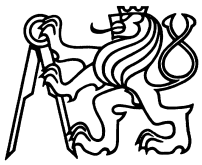
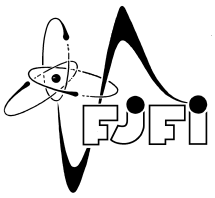


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



Prediction of energy demand in the power system with deep learning multi horizon forecasting methods

Predikce zatížení v elektroenergetické přenosové soustavě pomocí metod hlubokého učení pro více horizontů

Master's Degree Project

Author: **Yana Podlesna**

Supervisor: **Ing. Jiří Franc, Ph.D..**

Academic year: 2022/2023

Acknowledgment:
I would like to thank my supervisor, who, with his wisdom, patience, and confidence in my abilities, provided indispensable guidance during my thesis journey.

Author's declaration:
I declare that this Master's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 2, 2023

Yana Podlesna

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství

Akademický rok 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	**Bc. Yana Podlesna**
Studijní program:	**Aplikace přírodních věd**
Obor:	**Aplikace softwarového inženýrství**
Název práce česky:	**Predikce zatížení v elektroenergetické přenosové soustavě pomocí metod hlubokého učení pro více horizontů**
Název práce anglicky:	**Prediction of energy demand in the power system with deep learning multi horizon forecasting methods**

Pokyny pro vypracování:

1. Nastudujte problematiku predikce zatížení v elektroenergetické přenosové soustavě.
2. Proveďte rešerši používaných metod pro predikci časových řad na více kroků dopředu.
3. Seznamte se s metodami hlubokého učení používající transformery a architektury založené na pozornosti.
4. Aplikujte výše uvedené metody na reálná data z elektroenergetické přenosové soustavy a porovnejte kvalitu modelů pro různě dlouhé predikční horizonty a pro různou granularitu měření.

Doporučená literatura:

[1] Goodfellow, Y. Bengio, A. Courville, Deep Learning: Adaptive Computation and Machine Learning series. The MIT Press, 2016.

[2] A Géron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media, 2019.

[3] Bryan Lim, Sercan Ö. Arık, Nicolas Loeff, Tomas Pfister, Temporal Fusion Transformers for interpretable multi-horizon time series forecasting, International Journal of Forecasting, Volume 37, Issue 4, 2021

[4] Nti, I.K., Teimeh, M., Nyarko-Boateng, O. et al. Electricity load forecasting: a systematic review. Journal of Electrical Systems and Inf Technol 7, 13 (2020).

[5] Nielsen, Practical Time Series Analysis: Prediction with Statistics and Machine Learning. O'Reilly Media, 2019.

Jméno a pracoviště vedoucího práce:

Ing. Jiří Franc, Ph.D.
Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze

Jméno a pracoviště konzultanta:

–

.....
vedoucí práce

Datum zadání diplomové práce: 1. 2. 2022

Termín odevzdání diplomové práce: 5. 1. 2023

Doba platnosti zadání je dva roky od data zadání.

.....
garant oboru

.....
vedoucí katedry

.....
děkan

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-1-

V Praze dne 1. 2. 2022

Název práce:

Predikce zatížení v elektroenergetické přenosové soustavě pomocí metod hlubokého učení pro více horizontů

Autor: Yana Podlesna

Obor: Aplikace přírodních věd

Zaměření: Aplikace softwarového inženýrství

Druh práce: Diplomová práce

Vedoucí práce: Ing. Jiří Franc, Ph.D., Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Abstrakt: Tato práce se zabývá metodami hlubokého učení se zaměřením na Temporal Fusion Transformer (TFT) pro predikci zatížení v elektroenergetické přenosové soustavě pro více horizontů. Navrhovaná architektura modelu TFT pro zpracování časových řad zahrnuje dynamickou síť pro výběr proměnných, časové zpracování pomocí vrstvy kodéru dekodéru a mechanismus pozornosti. Výkonnost modelu TFT je porovnávána s tradičními modely strojového učení, jako jsou XGBoost a Random Forest, a je hodnocena na základě předpovědí pro denní trh (24 hodin dopředu) a mezidenní trh (6 hodin dopředu). Výsledky ukazují potenciál pokročilých metod hlubokého učení při zlepšování řízení energetické soustavy v kontextu rostoucí integrace obnovitelných zdrojů energie.

Klíčová slova: hluboké učení, pozornost, předpověď na více horizontů, strojové učení, temporal fusion transformer

Title:

Prediction of energy demand in the power system with deep learning multi-horizon forecasting methods

Author: Yana Podlesna

Abstract: This thesis addresses deep learning methods with a focus on Temporal Fusion Transformer (TFT) for multi-horizon load prediction in the power transmission system. The TFT model architecture proposed for time series processing includes dynamic variable selection network, temporal processing using encoder decoder layer and attention mechanism. The performance of the TFT model is compared with traditional machine learning models such as XGBoost and Random Forest, and is evaluated based on forecasts for the daily market (24 hours ahead) and the intraday market (6 hours ahead). The results show the potential of advanced deep learning methods in improving power system management in the context of increasing integration of renewable energy sources.

Key words: attention, deep learning, machine learning, multi-horizon forecasting, temporal fusion transformer

Contents

Introduction **10**

1 Machine Learning Foundations **13**
1.1 Decision Tree-Based Models . . . 14
1.1.1 Random Forest . . . 15
1.1.2 XGBoost . . . 15
1.2 Deep Learning Models . . . 16
1.2.1 Neural Network Basics . . . 16
1.2.2 Transformer based Models . . . 22
1.3 Performance Measures for Regression Analysis . . . 25
1.4 Embeddings and Feature Engineering . . . 26

2 Temporal Fusion Transformer Model **29**
2.1 Gated Residual Network . . . 31
2.2 Variable selection network . . . 32
2.3 Static Covariate Encoders . . . 33
2.4 Temporal Processing . . . 33
2.4.1 Locality Enhancement with Sequence-to-Sequence Layer and Static Enrichment 33
2.4.2 Temporal Self-Attention Layer . . . 34
2.4.3 Position-wise Feed-forward Layer . . . 35
2.5 Quantile Outputs . . . 35
2.6 Loss Functions . . . 36
2.7 Hyperparameters . . . 36

3 Implementation and Results **37**
3.1 Data collection and preprocessing . . . 39
3.2 Day-Ahead Market . . . 42
3.3 Intra-Day Market . . . 47

Conclusion **52**

Introduction

In a world, where resources are shrinking, more emissions is emitted every year and renewable energy generation yet to be deployed worldwide, the importance of energy demand management is vital as it gives the basis for giving decisions in power system planning and operation. Energy demand forecasting plays a important role in energy supply-demand management for both governments and private companies. Therefore, using models to accurately forecast the future energy consumption trends – specifically with nonlinear data – is an important issue for the power production and distribution systems. Matching electrical energy consumption with the right level of supply is crucial, because excess electricity supplied cannot be stored, unless converted to other forms, which incurs additional costs and resources. At the same time, underestimating energy consumption could be fatal, with excess demand overloading the supply line and even causing blackouts.

The accurate forecasting of energy demand, a variable influenced by temporal, climatic, socioeconomic, and demographic parameters, is an essential and challenging task [1]. Traditional forecasting techniques such as ARIMA [2] often fall short in dynamically changing environments and different types of data, necessitating more sophisticated methods. Time-series data present their unique challenges, varying from stationary, where characteristics remain steady over time, to non-stationary, wherein trends and seasonality add complexity to the forecasting process.

Moreover, forecasting often encompass a big variety of data sources, including known future information, exogenous time series, and static metadata, all of which interact in ways that may not be apparent a priori. This data heterogeneity, combined with a dearth of knowledge about their interactions, adds to the complexity of time series forecasting. Prediction of future values is often confronted with various challenges, when performing multi-step forecasts including the risk of error propagation, the requirement for sophisticated models capable of discerning temporal dependencies, and an inherent increase in uncertainty with more distant forecasts.

In the realm of energy demand forecasting, both traditional machine learning models and more advanced techniques have their place. Models such as XGBoost [3] and Random Forest [4], despite their simplicity, have been effective in many scenarios, thanks to their ability to capture non-linear relationships. However, they don't inherently handle the temporal dependencies present in time series data, which can limit its performance. These models will serve as baseline for comparison with more advanced methods.

Time-series forecasting approaches based on deep learning have significantly grown in recent years, with the development in neural network algorithms, available data, and hardware power. While earlier models like Recurrent Neural Networks (RNNs) [5] introduced the concept of preserving temporal information via iterative hidden states, they suffered from the vanishing gradient problem during long-term sequence simulations [6]. This issue was somewhat mitigated by improvements such as Long Short-Term Memory (LSTM) networks [7] and Gated Recurrent Units (GRUs) [8], which introduced gating mechanisms to selectively manage long-term memory.

Yet, the emergence of Transformer-based models [9], known for their unique attention mechanism, represents a contemporary and popular trend reshaping the field of time-series analysis. These models efficiently solved the long-term dependency problem by assigning weights that map the influence of prior time frames onto future targets, offering not only interpretability, but also finding applications in key artificial intelligence domains like natural language processing and computer vision [10].

One notable evolution of the Transformer model is the Temporal Fusion Transformer (TFT) [11]. It is attention-based framework that harmoniously combines multi-horizon prediction performance with an intelligible understanding of temporal dynamics. To identify temporal relationships at different scales, TFT uses recurrence layers that deal with immediate processing and self-awareness layers that can understand long-term dependencies in an interpretable way. The TFT model includes special components for feature selection and an array of gating layers to muting unnecessary elements, providing high performance in a wide range of scenarios. This study sets out to examine the proficiency of modern deep learning methodologies, with a special focus on the TFT, in accurately predicting future energy consumption. Rigorous experiments will focus on two distinct forecasting horizons: a short-term 6-hour prediction for intra-day market operations and a more extended 24-hour forecast for daily market planning.

The first chapter serves as a primer, establishing the necessary background and context for understanding the intricacies of electricity load forecasting methods. It aims to equip the reader with a fundamental understanding of the field, setting the stage for the more advanced concepts and techniques discussed in the subsequent chapters.

The second chapter thoroughly explores the TFT. It details its architectural specifics, key benefits, and potential areas for enhancement. The aim here is to provide an in-depth understanding of this deep learning-based methodology and its potential role in advancing the field of time-series forecasting.

The third and final chapter presents the experiments conducted in this study and their results. It provides a comparative analysis of the TFT and baseline models, specifically XGBoost and Random Forest, in the context of 6-hour and 24-hour energy demand forecasting. The ensuing discussion aims to provide a clear and precise understanding of the experimental results and their implications for the theory and practice of energy load forecasting.

Chapter 1

Machine Learning Foundations

In recent years, machine learning has emerged as a powerful tool for analyzing and making predictions from complex datasets. With its ability to uncover hidden patterns and relationships, machine learning has found applications in various domains, including finance, healthcare, and energy markets. Time series forecasting plays a crucial role in predicting future values of a variable based on its past observations. Accurate and reliable time series forecasting is essential for decision-making, resource allocation, and risk management. This complex but highly important task can be effectively addressed using machine learning techniques [13].

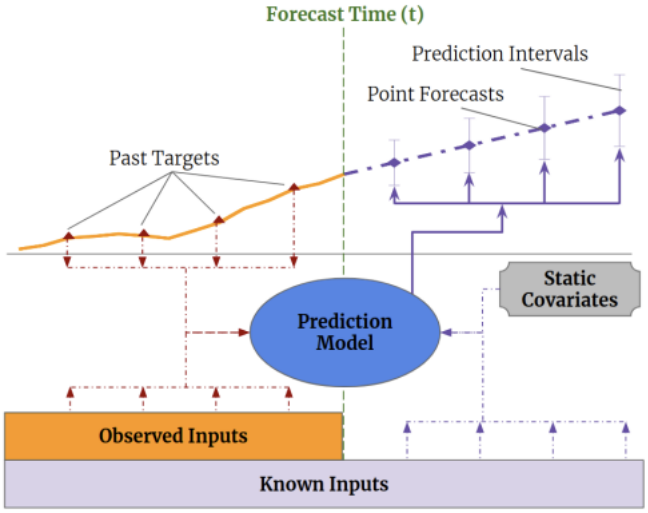


Figure 1.1: Diagram representing time-series forecasting, incorporating static covariates, past-observed inputs, and a priori-known future time-dependent inputs, adapted from [11].

In practical applications of time series forecasting, as shown in Figure 1.1, there is often access to multiple data sources. These may include expected future information, other external time series, and static metadata, all without any prior knowledge of their interactions. The diversity of these data sources, combined with a limited understanding of their interactions, greatly increases the complexity and challenges inherent in time series forecasting.

This thesis begins with introduction to decision tree-based models, specifically Random Forest and XGBoost, used as a baseline in this study. Section 1.2 introduces main concepts of neural networks needed for better understanding more advanced complex models. Building on these foundations, Section

1.3 then elaborates on the performance measures of regression analysis, which allows regression models to be efficiently estimated and compared. Section 1.4 delves into feature engineering, highlighting the importance of embeddings for categorical variables and discussing techniques to identify influential features.

1.1 Decision Tree-Based Models

Decision trees are a popular machine learning technique used for classification and regression tasks. They have been widely applied in various domains due to their interpretability, ease of use, and ability to handle both categorical and continuous input features. In recent years, decision trees have also gained attention for their potential in multi-step forecasting, where the task is to predict multiple future time steps given a set of historical data, including energy demand forecasting [14], [15], stock price prediction, and weather forecasting.

The goal of decision tree algorithms is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. Each node in the tree represents prediction, and each branch represents a possible outcome of that prediction. The tree starts at the top with a root node, and branches out into smaller sub-trees as the algorithm makes decisions and reaches the final prediction or decision. The root node represents the entire population or sample, and the branches represent the possible outcomes of the first decision. Each sub-tree that branches off from the root represents a subset of the population or sample. The process continues as the algorithm makes decisions at each node and reaches a leaf node that represents the final prediction or decision.

In regression tasks, the decision tree algorithm is a popular approach that involves examining all attributes and their respective values to identify the most optimal split. The objective or cost function for regression problems is usually the mean squared error, which needs to be minimized. Essentially, this is equivalent to using variance reduction as a feature selection criterion.

After identifying the best candidate split point, the dataset is split at that value (i.e., root node), and the attribute selection process is repeated for the other ranges. The algorithm continues this iterative process until either the terminal or leaf nodes reach every sample (i.e., no stopping criteria), or a specific stopping criterion is met. For example, a maximum depth may be set, allowing only a limited number of splits from the root node to the terminal nodes. Alternatively, a minimum number of samples may be specified in each terminal node to prevent excessive splitting beyond a certain point.

In recent years, ensemble methods have been widely adopted in the field of machine learning due to their ability to improve predictive accuracy and generalization performance compared to single models. Ensemble methods, which train multiple models on different subsets of the data or using different algorithms, can better capture the complexity and variability of the data than a single model. One reason for their success is that they compensate for the individual weaknesses of models like decision trees, which tend to have high variance and low bias, making them prone to overfitting. When these models are combined into an ensemble, their individual weaknesses are offset, resulting in models with reduced variance and improved predictive power. The most well-known ensemble methods are bagging [16], also known as bootstrap aggregation, and boosting [17].

Let $X \in \mathbb{R}^{d_x \times n}$ be a training set with $n \in \mathbb{N}$ samples, each having $d_x \in \mathbb{N}$ features. In bagging method a random sample of data of size $d_s \in \mathbb{N}$ from a training set X is selected with replacement creating $m \in \mathbb{N}$ new training subsets of the data $X_i \in \mathbb{R}^{d_x \times d_s}, i \in \{1, \ldots, m\}$ called "bags". After several bags are generated, $m \in \mathbb{N}$ models f_i are fitted using the above m bootstrap samples X_i and trained independently, and for regression task the average of those predictions yield a more accurate estimate. By training multiple instances of the model on different subsets of the data, bagging helps to reduce the correlation between

the models, thus decreasing the variance of the ensemble within a noisy dataset. Bagging is widely used to combine the results of different decision trees models and build the random forests algorithm.

Boosting is an ensemble modeling technique that attempts to build a strong classifier from the number of weak classifiers. Unlike bagging which is a parallel ensemble method, boosting methods are sequential ensemble algorithms where the weights $w_i \in \mathbb{R}, i \in \{1, \ldots, m\}$ are depending on the previous fitted models $f_1, \ldots, f_m$. Gradient boosting uses gradient descent to optimize an objective function by sequentially adding weak models. The basic idea behind boosting is to train the first model on the original dataset and then when training a new model in each iteration to identify miss-classified data points, increasing their weights so that the next classifier will pay extra attention to get them right. Then final output is a weighted sum of the individual models' outputs.

The main advantage of boosting is that it can reduce the bias of the model, making it more accurate. By training a sequence of models and focusing on the mistakes made by the previous models, boosting helps to improve the overall accuracy of the ensemble.

1.1.1 Random Forest

Random Forest is an ensemble learning method introduced in [4], that constructs multiple decision trees and aggregates their predictions to enhance the final prediction's robustness and accuracy. It incorporates the bagging technique and the principle of random subspace selection to induce diversity among the individual trees.

For a given training set $X \in \mathbb{R}^{d_x \times n}$, Random Forest starts by constructing $m \in \mathbb{N}$ decision trees. While growing these trees, at each node, instead of searching for the best feature across all features, Random Forest selects the best feature from a randomly chosen subset of size $k \in \mathbb{N}$ of all features (usually of size $k = \frac{d_x}{3}$ for regression tasks). Then each decision tree is built from a bootstrap sample $X_i \in \mathbb{R}^{k \times d_x}$ created by sampling $s \in \mathbb{N}$ instances randomly with replacement from the original dataset X.

This introduces the desired randomness and decorrelates trees, contributing to model generalization.

