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 Information Systems
Humboldt-University of Berlin

March 20, 2023

Contents

Abstract

This paper summarizes the findings of the course project, based on the experiments conducted in the article "Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting" [1] by Bryan Lim, Sercan O. Arik, Nicolas Loeff, Tomas Pfister.

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 the 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 endeavor above all else. Although there are various problems for time series forecasting, some of which will be cited later, 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 the time-series forecasting has changed considerably. Particularly, with the advent of Temporal Fusion Transformer (TFT) and Neural Prophet, 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" [1] by Bryan Lim, Sercan O. Arik, Nicolas Loeff, Tomas Pfister. The paper's aim, in that sense, is to reproduce and go over the results proposed by the article in question by applying the TFT model into 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, in general, is a vastly huge field with an abundance of resources in the forms of models and algorithms, but still has many potential 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-implementation the experimental results of the original paper [1] as a state-of-the-art model

2. Using different models, including benchmarks to compare with TFT/ Empirical comparison of alternative techniques for multi-horizon forecasting

3. Using a more user-friendly and readily available implementation of TFT by utilizing PyTorch application of the algorithm

The premises and findings of the original paper, the theory behind it as well as the findings of and the re-implementation by this paper will all be discussed at length in the following sections.

2 Background

2.1 Time-series forecasting

Time series forecasting, as stated earlier, 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 a 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 organizations 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 modeling.

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 periods to come. 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, which puts more burden on the algorithm in addition to what it already has due to the nature of time series forecasting. Second, a 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 [1] that was the primary focus of this work, researchers introduced their novel deep learning model for the time-series forecasting problems – Temporal Fusion Transformer (TFT). 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. Moreover, by taking advantage of the modified Multi-Head attention mechanism, the model provides interpretable insights into feature importance.

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 conveniently be divided into two major approaches: Iterative and direct algorithms. As stated earlier, 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, or horizon, 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 could baxed 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 parsimonious. 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 type 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 the algorithm before but also the global mechanisms and dynamics of the algorithm on the entire datasets, promising a more sturdy grasp about the whole picture.

3 Theory/ 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 [9]:

1. **Autoregression** (AR) that takes into account the dependence between an observation and certain past values.

2. **Integrated** (I) uses the differencing of raw observations necessary to make the series stationary.

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 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.

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 [2].

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/ Procedures

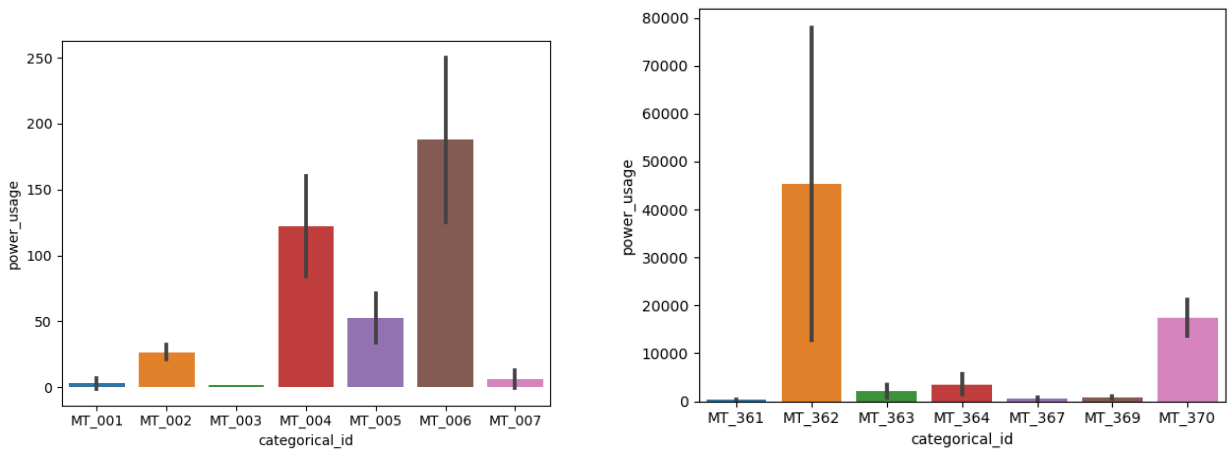
To fulfil the research objectives, there are different parts to the experiment. The first and most important part of this research, is to implement or reuse an existing TFT model and verify, that the model is able to learn from temporal data. After this sanity-check, the first major goal is to reproduce the performance stated in the TFT paper. To not just rely on the reported values, we are also implementing 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 have been used to display the model's strength and versatility vis-à-vis datasets with different characteristics, which could normally require multiple models for successful multi-horizon time-series forecasting. To that end, the article embarks on applying the model on 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 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 preferred.

