or handle Tensorflow graphs since they are handled by Tensorflow itself in version 2.

The final modification that was made on TFT was to change the manner in which it handles data during training. In the original TFT everything is stored in memory but in order to be able to train on larger data sets we had to modify the data pipeline so that TFT uses data generators instead.

4

Results

Both models were trained on a data set that contained eight stores and 835 articles after having applied the various filtering and preprocessing operations described in 3.1. The total number of consecutive time series amounted to 1 037 320 after having applied the sliding window function.

Several variants of our two models were examined, among them two DeepTCN versions and four TFT versions. For the DeepTCN model the kernel size as well as the dilation rates were modified whilst the remaining parameters were fixed. Two kernel sizes were used, $k = 2$ and $k = 3$. For $k = 2$ the dilation rates were set to $\left[1, 2, 4, 8, 16, 20, 32\right]$ while for $k = 3$ the dilation rates were set to $\left[1, 2, 4, 8, 16, 20, 30\right]$. The maximum dilation rate was restricted to 30 since the dilation factor times the kernel cannot exceed the length of the input sequence. Furthermore, an embedding size of 10 and a batch size of 64 were employed.

For the TFT model we experimented with various hidden layer size values, holding all other hyperparameters fixed. Four different hidden layer sizes were examined: $h = 8$, 16, 32 and 64. A batch size of 128 and a learning rate $\mu = 10^{-3}$ were used. An early stoppage criterion was also employed that would terminate the training process after 10 epochs if the validation loss did not improve during these 10 epochs.

As a baseline model a modified naive approach was utilized. The naive approach is to use the last observation as a forecast for the next time step, $y_{t+1} = y_t$, whilst our approach utilizes the average value of the last three observations: $y_{t+1} = \frac{y_t + y_{t-1} + y_{t-2}}{3}$. The approach is consequently used recursively to achieve a multi-horizon forecast. An SES model with $\alpha = 0.1$ was also employed as an additional baseline model.

In table 4.1 we have presented the results of all variants of our models. The results from this table indicate that all variants of our two models performed better than the baseline models according to the metrics MAE, MSLE and MAAPE. The fourth metric, normalized quantile, cannot be calculated for the two baseline models since they only produce point forecasts and not quantile forecasts like our neural network based models.

The DeepTCN model with $k = 2$ performed slightly better than all TFT variants except for the TFT version with $h = 64$ where the performance seems to be compatible between the two models. Increasing the hidden layer size for TFT presented a slight

Model	Normalized Quantile	MAE	MSLE	MAAPE
Baseline	–	5.2290	0.5920	36.3208
SES ($\alpha = 0.1$)	–	3.9817	0.4083	33.4432
TFT (h = 8)	0.3829	3.4936	0.3461	31.0104
TFT (h = 16)	0.3761	3.4117	0.3328	30.1734
TFT (h = 32)	0.3803	3.4480	0.3322	30.3295
TFT (h = 64)	0.3761	3.4040	0.3327	30.2640
TCN (K=2)	0.3551	3.2495	0.3433	30.6572
TCN (K=3)	0.3729	3.4240	0.3762	32.0394

Table 4.1: The following table shows the performance of the models on four chosen metrics (Normalized Quantile Loss, MAE, MSLE and MAAPE). The performance of the models where measured on the test data. All models presented in this table used the data splitting method described in algorithm 1 in section 3.1.1.

performance improvement while the opposite effect is observed when increasing the kernel size for DeepTCN.

The difference in metric values for the TFT variants with $h = 64$, $h = 32$ and $h = 16$ are arguably negligible since most of the difference is in the second or third decimal place. The largest jump in performance occurs between $h = 8$ and $h = 16$, where $h = 16$ is clearly better at making predictions than $h = 8$. The TFT version with $h = 64$ is better at generating forecasts than all other variants according to the metrics MAE, MAAPE and normalized quantile. The difference between the second best TFT variant, $h = 16$, and the best one is, however, minimal.

