Temporal Fusion Transformer for time series forecasting

Marius Kielhöfer

Applied Mathematics Thesis
In collaboration with Kraft Heinz

Temporal Fusion Transformer for time series forecasting

by

Marius Kielhöfer

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Friday 14 July 2023.

Student number:	5125502	
Project duration:	February 1, 2023 – July 15, 2023	
Thesis committee:	Piao Chen,	TU Delft, supervisor
	Tom Vroegwijk,	TU Delft

Abstract

The ability to accurately forecast sales volumes holds substantial significance for businesses. Current classical models struggle in capturing the impact of different variables upon the sales volume. These machine learning models are also not applicable to more than one specific product. The Temporal Fusion Transformer (TFT) is implemented to address these issues. The TFT is a powerful tool designed for time series forecasting. TFT leverages deep neural networks and self-attention to capture variable dependencies across all time steps, providing temporal context for improved accuracy. The developed TFT model showcases its efficacy in accurate sales forecasting. By considering both past and future variables, TFT generates predictions with errors of 30%. Moreover, the interpretability of the model highlights the importance of variables such as Covid lockdown periods and product distribution. The scalability of the TFT model allows it to generate forecasts for every product-retailer combination, making it a versatile tool for businesses. As a multi-horizon forecaster, TFT incorporates both past and future variables to generate predictions. This characteristic makes it an excellent candidate for evaluating the impact of changes in future inputs controlled by the business, such as pricing and distribution strategies.

In Layman’s Terms

The Temporal Fusion Transformer (TFT) is a forecasting tool that can predict future sales accurately. It uses a method called deep neural networks to understand how different factors like time, Covid lockdowns, and distribution affect sales of a certain product. By constantly learning and improving, it determines accurate forecasts. The produced model is successful and predicts sales for Beans, Ketchup and Soup categories. The model can make calculations for all combinations of products and retailers, showing that it can be used for many different situations. The current error rate is 20-30%, with potential for improvement. The TFT is useful because it can look at both past and future factors. This means that if the business would like to test out a new price for a product, that the model would determine an accurate forecast for the future sales volume (from that price change). In summary, the TFT model is a powerful forecasting models that predicts future sales. It learns from data and can explain its’ calculated results. With more computing power, it can be used for more types of products and regions. It’s a helpful tool for businesses to understand how changes affect sales.

Contents

1 Introduction 1

2 Data 3
- 2.1 Data Excerpt . . . 3
- 2.2 Notation . . . 5

3 Background Information 7
- 3.1 Neural Networks . . . 7
- 3.2 Self Attention . . . 11
  - 3.2.1 Multi-head Attention. . . . 12

4 TFT Architecture 15
- 4.1 TFT's Interpretable Multi-head Attention . . . 15
- 4.2 Variable Selection Blocks . . . 16
  - 4.2.1 Gated Residual Networks . . . 17
  - 4.2.2 Transformation of Variables . . . 19
  - 4.2.3 Computations of Variable Importances . . . 20
- 4.3 Static Covariate Encoders . . . 21
- 4.4 Temporal Processing . . . 21
  - 4.4.1 LSTM's . . . 21
  - 4.4.2 Static Enrichment . . . 22
  - 4.4.3 Attention Applied . . . 22

5 Methodology 25
- 5.1 Hyperparameter Tuning . . . 25
  - 5.1.1 Sequential Model-Based Global Optimization . . . 25
  - 5.1.2 Pruning Mechanism . . . 26
  - 5.1.3 Hyperparameter Results . . . 26
- 5.2 Training the Model . . . 28

6 Results 31
- 6.1 Validation Forecasts. . . . 31
- 6.2 Variable Importances . . . 35

7 Conclusion 39

8 Extensions 41
- 8.1 Hierarchical Learning . . . 41
- 8.2 Fractal Interpolation . . . 41

A Appendix A: Full list of variables TFT evaluates importances on 45

B Appendix B: Full list of Hyperparameters with definition 47

C Appendix C: Code to initialise dataset into dataloader 49

Bibliography 51

1 Introduction

In recent years, the field of machine learning has experienced significant advancements, propelling its impact across various domains. Notably, the paper by Vaswani et al. in 2017, titled "Attention is all you need," [30] accelerated development in the area of natural language processing (NLP), through the introduction of the attention mechanism. This mechanism has demonstrated remarkable efficacy in enhancing the performance of machine learning models. Machine learning models that make use of attention mechanisms are called transformers.

One domain in which transformers have recently been applied to are time series. Time series data represents a sequence of observations recorded at regular intervals, such as weekly sales volume for a specific product. The revenue generated from product sales constitutes a company's financial success. Consequently, businesses strive to gain insights into factors influencing sales to inform strategic decision-making and optimize their operations. Significant resources are invested each year in developing effective strategies, promotions, and other approaches aimed at increasing sales volumes.

Constructing a forecasting model with high accuracy presents a considerable challenge. Traditional algorithms have been widely employed in the past, however these have fallen short in various areas. These algorithms do not account for time varying inputs, but rather assume recursively fed future inputs [32]. Many time series forecasting algorithms have only managed to output one step forecasts, meaning that no long term dependencies could be captured.

Recent advancements in artificial intelligence have introduced a novel approach: the temporal fusion transformer (TFT) . The TFT represents a specialized transformer architecture tailored specifically for time series analysis [20]. By making use of self-attention mechanisms, the TFT effectively captures and identifies complex temporal patterns across multiple time steps. The TFT hereby captures long and short term dependencies with its superior attention based architecture. In comparison to similar work [27], the TFT has focused on providing improved interpretability whilst also maintaining better forecasting performance. TFT allows users to understand and analyse global behaviours, identify persistent patterns such as seasonality, as well as behaviour at different points in time [20]. As a result, it surpasses the performance of conventional time series models, yielding more accurate sales forecasts [32]. Other attention based based models have shown promising results and are based on LSTM networks. These however, have failed to incorporate static variables (see section 2) into its' architecture, unlike TFT [27].

The TFT has been tested to different time-series problems already. Wu et al [32] tested the TFT for interpretable wind speed predictions. The paper showcases that the TFT provides stable and accurate predictions for a variable with random characteristics such as wind speed.

In our work we test the capabilities of the temporal fusion transformer for forecasting future sales of specific products. This research aims to enhance sales forecasting accuracy and provide valuable insights to the business, facilitating better decision-making and optimization. The scope of this work will remain in the context of the sales for specific products of Kraft Heinz. The data is provided by Kraft Heinz. Due to the confidentiality of the data, it will not be discussed in detail. Possible extensions to improve the results beyond the TFT architecture are considered in the last chapter. With several altercations and additional steps, it is deduced that the TFT performs well in the area of sales.

2 Data

To provide further context throughout the whole paper, extracts of the data are discussed in the context of the TFT. The data provided by Kraft Heinz will be explained in this chapter, where it is then translated into appropriate notation.

2.1. Data Excerpt

The aim of the final prediction model is to output a forecast for the sales volume of a certain product. The model will determine which factors are important to determine future sales volumes, and how much of a role they play. An example of the data is shown in table 2.1.

Table 2.1 Example of (dummy) data

Product	**Week**	**Sales Volume**	**Average Price (Pounds)**	**TDP (Distribution)**
Ketchup product A	3/10/2020	2000	3.00	0.90
	3/17/2020	2300	2.79	0.91
	3/24/2020	2200	2.85	0.93
Quarter	**Sales In The Subcategory (Ketchup)**	**Last 12 Week % Of Subcategory Sales Volume**	**Competitor 1 Sales Volume**	**Average Price Of cannibalizer 1**
1	20000	0.11	3000	2.00
1	25000	0.12	3300	1.79
1	23000	0.11	2200	1.85
Month	**Holiday Flag**	**Tactic**	**TDP Of Competitor 1**	**New Covid Cases**
March	1	20% off	0.80	100
March	0	20% off	0.75	120
March	0	20% off	0.78	80

It can be seen that there is weekly sales data, with several different other variables. This type of data is shown on the product level, in this case Ketchup product A. This data is available for every type of product Heinz sells, for every retailer in the UK. This table could represent the sales for Ketchup product A in Tesco's, but there is also data for Asda. Within the product level of Ketchup product A, there are different barcodes. These are slightly different variations of Ketchup product A. This is why there is an average price, since this represents the average price of all the barcodes in Ketchup product A. The TDP represents the percentage of stores that the product can be found in. A TDP of 0.90 showcases that product A is prevalent in 90% of all the stores in that region (UK). See appendix A for the full list of variables that are in the data.

The target variable is set for what is trying to be forecasted, which in this case will be the sales volume at a product level. When preparing the data for the TFT, the other variables will be split into four different groups; static data, known future inputs (this will be split into categorical and continuous) and past inputs.

The static data is the data that always remains the same. This is, for example, in which subcategory that specific product is sold in (Ketchup, Mayonnaise, Soup, etc.).

The next category of variables are the known future inputs. In order to make future forecasts, this has to be based on certain variables (from the future). The model will select and then make predictions based on these variables. In some of the variables, the future inputs can be decided. For example, the company can decide to run a specific type of promotion, or decide to change the price of the product (this is what the entire tool is for, in order to see how promotions and changes in price could change the sales volume). However, there are other important variables that are used as input for the forecast that there is no control over. An example of this would be the price of a competitor product. This will never be known to the business, but is still an important factor for the sales of the business' own product.

For this reason, the "future competitor price" will be assumed to be the last known value. This will be assumed for all similar types of variables. This is the best estimation for these values without making another separate model for each variable. A possible solution to this issue whereby the model could be expanded will be discussed in later sections.

Finally, the last category of values are classified as past inputs. These are the values of variables in the past, so what the past price or volume itself was. These past inputs are necessary in order for the TFT to find patterns and dependencies between variables, so that it can predict the sales volume. Based on the interaction between past and future inputs, is how the transformer will decide the weights (importances) of every variable.

Let us look back at table 2.1 and classify these variables. Assuming a model is trained from 3/10/2020 to 3/24/2020, and we want to make a prediction for one week ahead. The categorization of the table will be as follows:

Table 2.2 Annotated dummy data

Product	**Week**	**Sales Volume**	**Average Price (Pounds)**	**TDP (Distribution)**
Ketchup product A	3/10/2020	2000	3.00	0.90
	3/17/2020	2300	2.79	0.91
	3/24/2020	2200	2.85	0.93
	3/31/2020	$\hat{y}_{t+1}$	2.70	0.93
Quarter	**Sales In The Subcategory (Ketchup)**	**Last 12 week % Of Subcategory Sales Volume**	**Competitor 1 Sales Volume**	**Average Price Of cannibalizer 1**
1	20000	0.11	3000	2.00
1	25000	0.12	3300	1.79
1	23000	0.11	2200	1.85
1	23000	0.11	2200	1.80
Month	**Holiday Flag**	**Tactic**	**TDP Of Competitor 1**	**New Covid Cases**
March	1	20% off	0.80	100
March	0	20% off	0.75	120
March	0	20% off	0.78	80
March	0	50% off	0.78	80

Table 2.3 Meanings Of Colours

Colour code	**Meaning**
🟥	Target Variable
🟩	Past Inputs
🟦	Static Data
🟧	Known Future Inputs
🟪	Estimated Future inputs (by last known)

From table 2.2 it can be seen that most of the variables will be estimated by last known, whereas some will be known by the company (such as prices of their own products). The cannibalizer price, for example will be known since a cannibalizer is another product from the company. This could be Ketchup product B for instance. If the price of Ketchup product B falls, it could mean a decrease in the sales of Ketchup product A.

The structure of such a time series forecasting mechanism is referred to as multi horizon forecasting, which is illustrated in figure 2.1. The reason why this mechanism is so useful is because the model takes inputs from every time horizon, and not only the past.

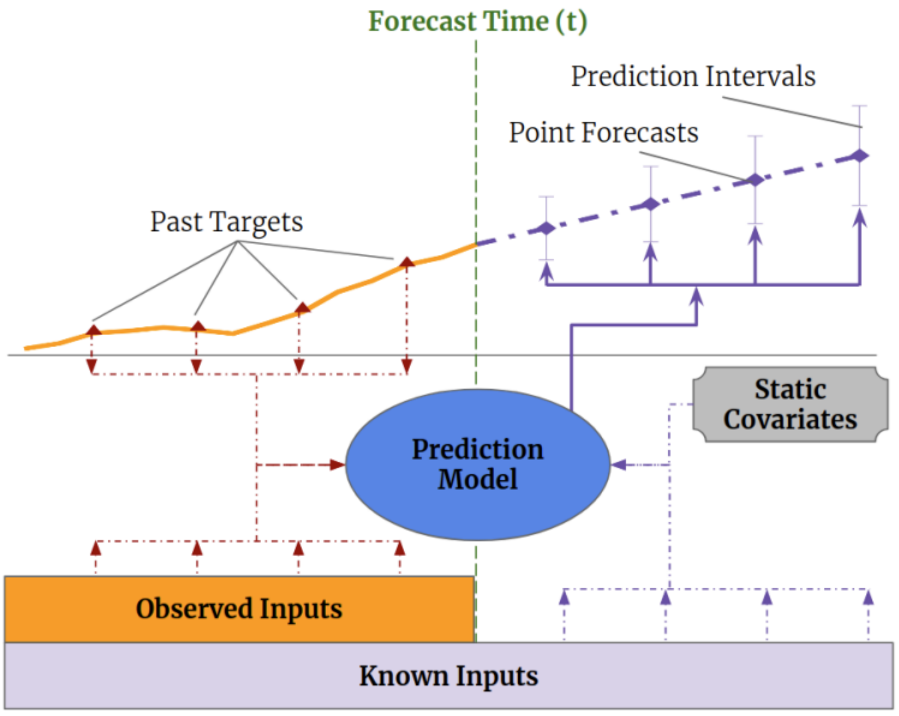


Figure 2.1 Model inputs. The figure shows how the model takes in information from three different sources: past targets, past inputs and known future inputs. This produces future point forecasts, which are accompanied by prediction intervals. Taken from [20].

2.2. Notation

The notation of the problem will be as follows.

We take I as the unique retailer-product combinations; (Asda, Tesco, Sainsbury's). Only retailers in the UK will explored for this problem. For these retailers there are different products. For example the classic 400ml bottle, or a different 200ml bottle. The TFT will not only look at Ketchup but also at products in other subcategories (Soup and Beans). For confidentiality reasons, these products will be labelled alphabetically as Ketchup product A, Ketchup product B, Beans product A etc.

It is important to define this since there are different levels in the data. The model could also focus on forecasting for the whole subcategory, and not forecast at the individual product level.

This is why we start by defining our predictor variable as the sales volume of a specific product at a specific retailer at time t:

$$y_{i,t} \in \mathbb{R}$$

Every retailer-product combination I has a set of static data.

$$\boldsymbol{s}_i \in \mathbb{R}^{m_s}$$

Where m_s is the length of the static data vectors for the specific model. Other than static features, there are time independent features, varying at time t for every i:

$$\chi_{i,t} = \boldsymbol{x}_{i,t} \in \mathbb{R}^{m_x}$$

The predictor is then defined as a function of the aforementioned variables. The architecture of the TFT opts to adopt quantile regression (see example in figure 2.2.). This is done to be able to analyse possible best and worst case scenarios of a prediction and form prediction intervals as is illustrated in figure 2.1.

$$\hat{y}_i(q,t,\tau) = f_q(\tau, y_{i,t-k:t}, \boldsymbol{x}_{i,t-k:t+\tau}, \boldsymbol{s}_i) \tag{2.1}$$

Where $\hat{y}_{i,t+\tau}(q,t,\tau)$ is the qth sample predicted sales volume for a specific product-retailer combination i, τ steps ahead of time t. In this case, all the past data that is used to train the model ranges from $t-k$ to t. The forecasts are made from t to $t+\tau$ which is the last point that is forecasted. All forecasts for τ_{max} are outputted simultaneously.

k represents the finite lookback window;

$$y_{i,t-k:t} = \{y_{i,t-k}, \ldots, y_{i,t}\}$$

This window represents all past sales volume data that will be inputted into the model. $t-k$ would be the time step for the first data point, whilst t is the most recent time step.

On the other hand, the time varying variables range from time $t-k$ to $t+\tau$ (since future values are known or estimated):

$$\boldsymbol{x}_{i,t-k:t+\tau} = \{\boldsymbol{x}_{i,t-k}, \ldots, \boldsymbol{x}_{i,t}, \ldots \boldsymbol{x}_{i,t+\tau}\}$$

The first time step for the variable is at $t-k$. The last known or estimated variable value is at final forecast time $t+\tau$.

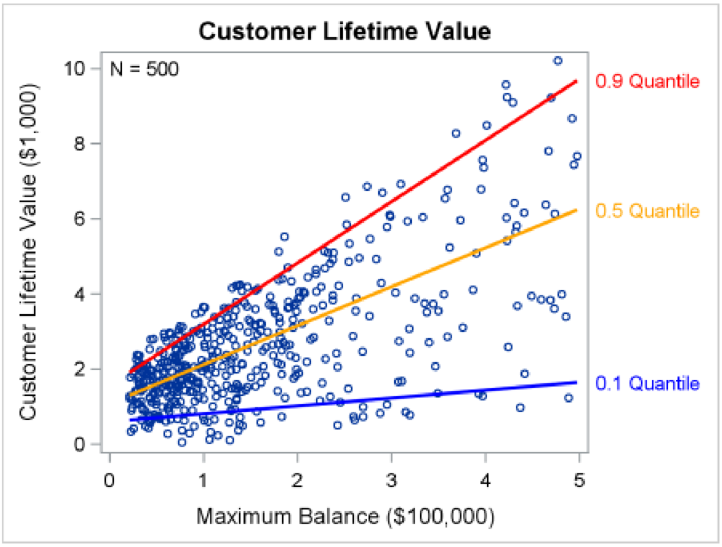


Figure 2.2 An example of quantile regression that showcases the 0.1, 0.5 and 0.9th quantile for customer lifetime value against maximum balance, taken from [24].

3 Background Information

There are two important concepts that the building blocks for the TFT. These are necessary for understanding the TFT. First, the concept of neural networks is explored, after which self attention is discussed.