For a new data point, the output of the Random Forest model is determined by the outputs of all the individual decision trees. For regression tasks, the output is the average prediction $\hat{y}$ of all the trees

$$\hat{y} = \frac{1}{m} \sum_{i=1}^{m} \hat{y}_i, \tag{1.1}$$

where $\hat{y}_i$ is the prediction of the i-th decision tree. This ensemble of decision trees typically results in a model with better predictive performance and less susceptibility to overfitting compared to a single decision tree. The injected randomness in the model creation also helps in dealing with high dimensional datasets effectively and provides a measure of feature importance.

1.1.2 XGBoost

XGBoost, short for eXtreme Gradient Boosting, is a scalable and efficient implementation of the gradient boosting framework. XGBoost was introduced [3] to address the computational challenges associated with the traditional gradient boosting algorithm and enhance its predictive performance.

The main idea behind XGBoost, similar to other gradient boosting algorithms, is to add new models to the ensemble sequentially. Each new model attempts to correct the errors made by the sum of the preceding models. Specifically, each new model fits the residuals from the sum of previous models to gradually reduce the prediction error. XGBoost leverages gradient boosting's concept but introduces several enhancements to optimize efficiency and predictive power. It achieves this by incorporating a more regularized model formalization that controls overfitting. A notable characteristic of XGBoost is

its use of a second-order Taylor expansion to approximate the loss function. Unlike traditional gradient boosting, which only considers the first-order derivative, XGBoost takes into account the second-order derivative as well. This provides a more accurate approximation of the loss function. It not only considers the residuals but also the rate of change of these residuals, optimizing the direction and magnitude of adjustments made to the model.

XGBoost also introduces a novel way to measure the quality of a split in a decision tree. It calculates a "gain" value that considers both the first and second-order gradients. A higher gain value suggests a better split, which helps guide the algorithm's decisions when constructing the decision trees. The addition of regularization terms in the model helps to control the complexity of the model and prevent overfitting. This regularization is often not present in traditional gradient boosting and contributes significantly to the improved performance observed with XGBoost.

1.2 Deep Learning Models

Deep learning, a subfield of machine learning, has influenced the progress of artificial intelligence in recent years. Its impact is primarily attributed to the use of neural networks - complex mathematical models that have shown exceptional performance in a variety of tasks [18]. These tasks span a diverse range of fields, from computer vision to natural language processing, and even speech recognition. Importantly, neural networks have also proven to be highly effective in regression-based tasks, where they are capable of modeling and understanding complex patterns in temporal data [19].

1.2.1 Neural Network Basics

A neural network is a mathematical model that approximates a function by means of a set of interconnected processing units, called neurons. Within this layered structure, every neuron is engineered to accept input data, perform a non-linear transformation upon these inputs using an activation function, and consequently generate an output as it shown at Figure 1.2.

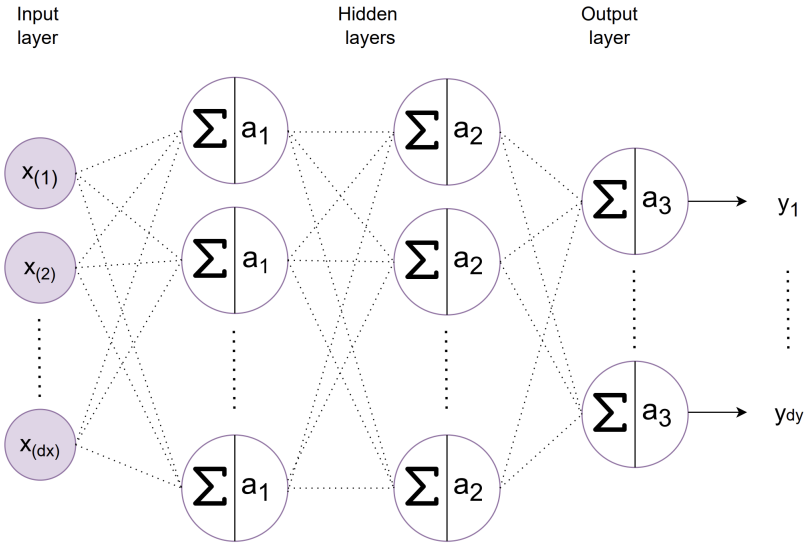


Figure 1.2: Neural Network with 3 layers.

Let $x \in \mathbb{R}^{d_x}$ be input vector, where $d_x \in \mathbb{N}$ is number of features. The mathematical representation of neuron $j \in \mathbb{N}$ in a neural network layer $i \in \mathbb{N}$ with $d_l \in \mathbb{N}$ neurons is

$$z_j^{(i)} = w_j^{(i)} x^{(i-1)} + b_j^{(i)}, \quad \forall j \in \{1, 2, \ldots, d_l\}, \tag{1.2}$$

$$x_{(j)}^{(i)} = a_i(z_j^{(i)}), \tag{1.3}$$

where $w_j^{(i)} \in \mathbb{R}^{d_x}$ is the weighted vector, $b_j^{(i)} \in \mathbb{R}$ is the bias term, $z_j^{(i)} \in \mathbb{R}^{d_l}$ is the pre-activation value, a_i is the activation function and $x_{(j)}^{(i)}$ is the output value of j-th neuron in i-th layer. Last layer of neural network then returns target vector $x^{(last)} := y \in \mathbb{R}^{d_y}$.

The activation function a, crucial for introducing non-linearity into the model and thereby enabling it to handle the complexities of real-world data, remaps the sum of a neuron's inputs into a defined range, allowing the network to recognize complex patterns. Commonly used activation functions are for example the sigmoid, hyperbolic tangent (tanh), ReLU (Rectified Linear Unit), and softmax, each with its unique characteristics and applications [20].

Following the non-linear transformation via the activation function, the output is generated. This output then either feeds into subsequent neurons in the network's hierarchy or directly contributes to the final output layer.

The weight vector and bias term are learned during training with backpropagation algorithm [21] and determine the neuron's behavior. It's done during training by minimizing a loss function L, which measures the difference between the predicted output of the neural network and the actual output, providing a measure of the network's performance. The choice of the appropriate loss function is important as it influences how well the network can learn from the data and make accurate predictions. This minimization is accomplished through an iterative process, typically employing gradient descent [22] or a variant thereof, where the gradients of the loss function with respect to the weights are used to iteratively update the weights and biases in the network.

The learning rate $\eta \in \mathbb{R}$ is hyperparameter that controls the step size during each iteration of the optimization process. A high learning rate may cause the algorithm to converge quickly, but it might overshoot the optimal solution and lead to unstable training due to large updates to the weights. On the other hand, a low learning rate may cause the algorithm to converge slowly, requiring many updates to the weights and a long time to reach the optimal solution, or it could get stuck in a suboptimal solution if it converges too early. There have been a lot of methods proposed to find the optimal learning rate such as Adaptive Moment Estimation [23], Ranger Optimizer [24] or One-cycle Policy [25].

Neural network with high number of hidden layers called deep neural network. Deep neural networks are capable to learn very complicated relationships between their inputs and outputs. However, when training data is limited, these networks can lead to overfiting and learning patterns that are specific to the training set but do not generalize well to new data. Various methods have been developed for reducing it. These includes stopping the training as soon as performance on a validation set starts to get worse [26], introducing weight penalties of various kinds such as L1 and L2 regularization [27], weight sharing which promotes parameter sharing between different parts of the network [28], or dropout [29], which randomly drops out weights during training to prevent reliance on specific features.

Another popular technique when dealing with deeper neural networks is residual connection. This approach, introduced in [30] proposes a design strategy that allows the construction of much deeper networks while mitigating the vanishing gradient problem [31] and preserving the network's ability to learn identity functions. Residual connections function by adding a "shortcut" or "skip connection" that allows the gradient to be directly backpropagated to earlier layers. It's done by transformation

$$x_{(j)}^{(i)} = a_i(z_j^{(i)}) + x_{(j)}^{(i-1)}, \quad \forall j \in \{1, \ldots, d_l\}, \tag{1.4}$$

where $x_{(j)}^{(i-1)}$ is output of neuron j layer $i-1$ and input for layer i, $x_{(j)}^{(i)}$ is output of layer i and function $a_i(z_j^i)$ represents the residual mapping to be learned. Rather than learning the direct mapping $x_{(j)}^{(i)} = a_i(z_j^{(i)})$, the network learns the residual mapping $a_i(z_j^{(i)}) = x_{(j)}^{(i)} - x^{(i-1)}$, effectively making the network learning process easier.

Training deep neural networks with significant number of hidden layers could be also challenging due to the internal covariate shift [32] which refers to change in the distribution of inputs during training. This phenomenon leads to slower training progress by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. To adress this issue techniques such as layer normalization [33] were proposed.

Layer normalization aims to address the internal covariate shift by enforcing zero mean and unit standard deviation for the inputs or pre-activations of layer. This is achieved by subtracting the mean of each input feature across the input and dividing it by the standard deviation. This normalization is applied independently to each instance within a layer, normalizing across the feature dimensions. To reduce the potential restrictions imposed by enforcing zero mean and unit standard deviation across all batches, two learnable parameters are incorporated: a scaling factor $\gamma \in \mathbb{R}$ and an offset $\beta \in \mathbb{R}$. These learnable parameters enable the network to adaptively adjust the normalized pre-activations, allowing for greater flexibility in capturing class-specific patterns.

Given input vector of the layer i as $x^{(i-1)}$, layer normalization for layer i can be expressed as follows:

$$\hat{x}^{(i)} = \gamma \frac{x^{(i-1)} - \mu_i}{\sigma_i} + \beta, \tag{1.5}$$

where μ_i is mean and σ_i is standart deviation across the layer i. The values of $\gamma \in \mathbb{R}$ and $\beta \in \mathbb{R}$ are learned during the training process and are used to fine-tune the normalized activations according to the specific requirements of the network and the task. This makes layer normalization well suited for sequence models such as transformers and recurrent neural networks (RNNs) that will be discussed futher.

Recurrent Neural Networks

Traditional machine learning models, may struggle to capture temporal dependencies, relationships or correlations between data points that are separated by time, for time series analysis and prediction tasks, such as forecasting electricity demand. Temporal data refers to a series of data points indexed in time order, typically at successive equally spaced points in time, denoted as $\{x_t\}_{t=1}^{T}$, where x_t is the observed data point at time step $t \in \mathbb{N}$ and $T \in \mathbb{N}$ is the length of the time series.

This is where recurrent neural networks [5], a type of deep learning model, excel. Unlike feedforward neural networks, which process each input independently, RNNs maintain an internal memory, or hidden state $h \in \mathbb{R}^{d_h}$, (with $d_h \in \mathbb{N}$ as a hidden size), that summarizes the information observed so far. This hidden state allows RNNs to consider past inputs when making predictions, enabling them to capture dependencies over time. In the context of an RNN's sequential architecture, the hidden state acts as an input to the subsequent unit in the chain, carrying forward the information from previous steps. Generalized algorithm of RNN is shown in Figure 1.3

In the most basic type of RNN with one layer, the hidden state $h_t = (h_{1t}, h_{2t}, \ldots, h_{d_h t})$ at time step t is calculated based on the input at the current time step x_t and the previous hidden state h_{t-1} using following equation

$$z_{jt} = w_{jh} h_{t-1} + w_{jx} x_t + b_{jh}, \tag{1.6}$$

$$h_{jt} = a_h(z_{jt}), \qquad \forall j \in \{1, \ldots, d_h\}, \tag{1.7}$$

where $w_{jh} \in \mathbb{R}^{d_h}$ is the weight vector that determines the impact of the previous hidden state on the current hidden state, $w_{jx} \in \mathbb{R}^{d_x}$ is the weight vector that determines the impact of the current input on

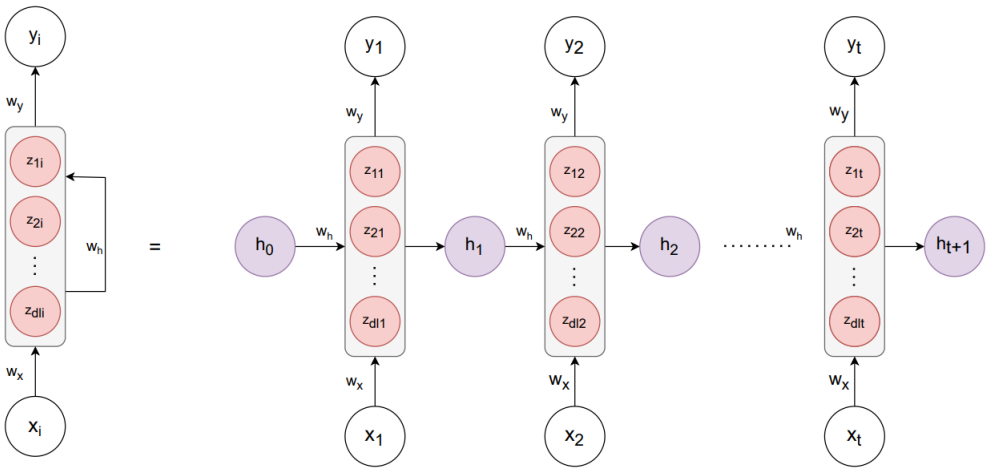


Figure 1.3: RNN with one hidden layer.

the current hidden state, $b_{jh} \in \mathbb{R}$ is bias term and a_h is the activation function applied to the sum of the weighted inputs and biases for the hidden state. Weight vectors are shared throughout the entire network. After output is calculated based on the current hidden state h_t

$$y_{jt} = a_y(w_{jy}h_t + b_{jy}), \qquad \forall j \in \{1, \ldots, d_y\} \tag{1.8}$$

where $w_y \in \mathbb{R}^{d_h}$ is the weight matrix (made from weighted vectors) that determines the impact of the current hidden state on the output, $b_{jy} \in \mathbb{R}$ are the bias terms and a_y is the activation function.

Unfortunately, it has been observed by [31] that it is difficult to train RNNs to capture long-term dependencies because the gradients tend to either vanish or explode. The update to the weights, say w_{jh}, involves calculating the gradient of the loss function L with respect to these weights. Using the chain rule of differentiation, this gradient is found to be a product of terms, each involving a Jacobian matrix J that is derived from the hidden state h_t. Specifically, each entry of J is given by $\frac{\partial h_{t+1,i}}{\partial h_{t,k}}$, where i, k are indices of J

$$\frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial h_t} = \frac{\partial L}{\partial h_{t+1}} \cdot J. \tag{1.9}$$

Depending on the magnitude of the largest eigenvalue of J, the gradient can either explode (if the eigenvalue is greater than 1) or vanish (if the eigenvalue is less than 1). When processing long sequences, this results in the gradient being scaled exponentially at each time step, leading to difficulties in learning long-term dependencies. This problem is called gradient vanishing problem.

There have been two dominant approaches by which many researchers have tried to reduce the negative impacts of this issue. One such approach is to devise a better learning algorithm than a simple stochastic gradient descent [34], [35]. The other approach, is to design a more sophisticated activation function than a usual activation function, consisting of affine transformation followed by a simple element-wise nonlinearity by using gating units. These help control the flow of information through the network, making the recurrent unit structure more complex and potentially more powerful. The earliest attempt in this direction resulted in an activation function, or a recurrent unit, called a long short-term memory (LSTM) [7].

LSTM

Long Short Term Memory networks (LSTM) are a special kind of RNN proposed to solve problem with learning of long-term dependencies by [7], and were refined and popularized by many people in following work. They work well on a large variety of problems, and are now widely used. They demonstrate high efficacy across a diverse range of problem domains and have gained widespread adoption in contemporary applications.

Recurrent neural networks follow a structure consisting of a series of self-connected reccurent units. In the case of standard RNNs, these repeating units usually have a simple structure, often including only simple activation function, such as tahn. LSTMs also leverages same repeating structure, but recurrent unit has more complex construction facilitates capturing long-term dependencies and solves the problem of vanishing.

The key to LSTMs is the cell state $C \in \mathbb{R}^{d_h}$, which retains and transports relevant information across different time steps. The LSTM reccurent unit remove or add information to the cell state, carefully regulated by structures called gates.

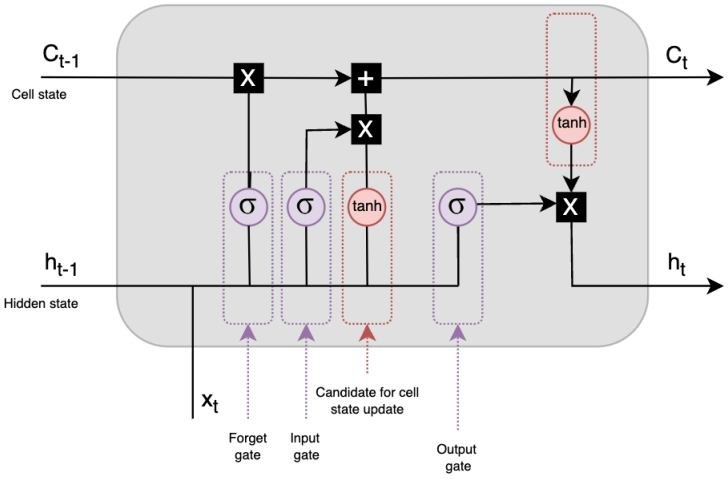


Figure 1.4: LSTM architecture.

As illustrated in the Figure 1.4, information traverses three gates: the forget, input, and output gates. A sigmoid neural net layer and a pointwise multiplication operation regulate the information flow through these gates. The sigmoid layer outputs values within the range of zero to one, indicating how much each component should be allowed through.

First the forget gate determines which pieces of information needs attention and which can be safely disregarded. Information from the current input $x_t \in \mathbb{R}^{d_x}$ and prior hidden state $h_{t-1} \in \mathbb{R}^{d_h}$ are passed through the sigmoid function assigning weights to these inputs in range from 0 to 1. Higher values, closer to 1, indicate the importance of retaining the corresponding information in the subsequent computations. This value of f_t will later be used by the cell for point-by-point multiplication. The LSTM model's forget gate is characterized by the following equation

$$f_t = \sigma(\mathbf{W}_f[h_{t-1}, x_t] + b_f), \tag{1.10}$$

where $x_t \in \mathbb{R}^{d_x}$ symbolizes the current timestamp input, $h_{t-1} \in \mathbb{R}^{d_h}$ denotes the previous timestamp's hidden state, $\mathbf{W}_f \in \mathbb{R}^{d_h \times (d_h + d_x)}$ is the weight matrix connected to input x_t and hidden state h_{t-1}, and $b_f \in \mathbb{R}^{d_h}$ represents the bias term.