The number of training epochs for both DeepTCN models were set to 50 but as is evident from figure 4.1, the training process terminated earlier than that as a result of the early stoppage criteria. The stoppage criteria is that the training process will terminate if the validation loss does not improve in ten epochs. One can also observe considerable variations in the validation loss for the $k = 3$ DeepTCN model, the same variations are not present in the $k = 2$ model.

Figure 4.2 depicts loss curves for the four TFT variants examined in our thesis. All versions terminated early, with the earliest termination occurring for the most shallow network with $h = 8$ at the 14th epoch. The variant that trained the longest time was the network with $h = 32$ which trained for 29 epochs. One can also observe that the more shallow the model is the higher the initial training loss is. The deeper the network is the lower the final loss is, except for $h = 64$ which has a higher final loss than both $h = 16$ and $h = 32$. The validation loss varies a great deal more than the training loss and the most shallow networks seem to overfit already after 8-10 epochs meanwhile the deeper models start overfitting at around 12-15 epochs.

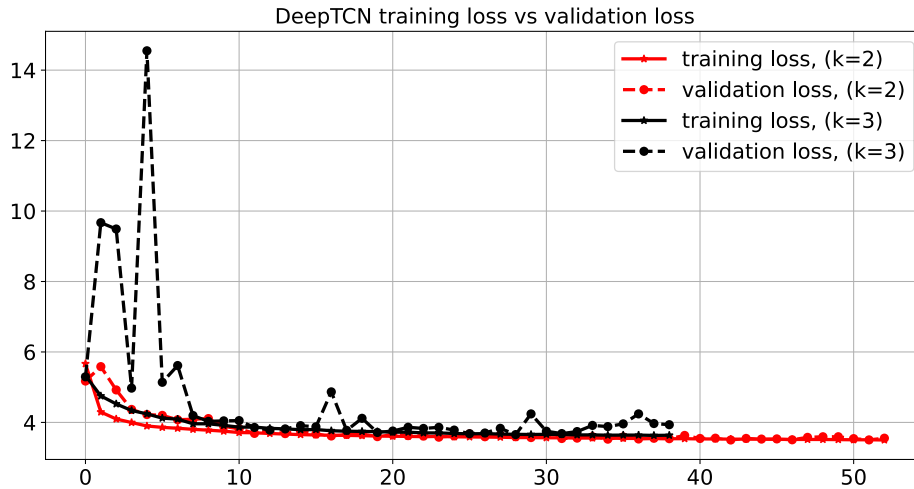


Figure 4.1: The training and validation loss for both DeepTCN models

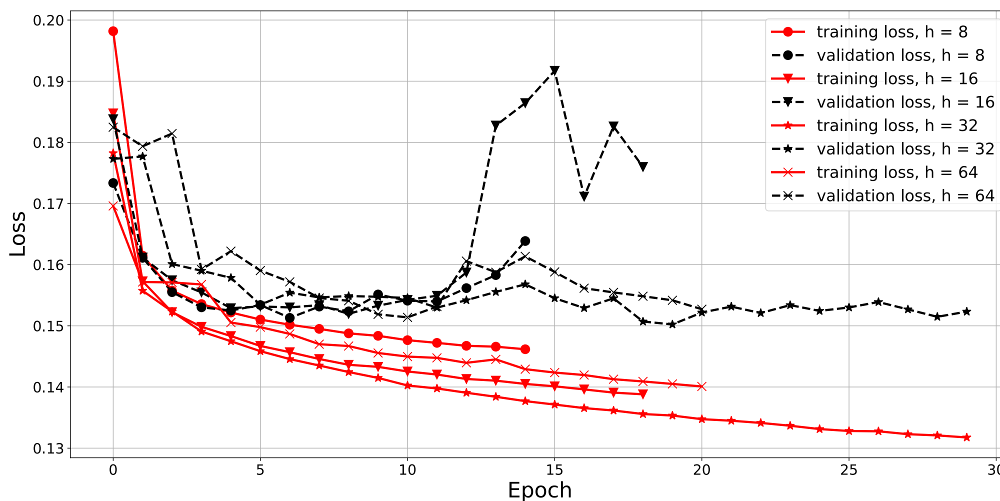


Figure 4.2: Loss curves for four different TFT networks with $h = 8$, $h = 16$, $h = 32$ and $h = 64$. Red lines indicate training loss and black lines depict validation loss.