Nevertheless, due to constraints on computational resources and time, this paper aims to take up on the electricity dataset used in the article. Its purpose is to apply the model via a more user-friendly application 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 electricity consumption of 369 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 day of the week, 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, which implies that no missing values are present and only 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 [1] and [3]. Similar to NeuralProphet, it is using auto-regression, but has the limitation of only modeling linear relationships. Because we are also modeling daily seasonality, the correct term would be S(easonal)ARIMA. ARIMA can predict confidence intervals, when set to 90%, those represent the 10th and 90th percentiles, making the comparison to TFT and NeuralProphet percentile predictions very intuitive. It is not designed for global modeling, 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 [1] 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 'TimeSeriesDataFrame'. 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 modeling 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 and serves as such in this work. However, the model was initially meant to be simple and user-friendly, performing relatively well in the context of local forecasting and one-step-ahead predictions. Global modeling, which is a prime focus for our objectives, introduces a recent addition to this forecasting tool. For more complex data and purposes not all components are yet optimized, e.g., possibility to train on GPU or using dataloaders. 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 [8] discussing their integration with PyTorch-forecasting

and PyTorch-lightning with an intention of making NeuralProphet more accessible, easy to work with, and more scalable.

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 suggest 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 lie inside. Again, the choice for second metric is similar to that of the first one: convenient operation with insights easy to understand and widespread use in the literature.

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 / Analysis" section.

4.4 Model Training

The training has been configured to replicate as much as possible the settings and configurations used in TFT to ensure that the models are comparable. Since the models are inherently very different, we attempt to give all of them the opportunity to analyze 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 are 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 (24 x 7). The model's detailed setups with the hyperparameters as well as fitted models are available in the related notebook.

4.4.1 TFT

In the TFT article, the training procedure for electricity dataset involves 6 iterations over the training set of *electricity* dataset. As stated earlier, this paper aims to reproduce and evaluate the results presented by the TFT article. Moreover, this paper focuses on 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-Forecastings' *optimize_hyperparameters* functionality, searching over a pre-defined range via OPTUNA algorithm [6].

OPTUNA is a framework designed to facilitate the hyperparameter tuning process by trying to find the optimal parameter combinations with utmost performance. It does that 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 have been searched over for 3h and 100 trials:

n_trials=100,
max_epochs=20,
$gradient\_clip\_val\_range = (0.01, 0.2)$,
$hidden\_size\_range = (16, 160)$,
$hidden\_continuous\_size\_range = (8, 64)$,
$attention\_head\_size\_range = (1, 4)$,
$learning\_rate\_range = (0.0005, 0.01)$,
$dropout\_range = (0.1, 0.3)$,
$reduce\_on\_plateau\_patience = 4$,
$trainer\_kwargs = dict(limit\_train\_batches = 100,\ \ max\_epochs = 20,$
$log\_every\_n\_steps = 5,\ \ accelerator = GPU,$
$devices = 1)$.

After the application of OPTUNA, the following rounded parameters have been returned:

$gradient\_clip\_val$: 0.052,
$hidden\_size$: 128,
$dropout$: 0.15,
$hidden\_continuous\_size$: 32,
$attention\_head\_size$: 2,
$learning\_rate$: 0.007.

So, after the hyperparameters tuning is handled and parameters with most use are decided, it was time for the final training. The final training was done on university provided servers with "NVIDIA Tesla V100" GPU to get the most computational power available to us.

Training time for the *elecricity* dataset took actually a very comparable time to that of TFT article, taking slightly above one hour for a full epoch over the train 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 missing 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 "ADAM", 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 the built-in ones.

