Evaluation of selected DL algorithmic methods for interpretable time series forecasting

Benedikt Rein
Chair of Information Systems
Humboldt-University of Berlin

Yuliya Vandysheva
Chair of Information Systems
Humboldt-University of Berlin

Alican Gündogdu
Chair of Management Science
Humboldt-University of Berlin

March 24, 2023

Contents

Abstract

Time series forecasting is a problem of major interest across many domains and still has many potentially uncharted territories waiting to be discovered. This paper addresses a global time series forecasting with a focus on interpretability and summarizes the findings of the seminar course project, based on the experiments conducted in the article "Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting" (Lim et al., 2020). Therefore, the primary objective of this research is to explore and reproduce the results demonstrated in the above-mentioned article. We evaluated 3 different algorithmic methods including ARIMA, Temporal Fusion Transformer (TFT), and NeuralProphet on the same dataset. The results demonstrate that the experiment can be replicated and comparable performance can be achieved.

1 Introduction

1.1 Motivation

For countless business establishments, one of the most crucial steps for ensuring the healthy and profitable operation is to plan ahead of time. In most manufacturing and service sector companies, this necessity drives the fundamental decision-making scenarios. Almost every aspect of an enterprise entails this process: setting the price of a good, deciding the amount of production vis-à-vis the expected demand, forecasting future revenues for fiscal and financial decisions, etc. In order to plan ahead for such aspects and beyond, a business has to make accurate future predictions based on past values and trends. As a result, time series forecasting is a problem of major interest across many domains, such as but not limited to retail, energy, ride-share, and healthcare.

It is indeed apt to call it "a problem" since time series forecasting is a challenging endeavour above all else. Although there are various problems with time series forecasting, problems for global time series forecasting are more relevant for the purposes of this paper because it aims to address a global time series forecasting algorithm. Main issues in that regard consist of the difficulty of interpretability, heterogeneity of data sources and multi-horizon forecasting as it involves forecasting at multiple time steps in the future. Even though they pose a significant challenge to the models attempting time series forecasting, the variety of different algorithms that are used to address such problems is quite large, ranging from traditional statistical time series forecasting models to deep neural networks.

Especially in the last two years, the situation with time series forecasting has changed considerably. Particularly, with the advent of Temporal Fusion Transformer and NeuralProphet, the scenery for time series forecasting, notably the multi-horizon forecasting, has been ushered into a whole new level.

This paper specifically deals with TFT, based on the experiments conducted in the article "Temporal Fusion Transformers for interpretable Multi-horizon Time Series

Forecasting" by Lim et al. (2020). This paper's aim is to reproduce and go over the results proposed by the article in question by applying the TFT model to the datasets as was done in the article, which are ranging from big datasets to small and noisy ones as well as from univariate to multivariate ones. In addition, this paper intends to carry out minor ameliorations by using a more user-friendly implementation of the algorithm utilizing PyTorch applications of TFT.

1.2 Research objective

Time series forecasting is a vastly huge field with an abundance of resources in the forms of models and algorithms, but still has many potentially uncharted territories waiting to be discovered. Although each model, old or new, deserves a scrutiny on their own, this paper focuses on TFT and the article introducing it, as mentioned above. Consequently, the goals of this study can be summarized as follows:

1. Reproducing and re-implementing the experimental results of the original paper (Lim et al., 2020) as a state-of-the-art model
2. Using different models, including benchmarks, for an empirical comparison of alternative techniques for multi-horizon forecasting
3. Using a more user-friendly and readily available implementation of TFT by utilizing the PyTorch application of the algorithm and comparing usability of all implemented models.

The premises and findings of the original paper, as well as the findings of this paper, will all be discussed at length in the following sections.

2 Background

2.1 Time-series forecasting

Time series forecasting is of major concern in business sector and brings together with itself a number of problems due to its nature. To define it in a nutshell, time series forecasting is the prediction of values of a variable in the future, either at a single point or over a period of time, based on the analysis of the past and current data and trends in order to come up with reliable projections to fuel planning processes. So, time series forecasting basically aims to enable us to plan for the future, which is intrinsically unknown for the present time. That is why it is hailed as a fundamental component of decision-making, not only for businesses of course, but also for almost every type of organization that requires planning ahead of time.

Nevertheless, time series forecasting is neither as easy nor as straightforward as it sounds through its definition. It is because time series forecasting involves inherently complex problems that are hard to overcome. Time patterns that display serious

fluctuations, such as trends or seasonality, along with the likely non-linear relationships variables have between each other are among the major issues a model needs to deal with when it comes to time series forecasting. Moreover, the often limited relevant time series data poses a significant strain on the algorithms for their training process. Last but not least, exogenous factors that affect time series data like economic and political environment, catastrophes etc. further complicates the modelling.

Time series forecasting can be basically classified into two sections: univariate and multivariate forecasting. The former implies forecasting the values of a single variable in the future, while the latter refers to forecasting the values of multiple variables in the future. This classification can be broadened by two more parameters. First, if a time series, whether it be uni- or multivariate, is forecasted at multiple points in the future, then it involves multi-horizon forecasting. Second, time series forecasting can be done either with an iterative or global method, meaning it can be done either via forecasting all points using the values of previous forecasted points or via forecasting a single point using the whole time series.

As stated earlier, there are numerous models and algorithms designed to deal with time series forecasting, for each type and method mentioned above, varying between traditional statistical models and deep neural networks. AR, ARIMA, DeepAR and LSTM can be given as examples for algorithms dealing with time series forecasting, the former two of which can be considered as statistical models whereas the latter two are neural networks. However, the introduction of TFT as a state-of-the-art model for multi-horizon and global time series forecasting has significantly stepped up the game, which will be investigated in more detail.

2.2 Related work

In the article (Lim et al., 2020) that is the primary focus of this work, researchers introduced their novel deep learning model for the time-series forecasting problems – Temporal Fusion Transformer. As the name implies, the proposed TFT architecture is a transformer-based DNN that integrates the mechanisms of several other neural architectures. In their experiment, TFT is demonstrated to outperform not only all benchmarks but also the other alternative neural networks of different types such as DeepAR, MQRNN, and LSTM Seq2Sep. The performance is evaluated on four datasets – two univariate and two complex datasets. According to the authors, the model is particularly versatile to handle a wide range of challenges, such as multi-horizon forecasting or a variety of different input data sources.

