NEURAL PROPHET

A PREPRINT

Alexey Voskoboinikov
Skolkovo Institute of Science and Technology
Moscow, Russia
dblokv@gmail.com

Polina Pilyugina
Skolkovo Institute of Science and Technology
Moscow, Russia
polina.pilyugina@skoltech.ru

May 22, 2021

ABSTRACT

This project aims to contribute to the open-source library NeuralProphet by adding state-of-the-art models and refactoring the code to adopt the best machine learning engineering practices.

***K*eywords** Time series forecasting · Neural Prophet · Neural forecasting

1 Problem Statement

NeuralProphet is a new library for time series forecasting built on PyTorch. It is inspired by the widely known Facebook library for time series forecasting called Prophet ([7]) and DeepAR model ([6]). However, while Prophet is an additive model focused chiefly on seasonal components and holiday effects, NeuralProphet additionally includes AutoRegression components. Moreover, NeuralProphet is built on PyTorch, which allows configuring the model more precisely.

This project aims to improve the existing NeuralProphet library to allow even more possibilities for its users. Currently, the forecasting model is written on pure PyTorch. It has a complicated structure, code is hardly reusable, making experiments with implementation difficult for outside users. We aim to use the PyTorch Lightning framework to structure the code in a more concise way for future research. Moreover, the current implementation of NeuralProphet does not support distributed training, while PyTorch Lightning allows the introduction of the distributed running of the models. Another problem of the current NeuralProphet implementation is that it has a rather specific API regarding initial data format and outputs. NeuralProphet has specific requirements for the model's inputs and its preprocessing procedure, which is not the same as other models. Moreover, it has several specific modules, which are not implemented in other models out-of-the-box, as explained in 5.1. All of this makes comparison with other models complicated, as users are required to write additional code to produce comparable results. We aim to introduce state-of-the-art models following the existing API and model output structure. In particular, we aim to add wrappers for inputs that will allow all the models to use the same initial inputs as NeuralProphet. Further, we will rewrite Pytorch Lightning model steps to support the NeuralProphet metrics calculation procedure to ensure compatibility.

2 Description of the project

2.1 Main goals

Taking into consideration existing drawbacks, we outlined the main goals of our project as follows:

- Refactor the main model class from NeuralProphet with PyTorch Lightning
- Refactor the rest of the code to support PyTorch Lightning in accordance with existing API
- Adapt and include existing implementations of state-of-the-art models for time series forecasting under the NeuralProphet API
- Add hyperparameter tuning with Ray Tune as additional module to NeuralProphet

- Recreate LIBRA framework for benchmarking in Python and run it on NeuralProphet and our additionally included models
- Add neccessary tests and documentation for introduced functional

2.2 Existing solutions

First part of the project is to structure existing code in accordance with existing PyTorch Lightning framework ([2]). PyTorch lightning is a lightweight PyTorch wrapper that allows to organise the code into separate PyTorch Lighting modules. This framework provides multiple advantages compared to usual PyTorch: models become hardware agnostic and structured, it provides integration with popular machine learning tools, while keeping flexibility of original PyTorch. Among such tools is Ray Tune for hyperparameter tuning.

PyTorch lightning also provides a robust architecture that will help us to implement four state-of-the art models for time series forecasting: N-Beats ([5]), LSTM ([3]), Temporal Fusion Transformers ([4]) and DeepAR ([6]). This will allow users to compare NeuralProphet with these reference models. We rely on existing implementations, available in PyTorch Forecasting library[1].

Another big part of the project is the implementation of additional functionality to the NeuralProphet model. It will include hyperparameter optimization and modules for evaluation. We will use Ray Tune library for hyperparameter optimization, as it has hooks to support PyTorch Lightning. Ray Tune has functional for fast distributed hyperparameter tuning, which allows for additional parallelization and scalability. Moreover, PyTorch Lightning has hooks to Ray Tune, which allow for a seamless connection between the model and Ray Tune. For benchmarking, we will rely on the work on LIBRA framework, described in [1]. LIBRA is an evaluation framework that evaluates and ranks forecasting methods based on their performance.

All these changes to the NeuralProphet should be accompanied by extensive testing and documentation. NeuralProphet already has modules with unit and integration tests for tests and integration, and we will extend them to cover all of the refactored and new code. As for documentation, we will add necessary documentation for all new functions provided. This will also require adding corresponding pages into the doc of NeuralProphet and adding minimal examples as notebooks to the GitHub repository.