3.1. Neural Networks

Before diving into the specifics of the TFT, it is crucial to have a good grasp of neural networks. The TFT, being a Deep Neural Network (DNN)-based model, builds upon fundamental principles and concepts of neural networks. By understanding the mechanics of neural networks, the unique architecture and innovations of the TFT will be understood with greater clarity. This will provide a foundation for comprehending design choices and training procedures specific to the TFT, enabling a deeper understanding of its' forecasting capabilities.

The objective of our problem is to get a forecasted value for the sales volume of a certain product. For simplicity sake, let's assume that the sales volume only depends on three (future) variables: sales volume in the subcategory, TDP and new covid cases. The aim is to create a model that has as input these three variables and outputs the sales volume.

The neural network learns to do this by the user inputting a *training dataset*. This would contain information on past data, more specifically columns for the three variables with their respective sales volume. The training data is defined from the date 3/10/2020 to 3/24/2020 for an i, say Asda Beans Product C;

Table 3.1 Training Data for Beans Product C

Week	**Subcategory sales**	**TDP**	**New covid cases**	**Sales volume**
3/10/2020	20000	0.90	100	1500
3/17/2020	22000	0.89	200	2300
3/24/2020	21500	0.93	120	2500

From this training data a neural network learns two different parameters: weights and biases. The weights signify the importance of a particular variable, whilst a bias is the preference learned during training. This bias works similar to how human bias works. One person might enjoy running the rain, whilst the next may not. This is exactly how the machine learns as well, but in this case it is the bias in certain contexts. The neural network might have a bias of preferring TDP as the most important variable for a certain situation. In contrast to humans however, the neural network has a bias term that corrects for the bias that occurred during training. After these weights and biases are computed, a simple weighted sum is carried out to compute the output [22].

$$Sw_S + Tw_T + Cw_C - b_1 = SV \tag{3.1}$$

For S the subcategory sales, T is TDP, C are covid cases and SV is the output, which is the forecasted sales volume. w_{SS}, w_T, w_C are specific weights determined by the network for each variables and b_1 is the bias term calculated.

The mechanism that makes neural networks special are the hidden layers. The hidden layers are nodes between the input and the output. Instead of directly calculating the weighted sum from the input to the output, an intermediary weighted sum is calculated from the input to a separate node in the hidden layer. The hidden layer contains as many nodes as the user of the neural networks specifies. The output of this weighted sum then represents the value of a certain node in the hidden layer. From all the values of nodes in the hidden layer, another weighted sum is executed to calculate the final output. All of the weights and biases of these weighted sums are parameters that the network calculates.

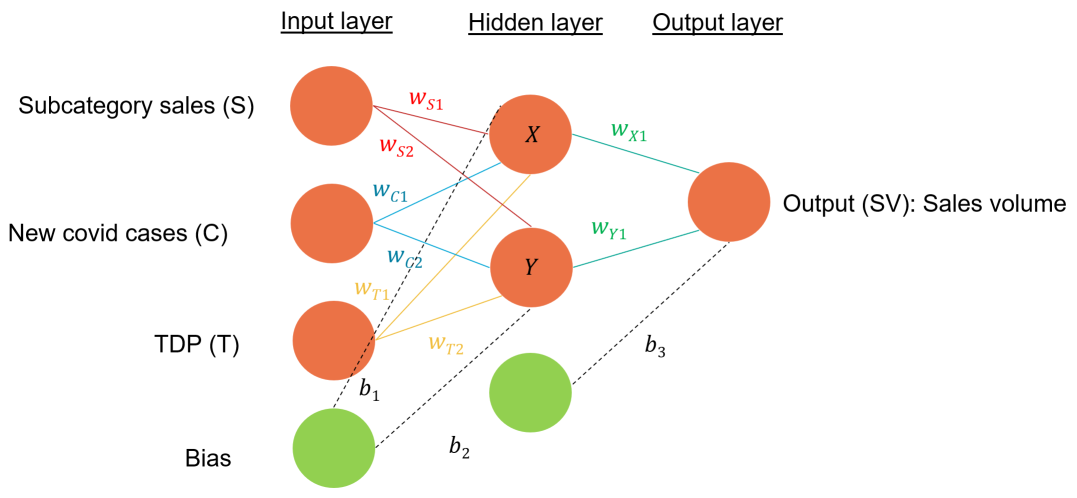


Figure 3.1 Example of how a neural network could look like for our problem. There are three inputs for the three different variables and one output node for the sales volume. In between, a decision was made to use two hidden layers nodes. At every node where a weight is computed, there is another node that determines the networks bias.

A weighted sum is then computed to the separate hidden layer nodes with an activation function, which is applied to equation 3.1:

$$f(Sw_{S1} + Tw_{T1} + Cw_{C1} - b_1) = X \tag{3.2}$$

$$f(Sw_{S2} + Tw_{T2} + Cw_{C2} - b_2) = Y \tag{3.3}$$

Where equation 3.2 computes the output for the first hidden layer node and equation 3.3 for the second. Theses values of the hidden layer nodes are then used to compute the output in a final weighted sum (with new weights and biases determined for the hidden layer values):

$$g(Xw_{X1} + Yw_{Y1} - b_3) = SV \tag{3.4}$$

To enhance the performance of the neural network, activation functions are implemented [26]. These are represented by f and g in the equations above. Typically, different activation functions are used in the hidden layer and in the output layer. An activation function is a function that can detect more complex and non-linear patterns. When looking activation functions in the hidden layer, the output of the activation function decides if a neuron in the network is “activated”. During the training process a threshold emerges (decided by the network). If the output from the activation function exceeds a certain threshold, the neuron will be activated and be used further, otherwise it will be pruned. An activation function in the output layer usually determines a final transformation for the output. If the neuron is activated it will pass its’ output towards the next weighted sum. In the example illustrated in figure 3.1, it was assumed that both outputs from the weighted sum exceeded a threshold so both nodes were activated. Since both were activated, both outputs were then used in the equation 3.4 to determine the final output. After the weighted sum with the bias is computed, the output of this is passed through an activation function (which is decided beforehand). This will then be the value of the node that it is passed through, as seen in the previous equations.

For a specific example, let’s take a look at the sigmoid activation function. The sigmoid/logistic function is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

This function takes any real number inputs and gives an output in the range of zero to one. The higher the input value is, the closer the output will be to one. This activation function is most commonly used when constructing a model that has as a final output a probability [4]. For example, take a neural network model that attempts to predict whether an item of clothing is real or fake based on various different features. The output would be a probability whether an item is fake. The sigmoid activation function would be well suited for this model since it gives outputs between zero and one, which can be interpreted as probabilities. A threshold might have been defined as 0.5, where an output of the sigmoid function being greater than 0.5 might classify a certain item as fake. The sigmoid function is not linear, so it is also able to capture complex patterns between input features and the output as a probability.

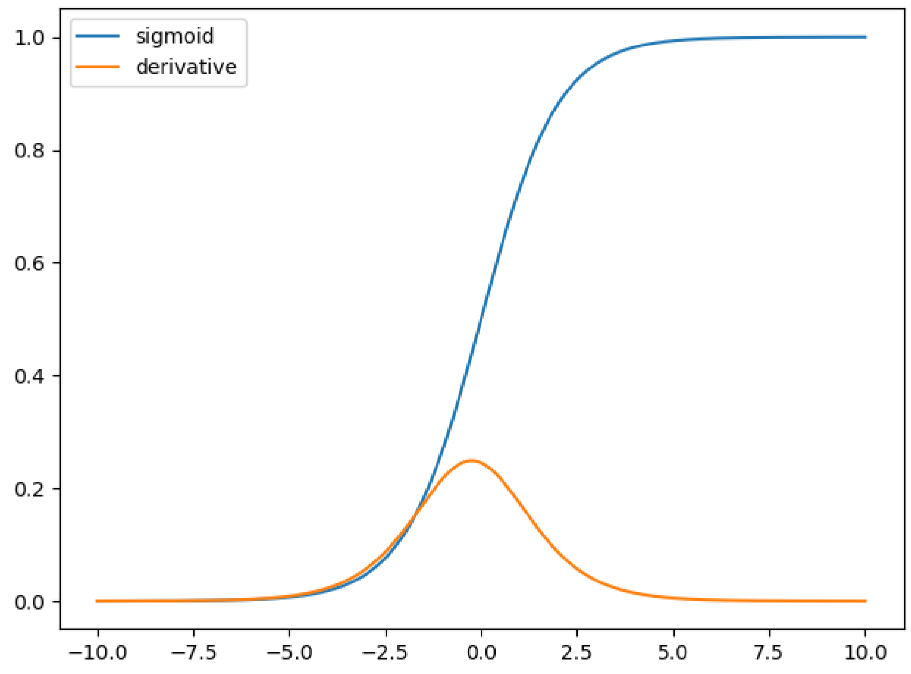


Figure 3.2 Graphic representation of the sigmoid activation function and its' derivative.

The weights and bias parameters are evident to have the most influence on the output. At the start of the training, the model tries random weights but learns quickly from the training data. At the start of training the model might still think that an extremely high price leads to high sales of the product, which does not make sense. The parameters are constantly updated to maximise performance with a process called backpropagation.

After the network makes a certain prediction, a loss (cost) function is used to evaluate performance [33]. This is decided by the user and could be represented by mean squared error (equation 3.5), mean absolute squared error, or other functions. It compares the output that the network predicted with the actual output from the training data. For example, it would predict the sales volume for 02/24/2020 with its' calculated parameters and compare it with the actual value as seen in table 3.1.

$$L = MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.5}$$

Many neural networks use gradient based optimization to update weights and biases, such as gradient descent. The goal of updating the parameters is to ultimately minimise the loss function. After every iteration (one run through the network), the network undergoes backpropagation to alter the weights, let us take a look at the algorithm of gradient descent that is summarized in equation 3.6 [25].

$$\boldsymbol{a}_{n+1} = \boldsymbol{a}_n - \gamma\nabla F(\boldsymbol{a}_n) \tag{3.6}$$

Where $\boldsymbol{a}_{n+1}$ represents the output of the loss function after iteration $n+1$ and $F(\boldsymbol{a}_n)$ is the calculation of the output of the neural network with weights associated to iteration n. γ represents the learning rate. The learning rate decides the scale of how the model learns. A high learning rate causes high steps which makes rapid changes to the model, which may be inaccurate. A small learning rate makes smaller changes so the model learns at a slower rate. This may improve accuracy but also takes longer to train. The learning rate is a hyperparameter [1] that is defined by the user. When taking fixed learning learning rates, the output of the loss function a will converge to a local minima [29].

Let us take the example from before; we use the sigmoid activation function on hidden layer nodes X,Y and assume there is no activation function to the output layer. After an iteration n, the network has determined that the output of the sigmoid function on Y has not exceeded the threshold. Now Y will not be used further. We take the loss function as mean squared error (eq. 3.5) and apply gradient descent to start backpropagation to the weights calculated in iteration n.

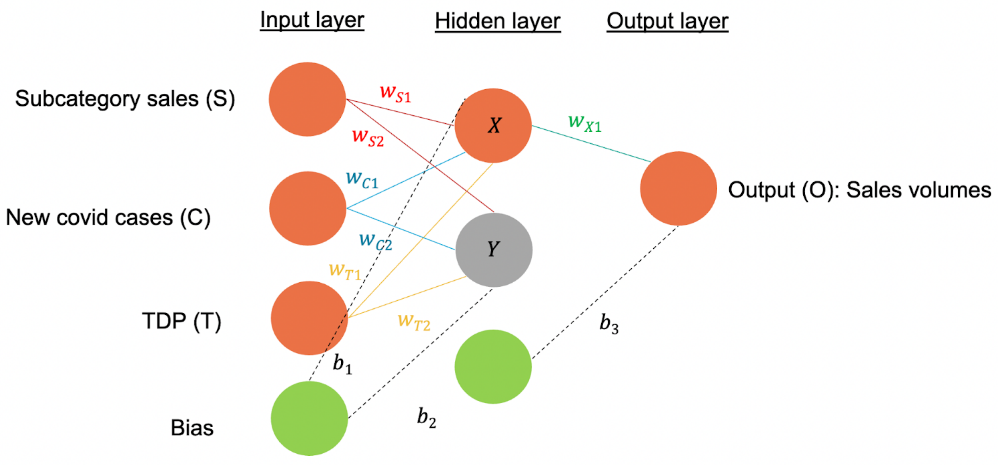


Figure 3.3 Neural network architecture for iteration n.

The calculations for this network are:

$$\sigma(Sw_{S1} + Tw_{T1} + Cw_{C1} - b_1) = X \tag{3.7}$$

$$Xw_{X1} - b_3 = \hat{y}_i \tag{3.8}$$

So,

$$\hat{y}_i = (\sigma(Sw_{S1} + Tw_{T1} + Cw_{C1} - b_1)) * w_{X1} - b_3 \tag{3.9}$$

Backpropagation is commenced for predicted sales volume $\hat{y}_i$

Gradient of the loss is first computed:

$$\frac{\partial L}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i)$$

The gradient of the loss with respect to the first weight w_{X1} is computed:

$$\frac{\partial L}{\partial w_{X1}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{X1}} = 2(\hat{y}_i - y_i) \cdot w_{X1}$$

Similarly, gradient of loss with respect to b_3:

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial b_3} = -2(\hat{y}_i - y_i)$$

Gradients of losses for weights w_j in hidden layer for $j = S1, T1, C1$, and $J = S, T, C$:

[1]A hyperparameter is a parameter that is manually defined by the user, and not computed see section 5.1 for a breakdown and appendix B for the full list of hyperparameters.

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_j} = 2(\hat{y}_i - y_i) \cdot \frac{\partial \hat{y}_i}{\partial X} \cdot \frac{\partial X}{\partial w_j} =$$

$$2(\hat{y}_i - y_i) \cdot \sigma(X) \cdot (1 - \sigma(X)) * w_{X1} * J$$

Whereas gradient for b_1:

$$\frac{\partial L}{\partial b_1} = -2(\hat{y}_i - y_i) \cdot \sigma(X) \cdot (1 - \sigma(X)) w_{X1}$$

Then the weights and biases are updated with learning rate γ:
For the weights $w_{i,n}$ in iteration n, where $i = X1, S1, T1, C1$, weights for iteration $n+1$ are:

$$w_{i,n+1} = w_{i,n} - \gamma \frac{\partial L}{\partial w_{i,n}}$$

and biases $b_{m,n}$, $m = 1,3$

$$b_{m,n+1} = b_{m,n} - \gamma \frac{\partial L}{\partial b_{m,n}}$$

This is then recursively repeated until convergence.

This method of gradient descent causes some issues with certain functions. Taking a look the gradient of the sigmoid function in figure 3.2, it can be seen to get quite flat. For any values outside of (-3,3) the function will have very small gradients. As the gradient approaches 0, a network ceases to learn, since this implies there are no changes possible anymore to improve the model. This is called the vanishing gradient problem. Hence why it's important to choose the correct combination of activation functions and loss functions.

3.2. Self Attention

To introduce the concept of self attention, it will be explained at the hand of large language models (LLM). LLM's are machine learning models that aim to understand the language of a user and respond in a human like manner. One could think about ChatGPT.

Let us compare two different input sentences that a LLM would want to understand:

1: The doors of the jaguar were open

2: The jaguar roared in the night

How would the model understand that the first sentence refers to the jaguar car and the second sentence refers to a jaguar as the animal? This is where self-attention comes into play. In the model, every word has a vector of **query**, **key** and **value** assigned. When the model starts to analyse the input sequences, a query is activated to check what the relation of that word is with all the others. This query is compared with the key of the other words, and if there is a match, a vector value is outputted. This will create the context of the situation.

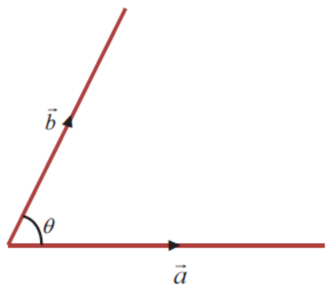
Let us take the word *doors* as an example in the first sequence. The model matches the query of the word *doors* with the words *the, of, jaguar, were* and *open.* It finds no match of the key vector of the first couple of words but when the query of doors is compared to the key of jaguars, there is a match. Since there is a match, the vector value informs the model that the first sentence refers to the jaguar being a car, and not the animal.

In terms of how this match is decided, it is done so with a similarity index. This is usually represented by a dot product, more specifically the scaled dot product.

For query $\boldsymbol{q}$, key $\boldsymbol{k_i}$ for variable/word i. Similarity index is defined as the scaled dot product;

$$s_i = f(\boldsymbol{q}, \boldsymbol{k_i}) = \frac{\boldsymbol{q}^T \boldsymbol{k}_i}{\sqrt{d}} \tag{3.10}$$

Where d is the dimension of the key $\boldsymbol{k}_i$



$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos\theta$$

Figure 3.4 Alternate method for calculating dot product of vector $\boldsymbol{a}$ and $\boldsymbol{b}$.

The logic behind this is that if two vectors are aligned and parallel they would be completely similar. The angle between them would be $\theta = 0$ and the output of the similarity index would be $s_i = \frac{|\boldsymbol{q}||\boldsymbol{k}_i|}{\sqrt{d}}$. If the vectors are pointing in complete opposite directions, then $\theta = \pi$ and $s_i = -\frac{|\boldsymbol{q}||\boldsymbol{k}_i|}{\sqrt{d}}$, which would be the lowest possible value for the similarity index. The scaling of the dot product is done to ensure both vectors are the same magnitude when calculating.

After the similarity indices are calculated, the attention weights for each query-key pair is determined with the softmax function.

$$a_i = \frac{exp(s_i)}{\sum_{j=1}^{n} exp(s_j)} \tag{3.11}$$

The softmax function takes a vector of real numbers as input and transforms it into a probability distribution over the classes. This exponentiates each element of the input vector and normalizes the results, ensuring that the sum of the probabilities is equal to one.

These attention weights are then inputted into a weighted sum with the corresponding value vectors to calculate the general attention value.

$$attention\ value = \sum_{i=1}^{m} a_i \boldsymbol{v_i} \tag{3.12}$$

Summarising self attention; a query is compared to all the keys to determine their similarity. The keys with the highest similarity are assigned greater weights, and a weighted combination of the corresponding values is used to generate the output [30]. This makes it more likely (given the probabilistic distribution) that the correct value vector (car or animal) is outputted to the model.