TFT, as the article suggests, utilizes and expands upon what a variety of DNN algorithms has to offer. In order to understand the true merits of TFT, it is necessary to go over significant DNN algorithms dealing with the same or similar problems regarding time series forecasting, just as done in the article.

Foremost, just like the generic multi-horizon forecasting algorithms, DNNs designed to address this forecasting problem can be divided into two major approaches: Iterative and direct algorithms (Brownlee, 2017). Iterative algorithms aim to predict each point

in the future by building upon the point before, taking it as the input, projecting the target point that way and then applying the same procedure all over again for the next point. To sum up, such algorithms function in a "t+1" manner, where t gets updated with each next point in the future. On the other hand, direct models strive for coming up with forecasts simultaneously for a multitude of points in the future, namely horizons. For iterative algorithms, Deep AR and Deep State-Space Models (DSSM) can be given as examples with underlying Long Short-term Memory (LSTM) methods whereas direct methods based on sequence-to-sequence models could be exemplified by, for instance, Multi-horizon Quantile Recurrent Forecaster (MQRNN) algorithm. Algorithms belonging to either class come with their own weaknesses and strengths. Iterative algorithms tend to be more basic and economical. Nevertheless, they presuppose that the variables used as inputs for predicting future points in question are given – their values are revealed. Yet, the reality about variables can be quite different and there can be many unknown ones in most cases. Direct algorithms are comparably more complex, but they can attain more accurate results. However, the main issue with direct algorithms revolves mostly around the fact that they offer low levels of interpretability, which is pretty important for such complex models. In that sense, one of the goals of TFT is to overcome such challenges for both types of methods by addressing the different nature of inputs for iteration-based models and by focusing on the interpretation of attention patterns for the direct methods. Last but not least, TFT also intends to deliver meaningful interpretation as to not only instance-wise insights into the attention patterns like other models but also the global mechanisms and dynamics of the algorithm on the entire datasets, promising a more sturdy grasp of the whole picture.

3 Methodology

3.1 ARIMA

ARIMA, i.e. Auto Regressive Integrated Moving Average, is a class of statistical models for understanding and prediction of a given time series based on its own past values, that is, its own lags and the lagged forecast errors. The algorithm consists of 3 components (Asteriou & Hall, 2016):

1. **Autoregression** (AR) that takes into account the dependence between observation and certain past values.
2. **Integrated** (I) uses the differencing of raw observations necessary to make the series stationary, meaning that any substantial trends are removed from the data.
3. **Moving Average** (MA) accounts for the dependent relationship between an observation and a residual error of a moving average model applied to certain past observations.

3.2 TFT

The TFT brings many components of deep neural networks together. Leveraging advantageous blocks while bypassing elements that might bring drawbacks for the specific configuration of the dataset and defined hyperparameters. It is one of the most state-of-the-art models for global time series forecasting, and thus one of the most interesting models to evaluate (Lim et al., 2020).

1. **Variable selection networks** are used at every time step to select the most relevant static data and the most relevant for the encoder and decoder. Encoder and decoder VSN can already be enriched with information from static variables.
2. **Static covariate encoders** are used for encoding the static input to add static context at multiple components in the model.
3. **Sequence-to-Sequence Layer** for implicit temporal and locality encoding. Using an encoder-decoder-structure to take past inputs and known future inputs. Additionally, it is the second possibility for enrichment with static data.
4. **Normalization and Gating** to skip the previous LSTM layer, if it is deemed disadvantageous for the overall performance.
5. **Static enrichment layer** using GRNs as the central element for encoding static data.
6. **Temporal Self-Attention** for capturing long-term trends and correlations in the data. The TFT introduces a new implementation of attention, aiming at improving explainability by using multiple heads that can learn different temporal patterns.
7. **Position-wise Feed-forward** to allow skipping the attention-mechanism if the added complexity is not necessary, and to provide a direct connection to the Sequence-to-Sequence layer.
8. **Quantile Outputs** for not only making point predictions but also minimizing the loss of all predefined quantiles, adding an extra layer of explainability.

3.3 Neural Prophet

NeuralProphet is developed in a fully modular architecture that combines classic statistical components and neural networks. Each module has its individual inputs and modeling processes and contributes an additive component to the forecast. The training proceeds with mini-batch stochastic gradient descent and imposes each module to revise its proper forecast to minimize the overall loss. The model predicts h outputs, where h defines the number of steps ("horizon") to be forecasted into the future at once (Triebe et al., 2021).

The authors defined six major components of the model, which are briefly summarized below.

1. **Trend** to capture the general pattern of time series. The trend effect is given by the steady growth rate multiplied by the difference in time. Moreover, NeuralProphet can identify the dates where there is a clear variation in the trend. Between these change points, the trend is modelled linearly. Thus, a trend can be modelled either as a linear or a piece-wise linear trend by using change points.
2. **Seasonality** for capturing the periodic movements of the time series. Seasonality is modelled using Fourier terms and thus can handle multiple seasonalities for high-frequency data. The model allows for three types of seasonal periodicities such as daily, weekly and or yearly seasonality depending on data frequency and length, which can be activated additively or multiplicatively to the trend.
3. **Auto-regression** to capture dependence over time among the variable in a series. Auto-regression is handled using an implementation of AR-Net, an Auto-Regressive Feed-Forward Neural Network for time series.
4. **Lagged Regressors** are used to correlate other external variables, which only have values for the observed period, to the target time series. Lagged regressors are also modelled using separate Feed-Forward Neural Networks.
5. **Future Regressors** are external variables which have known future values for the forecast period. Future regressors are modelled as covariates of the model with dedicated coefficients.
6. **Events and Holidays** are occasional events that may affect the behavior of the target over time and are modelled analogously to future regressors but as binary time series.