3 Main Challenges

3.1 API, architecture and style

One of the main challenges in this project is to maintain NeuralProphet style and API. We should structure the code in a reusable way, so it would be easy for ourselves and other people to contribute to the project in the future. To be more precise, it has inherited from Prophet structure of inputs and outputs. It makes NeuralProphet easily comparable with Prophet, but not with other models. Refactoring in accordance to PyTorch Lightning should not affect existing library consumers. Therefore we will need to introduce additional modules to the TimeNet model used in NeuralProphet. It possesses some challenges, connected to our goal to maintain all existing functionality in place.

And we aim to introduce new models in NeuralProphet in accordance with its existing API to preserve usability. This will require us to understand in detail the explicit structure of inputs and outputs to each model and reformat them accordingly. Additionally, we aim to create models in the same structure, to allow seamless introduction of Ray Tune hyperparameter optimization to all the models.

Existing PyTorch implementations of state-of-the-art models are also available in PyTorch Lightning framework, in PyTorch Forecasting library. However, they do not support NeuralProphet API out of the box, so we will need to refactor them in accordance. In particular, we aim to add metrics calculation and reporting the same way as it is done in NeuralProphet. Also we need to make similar methods and output reporting, in order to allow easy comparison of the results. Additionally, we will refactor existing modules of NeuralProphet to be used in new models.

3.2 Adapting LIBRA

The other challenge - implement LIBRA framework [1] for our purposes. It's implementation is available[2] only in R, so we will have to adapt it in Python as a part of the NeuralProphet. Further, we will run the benchmarking framework on

[1] `https://pytorch-forecasting.readthedocs.io/en/latest/models.html`

[2] `https://github.com/DescartesResearch/ForecastBenchmark`

400 time series from [1], which will require a lot of training time. In order to perform such benchmarking, we will need to test our pipeline and its scalability.

4 The roles for the participants

We distributed our main tasks and goals evenly, as described on the 1. Both of us were working on refactoring into PyTorch Lightning. Alexey focused on the main TimeNet model class, while Polina worked on the forecaster code. We also distributed addition of models, such that Polina implemented on N-Beats and LSTM, while Alexey implemented Temporal Fusion Transformers and DeepAR. We both have writen corresponding tests and documentation of implemented modules. Further, Polina focused on hyperparameter tuning addition, while Alexey implemented LIBRA framework in python. Afterwards, we both worked on the benchmarking using LIBRA framework and finalization of the project.

At a current stage, we have finished all the aims that were set for the project. The only bottleneck that we encountered was the computational intensity of benchmarking, therefore we completed benchmarking only on sample from all time series available.

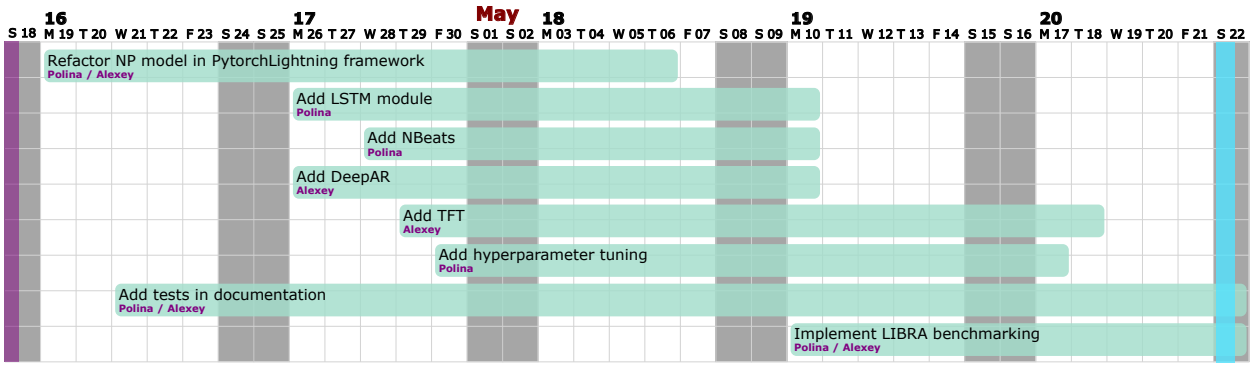

Figure 1: Roadmap of the project with Role Distribution

5 Project structure

5.1 Pytorch Lightning introduction to NeuralProphet

We finished refactoring the NeuralProphet code in the Pytorch Lightning framework. In particular, we implemented all required modules in TimeNet, which is the base model of NeuralProphet. Additionally, we added necessary hooks to connect the model with all the parts of NeuralProphet. Specifically, we preserved the metrics reporting structure and usage of specific optimizers and schedulers provided by the existing NeuralProphet model. NeuralProphet has its specific class for metric calculation called Metrics. It consists of several metrics such as MSE and MAE. This class is supplied with predictions and actual values for each batch on each epoch. On the epoch end, the metrics is computed. In order to introduce this to models, we changed several modules in Pytorch Lightning implementations: training step, validation step, training epoch end, validation epoch end. Additionally, we implemented a module to connect with the hyperparameter optimization functionality we added further. It is for internal use in the hyperparameter optimization module. It initializes the model and all necessary loaders with provided parameters and outputs it without fitting.

