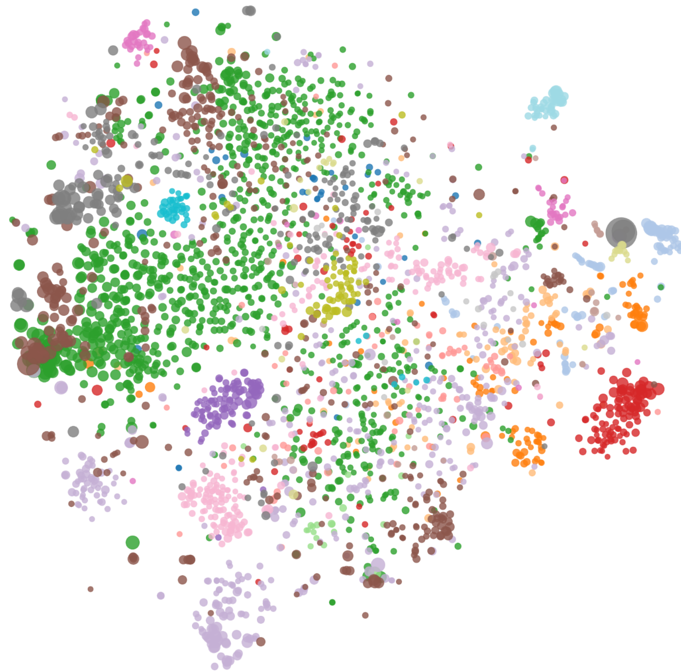
CHALMERS
UNIVERSITY OF TECHNOLOGY



Food Waste Reduction through Sales Forecasting using Temporal Fusion Transformers

A deep machine learning approach for predicting grocery sales

Master's thesis in Computer Science – Algorithms, Languages and Logic & Complex Adaptive Systems

ALEXANDER GUNNARSSON & CHARLOTTE FRANC

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Master's thesis 2021

Food Waste Reduction through Sales Forecasting using Temporal Fusion Transformers

A deep machine learning approach for predicting grocery sales

ALEXANDER GUNNARSSON & CHARLOTTE FRANC

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021

Food Waste Reduction through Sales Forecasting using Temporal Fusion Transformers
A deep machine learning approach for predicting grocery sales
ALEXANDER GUNNARSSON & CHARLOTTE FRANC

© Alexander Gunnarsson, Charlotte Franc, 2021.

Supervisor: Lennart Svensson, Department of Electrical Engineering
Examiner: Lennart Svensson, Department of Electrical Engineering

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

Cover: Food product entity embeddings as learned by the Temporal Fusion Transformer.

Typeset in LaTeX
Gothenburg, Sweden 2021

Food Waste Reduction through Sales Forecasting using Temporal Fusion Transformers
A deep machine learning approach for predicting grocery sales
Alexander Gunnarsson, Charlotte Franc
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Food waste contributes significantly to the world's greenhouse gas emissions, and reducing it is an important part of tackling climate change. Sales forecasting is one way in which waste can be reduced, allowing grocery stores to place more accurate orders and to avoid surpluses. We implement a Temporal Fusion Transformer (TFT) for this purpose, a recent deep learning architecture based on the increasingly popular Transformer. The TFT can learn an entity embedding for each food product, but only supports a fixed number of entities. We extend the TFT to be able to learn embeddings for a variable number of products by processing the name of a product using subword segmentation. This enables it to be used in applications where the number of entities frequently changes, such as when new products are added.

Results show that the TFT can perform better than some well-known baseline models, and that its performance can be increased further by using our extension to let it process the name of a product. The embeddings learned for each product are shown to be meaningful, as products belonging to the same category receive similar embeddings even when the model is given no information about product category. We also show that the forecasts are interpretable. A strong weekly pattern is learned by the model and is reflected by the model's attention weights. The model is also capable of learning to attend to similar promotion periods in the past when predicting sales during a new promotion, and learns to ignore these days during the absence of promotions.

Keywords: Transformers, Time series forecasting, Entity embedding, Deep Learning Interpretability, Attention mechanisms.

Acknowledgements

First and foremost we would like to thank our supervisor Lennart Svensson. His academic expertise has helped make this thesis possible, and his swift responses have continuously guided us towards the right track.

We would also like to thank our main company advisor, who wishes not to be mentioned by name. It was his initial ideas that sparked this thesis, and he has provided us with the data required to make it possible. Thank you for your insights making us understand the data, and providing the computational resources required to process it.

Alexander Gunnarsson, Charlotte Franc, Gothenburg, June 2021

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
1.1 Main Contribution 2
1.2 Problem Formulation 2
1.3 Delimitations 3
1.4 Thesis Outline 3

2 Background **5**
2.1 Time Series 5
2.1.1 Training Time Series Models 6
2.2 Common Time Series Models 6
2.2.1 Autoregressive Model 7
2.2.2 Long Short-Term Memory 7
2.3 Transformer 8
2.4 Temporal Fusion Transformer 10
2.4.1 Input Types 11
2.4.2 Variable Selection Networks 12
2.4.3 Gated Residual Networks 13
2.4.4 Sequence-to-Sequence Layer 13
2.4.5 Temporal Fusion Decoder 13
2.4.6 Quantile Outputs 14
2.4.7 Hyperparameters 14
2.5 Entity Embeddings 15
2.6 Subword Segmentation 16
2.7 Interpretability 16
2.8 Time Embedding 17

3 Methods **19**
3.1 String Input Type Feature 19
3.2 Data Selection 20
3.3 Feature Selection 21
3.4 Hyperparameter Selection 23
3.5 Baseline Models 24
3.5.1 Autoregressive Model 24

3.5.2 LSTM Model . . . 25
3.6 Temporal Fusion Transformer . . . 25
3.6.1 TFT Baseline . . . 26
3.6.2 Entity Embeddings . . . 26
3.6.3 Product Name Feature . . . 26
3.6.4 Subword Segmentation . . . 26
3.6.5 Product Category Classification Loss . . . 27
3.6.6 Time Embedding . . . 28
3.7 Evaluation . . . 28

4 Results **31**
4.1 Baseline Models . . . 31
4.1.1 Autoregressive Model . . . 31
4.1.2 LSTM . . . 31
4.2 Temporal Fusion Transformer . . . 32
4.2.1 TFT Baseline . . . 32
4.2.2 Entity Embeddings . . . 33
4.2.3 Product Name Feature . . . 34
4.2.4 Subword Segmentation . . . 36
4.2.5 Product Category Classification Loss . . . 37
4.2.6 Time Embedding . . . 41
4.3 Interpretability . . . 41
4.3.1 Attention . . . 41
4.3.2 Variable Selection . . . 42

5 Discussion **49**
5.1 Forecasting Performance . . . 49
5.2 Entity Embeddings . . . 51
5.3 Time Embedding . . . 52
5.4 Interpretability . . . 52
5.5 Unseen Entities . . . 53
5.6 Alternative Solutions . . . 54
5.7 Sustainability Aspects . . . 55
5.8 Future Work . . . 55

6 Conclusion **57**

Bibliography **59**

A Time2Vec Code **I**

List of Figures

2.1 Sliding window illustration. By using 3 historical days (blue) to predict 2 future days (red), multiple windows of length 5 are obtained from a single time series. 6
2.2 Illustration of an LSTM cell. Cells are chained together to form a complete LSTM architecture. From [5]. CC BY-SA. 8
2.3 Self-attention for the word *it*. Stronger colors represent stronger weights. Note that *didn't* and *tired* are built from subwords, further explained in Section 2.6. From [1]. CC BY-NC-SA. 9
2.4 Illustration of the TFT model architecture. Historical, future, and static inputs are treated differently when entering the network that in the end produces quantile forecasts. From [10]. Reproduced with permission. 10
2.5 GRN and Variable Selection Network, two components of the TFT. From [10]. Reproduced with permission. 11
2.6 Illustration of how a time series prediction model such as the TFT takes different kinds of features as input. From [10]. Reproduced with permission. 12
2.7 Quantile output performed by the TFT for a sample product. The 10th and 90th quantile together form an 80% confidence interval, while the 50th quantile is the point prediction. 14

3.1 Adding support for string valued features to the TFT. The new embedding network consists of an that LSTM takes the variable-length product name as input and outputs a vector in $\mathbb{R}^{d_{model}}$ that the TFT can use. 19
3.2 LSTM architecture used as a baseline. The 28 historical days are processed sequentially, before outputting the 14 future days all at once. 25
3.3 Our architecture with product category classification loss added. The embedding layer enables processing of the variable-length product names using an LSTM, processing either characters or subwords and outputs a vector in $\mathbb{R}^{d_{model}}$. The new classifier uses this vector to predict which category a product belongs to. 27
3.4 Illustration of Time2Vec added to the TFT. All date-based features such as day of the week are replaced by a scalar date ID passed into the Time2Vec layer that learns a vector representation of time in $\mathbb{R}^{d_{model}}$. 28

4.1 Validation loss curves during training for the TFT baseline model with different state sizes. 33

4.2 Validation loss curves during training for the TFT using fixed-number entity embeddings. By being able to treat each product independently, larger models easily overfit. 34

4.3 t-SNE projection of the learned fixed-number entity embeddings using a state size of 64. The color of a point indicates which category a product belongs to, and the size indicates the average sales per day. The model was not given any information about product categories during training. 35

4.4 Validation loss curves during training for the TFT with the product name as input using character-level processing. 36

4.5 Validation loss curves during training for the TFT with the product name as input using subword-level processing. 37

4.6 t-SNE projection of the learned embeddings based on the product name. The left image is the result of character-level processing and the right uses subword-level processing. Note that the number of points are much fewer than in Figure 4.3, as products with the same name get the same embedding so one point per unique name is presented. The size of a point then indicates the average sales per day for all products with the given name. The model state size is 64 for both models, other state sizes produced similar looking results. 38

4.7 Test losses for all experiments with varying state size. Note that the y-axis does not start at zero. The TFT baseline and fixed-number entity embeddings experiment had similar performance, while using the product name either with character-level or subword-level processing increased the performance similarly. 38

4.8 Validation loss curves during training for the TFT with the product name as input, adding product category classification loss with varying λ. Only the quantile loss is presented for comparability, seeing how it is affected by varying λ giving different weights to the classification loss which is not shown here. The TFT baseline is included as a reference. The state size is 16 for all models. 39

4.9 t-SNE projection of the learned embeddings for the product category classification loss experiment with varying λ. The model with $\lambda = 0$ is given no information about product categories. With increasing values of lambda, products belonging to the same category start to form clusters. All models have a state size of 16 and use character-level processing of the product name. 40

4.10 Validation loss curves during training of the TFT using Time2Vec. It converges to the same level as the TFT baseline but after much longer time. Another model was trained with no date features, and this had a similar learning curve making us believe our Time2Vec implementation is incorrect. The state size for all models is 32. 41

4.11 Attention weights for a selection of products. The model demonstrates a clear weekly pattern by attending to the same weekday from earlier weeks when making predictions, while avoiding to attend to irregular weeks such as when there was a promotion. 42
4.12 Predicting fruit sales during a promotion period. Greater attention weights are placed on a same weekday during another similar promotion period. 43
4.13 Historical and future variable selection weights from the TFT baseline model with a state size of 32, averaged between January 18, 2021 and February 28, 2021 for a sample of products. For the historical features, the number of sales is the most important variable for making predictions as expected, followed by day of the week. Promotions also appears to be a valuable categorical feature. For future variable selection weights, day of the week is the most significant variable of the known future inputs, followed by day of the year. Promotions received the highest weight out of the categorical features. 44
4.14 Historical and future variable selection weights for the best performing model with a state size of 32 using character-level processing of the product name. Weights are averaged between January 18, 2021 and February 28, 2021 for the same sample of products as in Figure 4.13. Sales has the largest weight, followed by day of the week out of the historical feature inputs. For the known future inputs, day of the week has the largest real-valued feature weight and promotions has the largest categorical feature weight. 45
4.15 Variable selection weights of categorical and real-valued features for a sample product. The weights for the 28 historical days are concatenated with the 14 future days. This example time window is from the TFT baseline model with a state size of 32. 46
4.16 Historical categorical and real-valued variable selection weights during December for a Christmas-related and non-Christmas related product. Results are obtained from the TFT with character-level processing of the product name and a state size of 32. 47

List of Tables

4.1 Autoregressive baseline model mean absolute errors. 31
4.2 LSTM baseline model mean absolute errors. 32
4.3 Metrics for TFT baseline model. 33
4.4 Metrics for TFT using fixed-number entity embeddings. 34
4.5 Metrics for the TFT with the product name as input using character-level processing. 36
4.6 Metrics for the TFT with the product name as input using subword-level processing. 37
4.7 Metrics for product category classification loss experiment using varying values of λ. The state size is 16 and the minibatch size is 512. The loss values presented are only the quantile losses. 39

1 Introduction

In 2018, about 1.3 million tons of food waste was thrown away in Sweden, of which about 7.7%, 100,000 tons came from grocery stores [2]. This food waste contributes to up to 3 percent of Sweden's total greenhouse gas emissions [17], and is a challenge not only for Sweden but also the rest of the world. One way to reduce this waste is through sales forecasting, so grocery stores can avoid ordering an unnecessary excess food supply that later needs to be thrown away. Today, forecasting sales is still done manually to a large extent and The Swedish Environmental Protection Agency identifies increased automation as a key action to reduce food waste [19].

Sales forecasting can be described as a time series problem. A time series is a sequence of data points ordered by time. Time series prediction has many use cases, sales forecasting being one example, with many different existing models for performing forecasts based on historical data [13]. In terms of grocery sales, each food product has its own unique time series of sales associated with it.

Time series models, both purely statistical and machine learning based, come along with certain challenges. Some popular statistical methods include the autoregressive (AR) and the autoregressive integrated moving average (ARIMA) models. These models, although good baseline models, have been outperformed by machine learning techniques such the Long Short-Term Memory (LSTM) model that is commonly used in machine learning. Although the LSTM provides some support in retaining information from long sequences of data compared to the vanilla recurrent neural networks by prioritizing what information to keep in memory with filtration techniques, it still suffers from long recurrences in sequential data. This makes it harder to retain information from many time steps back. As with most machine learning models, the LSTM is also not inherently interpretable meaning one cannot understand how it arrived at its conclusion.

In this thesis, not only one time series will be fed to the model, but multiple products across different stores need to be forecasted, each with their own time series data. Thus, feeding multiple inputs to the model is desired. Providing the model with other features such as the day of the week or month and other product-related data could potentially aid the model in finding interesting sales patterns and improving the forecasts. Leveraging information from each of the time series, in addition to retaining long-term dependencies, poses a challenge to the previously mentioned models.

With the rise of deep learning, many new and more complex machine learning models

for time series forecasting have emerged [11]. One interesting model is the Temporal Fusion Transformer (TFT) [10]. It is a recent model based on the highly influential Transformer [23] but specialized for time series and with several architectural additions. In contrast to recurrent neural networks such as the LSTM, the TFT can learn longer term temporal dependencies by making use of attention to attend directly back in time at any time step of interest when making a prediction [23].

The attention mechanism also offers interpretability insights of what information the model attends to. Furthermore, the TFT has a component called variable selection network that also increases model interpretability. It acts as a kind of filter that places larger weights on the features that are more important during training, while removing features that do not contribute to the model's learning. We evaluate the forecasting performance of the TFT, using the attention and variable selection mechanisms to also evaluate how interpretable the results are.

1.1 Main Contribution

Another benefit of the TFT is that it not only takes time series as input, but it can also take an identifier for each entity. An entity in our case is a food product. This allows one TFT model to be trained on all entities at once. Each entity is mapped to a learned vector, known as an entity embedding [6], that the TFT uses to make entity-specific forecasts. However, the number of entities needs to be fixed beforehand. This is problematic because the number of different food products in a store constantly changes.

To overcome this issue, we modify the TFT to be able to take a variable number of entities as input, making it more flexible and widening its possible use cases. This is done by providing a separate character string as input to the TFT. In our case, this is the name of a food product. This string is fed into a recurrent neural network, producing a vector as output that also serves the purpose of an entity embedding. The generation of these entity embeddings makes it possible to have a variable number of entities that the model makes predictions for. This appears to be a new approach since we have not found any literature where this has been done before, thus, making it the main contribution of this thesis. We call this implementation "variable-number entity embeddings".

1.2 Problem Formulation

The ultimate goal of this project is to reduce food waste through sales forecasting, which is in line with The Swedish Environmental Protection Agency's goal of increased automation [19]. To accomplish this, we also have research goals to evaluate if the forecasting accuracy when using a TFT can be increased by the use of our proposed variable-number entity embeddings, and if the embeddings produced for each product are meaningful. Furthermore, we investigate seasonality aspects and how these can be interpreted through the attention mechanisms. Different methods and models are compared.

The data we are working with is historical food sales data, mainly the number of units sold per day per product, and if there was a promotion going on or not. The data is described more in detail in Section 3.2 and 3.3. The task of the models is to output as many accurate forecasts as possible for future sales numbers, in the form of point predictions with a confidence interval. The models are also evaluated on the quality of the embeddings learned for each food product and how interpretable the forecasts are. We have formulated the following research questions:

- How is the accuracy of the TFT affected by making use of entity embeddings? Can our variable-number entity embeddings work as a replacement for fixed-number entity embeddings in cases where new entities appear frequently?
- How well are similar entities placed next to each other in the learned embedding space, motivating the use of the embeddings for other tasks?
- How well does the TFT model learn from the input features and forecast the short-term and longer term dependencies in an interpretable way? Can the model learn periodic and non-periodic temporal patterns including seasonal aspects, such as sales promotions or day of the week, to make more accurate predictions on future sales?

1.3 Delimitations

The models we develop have not been evaluated against newly introduced products. Predicting the sales of new products is a harder task, as it is largely influenced by marketing campaigns and there is no direct historical data to rely on. However, once a new product has been established and stabilized, our model can be used to forecast them without changing the architecture as it supports a variable number of entities. Fine-tuning the model should however always be done as new data appears with time.