When re-using the TFT solution, we did not manage to rescale the data properly, because the output data frame is very different, leaving us with only normalized output for visualizations.

When using the build-in normalization, the TimeSeriesDataFrame only contains the real values and an array with scaling factors. In contrast, the dataloader contains the normalized data but mapping those back to the correct time step and ID is not intuitive. 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, 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 autocorrelation and the requirement of stationarity, i.e. differencing the time series.

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 369 models on the respective data to achieve more specific predictions. We fit the models on around 50 percent of the full data, selecting the time steps right before the test data. Doing 10 optimization trials with the AutoARIMA function.

This is recommended as a good trade-off between time and performance in the documentation. Finally, we reduce the size of the training data, taking into account that sometimes between 50 and 150 data points can already be enough [4]. This number is supposed to be strongly linked to the number of data points per season. Considering the results of the TFT paper, 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 used 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.

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 modeling 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, 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, the built-in one is used for some visualizations.

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

5 Results and Finding / Analysis

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

pytorch forecasting/lightning as a combination made a lot of the project easier. central TimeSeriesDataSet class is an extrordinary usefull tool for cross-package/cross-model functionality and removing potential problems during the usage. neuralprophets change to those modules underlines their central roll in the future of deep learning based timeseries forecasting.

this TimeSeriesDataSet has also however caused a lot of headache during training. build in targetnormalizer, categorical encoding, and groupscaling intuitively seems like a high value add to the workflow. the datastructure of the datalioader made id difficult to compare the final input values for the model, because they only save the actual calues and some factors for normalizing, thus computing the actual inputs only during runtime. for more controll over our data we decided to do the needed scaling with scikit standardscaler, as a preprocessing step, reusing the code provided with the TFT paper.

5.1 Usability

arima: Working with a statistical model 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 invest more time to explore expecially bad performing models. For those there are many options to explore for iomproving performance. This again, leading to the big drawback of many overfitted models to be monitored and reimplement once performance decreased. A very positive experience was working with the AutoARIMA model. Giving the user many options on how to layout the model, but also to let the model find optimal parameters over a predefined search space.

neuralprophet: At the moment, the usability of the NeuralProphet module is the decisive drawback. Presently it is in a major development cycle, turning the module structure, taken from the Prophet module, 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 of data. Adding the need for a dataloader, parallelization and more sophisticated ways of saving the model. Some workflows and workarounds [7] for Prophet are not yet possible with NeuralProphet. 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. Where other models just need the electricity dataset and the saved model, with a combined size of 220MB, the fitted NeuralProphet model takes up around 14GB. However, after the integration with Foreacsting and Lightning is done, those drawbacks should be eliminated and usability should be very similar to the Forecasting implementation of the TFT. for interpreting the predictions, the trained model is still needed.