The existing library consists of several modules, from which the TimeNet module and forecaster module are the main ones. TimeNet module contains the model itself. We rewrote the model in Pytorch Lightning: we introduced additional required modules, which Pytorch Lightning requires, as well as we introduced necessary hooks for the model to preserve the structure of the forecaster and its modules. Moreover, as was suggested by our colleagues during peer review, we refactored the structure of the code to make it more modular, as will be described in section 7. The forecaster module is the main module used by users, with additional preprocessing and initialization functional. It contains the NeuralProphet class, which users use.

The main methods of NeuralProphet class are:

- init — initializes the class and all the parameters. In case of NeuralProphet, it also initializes metrics structure, which is used further.
- fit — fits the model on the dataset provided. This step consists of preprocessing the data, initializing data loaders, training the model and outputing the training and validation metrics, defined in init module.
- make_future_dataframe — creates a dataframe on which the prediction is done. It contains all the necessary inputs, which the model will require. It support supporting data for prediction on unseen future data, as well as historic prediction.
- predict — predicts the outputs, based on the data, provided in the make_future_dataframe
- plot — plots forecasts in the unified style

On figure 2 is an example of using a NeuralProphet model code structure

```
m = NeuralProphet(
    n_lags=10,
    n_forecasts=3,
    changepoints_range=0.95,
    n_changepoints=30,
    weekly_seasonality=False,
    batch_size=64,
    epochs=10,
    learning_rate=1.0,
)
metrics = m.fit(df, freq='5min')
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
fig = m.plot(forecast)
```

Figure 2: Example code structure with NeuralProphet model

There are additional methods, but the main changes we made were concerned with these modules. Originally, NeuralProphet included two functions for training_epoch and evaluation_epoch, which we removed and added through Pytorch Lightning into TimeNet. We also moved the main training loop into the Pytorch Lightning Trainer function. Further, it will allow to add new callbacks like EarlyStopping to NeuralProphet. Moreover, we added possibility to train on GPUs through Pytorch Lightning Trainer, which was previously not supported in NeuralProphet at all.

5.2 State-of-the-art models

We have added LSTM, NBeats, TemporalFusionTransformer and DeepAR models in accordance with NeuralProphet API.

LSTM For LSTM, we refactored existing PyTorch implementation in the Pytorch lightning framework. We changed training and validation steps to use the NeuralProphet Metrics class, as in the case of NeuralProphet refactoring. This change ensures that the LSTM model will provide metrics calculated on each epoch the same way NeuralProphet does. We used the existing NeuralProphet forecaster module for the model class and adjusted it to be used with this particular LSTM model. In this case, we made the connection as seamless as possible by changing all NeuralProphet modules to support the new model. Additionally, we added hooks for hyperparameter optimization of LSTM as well. On figure 3 is an example of using a LSTM model code structure

NBeats For NBeats we used Pytorch Forecasting NBeats model as a base. We refactored training and validation steps to support the same metrics reporting class as in NeuralProphet. For this we introduced wrappers of predictions, in order for them to be supported by all the metrics inside NeuralProphet metrics class. We implemented additional data preprocessing in order for the model to have the same input as the NeuralProphet. Models from Pytorch Forecasting library use specific TimeSeriesDataset class, so we needed to add wrappers to process NeuralProphet inputs into this class. We used similar initial preprocessing tools for imputing data as are used in NeuralProphet, to induce compatability. Additionally, we reformated the input dataframe in accordance with TimeSeriesDataset class requirements. For this we reformulated the TimeSeriesDataset parameters in accordance with NeuralProphet parameters. Moreover, the

```
m = LSTM(
    n_lags=10,
    n_forecasts=3,
    num_hidden_layers=1,
    d_hidden=64,
    learning_rate=0.1,
    epochs=10,
    batch_size=None,
    loss_func="Huber",
    optimizer="AdamW",
    train_speed=None,
    normalize="auto",
    impute_missing=True,
    lstm_bias=True,
    lstm_bidirectional=False,
)
metrics = m.fit(df, freq="5min")
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
fig = m.plot(forecast)
```