4 Experimental design

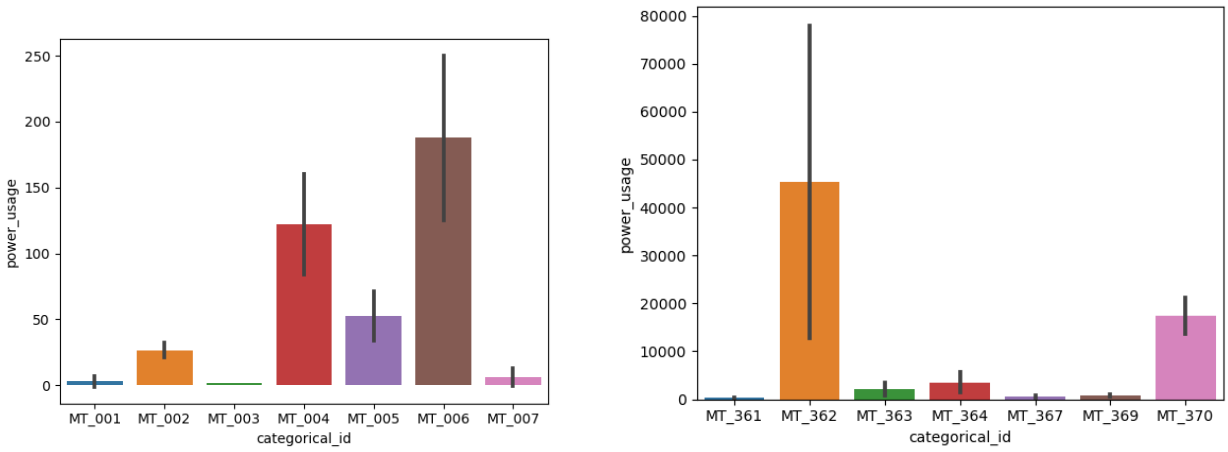
In order to fulfil the research objectives, there are different parts to the experiment. The first and most important part of this research is to implement or reuse an existing TFT model and verify, that the model is able to learn from temporal data. After this sanity check, the first major goal is to reproduce the performance stated in the TFT paper. To not rely solely on the reported values, we also implement a stochastic baseline model. In order to represent the vast field of time-series forecasting solutions, ARIMA is selected as a stochastic model and NeuralProphet as a neural network-based comparison. Both are used for benchmarking and comparing different techniques for multi-horizon forecasting. Besides performance, explainability and ease of use are the other major dimensions for evaluating the TFT model.

4.1 Dataset

In the TFT article, 4 datasets are used to display the model's strength and versatility vis-à-vis datasets with different characteristics. This could normally require multiple models for successful multi-horizon time-series forecasting. To that end, the article

embarks on applying the model to *electricity* and $traffic$ datasets, which exhibit univariate time series with known inputs. In addition, a *retail* dataset with multivariate time series and complex inputs is used for further display of the model's power. Finally, in order to see how the model would perform against a small and noisy dataset, a dataset for financial application of volatility forecasting is used.

Nevertheless, due to constraints on computational resources and time and especially to make sure, that we are able to offer an appropriate analysis, this paper aims to take up the *electricity* dataset used in the article. Its purpose is to apply the model via a more user-friendly application based on PyTorch, replicate the results of the article and ensure its superiority through quantifiable performance metrics. For the latter purpose, the same dataset will be applied to ARIMA and NeuralProphet algorithms for comparison.



(a) (b)

Figure 1: Distribution of the target variable for some IDs

The *electricity* dataset contains the electricity consumption of 370 customers, identifiable by the ID, aggregated on an hourly level. Additionally, there are several covariates derived from the *timestamps* alongside the target such as an *hour*, a $dayoftheweek$, and *months*, that are considered as known inputs. The distribution of a target variable, *power_usage*, is different for each ID, as shown in Figure 1. The dataset is preprocessed in the same way as provided in the Google paper (Lim et al., 2020), which implies that no missing values are present and only a normalization step is needed.

4.2 Model Selection

ARIMA

For a baseline, we select the stochastic ARIMA model. It is one of the preferred models for creating a baseline, as mentioned in Lim et al., 2020 and Li et al., 2019. Similar to NeuralProphet, it is using auto-regression but has the limitation of only modelling linear relationships. Because we are also modelling daily seasonality, the correct term

would be S(easonal)ARIMA. ARIMA can predict confidence intervals, when set to 90%, those representing the 10th and 90th percentiles, making the comparison to TFT and NeuralProphet percentile predictions possible. It is not designed for global modelling, thus making it necessary to create one model per time series. To keep the code base fully in Python, we selected the *pmdarima* package, which is an implementation of the R package *arima* that offers the ARIMA estimator and automated feature optimization inside the $AutoARIMA$ function.

Temporal Fusion Transformer

The TFT is a deep learning-based neural network and one of the most recognized time series forecasting solutions of the last years. For implementing our experimental setup, there are two options. We can either just re-run the code provided alongside the TFT paper (Lim et al., 2020) or use one of the PyTorch implementations provided in the Packages PyTorch-Forecasting or Darts.

As a first step, we selected using a PyTorch implementation, because we have some experience working with it and its general reputation of having easier workflows than TensorFlow. PyTorch-Forecasting is taking a big step towards standardizing time series forecasting, using their predefined $TimeSeriesDataSet$. Additionally, PyTorch-Lightning is offering a lot of functionality to take care of training handling, scaling and hosting models in the cloud, compared to the implementation offered alongside the paper, which is more on the research side. Those arguments, alongside a concise documentation, for the PyTorch packages, led to our decision.

NeuralProphet

NeuralProphet represents a theoretically strong improvement to our baseline ARIMA model, allowing modelling multiple seasonalities instead of just one and including neural networks for non-linear auto-regression. Thus, NeuralProphet bridges the gap between statistical models and global deep-learning models. However, the model was initially intended to be simple and user-friendly, performing relatively well in the context of local forecasting and one-step-ahead predictions. Global modelling, which is a prime focus for our objectives, is a recent addition to this forecasting tool. For more complex data and purposes, not all components are yet optimized, e.g., easy implementation to train on GPU or customizing data loaders. It is one of the models in the forecasting field that is gaining a lot of traction. The developers just recently published a preprint project discussing their integration with PyTorch-Forecasting and PyTorch-Lightning with an intention of making NeuralProphet more accessible, easy to work with, and more scalable (Voskoboinikov & Pilyugina, 2021).