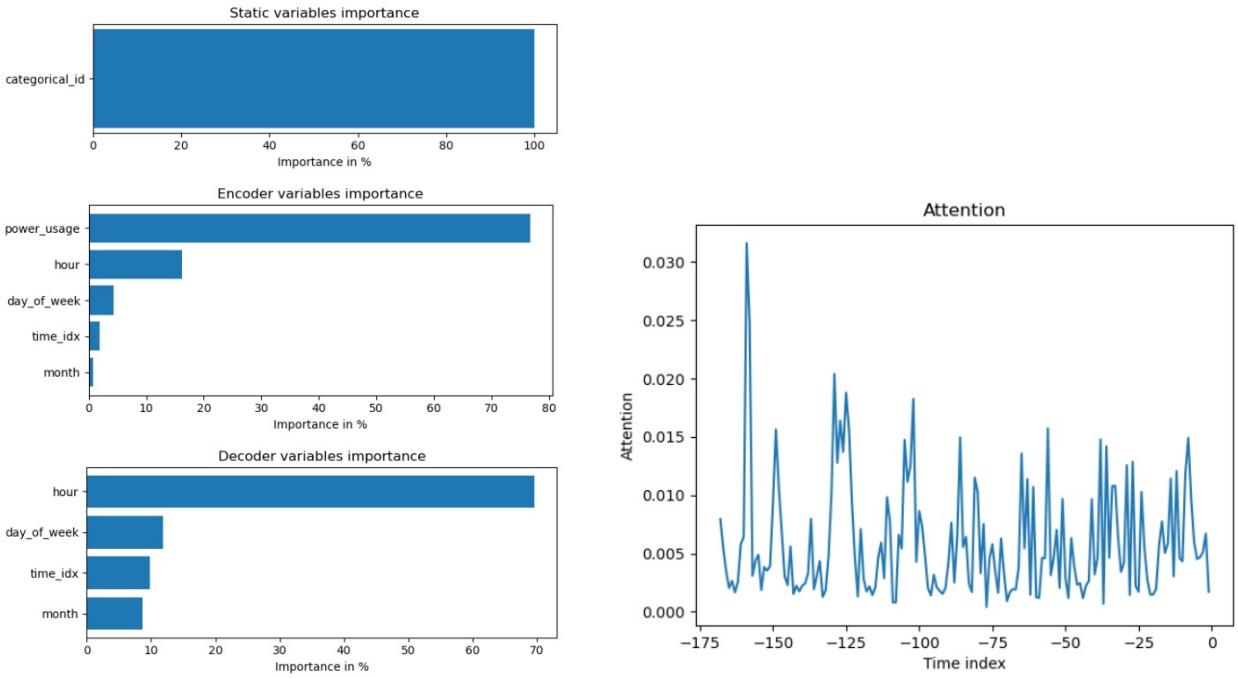
TFT: pytorch forecasting beeing build on lightning gives the module a very clear structure. lightnings documentation makes handling the training easier. modles are actually portable with model size of maximum of 6MB when keeping all folder structures etc. the same, one can just load the repository and reuse fitted models without a problem, which was not possible with neuralprophet

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 were 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 the Figure 2, TFT provides a plethora of visuals to achieve a better explainability. First of all, the variable importance, 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. Categorical ID has a quite high importance percentage, as can be seen in the Figure 2. In addition, the variables "power usage" and "hour" are the front runners for encoder variable importance and decoder variable importance respectively. Apart from that, how the attention mechanism varies, which refers to the structure that makes it possible for the model to selectively concentrate on different parts of the input data when carrying out predictions, is again aptly described via the Figure 2.

Neuralprophet: NeuralProphet offers many ways for plotting predictions, different seasonalities, trend, lagged regressors and uncertainty. Modular composability has shown to be clearly relevant for interpretability purposes. Making it very easy to get a better understanding of the underlying data and forecast components. Thus helping the user to make more



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

Figure 2: TFT

informed decisions about how to use the model, set parameters and maybe find irregularities in the data.

From Figure 3, which depicts the one-week forecast for some random id for the NeuralProphet, electricity consumption appears to have weekly and daily seasonality.