Another difficulty is forecasting products with few sales. With little data available, it becomes hard to detect patterns. As the focus of this thesis is to compare models, we decide to only include products with a minimum average of 20 sales per day, further discussed in Section 3.2. Additionally, we restrict the hyperparameter search, further described in Section 3.4.

1.4 Thesis Outline

Chapter 2 serves to educate the reader on topics referenced throughout this thesis, starting with a general description of time series data and existing models used for forecasting. It then introduces the Transformer architecture and how it is incorporated into the TFT, ending with a few more subjects related to the TFT and our modifications to it.

Next, Chapter 3 starts by describing our main modification performed to the TFT, allowing it to take a variable amount of entities as input. It then continues describing how we prepare our data before describing each experiment to be performed using

both existing functionality of the TFT, as well as using our modifications. Chapter 4 then largely follows the same pattern, presenting the results of each experiment explained in the previous chapter.

In Chapter 5 we then discuss the results obtained. The topics include forecasting performance, quality of entity embeddings and the interpretability of the forecasts. The chapter ends by again connecting the results to sustainability aspects regarding food waste before suggesting future work. Finally, Chapter 6 concludes this thesis with answering our research questions by summarizing the findings presented in prior chapters.

2
Background

This chapter describes major topics referenced throughout this report. First, general time series forecasting methods and techniques are defined, and then specific properties of the TFT are described. Finally, new approaches and additions to TFT are discussed.

2.1 Time Series

Time series data is a sequence of data points in chronological order. In our case, we are working with food item sales where each data point indicates the number of sales during one day, resulting in evenly spaced discrete time steps which is commonly the case for time series data. To forecast a time series is to predict one or more future data points given the historical data and possibly other additional external factors or features that might affect the time series, for example, the day of the week or the weather.

When making forecasts, a common assumption is that the current value to predict depends more on recent values preceding it. The weather on a given day correlates with the previous day, and the correlation becomes weaker further steps back. However, time series also commonly have seasonality aspects. The weather on a given day also correlates with the weather on the same day one year earlier, as there is a yearly seasonal pattern in weather data. Time series can also have increasing or decreasing trends, such as climate change affecting the weather over time. Knowing these aspect of a time series enables making forecasts into the future.

Food item sales also exhibit these seasonal patterns that can be exploited to make forecasts. Note that the term season refers to any repeating pattern, not just cultural seasons. Most products have a weekly seasonality, as people tend to shop less on Sundays for instance. Some products are more popular during certain days of the week. For example, candy sales could go up on weekends. Paydays may results in a monthly seasonality properties. There are also yearly seasonalities when some products are more popular during certain parts of the year. It is up to the model to learn these patterns and forecast future sales.

2.1.1 Training Time Series Models

Time series forecasting can be seen as a regression problem, using historical data to predict future unseen data. To get a model that performs well on real world future data, the training, validation and test sets are split up based on time [22]. The training set contains the earliest data, and the test set contains the latest data for each food product in our case. This means that all products exist in all dataset splits, which might not be the case in a typical regression problem.

Before training, a decision needs to be made on how far into the future to forecast. For instance if forecasting the weather, 7 days into the future might be a reasonable choice. If the training set is 100 days long, that leaves 93 days that can be used to make the forecast. However, this has two main issues. Firstly, it is unreasonable to assume that the weather this far back is a good predictor for the weather the coming week, and would result in unnecessary computations for something the model has to learn to ignore. Secondly, this leaves only one sample of 100 consecutive days for the model to train on. Instead, what is commonly done is to use sliding windows [22] and decide beforehand how many historical and future days should be used in each window. This is illustrated in Figure 2.1 where each way to slide the window becomes a training sample.

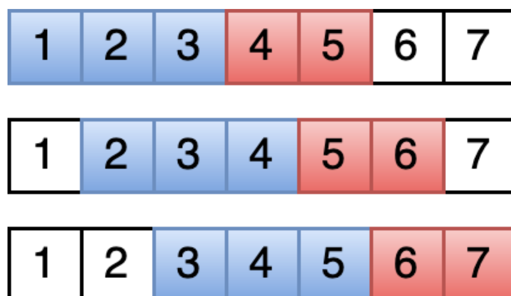


Figure 2.1: Sliding window illustration. By using 3 historical days (blue) to predict 2 future days (red), multiple windows of length 5 are obtained from a single time series.

The number of samples obtained from using sliding windows is calculated as *total days - window size + 1*. If the window size is equal to the total days, only one sample is obtained. Then for each decrease in window size, there is one additional way to slide the window. Note that *days* may be any other time unit for general time series.

In the weather example, 14 historical days might be used to forecast 7 days into the future, resulting in a window size of 21. This window slides over the training set consisting of 100 days. There are $100 - 21 + 1 = 80$ possible ways in which the window can be positioned, resulting in equally many training samples.

2.2 Common Time Series Models

There are many different kinds of time series models and they are continuously evolving to this day. One traditional time series model is called the autoregressive

(AR) model. Today there is a tendency to apply neural network models to make predictions, with one example being the Long Short-Term Memory (LSTM) model. The following two subsections describe the AR and LSTM models.

2.2.1 Autoregressive Model

The autoregressive (AR) model [24] is perhaps the most simple model that can be used to predict time series data. As the name indicates, it is "self regressive", inferring values based on previous values. It is defined by the equation

$$\hat{X}_t = b + \sum_{i=1}^{p} w_i X_{t-i} \tag{2.1}$$

where X_t is the value at time t, p is the order of the model, or the number of historical time steps to look at, w_i is the weight for the value i time steps back and b is a constant bias term. The weights and bias can be found using ordinary least squares.

Using a time series with daily time steps and an AR model of order $p = 7$, the last 7 days are used to make a prediction for the following day. This can then be applied recursively to predict the value arbitrarily far into the future. By first using days 1-7 to predict day 8, then day 9 can be predicted from day 2-8 using the value for day 8 that was just predicted.

2.2.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) networks [7], illustrated in Figure 2.2, is a recurrent neural network architecture that has been among the most popular to use for sequential data such as time series. The TFT uses an LSTM as part of its architecture, and we have used an LSTM when adding support for string input as described in Section 3.1.

To be able to remember information from several steps back in a sequence, the LSTM architecture includes a cell state that runs through a chain of LSTM cells (one for each time step) and the LSTM has the ability to add or remove information from the cell state. This is regulated by gates utilizing a sigmoid activation function that generates values between 0 and 1, deciding what information should be kept or be removed. The first step of the LSTM is called a "forget gate" defined as

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.2}$$

where x_t is the input at the current time step, h_{t-1} is the output from the previous time step, and W_f and b_f are the weights and biases of the forget gate, and σ is the sigmoid activation function. The next part consists of an "input gate", that determines the values to be updated, and a cell state update $\tilde{c_t}$ using a tanh activation function,

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.3}$$

$$\tilde{c_t} = \tanh(W_c[h_{t-1}, x_t] + b_c). \tag{2.4}$$

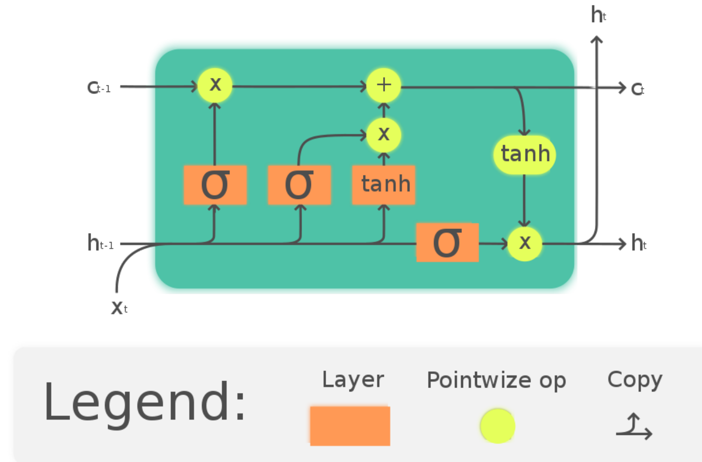


Figure 2.2: Illustration of an LSTM cell. Cells are chained together to form a complete LSTM architecture. From [5]. CC BY-SA.

Just like the forget gate, the input gate i_t and cell state update $\tilde{c}_t$ have their individual weights and biases. The forget gate and the input gate are then combined to update the cell state information c_t by elementwise multiplication,

$$c_t = f_t * c_{t-1} + i_i * \tilde{c}_t. \tag{2.5}$$

Finally, the output h_t of an LSTM block is calculated as follows:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{2.6}$$

$$h_t = o_t * \tanh(c_t). \tag{2.7}$$

Intuitively, the LSTM can learn to remember relevant information from many time steps back that is useful for producing the final output since they alleviate the issue of the vanishing gradient problem. The vanishing gradient problem arises when the partial derivatives in gradient descent calculations become very small or near zero in neural networks. Thus, the updated weights of the model barely change in each step and learning is no longer possible. This is something traditional recurrent networks struggled with as they do not have the same ability to decide what information to remember. This has caused LSTMs to for a long time be among the highest scoring models on sequential learning benchmarks such as machine translation. However, LSTMs can still have a hard time determining all the long-term dependencies while doing these continuous cell to cell operations. In more recent years, the Transformer architecture that we discuss next has increasingly started taking over.

2.3 Transformer

The Transformer architecture [23] was introduced in 2017 and it takes a newer approach to processing sequential data. Rather than processing a sequence sequentially, it processes all items in parallel using a mechanism called attention. While

doing this, each item can access information from all other items regardless of their position in the sequence, giving it an advantage over LSTMs for learning long-term dependencies.

When processing items independently in parallel, information about order is lost. Therefore, a positional encoding is first added to each item to indicate its relative position in the sequence. The Transformer also consists of several other parts, including an encoder and a decoder. We here choose to describe mainly the attention mechanism, as that is the part of the Transformer that the TFT uses. More specifically, we describe self-attention where each item in a sequence attends to items in the same sequence.

A self-attention layer starts by doing a linear transformation of each positionally encoded item into a query vector, a key vector, and a value vector. As this is done for all items at once, this is rather represented with matrices with Q, K and V being the query, key and value matrix respectively. The attention is then calculated as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.8}$$

where d_k is the dimensionality of the keys. Intuitively, the query Q and key K matrices are multiplied together and passed into a softmax function to calculate the attention given to each item, which is then used to perform a weighted average over the key values V.

An example of self-attention can be seen in Figure 2.3. It shows the attention for the word *it* which is the strongest for the first two words in the sentence. The model has in this case learned that *it* refers to the animal.

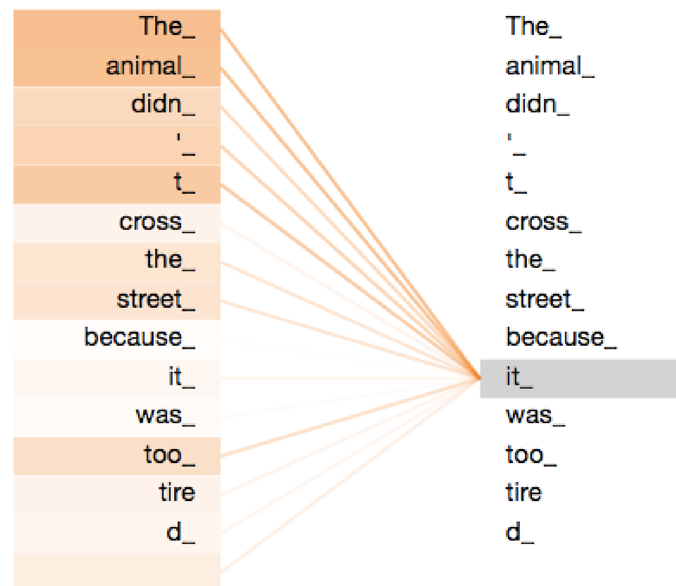


Figure 2.3: Self-attention for the word *it*. Stronger colors represent stronger weights. Note that *didn't* and *tired* are built from subwords, further explained in Section 2.6. From [1]. CC BY-NC-SA.

Additionally, multi-head attention [23] can be used to increase the capacity of the attention layer to allow it to understand more complex relationships. This splits the layer up into multiple *heads*, each learning a separate set of Q, K and V matrices[1]. The outputs of all heads is then combined into single output of the initial dimension.

2.4 Temporal Fusion Transformer

The Temporal Fusion Transformer (TFT) [10], which is the primary focus of this thesis, is a Transformer adapted for time series with several other architectural additions, published in December 2019. It makes use of multiple types of input features including past information (e.g., previous sales), future known inputs (e.g., holidays), and static features (e.g., product names) that do not change with time. The TFT takes advantage of attention for providing insights into the interpretability of the model and learning longer-term dependencies of time series data. An overview of the TFT can be seen in Figure 2.4 and 2.5. In the following sections, we go through the architecture from input to output.

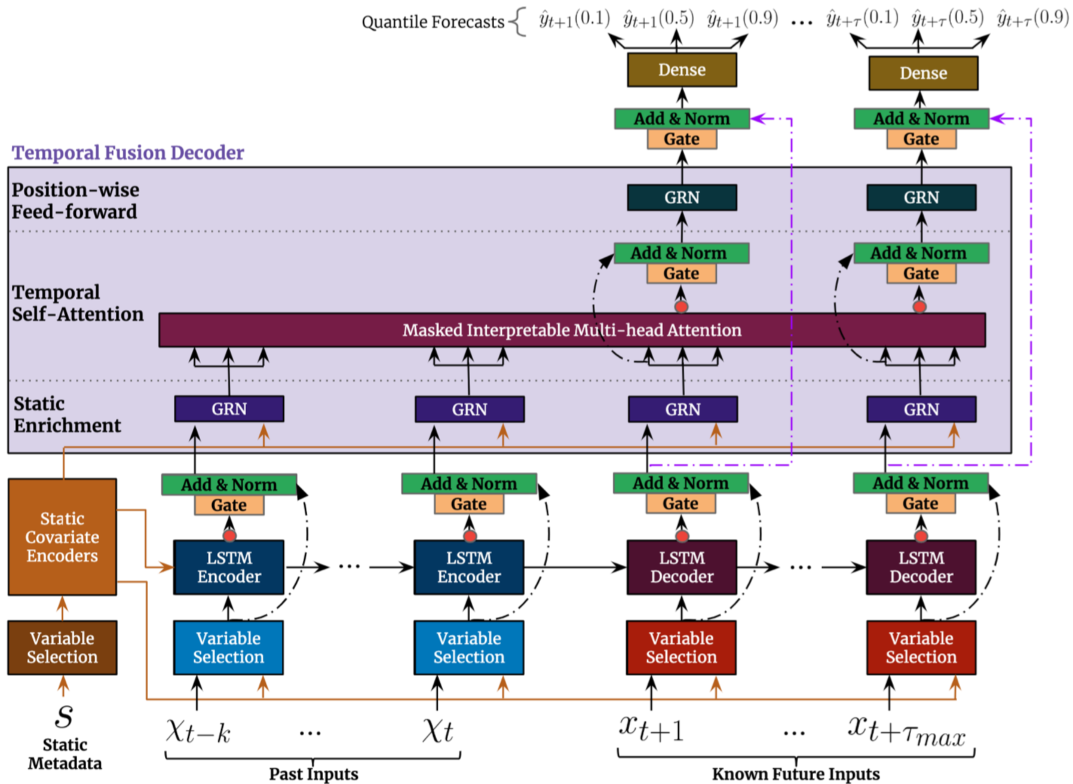


Figure 2.4: Illustration of the TFT model architecture. Historical, future, and static inputs are treated differently when entering the network that in the end produces quantile forecasts. From [10]. Reproduced with permission.

[1]The TFT uses "Interpretable Multi-Head Attention" [10], where all heads share the same V matrix and the final output is the average over each heads output.

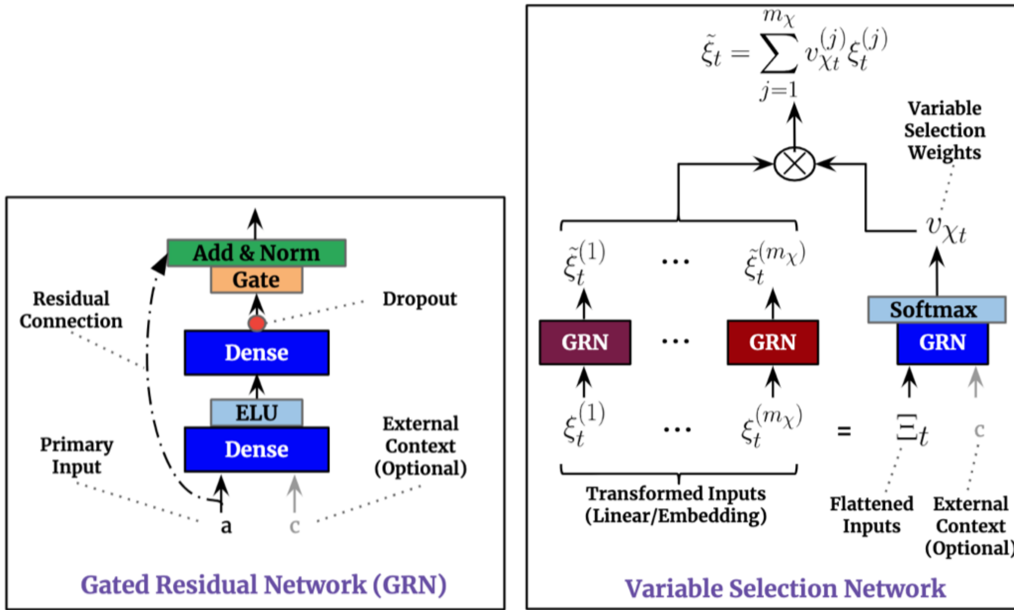


Figure 2.5: GRN and Variable Selection Network, two components of the TFT. From [10]. Reproduced with permission.

2.4.1 Input Types

The TFT can take both historical, future, and static features as input when making forecasts. The term *known inputs* is also used to represent both historical and future inputs. This is used because future features are also available historically, where one example would be if there is a promotion for a product or not. We know if a promotion is planned for the future, and we know if there was a promotion in the past. Historical features, such as the number of sales that we are predicting, are referred to as observed input, and static features are features that do not change with time such as the name of a product. Figure 2.6 illustrates how these different kinds of features are used as input to the model.