4.3 Performance Metrics

When it comes to measuring the performance of algorithms against a selected dataset and against other algorithms performing similar tasks, there are numerous metrics, and often a certain combination of them, that are used for such purposes. Even though all of them offer valuable insights and have their own merits, our first choice of such a metric will be P50 loss, which refers to the median or 50th percentile of the loss function values determined during the training procedure, and Mean Absolute Error (MAE), which calculates the average absolute difference among the projected and actual values of the target variable. The reasons for that choice are actually quite straightforward. The ease of use of those metrics coupled with their persistent use in previous papers as well as in the TFT article suggests that we should do the same. So, to facilitate comparability, we opted to go with those metrics as our first choice and to that end, we made sure that the target is always normalized the same way as was done in the TFT article and previous papers.

Our second choice in terms of performance metrics would be P90 loss and confidence interval. The former is the 90th percentile of the values calculated for the loss function throughout the training procedure, whereas the latter is the statistical metric that lays out a range of values that, with a given level of confidence, the real value of a target variable is predicted to fall within. Again, the rationale of the choice for the second metric is similar to that of the first one: convenient operation with insights easy to understand and widespread use in the literature.

Additional dimensions used for evaluating the different models are usability and explainability. Because there is not much real-world value to a model that only a few people are able to implement and achieve a good performance with, usability is a key factor when deciding on what packages to use. Explainability is crucial to get a better understanding of the underlying correlation in the data and to pick, for example, the correct seasonality and in the case of the TFT an appropriate encoder length.

Last but not least, we aim to report further metrics relating to variable importance accompanied by visualization of predictions, other confidence intervals and attention. Our first two choices along with further metrics will be explained and explored in the 5th part of the paper, namely the "Results and Finding" section.

4.4 Model Training

The training is configured to replicate the settings and configurations used in TFT as much as possible to ensure that the models are comparable. Since the models are inherently very different, we attempt to give all of them the opportunity to analyse as much data as possible. The models are applied separately on the same dataset. In order to align the results of all the models, we ensure that we use the same inputs for each model if possible. Data is standardized using a z-score normalization for each time series separately, reusing the code from the TFT article. The dataset is divided into three sets: training (for learning), validation (for evaluating a fitted model and

hyperparameter tuning), and test set (for evaluation of the performance of a final model). The models predict the next 24 hours based on the observations of last week (24h x 7). The model's detailed setups with the hyperparameters as well as fitted models are available in the related repository and Jupiter notebook. One notable deviation from other research is the test horizon. While other experiments predict on two weeks over the test set or implement a moving window, we only input the last week and predict on the last day of the test set, due to time constraints and internal comparability of our implementation. With the final goal of evaluating P50 and P90 loss, the ideal loss function for all models would be quantile loss of those predefined quantiles. This is however just possible with the TFT implementation, with NeuralProphet and ARIMA not allowing completely custom loss functions and building on either Torch or SciPy predefined ones, which do not offer quantile loss. For the NeuralProphet we select the MSE, for penalizing outliers more strongly and Limited-memory BFGS for the ARIMA model, because it is the default selection.

4.4.1 TFT

In the TFT article, the training procedure for the *electricity* dataset involves 6 iterations over the training subset. As stated earlier, this paper aims to reproduce and evaluate the results presented by the TFT article. Moreover, this paper focuses on the *electricity* dataset exclusively. Therefore, this paper will carry out the same procedure for the same dataset put forth by the TFT article. For hyperparameter tuning, we utilize PyTorch-Forecasting's *optimize_hyperparameters* functionality, searching over a pre-defined range via OPTUNA algorithm (Akiba et al., 2019). OPTUNA is a framework designed to facilitate the hyperparameter tuning process by trying to find the optimal parameter combinations with high performance. It does this by describing a search space of potential hyperparameters which may be discrete, continuous, or conditional. After that, it traverses through the search space using a combination of randomized search, Bayesian optimization, and pruning strategies to identify the ideal collection of hyperparameters that maximizes the performance of the machine learning model.

The following parameters are searched over for 3h and 100 trials:

n_trials=100,

max_epochs=20,

$gradient\_clip\_val\_range = (0.01, 0.2)$,

$hidden\_size\_range = (16, 160)$,

$hidden\_continuous\_size\_range = (8, 64)$,

$attention\_head\_size\_range = (1, 4)$,

$learning\_rate\_range = (0.0005, 0.01)$,

$dropout\_range = (0.1, 0.3)$,

$$
\begin{aligned}
&reduce\_on\_plateau\_patience = 4,\\
&trainer\_kwargs = dict(limit\_train\_batches = 100,\ \ max\_epochs = 20,\\
&\qquad\qquad\qquad log\_every\_n\_steps = 5,\ \ accelerator = GPU,\\
&\qquad\qquad\qquad devices = 1).
\end{aligned}
$$

After the application of OPTUNA, the following rounded parameters are returned:

$$
\begin{aligned}
&gradient\_clip\_val : 0.052,\\
&hidden\_size : 128,\\
&dropout : 0.15,\\
&hidden\_continuous\_size : 32,\\
&attention\_head\_size : 2,\\
&learning\_rate : 0.007.
\end{aligned}
$$

A deviation from the TFT article's implementation is the hyperparameter *hidden_continuous_size* being present, where the original article defines hidden layers just once. When adding both hidden sizes together, we however have exactly the same $hidden_layer$ size as in the references article. After the hyperparameters tuning is handled and parameters with the most use are decided, it is time for the final training. The final training is done on university-provided servers with "NVIDIA Tesla V100" GPU to get the most computational power available to us. Training time for the *electricity* dataset takes a very comparable time to that mentioned in the TFT article, taking slightly above one hour for a full epoch over the training dataset with 8 epochs in total. Validation loss is monitored for early stopping with 'EarlyStoppingCallback' and 'CheckPointCallback' to save training states to resume training in the event of a disconnect or a similar occurrence. In order to determine if we are overlooking any bottlenecks, we utilized 'DevicestatsMonitor'. Consequently, running training epochs and batches are reported as the main time consumer, therefore no alarming bottleneck is discovered. As a result, 'DevicestatsMonitor' is not used for final runs because the logging consumes a sizable amount of time. Lastly, for logging training information, we use 'TensorBoardLogger' and 'lr-finder' along with the 'ADAM' optimizer, which is an algorithm created for deep learning for stochastic gradient descent (SGD) optimization. However, logging libraries can be used with it in order to monitor and visualize training parameters in the course of the training process.