Figure 3: Example code structure with LSTM model

specificity of output processed with Pytorch Forecasting models and its prediction functions required us to completely rewrite main existing modules of NeuralProphet forecaster class, in order to support new model. In particular, by default predictions are stored as dictionaries of specific sort. Therefore in order to calculate metrics for each epoch, we preprocessed the network outputs to be compatible with NeuralProphet Metrics class. Moreover, the prediction function by itself outputs raw results, which we needed to further process and wrap into the same dataset structure, as in case of NeuralProphet. Therefore, we basically introduced a new forecaster holder for the NBeats model, which has the same structure of modules and produces the same outputs as NeuralProphet. It also follows the same logic, as in NeuralProphet: it contains initialization of the model, creation of dataset modules. We also refactored existing modules for data preprocessing and parameter selection, in order to work with new models and their parameters. Additionally, Pytorch Forecasting allows to define hyperparameters from the dataset, and we added this functional as well, by introducing a hyperparameter from_dataset. If this hyperparameter is set to True, the model parameters will be set automatically, which is particulary useful for new users of NBeats. However, we reserved the possibility to change main NBeats parameters manually, based on the users' desire. On figure 4 is an example of using a NBeats model code structure

```
m = NBeats(
    n_lags=12,
    n_forecasts=3,
    batch_size=None,
    epochs=100,
    num_gpus=0,
    patience_early_stopping=10,
    early_stop=True,
    weight_decay=1e-2,
    learning_rate=3e-2,
    auto_lr_find=False,
    num_workers=3,
)

metrics = m.fit(df, freq = freq)
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
```

Figure 4: Example code structure with NBeats model

DeepAR DeepAR is a probabilistic forecasting model, and Pytorch Forecasting provides a good base implementation. This model uses Normal Distribution Loss (NDL) as a default loss function, and data has to be normalised accordingly (NDL does not work with all-positive target values). As probabilistic model, for NDL it predicts 2 values - "loc" and "scale", and training network with this loss function also decreases other metrics, as shown in the example notebook. We override basic Pytorch Lightning methods (test/validation steps and methods for epoch end), and then wrapped this model to make it work with NeuralProphet datasets and produce metrics and predictions in the same way.

```
m = DeepAR(
        n_lags=32,
        n_forecasts=10,
        batch_size = 32,
        epochs = 10,
        num_gpus = 0,
        patience_early_stopping = 10,
        early_stop = True,
        learning_rate=5e-4,
        auto_lr_find=True,
        num_workers=8,
        hidden_size=10,
        rnn_layers=2,
        dropout=0.1,
)

metrics = m.fit(df, freq = freq)
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
```

Figure 5: Example code structure with DeepAR model

TemporalFusionTransformer (TFT) Temporal Fusion Transformer (TFT) is a novel architecture, which combines recurrent layers for local processing with self-attention layers for long-term dependencies. We used it's implementation in Pytorch Forecasting, wrapping the dataset into TimeSeriesDataset with proper scaling and normalisation of the target. It's baseline loss function is Quantile Loss, as it usually use for multihorizon forecast. As with other models, we used the same technique - override basic functions that are responsible for interacting with pytorch lightning, and add functional that mimic public methods of NeuralProphet - fit, make_future_dataframe, predict, and plot. TFT has multiple hyperparameters that determine model - number of hidden layers, number of attention heads etc., so best results can be achieved only after extensive hyperparameter tuning.

```
m = TFT(
    n_lags=32,
    n_forecasts=10,
    epochs=10,
    learning_rate=0.03,
    hidden_size=16,
    attention_head_size=1,
    dropout=0.1,
    hidden_continuous_size=8,
)

metrics = m.fit(df, freq = freq)
future = m.make_future_dataframe(df, n_historic_predictions=True)
forecast = m.predict(future)
```

Figure 6: Example code structure with TFT model

5.3 Results

In general, all the models we introduced have the same main modules, as NeuralProphet, and output the results in the same format. This allows for a fast comparison of training and validation metrics, as well as plotting of the results with existing NeuralProphet functional. In the future, it is possible to refactor it even further and make implementation easier to read and maintain. We created example notebooks for each of the model, in order to provide users with an introduction to how these models work.

On figure 7 we provide with example predictions plotting for three models: NeuralProphet, LSTM, TFT, NBeats and DeepAR, on a sample Yosemite dataset. These baseline results are contained in example notebooks of each model and can be easily rerunable, to check different hyperparameter combinations, for example.