3.2.1. Multi-head Attention

The TFT makes use of multi-head attention. In multi-head attention, projections are made for the Key, Query and Value vectors. The vectors are multiplied by different projection matrices in order to split into different heads. Depending on the size of the amount of heads that is used, the input sequence is projected to that size. For example, if three heads are used, the projection matrices would split the input sequence into three separate parts. For every head, different projection matrices are used. This mechanism splits the input sequence into different heads so the transformer can handle different sequences simultaneously. Some heads could focus on long term patterns and the other attention heads look at short term dependencies [1].

These matrices are determined by neural networks. Weights and biases are computed that represent these projection matrices and gradient based optimization algorithms are used to find the optimal values of the matrices to minimize error and losses (of the end results). These projection matrices are referred to as the individual weights for each head.

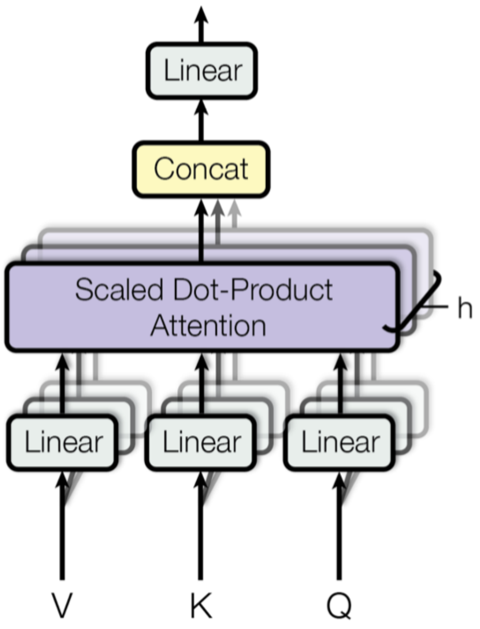


Figure 3.5 Multihead attention with three heads. The *linear* blocks represent the projection matrices [30].

The scaled dot product and softmax function is applied to every head, resulting in weighted attention values for each head. The result of these heads are concatenated into one tensor, where it is then passed through another projection to transform the results into the appropriate dimension. This projection matrix is a learnable parameter that is determined in the same way as the previous projection matrices. This is given by:

$$MultiHead(\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}) = Concat(H_1,\dots,H_{mH})\boldsymbol{W_O} \quad (3.13)$$
$$H_h = Attention\left(\boldsymbol{q}\boldsymbol{W}_q^{(h)},\boldsymbol{k}\boldsymbol{W}_k^{(h)},\boldsymbol{v}\boldsymbol{W}_v^{(h)}\right)$$

Where $\boldsymbol{W_O}$ is the projection matrix for the output and $\boldsymbol{W}_q^{(h)},\boldsymbol{W}_k^{(h)},\boldsymbol{W}_v^{(h)}$ are projection matrices for input query, key and value for every head h up to head H_{mH}.

For example, let three different projections be made for three heads. Then three different (scaled dot product) attention values for each projection are computed. Finally, these attention values are concatenated, after which a final projection is done to receive multi-head attention.

Masked multi-head attention is when multi-head attention is applied whilst incorporating a mask. The mask nullifies the probabilities of certain values, preventing them from being selected. This ensures that during output generation, the output only depends on previous outputs and not on future outputs, eliminating dependencies on outputs that have not yet been produced [30].

In our scenario, say that forecasts are produced until March 2024. It is essential to ensure that the forecasted sales for March 2024 are based solely on the sales volume preceding that period, rather than relying on data from a later month, such as November 2024. By applying masking, any connections that could introduce unnecessary dependencies are eliminated, allowing for smoother and more reliable forecasting.

This is ensured by the following calculation:

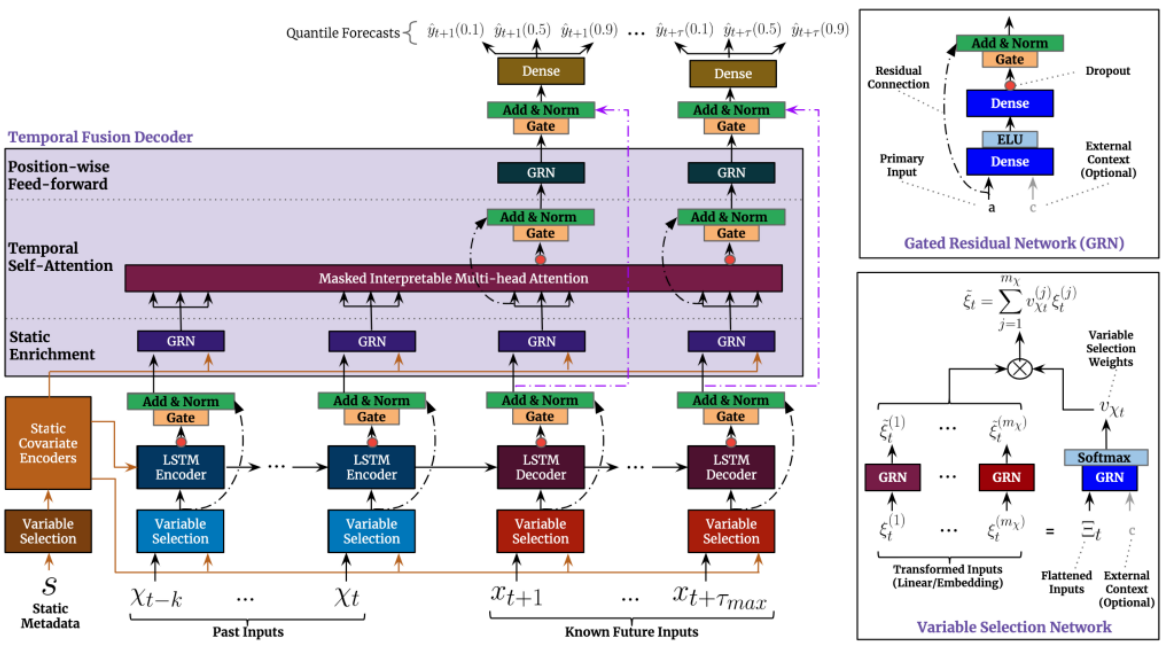
$$Masked\ attention(\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}) = softmax(\frac{\boldsymbol{q}^T\boldsymbol{k}_i + M}{\sqrt{d}}) * \boldsymbol{v_i} \quad (3.14)$$

Where M is the mask matrix that has entries zero or $-\infty$. Zero for entries that are considered when computing attention, and $-\infty$ for the outputs in the future that are not to be considered, this will make these outputs zero since $exp(-\infty) = 0$ in equation 3.13.

The amount of heads that are used is a hyperparameter that is decided upon before training the model.

4
TFT Architecture

Now that the prerequisites are clarified, the specifics will be explored. The architecture of the Temporal Fusion Transformer is shown in figure 4.1. The individual components will be analysed one at a time. This architecture was designed and explained in Lim et al [20].



Quantile Forecasts
$\hat{y}_{t+1}(0.1)$ $\hat{y}_{t+1}(0.5)$ $\hat{y}_{t+1}(0.9)$ ... $\hat{y}_{t+\tau}(0.1)$ $\hat{y}_{t+\tau}(0.5)$ $\hat{y}_{t+\tau}(0.9)$
Dense
Add & Norm
Gate
Temporal Fusion Decoder
Position-wise Feed-forward
GRN
Temporal Self-Attention
Masked Interpretable Multi-head Attention
Static Enrichment
Static Covariate Encoders
LSTM Encoder
LSTM Decoder
Variable Selection
s
Static Metadata
χ_{t-k} ... χ_t
Past Inputs
x_{t+1} ... $x_{t+\tau_{max}}$
Known Future Inputs
Residual Connection
Dropout
Dense
ELU
Primary Input
External Context (Optional)
a
c
Gated Residual Network (GRN)
$\tilde{\xi}_t = \sum_{j=1}^{m_\chi} v_{\chi_t}^{(j)} \tilde{\xi}_t^{(j)}$
Variable Selection Weights
v_{χ_t}
$\tilde{\xi}_t^{(1)}$ ... $\tilde{\xi}_t^{(m_\chi)}$
Softmax
GRN
$\xi_t^{(1)}$ ... $\xi_t^{(m_\chi)}$ = Ξ_t
c
Transformed Inputs (Linear/Embedding)
Flattened Inputs
External Context (Optional)
Variable Selection Network

Figure 4.1 Architecture for the Temporal Fusion Transformer; a deep neural network [20].

There are several different major components that make up the architecture of the TFT; *Gating Mechanisms, Variable Selection Networks, Static Covariate Encoders, Temporal Processing* and *Prediction Intervals.*

Part of the temporal processing component area will be discussed first since it represents the core of TFT. As seen in figure 4.1, this is where the crux of the TFT takes place. At the centre of temporal processing is attention. This is where the TFT employs a newly designed type of attention; interpretable attention.

4.1. TFT's Interpretable Multi-head Attention

Whilst the transformer architecture described in the Vasmani et al [30] has been widely used in many transformer based models, there is an important distinction with the TFT. The end goal of the TFT is to be able to tell the business what the upcoming sales volumes will be. In order to do this, it is crucial that the tool is able to explain its' forecasts. It is illogical to push the business to start using the TFT just because it is correct, whilst not being able to explain why. This is why in addition to implementing masked multi-head attention, the TFT utilises a modified interpretable multi-head attention. As the name suggests, it makes the outputs of the TFT interpretable, which enhances the explainability of such a tool [20]. This gives TFT the edge over other attention based models that do not let the user interpret the attention values [27].

Instead of concatenating the attention of every single head H_h as in the previous chapter, additive aggregation of all heads is implemented. In this sense, the importance of specific features can be deciphered. In multi-head attention the values in each head are different, so the attention weights do not indicate a particular features' importance after concatenation. Equation 3.13 is replaced by:

$$
\begin{aligned}
InterpretableMultiHead(\boldsymbol{q},\boldsymbol{k},\boldsymbol{v}) &= \boldsymbol{\hat{H}W_O} \qquad (4.1)\\
\boldsymbol{\hat{H}} &= \tilde{A}(\boldsymbol{q},\boldsymbol{k})\boldsymbol{v}\boldsymbol{W_v},\\
&= \left\{\frac{1}{H}\sum_{h=1}^{mH} A\left(\boldsymbol{q}\boldsymbol{W}_q^{(h)},\boldsymbol{k}\boldsymbol{W}_k^{(h)}\right)\right\}\boldsymbol{v}\boldsymbol{W}_v,\\
&= \frac{1}{H}\sum_{h=1}^{mH} Attention\left(\boldsymbol{q}\boldsymbol{W}_q^{(h)},\boldsymbol{k}\boldsymbol{W}_k^{(h)}\boldsymbol{v}\boldsymbol{W}_v\right)
\end{aligned}
$$

This way the attention weights are summed up for every attention head and averaged out. Each attention head is still able to learn different temporal patterns, however now a common set of input features is used for each head. This common set of value features is shared among the matrix $\boldsymbol{W}_v$. Which is not the same for every head in the standard multi-head attention. This makes the transformer able to represent its reasoning. The question is however, if this impacts performance or not. The whole point of using different heads is that different temporal patterns are identified. Now that the heads are sharing values it may make it more challenging to inspect different patterns. In this interpretable multi attention, the attention weights of every head are also averaged out instead of being concatenated. This way, the importance of a particular head is not captured.

Self-attention in the TFT is utilised in the temporal aspect, to find all possible long and short term patterns of the time series. This comes in useful for capturing seasonal effects. Let us take the sale of a ketchup bottle as an example. Let us assume, for instance, that during the summer months people tend to eat more ketchup since there are more outdoor barbecues. At the same time, the sales might be going down when the temperature is lower; in the winter months. This may not be a simple linear relationship though. There may be an optimal temperature range which causes sales to increase, and there may be other ranges where the temperature will not affect the sales at all. In the case where a forecast is made in this optimal temperature range, there will be more weight assigned to the temperature- the model will pay more attention to it. The self-attention part comes from the fact that the model knows that it has to pay more attention by iterating over past data variables- it tells itself the importance of all the variables. There will be a higher attention score towards temperature in the case of optimal temperature range.

The self-attention mechanism in the temporal fusion transformer allows the model to dynamically adjust the importance of different features at each time step, based on the patterns it has learned from the input data. This allows the model to capture complex temporal dependencies and make more accurate predictions.

Now that the core of a transformer has been explained, the architecture of the TFT will be explained from the start. Starting with the blocks for variable selection.

4.2. Variable Selection Blocks

These variable selection blocks are what the name suggests; these networks select the relevant input variables for each time step. Whilst there can be a long list of variables input in the TFT, it is up to the TFT to choose the most relevant and important ones. This is also necessary to remove unwanted noise and signals from variables that are not important. Doing the variable selection at the start of the process allows the forecasting process to become smoother since there will be less signals to interpret. Since the TFT will be applied to a real life data set, these variable selection blocks are crucial in the entire process. Real datasets are often noisy and can be extremely large. In our case, the amount of variables that will be passed into the TFT exceeds 100 (as seen in appendix A).

The variable selection blocks can be seen to be applied to the three different categories of variables that were discussed in previous sections; static data, past inputs and known future inputs (figure 4.1). Looking deeper into the variable selection networks, a block “GRN’ can be seen. This block decides which variables are selected and which are left out.

4.2.1. Gated Residual Networks

GRN is short for Gated Residual Network. A GRN is a type of neural network that incorporates different components of recent advancements of neural networks. The GRN makes use of ELU; the exponential linear unit activation function [6].

Similar to other linear units (LU’s), the ELU alleviates the vanishing gradient problem. ELU’s use the identity for positive values; this prevents gradients from becoming too small during backpropagation since the gradients will be non-zero for positive values. What makes the ELU special is that it also outputs negative values. The advantage of having negative values is that the function can provide more informative gradients and help the network with negative correlations, causing faster learning. The ELU was designed so that the mean number of activations of all the neurons in the layer are zero, and it does this by accommodating these negative values. The reason this is so important is because it improves a phenomenon known as bias shift.

The ELU function is given by:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ a(\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \tag{4.2}$$

The ELU activation function and its’ derivative is plotted in figure 4.2 for $a = 1$.

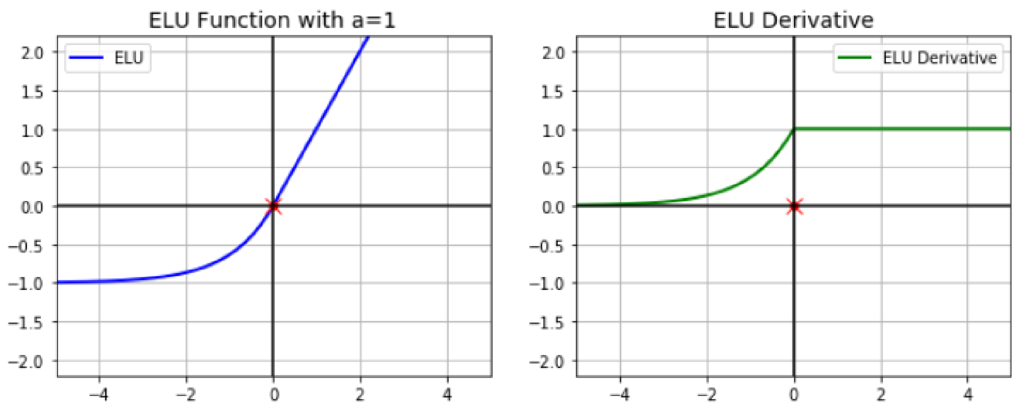


Figure 4.2 ELU activation function and its’ derivative.

Bias shift is when the mean activation of neurons in a layer significantly deviates from zero. Learning parameters with a bias shift causes a bias shift in the next layer. During learning, the gradients calculated will also be imbalanced. This can lead to inconsistent updates of the weights which hinder the learning process. Furthermore, imbalanced activation numbers causes the model to learn at a slower rate [16].

The reason why bringing the mean activation closer to zero enables faster learning, is by decreasing the gap between the normal gradient and the unit normal gradient. After every backpropagation in a neural network, the weights and specific biases calculated, are adjusted for every neuron or unit i. The unit normal gradient correction is applied to the weight update for a specific unit. Proofs and further details can be read in Clevert et al [6].

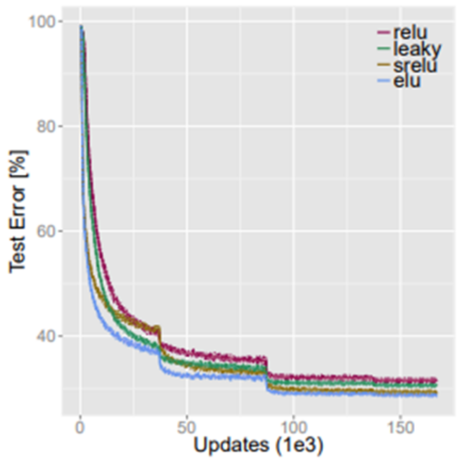


Figure 4.3 From this figure it is evident that the ELU has better test error when compared to the other linear units [6].

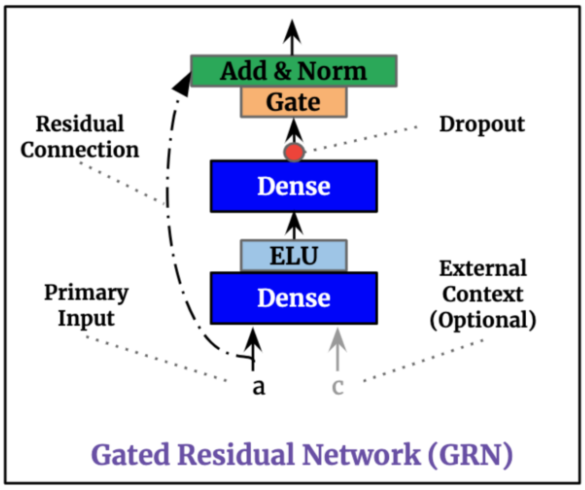


Figure 4.4 Architecture of the Gated Residual Network [20].