The input gate decides which new information should be stored in the cell state and subsequently used to update it. First, the current input x_t and the previous hidden state h_{t-1} are passed through a sigmoid function, which transforms the values between 0 (indicating unimportant) and 1 (indicating important) decides which values of the cell state C_t will be updated.

Next, the same information from the hidden state and the current input is passed through a hyperbolic tangent (tanh) function. The tanh function outputs a vector of new candidate values $\tilde{C}_t \in \mathbb{R}^{d_h}$, with values ranging from -1 to 1, which could be added to the state. The equations below illustrate the calculations performed by the input gate in the LSTM model

$$i_t = \sigma(\mathbf{W}_i[h_{t-1}, x_t] + b_i), \tag{1.11}$$

$$\tilde{C}t = \tanh(\mathbf{W}_c[h_{t-1}, x_t] + b_c), \tag{1.12}$$

where i_t represents the input gate activation, $\tilde{C}_t \in \mathbb{R}^{d_h}$ denotes the candidate cell state, $\mathbf{W}_i, \mathbf{W}_c \in \mathbb{R}^{d_h \times (d_h + d_x)}$ are weight matrices linked to input x_t and hidden state h_{t-1}, and $b_i, b_c \in \mathbb{R}^{d_h}$ are the respective bias terms.

The cell state is then updated

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t. \tag{1.13}$$

Lastly, the output gate determines the next hidden state h_t, considering the updated cell state. Similar to the input gate, it includes sigmoid and tanh layers. The sigmoid layer first decides which portions of the cell state should be retained. Following this, a tanh function is applied to scale the values between -1 and 1. This scaling serves as a form of normalization, preventing the network's activations from becoming excessively large or small, which could lead to problems such as exploding or vanishing gradients during backpropagation. This process can be expressed mathematically as:

$$o_t = \sigma(\mathbf{W}_o[h_{t-1}, x_t] + b_o), \tag{1.14}$$

$$h_t = o_t \tanh(C_t). \tag{1.15}$$

In these equations, x_t is the input at the current timestamp t, h_{t-1} represents the hidden state at the previous timestamp, $\mathbf{W}_o \in \mathbb{R}^{d_h \times (d_h + d_x)}$ is the output gate's weight matrix for input and hidden state at the previous timestamp, and $b_o \in \mathbb{R}^{d_h}$ is the output gate bias. Here, $C_t \in \mathbb{R}^{d_h}$ is the updated cell output.

Each gate (forget, input, output) within the LSTM cell possesses its own set of parameters (weights and biases), which are learned during the training process.

Encoder Decoder based models

RNNs can effectively map sequences to sequences when the alignment between inputs and outputs is known in advance. However, their application becomes challenging for problems where the input and output sequences have different lengths and exhibit complex, non-monotonic relationships. The encoder-decoder model was proposed to address these issues [6], [36]. The fundamental principle of this architecture lies in consisting of two RNNs that act as an encoder and a decoder pair as it shown at Figure 1.5.

Encoder takes the input data and compresses it into a fixed-size representation, also referred to as a context vector. The goal of the encoder is to effectively capture all the useful information from the input data in this context vector.

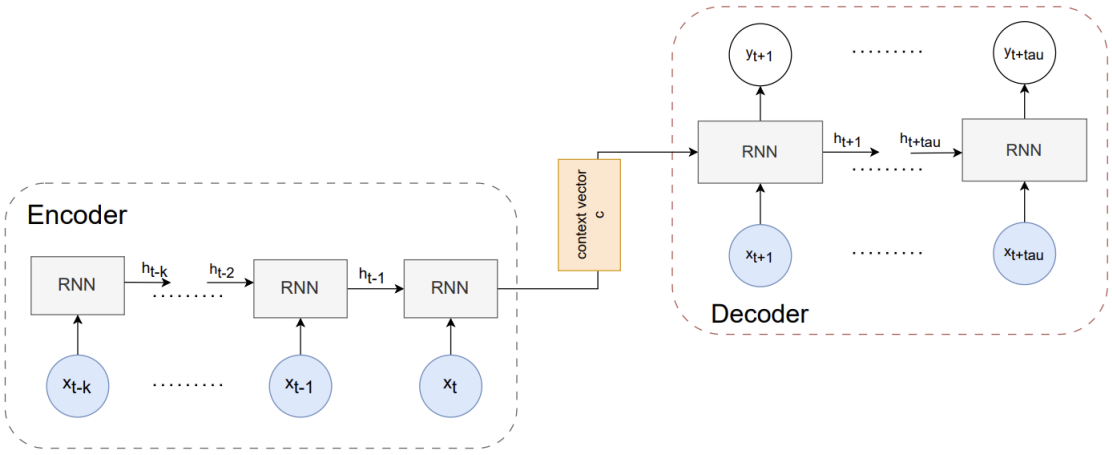


Figure 1.5: Encoder Decoder model architecture.

Given an input sequence $\chi_t \in \mathbb{R}^{d_x}$ and targets $y_t \in \mathbb{R}^{d_y}$ at each time-step $t \in [0, T]$. Time-dependent input features are subdivided into two categories $\chi_t = [n_t^T, x_t^T]^T$ – observed inputs $x_t \in \mathbb{R}^{d_x}$ which can only be measured at each step and are unknown beforehand, and known inputs $n_t \in \mathbb{R}^{d_n}$ which can be predetermined. The encoding process involves processing $k \in \mathbb{N}$ input time steps $\{x_1, x_2, \ldots, x_k\}$ with known inputs $\{n_1, n_2, \ldots, n_k\}$. At each time step t, the encoder updates its hidden state h_t based on the current input x_t, the known input n_t, and its previous hidden state h_{t-1} as follows:

$$h_t = a_e(x_t, n_t, h_{t-1}), \quad \forall t \in [1, k], \tag{1.16}$$

where a_e is the nonlinear transformation performed by the encoder. After processing all k inputs, the final hidden state h_k is used as the context vector $c = h_k \in \mathbb{R}^{d_h}$ that encapsulates the information from the input sequence.

The decoder then utilizes this context vector c to generate a sequence of $\tau \in \mathbb{N}$ future predictions $\{y_{k+1}, y_{k+2}, \ldots, y_{k+\tau}\}$, also taking into account the future known inputs $\{n_{k+1}, n_{k+2}, \ldots, n_{k+\tau}\}$.

At each time step t during the decoding phase, the decoder updates its hidden state h_t based on the context vector c, the future known input f_t, and its previous hidden state h_{t-1}:

$$h_t = a_d(F, n_t, h_{t-1}), \quad \forall t \in [k+1, \tau], \tag{1.17}$$

It then generates the output y_t at each time step:

$$y_t = a_y(h_t), \tag{1.18}$$

where a_d and a_y are functions representative of the decoder's structure. The process continues until the entire output sequence of length τ is generated.

1.2.2 Transformer based Models

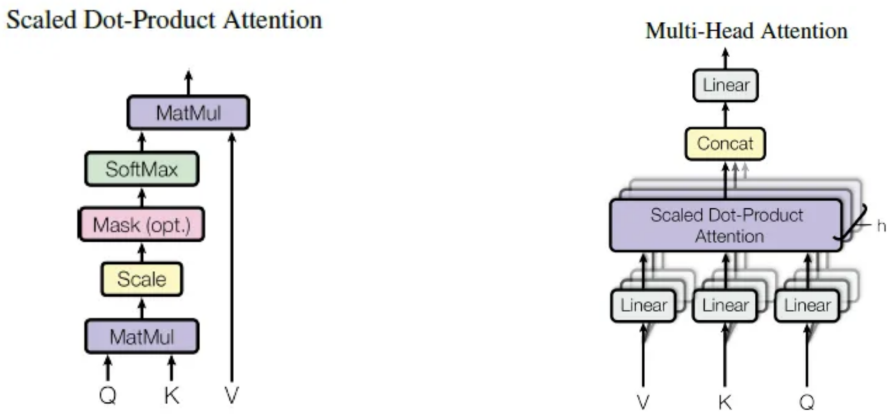
Transformer models are powerful deep learning techniques designed for processing sequential data. Since it was first brought out in [9], the original Transformer model gain a lot of popularity and has sparked many new models that go beyond just language tasks. These models are now used for tasks such as predicting the properties of a protein and predicting patterns over time. These models are known

for their self-attention mechanism which has the ability to focus on different parts of the input sequence when generating an output.

Unlike the sequential processing approach of RNNs and LSTMs, Transformers can process all parts of the sequence simultaneously, making them more efficient for tasks involving long sequences. Moreover, Transformers demonstrate better handling of long-term dependencies in comparison to RNNs and LSTMs, primarily because they are not afflicted by the vanishing gradient problem. Despite these advantages, Transformers do have some limitations, such as increased memory requirements due to the attention mechanism and a larger demand for training data.

Self-Attention Mechanism

The Transformer architecture follows an encoder-decoder structure mentioned in previous section, but does not rely on recurrence and convolutions in order to generate an output, instead it harnesses the power of self-attention mechanisms [37]. This mechanism allows the model to weigh and consider all data in the input sequence simultaneously, determining which data are important (i.e., should be attended to) in the context. As such, it enables the model to capture dependencies between tokens irrespective of their positions in the sequence, a feature that is particularly advantageous in handling long-range dependencies. An attention mechanism computes the dynamic weights, which represent the relative significance of the inputs in a sequence (known as the keys) for a specific output (the query). Multiplying the dynamic weights (the attention scores) with the input sequence (the values) will then weight the sequence. General self-attention mechanism is described in Figure 1.6a.



(a) Self-attention mechanism

(b) Multi-Head attention mechanism

Figure 1.6: Attention mechanism, Source: [9].

Unlike traditional models where only the last hidden state h_k from the encoder is passed to the decoder, the Transformer's encoder passes all of its hidden states to the decoder creating matrix $\mathbf{H} = \{h_1, \ldots, h_{k-1}, h_k\}$ and with this enriches the information available to the decoder. The decoder does an additional step before producing its output. In order to focus on the parts of the input that are relevant to this decoding time step, the decoder create three matrices: the Query matrix $\mathbf{Q}$, the Key matrix $\mathbf{K}$ and the Value matrix $\mathbf{V}$. These matrices are obtained by multiplying matrix $\mathbf{H}$ with three different weight matrices (referred to as $\mathbf{W}_q \in \mathbb{R}^{k \times d_q}, \mathbf{W}_k \in \mathbb{R}^{k \times d_k}$, and $\mathbf{W}_v \in \mathbb{R}^{k \times d_v}$, where $d_k, d_q, d_v \in \mathbb{N}$) that we learn during the training process

$$\mathbf{Q} = \mathbf{H} \cdot \mathbf{W}_q, \quad \mathbf{K} = \mathbf{H} \cdot \mathbf{W}_k, \quad \mathbf{V} = \mathbf{H} \cdot \mathbf{W}_v. \tag{1.19}$$

The Query relates to the current focus of attention, while the Key corresponds to all data points in the input, holding information that helps determine their relevance. Next, the model calculates the dot product between the Query matrix $\mathbf{Q}$ of the current data point and the Key matrix $\mathbf{K}$ of every other data point, followed by the application of a softmax function, which results in attention scores. The scores essentially represent the importance of each data point in the context of the current one. Lastly output is generated multiplying Value matrix with its corresponding attention weight (to amplify the influence of more relevant time steps and diminish that of less relevant time steps), and then summing up these weighted Value matrix

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}})V, \tag{1.20}$$

where $\frac{1}{\sqrt{d_k}}$ is scaling factor, which helps to decrease the magnitude of the dot product results,that can lead to vanishing gradients problem when passed through the softmax function.

In Transformer models, the attention mechanism repeats its calculations several times in parallel. Each of self-attention layers focuses on evaluating attention on a particular attribute and is called an attention head. The Attention module splits its Query, Key, and Value parameters number of ways and passes each split independently through a separate Head. All of these similar Attention calculations are then combined together to produce a final Attention score to computed Multi-Head attention Figure 1.6b

Multi-Head Attention

Instead of performing a single self-attention function it is beneficial to linearly project the queries, keys and values $h \in \mathbb{N}$ times with different, learned linear projections. The self-attention mechanism repeats its calculations several times in parallel. Each of self-attention layers focuses on evaluating attention on a particular attribute and is called an attention head. The Attention module splits its query, key, and value parameters and passes each split independently through a separate *Head*. All of these attention calculations are then combined together to produce a final Attention score to computed Multi-Head attention

$$MultiHead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = [Head_1, ..., Head_h)\mathbf{W}_{Head}, \tag{1.21}$$

$$Head_h = Attention(\mathbf{Q}\mathbf{W}_\mathbf{Q}(h), \mathbf{K}\mathbf{W}_\mathbf{K}(h), \mathbf{V}\mathbf{W}_\mathbf{V}(h)), \tag{1.22}$$

where $\mathbf{W}_\mathbf{Q}(h)$, $\mathbf{W}_\mathbf{K}(h)$, $\mathbf{W}_\mathbf{V}(h)$ and $\mathbf{W}_{Head}$ are learned linear projections, $h \in \mathbb{N}$ is number of attention heads. $\mathbf{W}_{Head}$ is used to transform the concatenated output of all heads into the final output representation. Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this. The use of multiple heads allows the model to capture different types of information and various relationships between the input data points.

1.3 Performance Measures for Regression Analysis

Evaluating the performance of machine learning models is a key aspect of any machine learning process [38]. This section focuses on the performance measures applicable to regression models, ensuring a robust assessment of their effectiveness. Let's say we have a data set of $N \in \mathbb{N}$ values marked $y = \{y_1, \ldots, y_N\}$, each associated with a fitted value $\hat{y} = \{\hat{y}_1, \ldots, \hat{y}_N\}$.

To quantitatively assess the performance of the models, the following metrics will be used:

1. Coefficient of Determination (R^2):
   The coefficient of determination is a statistical measurement that examines how differences in one variable can be explained by the difference in a second variable when predicting the outcome of a given event. Coficient of determination can be calculated as followes

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}, \tag{1.23}$$

   where $\overline{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$ is a mean of the observed data y. In the best case, the modeled values exactly match the observed values, which results in $R^2 = 1$. A baseline model, which always predicts $\overline{y}$, will have $R^2 = 0$. Models that have worse predictions than this baseline will have a negative R^2.

2. Mean Absolute Error (MAE):
   MAE is the average of all absolute errors. The absolute average distance between the real data y and the predicted data $\hat{y}$ calculated as

$$MAE = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i|. \tag{1.24}$$

   A lower MAE indicates better accuracy and predictive performance, as it signifies that the model's predictions are, on average, closer to the true values. MAE is often preferred when the magnitude of errors is critical and needs to be directly interpretable.

3. Mean Squared Error (MSE):
   MSE is a popular error metric for regression problems

$$MSE = \frac{1}{N} \cdot \sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{1.25}$$

   The difference between these two values is squared, which has the effect of removing the sign, resulting in a positive error value and also has the effect of inflating or magnifying large errors. That is, the larger the difference between the predicted and expected values, the larger the resulting squared positive error. This has the effect of "punishing" models more for larger errors when MSE is used as a loss function. It also has the effect of "punishing" models by inflating the average error score when used as a metric.

4. Mean Absolute Percentage Error ($MAPE$):
   Mean absolute percentage error ($MAPE$) is an accuracy measure based on percentage (or relative) errors

$$MAPE = \frac{100}{N}\sum_{i=1}^{N} \frac{|y_i - \hat{y}_i|}{|y_i|}. \tag{1.26}$$

   Mean absolute percentage error measures the average magnitude of error produced by a model, or how far off predictions are on average.

1.4 Embeddings and Feature Engineering

Machine learning algorithms are incapable of interpreting raw data, when it is non-numerical, incomplete or lacks discernible correlations. To ensure these algorithms function optimally, it is necessary to undertake a comprehensive data preparation process. It could include, but is not limited to, cleaning data, imputing missing values, transforming non-numerical data into a numerical format, and identifying and engineering relevant features. These steps increase the usability and clarity of the data, preparing it for effective use in machine learning models.

Feature selection plays a crucial role in machine learning algorithms as it ensures that the chosen features are relevant to the target variable. It's important to carefully evaluate the relevance of each feature. In some cases, it can be beneficial to create new features based on the existing ones such as ratios, products, differences, and polynomial combinations. Additionally, when working with time-series data, it can be beneficial to create features based on patterns over time such as lag or lead features, rolling statistics, and trend features. These techniques enhance the richness and predictive power of the feature set, enabling more accurate and insightful machine learning models.

Missing data can complicate the estimation of parameters and may result in biased, inefficient, and inconsistent estimations. It's a good practice to replace with a constant or summary statistics like mean, median, or mode of the feature, or using advanced techniques such as k-Nearest Neighbors [39] or regression imputation.

Data transformation

As machine learning models are typically designed to operate on numerical data, the transformation of categorical data into a numerical format is a necessary preprocessing step. Categorical data is qualitative and represents characteristics such as a color, person's gender, hometown, or the types of movies they like. The easiest way to encode categorical data is one-hot encoding. This technique converts each category value into a new column and assigns a binary value of 1 or 0. It could be represented as mapping categorical variable i-th distinct value x_i as follows

$$u_j : x_i \rightarrow \delta_{x_i\alpha}, \tag{1.27}$$

where $\delta_{x_i\alpha}$ is Kronecker delta function, where the element is only non-zero when $\alpha = x_i$. The one-hot encoding method is simple and effective for categorical variables with a limited number of unique categories, however, it can lead to a high-dimensional dataset if the categorical variable has many unique categories. This can cause certain algorithms to perform poorly due to the increased complexity of the model. Second problem, that it treats different values of categorical variables completely independent of each other and often ignores the informative relations between them.