Figure 4.3 shows forecasts made on three articles using the SES baseline model. One can observe that the output from this models is constant for the whole forecasting horizon. This baseline model performed somewhat well according to the results in table 4.1.

Figures 4.4 and 4.5 depict forecasts made by the best performing DeepTCN model with $k = 2$ and the best performing TFT model with $h = 64$. The forecasts have been made on the same stores and articles for both models and the differences between the forecasts are minimal according to these two figures. Both models seem to be good at predicting articles that have small variations while articles with larger variations proves to be more difficult to forecast. The 90th and 10th quantiles seem to capture most of the variations though. One final thing to note is that the forecast

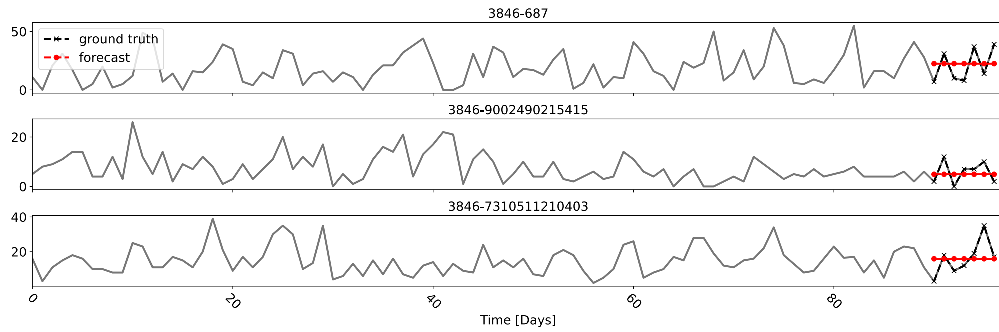


Figure 4.3: Forecasts made using the SES baseline model

intervals produced by DeepTCN seem to be narrower than the ones produced by TFT.

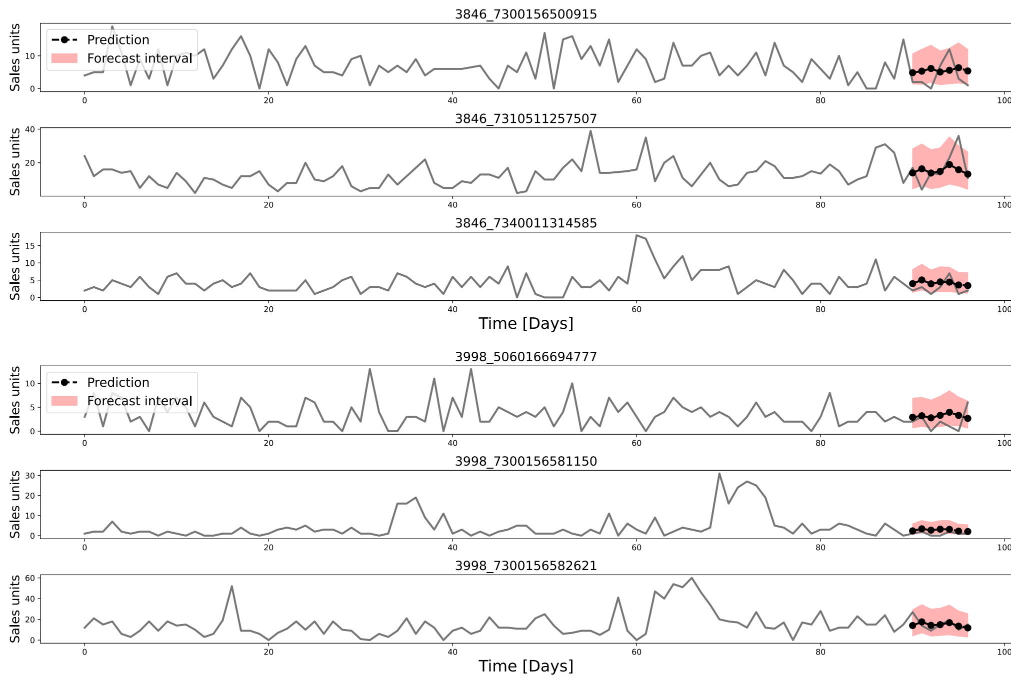


Figure 4.4: Time series forecasting done by TFT with $h = 32$ on six random articles from two stores, store 173 and store 3998. See figure 4.5 for a detailed description of the layout.