All inputs can also be of two different types, either real-valued or categorical. A real-valued feature is simply a scalar value in $\mathbb{R}$, and a categorical feature is a feature with a distinct number of possible values. We explain in Section 3.1 how we add a third type, namely string-valued features.

First, the input is transformed to a vector of size d_{model}, also referred to as the TFT state size. How this is done depends on the type of the feature. For real-valued features, the value is simply inserted into a linear layer

$$\xi^{(j)} = x_j w_j + b_j \tag{2.9}$$

where $x_j \in \mathbb{R}$ is the value of feature j, and $w_j \in \mathbb{R}^{d_{model}}$ and $b_j \in \mathbb{R}^{d_{model}}$ are the weights and biases for feature j respectively. The output $\xi^{(j)} \in \mathbb{R}^{d_{model}}$ is the vector representation of the feature in the TFT state size. For categorical features, each distinct value is mapped to a vector

$$\xi^{(j)} = x_j E_j \tag{2.10}$$

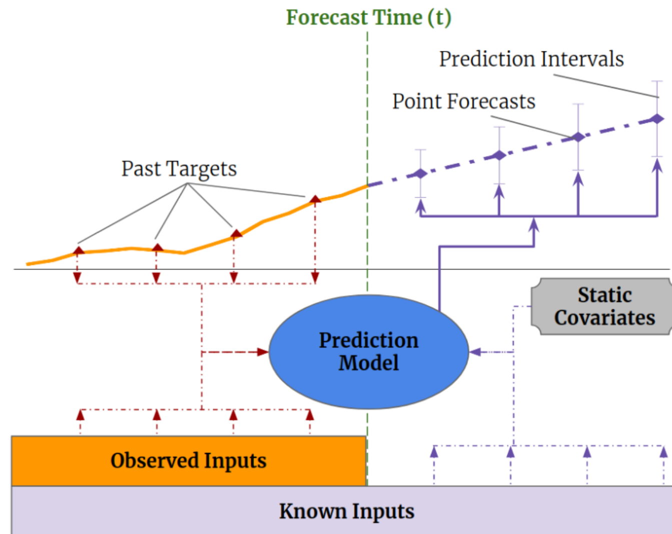


Figure 2.6: Illustration of how a time series prediction model such as the TFT takes different kinds of features as input. From [10]. Reproduced with permission.

where $x_j \in \mathbb{R}^{1 \times n_j}$ is the one-hot encoding of the categorical feature j, with n_j being the number of distinct values, and $E_j \in \mathbb{R}^{n_j \times d_{model}}$ is the matrix defining the vector representation $\xi^{(j)} \in \mathbb{R}^{d_{model}}$ for each categorical value. This transformation of each input feature into the TFT state size d_{model} is done at each time step for historical and future inputs, and once for static inputs.

2.4.2 Variable Selection Networks

Variable selection is applied to the transformed input variables to place stronger emphasis on features that are more useful for making accurate predictions, while possibly discarding irrelevant information. A variable selection network takes as input a set of values $\xi^{(1)}, ..., \xi^{(m_\chi)}$ where m_χ is the number of features for input type χ representing either historical, future, or static input. This is illustrated in Figure 2.5, with each input type having its own variable selection network with its own weights.

The variable selection network is composed of two branches. One branch consists of a series of Gated Residual Networks (GRNs, described in Section 2.4.3), further transforming the inputs. The other branch creates the variable selection weights. It has a GRN that is applied to a concatenated or flattened representation of all features, followed by a softmax layer to get their weights to finally output the weighted average $\tilde{\xi} \in \mathbb{R}^{d_{model}}$.

Furthermore, the historical and future variable selection networks can also make use of static context, illustrated in Figure 2.4. If some temporal features are more important depending on the static features they can then get higher weights. For example, if the product is considered to be a Christmas item or another type of seasonal food product, the combination of the time of the year and the particular static features are valuable to the model.

2.4.3 Gated Residual Networks

Gated Residual Networks (GRNs) are a type of gating mechanism that filters out irrelevant information using gates, illustrated in Figure 2.5. The GRN includes dense layers followed by dropout and a gate that can selectively cancel out values to remove unnecessary information. They are defined as

$$\mathrm{GRN}_\omega(a, c) = \mathrm{LayerNorm}(a + \mathrm{GLU}_\omega(\eta_1)) \tag{2.11}$$

$$\eta_1 = W_{1,\omega}\ \eta_2 + b_{1,\omega} \tag{2.12}$$

$$\eta_2 = \mathrm{ELU}(W_{2,\omega}\ a + W_{3,\omega}\ c + b_{2,\omega}) \tag{2.13}$$

where a is the input and c is an optional static context vector, ELU is the Exponential Linear Unit activation function, LayerNorm is used for standard layer normalization, GLU is a Gated Linear Unit defined below, $\eta_1 \in \mathbb{R}^{d_{model}}$ and $\eta 2 \in \mathbb{R}^{d_{model}}$ are intermediate layers, and the index ω is used for variables that share a set of weights W and biases b [10]. The GLU serves as a gating mechanism to filter out unnecessary parts of the model that do not support the learning, defined as

$$\mathrm{GLU}_\omega(\gamma) = \sigma(W_{4,\omega}\ \gamma + b_{4,\omega}) \odot (W_{5,\omega}\ \gamma + b_{5,\omega}) \tag{2.14}$$

where σ is the sigmoid activation function and $\odot$ is the elementwise product, allowing the gate to set values to zero that convey unnecessary information. A residual connection bypasses the GRN, as is commonly done in most deep machine learning models to keep strong gradients and avoid issues such as the well-known vanishing gradient problem where the gradients become too small.

2.4.4 Sequence-to-Sequence Layer

After variable selection comes a layer of sequence-to-sequence LSTMs capable of identifying local temporal relationships. Two sets of LSTMs are used, one to process past inputs (encoder) and one for future inputs (decoder) as seen in Figure 2.4. Historical input features are first encoded and later decoded for future time steps, while known inputs are passed as input directly to both sets of LSTMs. The encoder LSTM also receives information from the static context for its initial state. After each time step, the outputs are passed through a GLU gate before entering the Temporal Fusion Decoder block to perform attention. There is also a residual connection that bypasses the LSTMs and provides the variable selection directly to the next block.

Unlike the traditional Transformer, the sequence-to-sequence layer described here is used as a replacement for the positional encoding. Without it, no information about order is preserved and the following attention layer would not be able to extract meaning from the disordered set of input features.

2.4.5 Temporal Fusion Decoder

The Temporal Fusion Decoder start with a Gated Residual Network (GRN) that combines the inputs with direct contextual information from the static features. It is

followed by a temporal self-attention layer, performing standard self-attention from the Transformer architecture [23] over all time steps. After this stage, the model is now working only with future time steps as seen in Figure 2.4. A residual layer also allows information to bypass the attention layer. Finally, the outputs from the attention layer and the residual connection are passed to another set of GRNs, and then yet another residual connection is introduced before using a dense layer to generate quantile forecast outputs as described in the next section.

2.4.6 Quantile Outputs

The TFT generates quantile forecasts using various percentiles, commonly defined as the 10th, 50th, and 90th percentiles. The 50th is the most likely output according to the TFT, while the 10th and 90th together form an 80% confidence interval. An illustration of this can be seen in Figure 2.7. This is advantageous as we are given information about how certain a prediction is, unlike other models that might give just a point estimate.

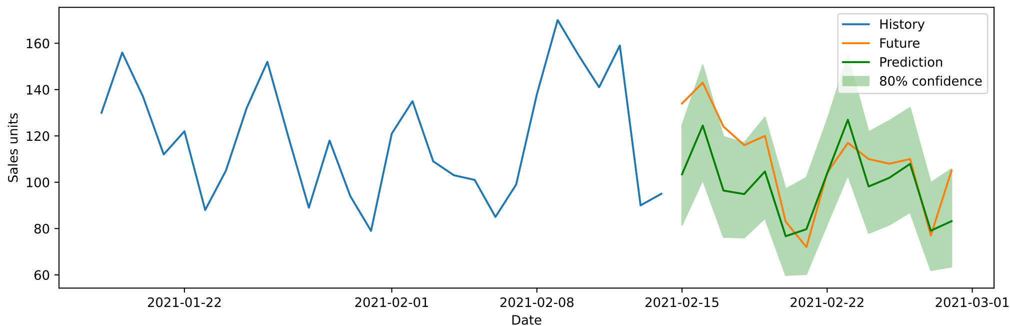


Figure 2.7: Quantile output performed by the TFT for a sample product. The 10th and 90th quantile together form an 80% confidence interval, while the 50th quantile is the point prediction.

The quantile loss function to be minimized when training the TFT is

$$\mathcal{L} = \sum_{y \epsilon \Omega} \sum_{q \epsilon Q} \sum_{t=1}^{t_{max}} \frac{QL(y_t, \hat{y}_{t,q}, q)}{M \cdot t_{max}} \tag{2.15}$$

$$QL(y, \hat{y}, q) = q(y - \hat{y})_+ + (1 - q)(\hat{y} - y)_+ \tag{2.16}$$

where Ω is the set of target time series $y \in \mathbb{R}^{t_{max}}$, Q is a set of desired output quantiles such as $\{0.1, 0.5, 0.9\}$ to be predicted at each time step t, and $\hat{y}_{t,q}$ is the prediction performed by the TFT for each time step t and quantile q. The number of samples is denoted M and $(.)_+ = \max(0, .)$.

2.4.7 Hyperparameters

The paper introducing the TFT defines six different hyperparameters that can be tuned. The following list provides an overview of them. In Section 3.4, we describe how we go about deciding hyperparameters.

- **State size:** The hidden layer size used for all parts of the model, denoted by d_{model}. All inputs are first translated to a vector in $\mathbb{R}^{d_{model}}$ and all hidden layers preserve this size until the output layer where scalar targets are produced for each future time step.
- **Dropout rate:** Dropout is used at several places in the model, indicated by a red dot in Figure 2.4. Dropout is used as a regularization technique to randomly drop a unit and its connection to simplify the model with a given probability. This hyperparameter defines the dropout rate for all places where it is used.
- **Minibatch size:** The minibatch size.
- **Learning rate:** The TFT is configured to use the ADAM optimizer and this hyperparameter defines the learning rate for it.
- **Max. gradient norm:** Used by ADAM to clip the gradient norms to avoid very large gradients.
- **Num. heads:** Number of heads used in the self-attention layer.

2.5 Entity Embeddings

Entity embedding is the mapping of entities to vectors in a common space where nearby entities are more similar [6]. In our case, an entity is a specific food product. Entity embedding is most widely used in natural language processing, with the most popular method being word2vec [15]. Compared to a one-hot encoding, this makes it easier for the model to treat similar entities in a similar way, leading to performance improvements. The model also becomes smaller as it does not have to learn weights for each individual entity.

Entity embeddings are learned while training the network [6, 3] and are implemented as a standard linear layer using a one-hot vector as input representing each unique entity. It can thus be described as $e = xE$ where $x \in \mathbb{R}^{1\times n}$ is the one-hot vector, with n being the number of unique entities, and $E \in \mathbb{R}^{n\times d_{model}}$ is the learned matrix defining the embedding $e \in \mathbb{R}^{d}_{model}$ for each entity with d_{model} being the desired dimensionality of the embedding to match the TFT state size. The dimensionality d_{model} should be chosen to be much lower than n to get the different entities to share features, providing a lower dimensionality representation for the rest of the network.

One issue with this way of performing embeddings is that the number of entities needs to be fixed beforehand, which is an issue as the number of food products in a store constantly changes. In section 3.1 we describe how we implement a solution for entity embeddings of a variable number of entities. We were unable to find other literature on this, as we only found resources performing entity embeddings on a fixed number of entities.

2.6 Subword Segmentation

When processing natural language, sentences need to be encoded in a way that a computer can understand. This is done by transforming the sentence to a list of tokens. Two common ways of doing this is character-level encoding and word-level encoding. By using each character as a token, the string “hello world” would be encoded as 11 tokens while if encoding each word, it would be only 2 tokens. The set of unique tokens is called the vocabulary.

If encoding is done on a character level, a simple scenario could have a vocabulary size of 27, having one token for each character A-Z and one token to represent all other characters. On the other hand, if encoding is done for each word, several thousand different tokens would be needed to be able to represent all possible words. Hence, character-level encoding results in long sequences of tokens with a small vocabulary, and word-level encoding results in shorter sequences but with a very large vocabulary.

Both long sequences and a large vocabulary make it harder for a network to perform well on tasks. In contrast, subword segmentation [21] solves both of these issues by learning *subwords* to be used as tokens. The algorithm for learning subwords can be described as follows, and is run as a separate step before training the neural network.

1. Start by defining the vocabulary letting each unique character in the training data become a token.
2. Find the pair of tokens that most frequently appear together, and form a new token by combining the pair. This new token is added to the vocabulary.
3. Repeat step 2 until a predefined target vocabulary size is reached.

To give an example, we take the sentence “hello bell”. In the first iteration, the tokens `e` and `l` are combined to form the new token `el`. In the next iteration, `el` and `l` are combined to form `ell` as that subword exists twice in the sequence. By using an appropriate target vocabulary size, the subword segmentation algorithm can achieve medium-length sequences and medium-sized vocabularies and has been used successfully in many different tasks [4]. In Section 3.6.4, we further describe how we use subword segmentation when processing food product names.

2.7 Interpretability

One of the common challenges of machine learning approaches is the ability to determine how an algorithm arrived at its conclusion, also referred to as the model’s interpretability. In this application, changes in customer demand may be due to seasonality factors, such as sales promotions, day of the week or month, and holidays. Identifying the significant variables and understanding the temporal patterns is possible when the model is interpretable.

Generally, there is an interpretability versus accuracy trade-off with more complex machine learning models [16]. Deep machine learning models often use model-

agnostic approaches, such as LIME and SHAP [20]. LIME creates local surrogate models for each individual data-point and SHAP considers features separately at each time step. These are not suitable methods for time series to explain the relationship between the time dependencies.

On the other hand, the TFT directly learns patterns during training and does not require additional model-agnostic methods and specifications for seasonality and lag analysis [10]. Interpretability insights are collected from both the variable selection network and the self-attention mechanism. The model supplies information through variable selection weights about which features are given more emphasis, and the model tracks long-range temporal patterns through its attention weights by attending to previous time steps. This makes the TFT advantageous compared to other time series models in terms of interpretability.

Self-attention is a significant advantage of the TFT, since weights show where in time the model attends to for making a prediction. For example, the model may attend to previous promotion periods for making predictions during another promotion. The larger attention weights for particular days in the past would indicate which days the model attends to. If there is a strong weekly pattern when items are being sold, the feature that provides information on which day of the week it is will have a large variable selection weight. Meanwhile, the attention weights will also fluctuate weekly in time.

2.8 Time Embedding

Just like entities can be embedded as vectors, time can be embedded as well. The simplest representation of time is a scalar value, indicating how much time has passed since a given start time. This conveys no information about periodic features or trends, making it harder for a model to make use of this value. If instead using a learned vector to represent time, such information can be included in the value.

One time embedding technique called Time2Vec [9] can be used to capture patterns that are either periodic or non-periodic. Some periodic events that may be present in time series include weekly or monthly patterns or seasonal patterns like holidays. Other events might be recurring and become more probable after a certain period of time. For example, higher sales are more likely for the days leading up to Christmas.

Time2Vec is implemented as a neural network layer and all parameters are learned. It takes as input a scalar representation of time τ, and outputs a vector representation which has a linear and periodic component. This vector representation can be referred to as a time embedding and is defined as

$$\mathbf{t2v}(\tau)[i] = \begin{cases} \omega_i \tau + \phi_i, & i = 0. \\ \sin(\omega_i \tau + \phi_i), & 1 \le i \le k. \end{cases}$$

where ω and ϕ are learnable parameters, frequency and phase-shift respectively, and $\mathbf{t2v}(\tau)$ is a vector with a size of $k + 1$ for time τ. Note that i is used to index into the vector.

Time2Vec has the advantage of being invariant to time rescaling [9]. For example, for a weekly pattern with the unit of τ as days, ω would be $\frac{2\pi}{7}$. If τ had been in a different time units like seconds, minutes or hours, ω would be rescaled accordingly. Similarly, phase shifts can be compensated for by the model learning a different value of ϕ.

When working with time data, careful consideration has to be taken when deciding how time is represented as we do in Section 3.3. Using Time2Vec could remove this need for manual feature engineering, as it allows the model to learn multiple patterns of varying frequency on its own.

3
Methods

This chapter describes how we implement and evaluate different variations of the TFT. First, the proposed support for a variable number of entity embeddings using a new string input type feature is explained, followed by a description of the data used and our approach to hyperparameter selection. Then after describing the autoregressive model and the LSTM model to be used as baselines, the experiments performed with the TFT is described with each step we take to modify it. At the end, how we evaluate the results is described.

3.1 String Input Type Feature

As seen in Section 2.4.1, the TFT supports real-valued and categorical features. The first thing that happens to an input is that it is transformed to a vector in $\mathbb{R}^{d_{model}}$. To support string valued features, we need a layer that does this for strings. Our proposed solution for doing this is to use an LSTM [7] with a state size of d_{model} that takes the string as input and outputs such a vector. Figure 3.1 illustrates how this is added to the TFT.

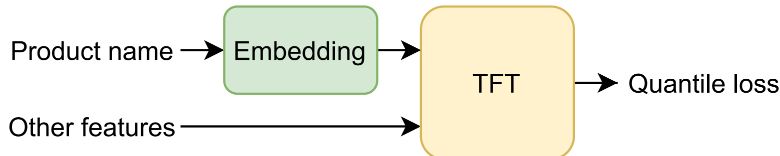


Figure 3.1: Adding support for string valued features to the TFT. The new embedding network consists of an that LSTM takes the variable-length product name as input and outputs a vector in $\mathbb{R}^{d_{model}}$ that the TFT can use.