Enother proposed method to convert categorical data to numerical data used in neural networks is entity embedding [40], that involves learning an embedding (a low-dimensional, dense vector of continuous values) for each category during the training process. It maps categorical variables in a function approximation problem, where discrete variable is mapped to a vector $\mathbf{x}_i \in \mathbb{R}^d$, where $d \in \mathbb{N}$ is desired dimension as

$$e_i : x_i \rightarrow \mathbf{x}_i. \tag{1.28}$$

This mapping is equivalent to an extra layer of linear neurons on top of the one-hot encoded input. The output of the extra layer of linear neurons given the input x_i is defined as

$$\mathbf{x}_i = \sum_\alpha w_{\alpha\beta}\delta_{x_i\alpha} = w_{x_i\beta}, \tag{1.29}$$

where $w_{\alpha\beta}$ is the weight connecting the one-hot encoding layer to the embedding layer and β is the index of the embedding layer. That way mapped embeddings are just the weights of this layer and can be learned in the same way as the parameters of other neural network layers. After we use entity embeddings to represent all categorical variables, all embedding layers and the input of all continuous variables (if any) are concatenated. The merged layer is treated like a normal input layer in neural networks and other layers can be build on top of it. In this way, the entity embedding layer learns about the intrinsic properties of each category, while the deeper layers form complex combinations of them.

Another way to improve dataset is encoding cyclical features, that repeat in a cyclical pattern, such as time of day, day of the week, month of the year, or angles in circular data. It is beneficial, because it helps capture the inherent periodicity and circular nature of certain variables in a way that is meaningful for machine learning algorithms.

Using traditional numerical encoding methods like label encoding or one-hot encoding on cyclical features may introduce incorrect linear relationships or excessive dimensions. For instance, if we encode months of the year as numbers 1 to 12, it may imply a linear order that doesn't exist (e.g., 12 is not necessarily greater than 1). One-hot encoding, on the other hand, would create 12 binary features, leading to high dimensionality.

A common method for encoding cyclical data is to transform the data into two dimensions using trigonometric functions, sine and cosine, which have a cyclic nature [41]

$$x_{sin} = \sin \frac{2\pi x}{\max(x)}, \tag{1.30}$$

$$x_{cos} = \cos \frac{2\pi x}{\max(x)}. \tag{1.31}$$

This transformation maps the data onto a unit circle in a two-dimensional space, effectively preserving the cyclical continuity of the feature as it's shown in Figure 1.7.

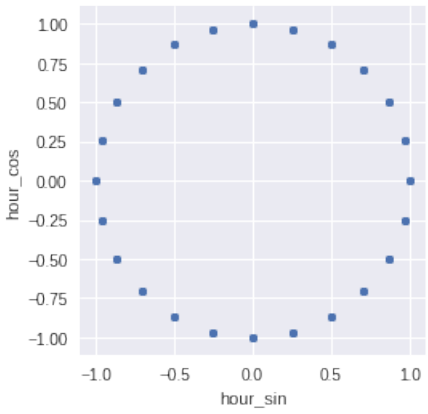


Figure 1.7: Cyclical continuity made by two encoded features.

Feature importance

Feature importance is an important part of building and interpreting accurate machine learning models, providing insight into the features that are most influential in making predictions. Understanding feature importance helps in comprehending the underlying complex relations between features and the target variable and discard the unimportant ones.

Feature importance can be established in various ways, with the appropriate method often depending on the type of model being used. Some machine learning algorithms, like decision trees and tree-based ensemble methods (such as Random Forest and Gradient Boosting), inherently provide feature importance based on the structure of the model [42], [43]. In these models, importance is typically determined by the total reduction of the criterion (like Gini impurity or entropy in classification, mean squared error in regression) brought by that feature, also known as the Gini importance.

By providing an understanding of which features are most influential in a model's predictions, feature importance enables more interpretable and reliable models. It also aids in the process of feature selection, allowing for more efficient models by reducing the dimensionality of the data.

Chapter 2

Temporal Fusion Transformer Model

Among the various statistical and machine learning models employed in time series forecasting, deep learning-based models have shown remarkable promise due to their capability to model complex, non-linear relationships and capture long-term dependencies. One of the recent and innovative contributions to this field is the Temporal Fusion Transformer (TFT) [11], a model that takes advantage of both the interpretability of traditional statistical models and the representational power of deep learning. The TFT is designed to fuse historical and future relevant context efficiently to yield more accurate multi-horizon forecasts [12].

A core feature of the TFT is its interpretability, breaking from the oft-quoted perception of deep learning models as black-box algorithms. By integrating attention mechanisms and gating, TFT provides a granular view of its decision-making process, delivering insights into the model's reliance on input features across different time horizons. This feature is particularly valuable in real-world applications where understanding the rationale behind model predictions is critical for decision-making.

The next section discusses the architecture of the TFT, describing its components and their role in the functionality of the model. The mathematical symbols in this section are taken from the original article [11] and may slightly differ from those used in the previous chapter.

TFT is capable of handling several unique entities (E) within a given time series dataset in parallel, providing increased efficiency, improved accuracy from capturing relationships between series, and a more comprehensive understanding of the data landscape through simultaneous analysis of diverse entities. TFT designed to work with canonical components to efficiently build feature representations for each input type such as static covariates $s_e \in \mathbb{R}^{d_s}$, inputs $\chi_{e,t} \in \mathbb{R}^{d_\chi}$ and scalar targets $y_{e,t} \in \mathbb{R}$ at each time-step $t \in [0, T_e]$. Time-dependent input features are subdivided into two categories $\chi_{e,t} = [z_{e,t}^T, x_{e,t}^T]^T$ – observed inputs $z_{e,t} \in \mathbb{R}^{d_z}$ which can only be measured at each step and are unknown beforehand, and known inputs $x_{e,t} \in \mathbb{R}^{d_x}$ which can be predetermined (e.g. the day-of-week at time t).
TFT provides output via prediction intervals by quantile regression (e.g. outputting the 10th, 50th and 90th percentiles at each time step), which can be useful for example when optimizing decisions and risk management. Each quantile forecast takes the form:

$$y_e(q, t, \tau) = f_q(\tau, y_{e,t-k:t}, z_{e,t-k:t}, x_{e,t-k:t+\tau}, s_e), \tag{2.1}$$

where $y_{e,t+\tau}(q, t, \tau)$ is the predicted qth sample quantile of the τ-step-ahead forecast at time t, and f_q is a prediction model. Output forecast is simultaneously generated for $\tau \in \{1, ..., \tau_{max}\}$ time steps. All past information is incorporated within a finite look-back window $k \in \mathbb{N}$, using target and known inputs only up till and including the forecast start time t ($y_{e,t-k:t} = \{y_{e,t-k}, \ldots, y_{e,t}\}$) and known inputs across the entire range ($x_{e,t-k:t+\tau} = \{x_{e,t-k}, \ldots, x_{e,t}, \ldots, x_{e,t+\tau}\}$).

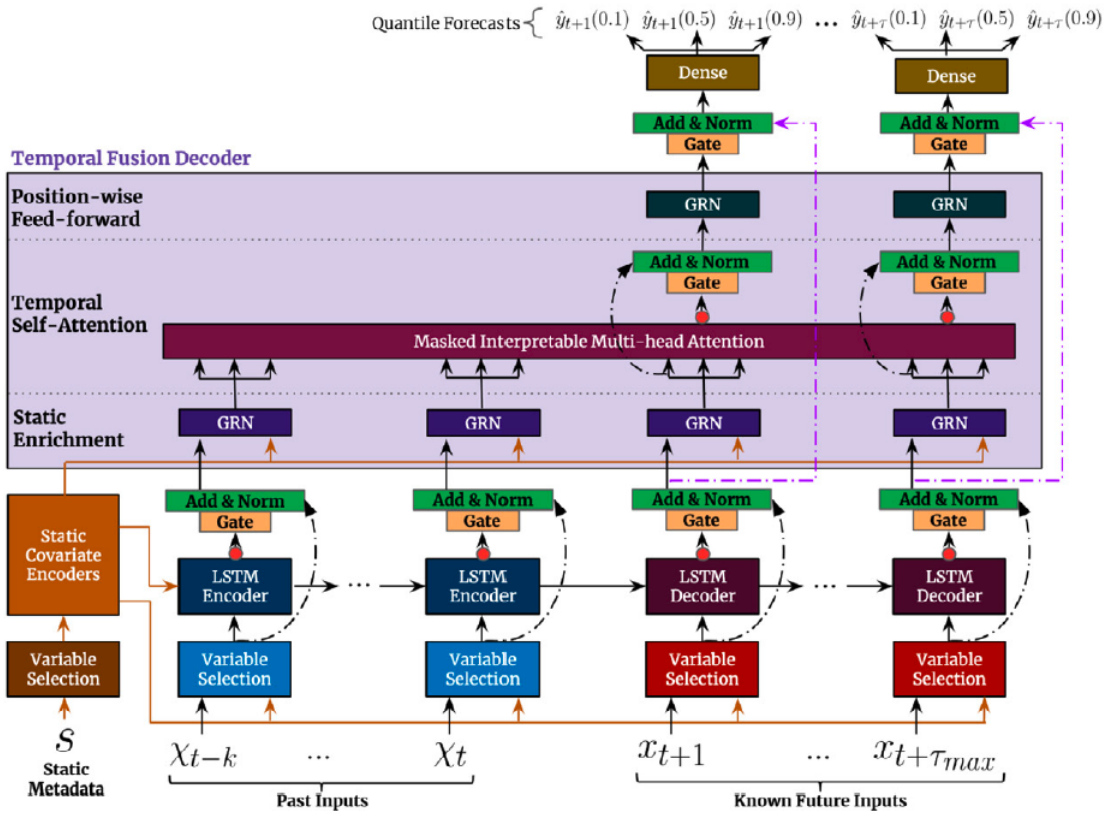


Figure 2.1: TFT Architecture, Source: [11].

Figure 2.1 shows model architecture which includes:

1. **Input Transformation** to normalise and transform data into suitable dimension.

2. **Variable Selection Network** to select relevant input variables at each time step.

3. **Temporal Processing:**

   - **Locality Enhancement with Sequence-to-Sequence Layer** build with LSTMs to handle variable-length series, capture temporal dependencies and helps enhance the locality of the model.
   - **Static Enrichment Layer** to enhance temporal features with static metadata.
   - **Interpretable Multi-Head Attention Layer** to learn long-term relationships across different time steps.

4. **Position-wise Feed-forward Layer** additional non-linear processing to the outputs of the self-attention layer.

5. **Output Layer** to predict intervals via quantile forecasts to determine the range of likely target values at each prediction horizon.

2.1 Gated Residual Network

Gated Residual Network (GRN) is a building block of TFT that skips over any unused component of the model (as learned from the data), providing adaptive depth and network complexity to accommodate a wide range of datasets and variations. It helps to estimate the performance of a specific architectural setup based on inputs such as number of layers in the network, each layer's size, the applied activation function, etc. Also it's used for variable selection and linear transformation across of the all TFT model. Architecture of GRN shown in the Figure 2.2.

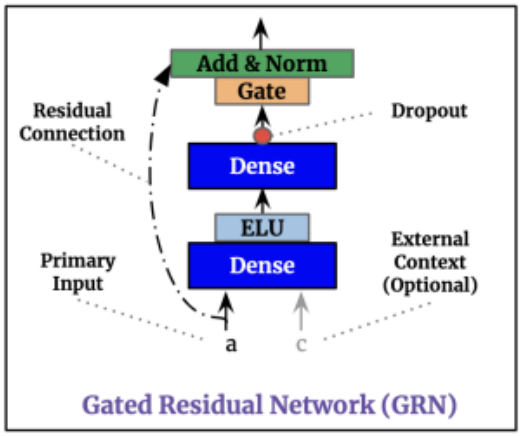


Figure 2.2: Gated Residual Network, Source: [11].

The GRN is made up of several layers of special blocks called gated residual blocks, where each block has two parts: a residual block and a gating mechanism. The residual block is responsible for transforming the input features into a higher-level representation and provide a residual connection as Eq. (1.7), which allow the network to learn identity functions, which can be useful if adding extra layers does not improve the model's performance.

The gating mechanism allows the model to selectively use or ignore certain parts of the input. Let $d_m \in \mathbb{N}$ be a hidden state size, common across TFT architecture. Then GRN takes a primary input $a \in \mathbb{R}^{d_m}$ and an optional context vector $c \in \mathbb{R}^{d_m}$ and transform it as follows:

$$GRN_\omega(a, c) = LayerNorm(a + GLU_\omega(\eta_1)), \tag{2.2}$$

$$\eta_1 = \mathbf{W}_{1,\omega}\eta_2 + b_{1,\omega}, \tag{2.3}$$

$$\eta_2 = ELU(\mathbf{W}_{2,\omega}a + \mathbf{W}_{3,\omega}c + b_{2,\omega}), \tag{2.4}$$

where ELU is the Exponential Linear Unit activation function, $\eta_1 \in \mathbb{R}^{d_m}, \eta_2 \in \mathbb{R}^{d_m}$ are intermediate layers with $\mathbf{W}_{1,\omega}, \mathbf{W}_{2,\omega}, \mathbf{W}_{3,\omega} \in \mathbb{R}^{d_m \times d_m}$ as a weight matrices and $b_{1,\omega}, b_{2,\omega} \in \mathbb{R}^{d_m}$ are biases, LayerNorm is standard layer normalization Eq.(1.7), GLU is Gated Linear Unit, and $\omega \in \mathbb{N}$ is an index to denote weight sharing.

Exponential Linear Unit [44] is defined as follows:

$$ELU(x) = \begin{cases} x, & x > 0 \\ \alpha\,(\exp(x) - 1), & x \leq 0 \end{cases}, \tag{2.5}$$

where hyperparameter $\alpha \in \mathbb{R}$ controls the value to which an ELU saturates for negative net inputs. Unlike traditional ReLU activation function, ELU allows for negative values which helps to push neuron outputs

closer to zero, easing the learning process. It also mitigates the vanishing gradient problem through its saturation property for negative inputs. When $\mathbf{W}_{2,\omega}a + \mathbf{W}_{3,\omega}c + b_{2,\omega}$ is much greater than zero, the *ELU* activation behaves like an identity function. However, when the same sum is significantly less than zero, the *ELU* activation outputs a consistent value, which leads to the layer behaving linearly.

Gating layers are based on Gated Linear Units (*GLU*s), which provide the flexibility to suppress any parts of the architecture that are not required for a given dataset:

$$GLU_{\omega}(\eta_1) = \sigma(\mathbf{W}_{4,\omega}\eta_1 + b_{4,\omega}) \odot (\mathbf{W}_{5,\omega}\eta_1 + b_{5,\omega}), \tag{2.6}$$

where σ is the sigmoid activation function, $\mathbf{W}_{4,\omega}, \mathbf{W}_{5,\omega} \in \mathbb{R}^{d_m \times d_m}$, $b_{4,\omega} \in R^{d_m}$ are the weights and biases, $\odot$ is the element-wise Hadamard product.

GLU allows TFT to control the extent to which the *GRN* contributes to the original input a – potentially skipping over the layer entirely if necessary as the *GLU* outputs could be all close to 0 in order to surpress the nonlinear contribution. For instances without a context vector, the *GRN* simply treats the contex input as zero c = 0 in Eq. (2.4). During training, dropout is applied before the gating layer and layer normalization to η_1 in Eq. (2.3).

2.2 Variable selection network

For given task multiple features could be available, but their relevance and specific contribution to the output are typically unknown. For adress this task variable selection network is a part of TFT model. It's not only providing insights into which variables are most significant for the prediction problem, but also allows TFT to remove any unnecessary noisy inputs which could negatively impact performance.

Variable selection network is applied separately on static covariates s_e and time-dependent covariates $\chi_{e,t}$. Before using variable selection network categorical inputs are transformed to entity embeddings Eq. (1.30), Eq. (1.31) as feature representations and continuous inputs are transformed via linear layer. With that each feature is represented as d_m-dimensional vector to match the dimensions in subsequent layers for skip connections. Figure 2.3 shows archicture of variable selection network.

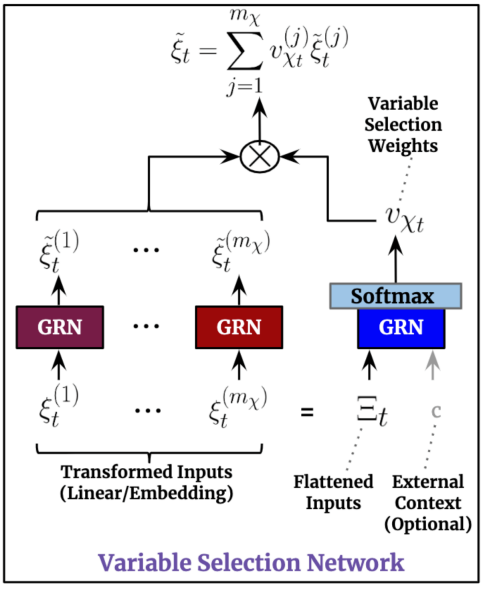


Figure 2.3: Variable Selection Network, Source: [11].