Looking at the architecture of the GRN in figure 4.4, the formula is as follows:

$$GRN_{\omega}(\boldsymbol{a}, \boldsymbol{c}) = LayerNorm\big(\boldsymbol{a} + GLU_{\omega}(\boldsymbol{\eta}_1)\big) \tag{4.3}$$

$$\boldsymbol{\eta}_1 = \boldsymbol{W}_{1,\omega}\boldsymbol{\eta}_2 + \boldsymbol{b}_{1,\omega}$$

$$\boldsymbol{\eta}_2 = ELU(\boldsymbol{W}_{2,\omega}\boldsymbol{a} + \boldsymbol{W}_{3,\omega}\boldsymbol{c} + \boldsymbol{b}_{2,\omega})$$

The input is define as the vector $\boldsymbol{a}$, whilst an optional context vector, $\boldsymbol{c}$ can also be added (this can provide supplementary information which enhances the networks capabilities) . ω is an index to denote weight sharing. Starting from the bottom, the ELU function is used on the weighted sum of the weights and biases that are estimated when the neural network runs. This ensures that when the weighted sum is larger than zero it acts as the identity function. If the weighted sum would be smaller than zero it generates a constant output and would output linear behaviour.

The output of the ELU function is then used as an input in the calculation of $\boldsymbol{\eta}_1$. $\boldsymbol{\eta}_1$ corresponds to a vector of nodes. This $\boldsymbol{\eta}_1$ is then passed through another activation function, namely the gated linear unit (GLU). The GLU is used to control the flow of information [3]. As the name suggests it acts as a gate, and decides which information to pass through to the next layer. This is where selection takes place. In the case of variable selection, the GLU decides which variables are worth to keep, and which ones are best to leave behind. The GLU is defined by:

$$GLU_{\omega}(\gamma) = \sigma(\boldsymbol{W}_{4,\omega}\gamma + \boldsymbol{b}_{4,\omega}) \odot (\boldsymbol{W}_{5,\omega}\gamma + \boldsymbol{b}_{5,\omega}) \tag{4.4}$$

It can be seen to take the sigmoid activation function, which outputs a certain probability of the weighted sum in question to be activated; this is the gate vector. Then an element wise Hadamard product ⊙ is taken to decide which information is passed through the gate. The sigmoid function assigns a probability of how likely information manages to pass through. The reasoning behind implementing this GLU is to control how much the GRN contributes to the original input vector $\boldsymbol{a}$. For example, if the outputs of the GLU are all close to zero, information surpasses this layer entirely.

Before the gated connection, there is a section on dropout. Dropout is when random inputs are removed, so that the neural network does not depend too much on specific inputs. The amount of random inputs that are removed- the dropout rate, is another hyperparameter [9].

After applying the gate, the final step is to take a Layernorm of the output. The use of this is to normalize the final output, which ensures smoother running and stabilizes the training process. There are other methods of normalization such as weight normalization and batch normalization but this method has been proven to speed up the training the most [17].

The final piece of a GRN is a residual skip connection. This residual skip connection is a shortcut for the input to flow directly to the output. It is necessary when the input undergoes many transformations in a deep neural network, such as here. Due to all these transformations there is a possibility that important details in the input data are lost. Adding this skip connection allows the network to combine the transformed input with the original input which, has been shown to improve the networks ability to learn complex patterns [28].

Sometimes adding many complex layers does not improve performance. So if the skip connections performs better than a complicated sequence of mechanisms, the sequence is scrapped, and skipped so that there is less complexity in the model.

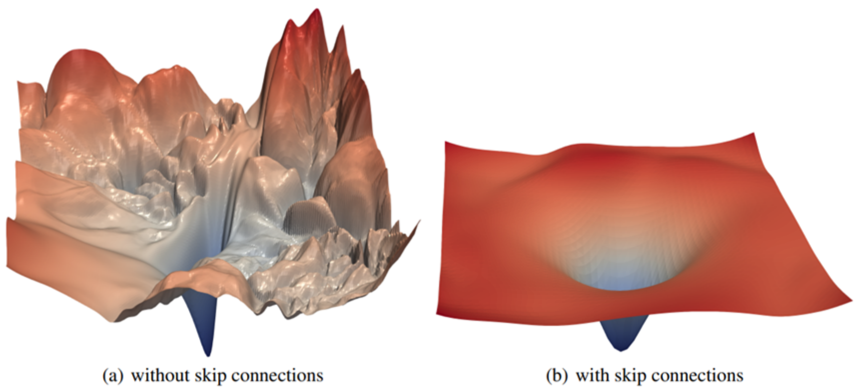


Figure 4.5 The loss surfaces of ResNet-56, a convolutional neural networks architecture, with and without skip connections. The addition of skip connections is seen to smoothen the surface out drastically [18].

4.2.2. Transformation of Variables

Before the list of variables (static, future or known) are entered into the GRN, they are first transformed. All variables need to be in the form of vectors that are consistent with the dimension of our general model. This dimension of our model represents the size of the hidden layers within our neural networks, which runs throughout the entire architecture. This hidden layer size is also a hyperparameter that is decided by the user before training. The hidden layer size represents the number of neurons in every layer of the hidden layer.

In order to transform the input into the desired dimension, a similar approach is done to the projection matrices as in section 3.2.1. Certain weight matrices are approximated and learned by the model during training, which are then matrix multiplied by the input to receive our input in the desired dimension. For the continuous variables this is just a simple multiplication. For the categorical variables it is a bit more complicated; entity embeddings are required.

Entity embeddings is the process of representing categorical variables in continuous vectors, so they may be used in future processes. This is commonly used in LLM's [7]. These models have to interpret different words. Without being able to read worded inputs as signals (in terms of numbers), it would be challenging for a LLM to actually work. Whilst embedding, relationships and similarities are captured between categories, that are represented in a vector form. The way it does this is by mapping similar categories close to each other in the embedding space. The general method is that categorical variables are encoded. For example for different retailers, Asda could be encoded as 1, Tesco as 2 etc. These numbers here however, are nominal numbers, since Tesco is not greater than Asda even though $2 > 1$. These entity embeddings are again trained with neural networks. The mapped embeddings of these words are represented by weights of specific layers. For our context of sales, different categories could be mapped to certain points in a map. The neural network then optimises so that the distances between similar categories are reduced. In the example of the retailers, the neural networks will calculate weights so that Lidl and Aldi are mapped closer in the embedding space. This is because Lidl and Aldi are both low-end retailers and are most similar to each other. The size of the vector that is created to capture similarities is another hyperparameter that is specified at the start of training the TFT. A larger embedding size could capture more accurate similarities, but may take the model longer to train as well. Higher level details about entity embeddings are further explained in Guo and Berkhahn [11].

4.2.3. Computations of Variable Importances

After the variables are transformed to the appropriate form, each variable *separately* enters a GRN. This is done to process each variable for non linear trends in *one variable for different time steps.* The output of this would be the processed variable vector with weights shared across all time steps. This is illustrated on the left hand side of figure 4.8.

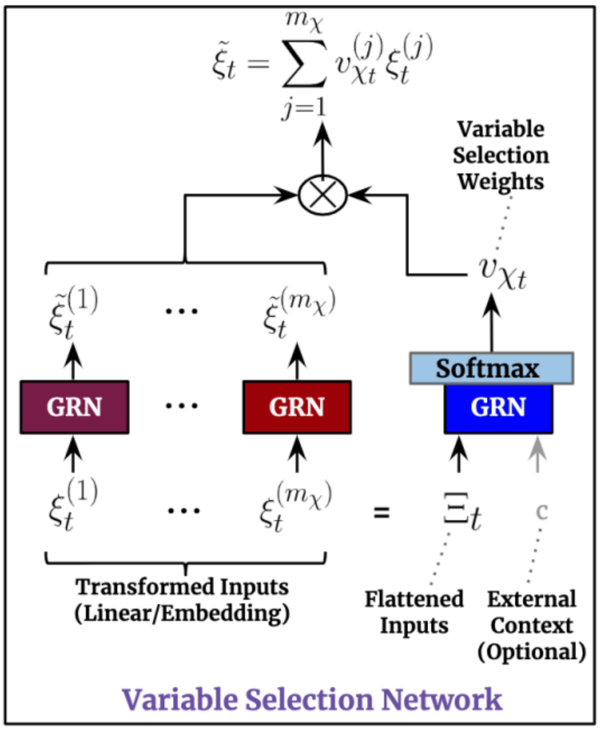

$\tilde{\xi}_t = \sum_{j=1}^{m_\chi} v_{\chi_t}^{(j)} \tilde{\xi}_t^{(j)}$
Variable Selection Weights
v_{χ_t}
$\tilde{\xi}_t^{(1)}$
$\tilde{\xi}_t^{(m_\chi)}$
Softmax
GRN
GRN
GRN
$\xi_t^{(1)}$
$\xi_t^{(m_\chi)}$
=
Ξ_t
c
Transformed Inputs (Linear/Embedding)
Flattened Inputs
External Context (Optional)
Variable Selection Network

Figure 4.6 Architecture of the Variable Selection network [20].

The projected or embedded variables $\boldsymbol{\xi}_t^{(1)^T},\ldots,\boldsymbol{\xi}_t^{(m_\chi)^T}$ are flattened into one vector

$$\Xi_t = \left[\boldsymbol{\xi}_t^{(1)^T},\ldots,\boldsymbol{\xi}_t^{(m_\chi)^T}\right]^T$$

This flattened input is then run through a gated residual network with an additional context vector c_s. This represents the static variables, and is retrieved from the static variable encoder (see next section). This context vector is omitted when calculating variable weights for the embedded static variables. Whilst the projected/embedded variables separately enter GRN's, the variable selection weights are calculated by taking a softmax of the output of the flattened vector Ξ_t; this assigns probabilities to each output:

$$\tilde{\boldsymbol{\xi}}_t^{(j)} = GRN_{\tilde{\xi}(j)}\left(\boldsymbol{\xi}_t^{(j)}\right) \tag{4.5}$$

Are the outputs for the projected/embedded variable j through separate GRN's at time t.

$$\boldsymbol{v}_{Xt} = Softmax\left(GRN_{v_X}\left(\Xi_t, \boldsymbol{c}_s\right)\right) \tag{4.6}$$

Are the variable selection weights at time t. The processed features from equation 4.5 are weighted by the weights determined in equation 4.6.

$$\tilde{\boldsymbol{\xi}}_t = \sum_{j=1}^{m_X} v_{Xt}^{(j)} \tilde{\boldsymbol{\xi}}_t^{(j)} \tag{4.7}$$

Are the final processed variables at time t, where $v_{Xt}^{(j)}$ is the j^{th} element of vector $\boldsymbol{v}_{Xt}$.

4.3. Static Covariate Encoders

The aforementioned context vector $\boldsymbol{c}_s$ is calculated in the static covariate encoder block. Unlike other forecasting architectures, the TFT takes static data into account. One way it does this is the embedding that was done to the variables in the previous section. The second approach is that it creates context vectors based on static data that are used throughout the entire architecture. There are four different context vectors in this block:

$\boldsymbol{c}_s$ is the vector created to provide context for temporal variable selection,

$\boldsymbol{c}_c$ and $\boldsymbol{c}_h$ are vectors for local processing of temporal features,

and $\boldsymbol{c}_e$ is the vector for the enriching of temporal features with static information.

After the static variable features are selected in the variable selection block, these four different vectors are created through four separate GRN encoders. The model splits the static variables into these different categories again with the use of neural networks and backpropagation. By backpropagating through the entire architecture, TFT adjusts these context vectors to maximize performance.

The $\boldsymbol{c}_s$ vector is used to help in selecting relevant variables that have an impact on the sales volume. For example, this vector assists in capturing product specific seasonality trends. If Ketchup product A's winter period has a large impact on the sales, but Ketchup product B's winter period does not have an impact on its respective sales. This is what this context vector would help capture.

The $\boldsymbol{c}_c$ and $\boldsymbol{c}_h$ vectors are for local processing of temporal features. These vectors help in identifying patterns within specific static data. For example, these would be used to process how different promotions affect the sales volume. This vector helps in capturing patterns or trends for specific product retailer combinations. The main difference between these vectors and $\boldsymbol{c}_s$ is that these help process temporal features whilst $\boldsymbol{c}_s$ helps to select temporal features. The vectors are also for *local* enhancement meaning for a closer neighbourhood of a data point. For example if a temporal feature is being locally processed and is at time step k, processing would occur in relation to data in close time steps such as $k-5$ or $k+5$, and not at long time periods such as $k-100$

Finally, $\boldsymbol{c}_e$ enriches temporal features with static information. It helps capture how specific static variables such as brand impact the sales volume of different products. This context vector is used in the static enrichment layer to add context to the input one last time before attention is applied.

These context vectors are then used in further operations of the TFT architecture.

4.4. Temporal Processing

As the name suggests, in the temporal processing blocks is where the variables and the inputs are processed.

4.4.1. LSTM's

The next blocks after variable selection are the LSTM- encoders and decoders. The LSTM encoders encode the past inputs, whilst the decoders decode the known future inputs. The purpose of these LSTM's is for locality enhancement. If a significant change happens in a pattern, this is often identified in relation to the previous point. For example, if the sales volume would be 3000, 3400, 3200 then 1000. This sudden drop is only identified as a sudden drop due to its surrounding values. Thus, the purpose of these encoders and decoders is for the TFT to learn local patterns, which have a significant impact on the forecasts. LSTMs are a type of recurrent neural network designed to handle sequential data (long short-term memory) [13]. LSTMs are special since they can remember information for long periods of time. The core to a LSTM is the cell state, in the cell state is where all the information that stays constant is stored. This is how the LSTM remembers information. Throughout the LSTM, several activation functions are implemented that update the cell state by forgetting some information and adding new information. LSTM's also have a hidden state, which is the

LSTM's memory at a specific time step. This is based on the current input and the previous hidden state. The cell state in our case is initialized by the context vector $\boldsymbol{c}_c$, whilst the hidden state is initialised by $\boldsymbol{c}_h$ so that static metadata can influence local processing. The specifics of LSTMs are left to the reader and is explored in Hochreiter and Schmidhuber [13].

From equation 4.7, the transformed past inputs $\tilde{\boldsymbol{\xi}}_{t-k:t}$ are inputted into the LSTM encoder, whilst the transformed future inputs $\tilde{\boldsymbol{\xi}}_{t+1:t+\tau_{max}}$ are inputted into the LSTM decoder. This generates a set of uniform temporal features that are indexed by a position index n:

$$\boldsymbol{\phi}(t,n) \in \left\{\boldsymbol{\phi}(t,-k),\ldots,\boldsymbol{\phi}(t,\tau_{max})\right\} \tag{4.8}$$

$$\text{for } n = -k,\ldots,\tau_{max}$$

A skip connection is also added over this layer of LSTMs to improve performance. After running $\phi(t,n)$ through a GLU to get to the desired output, this is combined with the output from the skip connection. This is then run through a Layernorm to get to the desired format, where we arrive at the final output of this layer:

$$\tilde{\boldsymbol{\phi}}(t,n) = LayerNorm\left(\tilde{\boldsymbol{\xi}}_{t+n} + GLU_{\tilde{\boldsymbol{\phi}}}\left(\boldsymbol{\phi}(t,n)\right)\right) \tag{4.9}$$

4.4.2. Static Enrichment

After the LSTMs, one final layer is applied before initiating self attention; the static enrichment layer. This layer enriches the temporal features with static metadata. This is done by feeding the set of temporal features through a gated residual network with the addition of the final context vector, $\boldsymbol{c}_e$ to lead the transformer into the right direction.

$$\boldsymbol{\theta}(t,n) = GRN_{\theta}\left(\tilde{\boldsymbol{\phi}}(t,n), \boldsymbol{c}_e\right) \tag{4.10}$$

This output is then grouped into a matrix:

$$\boldsymbol{\Theta}(t) = [\boldsymbol{\theta}(t,-k),\ldots,\boldsymbol{\theta}(t,\tau_{max})] \tag{4.11}$$

4.4.3. Attention Applied

Interpretable masked multi-head attention is then applied on the matrix in equation 4.11. The way this works in line with the queries, keys and values is that these are all represented from the matrix. This way the attention layer can find any pattern; long or short between all variables in relation to the sales volume. This is represented in the following equation:

$$\boldsymbol{B}(t) = InterpretableMultiHead\left(\boldsymbol{\Theta}(t), \boldsymbol{\Theta}(t), \boldsymbol{\Theta}(t)\right) \tag{4.12}$$

Which outputs attention weights for time t at every position index n;

$$\boldsymbol{B}(t) = \left[\beta(t,-k),\ldots,\beta(t,\tau_{max})\right] \tag{4.13}$$

Again, a skip connection to skip the entire section on self-attention is present if the model deems it to be necessary, whilst the weights are sent through another GLU:

$$\boldsymbol{\delta}(t,n) = LayerNorm\left(\boldsymbol{\theta}(t,n) + GLU_{\delta}(\beta(t,n))\right) \tag{4.14}$$

This output of the self attention layer is run through a GRN. Furthermore, another skip connection is added that skips through the entire transformer block. Here again, the weights of the GRN are shared across the entire layer, meaning that each neuron in the layer receives the same weights as inputs. **This promotes sharing of information and is useful since all neurons are attempting to detect similar patterns but in different time regions**.

$$\boldsymbol{\psi}(t,n) = GRN_{\psi}(\boldsymbol{\delta}(t,n)),$$

$$\tilde{\boldsymbol{\psi}}(t,n) = LayerNorm\left(\tilde{\boldsymbol{\phi}}(t,n) + GLU_{\tilde{\psi}}(\boldsymbol{\psi}(t,n))\right) \tag{4.15}$$

Via equation 4.9.
Finally, the quantile forecasts are produced at each future time step τ for desired quantile q:

$$\hat{y}(q,t,\tau) = \boldsymbol{W}_q \tilde{\boldsymbol{\psi}}(t,\tau) + b_q \tag{4.16}$$

Where $\boldsymbol{W}_q, b_q$ are linear coefficients for the specified quantile.