For data normalization, we have two options. Either we can reuse the implementation in the TFT paper, or we can use PyTorch-Forecastings built-in ones.

When re-using the TFT articles solution, we did not manage to rescale the data in the correct way, using the previously fitted normalizers. Thus, leaving us with only normalized output for visualizations.

When using the built-in normalization, the *TimeSeriesDataSet* only contains the real values and an array with scaling factors. In contrast, the data loader contains the normalized data but mapping those back to the correct time step and ID is not trivial. Additionally, the model does not offer the option to output the normalized values. Because benchmarking is mostly done on normalized loss, the need to create a normalized output is the main driver for the decision to re-use normalization from the TFT paper.

4.4.2 ARIMA

Since ARIMA acts as a baseline model in this work, investing too much time in hyperparameter tuning seems unreasonable. We utilize pmdarima Python library with the AutoARIMA functionality that not only estimates the best parameters but also handles automatically auto-correlation and the requirement of stationarity, i.e., differencing the time series (Smith et al., 2017).

As stated earlier, ARIMA is not intended to be a global model, and there is a large divergence between the different id-specific time series in the *electricity* dataset. Thus, the adopted approach involves fitting 370 models on the respective data to achieve more specific predictions. We fit the models on around 50% of the full data, selecting the time steps right before the test data. Using this data to do 10 optimization trials with the AutoARIMA function. This is recommended as a good trade-off between time and performance in the documentation. Finally, for reducing the size of the training data, taking into account that sometimes between 50 and 150 data points can already be enough (Box & Tiao, 1975). This number is supposed to be strongly linked to the number of data points per season. Considering the results of the TFT article, we make the assumption that daily seasonality is most prevalent in the data – thus, the number of time periods should be a multiple of 24. For our experiment, we use approximately three months of data (with 2000 data points), which should be sufficient to learn this seasonality.

This leads to an assumed final runtime of around 5–10 hours on a CPU. Given the fact, that no parallelization is used, it can be assumed to be a smaller training investment, compared to the 5 hours of GPU training of the TFT. Parallelization is possible according to the documentation, however, a strong recommendation for an iterative approach is made (Smith et al., 2017).

To keep comparability in mind, we keep the same time horizon for testing.

4.4.3 NeuralProphet

In the framework of our experiment, we test both methods of normalization, one from the TFT implementation and one built into the NP model. For the latter, global modelling is enabled with local normalization, which implies exactly the same procedure as applied for the TFT, i.e., each time series has data normalization parameters associated with each ID provided. The model is fitted on the full training data set to predict

the same quantiles 0.1, 0.5, and 0.9 as other models. Training takes around 60 minutes with 10 epochs over the whole data and the parameter *num_workers* set to 40 on a Xeon CPU. All the 3 default seasonalities are included additively, as well as additional regressors. For the final evaluation, the implementation with external normalization is used, and the built-in one is used for some visualizations.

Finally, we intend to save the trained model for later use.

5 Results and Findings

Before looking at the most crucial topic of predictive performance, we think usability and insights we gained while working with the models, especially because they are so new, are very important.

5.1 Usability

Temporal Fusion Transformer being built on top of already quite established packages like PyTorch-Lightning gives the TFT a very clear structure. The provided documentation makes handling the training manageable. The trained models are portable with a model size of around 3 MB, when keeping all folder structures etc. the same, one can just download the repository and reuse fitted models without a problem. The central *TimeSeriesDataSet* class is an extraordinarily useful tool for cross-package or cross-model functionality and removing potential problems during usage. NeuralProphets current transition to those packages underlines their central role in the future of deep learning-based time series forecasting.

This *TimeSeriesDataSet* has however also caused a lot of headaches during training. The build-in 'TargetNormalizer', 'NaNLabelEncoder', and 'GroupScaling' are a high value-add to the workflow. The data structure of the data loader made it difficult to compare the final input values for the model, because they only save the actual values and the factors for normalizing, thus computing the actual inputs only during runtime. When setting the TFT model to not use the inbuilt target scaling, the model still shows being instantiated with target scaling, creating some confusion. The performance of the model still leads to the assumption, that it is not actually used, or, that already normalized values would not be changed with an added normalization step. Admittedly, in a production setting, this would not really matter.

Getting used to this complex model and very extensive libraries took a lot of time, but once this is done, the libraries take care of many of the tasks, letting the user focus on the parts, they deemed important.

Working with the statistical model, **ARIMA** made the usually mentions drawbacks quite clear. Not having the option to create a global model, users are forced to fit a model for every time series, leading to a potentially more complex backend for handling more than 100 different models. This also brings one positive aspect with it, the

option to potentially invest more time to explore especially bad performing models. For those, there are many options to explore for improving performance. This, again, leading to the big drawback of many overfitted models to be monitored and reimplement once performance decreased. Working with the AutoARIMA model was a very positive experience. Giving the user many options on how to lay out the model, but also to let the model find optimal parameters over a predefined search space.