Without loss of generality, variable selection network for time-dependent covariates will be described, noting that those for other inputs take the same form. Vector $\xi_t^{(j)} \in \mathbb{R}^{d_m}$ denote transformed inputs of the j-th variable at time t, where $j \in [1, d_\chi]$. Each of processed feature vectors $\xi_t^{(j)}$ goes through it's own *GRN* generating vectors $\tilde{\xi}_t^{(j)}$

$$\tilde{\xi}_t^{(j)} = GRN_{\tilde{\xi}(j)}(\xi_t^{(j)}), \quad \forall j \in \{1, \ldots, d_\chi\}, \tag{2.7}$$

with weight sharing across all time steps t. Embedding matrix $\Xi_t = [\xi_t^{(1)}, \ldots, \xi_t^{(m_\chi)}]$ is a concatenation of variable embeddings at the time t. Variable selection weights are generated by feeding both Ξ_t and an external context vector c_s through a *GRN*, followed by a $Softmax$ layer (The softmax activation function transforms the raw outputs of the neural network into a vector of probabilities):

$$v_{\chi_t} = Softmax(GRN_{v_\chi}(\Xi_t, c_s), \tag{2.8}$$

where $v_{\chi_t} \in \mathbb{R}^{d_\chi}$ is a vector of variable selection weights, and c_s is context vector obtained from a static covariate encoder (to be explained in Section 2.3). For static variables, we note that the context vector c_s is omitted – given that it already has access to static information.

Processed features are then weighted by their variable selection weights and combined as follows

$$\tilde{\xi}_t = \sum_{j=1}^{d_\chi} v_{\chi_t}^{(j)} \tilde{\xi}_t^{(j)}, \tag{2.9}$$

where $v_{\chi_t}^{(j)}$ is the j-th element of vector v_{χ_t} and the outcome vector $\tilde{\xi}_t \in \mathbb{R}^{d_m}$.

2.3 Static Covariate Encoders

In contrast with other time series forecasting architectures, the TFT is carefully designed to integrate information from static metadata, using separate *GRN* encoders to produce four different context vectors, including context vector c_s for temporal variable selection, vectors c_c, c_h for local processing of temporal features, and vector c_e enriching of temporal features with static information.

All of the context vectors are firstly encoded from the output of VSN by going through another *GRN* layer. As an example, taking $\zeta \in \mathbb{R}^{d_m}$ to be the output of the static variable selection network, contexts for temporal variable selection would be encoded according to

$$c_s = GRN_{c_s}(\zeta). \tag{2.10}$$

These contect vectors are then wired into various locations in the temporal fusion decoder, where static variables play an important role in processing.

2.4 Temporal Processing

2.4.1 Locality Enhancement with Sequence-to-Sequence Layer and Static Enrichment

Key characteristics in time series data such as anomalies, change-points, or cyclical patterns are typically discerned by their relationship to neighboring values. Therefore, an improvement in the performance of attention-based architectures can be achieved by developing features that integrate pattern data on top of individual point values, thereby augmenting the local context.

As such, application of a encoder decoder model (Section 1.2.1) to handle these differences was proposed, feeding $\tilde{\xi}_{t-k:t}$ as well as context vectors c_s and c_h into the encoder and $\tilde{\xi}_{t+1:t+\tau_{max}}$ into the decoder constructed with LSTM layers (as it described at Section 1.2.1). This then generates a set of uniform temporal features which serve as inputs into the temporal fusion decoder itself, denoted by $\phi(t,n) \in \{\phi(t,-k),\ldots,\phi(t,\tau_{max})\}$ with $n \in [-k,\tau_{max}]$ being a position index

$$\phi(t,n) = LSTM(\tilde{\xi}_{t:n},(c_h,c_s)). \tag{2.11}$$

After that gated residual connection was implemented as follows:

$$\tilde{\phi}(t,n) = LayerNorm(\tilde{\xi}_{t+n} + GLU_{\tilde{\phi}}(\phi(t,n)), \tag{2.12}$$

where GLU is the same Eq. (2.6), LayerNorm is layer normalisation Eq. (1.7).

Static covariates can have significant influence on the temporal dynamics (for example information of different countries loads for electricity load forecast). After the encoder-decoder layer data are enhances temporal features with static metadata. Static enrichment takes the form:

$$\theta(t,n) = GRN_{\theta}(\phi(t,n),c_e), \tag{2.13}$$

where n is index, the weights of GRN_{ϕ} are shared across the entire layer, and $c_e \in \mathbb{R}^{d_m}$ is a context vector from a static covariate encoder and output vector $\theta(t,n) \in \mathbb{R}^{d_m}$.

2.4.2 Temporal Self-Attention Layer

Following static enrichment, self-attention layer is applied on matrix of grouped temporal features $\mathbf{\Theta}(t) = [\theta(t,-k),\ldots,\theta(t,\tau)]^T$ as discussed at Section 1.2.2 to learn long-term relationships across different time steps. After that TFT use special version of multi-head attention layer that was modified to enhance explainability.

Traditional multi-head attention mechanisms operate with the premise of assigning different heads to diverse representation subspaces Eq. (1.23), Eq. (1.24). In this context, each head uses distinct values, implying that attention weights alone would not be representative of a specific feature's importance. To improve this interpreted multi-head attention was proposed. It shares values in each head, and then employ sum all heads:

$$InterpretableMultiHead(\mathbf{Q},\mathbf{K},\mathbf{V}) = \tilde{H}\mathbf{W}_H, \tag{2.14}$$

$$\tilde{H} = \tilde{A}(\mathbf{Q},\mathbf{K})\mathbf{V}\mathbf{W}_{\mathbf{V}} = \frac{1}{h}\sum_{i=1}^{h} A(\mathbf{Q}\mathbf{W}_{\mathbf{Q}}{}^{(i)},\mathbf{K}\mathbf{W}_{\mathbf{K}}{}^{(i)}) = \frac{1}{h}\sum_{i=1}^{h} Attention(\mathbf{Q}\mathbf{W}_{\mathbf{Q}}{}^{(i)},\mathbf{K}\mathbf{W}_{\mathbf{K}}{}^{(i)},\mathbf{V}\mathbf{W}_{\mathbf{V}}), \tag{2.15}$$

where $h \in \mathbb{N}$ is number of heads, $\mathbf{W}_{\mathbf{Q}}$ is a query weight matrix, $\mathbf{W}_{\mathbf{K}}$ is a key weight matrix and $\mathbf{W}_{\mathbf{V}}$ is a value weight matrix. Here, $\mathbf{W}_{\mathbf{V}}$ are value weights that are shared across all heads, and $\mathbf{W}_{\mathbf{H}}$ is used for the final linear mapping. This structure enables each head to learn different temporal patterns, while attending to a common set of input features, thereby achieving a form of ensemble over attention weights and consolidating them into the combined matrix $\tilde{A}(\mathbf{Q},\mathbf{K})$. Compared to $A(\mathbf{Q},\mathbf{K}),\tilde{A}(\mathbf{Q},\mathbf{K})$ results in a more substantial representational capacity in a computationally efficient manner.

In the context of the TFT algorithm, the InterpretableMultiHead function is utilized at each forecast time (with $N = \tau_{max} + k + 1$) as follows

$$B(t) = InterpretableMultiHead(\mathbf{\Theta}(t),\mathbf{\Theta}(t),\mathbf{\Theta}(t)), \tag{2.16}$$

to get $\mathbf{B}(t) = [\beta(t,-k),...,\beta(t,\tau_{max})] \in \mathbb{R}^{\tau_{max}-k\times d_m}$. Decoder masking [17, 12] is applied to the multi-head attention layer to ensure that each temporal dimension can only attend to features preceding it.

The attention layer grants TFT the capacity to detect and learn long-range dependencies that might pose challenges for RNN-based architectures. Subsequent to the attention layer, TFT implements an gating layer to optimize the learning process even more:

$$\delta(t,n) = LayerNorm(\theta(t,n) + GLU_{\delta}(\beta(t,n))). \tag{2.17}$$

This additional gating layer utilizes the Layer Normalization and *GLU* functions on the temporal features $\theta(t,n)$ and $\beta(t,n)$. Layer Normalization is employed to stabilize the learning process by reducing internal covariate shift, while *GLU* provides a gating mechanism that introduces non-linearity to the model, thereby improving its expressive capability. This synergistic operation of normalization and gating enhances the model's ability to capture complex feature interactions over time.

2.4.3 Position-wise Feed-forward Layer

An additional step of non-linear processing is applied to the outputs from the self-attention layer, leveraging Gated Residual Networks (*GRN*s):

$$\psi(t,n) = GRN_{\psi}(\delta(t,n)), \tag{2.18}$$

Here, the weights of GRN_{ψ} are evenly shared throughout the layer. As illustrated in Figure 2.1, a gated residual connection is also established, bypassing the entire transformer block, hence providing a direct route to the encoder-decoder layer. This results in a more streamlined model when additional complexity is deemed unnecessary, as indicated below:

$$\tilde{\psi}(t,n) = LayerNorm(\tilde{\phi}(t,n) + GLU_{\tilde{\psi}}(\psi(t,n)), \tag{2.19}$$

Here, $\tilde{\psi}(t,n)$ utilizes a combination of Layer Normalization and a *GLU* operation on $\tilde{\phi}(t,n)$ and $\psi(t,n)$, further enhancing the ability of the model to capture intricate patterns in the temporal data.

2.5 Quantile Outputs

TFT also designed to generate prediction intervals on top of point forecasts, which can help to better understand the scope of potential outcomes and their associated risks. For example, over-predicting and under-predicting electricity demand both have different costs associated with them. Over-prediction might lead to unnecessary production and subsequent waste of electricity. On the other hand, under-prediction could result in blackouts or the need to purchase additional power at high costs. Quantile regression can help to optimize predictions taking into consideration these asymmetric costs. This is achieved by the simultaneous prediction of various percentiles (e.g. 10th, 50th and 90th) at each time step. Quantile forecasts are generated using linear transformation of the output from the temporal fusion decoder:

$$\tilde{y}(q,t,\tau) = w_q\tilde{\psi}(t,\tau) + b_q, \tag{2.20}$$

where $w_q \in \mathbb{R}^{d_m}, b_q \in \mathbb{R}$ are linear coefficients for the specified quantile q.

2.6 Loss Functions

TFT utilizes a training method that minimizes the quantile loss over all quantile outputs concurrently. The loss function is given by:

$$L(\Omega, W) = \sum_{y_t \in \Omega} \sum_{q \in Q} \sum_{\tau=1}^{\tau_{max}} \frac{QL(y_t, \hat{y}(q, t-\tau, \tau), q)}{M\tau_{max}}. \tag{2.21}$$

The quantile loss function is:

$$QL(y, \hat{y}, q\hat{}) = q(y - \hat{y}) + +(1-q)(\hat{y} - y)_+. \tag{2.22}$$

Here, Ω denotes the set of training data instances with $M \in \mathbb{N}$ samples. Q represents the set of quantiles output (0.1,.., 0.9), and the expression $(.)_+$ is equivalent to maximization.

For out-of-sample evaluation, normalized quantile losses across the entire forecasting horizon are considered. Specifically, the P50 and P90 risks are prioritized to maintain coherence with prior research. This is represented as:

$$q - Risk = \frac{2 \sum_{y_t \in \tilde{\Omega}} \sum_{\tau=1}^{\tau_{max}} QL(y_t, \hat{y}(q, t-\tau, \tau), q)}{\sum_{y_t \in \tilde{\Omega}} \sum_{\tau=1}^{\tau_{max}} |y_t|}. \tag{2.23}$$

In this context, $\tilde{\Omega}$ refers to the domain of the test samples. This training and testing approach facilitates a robust model capable of generating accurate quantile forecasts, thereby effectively capturing the uncertainty in the data and enhancing the model's overall predictive capability.

2.7 Hyperparameters

In the TFT model, hyperparameters play a crucial role in determining the model's performance. Hyperparameters are parameters that are not learned from the data during the training process, but are instead set before the training process begins. They control aspects of the model's architecture and the training process, such as the number of layers in the model, the learning rate, and the batch size.

The first set of hyperparameters in the TFT model relates to its architecture. This includes the number of time steps the model looks back in the data (sequence length k), the number of attention heads h and hidden units in each layer d_l, and the dropout rate. These parameters control the complexity of the model, with larger values typically resulting in a more complex model that can capture more intricate patterns in the data, but that is also more prone to overfitting.

The second set of hyperparameters relates to the training process. This includes the learning rate η, which controls the step size taken during gradient descent, and the batch size, which determines the number of examples used to compute each update to the model parameters and number of LSTM layers. These parameters can significantly affect the speed and stability of the training process, as well as the final performance of the model.

Hyperparameter tuning is the process of finding the optimal set of hyperparameters that yield the best performance on a validation dataset. This is typically done through a process of trial and error, systematically adjusting the hyperparameters and observing the impact on the model's performance. Common strategies for hyperparameter tuning include grid search, where all possible combinations of hyperparameters are evaluated, and random search, where hyperparameters are sampled randomly from a predefined range. More advanced methods, such as Bayesian optimization, use a probabilistic model to guide the search process and can often find good hyperparameters more efficiently.

Chapter 3

Implementation and Results

Electricity demand prediction serves a critical role in managing power systems and markets. Unlike other commodities, electricity must be generated as it's consumed, with only a limited amount capable of being stored. Therefore, accurate prediction of electricity demand is fundamental in ensuring system reliability, promoting economic efficiency, and fostering environmental sustainability.

System reliability remains a common priority in managing power grids. Precise demand forecasts facilitate necessary adjustments in energy generation and consumption, maintaining equilibrium across the grid. This precision allows operators to match supply effectively, thereby ensuring a stable grid frequency vital for uninterrupted service.

Furthermore, accurate demand prediction carries significant ecological implications. When demand is underestimated, power plants often resort to the rapid generation of electricity using fossil fuels, which contributes to environmental degradation. Accurate forecasting allows for better incorporation and optimization of renewable energy sources into the grid mix. With renewables being dependent on weather conditions and inherently variable, accurate demand predictions can help maximize their use when conditions are favorable. This not only reduces reliance on carbon-intensive power plants but also minimizes unnecessary carbon emissions, crucial for climate change mitigation efforts.

In deregulated electricity markets, electricity is traded much like any other commodity. The price of electricity fluctuates based on supply and demand, creating opportunities for traders to profit from these price movements. The ability to accurately forecast electricity demand is an invaluable tool in this context. In the electricity market, the impact of accurate demand forecasting is equally profound. One of the most direct effects is on price stability. By ensuring that supply and demand remain in balance, accurate forecasting can help to prevent price volatility. Unexpected surges or drops in demand can cause prices to fluctuate wildly, potentially disrupting the normal functioning of the market. By providing a clear view of future demand, accurate forecasts help market participants to plan ahead, smoothing out potential price spikes and troughs.

For energy traders, meanwhile, accurate demand forecasts can open up profitable opportunities. By predicting when demand will rise or fall, traders can make informed decisions about when to buy and sell power contracts. This foresight can allow them to secure contracts at low prices ahead of anticipated demand surges and sell when prices are high, thereby capturing profit. Accurate forecasting also provides a basis for risk management, helping traders to hedge against price volatility and protect their positions in the market.

There are many types of wholesale electricity markets, such as Forward and Futures Market, Day-Ahead Market, Intra-day Market or Balancing Market. This study will be focused on two of them:

- Day-Ahead Market: In this market, electricity quantities are traded one day ahead of the actual delivery. Participants submit their bids for each hour of the next day starting at 12 pm.
- Intra-day Market: In this market, trading occurs in real-time or close to the actual delivery of electricity. Prices in this market are typically more volatile, and any inaccurate previous predictions of demand can be corrected here.

In this section, the focus will be on the application and comparison of Random Forest, XGBoost, and Temporal Fusion Transformer - in forecasting electricity demand in the context of both the day-ahead and intraday markets, specifically using data from the Czech Republic. Implementation is published on Github. [1]

Initially, the process of data collection and preprocessing will be detailed. This will include an explanation of the sources of data, the type of data collected, and the steps taken to prepare this data for use in the models. This stage is crucial to ensure the accuracy and reliability of the subsequent analysis.

Following this, each predictive model - Random Forest, XGBoost, and TFT - will be elaborately discussed in the context of their application to each market case. This will encompass the specifics of their implementation, the unique aspects of their operation in the different market contexts, and an evaluation of their performance. The aim is to provide a clear understanding of the strengths and weaknesses of each model and how they perform in the real-world scenarios of the day-ahead and intraday electricity markets. The generalised algorithm is shown in the Figure 3.1.

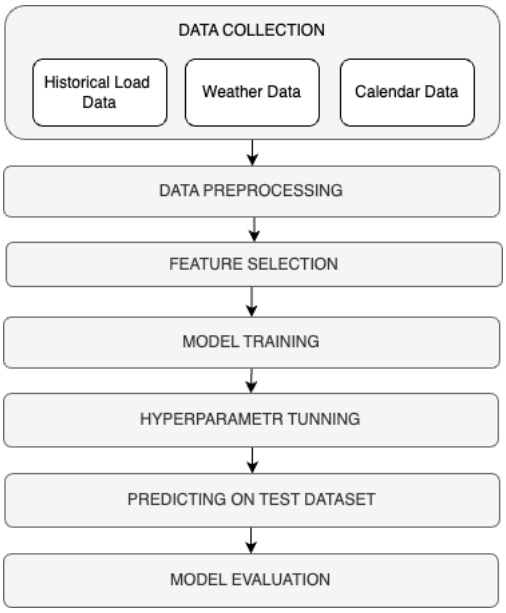


Figure 3.1: Generalised algorithm of study.

In this study, three distinct models were utilized in Python [45], each drawn from a different library. The Random Forest model was implemented using the RandomForestRegressor from the sklearn.ensemble library [46]. The TFT was sourced from the pytorch-forecasting library [48], and the XGBoost model was implemented using the xgboost library [47].

[1] `https://github.com/podleyan/master_thesis_yp`

3.1 Data collection and preprocessing

Obtaining relevant data is the essential first step in achieving accurate prediction. For this study several types of data was used including historical electricity load, meteorological and calendar data. For this purpose three functions were implemented - `entsoe_data()`, `calendar_data()` and `weather()`. These functions return hourly data on historical electricity load, calendar information, and weather conditions for a specific country and date range. Function `calendar_data()` will return day, month, year, week, weekday and information about public holidays (1 for holiday, 0 for no holiday). The public holidays data was retrieved using the 'holidays' Python library. This open-source library allowed us to obtain public holidays for numerous countries. As part of preprocessing the calendar data, cyclical calendar variables was transformed to represent the data sequentially as it was described in Section 1.4. For example, December and January are 1 month apart, although those months will appear to be separated by 11 months. To avoid this, 2 new features were created, deriving a sine and a cosine transform of the original feature for hours and month features.