By using a static string that identifies an entity, such as the name of a food product, the output vector of the LSTM also serves as an entity embedding. We call this implementation *variable-number entity embeddings*. Compared to *fixed-number entity embeddings* described in Section 2.5, our implementation supports any number of entities as we can input any string. One difference is that different entities get the same embedding if their static string have the same values. Nothing however prevents a standard entity embedding from mapping two entities to almost identical vectors if it is beneficial to do so. It might even be reasonable for two products with the same name.

First, the LSTM network is used to process strings one character at a time. As explained in the previous chapter, this results in long sequences and it can be hard for the network to learn to combine the characters and understand the full product name. Another alternative is to process words. This results in short sequences as most product names have few words in them. However, the number of different words is high, resulting in a large network that could overfit on rare product names and also does not understand unseen words. To overcome the issues of both character-level and word-level processing, subword segmentation [21] is also implemented and both character-level and subword-level processing are evaluated.

3.2 Data Selection

We have access to a proprietary dataset consisting of sales data for over 200 grocery stores in Sweden. For the sake of this thesis, we set a filter to include only products with an average of at least 20 sales per day, giving us roughly 3,500 products. The other products with much fewer sales than this look more like noise and do not seem to have a pattern that could be learned. Using this threshold, visual inspection of a sample of products shows apparent patterns that our models should be able to learn, giving us more insightful results.

The time series for each product begins in mid 2019 and ends in early 2021, covering a bit more than 600 days in total. We set aside 100 days each to use as a validation and test set. The final 100 days belong to the test set, and the 100 days before that belong to the validation set. The rest of the data is used as a training set. This gives roughly a 70/15/15 % split for the training, validation, and test set respectively.

This means that the training set contains just over a year of data. We investigate if this data along with information about which day of the year it is will be enough for the model to learn yearly trends, assuming that other years are relatively the same to one another. This may not be true if there are irregular patterns, for example the COVID pandemic starting in March 2020 may have changed regular sales patterns. Since we do not have data beyond February 2021, this is not evaluated further and it does not prevent the comparison between models.

The validation set encompasses Halloween and a school holiday the following week. Aside from that, it is very regular. The test set however goes through the Christmas and New Years period, having the most significant trend deviations seen throughout the time series. Therefore, the scores are expected to be worse on the test set as it is harder to predict.

As described in Section 2.1.1, we need to decide how far into the future we want to forecast and how many historical days to use. A forecast of 14 days into the future seems like a reasonable horizon for grocery stores when placing orders. The choice of how many historical days to use is harder. A small value would make it harder for the model to make predictions, while a large value would reduce the number of training samples available and possibly make it harder to learn useful entity embeddings as the model can see so much historical data at once and does not have to rely on the embeddings. The most extreme case for that purpose would be to use zero historical

days, making the model have to fully memorize each product's sales pattern in the entity embeddings. We decide to use 28 historical days, also motivated in Section 3.5.1 by a small scale example.

3.3 Feature Selection

There are multiple features available that we can feed to the TFT. The feature we are predicting is the number of sales units for each product. This is fed into the model as real-valued historical input, and the model's task is to predict future values. Another feature is if there is a promotion or not, which is known beforehand so future predictions are aware if there will be a promotion on the target date. We input boolean features like this as categorical features, so the TFT learns to embed it into two independent vectors.

There are several calendar features to consider, such as day of the week or day of the month. First, let us consider the day of the year as an example. There are 365 different values in a standard year. Doing a categorical encoding of this has several issues, such as nearby values being handled independently, in addition to that we have only about one year of training data. Thus, a year's worth of daily sales data is seen only once for each product. Treating it as a real-valued feature from 1 to 365 is also problematic because this would cause day 1 to be very far from day 365 which should not be the case as these are only one day apart. Therefore, a cyclical real-valued encoding [8] is chosen for this purpose which is calculated as follows, where d_{year} is the day of the year:

$$d_{year,\sin} = \sin\left(2\pi\frac{d_{year}}{365}\right) \tag{3.1}$$

$$d_{year,\cos} = \cos\left(2\pi\frac{d_{year}}{365}\right). \tag{3.2}$$

This maps the day of the year feature into two features, $d_{year,\sin}$ and $d_{year,\cos}$, that together represent the day of the year on a unit circle, correctly preserving the relation of nearby days. For leap years, the division is done by 366 instead. The same procedure was done for day of the month. For day of the week however, there are only 7 distinct features which would be well suited for a categorical encoding. Despite this, we decide to use a cyclical encoding to be able to more accurately compare variable selection weights between day of the week, month and year.

Note that week number and month number are not explicitly included as features, as these have the same yearly periodicity as day of the year. A division by 52 and 12 respectively would give these roughly the same value as day of the year divided by 365 but with a lower resolution, so they are not expected to provide additional value.

Another feature we include is a boolean feature for if there is a Swedish payday or not, defined to be true on the 25th of each month, or the Friday before that day if it is a weekend. This is not always correct as there might be holidays shifting it, but

we also include a boolean holiday feature based on the Python library *workalendar* that defines all common Swedish holidays. In addition to this, we define a feature called *Christmas season* which is zero most of the year, but starts at the value 1 on December first, increasing by 1 every day until January 10th after which it resets back to zero, to consider sales that might be influenced by the holidays.

When it comes to features related to individual products, we have access to ID numbers to distinguish between products. However, these are not consistent between stores, which is why we decide to use the product name as a string which also allows the model to handle unseen products which would have new IDs. We modify the TFT to be able to take static string inputs as described in Section 3.1. The product name string is evaluated both on a character level and using subword segmentation as described in Section 3.6.3 and 3.6.4. Finally, we have access to the product category which could be used as a feature, but instead we use it as part of the loss function to encourage products with the same category to be mapped near each other in the embedding space, further discussed in Section 3.6.5.

Again, the TFT can utilize the variable selection network to learn to ignore features that are not helpful so they do not waste model capacity. To demonstrate this, we include noise features that the TFT should learn to ignore. Both real-valued and categorical noise is used to see if there is any difference.

Below is a summary of all features just discussed. We made a delimitation to not distinguish between which store a product belongs to. Note that depending on the experiment, the number of features used varied.

- **Sales units:** The target value to predict. Real-valued historical input.
- **Promotion:** Boolean feature if there is a promotion. Categorically encoded known input.
- **Day of week:** Cyclical encoding using sine/cosine.
- **Day of month:** Cyclical encoding using sine/cosine.
- **Day of year:** Cyclical encoding using sine/cosine.
- **Payday:** Boolean feature if the current day is a payday. Categorically encoded known input.
- **Holiday:** Boolean feature if the current day is a holiday. Categorically encoded known input.
- **Christmas season:** Increasing values around Christmas season (month of December and the twelve days of Christmas) and zero elsewhere. Real-valued known input.
- **Real noise:** Gaussian noise. Real-valued known input.
- **Categorical noise:** A random variable with 10 distinct values. Categorically encoded known input.
- **Product name:** Our only static feature, to be processed as a sequential string.

- **Product category:** Used as a secondary loss function to enforce products with the same category to be mapped close together in the embedding space, further explained in Section 3.6.5.

As it might be clear, selecting features can be difficult. Some choices could be questioned, and more features could be created such as weather related data. In Section 3.6.6 we cover Time2Vec which is used a replacement for manually crafted date-based features, letting the model learn such features automatically.

3.4 Hyperparameter Selection

As the TFT has six different hyperparameters, each with multiple possible values, exploring all possible combinations of hyperparameters is infeasible as each experiment can take hours to run. In the TFT paper [10], the authors defined a set of values to try for each hyperparameter and perform a random search over combinations of these. This was carried out for each of the four different datasets, resulting in different but similar best hyperparameters found for each dataset.

To avoid having to go through a very thorough hyperparameter search, some hyperparameters are chosen based on the results obtained in the TFT paper. In particular, one of the datasets used in the paper comes from a retail store. As we are working with similar kind of data, extra attention is paid to the best hyperparameters found for this dataset.

The following list describes how we go about deciding each hyperparameter.

- **State size:** The state size is explored thoroughly, since it is an important hyperparameter in determining the size and performance of the model. A wide selection of exponentially spaced values (such as 8, 16, 32...) are tested in a set of experiments, until the optimal dimension is found.
- **Dropout rate:** The TFT paper tried nine different values, ranging linearly from 0.1 to 0.9. In two of the datasets, including the retail dataset, 0.1 was the best value found. We choose to keep a constant dropout rate of 0.1 in our experiments, also because of the nature of dropout with higher values generally making the training more unstable.
- **Minibatch size:** For the different datasets in the TFT paper, 64 and 128 were the best values found. After some experimentation, we found that higher values lead to better looking loss curves, as fewer steps are taken per epoch making each epoch more distinct. Higher values also lead to better GPU utilization and shorter epoch runtimes. Values of 256 and 512 are used in the experiments later presented.
- **Learning rate:** The TFT paper tried 0.0001, 0.001 and 0.01 as learning rates. For three of the datasets, including the retail dataset, 0.001 was the optimal value found so we decide to keep this value fixed during our experiments.
- **Max. gradient norm:** The values tried in the TFT paper are 0.01, 1.0 and 100.0. Two of the datasets used 0.01 and two other datasets had set the

parameter to 100.0 when the best results were found. The middle value of 1.0 was thus never selected and the authors did not explore this further. Because the two extreme values were chosen and not the middle one, we believe this choice does not have a strong influence on the model performance and the best values found might be due to the random search. In the case that other hyperparameters have a much stronger impact on the learning, then when good values were found for those hyperparameters, the choice of max gradient norm would be to a larger extent random which could explain the results found. We decide to fix this value to 100.0, considering the authors of the original paper used this value most often in combination with a 0.001 learning rate, and also because this value was used for the retail dataset. Furthermore, the combination of a high value for maximum gradient norm with the low learning rate should effectively disable gradient clipping as the gradients will most often not have such high values.

- **Num. heads:** In the TFT paper, either 1 or 4 heads were tried. Three of the datasets had the best results using 4 heads. For simplicity, we start with only using one head and have not yet investigated the results of using more heads.

3.5 Baseline Models

The main focus of this thesis is to analyze different aspects of the TFT, but it would also be of interest to see how it compares against other models. Therefore, two baseline models are implemented including an autoregressive model and an LSTM model. These models are described relatively briefly, as their results only serve as baselines.

3.5.1 Autoregressive Model

As a most simple baseline, an AR model [24] is implemented using the Python library *statsmodels* and only taking historical sales as input. Due to its simplicity, an AR model can only handle a single time series. Since there are multiple products, each with its own time series, one AR model per product is created. The training is performed on the training set. As there are no hyperparameters to tune once the number of historical days has been fixed, the validation set is not used to guide the training like with the TFT that picks the model from the best epoch in terms of validation set loss. This gives a bit of an unfair comparison between the models in regards to the validation set. However, the test set should give a more fair comparison since it is unseen during training for both models. Another alternative would have been to train the AR model on the training and validation set combined, likely improving its performance on the test set somewhat.

Initially, a small scale experiment was carried out comparing models of order 7, 14, 21 and 28 against a subset of all products. Multiples of 7 were used because of the clear weekly patterns in the sales data. With each increase in order, the error on the validation set was improved but the improvement from 21 to 28 was very minor on average, indicating that further increases in model order would not help

much. Only a minority of the products had a lower error using a model of order 21. This motivates that using 28 historical days is a reasonable choice. To make things simple, we decide to use 28 historical days for all AR baseline models just as for the TFT.

3.5.2 LSTM Model

Another baseline model that is compared to the TFT is the LSTM. There are many possible ways in which this can be constructed to optimize its performance. As we are only using it as a baseline, a relatively simple model is chosen with hyperparameters similar to the TFT where applicable.

The LSTM baseline's input features include sales data in addition to information on which day of the week it is, represented using cyclical encoding as described in Section 3.3. It is trained using the same train, validation, and test sets as the TFT and AR models. Batching is done in the same way as the TFT does, with sliding windows consisting of 28 historical days and 14 output days. The state size is chosen to be 32, the minibatch size is 512 and the ADAM optimizer is used with default parameters. The LSTM processes the 28 historical days and is followed by a dense layer with an output dimension of 14 to represent all future days. This makes it a single-shot model [22], processing the historical data sequentially and outputting all future predictions at once. An illustration of the model can be seen in Figure 3.2.

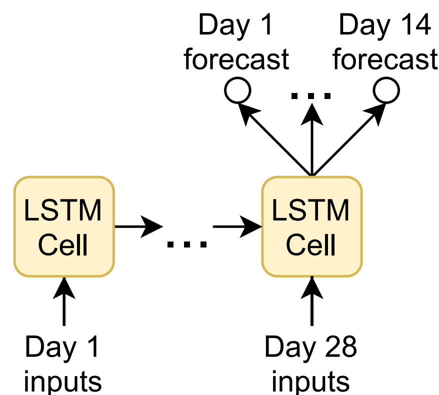


Figure 3.2: LSTM architecture used as a baseline. The 28 historical days are processed sequentially, before outputting the 14 future days all at once.

Unlike the AR model, the validation set is used during training and the best model weights are updated and saved until the validation loss stops decreasing and the training stops, which is how the TFT training is implemented as well. Furthermore, the mean squared error is used as loss function. Also unlike the AR, one model is trained to forecast all products just like the TFT.

3.6 Temporal Fusion Transformer

After generating the AR and LSTM baselines, multiple configurations of the TFT are evaluated using different hyperparameters. A TFT baseline is first trained using only

existing TFT functionality, with and without letting the TFT distinguish between products. Our proposed modifications are then gradually added as described in the following sections.

3.6.1 TFT Baseline

To start with, a TFT is trained using the number of units sold, the promotion feature, all known calendar features and the noise features. In short, all features listed in Section 3.3 except product name and category. This means that the TFT cannot distinguish between products, and has to perform the same calculations for all time series. This is our simplest TFT model, to be used as a baseline.

3.6.2 Entity Embeddings

As mentioned in the introduction, the TFT supports entity embeddings but the number of embeddings must be fixed. To evaluate how this fixed-number entity embedding performs against our variable length solution presented in Section 3.1, an experiment is carried out identical to the TFT baseline but adding a unique identifier for each product as a static categorical feature. This allows the TFT to distinguish between each individual food product.

3.6.3 Product Name Feature

Using the approach described in 3.1, the product name is used as a static string input processing one character at a time. This should allow the TFT to treat products with similar names similarly, and also lets it recognize products that have the same name across multiple stores.

3.6.4 Subword Segmentation

Subword segmentation is implemented using the Python library *sentencepiece*. It has several options that can be tuned when learning the subword tokens, such as the maximum length of a subword and other thresholds regulating which subwords to combine into new tokens. We choose to focus only on the vocabulary size and left all others parameters at their default settings.

After some experimentation, a vocabulary size of 561 was chosen. Anything higher than this resulted in an error message that no new subwords could be formed, being limited by the other parameters and our data. The results for this vocabulary size are also reasonable, the algorithm had learned common Swedish words such as *mjölk*, *smör*, *tomat* and *ekologisk* to give a few examples.

Also, in some cases the algorithm did not fully learn a word. For instance, *bröd* requires two subwords, *b+röd*, and *grädde* consists of *grädd+e*. This is reasonable as *röd* is for instance also used to form *röd mjölk*, and *gräddfil* can now be represented as *grädd+fil*. Note that the algorithm does not treat whitespace as a special character, and these learned subwords were often part of longer words.

Using this vocabulary size of 561, the average sequence length of all product names is 5.5 tokens. In contrast, the character level encoding results in an average length of 14.9 tokens, with a vocabulary size of 46 to represent each character in the dataset. We think this is a reasonable compromise between vocabulary size and sequence length, also considering that the fixed-number entity embeddings can be seen as one unique token for each of the 3,500 products with a sequence length of one.

3.6.5 Product Category Classification Loss

To encourage products belonging to the same category to appear near each other in the embedding space, a classifier network is added to the TFT that is trained to take the embedding vectors as input and output a classification for which category a product belongs to. This is illustrated in Figure 3.3.

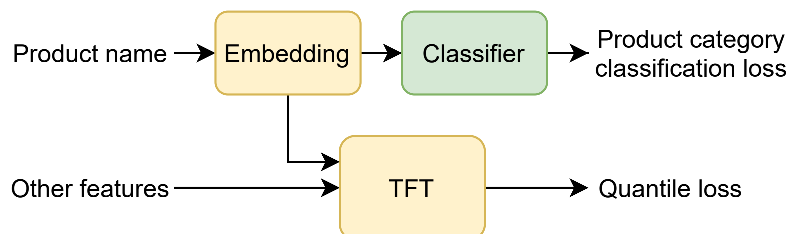


Figure 3.3: Our architecture with product category classification loss added. The embedding layer enables processing of the variable-length product names using an LSTM, processing either characters or subwords and outputs a vector in $\mathbb{R}^{d_{model}}$. The new classifier uses this vector to predict which category a product belongs to.

The classifier is implemented as a single layer using multiclass logistic regression. This layer learns a set of weights $W \in \mathbb{R}^{d_{model} \times k}$ and biases $b \in \mathbb{R}^k$ where k is the number of product categories. The probability of each class is then calculated as $\text{softmax}(eW + b)$ where $e \in \mathbb{R}^{1 \times d_{model}}$ is the embedding vector received as input and the softmax activation function is used to make the output probabilities sum up to one.

The new loss function $\mathcal{L}'$ using product category classification loss is defined as

$$\mathcal{L}' = \mathcal{L}_1 + \lambda \mathcal{L}_2 \tag{3.3}$$

where $\mathcal{L}_1$ is the previously used quantile loss, $\mathcal{L}_2$ is the new product category classification loss and λ is a new hyperparameter defining how the two losses are weighed together. The new loss $\mathcal{L}_2$ is defined as the categorical cross-entropy loss, a commonly used loss function for classification of multiple classes.

While keeping all other hyperparameters fixed, the product category classification loss experiment focuses only on exploring λ. The state size during this experiment is chosen based on the results of previous experiments. Either character-level or subword-level processing of the product name can be used, the choice of this is also made based on how those experiments perform.

As a simplification, we let $k = 21$. In these 21 categories, the 20 most common categories are included and all other products are assigned to a residual category.