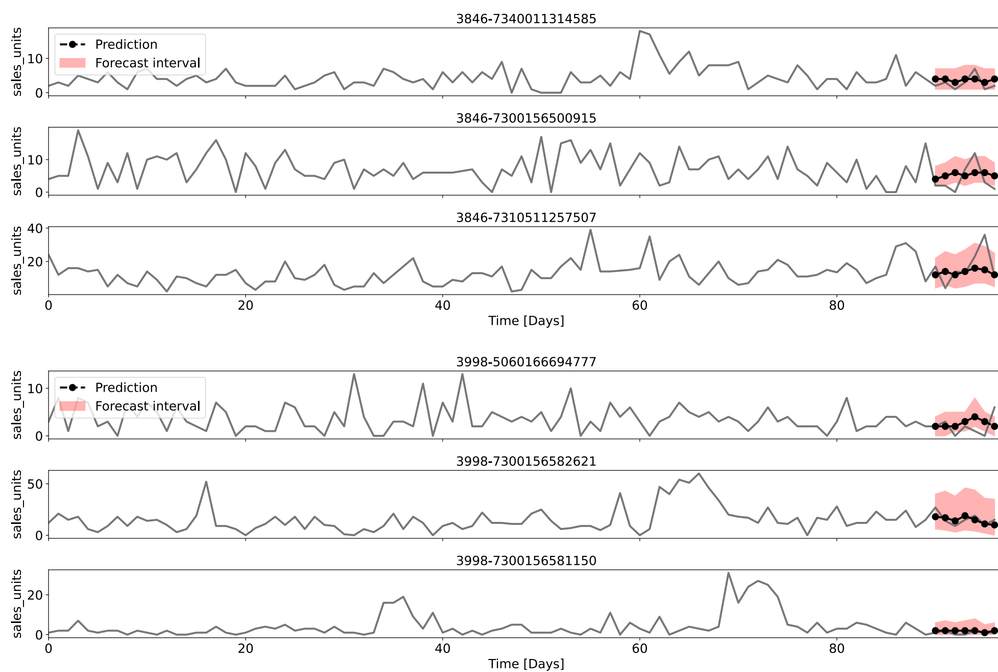


Figure 4.5: Forecasting done by DeepTCN with k=2 for a some random samples in the test data. The transparent red area defines the models quantile forecasts - the upper boundary is the 90th quantile while the lower boundary is 10th quantile. The grey line represents the historical data that the model has access too. The data points marked by black dots are the actual predictions.

5 Discussion

In this chapter we will discuss various technical challenges that we faced, the results that we obtained and how the predictive capabilities of our models can be improved by suggesting different modifications to both models. In the end a conclusion will be provided about the results obtained as well as our expectations.

5.1 Challenges

There were numerous technical challenges that we faced and although some satisfactory solutions were found they took up ample time that could have been spent on more important tasks that could result in better performing models. This is in particular more relevant in regards to TFT where an already existing repository written in Tensorflow 1.0 was modified to suit our needs. We had to migrate the code from Tensorflow 1.0 to Tensorflow 2 which proved to be more difficult than anticipated.