The `weather_data()` function returns hourly information about temperature, precipitation, dew point, wind speed, and other weather data within a date range across ten geographical points in a specific country. This data is retrieved using the 'meteostat' Python library.

Weather forecast was download from Open-Meteo API.

Lastly, function `entsoe_data()` returns hourly information about electricity load for specific country with in data range using ENTSO-E Transparency Platform's API. The ENTSO-E Transparency Platform is an invaluable resource for detailed and up-to-date data about Europe's electricity market. For using API a key was generated and obtained from personal account in ENTSO-E Transparency Platform website.

After all the neccessary data was obtained, the next step was to ensure its usability for subsequent analysis. This stage involved the identification and handling of missing values, duplicates, and potential outliers. Missing values was replaced with mean of the surrounding data, duplicate entries in the data, if any, were also identified and removed. After this three dataframes were merged into one and was stored locally for futher preprocessing and analysis. This was done to minimize the need for repeated API calls, which can be time-consuming and may be subject to rate limitations.

Past events can significantly influence future occurrences, but models such as Random Forests or XGBoost are not natively capable of handling time-series data or capturing temporal dependencies. Because of that lagged features was created. By creating new features that represent past values (lags) from the time-series, we can provide this types of model with a way to consider not only current situation, but also how things have changed over time. Specifically, lagged features for load data was created at intervals of 1, 24, 48, and 168 hours prior. The aim was to catch the load pattern at the same hour one day, two days, and one week before the prediction point, respectively. In TFT model, using lagged features could be also helpful. They allow us to shorten the length of the encoder without loss of quality, which makes the model run faster. This way, important information from past data remains kept without slowing down the model's training or prediction speed. For TFT using for example 24, 48 and 168 hours lags was considered.

Another helpful features could be 24-hour laged and lead values was for holidays. These inputs can provide valuable temporal context and capture the cyclic nature of daily patterns and the potential impact of holidays on the time series.

In addition to the electricity load data and static covariates, exogenous variables such as temperature forecasts were intended to be incorporated into the models. These variables can provide valuable context, potentially improving the model's ability to capture external factors that influence the electricity load.

However, due to challenges in obtaining lagged temperature forecasts for the necessary time periods and locations, temperature forecast hour ahead observations were used as a stand-in for the ideal forecast. This approach assumes almost perfect foresight regarding temperature, which is, of course, not possible in a real-world forecasting situation.

While this approach allows for the exploration of the TFT model's ability to incorporate exogenous variables, it is important to acknowledge that the use of actual temperature data likely inflates the model's performance. Future work will aim to incorporate real temperature forecasts once appropriate data is available.

In this study, data was collected for the date range from January 2021 to June 2023 for three countries: the Czech Republic, Slovakia, and Hungary. For each case, the data was divided into training and test datasets with 15 March 2023 splitting date using the `split_in_time` function. Training dataset will be used to train predictive model to learn patterns, relationships between features and target. After the model has been trained on the training data, it's important to evaluate its performance on unseen data to ensure that it can generalize well to new, unseen situations. For this purpose test data will be used. The model's performance on the test data gives us a good indication of how it would perform in a real-world scenario.

While it's possible to obtain a big number of features, not all might be beneficial for the prediction task. To reduce the number of features, the next step involved identifying the most informative features for task. In this study following methods was used:

Correlation Analysis: Correlation analysis was conducted using function corr_matrix provided by pandas library [49] to measure the linear relationship between each feature and the target variable. Features with strong correlations were deemed potentially useful, given their strong relationship with the electricity load.

Random Forest Feature Importance: To go deeper, feature importance method implemented in the RandomForestRegressor from the sklearn.ensemble library [46] was used. It provides a ranking of feature importance, enabling us to identify the features that contribute the most to the model's predictions by calculating the decrease in the model score when the feature's values are randomly shuffled.

Variable Selection Network in Temporal Fusion Transformer: Lastly, when using TFT, it comes with built-in Variable Selection Network described at Section 2.2. By examining weights generated by VSN, we can understand which features the model finds the most valuable for its predictions, adding another layer of interpretability.

By using these methods, feature set was pruned and only the most informative and relevant features for electricity load forecasting was retained. For this example the most relevant features are historical load and it lags. Weather data was reduced to temperature, dew point and total sunshine duration features for 10 distinct points. From the calendar data, the month, hour, day, weekday, and holiday indicators (including of their lagged and lead values) were identified as the most relevant for this specific scenario.

TFT Data Processing

The initial step in implementing the TFT within the PyTorch Forecasting library revolves around the appropriate preparation of the dataset. For successful implementation, the library requires a pandas DataFrame comprising, at a minimum, a time index column, a target column (indicating the variable to be forecasted), and a group_ids column (used to differentiate between various time series). In this study, the target column corresponds to the electricity load, and the group ids denote different countries. One of

the key features of the TFT is its ability to handle multiple time series concurrently during training. This feature is exploited in this study by incorporating electricity load data from both Slovakia and Hungary, allowing the model to capture potential cross-country dependencies and influences on the electricity load.

This is achieved using the TFT's static covariate encoder described at Section 2.3, which processes static (non-time-varying) inputs that are shared across all time steps. In this case, the static covariate would be the country identifier (Slovakia or Hungary). The static covariate encoder generates a fixed-length context vector that represents the static input, and this context vector is then used in various parts of the model, influencing the processing of the dynamic (time-varying) inputs and hence the final prediction.

A prominent feature of the TFT is its compatibility with distinct types of time-varying variables, which are designated by the arguments: `time_varying_known_categoricals`, `time_varying_known_reals`, `time_varying_unknown_categoricals`, and `time_varying_unknown_reals`. Within the context of time series forecasting, "known" variables denote those that are known for future time periods, while "unknown" variables correspond to those that are not predictable. The terms 'categorical' and 'real' delineate the data types of these variables.

For this study, the `time_varying_known_reals` variables encompass timestamp, hour, month, weekday, and holiday. These represent various time-related features and a binary feature that indicates whether a particular day is a public holiday. The chosen `time_varying_unknown_reals` variable is the temperature and dew point in 10 different points, which is unknown since real-time future temperatures cannot be predetermined and are, therefore, part of the forecasting exercise.

The variable `max_encoder_length` stipulates the number of time steps in the past to be input into the model, and the `max_prediction_length` variable determines the maximum number of steps to forecast into the future.

The Python code provided subsequently illustrates the instantiation of the `TimeSeriesDataSet` class, using the prepared data and the defined parameters.

```
training = TimeSeriesDataSet(
    X_train[lambda x: x.index <= training_cutoff],
    target="Actual Load",
    time_idx= 'time_idx',
    group_ids = ["country"],
    min_encoder_length=max_encoder_length,
    max_encoder_length=max_encoder_length,
    min_prediction_length=1,
    max_prediction_length=max_prediction_length,
    time_varying_known_categoricals=[],
    time_varying_known_reals=["timestamp","hour_sin", "hour_cos", "month_sin",
    "month_cos", "weekday_binary","holiday","holiday_lag", "holiday_lead",
    "load_lag_24", "load_lag_48", "load_lag_168", "fct_temp"],
    time_varying_unknown_categoricals=[],
    time_varying_unknown_reals=[
        "01_temp", "02_temp", "03_temp","04_temp","05_temp",
        "06_temp","07_temp","08_temp","09_temp","10_temp",
        "08_tsun",
        "01_dwpt", "02_dwpt", "03_dwpt","04_dwpt","05_dwpt",
        "06_dwpt","07_dwpt","08_dwpt","09_dwpt","10_dwpt"
 ],
)
```

3.2 Day-Ahead Market

For the Day-Ahead Market, it is essential to have a 24-hour electricity load prediction by 12 pm for the following day. This forecast allows market participants to better plan their bids for each hour of the next day, hence helping to ensure efficient market operation and maximize economic returns. The goal in this study is to predict electricity loads from 12 to 36 hours ahead.

XGBoost and Random Forest generated predictions recursively, using their own previous predictions as inputs for subsequent predictions implemented by function `reccursive_prediction()`. In this approach, the models make a one-step ahead forecast, and then feed this prediction back into the model as an input for making the next step prediction. This process is repeated until the required prediction horizon is reached.

For instance, to make a 36-hour ahead forecast, the models would first predict the load for the next hour, then use this prediction as part of the input data to predict the load for the hour after that, and so on, until the 36-hour forecast is completed. This recursive strategy allows the models to capture dynamic changes in the electricity load, as each prediction is influenced not only by the historical data, but also by the very recent predictions made by the model.

This strategy, however, also means that any error in the model's predictions can be propagated forward, potentially affecting the accuracy of the subsequent forecasts. As such, while XGBoost and RF are powerful models, their performance in this task will heavily depend on their ability to make accurate short-term forecasts. The XGBoost and RF models were trained using lags of 1, 24, 48, and 168 hours, where unknown the 1-hour and 24-hour lags were derived from the models' own forecasts as well.

In the application of TFT, a prediction length of 36 hours and an encoder length of 25 hours were set. As time unknows values temperature and dew point of 10 different points was used. 48-hour and 168-hour lags as well as weather forecast and calendar data were incorporated into the model as a future known variable.

Variable Selection

Initially, a feature importance method was used to preselect the features. A Random Forest model was trained using all these preselected attributes, but when the same set of features was used in an XGBoost model, it underperformed. The XGBoost model performed optimally when it was streamlined to include only a single weather feature - the most relevant temperature parameter. This is due to the greedy nature of the algorithm, which means it sometimes selects features that provide the best immediate split or gain, rather than the ones that may be most beneficial in the long run. When the number of features is large, there is an increased likelihood that XGBoost may choose features that are less important or even irrelevant, leading to suboptimal trees and, consequently, reduced model performance.

When training TFT - it has advantage with prebuild VSN, automatically selecting and using the most important features for prediction. This network ranks the importance of the input variables for the encoder and decoder separately, allowing the model to focus on different variables at different stages of the forecasting process. This ranking is learned during training and is represented by weights in the VSN, matching the weights from Eq (2.8).

The results from the variable selection network, presented in Figure 3.2, highlight the most imporatant variables for the model. For the encoder, which processes the past and present inputs, the weekday, the 168-hour lag, and the temperature are ranked as the most important. The significance of these variables suggests that the electricity load exhibits weekly patterns (captured by the weekday and 168-hour lag) and is influenced by the temperature, which is consistent with known behavior of electricity load data.

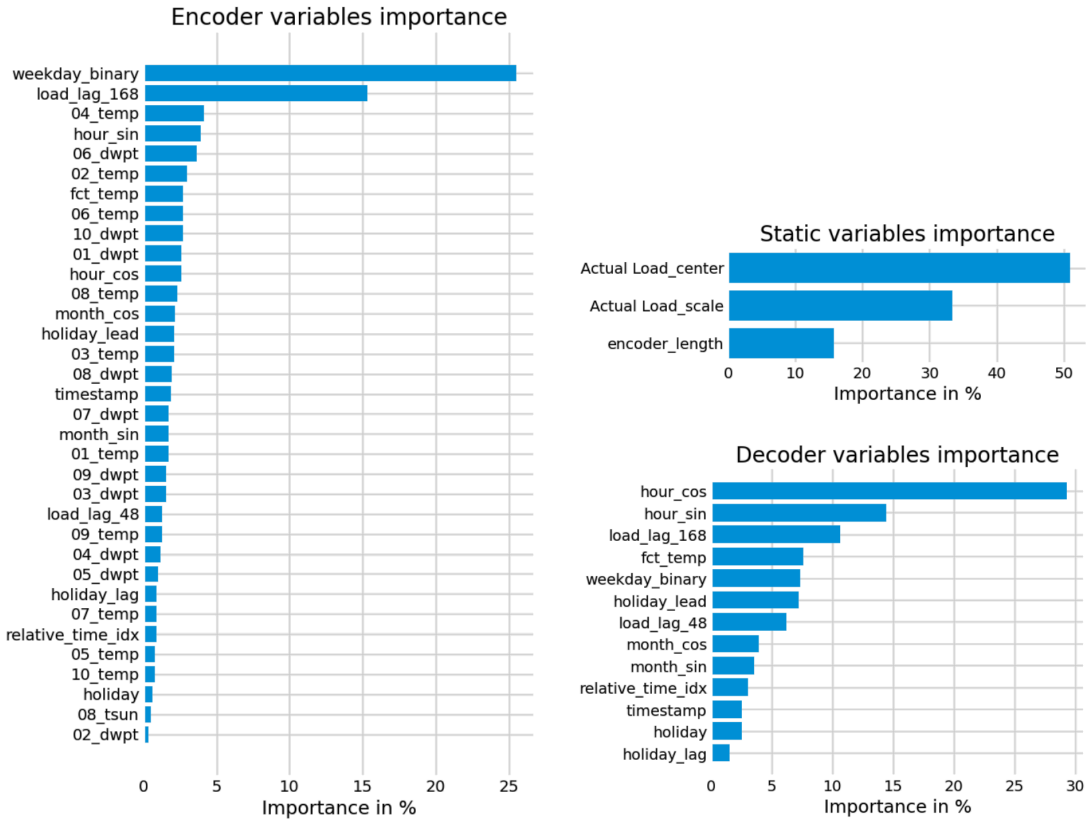


Figure 3.2: TFT VSN for 24-hour ahead prediction.

For the decoder, which makes the actual forecasts, the hour (represented as sine and cosine to capture the daily cycle), the logarithm of the 168-hour lag, and the temperature forecast are found to be the most crucial. The importance of these variables indicates that the daily pattern (captured by the hour) and the weekly pattern (captured by the logarithm of the 168-hour lag) are key factors in making accurate short-term forecasts. The significance of the temperature forecast further confirms the impact of temperature on the electricity load.

Hyperparameters Tunning

In this study, automated hyperparameter optimization process was employed to find the optimal set of hyperparameters that generates the best performance on a validation dataset.

For the XGBoost and Random Forest models was used a strategy that combines Cross Validation with Time Series Split and Random Search from python library sklearn.model_selection [50] to find the best hyperparameters. Traditional Cross Validation techniques such as K-Fold can lead to data leakage when dealing with time series data, as these methods assume that all data points are independently and identically distributed, which is not the case in time series data. Therefore, Time Series Split Cross Validation was used, which respects the temporal order of observations and prevents training the model on data from the future to predict the past. Random Search is a hyperparameter optimization technique that involves training the model on a random selection of hyperparameters from specified ranges. For each set of hyperparameters, models were fitted on the training folds and evaluated them on the validation fold, repeating this process for a large number of randomly selected sets of hyperparameters. Data was split up into three splits, with intervals showed in Figure 3.3.

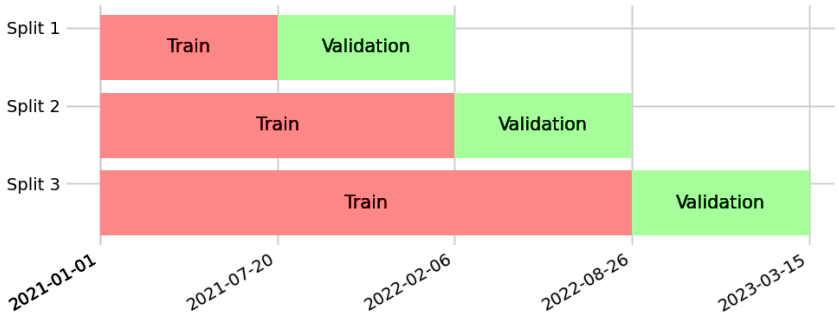


Figure 3.3: Time Series Cross Validation Split.

The best hyperparameters were the ones that yielded the best average performance on the validation folds measured with MSE. For XGBoost maximum depth of the tree, number of the estimators, learning rate, L1 and L2 regularization hyperparameters were tunned. For Random Forest maximum depth of the tree, number of features to consider when looking for the best split and number of the estimators were tuned. By using this strategy, optimal hyperparameters for XGBoost and Random Forest models were selected in a time-conscious manner that respects the temporal structure of our data, providing us with models that perform optimally on unseen, future data.

For optimizing TFT function optimize_hyperparameters in Optuna library [51] was used. The hyperparameters that was optimized included the learning rate, the size of the hidden layers, the number of attention heads, the dropout rate, the size of the hidden layers for continuous input variables and the number of LSTM layers. Optuna employs a tree-structured Parzen estimator (TPE) algorithm, a type of Bayesian optimization, to propose new sets of hyperparameters based on the results of previous trials. The resulting optimal hyperparameters as well as search ranges are presented in Table 3.1.