During visualization we only look at the 20 most common categories, as a unique color is used to represent each category and it becomes hard to distinguish between the colors if more are used.

3.6.6 Time Embedding

Time2Vec is integrated into the TFT as an additional neural network layer to attempt to learn periodicity in the sales data with a sine activation function and a linear component for non-periodic events. The new Time2Vec layer comes before the variable selection network and produces and vector in $\mathbb{R}^{d_{model}}$ that can be used by the subsequent layers in the TFT.

To create the time embedding, Time2Vec takes a date ID as input for each historical and future time step. The date ID is any scalar representation of time, such as the number of days since a date defined to be day zero. The choice of starting date is arbitrary as the Time2Vec layer includes a learnable bias corresponding to a phase shift. Figure 3.4 shows how Time2Vec is added to the TFT.

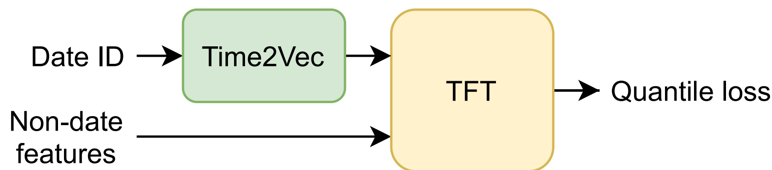


Figure 3.4: Illustration of Time2Vec added to the TFT. All date-based features such as day of the week are replaced by a scalar date ID passed into the Time2Vec layer that learns a vector representation of time in $\mathbb{R}^{d_{model}}$.

The experiment using Time2Vec is based on the TFT baseline experiment but replacing all date-based input features with the date ID to be processed by Time2Vec. That means the remaining features are the sales units, promotions, and real and categorical noise in addition to the date ID. These features are all transformed accordingly to a vector in $\mathbb{R}^{d_{model}}$ where the variable selection network forms a weighted average to be used by the rest of the TFT. The future variable selection network naturally does not have access to the sales units as this is what is being predicted.

3.7 Evaluation

After training, the models are tasked with predicting each sliding window in the validation and test set for each product. To get a metric that can be compared between all models, the mean absolute error (MAE) is used, defined as

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{3.4}$$

where n is the number of predictions made, $\hat{y}_i$ is the predicted value and y_i is the true value for each prediction i. All prediction dates from 1 to 14 days into the future are thus given equal weights. This gives an interpretable metric, a MAE of

for instance 10 means that a model on average predicted the number of units sold to be 10 more or less than the true value. In addition to the MAE, the quantile loss values are presented for the TFT models to reflect how accurate their confidence intervals are.

For evaluating the learned embeddings, t-SNE is used in the same way as in [6] and [3]. This gives a visual representation of all embeddings in 2D space that can be compared between models and give insights into what each model has learned. The embeddings are colored depending on which category a product belongs to (dairy, bread, ...) and the average sales per day of each product is also represented to increase interpretability of the visualization.

The interpretability of the TFT is evaluated by looking at the two different types of weights it can output. Attention weights show where in time the model attends to in the past to make a new prediction. Variable selection weights additionally indicate which features are the most important for the model to focus on. The model then does not need to retain so much information about features that do not contribute to the model's performance and instead bases its prediction on and attends to the features that matter the most.

4

Results

This chapter presents the results of all experiments described in the Method section in the same order they were introduced there. The forecasting performance of the baseline models are first briefly presented, followed by the different implementations of the TFT. The learned entity embeddings are also visualized. Finally, we look at attention and variable selection weights produced by the TFT.

4.1 Baseline Models

These subsections describe the results of the AR and LSTM models used as baselines. The results are only briefly described and attention should mostly be paid to the MAE values that can be used to compare the baselines to the TFT.

4.1.1 Autoregressive Model

An autoregressive model was trained for each product as described in Section 3.5.1. The MAE values averaged across all products are presented in Table 4.1. Training and evaluating all models took about 30 minutes running in parallel on a quad core laptop with hyperthreading[1].

Table 4.1: Autoregressive baseline model mean absolute errors.

Validation MAE	Test MAE
12.2163	15.2198

4.1.2 LSTM

The LSTM model described in Section 3.5.2 was trained on all products at once and took nearly 4 minutes to converge on an NVIDIA Tesla K80 GPU. The results are presented in Table 4.2. The MAE improves by about 1 unit compared to the AR model, which is expected since the LSTM is a more complex model.

[1]The implementation of the AR model was naive and could probably be made faster, but we do not go into detail on that here as it is not the focus of this report.

Table 4.2: LSTM baseline model mean absolute errors.

Validation MAE	Test MAE
11.4873	14.2718

4.2 Temporal Fusion Transformer

The following sections present the results for the different TFT configurations discussed in the previous chapter. Unless otherwise mentioned, each experiment was carried out using the fixed hyperparameters described in Section 3.4. Loss curves, point metrics, and embedding visualizations are presented. Curves for the MAE are not presented, but the trajectories looked almost identical to the loss curves for all experiments.

Training was performed using checkpointing and early stopping with a threshold of 5 epochs. This makes the training stop only when the validation loss has not improved for 5 epochs, after which the best model checkpoint from 5 epochs back is loaded. This is why the loss curves presented go on for a different number of epochs. Aside from these curves, all metrics presented are for the model from the best epoch for each experiment.

We initially started running the experiments with a minibatch size of 256, but soon changed to 512. This is reflected in the tables showing the results for each experiment. When doubling the minibatch size, half as many gradient steps are taken in each epoch which should lead to smoother and more gradually descending loss curves. Fewer gradient steps in each epoch also means each epoch is completed faster. Experiments with smaller state sizes particularly benefit from larger batches, leading to better GPU utilization. Other than this, we do not believe this choice has a significant impact on the end results.

Training was performed on an NVIDIA Tesla K80 GPU. The training times ranged between 6 and 16 minutes per epoch, mostly depending on the state size. The choice of features did not have a significant impact on the epoch times, as after the initial variable selection network, the number of different features does not affect the model complexity. Typical training times can thus be said to range from 1 to 4 hours depending on the number of epochs each model needs to converge.

4.2.1 TFT Baseline

Initially, a TFT baseline model was trained using all non-product-specific features. In this case, the model cannot distinguish between products. It was trained with a state size of 8, 16, 32 and 64. The validation loss curves are presented in Figure 4.1.

As we see, increasing the state size increases the performance of the model. This improvement stops at a state size of 64 which is where we decided to stop increasing it. The model with a state size of 8 shows clear signs of overfitting and had the highest MAE. Models with a larger state size seem to converge faster.

The metrics for the best epoch for each state size are presented in Table 4.3. These

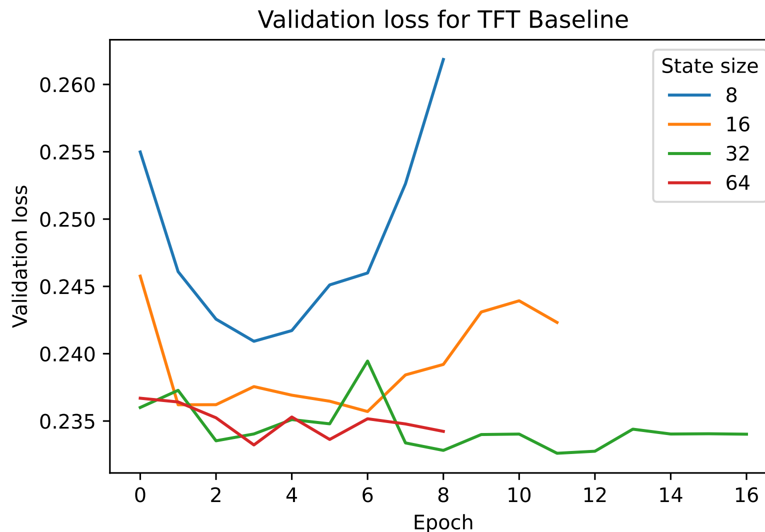


Figure 4.1: Validation loss curves during training for the TFT baseline model with different state sizes.

metrics serve as a baseline for all upcoming models to see how their changes affect model performance.

Table 4.3: Metrics for TFT baseline model.

State size	Minibatch size	Validation Loss	Validation MAE	Test Loss	Test MAE
8	512	0.2409	10.0364	0.2936	12.1633
16	512	0.2357	9.6719	0.2949	12.0777
32	256	**0.2326**	9.5288	0.2793	11.61
64	512	0.2332	**9.5199**	**0.2789**	**11.4916**

4.2.2 Entity Embeddings

The results from training the TFT and letting it learn an embedding for each product is presented in Figure 4.2. This is the fixed-number entity embeddings already supported by the TFT when providing a unique identifier for each product as input. Other than adding this input, the same features as the TFT baseline were used and the same state sizes were evaluated.

As can be seen from the loss curves, larger models quickly started overfitting the data. Only the model with the smallest state size of 8 did not overfit. To investigate this overfitting, the model with a state size of 32 was additionally trained with a dropout rate of 0.3 instead of 0.1 but the loss curve looked about the same. Larger values were not tried.

The metrics for the best epochs are presented in Table 4.4. Note that the best epoch is the first epoch for all models but the smallest. This is further discussed in the Discussion chapter.

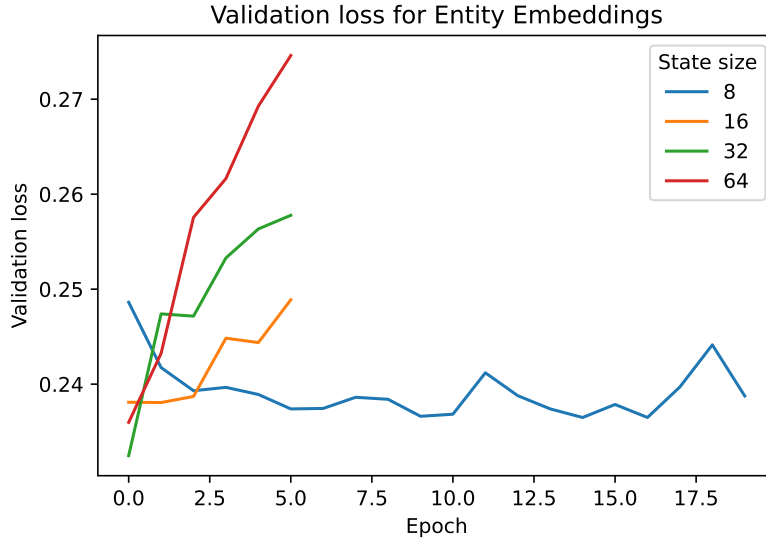


Figure 4.2: Validation loss curves during training for the TFT using fixed-number entity embeddings. By being able to treat each product independently, larger models easily overfit.

Table 4.4: Metrics for TFT using fixed-number entity embeddings.

State size	Minibatch size	Validation Loss	Validation MAE	Test Loss	Test MAE
8	512	0.2365	9.99	0.3037	12.5838
16	512	0.2381	10.0568	0.297	12.279
32	256	**0.2325**	**9.6588**	0.2801	11.709
64	512	0.2360	9.7539	**0.2753**	**11.4328**

The learned embeddings for each product was projected down to two dimensions using t-SNE and are presented in Figure 4.3. These particular embeddings are for the model of state size 64, but the others looked very similar. As can be seen, the model was able to learn surprisingly accurate embeddings even though it was given no information about which category a product belongs to or its name. It learned these embeddings based only on the sales patterns of each product.

4.2.3 Product Name Feature

A TFT model was trained using the name of a product as a replacement for the unique identifier used in the fixed-number entity embedding experiment in the previous section. Rather than treating all products independently, products with the same name are now treated the same and get the same learned embedding. The names were processed one character at a time by an LSTM as described in Section 3.1. The results can be seen in Figure 4.4.

In this case there was no apparent improvement from increasing the state size beyond 16, judging from the validation loss. Table 4.5 shows the metrics for each state size,

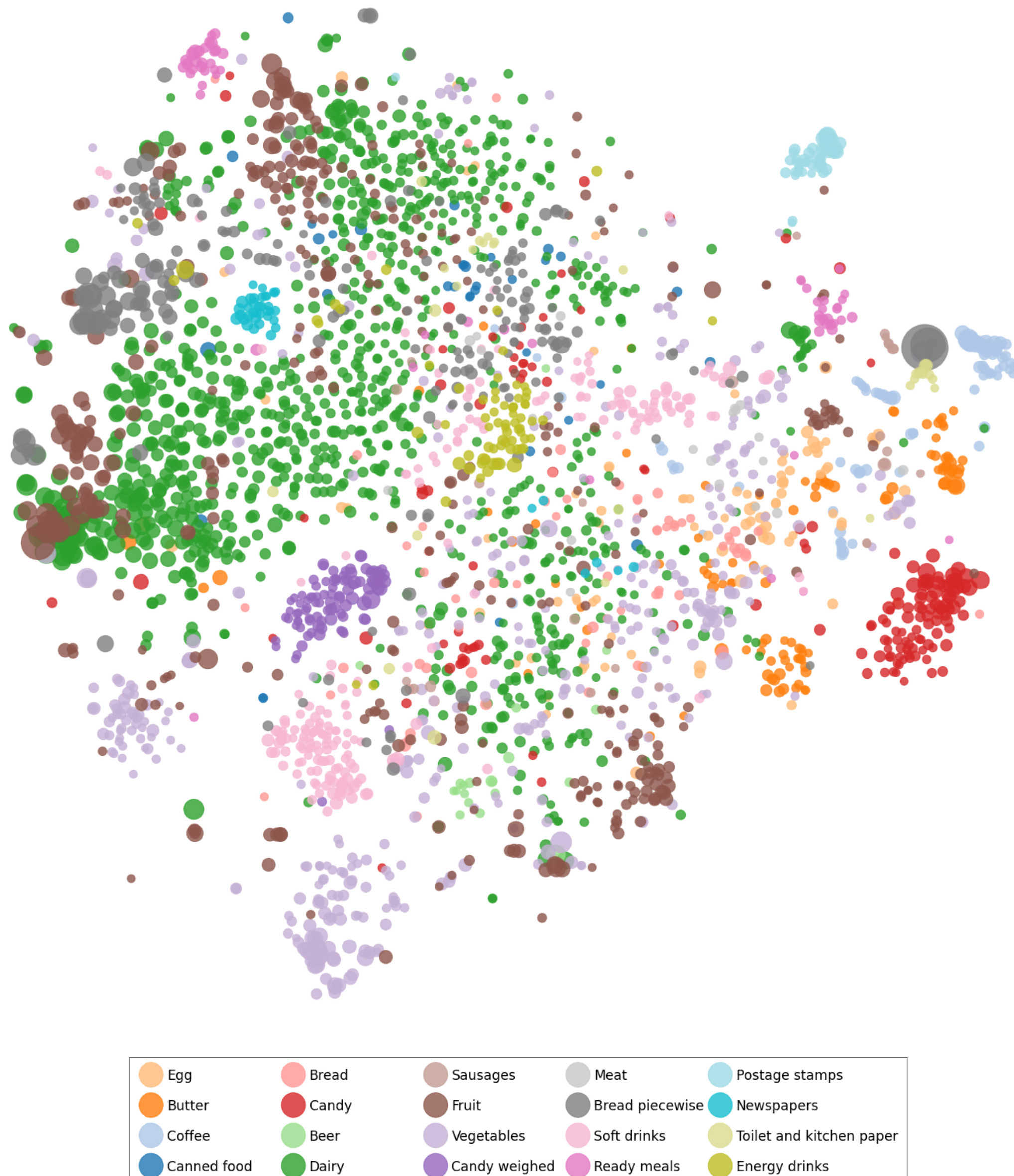


Figure 4.3: t-SNE projection of the learned fixed-number entity embeddings using a state size of 64. The color of a point indicates which category a product belongs to, and the size indicates the average sales per day. The model was not given any information about product categories during training.

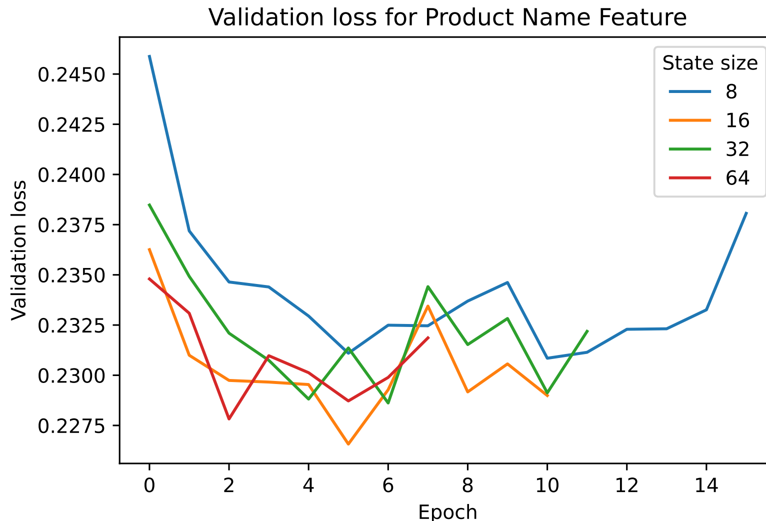


Figure 4.4: Validation loss curves during training for the TFT with the product name as input using character-level processing.

and Figure 4.7 compares the test losses with the other experiments. As can be seen, using the product name feature achieves consistently better test losses than both the TFT baseline model and the model learning an entity embedding for each unique product.

Table 4.5: Metrics for the TFT with the product name as input using character-level processing.

State size	Minibatch size	Validation Loss	Validation MAE	Test Loss	Test MAE
8	512	0.2308	9.6618	0.2819	11.7588
16	512	**0.2266**	9.5198	0.2767	11.4936
32	256	0.2286	9.4188	**0.2569**	**10.6705**
64	512	0.2278	**9.3767**	0.2651	11.0075

The learned embeddings are visualized in Figure 4.6 together with the results of using subword segmentation as presented in the next section. The presented embeddings are again for the model with a state size of 64, and the other models again resulted in similar embeddings.