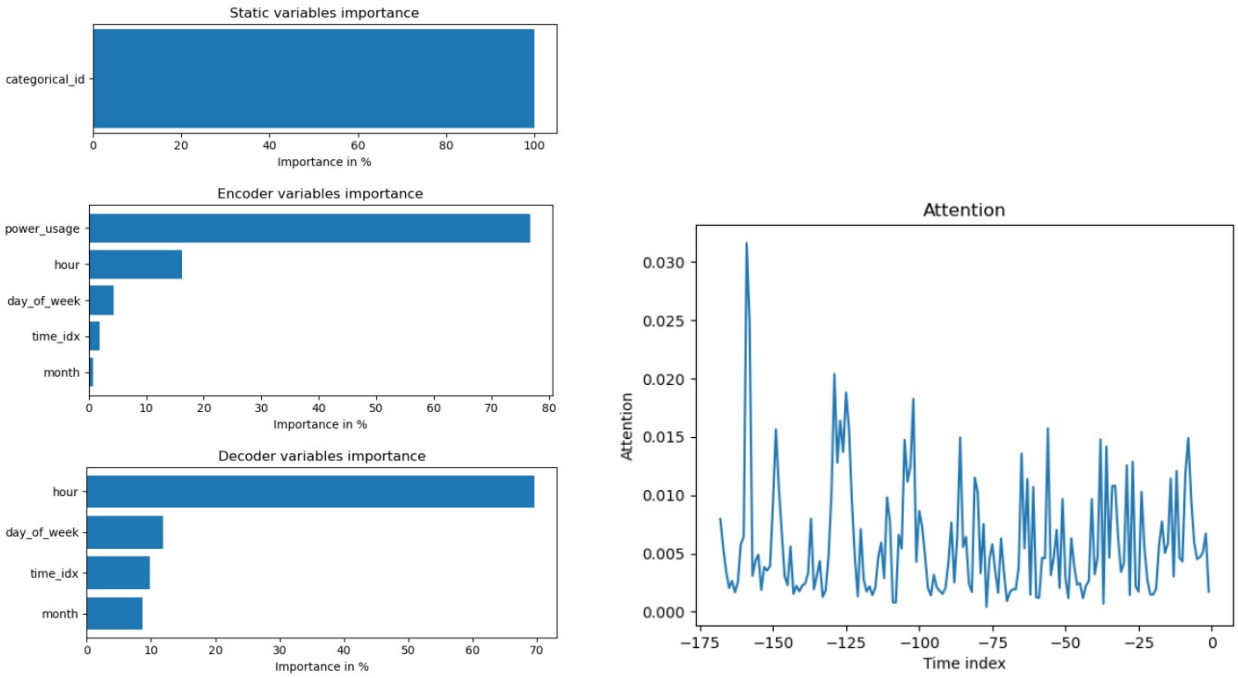
At the moment, the usability of the **NeuralProphet** package is the decisive drawback. Presently it is in a major development cycle, turning the package structure, taken from the Prophet package, into a structure compatible with PyTorch-Forecasting and PyTorch-Lightning. The Lightning-Trainer is already in use but is not easily accessible via the API. The central problem seems to be, turning a local, statistical model into a global deep-learning model which needs to be able to handle a magnitude more data. Adding the need for a data loader, parallelization and more sophisticated ways of saving the model.

Some workflows and workarounds for Prophet are not possible with NeuralProphet (github.com, 2019). For example, to reduce the model size, you can drop the saved dataset inside the Prophet model, this is however not possible with NeuralProphet currently. While other models just need the *electricity* dataset and the saved model, with a combined size of 220 MB, the fitted NeuralProphet model takes up around 14 GB. However, after the integration with PyTorch-Forecasting and PyTorch-Lightning is finished, those drawbacks should be eliminated and usability should be very similar to the PyTorch-Forecasting implementation of the TFT. Additionally, for interpreting the predictions, the trained model is still needed.

5.2 Explainability

The TFT article suggests that TFT, among many other things, differentiates itself from other similar DNNs through its enhanced explainability, which is a crucial feature for a complex algorithm like a DNN. In our application aiming for the reproduction of the results presented by the TFT article, we are able to witness the superiority of TFT in that matter. Firstly, the user is offered a variety of functions to test the statistical correlations and dependencies in the selected data, making it easy for the user to make informed decisions on how to work with the data. Secondly, a multitude of visualizations are offered for understanding seasonality and trends in the data.

As we can see in Figure 2, TFT provides a plethora of visuals to achieve better explainability. Firstly, the variable importance, extracted from the weights of the variable selection network, which is the quantification of the effect each variable has in relation to the prediction target of the algorithm, is displayed in detail. It is divided into three subcategories: static variables importance, encoder variable importance and decoder variable importance. $CategoricalID$ has a quite high importance percentage, as can be seen in Figure 2. The variables *powerusage* and *hour* are the front-runners for encoder variable importance and decoder variable importance, respectively. Apart from that,



(a) *variable importance* (b) *attention plot*

Figure 2: TFT

the attention mechanism makes it possible for the model to selectively concentrate on different parts of the input data when carrying out predictions. This is again aptly described via Figure 2. Lastly, TFT offers valuable insights and visualizations of the predictions, plotting observed and predicted normalized values, attention score, P50 and P90 predictions all in one easy to understand plot, as seen in Figure 3. Showing the prediction specific loss, indexing the input period with negative values and the prediction period with positive ones. You can clearly see a daily seasonality, which the TFT is learning, albeit missing a small intraday spike, which is still almost covered by the P90 prediction.