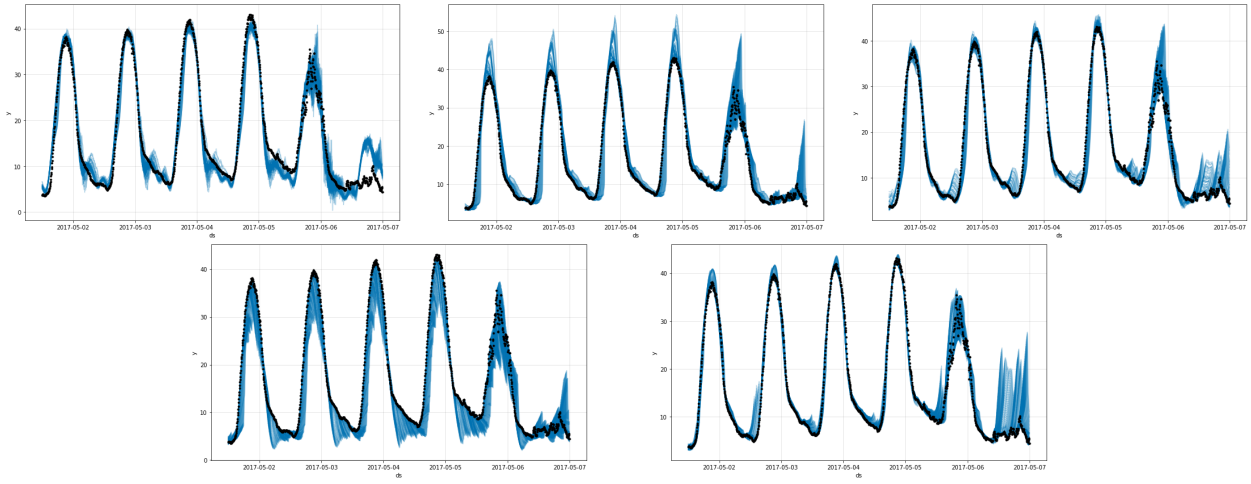

Figure 7: Example of predictions plotting for models (from left to right): NeuralProphet, NBeats, LSTM, DeepAR and TFT at the bottom

5.4 Hyperparameter optimization

As for hyperparameter tuning, we created a separate module in NeuralProphet library for this sake. We added hooks to each of the models we implemented and NeuralProphet itself, using the functionality provided by Pytorch Lightning, to fastly connect the optimization module with the main model modules. Moreover, our hyperparameter optimization works not only with NeuralProphet, but for all the models we implemented. Currently, the function works in two modes: auto and manual. The automatic parameter selection uses predefined by us sets of hyperparameters to tune over. The manual mode requires user to provide their own configuration, in accordance with tune API. We have added a notebook example of how this can be done, so the users will have an idea on how to use Ray Tune functionality as well. This function is called tune_hyperparameters and it out-of-the-box requires only three parameters: the model name, the dataframe with data, and frequency of dates in data. Its default set up can be useful for fast basic tuning and also for new users, which have no experience with the model. We evaluated the tuning progress on the Yosemite dataset, included in NeuralProphet. We compared the default one-step-ahead forecasting initialization of NeuralProphet with its tuned version, and it showed significant improvement. Therefore, the proposed functional can be exceptionally useful for new users, as it includes automatic mode which requires no initial understanding of the model parameters and provides reasonable improvement out-of-the-box. On figure 8 we provide a resulting metrics of two NP models: with default parameters, and with tuned parameters

SmoothL1Loss	0.005514	SmoothL1Loss	0.000469
MAE	3.886362	MAE	1.086094
MSE	29.934882	MSE	3.560099
RegLoss	0.000000	RegLoss	0.000108
SmoothL1Loss_val	0.170988	SmoothL1Loss_val	0.001177
MAE_val	27.786178	MAE_val	1.659725
MSE_val	928.262373	MSE_val	8.928988

Figure 8: Left is metrics of default model, and right is metrics on tuned model

We introduced the same functional for all the models we introduced. It works in the same way, as with NeuralProphet. We added an example in the notebook for LSTM model as well.

Ray Tune additionally provides a functionality to distribute resources among trials of experiments on both CPUs and GPUs, which allows for fast training. We used AsyncHyperBandScheduler for hyperparameter tuning. This scheduler is an improved version of HyperBand scheduler. The idea behind it, is that it early stops low-performing trials using the HyperBand optimization algorithm. This allows not to waste computational resources on low-performing hyperparameter settings, which also induces faster optimization.