The TFT is optimized through jointly minimizing quantile loss across all quantiles, minimizing the error of every prediction. One epoch is a full run through of the training data. In one epoch different forecasts are made in different batches. From these batches the quantile loss is then calculated. Backpropagation is then done by propagating the joined quantile loss backwards through the network. Each layer in the network receives the gradient from the previous layer and uses it to update its' own parameters. This allows the model to learn from its mistakes and gradually adjust its' parameters to have a better fit. This is done in a similar manner to the calculation in chapter 3, but the loss function (equation 4.17) is the joined quantile loss. In this case the TFT architecture is also a neural network but much more complex than a simple network as worked out in chapter 3.

$$L(\Omega, \boldsymbol{W}) = \sum_{y_t \in \Omega} \sum_{q \in Q} \sum_{\tau=1}^{\tau_{max}} \frac{QL\big(y_t, \hat{y}(q, t-\tau, \tau), q\big)}{M\tau_{max}} \tag{4.17}$$

For QL being quantile loss;

$$QL\big(y, \hat{y}, q\big) = q(y-\hat{y})_+ + (1-q)(\hat{y}-y)_+ \tag{4.18}$$

For $(\cdot)_+ = \max(0,\cdot)$. Ω being the domain of the training data with M samples, and Q the set of output quantiles. When $q = 0.5$ (the median quantile), equation 4.18 is the equation for mean absolute error [31].

To summarise the TFT architecture, the different independent variables first undergo several transformation to go into an appropriate dimension. The importances of these variables are then determined through gated residual networks. That output is then put into LSTM's for local pattern identification, after which it enters the temporal fusion decoder. In the temporal fusion decoder the input is first enriched with static metadata and then sent through the attention mechanism. After several more transformations through GRN's and skip connections, quantile forecasts are produced from the attention values. The TFT model learns by jointly minimizing quantile loss across all quantiles (equation 4.17) during the process of backpropagation.

5 Methodology

The TFT architecture is now applied to the data that was shown in chapter 2. Before applying the architecture however, the hyperparameters are determined.

5.1. Hyperparameter Tuning

In the previous chapters an amount of hyperparameters were discussed. These were to be specified by the user. Instead of guessing values for the hyperparameters, there is a python package that runs iterations to find the most suitable hyperparameters specifically for the users' data. This python package is called Optuna. The Optuna package has two mechanics in which the optimal hyperparameters are chosen. The first mechanic suggests an algorithm to select optimal hyperparameters based on the data. The second mechanic is a pruning algorithm that prunes (gets rid of) trials that will not give optimal performance, in order to save computing time.

Since there are a number of optimization algorithms that Optuna suggests, only one will be explained in detail. There are different types of optimizations algorithms; independent and relational sampling. The difference between them is that relational sampling looks at the concurrent relationships between the hyperparameters. Independent sampling however, looks at the individual hyperparameters. The reason why Optuna is used is because it combines different algorithms together for the best possible outcome. One algorithm could be used for the first 20 trials, whereas another is implemented for the remaining trials. This mixture of algorithms is what makes Optuna the best performer when it comes to hyperparameter tuning. Further details about all the algorithms can be found in Optuna's paper [8].

5.1.1. Sequential Model-Based Global Optimization

One of such an algorithm that is widely used is sequential model-based global optimization (SMBO). This algorithm is used when the evaluation of a function is too expensive to compute every trial, so a surrogate model is introduced to approximate this function [5]. The function in our case would be the quantile loss. There are certain points (hyperparameter combinations) that optimise the surrogate model, which are then proposed as candidates for the true function. This is then repeated until a candidate gives the best evaluation of the true function. This candidate will then represent the chosen hyperparameters [12].

An example would be to use Gaussian processes as the surrogate model. This surrogate model is a model that estimates the performance of the algorithm as a function of the hyperparameters. The Gaussian process is represented by:

$$y \sim N(\mu, \Sigma) \tag{5.1}$$

Where y represents the output of the objective function, which in our case would be the quantile loss. The N represents an N-dimensional multivariate normal distribution. The data is normalized, whilst Σ is the covariance matrix from that data which is represented by:

$$\Sigma_{i,j} = K(x_i, x_j) \tag{5.2}$$

For hyperparameters x_i, x_j and kernel function K. This kernel function has many options, as an example it could be the rational quadratic kernel:

$$K(x_i, x_j) = \sigma^2 \left(1 + \frac{(x_i - x_j)^2}{2\alpha\ell}\right)^{-\alpha} \tag{5.3}$$

Where α , σ and ℓ are the parameters that will be optimised in the process to minimise the objective function (the quantile loss). This surrogate function is updated throughout the whole process of hyperparameter tuning to see which combination of hyperparameters yields the best results. Not all possible combinations of hyperparameters are evaluated, since this would be too computationally expensive. An acquisition function, such as the expected improvement is therefore implemented [12]:

$$EI(x) = \mathbb{E}\left[max\left(0, f(x) - f(\hat{x})\right)\right] \tag{5.4}$$

Where $f(x)$ s the gaussian process' estimation of the function whilst $f(\hat{x})$ are the current best performing hyperparameters. These estimated values of $f(x)$ are computed as follows:

$$p\left(f(x)|x\right) \sim N(\mu(x), \sigma(x)) \tag{5.5}$$

Combining equations 5.4 and 5.5 yields;

$$EI(x) = \int_{f(\hat{x})}^{\infty} p\left(f(x)|x\right) f(x) - f(\hat{x}) df(x) \tag{5.6}$$

This proof is omitted but can be found in Jones et al [14].

On the basis of the expected improvement, the next configuration of hyperparameters are then chosen to test. There are two important areas that have to be kept in mind for this: exploration and exploitation. Exploration is looking for many different values, which may or may not increase performance. Exploitation is when the algorithm stays in a similar range because it knows that those values will yield good results. More will be discussed for our specific results regarding this.

When a fitting hyperparameter combination is evaluated, the surrogate model is updated to incorporate this new information. This process is then repeated: hyperparameters are found that perform best on the surrogate model, which are then properly evaluated on the objective function. After an evaluation the surrogate model is updated to incorporate these results [5]. This is repeated until a specified stopping condition has been met.

5.1.2. Pruning Mechanism

When running a study, some trials are also pruned. This is decided by a variant of the successive halving algorithm [19]. When a trial is underway, intermediate objective values are reported. Based on these values, the trial will either be pruned or will be allowed to keep running (to find better hyperparameters). The idea behind successive halving is that there is a set of hyperparameter configurations, which are then evaluated with the objective function.The bottom half of the worst scoring configurations are then thrown out. This is repeated until one configuration remains [19]. This is computationally not optimal so a variation; asynchronous successive halving is used. This allocates computational resources in an optimal manner to determine whether a trial should be pruned or not. The specific version of asynchronous successive halving that is used in Optuna is explained in their paper [2].

5.1.3. Hyperparameter Results

Optuna is applied to select the hyperparameters using the provided dataset. Whilst not all hyperparameters of the TFT could be optimized at present, a select list of hyperparameters were eligible for optimization through Optuna;

```
{'gradient_clip_val' , 'hidden_size', 'dropout','hidden_continuous_size',
'attention_head_size', 'learning_rate'}
```

gradient_clip_val is the value for gradient clipping. Gradient clipping ensures that the gradient in backpropagation is clipped to a certain threshold to prevent any sudden jumps in the gradient. This value alters the calculated gradient accordingly if the said threshold is exceeded. This ensures smoother backpropagation.

hidden_size and *hidden_continuous_size* are the hidden layer sizes and the hidden layer size for the continuous variables specifically. *dropout* is the dropout rate that was discussed in chapter 4, whilst *attention_head_size* are the amount of heads used for multi-head-attention. Finally, *learning_rate* is the learning rate as discussed in chapter 3.

For these hyperparameters a range was specified for the algorithm to search the best values within that range. The ranges were set as broad as possible (although kept realistic) so that the algorithm may decide for itself what the best values may be.

The Optuna package has a built in visualization tool that allows the user examine specifics of the study. The optimization history of the study is examined:

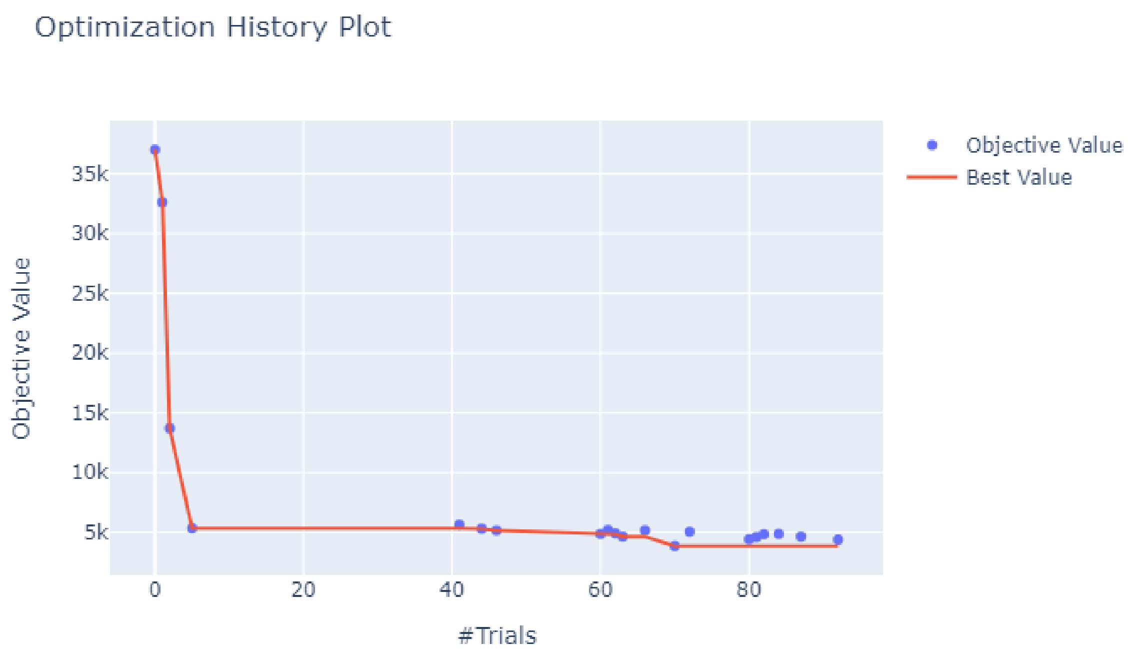


Figure 5.1 Optimization history of the 100 trials that were run to find the best set of hyperparameters.

From this figure it can be seen that in the first 10 trials, the objective value improves at a fast rate, whereas it seems to plateau afterwards. The only major improvement is the best trial at around trial 70. This shows that running a study for more than 100 trials is not necessary since the improvement would be minimal. More than 100 trials would also be computationally expensive since 100 trials already took several hours to complete.

A deep dive can also be done to see in which regions the model is tuned. This is done to check if the model has indeed explored into the full range that was specified, or if it just stuck with a specific configuration that worked. These deep dives also uncover the relationships between certain hyperparameters.

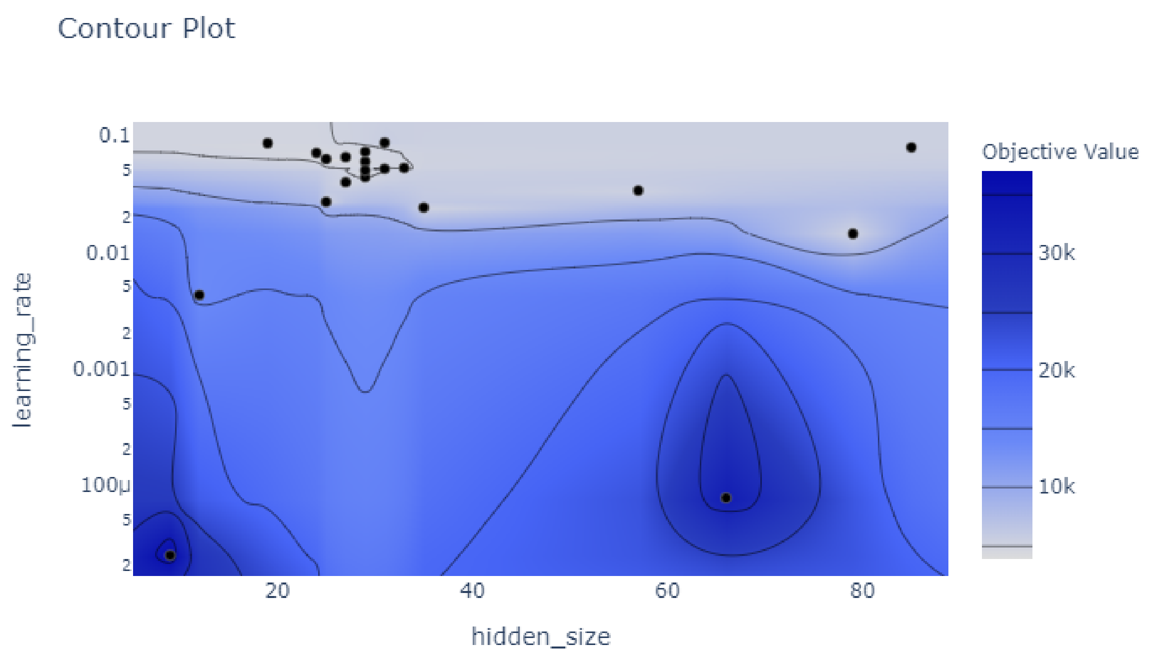


Figure 5.2 Contour plot between the different trial values for *learning_rate* and *hidden_size*.

It can be seen that the study did try the whole range of values for these two hyperparameters (very small learning rate and hidden size). With about 10 points that yielded relatively high objective values, the remaining trials were focused on exploitation. The majority of the remaining points are located in the region between 20 and 40 for the hidden size and ~ 0.05 for the learning rate. This is where the objective value is the lowest and the combination of hyperparameters are the best performing.

After some fine tuning of the ranges from analysing different plots, the final study determined that the hyperparameters for best performance were:

Table 5.1 Hyperparameter Values

Hyperparameter	**Value**
gradient_clip_val	0.5411877293345172
hidden_size	25
hidden_continuous_size	21
attention_head_size	2
learning_rate	0.06303513692563587
dropout	0.18722626612040522

These are the hyperparameters that will now be used to train the temporal fusion transformer model.

5.2. Training the Model

PyTorch Forecasting is a comprehensive library built on PyTorch, a popular deep learning framework. It provides powerful tools and functionalities for time series forecasting tasks. It offers a wide range of pre-built models such as the TFT, along with various utilities for data preprocessing, model training, evaluation, and deployment. This is what will be used to train on our data. Within PyTorch, the entire TFT architecture that was explained in chapter 4 is coded into the *TemporalFusionTransformer* package. This package will be used to create a TFT model for our data.

After the data is initialised in the python instance, the variables need to be categorised as was explained in chapter 2.1.

First, the two target variables, sell-out and sell-in are defined to tell the TFT what to predict. To keep the training time relatively short, only Soup, Ketchup and Beans products for Asda and Tesco are initialised.

```
targets = ['sales_volume','VOLUME_CS_SELL_IN']
```

The aim is to predict two variables at once; sell-in and sell-out. Sell-in is the volume sold from the company to the different retailers. On the other hand, sell-out is the volume sold by the retailers to the customers.

Now the list of static data that will be encoded as context vectors are defined;

```
st_cat = ['market_long_description','khc_product_subcategory',
'sell-in M ind','product_ppg','level']
```

Where the market variable represents the retailer name. The level variable will tell the TFT for which level it should predict. In our case, the TFT is forecasting for specific retailer-product combinations. The level variable will, for example, show Tesco Ketchup product A or Asda Beans product C.

Finally the time varying variables are split into continuous and categoricals.

```
tv_c_k = ['month','promo_mechanism','holiday_flag','epl_flag',
'football_wc_flag',  ...]

tv_r_k = ['tdp', 'avg_price', 'new_covid_cases','co2_emissions','avg_temp',
'lag1_tdp', 'lag2_tdp', 'rainfall_days',...]
```

Now the decision has to be made which of these variables will also be classified as future known inputs. All of these variables are already inputted into the encoder past inputs (time step $t-k:t$). The aim of the tool is to let the user experiment with different future inputs to make a business plan. For example, the price or promotion variables could be played with to see the effect on the sales volume. The user does not want to be overwhelmed and choose hundreds of variables to run a promotion though. This is why some will be assumed as last known for the time being. The business releases a promotion plan for the future that has future data regarding price and promotions. This will be used to make forecasts at the moment. The other time varying variables are assumed to be last known (for future time steps $t+1:\tau$).

In this framework, time-varying variables are integrated into both the encoder and decoder, capturing past, present, and future information. By incorporating future inputs in the decoder, proactive decision-making is facilitated while considering relevant historical and current data. This methodology efficiently addresses the temporal evolution of variables.

After several trial runs, it was identified that some variables are considered extremely important by the TFT. This is why it was experimented that another variable was added to the list; the lags of the important variables. For example, TDP is seen to be important by the TFT, hence new variables were implemented that represent two week lags; *lag1_tdp, lag2_tdp.*

It makes sense that the supply of the product in the stores has a prevalent impact on the demand of that product. This is what the TFT picked up on.

To train a model, the data has to be split into training and validation data. The training data will be used by the TFT model to learn patterns and behaviour. The validation data is an unbiased selection of the data that has never been seen by the TFT. This data is used to assess the performance of the forecasts.