A second challenge was feeding the data to the models without running out of memory. We tried various methods but decided on a generator in the end. In Tensorflow there are two approaches to implementing a generator, one can either subclass the Sequence class and implement the requested methods, or one can define a generator function that yields two tensors corresponding to the input data and the targets. The second approach entails using Tensorflows tf.data.Dataset API for the creation of a data pipeline that takes the output from the generator and feeds it to the model. Both approaches require that the data is loaded in chunks, processed and transformed to a list of time series by means of a sliding window. After all of this has been done the data is batched and fed to the network. Although both methods were used to train networks with, a decision was made to discard the Sequence approach, since no satisfactory thread safe implementation could be found. An unexplained memory leak as well as a failure to properly shuffle the data were secondary motivations for discarding the Sequence approach. The secondary approach turned out to be more robust and allowed us to to efficiently create a data pipeline where we among other things could shuffle and cache the data to a file if required.

A third challenge that we had was the size of the data set used in training our models. The full data set consisted of 3.2GB of raw, processed, data from 242

stores but we had to use a subset of this data set, consisting of 8 stores. These 8 stores amounted to 120MB of raw data but after windowing the data the total memory needed for just the data amounted to 5.7GB which was the reason why generators were needed. We tried training on the larger data set but ultimately decided against it since training a TFT model with hidden layer size 64 for just one epoch took almost 13 hours.

A final challenge that we had, that was more persistent in TFT than in DeepTCN, were the long training times. Using the same virtual machine the second deepest TFT network with a hidden layer size of 32 took around 30 minutes per epoch to train while the largest DeepTCN network took around 10 minutes per epoch to train. The longer training times cannot be explained by the number of trainable parameters in the models since the difference is only 28% while the difference in training times is up to 300%. One explanation for the long training times in TFT is the fact that it has a sequence to sequence module to capture local temporal correlations. This module requires the sequence to be transformed to a windowed input. In [3] the authors have speed up TFT by a factor of three by modifying it so that certain modules can run in parallel, in effect vectorizing computations in TFT.

5.2 Future Work

Although both TFT and DeepTCN performed well we are confident that the forecasting abilities of both models can be improved upon. Improvements can be made by modifying the models themselves as well as the manner in which we preprocessed data. Two obvious improvements are to extend the data set with more features and increase the number of articles and stores present in the data set by including a larger subset of the full data set. The second improvement would, however, most likely require extending the depth of models to handle the extra data.

A third, possible, improvement to the manner in which we preprocessed data is to experiment with other encoding methods. Our dataset included various cyclical features such as "day of year" and "day of weak" that were treated as regular categorical features, and thus encoded using Tensorflow's Embedding layer. The usage of Embedding layers may impede the performance of the models and an alternative encoding approach could therefore be to transform the features utilizing the $sin(x)$ and $cos(x)$ functions. The usage of these functions allows the models to not only capture the cyclical nature of the features but also transforms them to a bounded range of $[-1, 1]$. The downside with this approach is that it entails using two vectors to represent one feature. A similar periodic function could also be utilized to encode non-categorical features with the objective of bounding the features instead. This process is, however, irreversible and can therefore not be used to bound the target.

A fourth improvement that can be made is the way we handled outliers in the data set. In section 3.1 we discussed the large presence of outliers in the data set and lack of efficient methods for dealing with them. The presence of outliers could in reality affect the performance of the models considerably, finding an adequate solution to the problem of outliers may therefore result in substantial performance

improvements. To that end we experimented with filtering the data by removing outliers but decided ultimately against it. See section 3.1 for why we came to this conclusion. We also experimented with a hampel filter which turned out to be very time consuming and ultimately did not give any indications that it improved the performance of the models and was therefore discarded. Further experimentation with various filters, including the hampel filter, is required before a final decision is made on whether to or not to completely discard them.

A fifth possible improvement is to change the normalization procedure for the data set. For the DeepTCN model robust scalers were used to normalize the data instead of regular z-score scalers, with the expectation that it would minimize the influence of the outliers. These expectations were fulfilled as the performance of different DeepTCN variants were improved by using robust scalers. Different variants of TFT were trained using robust scalers but, in contrast to DeepTCN, no performance improvements were observed and the decision was therefore made to use regular z-score scalers when transforming the data for TFT. Here, as was the case with filters, further experimentation is required before a definitive conclusion is agreed upon, but it is quite perplexing that the robust scaler improves the performance of DeepTCN whilst having minimal influence on the performance of TFT.