5.5 Tests and documentation

We have provided each introduced class and function with detailed description of each model parameters. We also have added descriptions on how to use all main functions in README.MD. As for the tests, we have added integration tests for each of the functional we provide, in the same manner, as in original NeuralProphet. By default, they are run automatically when pushing the changes to GitHub repository, if developers version is installed. Or one can run the debug code from tests module, in order to check the current state of the code. More details on how to run these tests, see README.MD in the GitHub repository.

5.6 LIBRA evaluation

In order to evaluate our results, we used LIBRA framework. We used the LIBRA dataset containing 400 time series from four different usecases:

- Economics — (gas, sales, unemployment, etc.)
- Finance — (stocks, sales prices, exchange rate, etc.)
- Human access — (calls, SMS, Internet, etc.)
- Nature and demographics — (rain, birth, death, etc.)

Following the LIBRA framework, the benchmarking was implemented for different methodologies: one-step-ahead and multi-step-ahead forecasting. One-step-ahead forecasting constitutes to forecasting one period ahead from the set date. Multi-step-ahead forecasting constitutes to forecasting several steps ahead in the future.

We provide results averaged over ten time series from each domain and each methodology. Overall, the challenge with these datasets was that they contained originally no timestamps, and we needed to review the R implementation to find something resembling actual frequencies. For those datasets, which had some interpretable frequency, we used it. For those without interpretable frequencies, we manually set the frequency to be daily. Moreover, these time series were drastically different and size and while for some of them models ran fast, it was usually not the case. In order to perform the benchmarking we therefore limited number of datasets on which we run the model to a sample of 10 datasets from each domain.

As for the metrics, we have also followed the procedure of original LIBRA framework, and included the following metrics: symmetrical mean absolute percentage error (sMAPE), mean absolute scaled error (MASE), wrong-estimation shares (Mean Under- and Over-Estimation Shares, MUES/MOES) and the mean wrong-accuracy shares (Mean Under- and Over-Accuracy Shares, MUAS/MOAS). Both sMAPE and MASE measures are independent of the scale and can be used across different time series. While MUES/MOES and MUAS/MOAS provide additional inside on whether the model tends to under or over estimate the data. We implemented this errors, so they can also be used in the main NeuralProphet by users. On 1 we provide results of this benchmarking on economics usecase. The best model for multi-step-ahead forecasting is TFT. The best model for one-step-ahead forecasting is NBeats. The results for other usecases can be found in .

In general, performance of NeuralProphet is usually inferior to other models. However, for most other models we used available out-of-the-box in PyTorch Forecasting initialization, which is based on the dataset. And for benchmarking we have only set n_lags, which is fixed in the dataset (as implied frequency); n_forecasts, which is either 1 for one-step-ahead-forecasting or $min(n\_lags, 0.2 * len\_ts)$. The other parameters were learning_rate and number of epochs, which were also the same for all models on all time series. The main difference is that for NBeats, DeepAR and TFT, the implementation uses the default function to infer parameters based on the time series, therefore they were better adjusted than NeuralProphet. Although NeuralProphet has some parameter adjusting, it is still a work in progress, which will likely improve the model performance.

As for the code, we implemented a module to run the LIBRA benchmarking. Example of its usage is available in LIBRA notebook in example notebooks section of our repository. Note, that even small number of time series in

benchmarking is still computationally intense, as it includes deep models. We also provide a notebook to combining results into nice LaTex tables, as can be seen in evaluation results notebook in our repository.

metrics	method	economics_DeepAR	economics_LSTM	economics_NBeats	economics_NP	economics_TFT
mase	multi_step_ahead	**1.95**	26.89	2.73	66.78	55.42
moas	multi_step_ahead	0.07	0.06	0.05	0.64	**0.02**
moes	multi_step_ahead	0.39	0.19	0.32	0.41	**0.09**
muas	multi_step_ahead	**0.05**	0.30	0.06	1.29	0.71
mues	multi_step_ahead	0.61	0.81	0.68	**0.59**	0.91
smape	multi_step_ahead	10.54	46.34	**10.44**	2281.54	131.75
mase	one_step_ahead	2.97	29.45	**2.23**	61.12	11.72
moas	one_step_ahead	0.06	**0.03**	0.06	0.76	0.07
moes	one_step_ahead	0.35	**0.14**	0.33	0.53	0.29
muas	one_step_ahead	**0.06**	0.37	**0.06**	0.54	0.46
mues	one_step_ahead	0.65	0.86	0.67	**0.47**	0.71
smape	one_step_ahead	11.33	57.07	**10.71**	316.01	91.61

Table 1: Results of benchmarking on economics usecase

6 A link to the GitHub repository

For this project we forked the original repository and will contribute to it. It will allows us to create a pull request into the master in the future. GitHub repository is available through `https://github.com/adasegroup/neural_prophet/tree/master`. We have preserved the initial GitHub repository structure. On figure 9 we provide an overview of GitHub repository structure. We outlined the files that were added or changed on the course of this project. This includes: refactoring of existing main modules and structure, documentation, notebooks and tests.