The lookback period of our model (how far back in time the model will look to produce forecasts) is specified as four years, since this is the amount of data that is available. The first available data point being on 01/01/2018 for sell-out volume and 01/01/2021 for sell-in volume. For the time being forecasts are made for maximum the next 52 weeks (one year). These are specified as max_encoder_length and prediction_length respectively. The full code to initialise the dataset as a dataloader is seen in appendix C.

```
max_prediction_length = 52
max_encoder_length = 4 * 52
```

Now the training can start using the initialised data loader. The hyperparameters that were optimised are inputted. Note that some hyperparameters were not available to be trained by Optuna, so these were decided through manual experimentation (*lstm_layers, reduce_on_plateau_patience, log_interval*). One epoch is when the model has gone through all of the training data once. The maximum is set to 20 since after usually 11 epochs the model finds no more possible improvements anymore (the training then stops as per the early stopping mechanism).

```
    early_stop_callback = EarlyStopping(monitor = "val_loss", min_delta = 30,
    patience = 3, verbose = True, mode = "min")
lr_logger = LearningRateMonitor()


trainer = Trainer(max_epochs = 20, # total dataset epoch
                  accelerator ='gpu',
                  devices = [0,1,2,3],
                  enable_model_summary = True,
                  auto_scale_batch_size = True)

tft = TemporalFusionTransformer.from_dataset(training,
                  optimizer='adam',
                  learning_rate = 0.06303513692563587,
                  hidden_size = 25,
                  lstm_layers=1,
                  attention_head_size= 2,
                  dropout = 0.18722626612040522,
                  hidden_continuous_size = 21,
                  output_size = [7,7], # there are 7 quantiles by default: [0.02,
                  0.1, 0.25, 0.5, 0.75, 0.9, 0.98]
                  loss = MultiLoss([QuantileLoss(), QuantileLoss()]),
                  log_interval = 10,
                  reduce_on_plateau_patience = 4)
```

Four GPU's were used in parallel to maximise speed of training. The batch size decides how many data points are run in parallel per worker.

The optimizer that is used is the Adaptive Moment Estimation optimization algorithm (adam) that efficiently updates weights of the neural networks [15].

6 Results

6.1. Validation Forecasts

After the training is completed (which takes several hours), the results of the TFT can be analysed. The following results are from model 7, which delivered the best results so far. The PyTorch package has built in visualisation options that allows the user to immediately plot the outcome of specific forecasts using the TFT weights. Since there are many products for which forecasts are made, only the plots of two products will be looked at in detail; Asda Beans product A and Asda Ketchup product C.

First taking a look at the forecasts for Asda Beans product A;

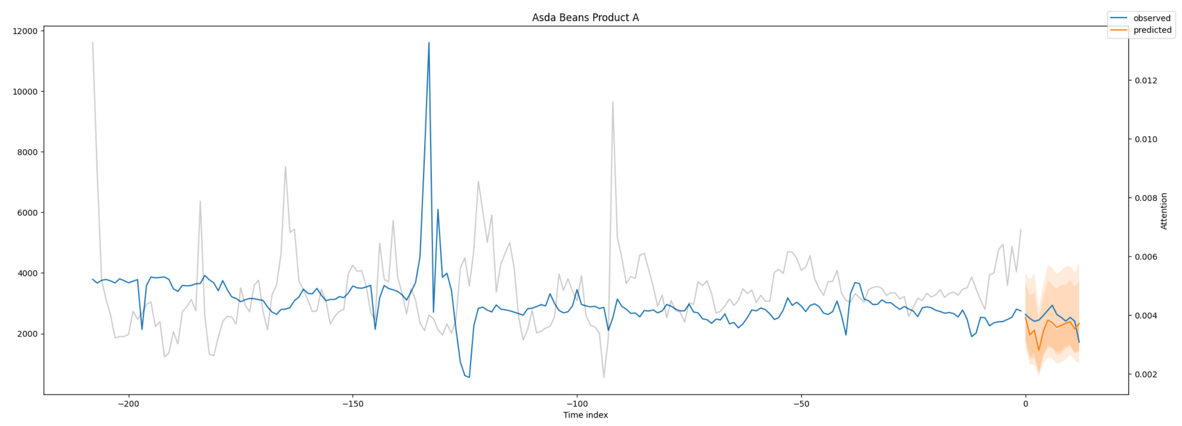


Figure 6.1 Time series data and quantile forecasts for Asda Beans product A.

This shows the time series for sell-out data for Asda Beans product A with the quantile forecasts after time index of $t = 0$ for the next $\sim$ 12 weeks. This is compared to the unbiased validation data of our target (the blue line). The grey line represents the attention for every past time step. The higher the attention in that time step was, the more of that time step was used to make predictions. There can be seen to be a spike in the attention line graph every 25 time steps, indicating a seasonal pattern. This is what attention is supposed to capture in the first place, which is a good sign. Furthermore, there is large spike at time index -95. This is not due to the data though. At this time step sell-in data was also introduced. Sell-in data was was only available from 2021, whilst sell-out data is available since 2018. The attention graph should therefore be analysed in two halves; one when sell-in is introduced (so starting at index -95). The other half would be before this, when the model only looks at sell-out and assumes sell-in is zero. This is illustrated in the following figure;

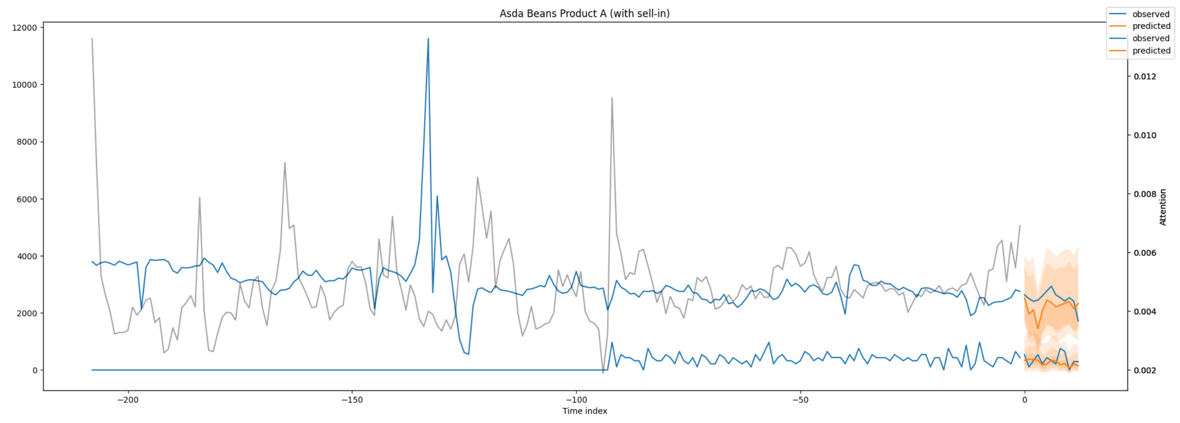


Figure 6.2 Time series data and quantile forecasts for both sell-out and sell-in.

Here the sell-out data is represented by the upper line, whilst the sell-in data is on the lower blue line. The grey line represents the amount of attention that is give to certain time points. The spike in attention can be seen to happen at the exact same time when sell-in data is introduced. It is important to note that the same set of independent variables are used to forecast both sell-in and sell-out. For this case the sell-in forecast does not seem to be following the actuals quite well, whilst the sell-out forecast looks more accurate. Taking a look closer at the median (0.5th) quantile sell-out predictions of Asda Beans product A:

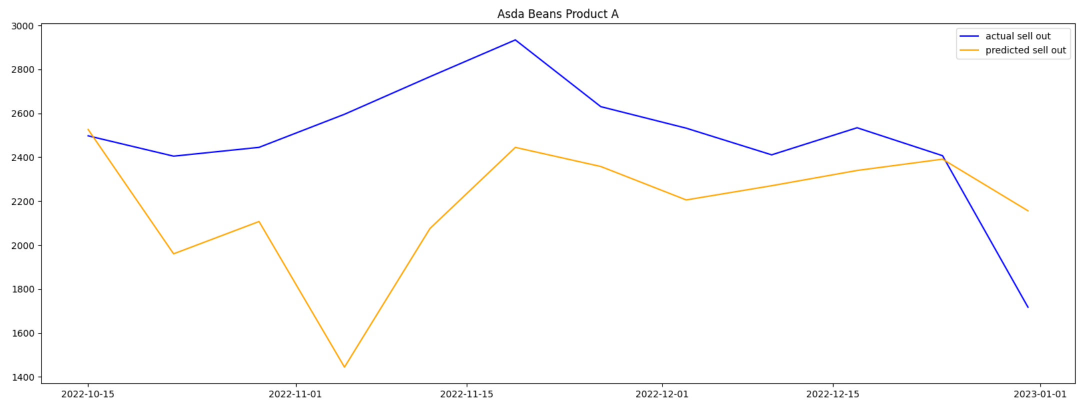


Figure 6.3 Closer look at the forecast for Asda Beans Product A.

At the start of the validation, the model seems to be under-predicting for the first week of November. In the weeks following this, the model performs better and follows the trend for the actuals quite closely. The only region where the model does not perform is the aforementioned drop in the first week of November. At the first week of 2023, the model does anticipate the drop in the sales volume. The drop, however, turned out to be more steep than was predicted, causing the model to over-predict for the first time.

Taking a look at a product in a different category; Asda Ketchup product C.

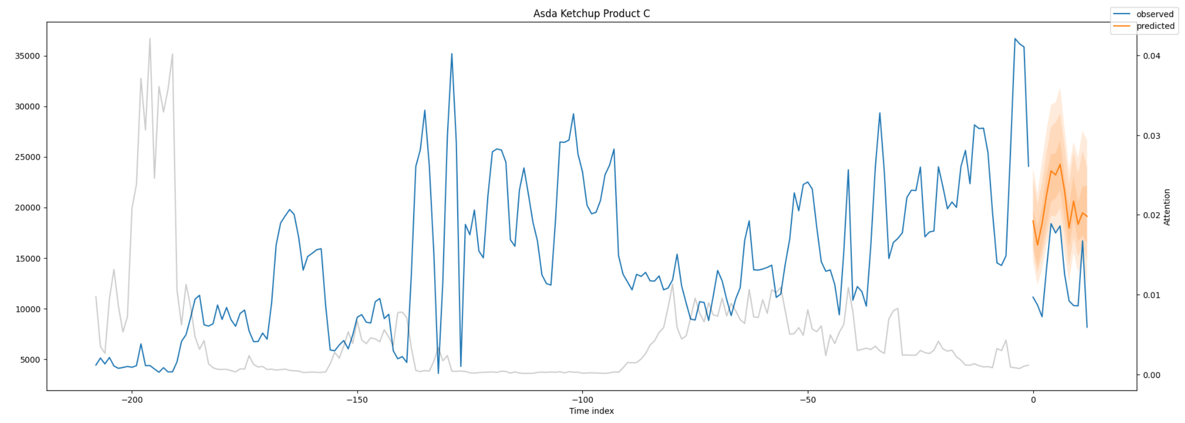


Figure 6.4 Time series data and forecasts for Ketchup product C.

From this plot the attention is quite high at the start. In the business it is said that Ketchup products are notoriously difficult to predict since the volumes are so volatile. This is evident by looking at the observed data in the figure. Similar to the final week of validation for Beans product A, the model anticipated a drop in sales volume (this time at time index 0). The predicted was, however, not steep enough. It is still quite impressive that the model anticipated this sudden drop when it happened. When the covid outbreak occured there were supply issues, this is why the massive fluctuations in the observed data can be seen at around time index -130. What is interesting is that the model did not pay much attention in this section, there are only a couple of small spikes. This backfired when predicting in the future because as can be seen in the following figure, the model overpredicted.

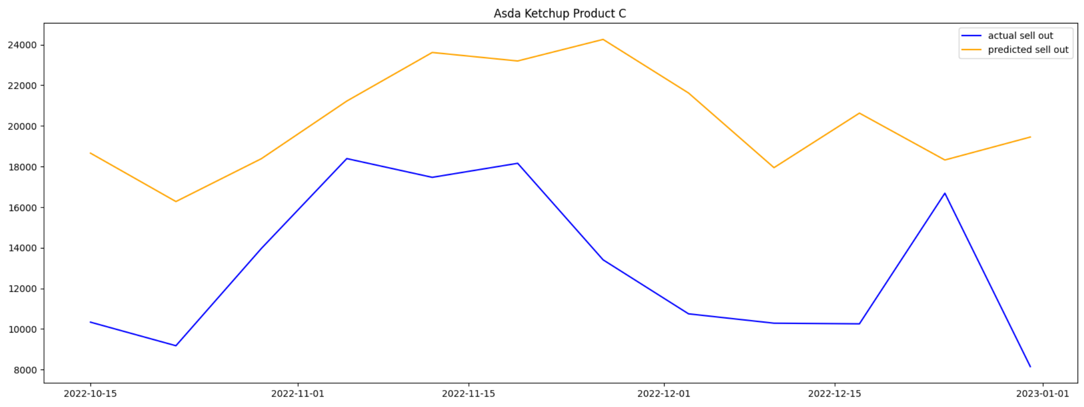


Figure 6.5 Closer look at the 0.5th quantile sell-out forecast for Asda Ketchup Product C.

Even though the model over-predicts the sales volume by some 8000 units, it can be seen to follow the pattern of the actuals quite well. The model does seem to struggle with sudden drops or increases, with the increase on the first of November not being as steep as the actuals. The drop in sales volume is again realized at the start of December but again does not have as high as a slope. For the first of January, the model predicts a rise, which is the opposite of the actuals. Similar to Asda Beans product A, it struggles to predict the first week of the new year. A reason for this may be that the business often comes up with new strategies and promotions for the start of the new year.

The method that is used to assess the accuracy of the forecasts is with the weighted mean absolute percentage error (WMAPE);

$$WMAPE = \frac{\sum_{t=1}^{\tau_{max}} |y_t - \hat{y}_t|}{\sum_{t=1}^{\tau_{max}} |y_t|} \tag{6.1}$$

For y_t and $\hat{y}_t$ being the actual and (median quantile) predicted sales volume at forecasted time index t up to the maximum forecasted time index τ_{max}.

The WMAPE is then calculated for every product-retailer combination for the sell-out and sell-in;

Table 6.1 WMAPE's for select products

Product	**Sell-out WMAPE**	**Sell-in WMAPE**
Asda Beans Product A	15.76	50.29
Tesco Beans Product A	21.23	28.43
Asda Beans Product B	19.37	31.76
Tesco Beans Product C	14.37	21.78
Asda Beans Product J	12.23	62.31
Tesco Ketchup Product A	29.83	65.30
Asda Ketchup Product C	56.22	20.31
Tesco Ketchup Product D	22.3	45.24
Tesco Ketchup Product G	24.65	46.76
Asda Ketchup Product H	25.15	46.88
Asda Soup Product A	26.42	25.06
Tesco Soup Product A	30.66	18.42
Asda Soup Product B	31.10	34.32
Tesco Soup Product E	55.54	62.60
Asda Soup Product G	1595.90	-

The best performing category are the Beans products, with all WMAPES being less or around 20 for sell-out. The difficulty to predict Ketchup is evident here with the WMAPES being higher than 20. Soup also seems to be struggling with relatively high WMAPES. The model seems to have much more difficulty with predicting sell-in data rather than sell-out. A reason for this is because there is much less sell-in data available than sell-out, so the model would have less opportunity to learn. Sell-in data is also harder to predict since it is not as dependent on the customer, but rather on the retailer, which makes it more unnatural. For these reasons, the discussions will focus more on sell-out data rather than sell-in. The products showing an extremely high WMAPE are low volume products. For these products only a couple hundred units are sold every week. Most of the time these low volume products are decided to be sold only at certain time periods, with the business sometimes deciding not to sell the product for a long period of time at all. This causes there to be much less data available for these products. This makes it confusing for the model to understand and build forecasts, hence why the WMAPE is extremely high (Asda Soup product G). It is more important to get good WMAPE's for the highest selling products. In order to get a better overview of the importances without revealing private information, a weighted error can be calculated for each subcategory. The error is weighed by every product's contribution to the subcategory sales volume; this means that the higher selling products (the important ones), will have a higher weight coefficient. This is done for sell-out data in Tesco:

Table 6.2 Weighted error by contribution for Tesco subcategories

Subcategory	**Weighted Error**
Beans	22.63
Ketchup	27.50
Soup	32.67

Forecasting beans is the easiest for the TFT, followed by the Ketchup and Soup subcategories. It gives a promising summary of the power of the TFT model, with the highest error being 30% which is a good error percentage taking into account the scalability of the model. These numbers could definitely still be improved with changes in data quality and more test runs, nevertheless they are still encouraging results.

Whilst the attention graph has been separately analysed for two specific products, the average attention for every product-retailer combination can also be plotted;

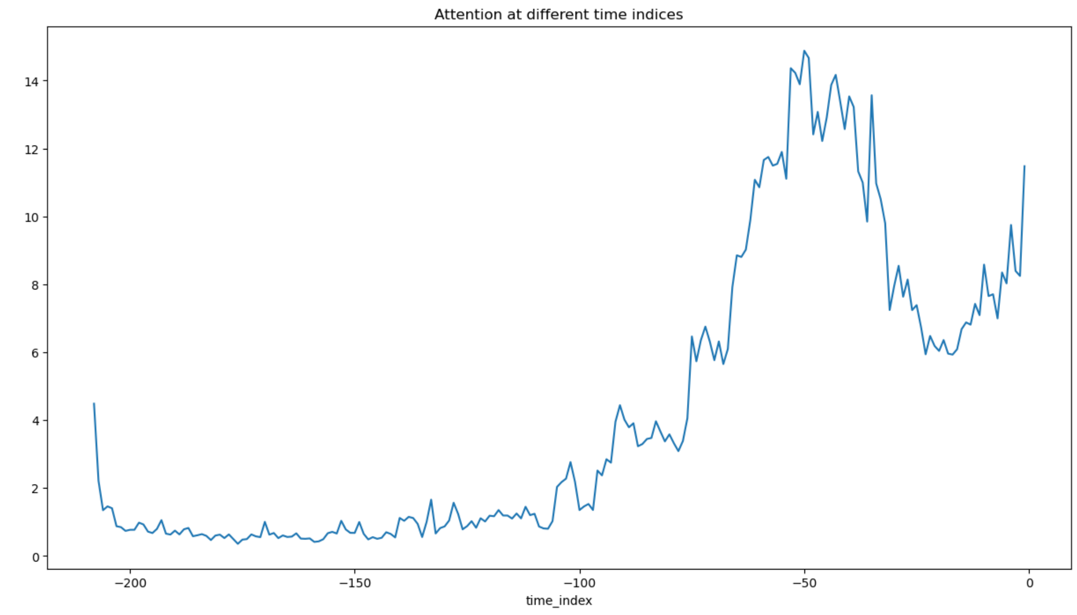


Figure 6.6 Average attention at all past time steps.