4.2.4 Subword Segmentation

An identical TFT model was trained as in the product name feature experiment in the previous section, but this time processing the name of the food product using subword segmentation instead of character-level processing. This reduced the average length of the sequences to be processed from 14.9 to 5.5. The vocabulary size was also increased from 46 to 561 as described in 3.6.4, increasing the number

of learnable parameters for the LSTM processing the product name. The learning curves are presented in Figure 4.5.

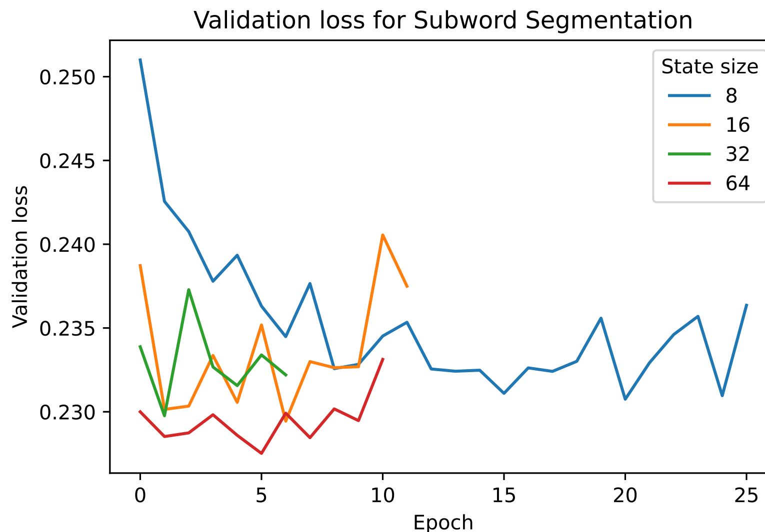


Figure 4.5: Validation loss curves during training for the TFT with the product name as input using subword-level processing.

Again we see how increased state size helped the learning. The smallest model had the slowest learning but eventually reached a similar loss as the other state sizes. The metrics for the best epoch are presented in Table 4.6.

Table 4.6: Metrics for the TFT with the product name as input using subword-level processing.

State size	Minibatch size	Validation		Test	
		Loss	MAE	Loss	MAE
8	512	0.2307	9.686	0.278	11.595
16	512	0.2294	9.5323	**0.2608**	**10.8541**
32	256	0.2298	9.4838	0.2703	11.319
64	512	**0.2275**	**9.3785**	0.2706	11.2579

The learned embeddings are visualized in Figure 4.6 together with the learned embeddings from the character-level processing. The model with a state size of 64 is visualized but all models had similar results. It can be seen how the models have learned somewhat meaningful embeddings, but not as well as in the fixed-number entity embeddings experiment where one embedding was learned for each unique product.

4.2.5 Product Category Classification Loss

The product category classification loss experiment is based on the TFT with the product name as input and adds the new classifier loss. We choose to use character-level processing of the product name as this had similar results as subword-level

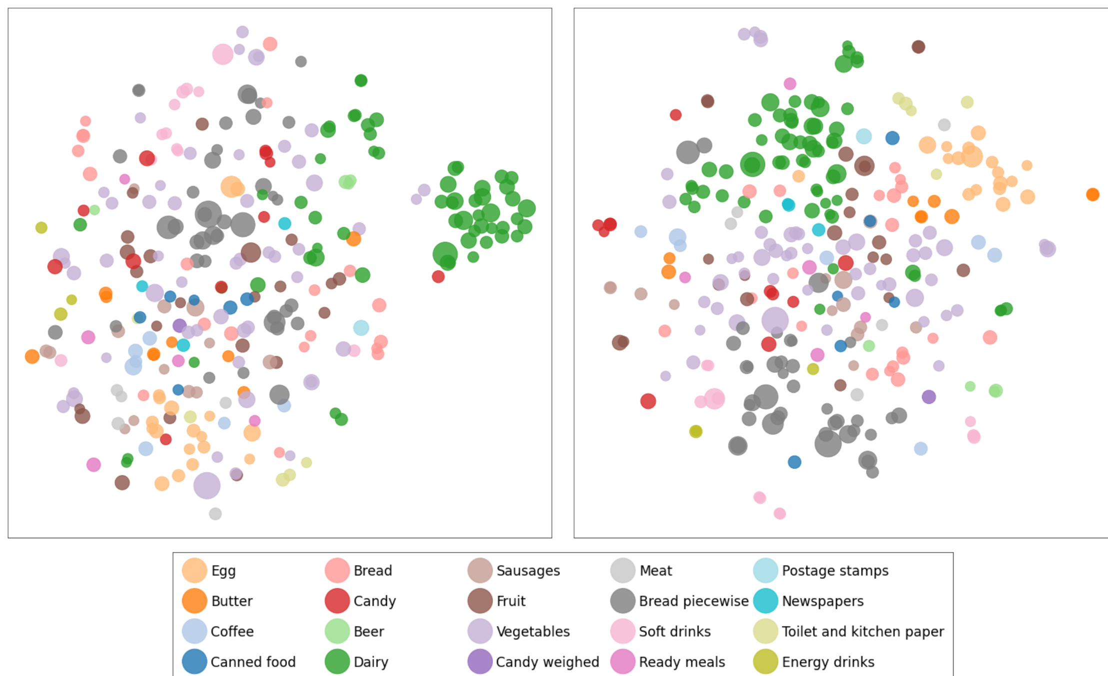


Figure 4.6: t-SNE projection of the learned embeddings based on the product name. The left image is the result of character-level processing and the right uses subword-level processing. Note that the number of points are much fewer than in Figure 4.3, as products with the same name get the same embedding so one point per unique name is presented. The size of a point then indicates the average sales per day for all products with the given name. The model state size is 64 for both models, other state sizes produced similar looking results.

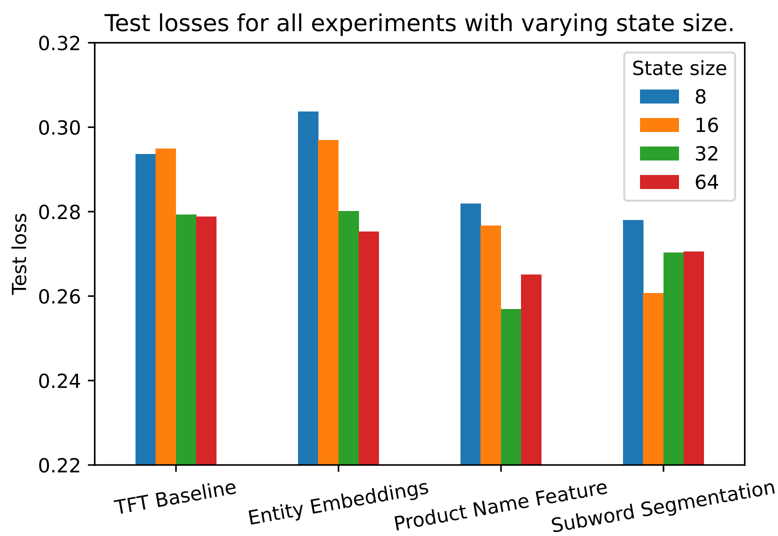


Figure 4.7: Test losses for all experiments with varying state size. Note that the y-axis does not start at zero. The TFT baseline and fixed-number entity embeddings experiment had similar performance, while using the product name either with character-level or subword-level processing increased the performance similarly.

processing and it should make it harder for the classifier to overfit. The state size is chosen to be 16 while exploring varying values of λ.

As can be seen in Figure 4.8, the model is able to achieve a lower quantile loss than the TFT baseline for all tried values of λ, but the chosen value does not seem to significantly affect the performance. With $\lambda = 0$, the experiment becomes identical to the product name experiment without classification loss so the result of that experiment is reused.

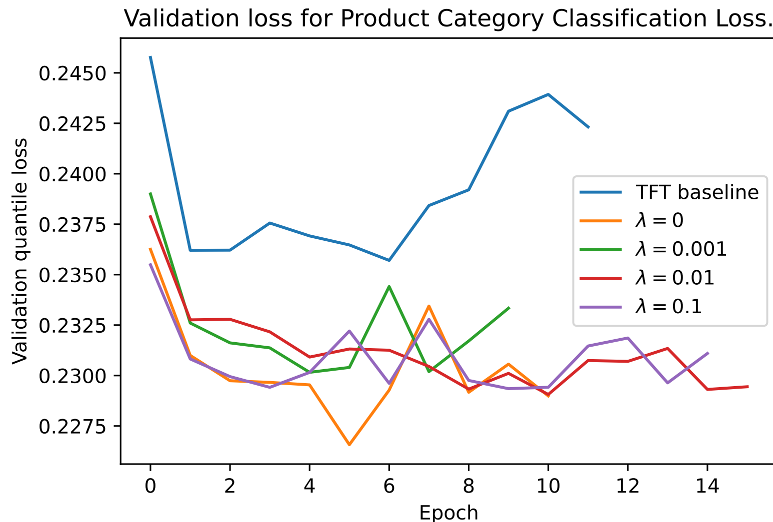


Figure 4.8: Validation loss curves during training for the TFT with the product name as input, adding product category classification loss with varying λ. Only the quantile loss is presented for comparability, seeing how it is affected by varying λ giving different weights to the classification loss which is not shown here. The TFT baseline is included as a reference. The state size is 16 for all models.

Table 4.7 shows the metrics for the best epoch for each model trained. Just as with the learning curves, it is hard to say if any value of λ is definitely better as the results do not differ significantly.

Table 4.7: Metrics for product category classification loss experiment using varying values of λ. The state size is 16 and the minibatch size is 512. The loss values presented are only the quantile losses.

	Validation		Test	
λ	Loss	MAE	Loss	MAE
0	**0.2266**	9.5198	0.2767	11.4936
0.001	0.2302	9.5141	0.2760	11.4807
0.01	0.2291	**9.4474**	**0.2710**	**11.1806**
0.1	0.2293	9.4891	0.2725	11.4506

The learned embeddings are visualized in Figure 4.9. Here we clearly see a difference between the models. A larger λ corresponds to a larger weight given to the product

category classification loss during training, making that loss lower and the categories more separated. This is seen in the visualization as the different categories start to form more clear clusters with increasing values of λ.

Figure 4.9: t-SNE projection of the learned embeddings for the product category classification loss experiment with varying λ. The model with $\lambda = 0$ is given no information about product categories. With increasing values of lambda, products belonging to the same category start to form clusters. All models have a state size of 16 and use character-level processing of the product name.

4.2.6 Time Embedding

A model based on the TFT baseline was trained using Time2Vec as a replacement for all date features as described in Section 3.6.6 using a state size of 32 and minibatch size of 512. The resulting validation loss curve is shown in Figure 4.10. As can be seen, it trains much slower than the TFT baseline without reaching any better result. This made us suspect our Time2Vec implementation might be incorrect, so an identical model was trained but without Time2Vec. This model is thus given no information about date features at all, but still exhibits a similar learning curve that is even slightly better than the Time2Vec experiment. The implications of this is further discussed in Section 5.3.

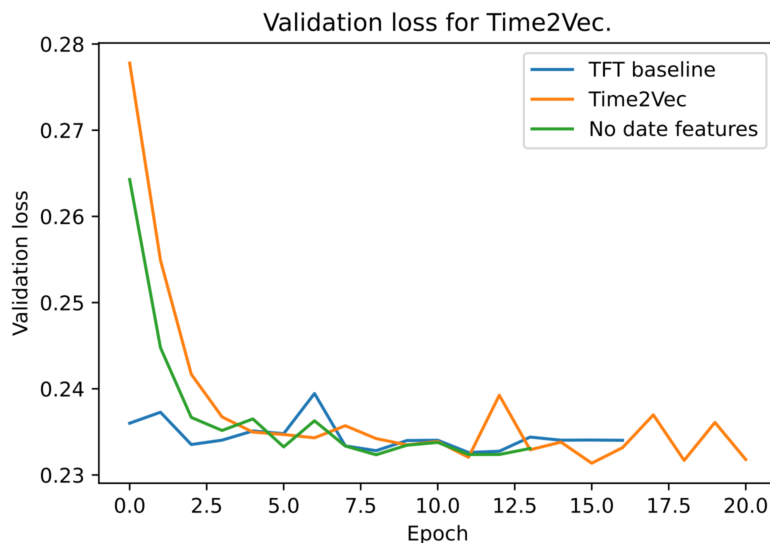


Figure 4.10: Validation loss curves during training of the TFT using Time2Vec. It converges to the same level as the TFT baseline but after much longer time. Another model was trained with no date features, and this had a similar learning curve making us believe our Time2Vec implementation is incorrect. The state size for all models is 32.

4.3 Interpretability

The following subsections present the attention weights and the variable selection weights learned by the TFT, providing some insights into the models' interpretability. To evaluate the interpretability, the test set is used. The time windows selected are thus within the November 2020 to February 2021 time frame.

4.3.1 Attention

Attention weights are used to emphasize which days to look back to when making a prediction. Typically, a product exhibits weekly patterns as shown by the attention weights in Figure 4.11, looking at the last 42 days of the test set. The model demonstrates its ability to learn which of the days in previous weeks it should

attend to, which are generally a factor of 7 days apart. If, for example, the weekly pattern is interrupted by a promotion period or holiday, the model also learns to skip these weeks, as seen for candy and soft drink products in Figure 4.11. On the other hand, if there will be a promotion on the predicted day, the model learns to attend to previous promotion days as shown in Figure 4.12.

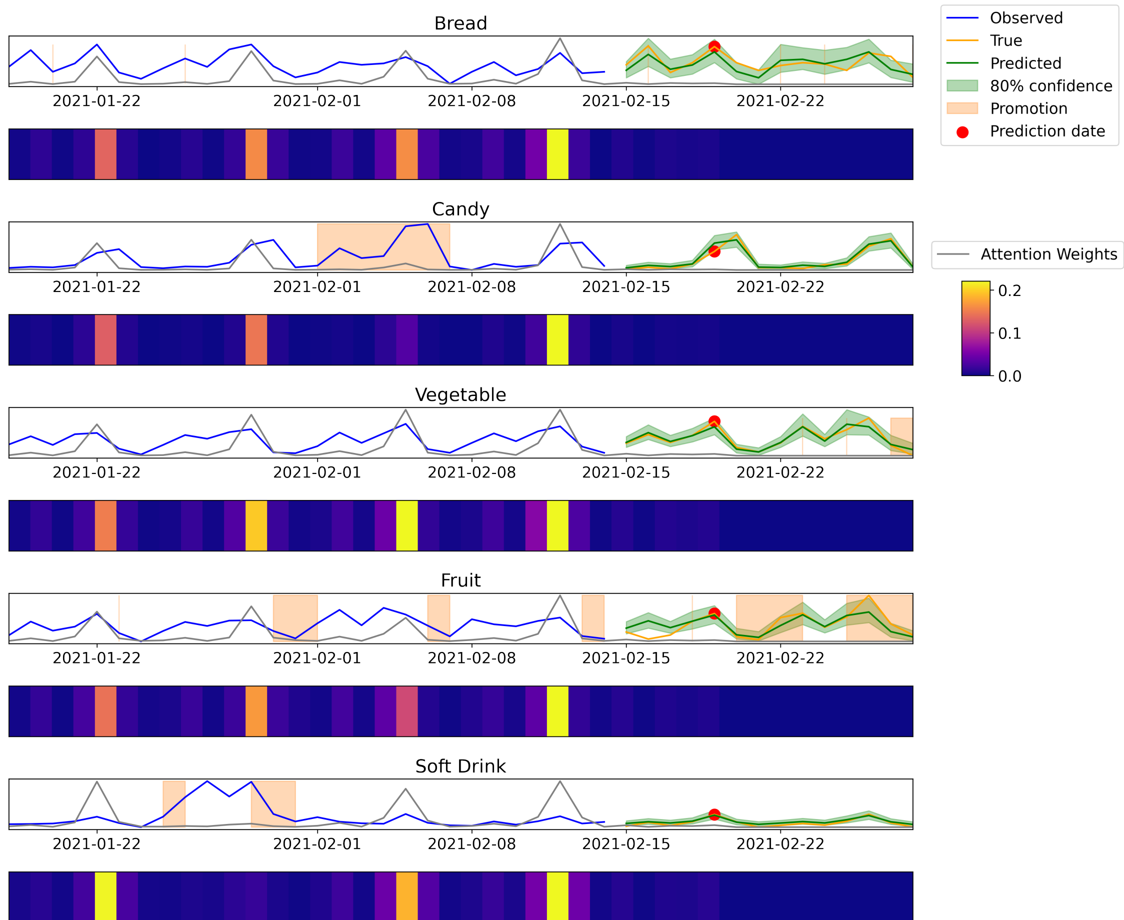


Figure 4.11: Attention weights for a selection of products. The model demonstrates a clear weekly pattern by attending to the same weekday from earlier weeks when making predictions, while avoiding to attend to irregular weeks such as when there was a promotion.