7 Peer review suggestions

We carefully read all peer reviews, and particularly stopped on the review by Cohortney team. We are very grateful to the colleagues, as their suggestions on improving the code structure were very helpful. The team suggested to make the structure modular. Original NeuralProphet, and our re implementation were of the same structure, initially. However, the original code structure of NeuralProphet indeed lacked modularity and it was extremely hard to read the code, not to say about refactoring it and understanding. We also believe, that the original structure was too chaotic, so we took the advice of colleagues and their review and refactored the code structure as well. Currently, we have made separate modules in NeuralProphet with forecasters, models, tools, metrics etc. We also refactored initialization files so that each new model is imported directly from NeuralProphet, and not from some of its modules. There was also a suggestion by the team to move trainer calling on the model from forecaster, however, we wanted to maintain the original API and thus having train in forecaster class was necessary. Overall, this was a great input which, in our opinion, allowed us to harshly improve understandability of NeuralProphet structure.

8 Prospects for further work

During the project, we deeply understood the structure and features of NeuralProphet. And although we did a hard job on its improvement, there are still areas in which future work could be conducted. First and the most important, in our opinion, is further refactoring to induce its performance on GPU, as currently not all the parts of NeuralProphet are GPU-scalable. Secondly, we have seen on the evaluation, that other models are behaving better without hyperparameter tuning. We have added the hyperparameter optimizator as a possibility for NeuralProphet, but in future it will be useful to add more sophisticated initial parameter set up, alike in PyTorch Forecasting models. Another useful insight we encountered, when working with non-time-stamped data from Libra. NeuralProphet is currently not timestamp-agnostic, however it might be exceptionally useful in some cases, where underlying timestamp is not defined or does not make sense. Therefore, we look forward to continue working on it in the future, as we see a potential behind this library.

References

[1] André Bauer et al. “Libra : A Benchmark for Time Series Forecasting Methods Libra : A Benchmark for Time Series Forecasting Methods”. In: April (2021).

[2] WA Falcon and .al. "PyTorch Lightning". In: *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning* 3 (2019).

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. ISSN: 08997667. DOI: 10.1162/neco.1997.9.8.1735.

[4] Bryan Lim et al. "Temporal fusion transformers for interpretable multi-horizon time series forecasting". In: *arXiv* Bryan Lim (2019), pp. 1–27. arXiv: 1912.09363.

[5] Boris N. Oreshkin et al. "N-BEATS: Neural basis expansion analysis for interpretable time series forecasting". In: *arXiv* (2019), pp. 1–31. ISSN: 23318422. arXiv: 1905.10437.

[6] David Salinas et al. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3 (2020), pp. 1181–1191. ISSN: 01692070. DOI: 10.1016/j.ijforecast.2019.07.001. arXiv: 1704.04110. URL: https://doi.org/10.1016/j.ijforecast.2019.07.001.

[7] Sean J. Taylor and Benjamin Letham. "Forecasting at Scale". In: *American Statistician* 72.1 (2018), pp. 37–45. ISSN: 15372731. DOI: 10.1080/00031305.2017.1380080.

9 Appendix

metrics	method	finance_DeepAR	finance_LSTM	finance_NBeats	finance_NP	finance_TFT
mase	multi_step_ahead	3.22	7.05	2.85	30.9	11.44
moas	multi_step_ahead	0.38	0.27	0.35	0.75	0.07
moes	multi_step_ahead	0.53	0.34	0.52	0.38	0.17
muas	multi_step_ahead	0.12	0.27	0.09	1.72	0.56
mues	multi_step_ahead	0.47	0.66	0.48	0.62	0.83
smape	multi_step_ahead	26.43	44.7	24.92	591.34	105.08
mase	one_step_ahead	-	-	-	-	-
moas	one_step_ahead	0.43	0.33	0.38	0.82	0.32
moes	one_step_ahead	0.53	0.36	0.53	0.52	0.13
muas	one_step_ahead	0.06	0.18	0.07	0.39	0.45
mues	one_step_ahead	0.47	0.64	0.47	0.48	0.87
smape	one_step_ahead	24.66	37.48	24.04	539.62	94.61