There is a general trend that as the time index approaches closer to the forecasting period, the attention increases. This means that more recent data points influence futures forecasts more, which makes sense. Another observation is that the attention of the transformer starts to fall more at the -50 time index mark. This would be one year before the forecast (52 weeks). The attention keeps dropping until about 6 months before the forecast, where it increases again. The model believes that the data 6-12 months before forecasts is not as important as longer than 12 months. The forecast for the data is made for the end of 2022/start of 2023, so one year before would make it the end of 2021. The start of 2022 is when the final Covid lockdown rules were lifted in the UK. Through the middle of 2020 is when the lockdowns first started. This corresponds to a time index of around -125, which is when the upwards linear trend starts. The reason why the model would pay more attention during the lockdown period is due to the massive fluctuation in the sales volumes. When there is a lockdown period, people are restricted to go into stores to buy groceries, causing the sales volume to crash downwards. The constant lifting and restricting pattern is what the model focuses on. When all lockdowns were lifted at the start of 2022, there would be less fluctuations, so the model would pay less attention to it. The attention only increased again when the time period got within 6 months of the forecast. This is supported by the fact that a lockdown index is one of the most important variables to predict sales volume (as seen in the next section).

6.2. Variable Importances

The python package also outputs the average variable importances of every product at every forecasting time step so that the user may know what affects the sales volume the most. Since there are three different variable categories, the model has separate weights for all.

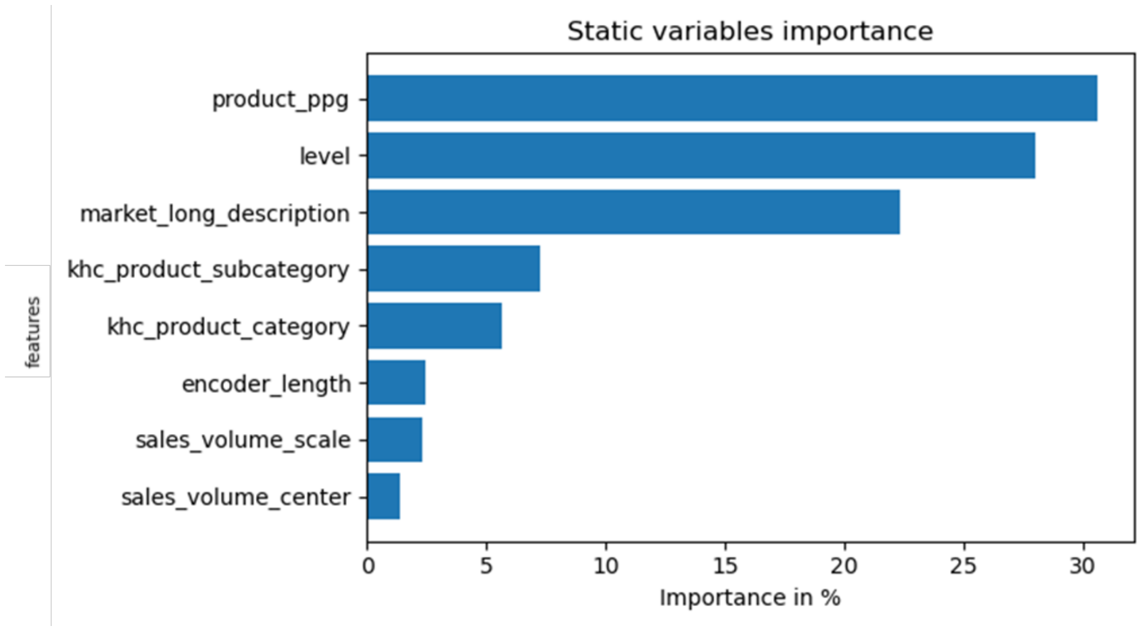


Figure 6.7 Static variables importances.

The most important static variable is the product, which is logical. This is followed closely by the level, which represents the combination of product and retailer. This could also be the most important variable for products where there are a lot of different between the same product at different retailers. The different retailers are represented in the third most important variable. It is logical that the subcategory (e.g. Ketchup) is more important than the category (e.g. Table sauces) since it is closer in the hierarchy (The category Table sauces includes all the subcategories of Ketchup, Mayonnaise etc.). The lookback period and scale and median of a products' sales volume are the least important static variables.

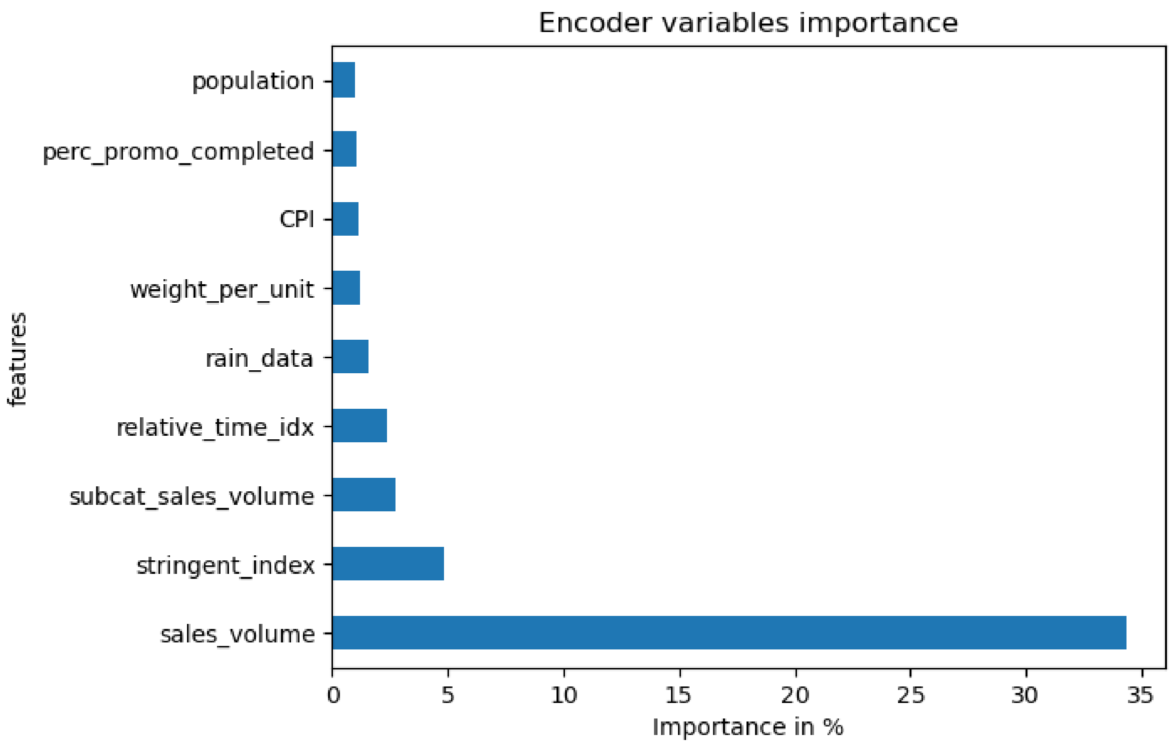


Figure 6.8 Encoder variables importances.

The past data that is inputted is represented by the encoder variables. As expected, the previous sales volume gives the most indication as to what the future sales volume would be. The second most important is the stringent index which represents lockdown periods in the U.K. The other notable important variables are the sales in the subcategory and the relative time index. The relative time index is an index that shows how close past data is to the predictions made. This is important since more recent data would have a greater effect on future forecasts.

Note that for the encoder and decoder, only the variables that have importances greater than 1% are visualised, since the variable list exceeds 100.

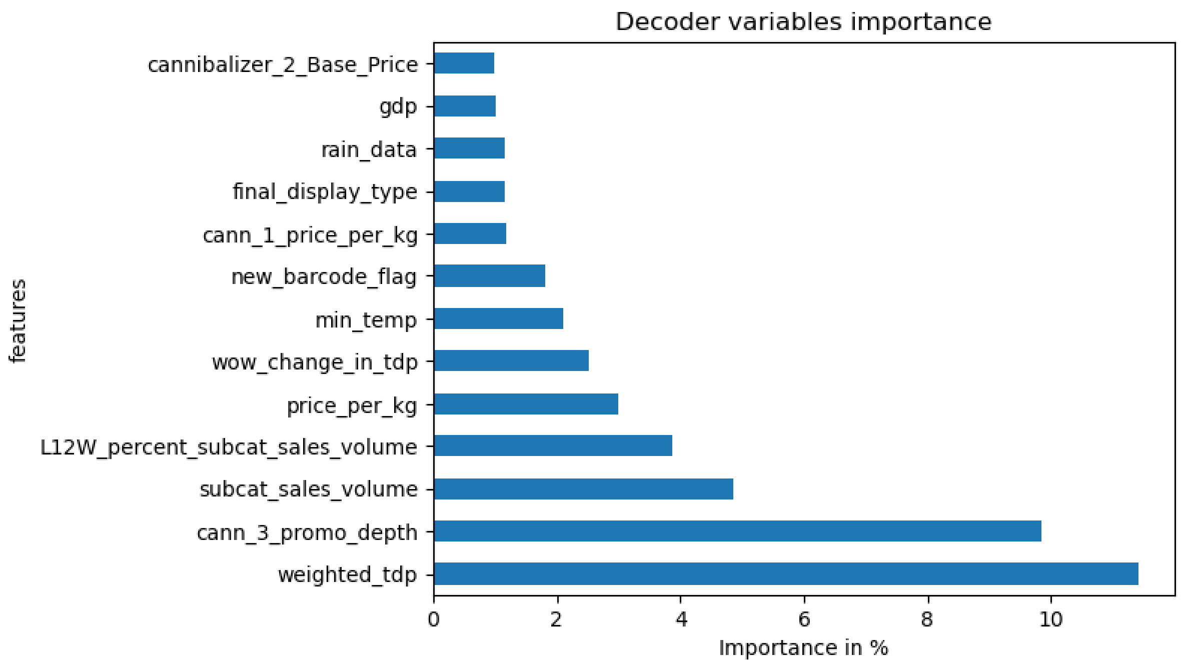


Figure 6.9 Decoder variables importances.

Finally, the importances of the variables in the decoder are looked at. The most important variable in the future is the weighted TDP. The weighted TDP is the volume weighted TDP by different barcodes. Within a product, there are different barcodes. *TDP* measures the average distribution of all the barcodes within the product whilst weighted TDP weighs them by their volume. This gives a better representation for the more important barcodes within the product. The second most important variable is the promo depth of cannibalizer 3. The promo depth is the price difference between the original (base) price of a product, and the price when it is on that promotion. cannibalizer 3 for Ketchup Product A, could be Ketchup Product C, for example. This is not specified but already initialised in the data beforehand. It is peculiar since the products' own promo depth does not seem to be important at all. It could be that the promo depth for cannibalizer 3 had significant fluctuations that it became highly relevant for future forecasts. The sales in the subcategory and the percentage that the product made up of those subcategory sales in the previous 12 weeks, are the next two important variables. This is logical since changes in a product's subcategory should impact the products within it. The price per kg has a ~ 3% importance in the decoder variables. This is surprising since this is the first mention of the price of the product being important for forecasting sales. one would expect that the price of a product would be the driving factor for the sales. Apparently, this is not the case. The model considers distribution and external factors more important. This may be since the past pricing data does not fluctuate as much as other factors (such as TDP), so the model may not pick up on certain patterns. There are also no important factors regarding data of competitor products, only cannibalizers. This is a good thing since it means that the business is in control of affecting its' sales, and it is not too dependent on competitors' choices. The last few significant variables are seasonal variables such as rain and temperature, as well as other factors relying on cannibalizers. Final display type is how the product is placed (whether it is on the bottom of the shelf, or at the front of the store etc.).

7
Conclusion

To conclude, the TFT is a powerful architecture that has been carefully engineered to produce accurate forecasts for time series problems. It is a deep neural network that uses a combination of recently developed mechanisms. Most notably, self attention is used to provide context for the model in the temporal aspect. This makes the TFT aware of all variables and their dependencies throughout every time step, in order to pay more attention to certain variables when deemed necessary. In contrast to other algorithms, the architecture has proven to still be effective when implementing multi horizon forecasting. This makes TFT especially useful for the business due to the ability of using time varying future inputs. From this research as well as other studies [32], the TFT is evident to be applicable to different time-series problems.

We have built a successful model to make forecasts for three separate subcategories. The model currently has a 20-30% error rate per subcategory, with potential to get it even lower. Being able to make forecasts for every product-retailer combination shows the scalability and potential of using this TFT model. By using interpretable attention in the architecture, the model is also able to show the importances of the variables that are used to make forecasts. These were shown to be Covid lockdown periods, distribution and the previous sales volume. Being able to justify forecasts, makes a tool using the TFT more reliable.

We see the TFT as an excellent candidate to be used as a tool. Several future inputs are controlled by the business, such as price and even distribution to some extent. To be able to know how such a change would affect the sales volume of specific products, gives the company extremely useful insights. Especially considering that the output of the TFT model gives an accurate forecasted number with an error of 30%.

As an area for expansion, we see that such a model can produce forecasts for more subcategories and in further regions rather than just the UK. This would increase the capability and the tool could cover the whole globe.

8 Extensions

8.1. Hierarchical Learning

Throughout the paper, the notion of estimating the future ‘known’ was discussed. In an ideal scenario, the user has the choice to change these variables. This is unrealistic since there are too many variables and seasonal data such as rainfall days would be pointless for the user to estimate. These variables were estimated using last known or last years’ value in case of seasonal data. Whilst taking last known values may give a general estimation for these variables, it is a naive assumption. Many variables can drastically change in the short term, a competitor could change its’ price and seasonal data such as rainfall is never exactly the same as previous years. The results that were analysed in the previous chapter used a future data set with these ’last known assumptions’. A test was run using a future data set that had the exact real values instead of using assumptions. For this, the model was trained up to data of October 2022, and predictions were analysed for the end of 2022 and start of 2023. Instead of using an estimated future data set for predictions, actual data was inputted. For example, TDP in the future data set of Ketchup product A in November is assumed to be the last known, so it is set to the TDP for October. In this case however, the actual reported value for TDP of Ketchup product A is used as TDP in November. This is done for every variable in the decoder. The WMAPES of certain products for sell-out data are below.

Table 8.1 Potential WMAPE’s for select few products

Product	**Best Potential WMAPE**	**Current WMAPE**
Asda Beans Product C	2.87	11.71
Asda Ketchup Product B	6.91	27.21
Asda Soup Product B	1.36	31.10

This shows a massive improvement and potential if there is a better way of estimating the future data set. Instead of assuming last known, separate time series models can be created for every variable in the future. For example, an auto regressive integrated moving average (ARIMA) model could be created for Ketchup product B to estimate the TDP in future time steps. This estimation of future TDP would then be used as an input into the TFT to make predictions for future sales volume. Other machine learning models such as XGboost and CATboost could also be implemented to predict future variable values. The better these models perform in getting closer to the actual future variable values, the better the TFT will perform in its prediction. This would represent a hierarchical learning architecture, with different models working together in unison to provide the final output.

8.2. Fractal Interpolation

Real data is often quite noisy, which makes it challenging for models to detect patterns. Machine learning models especially, learn by interpreting signals. In Raubitzek and Neubauer [23], a procedure was outlined to improve the performance of an LSTM based time series forecaster. Raubitzek describes using fractal and linear interpolation to manipulate the data so that the LSTM layers can interpret the data better and make more accurate forecasts. The interpolation procedure would generate a more fine grained time series. The main idea of the paper is to use the Hurst exponent [8].

The Hurst exponent is a measure of complexity and randomness, which determines whether a series is mean reverting. The Hurst exponent ranges from zero to one and the closer to zero it is, the stronger the mean reversion process is. This means that a high value is more likely to be followed by a low one. If the value is closer to one, the series is persistent, which means that a high value is followed by a higher one. A value of $H = 0.5$ signifies that the series is a geometric random walk. The Hurst exponent is therefore used to measure the amount a series deviates from a random walk. The most accurate method of determining the value of the Hurst exponent is using rescaled range analysis [10].

Fractal interpolated series reproduce data with similar complexity to real life data than for example, polynomials. This is because fractal interpolation is based on iterated function systems, rather than elementary functions; for our time series defined as;

$$\{(u_m, v_m) : m = 0, 1, \dots, M\}$$

The interpolation points are defined as a subset of the time series points;

$$\{(x_i, y_i) : i = 0, 1, \dots, N\}$$

With both sets being linearly ordered. Then,

$$\{\mathbb{R}^2; w_n, n = 1, 2, \dots, N\}$$

Is An iterated function system with affine transformations

$$w_n \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_n & 0 \\ c_n & s_n \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} d_n \\ e_n \end{bmatrix}$$

Which is constrained to satisfy

$$w_n \begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix}, w_n \begin{bmatrix} x_N \\ y_N \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix}$$

for every $n = 1, 2, \dots, N$

Which gives solutions;

$$a_n = \frac{x_n - x_{n-1}}{x_N - x_0} \tag{8.1}$$

$$d_n = \frac{x_N x_{n-1} - x_0 x_n}{x_N - x_0} \tag{8.2}$$

$$c_n = \frac{y_n - y_{n-1}}{x_N - x_0} - s_n \frac{y_N - y_0}{x_N - x_0} \tag{8.3}$$

$$e_n = \frac{x_N y_{n-1} - x_0 y_n}{x_N - x_0} - s_n \frac{x_N y_0 - x_0 y_N}{x_N - x_0} \tag{8.4}$$

For s_n being a free parameter representing the vertical scaling factor. The proof and further details for the IFS can be read in Manousopoulos et al [21].

This method is then applied on the time series. The time series is split in i subsets, so that fractal interpolation is applied separately to every subset. The Hurst exponent is then calculated for every subinterval; H_i. The iterative method described above is applied to the subset for random values for the parameter s_n (between -1 and 1). The Hurst exponent for this value of s_n on the subset i is now calculated, H_{old}. For every value for s_n, a new Hurst, H_{new} is calculated.

$$\text{if } |H_i - H_{new}| < |H_i - H_{old}|, \text{ then } H_{old} = H_{new}$$

This is repeated 200 times so that a value for s_n is found to give closest possible interpolated Hurst exponent to the original Hurst exponent. The reason why this is important is so that the complexities between the original and the fractal interpolated data are as similar as possible. Having similar complexities will then make the forecasts more accurate. This procedure is done until every subset of the time series is fractal interpolated.