Our models are exclusively neural network based but an alternative approach is to incorporate classical linear models into the architecture, creating a hybrid model. This is not a new idea and has been examined extensively resulting in various proposals for different hybrid architectures [30]. The motivation for a hybrid model, consisting of an exponential smoothing module and a complex neural network module, is that the former is a linear module capable of capturing linear features such as seasonality, level and trend, whilst the later component is designed to capture non-linearities. Only using non-linear models could result in a failure to capture linear features [21]. It can therefore be interesting to examine how a hybrid model fares on our data set. Additional approaches worth examining could be to experiment with various other TCN based architectures such as SeriesNet[27] and M-TCN[35], as well as experiment with various decoders, at the moment the decoder module in DeepTCN is very simple and therefore can be improved upon. Similarly one could experiment with various encoder-decoder architectures in the sequence to sequence module in TFT. Instead of using the LSTM based encoders and decoders one could experiment with GRNs or other types of recurrent networks.

Finally, the temporal fusion decoder of TFT contains a multi-head attention module, that is able to produce interpretable outputs in the sense that one can view what which features are important when the model is generating predictions. This added interpretability allows one to examine what variables might be obsolete for certain prediction tasks. Due to time constraints it was decided to not include the interpretability of this attention module in our version of TFT. This also made the comparisons between TFT and DeepTCN more fair since DeepTCN does not have any interpretability incorporated in it. We think that by including intrepretability, detecting obsolete variables and removing these one might help the training of the network by removing features that are simply noise to the network.

5.3 Conclusion

Table 4.1 indicates that all our models performed well by having a better performance than both the naive baseline model as well as the SES model for all of our chosen metrics. Based on table 4.1 the DeepTCN variant with $k = 2$ performed the best by slightly outperforming the best TFT with $h = 64$ according to two of the four metrics (Normalized Quantile and MAE). TFT did, however, perform better than DeepTCN according to the other two metrics. This can be interpreted as TFT being slightly better in making point forecasts i.e. the 50th quantile as is evident by the performance measurement on MAAPE and MSLE, whilst DeepTCN is superior in generating more accurate probabilistic ranges. In the end the difference in performance is not that big between the two models, as can be seen in figures 4.5 and 4.4 where forecasts of different stores and products have been plotted. The probabilistic ranges seem to be somewhat similar in magnitude, with DeepTCN having slightly narrower probabilistic ranges.

In regards to our own expectations, we expected the models to perform slightly better than what the results have shown. We are unsure about why the models do not approximate as well as we had anticipated. The figures 4.4, and 4.5 all have large probability ranges, whilst we expected a smaller confidence interval. This may constitute an unreasonable expectation though, we are after all trying to forecast retail sales with 90% probability thus one should expect large uncertainty intervals if one wants to forecast with this high of a probability. The large uncertainty intervals make even more sense if one takes into consideration the presence of outliers that was discussed in 3.1. The point forecasts on the other hand do look reasonable and substantially better than the baseline and ES model which produces almost constant forecasts, as seen in figure 4.3.

The results presented in section 4 indicate that neural network based models are suitable for time series forecasting in retail sales. We strongly believe that the performance of the models presented in this thesis can be improved upon by addressing the issues outlined in sections 5.1 and 5.2.

Good job! Nice report!

Are all images your own?
Otherwise check copyrights.

Bibliography

[1] Ratnadip Adhikari and Ramesh K Agrawal. An introductory study on time series modeling and forecasting. *arXiv preprint arXiv:1302.6613*, 2013.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[3] AmpX-AI. Tft vectorization. `https://github.com/AmpX-AI/tft-speedup`, 2020.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] Carlos Capistrán, Christian Constandse, and Manuel Ramos-Francia. Multi-horizon inflation forecasts using disaggregated data. *Economic Modelling*, 27(3):666–677, 2010.

[6] Yitian Chen, Yanfei Kang, Yixiong Chen, and Zizhuo Wang. Probabilistic forecasting with temporal convolutional neural network. *Neurocomputing*, 399:491–501, 2020.