Table 2: Results of benchmarking on finance usecase

metrics	method	nature_DeepAR	nature_LSTM	nature_NBeats	nature_NP	nature_TFT
mase	multi_step_ahead	1.05	1.67	0.5	5.21	2.81
moas	multi_step_ahead	0.27	0.24	0.07	0.84	0.08
moes	multi_step_ahead	0.46	0.33	0.56	0.54	0.34
muas	multi_step_ahead	0.1	0.17	0.05	0.61	0.39
mues	multi_step_ahead	0.54	0.67	0.44	0.46	0.66
smape	multi_step_ahead	15.11	22.16	6.12	113.25	74.52
mase	one_step_ahead	-	-	-	-	-
moas	one_step_ahead	0.04	0.08	0.03	0.07	0.02
moes	one_step_ahead	0.43	0.39	0.48	0.21	0.27
muas	one_step_ahead	0.04	0.08	0.03	0.58	0.37
mues	one_step_ahead	0.57	0.61	0.52	0.79	0.73
smape	one_step_ahead	7.93	14.94	5.74	180.59	72.83

Table 3: Results of benchmarking on nature usecase

metrics	method	human_DeepAR	human_LSTM	human_NBeats	human_NP	human_TFT
mase	multi_step_ahead	-	-	-	-	-
moas	multi_step_ahead	0.73	0.73	0.28	4.26	0.2
moes	multi_step_ahead	0.57	0.57	0.44	0.57	0.49
muas	multi_step_ahead	0.16	0.23	0.24	1.9	0.22
mues	multi_step_ahead	0.43	0.43	0.56	0.43	0.51
smape	multi_step_ahead	58.49	59.61	51.59	1206.33	34.42
mase	one_step_ahead	-	-	-	-	-
moas	one_step_ahead	0.13	0.32	0.1	0.53	0.1
moes	one_step_ahead	0.55	0.56	0.44	0.43	0.41
muas	one_step_ahead	0.06	0.19	0.03	1.06	0.2
mues	one_step_ahead	0.45	0.44	0.56	0.57	0.59
smape	one_step_ahead	33.72	34.95	17.82	492.24	52.7

Table 4: Results of benchmarking on human usecase

```
└── neural_prophet
    ├── README.md
    ├── docs
    │   ├── images
    │   ├── model
    │   │   ├── additional_models.md
    │   │   ├── hyperparameter_optimization.md
    │   │   └── zh
    ├── example_data
    │   ├── LIBRA
    ├── example_notebooks
    │   ├── DeepAR_example.ipynb
    │   ├── LIBRA.ipynb
    │   ├── LSTM_example.ipynb
    │   ├── NBeats_example.ipynb
    │   ├── TFT_example.ipynb
    │   ├── evaluation_results.ipynb
    │   ├── hyperparameter_optimization_example.ipynb
    ├── neuralprophet
    │   ├── dataset
    │   ├── forecasters
    │   │   ├── forecaster.py
    │   │   ├── forecaster_DeepAR.py
    │   │   ├── forecaster_LSTM.py
    │   │   ├── forecaster_NBeats.py
    │   │   ├── forecaster_NBeats_old.py
    │   │   └── forecaster_TFT.py
    │   ├── hyperparameter_tuner.py
    │   ├── libra.py
    │   ├── models
    │   │   ├── DeepAR.py
    │   │   ├── LSTM.py
    │   │   ├── NBeats.py
    │   │   ├── TFT.py
    │   │   └── time_net.py
    │   ├── tools
    │   │   ├── configure.py
    │   │   ├── hdays.py
    │   │   ├── metrics.py
    │   │   ├── metrics_libra.py
    │   │   ├── plot_forecast.py
    │   │   └── plot_model_parameters.py
    │   └── utils
    │       ├── df_utils.py
    │       ├── utils.py
    │       └── utils_torch.py
    ├── notes
    ├── peer_reviews
    ├── reports
    ├── requirements.txt
    ├── results_benchmarking
    │   └── example
    │       ├── results_libra_multistep_economics.csv
    │       ├── results_libra_multistep_finance.csv
    │       ├── results_libra_multistep_human.csv
    │       ├── results_libra_multistep_nature.csv
    │       ├── results_libra_onestep_economics.csv
    │       ├── results_libra_onestep_finance.csv
    │       ├── results_libra_onestep_human.csv
    │       └── results_libra_onestep_nature.csv
    ├── roadmap_gantt.png
    ├── scripts
    ├── setup.py
    └── tests
        ├── debug.py
        └── test_integration.py
```

Figure 9: Structure of GitHub repository with outlining the files added or changed during the project, as for the current state