The fractal interpolated time series is then run through a LSTM neural network to make forecasts. The results against the original time series data are then compared.

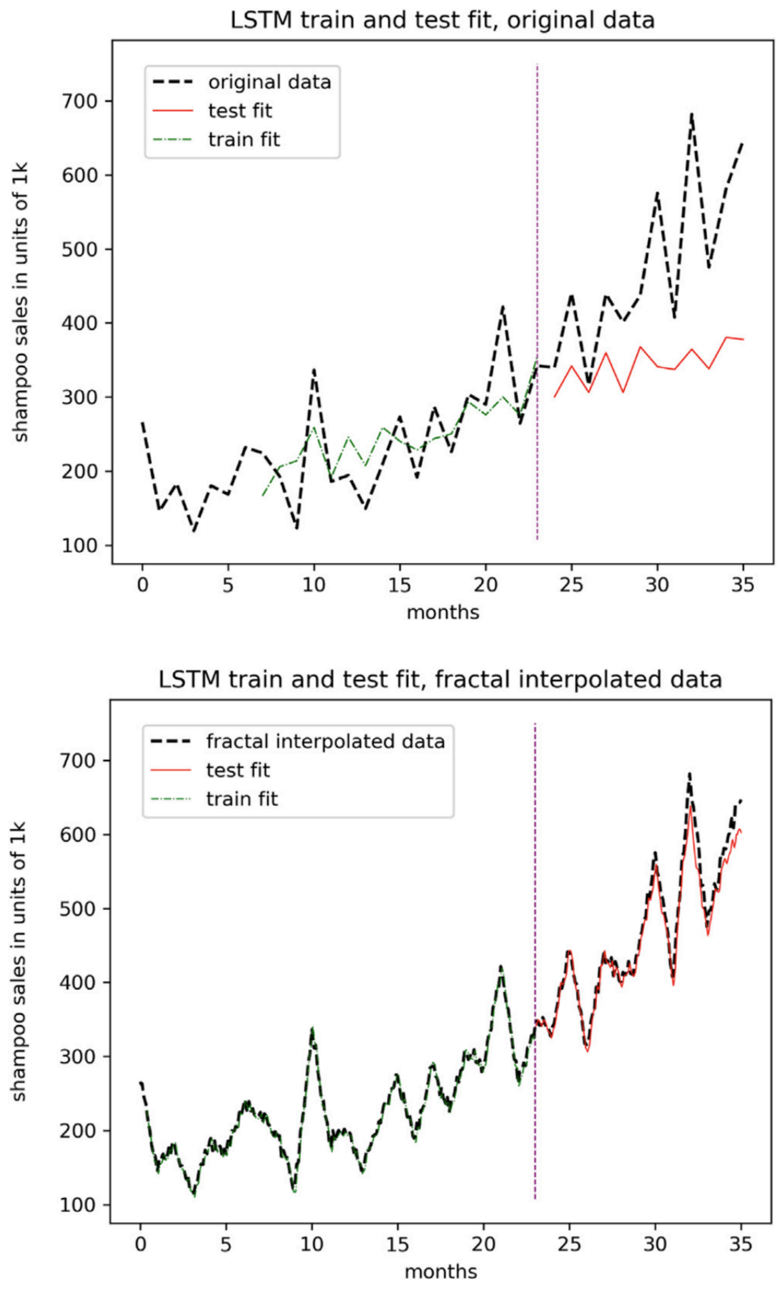


Figure 8.1 Comparison of forecasts between original data and after fractal interpolation (for predictions of shampoo sales) [23].

The fractal interpolated data seems to perfectly follow the original data, with the test fit being practically on top of the actual data. In comparision, the forecasts made from the orginal data does not seem to come close to the actuals whatsoever. Fractal interpolation in this case shows a drastic improvement for the prediction power of the LSTM neural network.

This could also be done to the time series data for the case of Ketchup or Beans. A challenge is that there is a huge amount of data for the forecasting problem, but this technique could first be attempted on the most important product-retailer combinations to see if it helps with the predictions, and then applied to all the data if deemed helpful. Although the paper only uses LSTM's for the forecasting, and not the TFT, parts of the architecture in the TFT uses LSTM's. Even so, using this technique seemingly made a more visible pattern out of the time series, which would be easier for the TFT to pick up on.

A
Appendix A: Full list of variables TFT evaluates importances on

```
{'tdp', 'avg_price', 'price_per_kg', 'weighted_tdp', 'subcat_sales_volume',
'percent_subcat_sales_volume', 'SELL_IN_ret_subcat', 'SELL_IN_ret_cat',
'SELL_IN_mkt_cat', 'SELL_IN_mkt_subcat', 'cann_1_tdp', 'cann_1_weighted_tdp',
'cann_2_tdp', 'cann_3_tdp', 'cann_3_weighted_tdp', 'comp_1_tdp',
'comp_1_weighted_tdp', 'comp_2_tdp', 'comp_2_weighted_tdp', 'comp_3_tdp',
'comp_3_weighted_tdp', 'cann_1_price_per_kg', 'cann_2_price_per_kg',
'cann_3_price_per_kg', 'comp_1_price_per_kg', 'comp_2_price_per_kg',
'comp_3_price_per_kg', 'cann_1_avg_price', 'cann_2_avg_price', 'cann_3_avg_price',
'comp_1_avg_price', 'comp_2_avg_price', 'comp_3_avg_price', 'week_of_promo',
'length_of_promo', 'promo_flag', 'week_since_last_promo', 'promo_price',
'non_promo_price', 'cann_1_week_of_promo', 'cann_1_length_of_promo',
'cann_1_promo_flag', 'cann_1_week_since_last_promo', 'cann_1_promo_price',
'cann_1_non_promo_price', 'cann_2_week_of_promo', 'cann_2_length_of_promo',
'cann_2_promo_flag', 'cann_2_week_since_last_promo', 'cann_2_promo_price',
'cann_2_non_promo_price', 'cann_3_week_of_promo', 'cann_3_length_of_promo',
'cann_3_promo_flag', 'cann_3_week_since_last_promo', 'cann_3_promo_price',
'cann_3_non_promo_price', 'stringent_index', 'new_covid_cases',
'gdp', 'population', 'co2_emissions', 'flood_warning_flag', 'air_frost_days',
'avg_temp', 'min_temp', 'rainfall_days', 'rain_data', 'sunshine_hours',
'football_event', 'Stringent_index_lag_1', 'Stringent_index_lag_2',
'Stringent_index_lag_3', 'Stringent_index_lead_1', 'Stringent_index_lead_2',
'Stringent_index_lead_3', 'Christmas_Flag', 'Year_Start_Peak_Flag',
'school_start_flag', 'promo_mechanic_pack_feature', 'Base Price',
'CPI','euro_flag','promo_fraction','unique_barcodes', 'new_barcode_flag',
'perc_promo_completed','distinct_ppg_count', 'lag1_percent_subcat_sales_volume',
'lag2_percent_subcat_sales_volume', 'lag3_percent_subcat_sales_volume',
'lag4_percent_subcat_sales_volume', 'lag5_percent_subcat_sales_volume',
'lag6_percent_subcat_sales_volume', 'lag7_percent_subcat_sales_volume',
'lag8_percent_subcat_sales_volume', 'lag9_percent_subcat_sales_volume',
'lag10_percent_subcat_sales_volume', 'lag11_percent_subcat_sales_volume',
'lag12_percent_subcat_sales_volume', 'L12W_percent_subcat_sales_volume',
'VOLUME_CS_SELL_IN_lag_1', 'VOLUME_CS_SELL_IN_lag_2',
'VOLUME_CS_SELL_IN_lag_3', 'VOLUME_CS_SELL_IN_lag_4', 'VOLUME_CS_SELL_IN_lag_5',
'VOLUME_CS_SELL_IN_lag_6', 'VOLUME_CS_SELL_IN_lag_7', 'VOLUME_CS_SELL_IN_lag_8',
'VOLUME_CS_SELL_IN_lag_9', 'VOLUME_CS_SELL_IN_lag_10', 'VOLUME_CS_SELL_IN_lag_11',
'VOLUME_CS_SELL_IN_lag_12', 'cannibalizer_1_Base_Price',
'cannibalizer_2_Base_Price', 'cannibalizer_3_Base_Price',
'wow_change_in_percent_sales', 'product_ppg_promo_depth', 'cann_1_promo_depth',
'cann_2_promo_depth', 'cann_3_promo_depth', 'lag1_tdp', 'lag2_tdp',
'wow_change_in_tdp', 'cann_1_percent_diff_in_kg_price',
'cann_2_percent_diff_in_kg_price', 'cann_3_percent_diff_in_kg_price',
'cann_1_percent_diff_in_unit_price', 'cann_2_percent_diff_in_unit_price',
'cann_3_percent_diff_in_unit_price', 'lag1_promo_depth', 'lag2_promo_depth',
'wow_change_in_promo_depth', 'PPG_Count_diff', 'weeks_since_change_in_ppg_count',
'dynamic_week_percent_subcat_sales_volume', 'weight_per_unit', 'pack_size',
'weight_in_kg', 'cannibalizer_1_pack_size', 'cannibalizer_1_weight_per_unit',
'cannibalizer_1_weight_in_kg', 'cannibalizer_2_pack_size',
'cannibalizer_2_weight_per_unit', 'cannibalizer_2_weight_in_kg',
'cannibalizer_3_pack_size', 'cannibalizer_3_weight_per_unit',
'cannibalizer_3_weight_in_kg', 'competitor_1_pack_size',
'competitor_1_weight_per_unit', 'competitor_1_weight_in_kg',
'competitor_2_pack_size', 'competitor_2_weight_per_unit',
'competitor_2_weight_in_kg', 'competitor_3_pack_size',
'competitor_3_weight_per_unit', 'competitor_3_weight_in_kg',
'cann_1_weight_percent_diff', 'cann_2_weight_percent_diff',
'cann_3_weight_percent_diff', 'ppgs_on_promo', 'Barcode Shift'}
```

B

Appendix B: Full list of Hyperparameters with definition

Table B.1 Definition of all Hyperparameters

Hyperparameter	**Definition**
hidden_size	Hidden size of network which is its main hyperparameter and can range from 8 to 512.
lstm_layers	Number of LSTM layers.
dropout	Dropout rate.
output_size	Number of outputs (e.g. number of quantiles for QuantileLoss and one target or list of output sizes).
attention_head_size	Number of attention heads.
hidden_continuous_size	Default for hidden size for processing continuous variables.
learning_rate	Rate that determines the step size at which the model adjusts its parameters during the training process.
reduce_on_plateau_patience	Patience after which learning rate is reduced by a factor of 10.
log_interval	Log predictions every x batches, do not log if 0 or less, log interpretation if >0. If <1.0 , will log multiple entries per batch.
gradient_clip_val	Threshold that limits the magnitude of gradients during backpropagation to prevent instability and exploding gradients during training.
embedding_sizes	Dictionary mapping (string) indices to tuple of number of categorical classes (to handle categorical variables).

C

Appendix C: Code to initialise dataset into dataloader

```
max_prediction_length = 52
max_encoder_length = 4 * 52
t_dt = '2022-10-15'

# Multi-Normalizer for the 2 targets
normalizer = MultiNormalizer([GroupNormalizer(),GroupNormalizer()])
normalizer.fit(m_ds[targets[2:4]], m_ds['level'])

keys = m_ds.select_dtypes(include='object').columns.tolist()

# Initialize the dictionary using a dictionary comprehension and define encoders
for Categorical variables
encoders = {key: NaNLabelEncoder(add_nan = True) for key in keys}
scalers =  {key: RobustScaler() for key in tv_r_k}

print(encoders, '\n')
print(scalers, '\n')

# get time idx for training cut-off
training_cutoff = m_ds[m_ds.period_end_dt < t_dt].time_idx.max()

#Create Time Series Dataset for training
training =  TimeSeriesDataSet(m_ds[m_ds.period_end_dt < t_dt]
[m_ds.columns.drop_duplicates().tolist()].copy(),
                            time_idx = "time_idx",
                            target = targets[2:4],
                            group_ids = ["level"],
                            allow_missing_timesteps = True,

                            min_encoder_length = 4,
                            max_encoder_length = max_encoder_length,
                            min_prediction_length = 1,
                            max_prediction_length = max_prediction_length,

                            static_categoricals = st_cat,
                            time_varying_known_reals = tv_r_k,
                            time_varying_known_categoricals = tv_c_k,
                            time_varying_unknown_reals = targets[2:4],
                            scalers = scalers,
                            categorical_encoders = encoders,
                            target_normalizer = normalizer,
                            # we normalize by group

                            add_relative_time_idx = True,
                            add_target_scales = True,
                            add_encoder_length = True)

#Create Time Series Dataset for validation
validation = TimeSeriesDataSet.from_dataset
(training, m_ds, min_prediction_idx= training_cutoff + 1, stop_randomization= True,
predict = True, min_prediction_length = 1)

# create dataloaders for our model
batch_size = 32 # step batch size, higher more memory but faster training

# if you have a strong GPU, feel free to increase the number of workers
train_dataloader = training.to_dataloader(train=True, batch_size=batch_size,
num_workers= 24 , persistent_workers = True) # num of cpus is 8 on cluster
val_dataloader = validation.to_dataloader(train=False, batch_size=batch_size * 10,
num_workers = 24  , persistent_workers = True)
```

Bibliography

[1] Nikolas Adaloglou. Why multi-head self attention works: math, intuitions and 10+1 hidden insights. 2021. URL https://theaisummer.com/self-attention/.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. *CoRR*, abs/1907.10902, 2019. URL http://arxiv.org/abs/1907.10902.

[3] Ramya Akula and Ivan Garibay. Interpretable multi-head self-attention architecture for sarcasm detection in social media. *Entropy*, 23(4), 2021. ISSN 1099-4300. doi: 10.3390/e23040394. URL https://www.mdpi.com/1099-4300/23/4/394.

[4] Praghati Baheti. Activation functions in neural networks [12 types use cases], May 2021. URL https://www.v7labs.com/blog/neural-networks-activation-functions.

[5] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. 24, 2011. URL https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf.

[6] Clevert, Unterthiner, and Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). Feb 2016. URL https://arxiv.org/abs/1511.07289.

[7] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *CoRR*, abs/1612.08083, 2016. URL http://arxiv.org/abs/1612.08083.

[8] HURST H. E. Long term storage. *An Experimental Study*, 1965. URL https://cir.nii.ac.jp/crid/1573387449265485184.

[9] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/076a0c97d09cf1a0ec3e19c7f2529f2b-Paper.pdf.

[10] M. Gilmore, C. X. Yu, T. L. Rhodes, and W. A. Peebles. Investigation of rescaled range analysis, the Hurst exponent, and long-time correlations in plasma turbulence. *Physics of Plasmas*, 9(4):1312–1317, 03 2002. ISSN 1070-664X. doi: 10.1063/1.1459707. URL https://doi.org/10.1063/1.1459707.

[11] Cheng Guo and Felix Berkhahn. Entity embeddings of categorical variables. *ArXiv*, abs/1604.06737, 2016. URL https://arxiv.org/abs/1604.06737.

[12] Brendan Hasz. Bayesian hyperparameter optimization using gaussian processes, Mar 2019. URL https://brendanhasz.github.io/2019/03/28/hyperparameter-optimization.html.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[14] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998. URL http://link.springer.com/article/10.1023/A:1008306431147.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

[16] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2. URL https://doi.org/10.1007/3-540-49430-8_2.

[17] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. *arXiv e-prints*, art. arXiv:1607.06450, July 2016. doi: 10.48550/arXiv.1607.06450.

[18] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf.

[19] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *CoRR*, abs/1810.05934, 2018. URL http://arxiv.org/abs/1810.05934.

[20] Bryan Lim, Sercan Ö. Arık, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021. ISSN 0169-2070. doi: https://doi.org/10.1016/j.ijforecast.2021.03.012. URL https://www.sciencedirect.com/science/article/pii/S0169207021000637.

[21] Polychronis Manousopoulos, Vassileios Drakopoulos, and Theoharis Theoharis. *Curve Fitting by Fractal Interpolation*, pages 85–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-79299-4. doi: 10.1007/978-3-540-79299-4_4. URL https://doi.org/10.1007/978-3-540-79299-4_4.

[22] Phil Picton. *ADALINE*, pages 13–24. Macmillan Education UK, London, 1994. ISBN 978-1-349-13530-1. doi: 10.1007/978-1-349-13530-1_2. URL https://doi.org/10.1007/978-1-349-13530-1_2.

[23] Sebastian Raubitzek and Thomas Neubauer. A fractal interpolation approach to improve neural network predictions for difficult time series data. *Expert Systems with Applications*, 169:114474, 2021. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2020.114474. URL https://www.sciencedirect.com/science/article/pii/S0957417420311234.

[24] Robert N. Rodriguez and Yonggang Yao. Five things you should know about quantile regression. URL https://support.sas.com/resources/papers/proceedings17/SAS0525-2017.pdf.

[25] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

[26] Sagar Sharma. Activation functions in neural networks, Nov 2022. URL https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[27] Huan Song, Deepta Rajan, Jayaraman Thiagarajan, and Andreas Spanias. Attend and diagnose: Clinical time series analysis using attention models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. doi: 10.1609/aaai.v32i1.11635. URL https://ojs.aaai.org/index.php/AAAI/article/view/11635.

[28] Sivaram T. All you need to know about skip connections, Aug 2021. URL https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/.

[29] Ryan Tibshirani. Gradient descent: Convergence analysis, Sep 2013. URL https://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

[31] Wen, Torkkola, Narayanaswamy, and Madeka. A multi-horizon quantile recurrent forecaster. Jun 2018. URL https://arxiv.org/abs/1711.11053.

[32] Binrong Wu, Lin Wang, and Yu-Rong Zeng. Interpretable wind speed prediction with multivariate time series and temporal fusion transformers. *Energy*, 252:123990, 2022. ISSN 0360-5442. doi: https://doi.org/10.1016/j.energy.2022.123990. URL https://www.sciencedirect.com/science/article/pii/S0360544222008933.

[33] Vishal Yathish. Loss functions and their use in neural networks, Aug 2022. URL https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9.