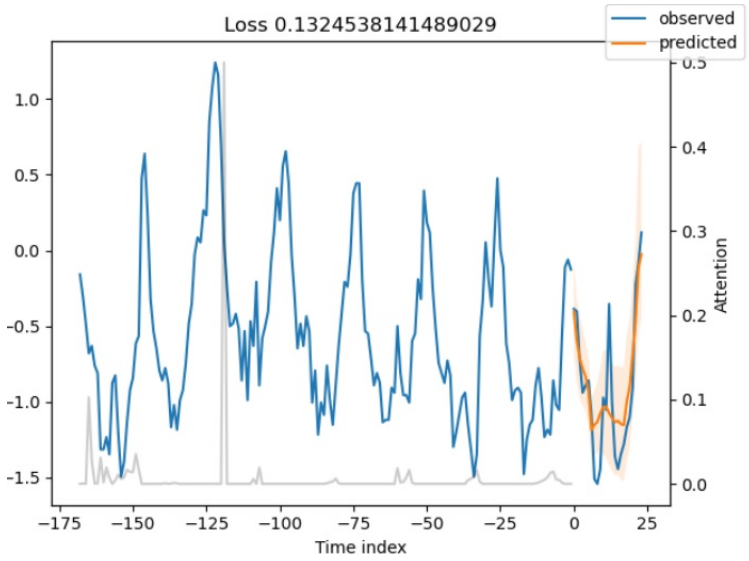
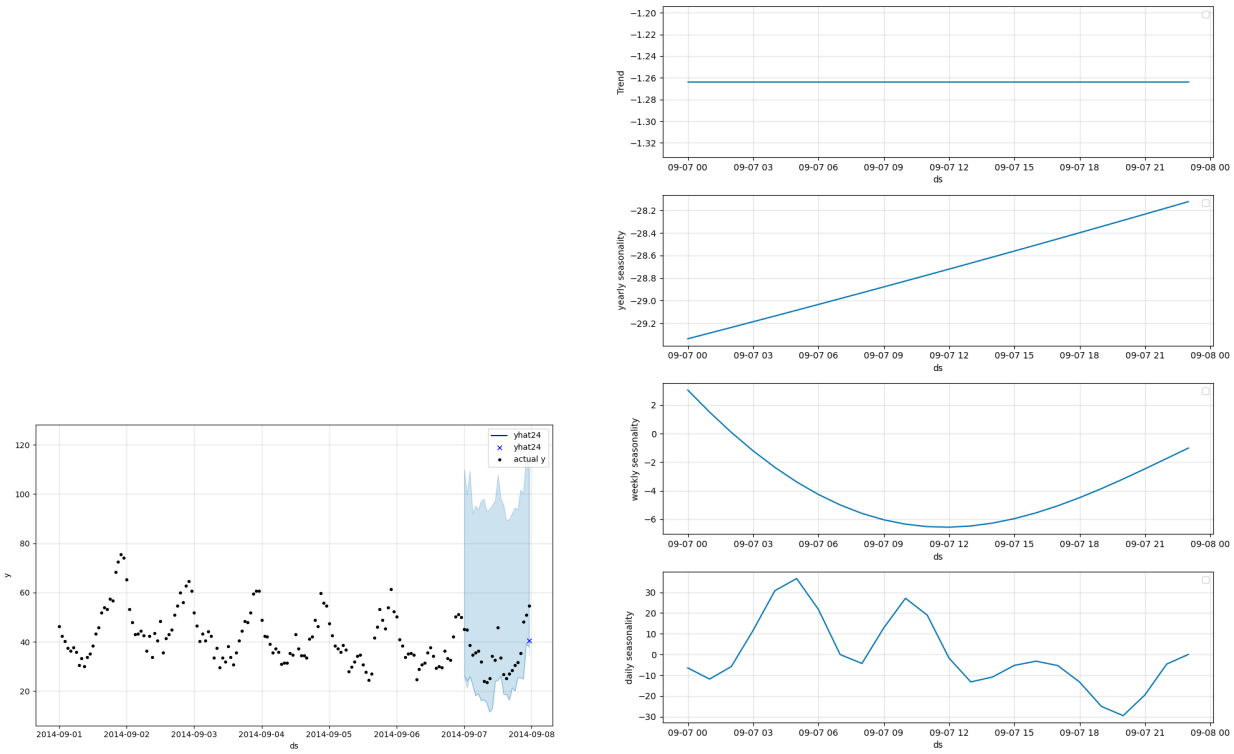


Figure 3: Example test predictions for ID MT_004

NeuralProphet also offers many ways for plotting predictions, different seasonalities, trends, lagged regressors, and uncertainty which makes it very easy to get a better understanding of the underlying data and forecast components. Thus helping the user to make more informed decisions about how to use the model, set parameters, and find possible irregularities in the data. From Figure 4, which depicts the one-week forecast for some random ID for the NeuralProphet, electricity consumption appears to have weekly and daily seasonality, while also showing a yearly trend. Using in-build functions for visualizations again shows, that handling data is not trivial. Even though the model has the whole data frame saved inside itself, yearly and weekly seasonalities are plotted over one day, and customization is not offered via the API. Thus, we should mention, that the visualizations created with NeuralProphet are not completely dependable. In general, modular composability is shown to be clearly relevant for interpretability purposes.



(a) *forecast plot* (b) *component plot*

Figure 4: Neural Prophet

5.3 Performance

With a comparably small time-investment, we are able to achieve a normalized P50 and P90 loss, which is in a comparable range to other reported **ARIMA** baselines. Please keep in mind, that other implementations use a rolling window (Salinas et al., 2020) to do predictions over the test dataset, while we only predicted on the last day of the dataset. We do this for all models to keep internal comparability high while accepting a loss of comparability to other research. With a P50 loss of 0.102, we are outperforming other baselines by around 33%, due to the above-mentioned reasons we do not want to attribute this outcome any scientific significance. The P90 loss shows an improvement of around 40% compared to other baselines.

When looking at **NeuralProphets** performance, on the one hand, we can report training metrics, that are not as good as the baseline or the TFT, but at least show, that the model is fitting to the data in some way. Descriptive values would be a validation MAE of 0.359 or an MSE training error of 0.229. On the other hand, we are not able to extract the prediction values in a way, to reproduce the MAE on the test set or calculate the P50 and P90 loss. While the model might actually perform in a comparable, but worse, range, we are not able to prove this. For a closer look at all the training metrics, we are providing a notebook in the repository accompanying

this paper. According to our calculations, NeuralProphet has a normalized P50 loss of 0.462 and a normalized P90 loss of 0.832. Those values are so far off from what is shown to be possible by other models and also so far off from the training metrics, that we have to assume a mistake on our side. We redesigned the way of extracting the needed data from the predictions at least once, due to some mistakes.