Hyperparameter	**Optimal Value**	**Minimum**	**Maximum**
Learning rate	0.001	0.001	0.1
Hidden size	60	8	128
Attention head size	2	1	4
Dropout rate	0.7	0.1	0.8
Continuous hidden size	30	8	128
Number of LSTM layers	2	1	4
Loss function	Quantile Loss		

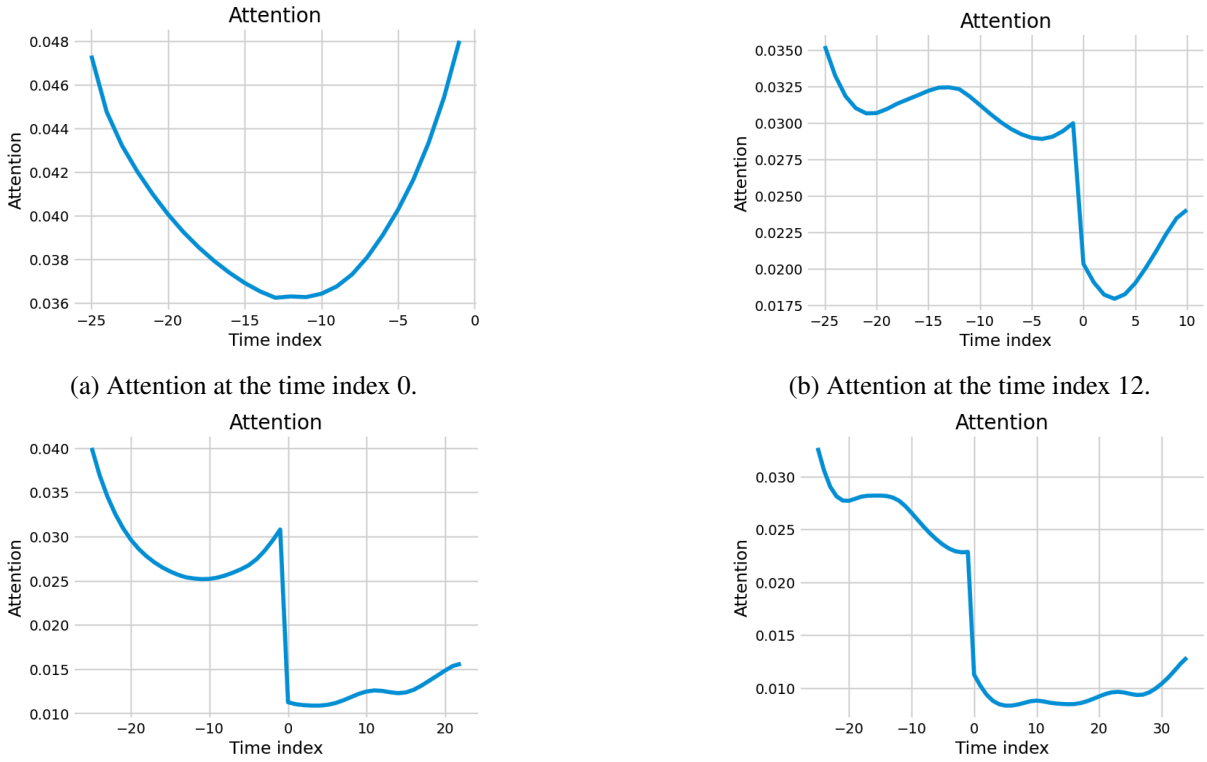
Table 3.1: Selected hyperparameters for the TFT model.

The TFT network comprises approximately 586.5k parameters. These hyperparameters suggest a careful balance between model complexity and overfitting, allowing the TFT model to capture intricate patterns in the data while also preventing overfitting. It's worth noting that these values represent the optimal configuration for our specific task and dataset, and might not generalize to other tasks or datasets. Therefore, we recommend that practitioners always conduct a similar hyperparameter optimization process when applying the TFT model to their own tasks.

Attention

The TFT model employs an attention mechanism described at Section 2.4.2, allowing the model to focus on different inputs at each time step. Figure 3.4 shows attention scores for time indexes 0, 12, 24,

36.



(a) Attention at the time index 0.

(b) Attention at the time index 12.

(c) Attention at the time index 24.

(d) Attention at the time index 36.

Figure 3.4: Attention scores for 36 hour ahead forecasting horizon.

For the immediate forecast horizon (0 hours), the model shows an increased focus on the most recent data points. This is consistent with initial assumptions, as the immediate past often contains the most relevant information for predicting the near future. In addition, the model shows increased attention to historical data relating to the same hour as the predicted hour. This suggests that the TFT model is effective in identifying and applying temporal patterns and cycles specific to a particular time of day. As the prediction horizon extended to 12, 24 or 36 hours, the attention map continues to show a similar pattern. The model retains a significant degree of attention to recent historical data points, as well as to data points occurring at the corresponding time the day before. At these forecast horizons, the model also has access to future known variables and pays some attention to nearest to this data points.

Models Evaluation

The model was evaluated on test dataset using four performance measures: MAPE, MSE, MAE and R^2 score. The results of the evaluation are presented in the Table 3.2. External Forecasts from the ENTSOE platform were also included for comparison.

The XGBoost model, which has the lowest MAPE, showed the most accurate results, indicating that its forecasts deviate from real values by only 2.38% on average. This is the smallest prediction error among all models, making it the most accurate. In addition, the model has the lowest MSE of 46,242.78, indicating that it has the smallest magnitude of squared difference error among all the models. This means that the predictions of the model were found to be quite close to the actual observations. In

Model	MAPE	MSE	MAE	R^2
TFT	3.1	96 239.34	214.39	0.91
XGBoost	2.38	46 242.78	170.18	0.95
Random Forest	2.39	58 828.69	106.58	0.94
Entsoe Forecast	3.57	74 961.43	109.02	0.93

Table 3.2: Performance metrics for the models.

addition, the XGBoost model has the second highest MAE and the highest R^2 value, which means that it was able to explain 95% of the variance of the data and was the best model on this dataset.

The Random Forest model also performed well, similar to the XGBoost model. It has the second highest MAPE and the lowest MAE, indicating the lowest average prediction error in absolute terms. It also has a high R^2 score 94%, indicating that it captures most of the variation in the data.

The TFT model, on the other hand, performed the worst in this example due to long prediction horizon. It has the highest MAPE and MAE scores, indicating that its predictions deviate more strongly from actual values in both relative and absolute terms. It also has the highest MSE, indicating the largest squared difference between forecasts and actual values. The 36-hour predictions made by all three models are displayed in Figure 3.10.

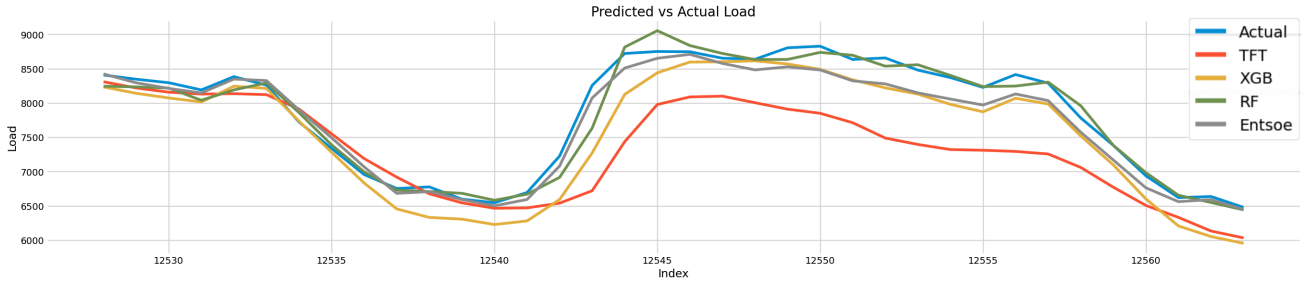


Figure 3.5: 36-hour predictions by TFT, XGBoost, and Random Forest.

This visualisation shows that the TFT model provides high forecast accuracy in the initial 15-hour time interval. However, its predictive performance tends to decrease for periods beyond this range. Since the TFT model relies primarily on time dependencies, the increasing complexity and unpredictability of these dependencies as the forecast horizon increases tends to reduce its performance. This effect is getting worse as the forecast horizon increases due to the increased uncertainty inherent in future forecasts.

3.3 Intra-Day Market

For intra-day electricity load forecasting, accurate prediction of electricity load for several hours ahead is of paramount importance. Power system operators and market participants need to rapidly react to load changes or unexpected events within the day. Having an accurate forecast enables these stakeholders to make well-informed decisions about dispatching resources, optimizing operational strategies, and managing potential risks. In this example, goal is to generate accurate forecasts of electricity loads 6 hours ahead.

In this case Random Forest and XGBoost models were implemented in direct way by creating different models for every hour separately. Since each forecast horizon has its own dedicated model, you can tailor the features and even the model architecture to best suit the specific forecasting task. For instance, different lagged variables or other relevant features can be included for each forecast horizon. However, the direct approach also comes with some challenges. One of the main drawbacks is the need to manage multiple models, which can increase computational complexity and require additional time for training and tuning. Additionally, while the direct approach can effectively capture the unique characteristics of each forecast horizon, it may fail to exploit potential dependencies between different horizons.

In the application of TFT, a prediction length of 6 hours and an encoder length of 25 hours were set. As time unknows values temperature and dew point of 10 different points was used. 24-hour, 48-hour and 168-hour lags as well as weather forecast and calendar data were incorporated into the model as a future known variable.

Variable Selection

For Random Forest and XGBoost a distinct set of features was developed for each model, incorporating the most recent lagged feature of the electricity load, historical temperature, dew point and the corresponding temperature forecast were used, both lagged and real-time. This method allowed for a tailored, hour-specific modeling strategy, where each model could effectively learn and adapt to the unique temporal dynamics of its respective forecast hour. For example for the 3-hour ahead model, the feature set utilized 3-hour lagged load, within 24-hour, 48-hour, and 168-hour lags. The temperature data for all 10 regions were taken 3 hours prior. Also, the temperature forecast for the next 3 hours, 2 hours, 1 hour, and the hour of prediction were used. The calendar data, on the other hand, was directly incorporated into the models without any shift since it's known data and doesn't require any time adjustment.

For the TFT model, similar strategy to the one used for day-ahead prediction was adopted. Feature set included temperature and dew point of 10 different points as well as 24-hour, 48-hour and 168-hour lags, weather forecast and calendar data were incorporated into the model as a future known variable. The outcomes from the variable selection network for 6 hour ahead prediction is shown at Figure 3.6, underscore the most critical variables for the model. Similarly the encoder choose the weekday the 24-hour lag, and the temperature are ranked as the most important. For decoder the most important features were hour, holiday lag and weekday and 48-hour load lag.

Hyperparameter Tunning

For Random Forest and XGBoost models hyperparameters were tuned similarly as previous example by Cross Validation with Time Series Split and Random Search, separately for each of models. TFT hyperparameters were also found by optuna test with results presented at the Table 3.3.

Number of parameters in network was 265.9k. The hidden size for the 36-hour model is larger than the 6-hour model, since it refers to the dimensionality of the output space. This suggests that the 36-hour model requires more complex internal representations to learn the patterns in the data due to the longer

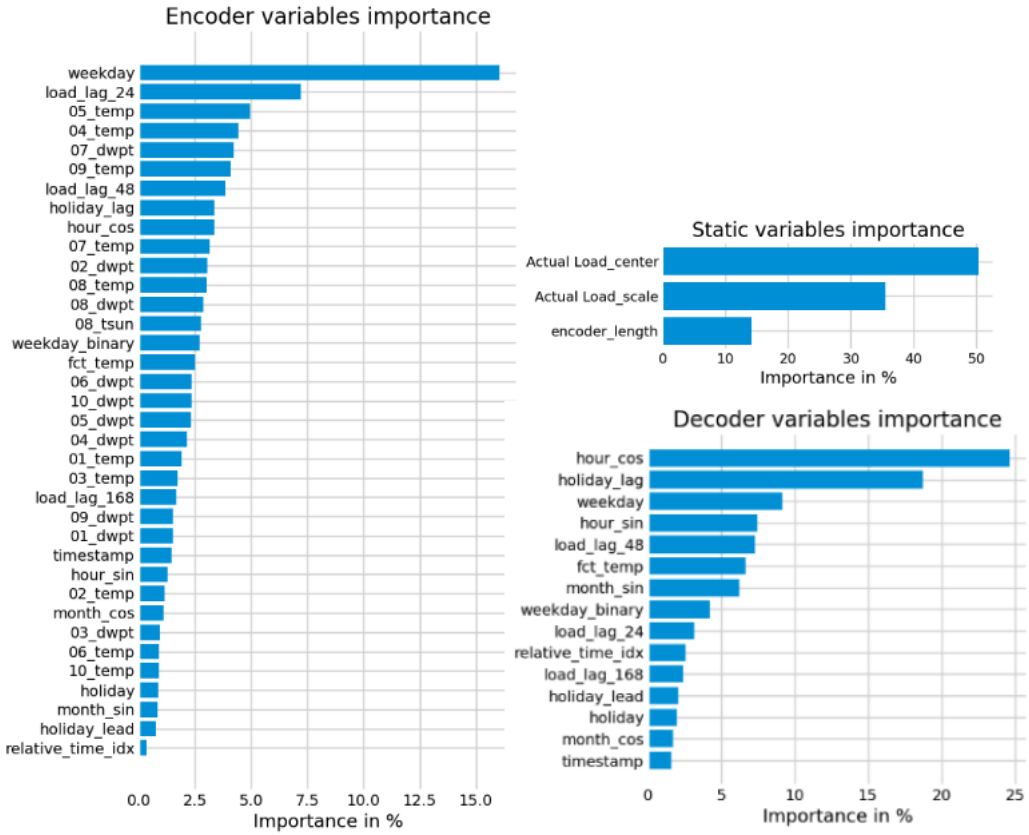


Figure 3.6: TFT Variable selection network for 6 hours ahead prediction.

Hyperparameter	**Optimal Value**
Learning rate	0.001
Hidden size	30
Attention head size	1
Dropout rate	0.3
Continuous hidden size	25
Number of LSTM layers	2
Loss function	Quantile Loss

Table 3.3: Optimal hyperparameters for the TFT model for 6 hour ahead prediction.

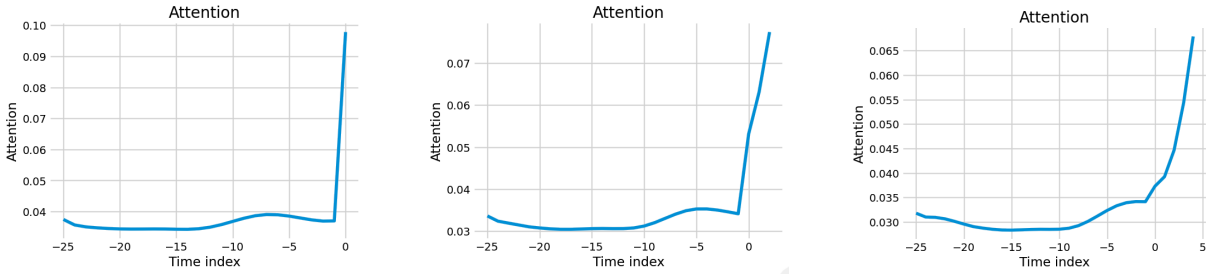
prediction horizon. Similarly, the 36-hour model has a larger attention head size compared to the 6-hour model. This is likely because a longer forecast horizon requires more complex attention mechanisms to adequately capture the temporal dependencies in the data.

The 36-hour model has a higher dropout rate than the 6-hour model. Dropout is a regularization technique that helps prevent overfitting by randomly setting a fraction of input units to 0 during training. A higher dropout rate in the 36-hour model suggests that it needs a stronger regularization to prevent overfitting due to the increased complexity from the longer prediction horizon.

Lastly, the continuous hidden size in the 36-hour model is also greater than in the 6-hour model. This indicates that the longer forecast horizon requires more complex transformations of the continuous inputs.

Attention

Figure 3.7 shows model attention for time steps 1, 3 and 6 hour ahead.



(a) Attention at the time index 0. (b) Attention at the time index 3. (c) Attention at the time index 6.

Figure 3.7: Attention scores for 6 hour ahead forecasting horizon.

In the case of the 6-hour ahead prediction, the model tends to prioritize recent hours leading up to the forecast. This is due to the fact that the near-term patterns and trends in the data often have a significant impact on short-term forecasts. As the prediction horizon decreases, the immediate past becomes increasingly relevant for accurate forecasting.

Furthermore, the model assigns higher attention to its own forecasts. This indicates that the model recognizes its own predictions as useful input for subsequent forecasting. Essentially, the model's own forecasts serve as an estimate of future trends based on the patterns it has learned, providing valuable context for short-term forecasting.

Model Evaluation

The performance of the models on the test dataset for each forecast hour is presented in Tables 3.4-3.9. These tables show the relative performance of each model as the forecast horizon increases. The hourly presentation of the performance measures provides a measure of not only the overall accuracy of each model, but also how well they handle the additional uncertainty and complexity that arises when forecasting over a longer time horizon.

Looking initially at the 1-hour prediction results, it is apparent that the XGB model demonstrated the best performance among the three models, achieving the lowest MSE, MAE, and MAPE, along with the highest R^2 value. The TFT model, despite outperforming the RF model in terms of R^2, lagged in the other three metrics. Baseline model, that predicts the same value as actual value for one hour ahead has MSE metric - 86 228.67, MAE - 211.95, R^2 score - 0.949 and MAPE - 0.029.

However, as we move further into the future, a significant shift is observable in the 2-hour prediction results. The TFT model starts to showcase its strengths, besting the RF model in all the metrics and narrowly trailing the XGB model, with nearly identical MAE and MAPE values. This trend of the TFT model improving its relative performance becomes more pronounced with the increase in prediction horizon. From the 3-hour prediction onwards, the TFT model consistently outperforms the other two models in all metrics. It delivers the lowest MSE, MAE, and MAPE, indicating smaller errors, and highest R^2, implying a better fit to the actual data. By the 6-hour prediction, the TFT model demonstrates a marked superiority, yielding lower error rates (MSE, MAE, MAPE) and a higher goodness-of-fit (R^2) than both the RF and XGB models. Its performance clearly highlights the strength of the TFT model in handling the complexities and uncertainties associated with longer-term predictions, affirming its effectiveness as a temporal sequence forecasting tool.

Metric	RF	XGB	TFT
MSE	19 030.74	11 869.06	27 997.44
MAE	98.93	79.76	126.04
MAPE	0.015	0.012	0.019
R^2	0.983	0.989	0.975

Table 3.4: Metrics for 1 hour ahead prediction model.

Metric	RF	XGB	TFT
MSE	45 589.36	31 578.79	30 142.08
MAE	150.15	130.59	129.36
MAPE	0.022	0.019	0.019
R^2	0.958	0.972	0.973

Table 3.5: Metrics for 2 hour ahead prediction model.