4.3.2 Variable Selection

Variable selection weights give an indication to which of the features are the most beneficial for the model. To obtain an overview of these weights, the mean of the weights over a time window ranging from January 18, 2021 to February 28, 2021 for the TFT baseline model with a state size of 32 is shown in Figure 4.13. In order to simplify the visualization of cyclically encoded features, the sum of the sine and cosine for the respective feature weights are presented. The results make it clear that the prominent weight is associated for the day of the week feature, reflecting the weekly periodicity in the data. Moreover, regarding categorical features, promo-

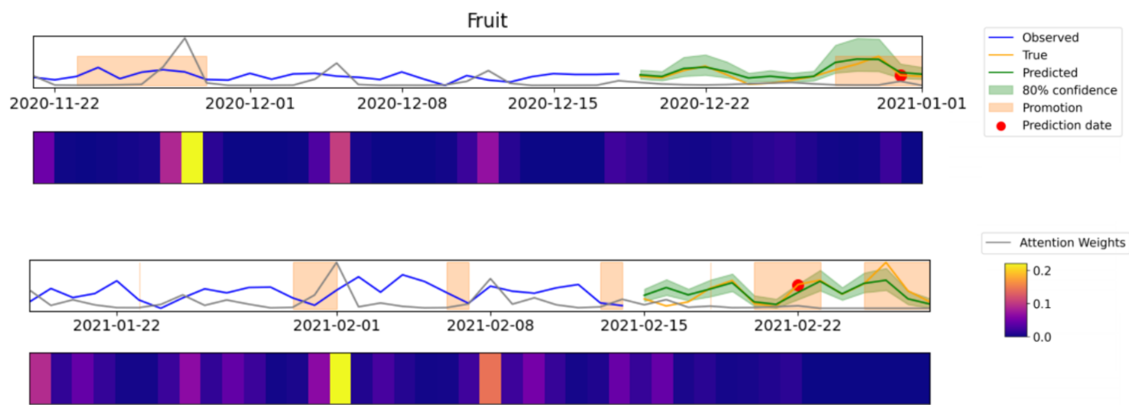


Figure 4.12: Predicting fruit sales during a promotion period. Greater attention weights are placed on a same weekday during another similar promotion period.

tions also appears valuable to the model. As expected, observed sales are the most valuable input to the model. In contrast, Christmas season and holiday features are used little in comparison, and this is also reasonable since the selected time window does not contain any particular holiday other than Valentine's Day. The day of the year feature received a relatively low weight but still a significantly higher weight than the weight given to real-valued noise. Categorical noise, on the other hand, appears to have quite a large weight and maybe even slightly higher than payday and holiday in this case. This is further discussed in the Section 5.4.

Similarly, for the best performing model with character-level processing of the product name, the largest weights are allocated to sales and day of the week features. See Figure 4.14. Promotions again receive the largest weight out of the categorical features on average.

The variable selection weights can also be visualized over time. In a sample time window shown in Figure 4.15, categorical and real-valued features are separated. Out of the categorical features, we again see that promotions has large weights compared to holiday and payday feature weights. Both holiday and payday features seem to be fluctuating roughly on a weekly basis. As for the real-valued features, both observed sales and day of the week features obtained consistently large weights. Note that observed sales appear in the first 28 days of the time window. This causes a shift in weights in the last 14 days, such that the weights still sum up to 1. In the first 28 days, the day of the week feature appears to fluctuate every 7 days, while day of the year remains relatively close to zero in weight.

Another time window in the month of December is shown in Figure 4.16 to demonstrate holiday sales patterns for a seasonal versus a non-seasonal item during the holidays. The Christmas season feature seems capable of differentiating between a product associated with Christmas versus a product not related to Christmas. The Christmas season feature weights gradually increases the days leading up to Christmas for the popular Christmas food item.

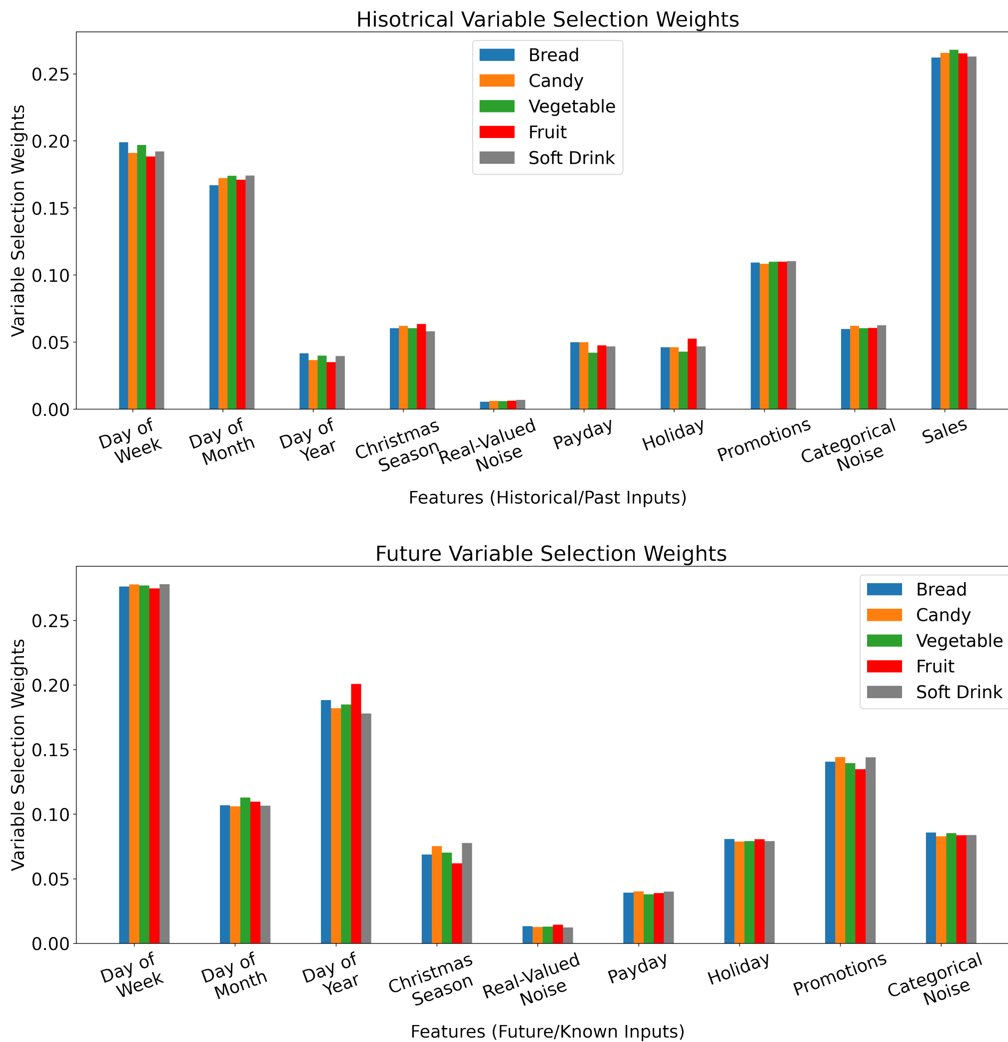


Figure 4.13: Historical and future variable selection weights from the TFT baseline model with a state size of 32, averaged between January 18, 2021 and February 28, 2021 for a sample of products. For the historical features, the number of sales is the most important variable for making predictions as expected, followed by day of the week. Promotions also appears to be a valuable categorical feature. For future variable selection weights, day of the week is the most significant variable of the known future inputs, followed by day of the year. Promotions received the highest weight out of the categorical features.

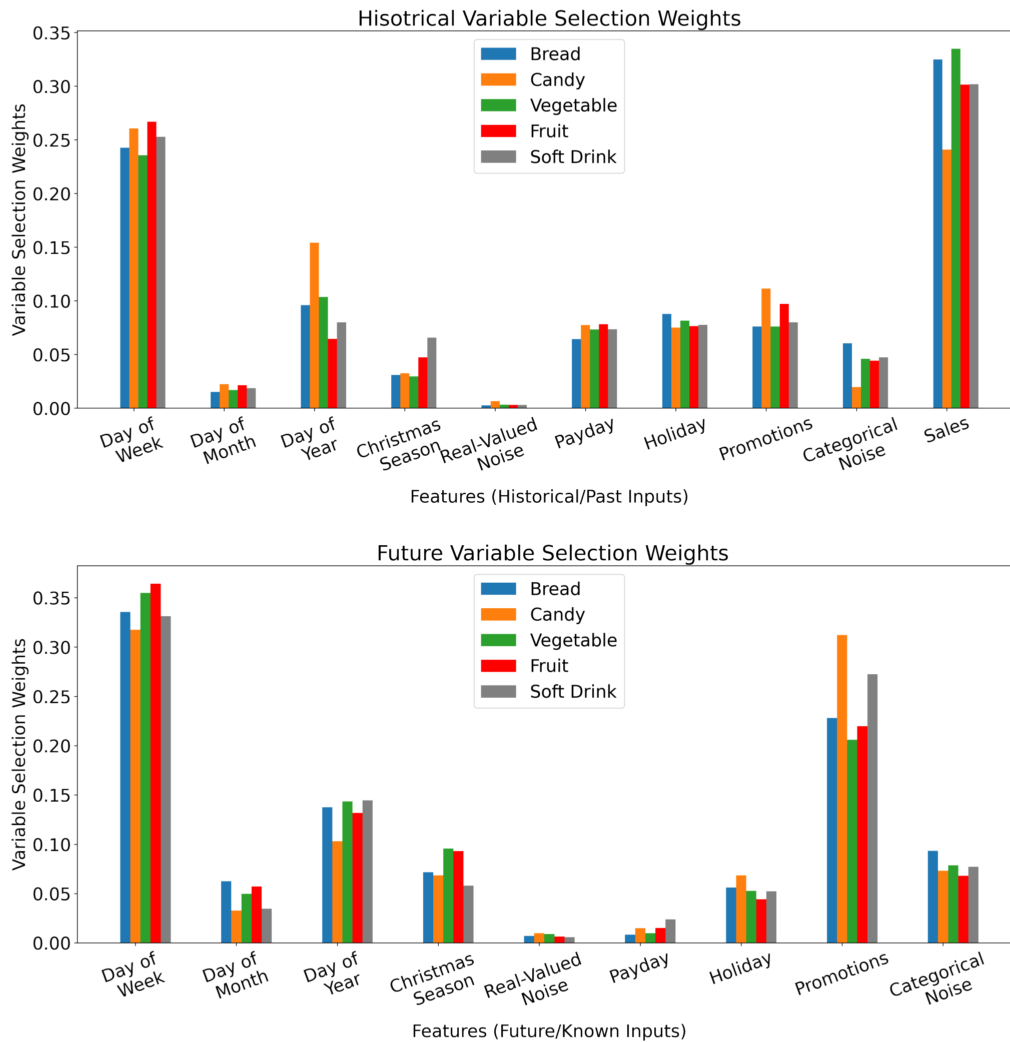


Figure 4.14: Historical and future variable selection weights for the best performing model with a state size of 32 using character-level processing of the product name. Weights are averaged between January 18, 2021 and February 28, 2021 for the same sample of products as in Figure 4.13. Sales has the largest weight, followed by day of the week out of the historical feature inputs. For the known future inputs, day of the week has the largest real-valued feature weight and promotions has the largest categorical feature weight.

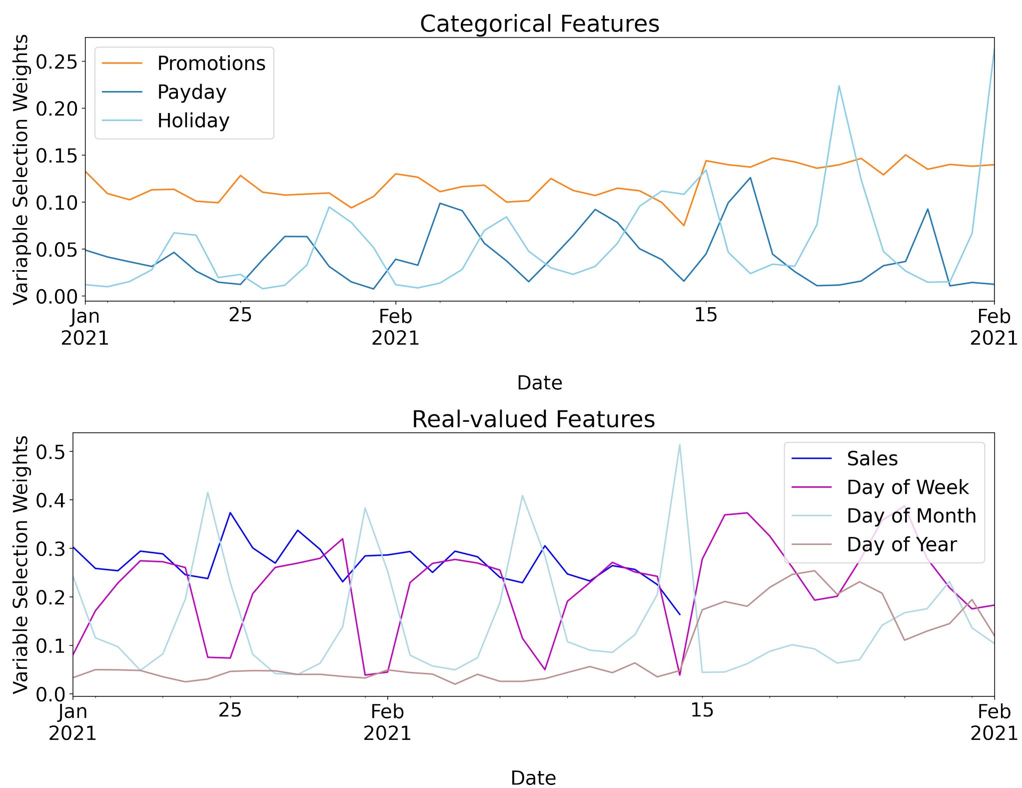


Figure 4.15: Variable selection weights of categorical and real-valued features for a sample product. The weights for the 28 historical days are concatenated with the 14 future days. This example time window is from the TFT baseline model with a state size of 32.

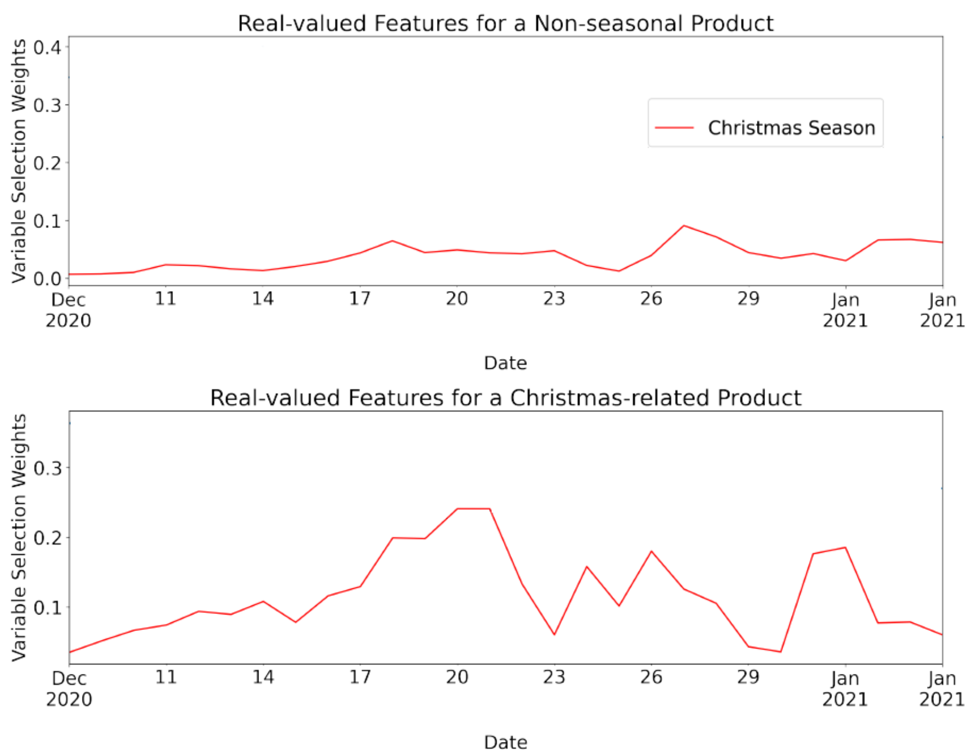


Figure 4.16: Historical categorical and real-valued variable selection weights during December for a Christmas-related and non-Christmas related product. Results are obtained from the TFT with character-level processing of the product name and a state size of 32.

5

Discussion

This chapter discusses the results obtained in the previous chapter, and our methods used to obtain them. First, the performance of the different models are compared and analyzed. The embeddings learned by the different models are then evaluated before looking into interpretability. How our models are expected to perform on unseen entities are then discussed before finally giving suggestions for future work.

5.1 Forecasting Performance

The TFT models developed have a MAE of between 10.7 and 12.6 sales units on the test set. The lowest MAE achieved is for the experiment with the character-level product name feature with state size 32. All TFT experiments performed significantly better than the AR baseline having a MAE of 15.2, and the LSTM with a MAE of 14.3. Since results show that the TFT can outperform such simpler models, the rest of this chapter concerns the different TFT models only. The quantile loss is used to compare forecasting performances between different TFT experiments, as the MAE does not account for the confidence interval the TFT produces through its quantile output.

Figure 4.7 shows an overview of all experiments that were done with varying state sizes in comparison to the TFT baseline. Generally the test loss is lower with a higher state size of 32 and 64 compared to 8 and 16, with the exception being for the subword segmentation experiment. The experiment performed with entity embeddings did not perform much better than the TFT baseline without product names as input, and results suggest even weaker performance especially at lower state sizes. However, a significant improvement is seen for all state sizes when processing the product name either on a character-level or using subword segmentation.

Generally, there is a large difference in validation and test performance results for loss and MAE. How the data is split and what seasonality it contains, such as holidays, is one major reason for this. This would be alleviated by increasing the amount of available data to work with since we have only access to just over one year of data, allowing a more equal comparisons of days in the year used in the train, validation, and test sets.

Looking at the learning curves for the TFT baseline model in Figure 4.1, we see that the model with a state size of 8 quickly starts overfitting, indicated by increasing loss values. We are perplexed by these results, as it should be harder for smaller

models to overfit on the training data. In contrast, it is clearly visible for the entity embedding loss curves, shown in Figure 4.2, that a smaller state size reduced the possibility of overfitting as expected. Another major difference between the loss curves is that more gradual learning occurs during the first two epochs of training for state sizes of 8 and 16, while the loss curves are relatively flat for state sizes of 32 and 64.

Using entity embeddings, the model can tailor its forecasts for each unique product, increasing the capability to overfit. This is also what the results show in Figure 4.2, it was only the smallest model that did not start overfitting, and the largest model overfitted the fastest. There is however reason to be critical of this experiment as it starts overfitting already after the first epoch. This suggests that training is performed too fast, and a better validation loss could be achieved with slower training. In future work, ways of improving on this could be explored, such as using a lower learning rate, or using regularization techniques such as dropout after the embedding layer.