Just as a hypothetical consideration, one could discuss that with the training loss of around 0.23 and the P90 usually being half of the P50 loss, we could assume a P50 loss of around 0.115 and a P10/P90 loss of around 0.055. However, those are just assumptions and the loss is calculated with the MSE, not quantile loss.

For now, we cannot validate the performance of the model stated in their publication. Further research is needed, and we recommend a reimplementation once NeuralProphet has transferred to PyTorch-Lightning and PyTorch-Forecasting.

Lastly, we look at the performance of the **TFT** model. During training, we could not report any validation loss improvements after the first epoch. When loading the checkpoint states from multiple epochs and evaluating using the test dataset, the model from the first epoch is also proving to be the best performing. Because the learning rate should decay after continuous stagnation over 4 epochs by a factor of 0.2, it is difficult to argue for a too high learning rate, which fits to the data very fast but also plateaus very fast.

We are able to show an MAE improvement of 16.6% compared to the ARIMA model, a P50 improvement of 13.3% and a P90 improvement of 41.0%. This should be seen in the context of the ARIMA model already performing better than other ARIMA baselines as mentioned above.

Compared to the TFT article's metrics, our model performs 39% worse on the P50 loss and 36% worse on the P90 loss than the TFT. With the TFT paper's TFT model outperforming their ARIMA baseline by 180% and 278% respectively, our ratio of improvement is significantly lower. Again leading to the conclusion, that future research should emulate the testing part of the previous research as closely as possible. Due to the time constraints of this research, we are not able to make those changes in a timely manner.

The fact, that our best-performing model just took one epoch, leads us to conclude, that training metrics and hyperparameter tuning should be evaluated more closely. If many hyperparameter trials already reach their best performance during epoch one, it is likely, that not much insight is gained. This indeed was the case for some trials, which is however also to be expected during hyperparameter tuning, according to our experience.

6 Discussion

All in all, time series forecasting is a bold, complex and challenging endeavor, even more so when it is multi-horizon and done in a global manner. Our approach, in

	ARIMA	**NP**	**TFT**
MAE	0.319	0.925	0.276
P50	0.102	0.462	0.090
P90	0.059	0.831	0.042

Table 1: Normalized MAE, P50 and P90 losses on electricity dataset.

that sense, was to explore TFT, a state-of-the-art multi-horizon global time series forecasting algorithm, in relation to its original paper. Moreover, we sought to present a comparison between TFT and other existing models. ARIMA, a traditional statistical time series forecasting model, and NeuralProphet, Meta's DNN, were our selections to compare and contrast TFT with different and similar algorithms.

In our application of TFT via PyTorch, we focus on one dataset, namely electricity, and we are able to replicate the results of the TFT article and reached similar performances, which is in tandem with our research objectives. Although the improvement over ARIMA is not as drastic as it is in the article, it is still quite significant, and the difference is to be expected when one takes the disparity in terms of machine-learning research experience and time into account. Aside from that, TFT's main strength comes from its highly improved explainability vis-à-vis other black box models. Especially through PyTorch-Forecasting, it offers numerous functions and visualizations for testing the statistical correlations and dependencies in the selected data. Furthermore, PyTorch application of TFT was user-friendly and helpful. It did take some time to calibrate and deploy the model, but after that, it can be considered user-friendly. It would be safe to assume that PyTorch applications could be the future of time series forecasting after further improvements and simplification of use. In brief, TFT is overall a robust and easy-to-use algorithm, promising significantly improved results over the previously existing models. Also, the PyTorch applications facilitate its usage and promise to deliver more in the future.

Our second DNN for comparison, NeuralProphet, is also a very promising model for time series forecasting. Nevertheless, its current state necessitates further research and ameliorations, especially in terms of its PyTorch application. It is currently not very optimized when it comes to working with big data sets. In addition, the output of predictions as well as visualizations are suboptimal and not quite as easy to deploy as opposed to TFT. However, NeuralProphet can be expected to have a bright future once those challenges are overcome because it has a quite simple yet robust structure. So, it can again be one of the hallmarks of time series forecasting in the future, along with TFT as a PyTorch application.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). Optuna: A Next-generation Hyperparameter Optimization Framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. https://optuna.org/

Asteriou, D., & Hall, S. (2016). ARIMA Models and the Box-Jenkins Methodology. In *Applied Econometrics, 3rd edition* (pp. 275–296).

Box, G. E. P., & Tiao, G. C. (1975). Intervention analysis with applications to economic and environmental problems. *Journal of the American Statistical Association*, *70*. https://doi.org/10.1080/01621459.1975.10480264

Brownlee, J. (2017). 4 Strategies for Multi-Step Time Series Forecasting [Online; accessed 2023-03-10]. https://machinelearningmastery.com/multi-step-time-series-forecasting/

github.com. (2019). [Online; accessed 2023-03-12]. https://github.com/facebook/prophet/issues/1159

Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X., & Yan, X. (2019). Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. *arXiv e-prints*. https://doi.org/10.48550/arXiv.1907.00235

Lim, B., Arik, S. O., Loeff, N., & Pfister, T. (2020). Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting. *arXiv e-prints*. https://doi.org/10.48550/arXiv.1912.09363

Salinas, D., Flunkert, V., Gasthaus, J., & Januschowski, T. (2020). DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, *36*. https://doi.org/10.48550/arXiv.1704.04110

Smith, T. G., et al. (2017). pmdarima: Arima estimators for Python [Online; accessed 2023-03-22]. http://alkaline-ml.com/pmdarima/tips_and_tricks.html#parallel-vs-stepwise

Triebe, O., Hewamalage, H., Pilyugina, P., Laptev, N., Bergmeir, C., & Rajagopal, R. (2021). NeuralProphet: Explainable Forecasting at Scale. *arXiv e-prints*. https://doi.org/10.48550/arXiv.2111.15397

Voskoboinikov, A., & Pilyugina, P. (2021). NeuralProphet. *github.com*. https://github.com/neuralprophet/neural_prophet_lightning/blob/master/reports/Third_Status_Report_TFDS.pdf