Metric	RF	XGB	TFT
MSE	79 918.75	51 931.29	32 504.74
MAE	192.6	165.21	132.37
MAPE	0.0286	0.024	0.020
R^2	0.927	0.953	0.971

Table 3.6: Metrics for 3 hour ahead prediction model.

Metric	RF	XGB	TFT
MSE	88 545.66	61 974.87	34 649.29
MAE	200.75	178.72	135.34
MAPE	0.03	0.026	0.020
R^2	0.92	0.943	0.969

Table 3.7: Metrics for 4 hour ahead prediction model.

Metric	RF	XGB	TFT
MSE	89 435.30	72 090.319	36 791.83
MAE	200.85	190.75	139.09
MAPE	0.03	0.028	0.020
R^2	0.919	0.934	0.967

Table 3.8: Metrics for 5 hour ahead prediction model.

Metric	RF	XGB	TFT
MSE	89 832.28	72 925.27	39 717.40
MAE	200.07	191.13	143.83
MAPE	0.029	0.0284	0.021
R^2	0.918	0.934	0.964

Table 3.9: Metrics for 6 hour ahead prediction model.

Figures 3.8 - 3.11 shows prediction of this three models with 1 hour, 3 hour and 6 hour forecast horizons. In Figure 3.8, which shows the 1-hour prediction, we might observe a close alignment of the models' predictions with the actual values. As the prediction horizon extends to 3 and 6 hours, as shown in Figures 3.9 and 3.10, the disparity between predicted and actual values tends to increase for all models due to the inherent complexities in longer-term forecasting. However, TFT model's predictions, while not improving, degrade at a slower rate than those of the RF and XGBoost models. This relative stability of the TFT model's performance over time, in contrast to the more rapidly deteriorating accuracy of the RF and XGBoost models, underscores the TFT's resilience in the face of longer-term forecasting challenges.

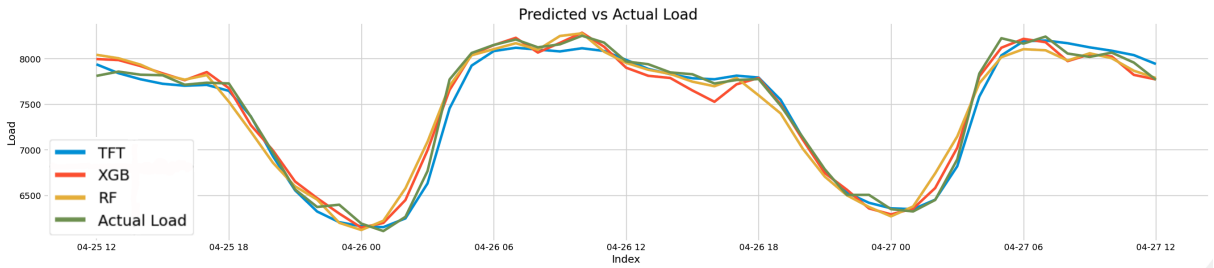


Figure 3.8: Predictions by 1 hour ahead models TFT, XGBoost, and Random Forest.

Lastly, Figure 3.11 presents prediction made by all 6 models for Random Forest, XGBoost and TFT, reinforcing the observations made from the previous figures. With each incremental hour, the performance of Random Forest and XGboost models degrades, deviating more from the actual values contrasted with the relative stability displayed by the TFT model. This trend is consistently noticeable

across all forecast horizons, highlighting the limitations of RF and XGBoost in managing the complexities of longer-term forecasting.

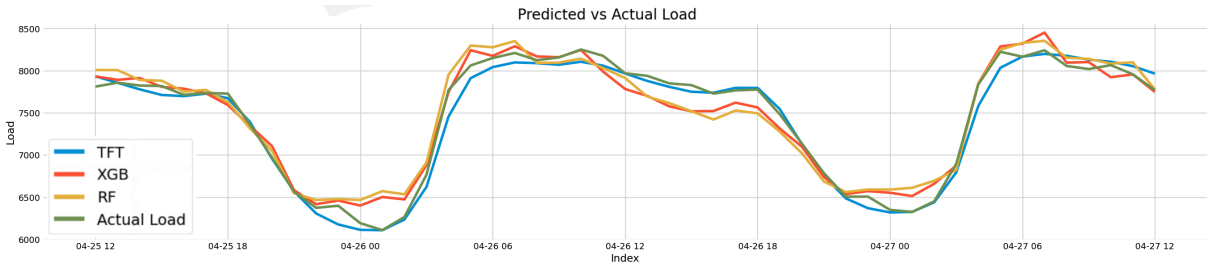


Figure 3.9: Predictions by 3 hour ahead models TFT, XGBoost, and Random Forest.

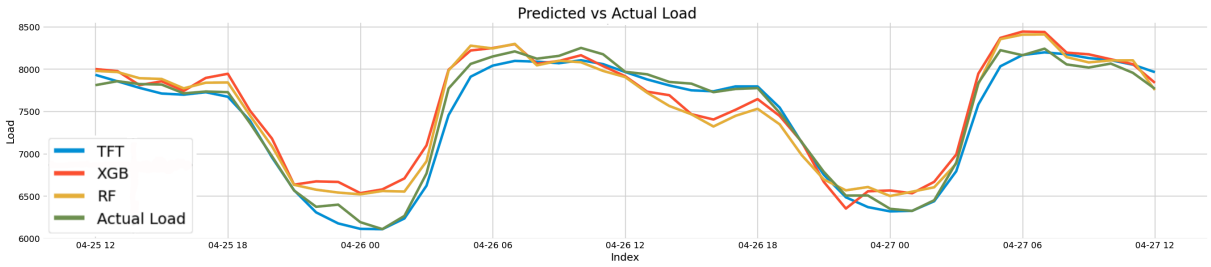


Figure 3.10: Predictions by 6 hour ahead models TFT, XGBoost, and Random Forest.

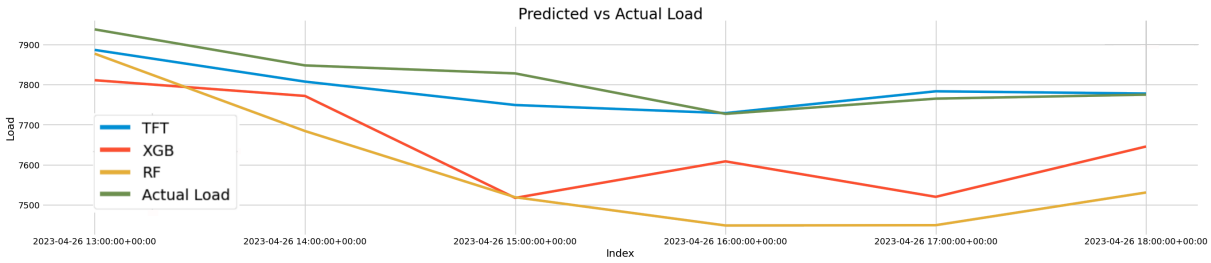


Figure 3.11: 6 hour prediction made by different model each hour TFT, XGboost and Random Forest.

Ensemble Model

In an attempt to further improve the accuracy and robustness of predictions, an ensemble model was implemented. This approach takes advantage of the individual strengths of each of the previously evaluated models - Random Forest, XGBoost and TFT. The model that performs best in a given environment is selected, resulting in a more adaptable model that can perform under a wide range of conditions. Also ensemble approach can improve the reliability of predictions. Due to the diversity of models in the ensemble, the final model is less susceptible to the limitations, biases, and potential errors of each individual model.

After predictions of all three models was made, these predictions were employed as training data for an additional model. A second-layer Random Forest model was established, specifically trained on the output of the initial models. Alongside these predictions, the model was also informed by relevant calendar data. The ensemble model was applied using a direct method, i.e. a different model was used for

each individual hour of forecasting. The basis for this approach was the differences in the performance of the baseline models at different time horizons.

To validate and evaluate the model, the dataset was divided into a training and a test dataset. The training set included data from 26 August 2022, which was validation set for Random Forest and XGBoost, onwards and the test set then included data from 15 May 2023 to 1 July 2023.

The results presented in Table 3.10 confirm the performance of the ensemble model, showing its excellence over the individual models in all metrics.

Hour	**MSE**	**MAE**	**MAPE**	R^2
1	11 986.67	84.26	0.0127	0.989
2	22 054.96	111.01	0.0165	0.980
3	26 843.78	120.90	0.0179	0.976
4	28 290.67	122.72	0.0182	0.974
5	29 828.21	126.36	0.0187	0.973
6	32 899.92	141.03	0.0205	0.969

Table 3.10: Performance metrics of Ensemble model.

In the all of prediction horizons, the ensemble model outperforms the standalone models, demonstrating the enhanced predictive power achieved by combining individual forecasts. As we extend the prediction horizon, the ensemble model exhibits stability despite the increase in the complexity associated with longer-term forecasts.

Figure 3.12 shows prediction made by ensemble models. By comparing the error spread of the ensemble model with those of standalone models, it's evident that the ensemble approach has resulted in a reduction of prediction errors.

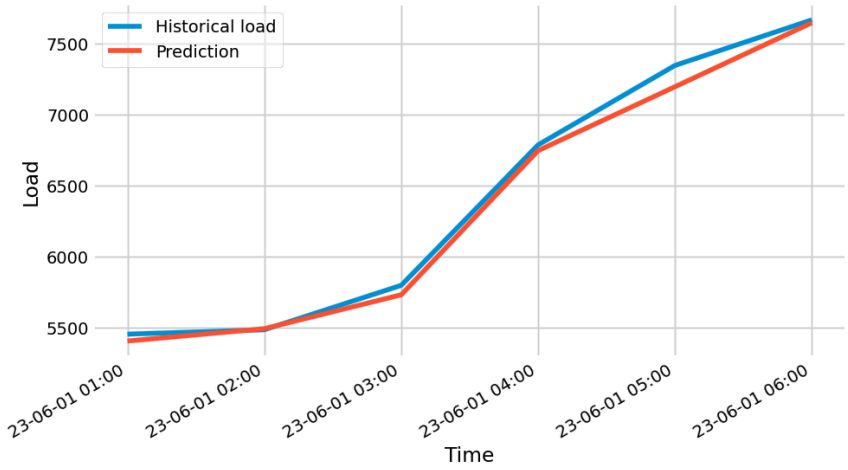


Figure 3.12: 6 hour prediction by Ensembe model.

Conclusion

The objective of this study was to investigate and validate the effectiveness of deep neural network methodology for accurate prediction of future energy consumption, specifically using the Temporal Fusion Transformer model as an example. The research covered both short-term (6 hours) and long-term (24 hours) forecasting scenarios, reflecting the practical needs of intraday market operations and daily market planning, respectively. The models, trained using historical, weather and calendar data, offered promising results. It was discerned that while conventional models like XGBoost and Random Forest demonstrate effectiveness in capturing non-linear relationships, they struggle with handling the nuances of temporal dependencies inherent in time-series data. In contrast, the TFT, underpinned by its intricate architecture and competency in discerning temporal dynamics, significantly outperformed the other models in the 6-hour forecasting scenario.

However, it's noteworthy that the TFT model did not retain its superior performance when stretched to the 36-hour prediction horizon. In this longer-term forecasting scenario, the TFT was observed to be less effective, indicating that the model's performance may vary significantly based on the complexity and length of the prediction horizon. This finding underscores the importance of selecting suitable models based on specific forecasting tasks and their requirements.

The study took a leap further by implementing an ensemble model, which amalgamated the predictions of the TFT, Random Forest, and XGBoost models. It combined the unique strengths of each model, mitigating their individual weaknesses and enhancing prediction accuracy.

In conclusion, as we traverse the landscape of the energy sector marked by increasing complexity and a pressing need for efficient management, advanced forecasting models like TFT and ensemble methods can play crucial roles. The study demonstrates the potential of advanced deep learning models and the efficacy of ensemble modeling in the context of energy demand forecasting. However, it's crucial to emphasize that the performance of these models can significantly vary based on the prediction horizons and the nature of the task at hand. Therefore, continuous efforts are required to optimize these models for different scenarios and to explore their integration with other machine learning techniques. The next step could also involve testing models with relevant forecast weather data and bigger train and testing datasets for ensemble model.

Bibliography

[1] Nti, I. K., Teimeh, M., Nyarko-Boateng, O., & Adekoya, A. F. (2020). Electricity load forecasting: A systematic review. Journal of Electrical Systems and Information Technology, 7, 13.

[2] Ediger, V. Ş., & Akar, S. (2007). ARIMA forecasting of primary energy demand by fuel in Turkey. Energy Policy, Elsevier.

[3] Chen, T., and Guestrin, C. (2016, August). Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining (pp. 785-794).

[4] Breiman, L. (2001). Random Forests. Machine Learning, 45, 5–32. https://doi.org/10.1023/A:1010933404324

[5] Dupond, Samuel (2019). "A thorough review on the current advance of neural network structures". Annual Reviews in Control. 14: 200–230.

[6] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in Neural Information Processing Systems (pp. 3104-3112).

[7] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780.

[8] Chung, J., Gülcehre, C., Cho, K., & Bengio, Y. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. CoRR, abs/1412.3555.

[9] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. CoRR, abs/1706.03762.

[10] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

[11] Lim, B., Zohren, S., & Roberts, S. (2020). Temporal fusion transformers for interpretable multi-horizon time series forecasting. In Proceedings of the 8th International Conference on Learning Representations (ICLR 2020).

[12] Wu, N., Green, B., Ben, X., & O'Banion, S. (2020). Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case. arXiv preprint arXiv:2001.08317.

[13] Nielsen, A. (2019). Practical time series analysis: prediction with statistics and machine learning. O'Reilly.

[14] Wang, Z., Wang, Y., Zeng, R., Srinivasan, R. S., & Ahrentzen, S. (2018). Random Forest based hourly building energy prediction. Energy and Buildings, 171, 11-25.

[15] De Felice, M., Alessandri, A., & Catalano, M. (2020). Random forests for high-dimensional and large-scale problems. Information Sciences, 509, 289-307. DOI: 10.1016/j.ins.2019.09.001.

[16] Breiman, L. (1996). Bagging predictors. Machine Learning, 24, 123–140.

[17] Freund, Y., & Schapire, R. E. (1999). A Short Introduction to Boosting. AT&T Labs Research.

[18] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

[19] APA. Chollet, F. (2017). Deep learning with python. Manning Publications.

[20] Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics.

[21] Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep Learning. MIT Press.

[22] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint. Retrieved from https://arxiv.org/abs/1609.04747.

[23] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference for Learning Representations.

[24] Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., & Han, J. (2019). On the variance of the adaptive learning rate and beyond. arXiv preprint. Retrieved from https://arxiv.org/abs/1908.03265

[25] Smith, L. (2018). A Disciplined Approach to Neural Network Hyperparameters.

[26] Prechelt, L. (1998). Early stopping — But when? In G. Orr & K. Müller (Eds.), Neural networks: Tricks of the trade. Springer.

[27] Bengio, Y. (2012). Practical Recommendations for Gradient-Based Training of Deep Architectures.

[28] Hinton, G. E. (1986). Distributed representations. In Proceedings of the Eighth Annual Conference of the Cognitive Science Society.

[29] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15(56), 1929-1958.

[30] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. arXiv preprint. Retrieved from http://arxiv.org/abs/1512.03385

[31] Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2).

[32] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint. Retrieved from http://arxiv.org/abs/1502.03167

[33] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. arXiv preprint. Retrieved from https://arxiv.org/abs/1607.06450

[34] Pascanu, R., Gulcehre, C., Cho, K., & Bengio, Y. (2013). How to construct deep recurrent neural networks.

[35] Martens, J., & Sutskever, I. (2011). Learning recurrent neural networks with Hessian-free optimization. In Proceedings of the International Conference on Machine Learning.

[36] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 1724-1734).

[37] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate.

[38] Plevris, V., Solorzano, G., Bakas, N., & Ben, S., M. (2022). Investigation of performance metrics in regression analysis and machine learning-based prediction models. 10.23967/eccomas.2022.155.

[39] Fix, Evelyn; Hodges, Joseph L. (1951). Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties (PDF) (Report). USAF School of Aviation Medicine, Randolph Field, Texas. Archived (PDF) from the original on September 26, 2020.

[40] Guo, C., & Berkhahn, F. (2016). Entity embeddings of categorical variables. Neokami Inc.

[41] Mahajan, T., Singh, G., & Bruns, G. (2021). An experimental assessment of treatments for cyclical data. California State University Monterey Bay

[42] Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). Classification and regression trees. CRC press.

[43] Lundberg, S. M., & Lee, S. I. (2017). A unified approach to interpreting model predictions. In Advances in neural information processing systems (pp. 4765-4774).

[44] Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016). Fast and accurate deep network learning by exponential linear units (ELUs). In Proceedings of the International Conference on Learning Representations (ICLR).

[45] Géron, A. (2017). Hands-On Machine Learning with Scikit-Learn and TensorFlow. O'Reilly Media, Inc.

[46] RandomForestRegressor - scikit-learn 0.24.2 documentation. Scikit-learn. Retrieved 30.7.2023, from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[47] XGBoost Documentation (version 1.4.2). XGBoost. Retrieved 30.7.2023, from https://xgboost.readthedocs.io/en/stable/

[48] TemporalFusionTransformer - PyTorch Forecasting 0.9.0 documentation. Retrieved 30.7.2023, from https://pytorch-forecasting.readthedocs.io/en/stable/api/pytorch_forecasting.models.temporal_fusion_transformer.TemporalFusionTransformer.html

[49] pandas 1.3.3 documentation. pandas. Retrieved 30.7.2023 from https://pandas.pydata.org/docs/

[50] sklearn.model_selection - scikit-learn 0.24.2 documentation. Scikit-learn. Retrieved 30.7.2023, from https://scikit-learn.org/stable/modules/classes.html#module-sklearn.model_selection

[51] Optuna Documentation (version 2.10.0). Optuna. Retrieved 30.7.2023, from https://optuna.readthedocs.io/en/stable/