The models seen so far that do not overfit have relatively flat learning curves already from early epochs. This indicates that one epoch contains a lot of data that is also similar. With roughly 3,500 products belonging to over 200 stores, many of these products are in fact the same product existing in multiple stores. It is reasonable to assume that a product's sales pattern is generally similar between different stores, causing much of the training data to be similar. In addition to this, most products are affected by the general weekly pattern of fewer sales on Sundays for instance. The sliding window approach with a shift of only one day also means that windows overlap which causes the same data point to be seen multiple times in one epoch, further contributing to large epochs with much similar data. Thus, a model has learned much already by the end of the first epoch.

Looking at the experiment using the product name feature on a character-level in Figure 4.4 shows a different pattern. It achieved a lower loss for all state sizes compared to the baseline, so it has managed to use the product name to improve its prediction. It however takes a few more epochs to do so compared to the baseline, making the curves look less flat. The only architectural difference compared to the baseline is the LSTM processing the product name one character at a time, and so it must be this that is harder to train than the rest of the TFT, making it require a few extra training steps to learn the similarity with certain food product names. We think this is a reasonable result, the sequences are somewhat long (on average 14.9 characters per product name) and all characters can appear in all product names, making the LSTM need to learn to combine characters to distinguish between products.

The character-level processing of the product name can be compared with using subword-level processing in Figure 4.5. The results on the test set are on average about the same when looking at all state sizes together. Here however, there is no initial gradual drop in the loss curves except for state size 8. The sequences are shorter (on average 5.5 subwords per product name) and each subword can be connected to a smaller set of products, which is quicker for the model to learn. There are also possible signs of overfitting looking at the larger state sizes, which

is not surprising as that was the case when using entity embeddings where it could distinguish between all unique products.

Another thing to note about the test losses presented in Figure 4.7 is that when using a state size of 8 or 16, subword segmentation had a lower loss than character-level processing. But for a state size of 32 or 64, the opposite is true. This also suggests that using subword segmentation increases the risk of overfitting.

The experiment adding the product category classification loss, shown in Figure 4.8 and Table 4.7, indicates that the choice of λ does not not have a significant impact on forecasting performance. As we discuss in the next section, increasing λ makes products with the same category group together. The embedding dimension with a dimensionality of 16 in this case seems to be large enough to represent both sales patterns and product category.

As all experiments using the product name as a feature had a better result on the validation and test set than treating each unique product independently, we believe treating a product based on its name can have a regularizing effect. The fixed-number entity embeddings experiment was able to overfit when treating each unique product individually, which was not the case when treating products based on their names. As these experiments achieved better results on the test set than the baseline, we have shown that our implementation processing the name of a product using an LSTM has enough capacity to improve the model performance using either character-level or subword-level processing.

5.2 Entity Embeddings

The fixed-number entity embeddings learned by the TFT visualized in Figure 4.3 look surprisingly good. The model is capable of learning these embeddings without any knowledge of which category a product belongs to. As the loss function of the TFT is designed to accurately predict future sales, it must have learned to group entities based on their sales patterns and that products of the same category often share sales patterns. These particular embeddings were also learned from only one epoch of training, strengthening the argument that the dataset contains a lot of similar data which is enough for the model to learn much from only one epoch.

The embeddings learned by processing the name of a product are visualized in Figure 4.6. Here the results are not as good looking, it is perhaps only the dairy products that are clearly separated from the rest. It is not obvious if using subword segmentation improved the embeddings. As it is the name of a product that is being embedded, much fewer unique points are available. We believe having more unique names would have made the embeddings look better.

This embedding of product names differs in two ways from the fixed-number entity embeddings experiment. As it is based on the product name, products with the same name get the same embedding. The other difference is that it uses an LSTM to process the name. It would also be possible to use the fixed-number embeddings approach but let it learn an embedding for each unique name. A new experiment doing this would more clearly show the effects of using an LSTM to process the

name.

Figure 4.9 shows the effects on the embeddings when adding the product category classification loss with varying values of λ. As expected, increasing values of λ results in more clear clusters. While these embeddings look better, the forecasting performance remains unchanged and it is not obvious how other tasks could make use of the embeddings. Other methods of using the product category could be explored to see if it can improve the performance, such as feeding it directly into the TFT.

5.3 Time Embedding

As seen in the previous chapter, our implementation of Time2Vec arrived at similar results to the identical experiment without Time2Vec, meaning it did not improve the model in any way. In the paper introducing Time2Vec, the authors report performance increases [9] which makes us believe that our implementation is incorrect. The authors provide a link to a GitHub repository containing code, but that repository no longer exists so we had to make our own implementation based on the description in the paper. As Time2Vec is not the main focus of this thesis, we conclude that we probably have made an incorrect implementation and include our code in Appendix A if further investigation is to be made.

5.4 Interpretability

From the attention and variable selection weights shown in Section 4.3, it is clear that the TFT model learns to identify a weekly periodicity in sales. The model attends to historical days spaced seven days apart. This is also reflected by the large variable selection weights for the day of the week feature, both as a historical and future known inputs. Also the TFT primarily attends to historical days for making predictions and the historical variable selection weights are a good indicator of which features are the most important for the model.

Knowledge about future promotion periods is also useful for the model. For a prediction during a promotion period such as the examples in Figure 4.12, the model still uses day of the week but prioritizes its attention to a day that also takes place during a promotion.

For the sample product within the specified time window shown in 4.15, day of the year does not seem to be contributing. Although this is often the case, during our investigations some cases were found where the day of the year variable selection weights peak on days with holidays or when there is a shift in sales due to promotions. It is difficult to say how much the model benefits from this feature exactly, but this could potentially be further explored by removing it from an experiment or seeing if more data would help with identifying other yearly patterns.

Seasonality in sales related to holidays is also captured by the model, as seen in Figure 4.16. It appears the TFT is capable of distinguishing between holiday items

rather well and also makes use of the Christmas season feature to identify a climb in sales before Christmas.

When comparing the variable selection weights of the TFT baseline (Figure 4.13) to the optimal performing TFT using the product name feature (Figure 4.14), the model distinguishing between products learned that the importance of certain features varies between products. For example, candy in Figure 4.14 is almost an outlier when it comes day of the year, promotions, and observed sales features. Thus, it could be that there was an increase in sales at a certain time in this period perhaps due to a long promotion for this product. In fact, this is actually the case. When looking at 4.11, one can see that there was a large promotion period in the first week of February.

It is interesting that none of the categorical features obtained zero-valued weights. Furthermore, when looking at Figure 4.15, payday and holiday features fluctuate weekly on a roughly similar trajectory to one another with just a phase shift. Since the time window ranges from January 18, 2021 and February 28, 2021, a reasonable assumption would be that that there would be no major changes in sales patterns due to holidays. Thus, holidays, like paydays, are perhaps not that meaningful in this case. Note that the categorical noise is around the same magnitude in mean weight as the holiday feature, in contrast to the real-valued noise feature which is almost zero on average as expected. Thus the categorical features are more difficult to interpret. Additionally, keeping in mind that the total sum of variable selection weights should always add up to 1, it is likely that the oscillations for these features are caused by the fluctuations in the weekly feature. For the same experiment, day of the month also fluctuates on a weekly basis. It increases whenever the day of the week feature decreases.

Another observation is the jump in trajectories during the future time steps, as seen in Figure 4.15. One explanation for this is because observed sales is not known in these future time steps so the weights are redistributed between the remaining features. Nevertheless, it seems that there is some loss in interpretability. For example, the day of the year feature went from having a near zero weight to having a larger weight than the day of month feature for no obvious reason. Hence, historical weights along with attention are perhaps more insightful in how the model makes its decisions.

5.5 Unseen Entities

We have evaluated different models on the same food products that they were trained on, in line with the delimitations we set out for this thesis. While this showed improved performance through the use of our proposed variable-number entity embeddings, the performance on previously unseen products has not been evaluated.

If a new product is introduced with a name that the model has seen before, and that product has a similar sales pattern to other products with the same name, we believe it would benefit from using the existing embeddings. However, if a product with a new name is introduced, this remains uncertain. For it to be beneficial, it would

rely on both the assumption that products with similar names have similar sales patterns, and that products with similar names receive similar embeddings. Neither of these are necessarily true, and risk leading to worse performance compared to not using entity embeddings.

As the performance of our proposed model for unseen products remains unknown, its main benefit is rather that it can be retrained on these new products without architectural changes. As new data appears with time, retraining a model is a natural part of its life cycle when used in real world applications. This training can continue from where the previous training left off, but now including the new product which would not be possible using fixed-number entity embeddings. After retraining, the new product would no longer be considered unseen and can now benefit from the increased performance we have shown that our variable-number entity embeddings can give.

5.6 Alternative Solutions

Alternative solutions to supporting a variable number of entities are worth exploring. As explained in the Background chapter, the fixed-number entity embeddings implementation learns an embedding matrix $E \in \mathbb{R}^{n \times d_{model}}$ where n is the number of entities, around 3,500 in our case representing each food product. As a new product appears, this can be replaced with an expanded matrix $E \in \mathbb{R}^{(n+1) \times d_{model}}$ to account for the new product. This is compatible with the rest of the network that expects an embedding vector in $\mathbb{R}^{d_{model}}$ as input. The model can then be retrained using only this product, while freezing the weights of all other layers. This would leave the rest of the model unchanged, while learning an embedding for only this product that should be close in the embedding space to similar products.

Another improvement to this might be to use the product name as input to the fixed-number embedding layer, rater than using a unique identifier for each product. We have close to 400 unique product names in our dataset, which would result in a learned embedding matrix $E \in \mathbb{R}^{400 \times d_{model}}$. If a new product appears with the same name, it already has a known embedding. If it has a new name, the embedding matrix can be expanded as explained. We saw that treating each product independently as done in the entity embeddings experiment easily resulted in overfitting. Treating it based on its name instead could increase the model performance as seen when using character-level or subword-level processing, resulting in products with the same name being treated the same.

It is also possible to use embeddings for associated entities, such as which store or category a product belongs to. If the model also would learn an embedding for each store, performance might be improved as different stores also exhibit different sales patterns. The suggestions presented in this section can be considered as future work.

5.7 Sustainability Aspects

Training a machine learning model can take a lot of time and thus also electricity, especially for big models with large amounts of data as in our case. We have used an NVIDIA K80 GPU which has a peak power consumption of 300W [18]. Our very rough estimate for how long we have been training our models on this GPU during this thesis is 60 hours. This results in $60 * 300 = 18000$ Wh of energy used, assuming the GPU would always run on full power which is not the case in practice. We assume a rough estimate of 20 kWh used in total to also account for the CPU and other components.

For Nordic countries, the average emissions from electricity is 125g CO_2eq/kWh [14]. This results in 2.5kg CO_2eq emissions for training the models. In comparison, 1kg of beef results in 23-39kg CO_2eq emissions [12]. So if our model starts being used in reality, it can have a net positive climate impact already from the first package of beef that it manages to prevent from becoming waste.

As the grocery stores learn to order fewer products, the wholesalers will receive smaller orders. The patterns of the orders will also change, pushing demand variations up the supply chain. In the end, producers can avoid producing what would later become waste.

5.8 Future Work

We have touched upon suggestions for future work in the preceding sections. These include preventing the fixed-number entity embeddings from overfitting by embedding the name of a product, making use of the product category or store in different ways, further investigations into variable selection weights interpretability, and dynamically expanding the fixed-number entity embeddings. In addition to these, we present a few more ideas in this section.

One obvious recommendation is experimenting with more data when several years worth of data has been collected. Perhaps this will make it easier for the model to learn predictions that have yearly fluctuations, such as for seasonal products. We then expect the day of the year feature to have a larger variable selection weight for such products.

Different number of heads with multi-head attention could also be explored to see if this would be beneficial for the model. For instance, one head could focus on attending to prior weekdays and another head for attending to previous promotion periods, done separately compared to self-attention with a single head.

As mentioned previously, further tuning hyperparameters and applying different amounts of regularization (such as dropout) to models that overfitted could also be investigated.

6
Conclusion

In this thesis, we have shown that our proposed variable-number entity embeddings can improve the forecasting performance of a TFT, also allowing it to be used in scenarios where the number of entities frequently changes such as when working with food products. We also saw how the TFT easily overfitted using traditional fixed-number entity embeddings, which is not seen using our solution. The choice of using character-level or subword-level processing of the product name does not seem to have a significant impact on the performance.

The embeddings learned using fixed-number entity embeddings seem very reasonable, with products from the same category being grouped together indicating that the learned embeddings encapsulate the sales patterns of each product. The grouping of product categories is not seen when using variable-number entity embeddings, unless including the product category classification loss which does not seem to have an effect on forecasting performance. While the TFT is able to improve its performance using these embeddings, their applicability for other tasks remains to be explored.

Interpretability insights are obtained from both the TFT attention weights and variable selection weights. Overall, it is clear that the model learned a weekly pattern in sales in these sets of experiments. Historical promotion periods are useful for making predictions during a new promotion. Holiday seasonality is also captured by the model for seasonal related products that are sold year-round. Furthermore, when product names are being input to the model, variable selection weights become more distinguishable between products. Overall, the attention weights and historical feature weights appear to be more interpretable compared to future feature weights and real-valued features also appeared to be slightly more interpretable than categorical features.

Existing forecasting models used in the industry have not been evaluated. Instead, we developed two different baseline models and show that the TFT can achieve significantly higher performance than these simpler models. As prior work has shown that the TFT can outperform other state-of-the-art models, we believe that it has a good chance of also outperforming existing models used in the industry, making it a viable tool for combating food waste through sales forecasting.

Bibliography

[1] Jay Alammar. The Illustrated Transformer. https://jalammar.github.io/illustrated-transformer/. [Online; accessed 1-June-2021].

[2] Tova Andersson and Sandra Stålhandske. *Matavfall i Sverige*. Naturvårdsverket, 2018.

[3] A. D. Brébisson, Étienne Simon, Alex Auvolat, P. Vincent, and Yoshua Bengio. Artificial neural networks applied to taxi destination prediction. *CoRR*, abs/1508.00021, 2015.

[4] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[5] Guillaume Chevalier. The LSTM Cell. https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg. [Online; accessed 2-June-2021].

[6] Cheng Guo and Felix Berkhahn. Entity embeddings of categorical variables. *CoRR*, abs/1604.06737, 2016.

[7] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9:1735–1780, 1997.

[8] Ian London. Encoding cyclical continuous features - 24-hour time. https://ianlondon.github.io/blog/encoding-cyclical-features-24hour-time/. [Online; accessed 6-May-2021].

[9] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. Time2Vec: learning a vector representation of time. *arXiv*, abs/1907.05321, 2019.

[10] Bryan Lim, Sercan Ö. Arik, Nicolas Loeff, and T. Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *arXiv*, abs/1912.09363, 2019.

[11] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: A survey. *Philosophical Transactions of the Royal Society A*, 379, 2021.

[12] Livsmedelsverket. Kött och chark. https://www.livsmedelsverket.se/

matvanor-halsa--miljo/miljo/miljosmarta-matval2/kott. [Online; accessed 2-December-2020].

[13] Helmut Ltkepohl. *New Introduction to Multiple Time Series Analysis*. Springer, 2007.

[14] Fredrik Martinsson, Jenny Gode, Jenny Arnell, and Jonas Höglund. *Emissionsfaktor för nordisk elproduktionsmix*. IVL Svenska Miljöinstitutet, 2012.

[15] Tomas Mikolov, Kai Chen, G. S. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[16] T. Mori and Naoshi Uchihira. Balancing the trade-off between accuracy and interpretability in software defect prediction. *Empirical Software Engineering*, 24:779–825, 2018.

[17] Naturvårdsverket. Fakta om matavfall. https://www.naturvardsverket.se/Sa-mar-miljon/Mark/Avfall/Matavfall/, 2020. [Online; accessed 25-November-2020].

[18] NVIDIA. TESLA K80 GPU ACCELERATOR - Board Specification. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317-001-v05.pdf. [Online; accessed 27-May-2021].

[19] Mattias Eriksson och Ingrid Strid. *Svinnreducerande åtgärder i butik*. Naturvårdsverket, 2013.

[20] Christian A. Scholbeck, Christoph Molnar, Christian Heumann, Bernd Bischl, and Giuseppe Casalicchio. Sampling, intervention, prediction, aggregation: A generalized framework for model-agnostic interpretations. *Communications in Computer and Information Science*, page 205–216, 2020.

[21] Rico Sennrich, B. Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, 2016.

[22] TensorFlow. Time series forecasting. https://www.tensorflow.org/tutorials/structured_data/time_series. [Online; accessed 30-April-2021].

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, page 5998–6008, 2017.

[24] Wikipedia contributors. Autoregressive model — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Autoregressive_model. [Online; accessed 5-May-2021].

A

Time2Vec Code

The following piece of Python code is our implementation of Time2Vec as a neural network layer in TensorFlow/Keras. As discussed in Section 5.3, we believe this code is incorrect as it did not achieve the expected results.

```
import tensorflow as tf
from tensorflow.keras.layers import Layer


class Time2Vec(Layer):
  def __init__(self, dim):
    super(Time2Vec, self).__init__()
    self.dim = dim

  def build(self, input_shape):
    self.w_linear = self.add_weight(name='w_linear',
                                    shape=(1,),
                                    initializer='uniform',
                                    trainable=True)
    self.bias_linear = self.add_weight(name='bias_linear',
                                       shape=(1,),
                                       initializer='uniform',
                                       trainable=True)

    self.w_periodic = self.add_weight(name='w_periodic',
                                      shape=(self.dim - 1,),
                                      initializer='uniform',
                                      trainable=True)
    self.bias_periodic = self.add_weight(name='bias_periodic',
                                         shape=(self.dim - 1,),
                                         initializer='uniform',
                                         trainable=True)

  def call(self, x):
    t2v_linear = self.w_linear * x + self.bias_linear
    t2v_periodic = tf.math.sin((x * self.w_periodic) + self.bias_periodic)
```

```
    t2v = tf.concat([t2v_linear, t2v_periodic], axis=-1)

    return t2v

  def compute_output_shape(self, input_shape):
    return (input_shape[0], self.dim)

  def get_config(self):
    config = super(Time2Vec, self).get_config()
    config.update({
      'dim': self.dim
    })
    return config
```