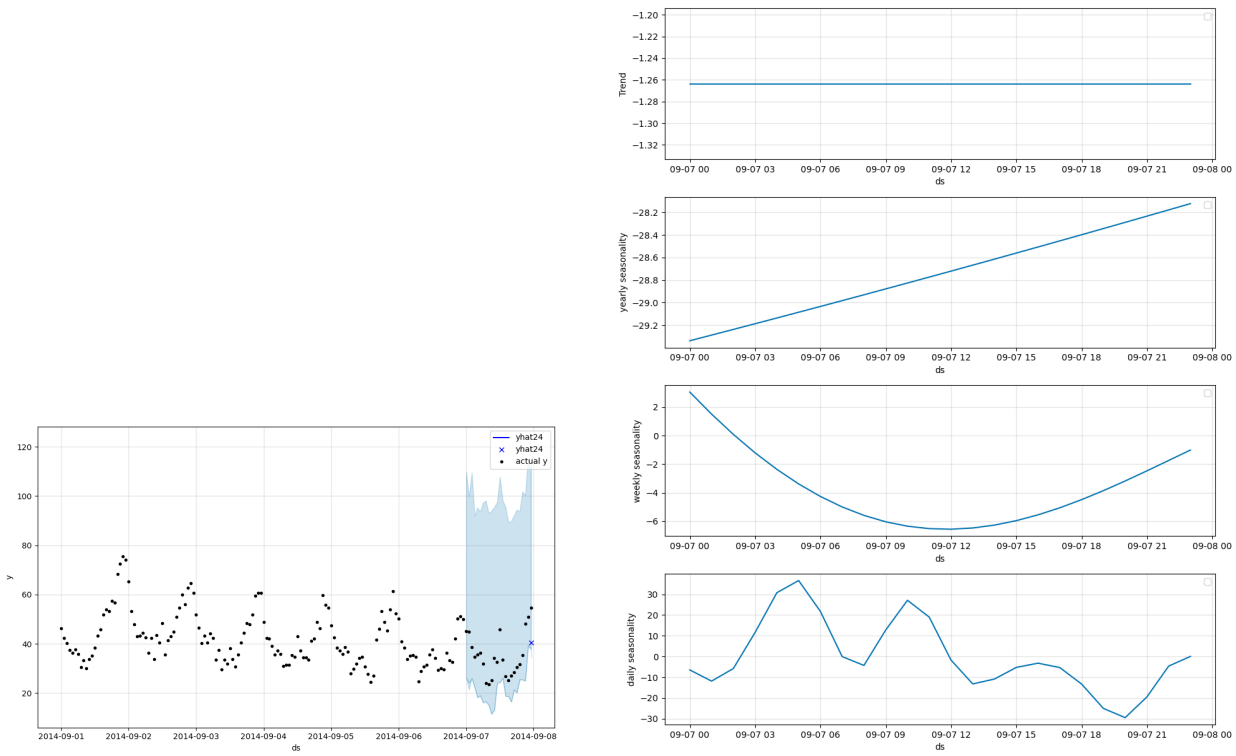
5.3 Performance

neuralprophet: In our experience, the normalized losses are usually lower than the MAE and especially a 10-fold increase compared to other baselines in the TFT paper lead to the assumption, that there is a mistake in the implementation. Handling the predictions dataset is already so impractival, that the mistake could have happened here.

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

	ARIMA	**NP**	**TFT**
MAE	ARIMA	NP	0.334
P50	ARIMA	NP	0.116
P90	ARIMA	NP	0.046

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



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

Figure 3: neural prophet

6 Conclusions/ Discussion

Here you briefly summarize your findings.

(NP at a difficult crossroad: switching to pytorchlightning + forecasting integration + API not all NP components are yet optimized to run on GPU, thus does not make too much sense to run on gpu already. no real documentation available for newer implementation documentation completly without lightning or forecasting hints some integration with forecasting lightning can be seen when running code, e.g. already using the dataloader)

References

[1] Lim, Bryan and Arik, Sercan O. and Loeff, Nicolas and Pfister, Tomas, *Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting*, (arXiv, 2019), https://doi.org/10.48550/arXiv.1912.09363.

[2] Triebe, Oskar and Hewamalage, Hansika and Pilyugina, Polina and Laptev, Nikolay and Bergmeir, Christoph and Rajagopal, Ram, *NeuralProphet: Explainable Forecasting at Scale*, (arXiv, 2021), https://arxiv.org/abs/2111.15397

[3] S. Li, et al., *Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting*, in: NeurIPS, 2019.

[4] Box, G. E. P., and G. C. Tiao. 1975. *Intervention analysis with applications to economic and environmental problems*. Journal of the American Statistical Association 70: 70-79.

[5] D. Salinas, V. Flunkert, J. Gasthaus, T. Januschowski, *DeepAR: Probabilistic forecasting with autoregressive recurrent networks*, International Journal of Forecasting, Volume 36, Issue 3, 2020, Pages 1181-1191, ISSN 0169-2070, https://doi.org/10.1016/j.ijforecast.2019.07.001.

[6] https://optuna.org/

[7] https://github.com/facebook/prophet/issues/1159

[8] Voskoboinikov, Alexey and Pilyugina, Polina, 2021, *NeuralProphet*, https://github.com/neuralprophet/neural$_p$*rophet*$_l$*ightning/blob/master/reports/Third*$_S$*tatus*$_R$*eport*

[9] Asteriou, Dimitrios and Hall, Stephen, *ARIMA Models and the Box-Jenkins Methodology*, (2016), doi0.1057/978-1-137-41547-9$_1$3.