[7] Davide Chicco, Peter Sadowski, and Pierre Baldi. Deep autoencoder neural networks for gene ontology annotation predictions. In *Proceedings of the 5th ACM conference on bioinformatics, computational biology, and health informatics*, pages 533–540, 2014.

[8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[9] Pascal Courty and Hao Li. Timing of seasonal sales. *The Journal of Business*, 72(4):545–572, 1999.

[10] FAO. Global food losses and food waste – extent, causes and prevention, 2011.

[11] Everette S Gardner Jr. Exponential smoothing: The state of the art. *Journal of forecasting*, 4(1):1–28, 1985.

[12] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE workshop on automatic speech recognition and understanding*, pages 273–278. IEEE, 2013.

[15] Jenny Gustavsson, Christel Cederberg, Ulf Sonesson, Robert Van Otterdijk, and Alexandre Meybeck. Global food losses and food waste, 2011.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] G Hyndman, R.J. & Athanasopoulos. Forecasting: principles and practice.

[18] R.J. Hyndman and G Athanasopoulos. Forecasting: principles and practice, 2021.

[19] IA Iwok and AS Okpe. A comparative study between univariate and multivariate linear stationary time series models. *American Journal of Mathematics and Statistics*, 6(5):203–212, 2016.

[20] Sungil Kim and Heeyoung Kim. A new metric of absolute percentage error for intermittent demand forecasts. *International Journal of Forecasting*, 32(3):669–679, 2016.

[21] Kin Keung Lai, Lean Yu, Shouyang Wang, and Wei Huang. Hybridizing exponential smoothing and neural network for financial time series predication. In *International Conference on Computational Science*, pages 493–500. Springer, 2006.

[22] Colin Lea, Michael D. Flynn, René Vidal, Austin Reiter, and Gregory D. Hager. Temporal convolutional networks for action segmentation and detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1003–1012, 2017.

[23] Bryan Lim, Sercan O Arik, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *arXiv preprint arXiv:1912.09363*, 2019.

[24] Bernhard Mehlig. Artificial neural networks. *arXiv preprint arXiv:1901.05639*, 2019.

[25] Mohsen Mohammadi, Faraz Talebpour, Esmaeil Safaee, Noradin Ghadimi, and Oveis Abedinia. Small-scale building load forecast based on hybrid forecast engine. *Neural Processing Letters*, 48(1):329–351, 2018.

[26] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan,

Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[27] K Papadopoulos. Seriesnet: a dilated causal convolutional neural network for forecasting. In *Proceedings of the International Conference on Pattern Recognition and Machine Intelligence, Union, NJ, USA*, pages 1–4, 2018.

[28] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[29] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

[30] Slawek Smyl. A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, 36(1):75–85, 2020.

[31] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[32] Chris Tofallis. A better measure of relative prediction accuracy for model selection and model estimation. *Journal of the Operational Research Society*, 66(8):1352–1362, 2015.

[33] Mohammad Valipour, Mohammad Ebrahim Banihabib, and Seyyed Mahmood Reza Behbahani. Parameters estimate of autoregressive moving average and autoregressive integrated moving average models and compare their ability for inflow forecasting. *J Math Stat*, 8(3):330–338, 2012.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[35] Renzhuo Wan, Shuping Mei, Jun Wang, Min Liu, and Fan Yang. Multivariate temporal convolutional network: A deep neural networks approach for multivariate time series forecasting. *Electronics*, 8(8):876, 2019.

[36] Ruofeng Wen, Kari Torkkola, Balakrishnan Narayanaswamy, and Dhruv Madeka. A multi-horizon quantile recurrent forecaster. *arXiv preprint arXiv:1711.11053*, 2017.

[37] Billy M Williams. Multivariate vehicular traffic flow prediction: evaluation of arimax modeling. *Transportation Research Record*, 1776(1):194–200, 2001.

[38] Jining Yan, Lin Mu, Lizhe Wang, Rajiv Ranjan, and Albert Y Zomaya. Temporal convolutional networks for the advance prediction of enso. *Scientific reports*, 10(1):1–15, 2020.

[39] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY