# NeuralProphet project for FDS course

Alexey Voskoboinikov and Polina Pilyugina

Skolkovo Institute of Science and Technology

May 30, 2021

# Overview

# Problem statement

NeuralProphet is a novel library for time series forecasting, inspired by **existing Prophet library** and DeepAR model. Our project aims to improve NeuralProphet library, in order to **make its usage and comparison** with other models **more convenient**.

Main NeuralProphet specificities:

- NeuralProphet is based on **pure PyTorch and lacks structure**, therefore it is hard to experiment with
- NeuralProphet uses similar to Prophet **specific API**, which makes comparison with other models tedious
- NeuralProphet by default **does not provide automatic hyperparameter** tuning

# Possible applications

Applications of NeuralProphet library:

- Prediction of temperature in a particular place (i.e. Yosemite dataset, which is used in examples)
- Forecasting of number of air passengers traveling to a particular location

Who will benefit from our project:

- **Future contributors** to NeuralProphet. For example, PyTorch Lightning has many useful callbacks, which now are easy to add.
- Users aiming to **evaluate various models** on their dataset. With our implementation, no additional preprocessing code is required to run 5 different models, as they rely on the same API.
- **New users**, aiming to start working with NeuralProphet. We provide a hyperparameter tuning module, which will allow new users to obtain a reasonable model with no prior knowledge on hyperparameters

# Project outline

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.

Main results:

- **Refactored existing NeuralProphet**: its structure to **induce modularity**, and main modules to adopt **PyTorch Lightning**
- Added **SOTA models** and induced their **compatibility** in terms of API and output structure with NeuralProphet
- Introduced **hyperparameter tuning** functionality for all models
- Created a **framework for benchmarking** based on LIBRA
- We supported all new functionality with **tests and documentation**

# Refactoring NeuralProphet

We refactored the structure of the NeuralProphet library, to make it modular. Moreover, we refactored the code of the model into PyTorch Lightning.

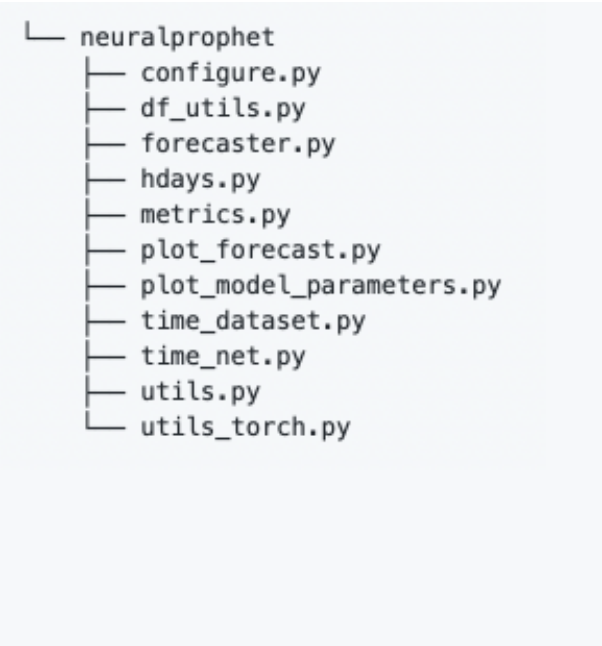


Figure: Old NeuralProphet structure

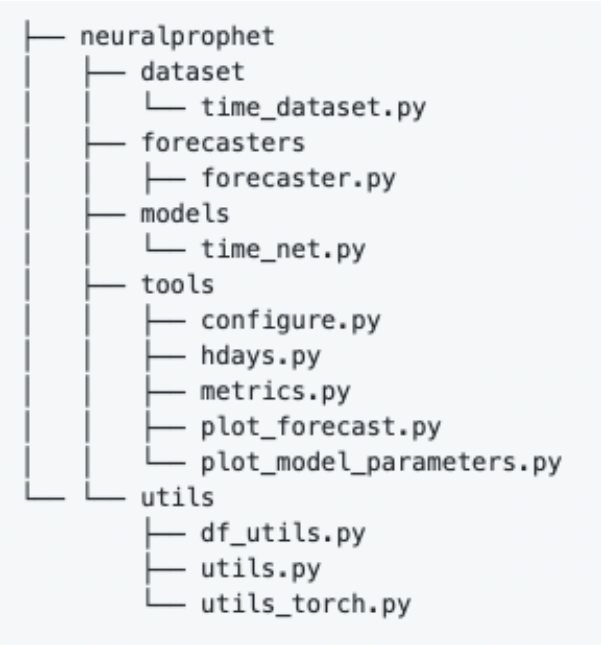


Figure: Refactored NeuralProphet structure

# Refactoring NeuralProphet

Moreover, we refactored the code of the model into PyTorch Lightning. We added necessary modules, to convert the TimeNet class into PyTorch Lightning model.

```
class TimeNet(pl.LightningModule):
    def training_step(self, batch, batch_idx):...
    def validation_step(self, batch, batch_idx):...
    def test_step(self, batch, batch_idx):...
    def training_epoch_end(self, outputs):...
    def validation_epoch_end(self, validation_step_outputs):...
    def configure_optimizers(self):...
```

Figure: Additional modules to TimeNet

# Refactoring NeuralProphet

An example of using the NeuralProphet class is provided on figure 4. The refactored version is completely the same from users perspective, as the previous version. We ran our implementation on the Yosemite dataset, which was used in example notebooks of original NeuralProphet.

```
m = NeuralProphet(
    n_lags=6 * 12,
    n_forecasts=3 * 12,
    changepoints_range=0.95,
    n_changepoints=30,
    weekly_seasonality=False,
    batch_size=64,
    epochs=10,
    learning_rate=1.0,
)
metrics_NP = m.fit(df, freq="5min")
```

Figure: Example of code for NeuralProphet model

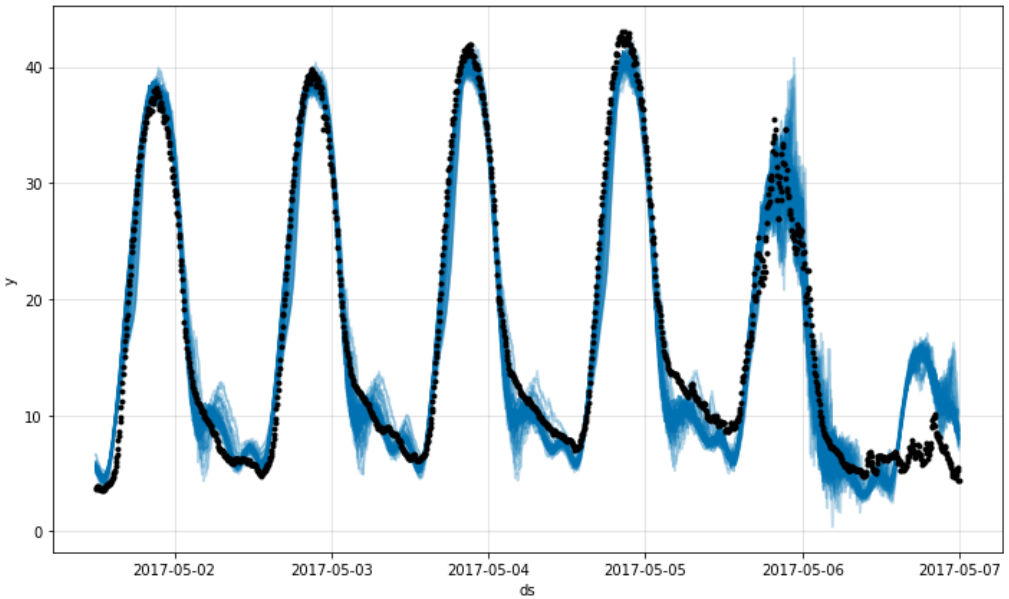


Figure: Results of NeuralProphet on Yosemite dataset

# Overview of models implemented

We have implemented four additional models into NeuralProphet: LSTM, DeepAR, NBeats and TemporalFusionTransformer (TFT). These models are available under the same API and are imported straight from NeuralProphet.

- LSTM: we created implementation, based on PyTorch LSTM refactored into PyTorch Lightning. We replaces the TimeNet from NeuralProphet with this LSTM and changed some modules of existing forecaster to support new model
- NBeats, DeepAR and TFT: we used implementations from PyTorch FOrecasting library, refactored to adapt NeuralProphet API. Additionally, we created specific forecaster classes for these models, that have same modules, as NeuralProphet, but the code is was written from scratch

# LSTM

An example of LSTM usage can be seen on snapshot figure 6. We run the analogous setup on Yosemite dataset and provide a plot with forecasts on figure 7.

```
m = LSTM(
    n_lags=6 * 12,
    n_forecasts=3 * 12,
    num_hidden_layers=1,
    d_hidden=64,
    learning_rate=0.1,
    epochs=10,
    loss_func="Huber",
    optimizer="AdamW",
    train_speed=None,
    normalize="auto",
    impute_missing=True,
    lstm_bias=True,
    lstm_bidirectional=False,
)
metrics_LSTM = m.fit(df, freq="5min")
```

Figure: Example of code for LSTM model

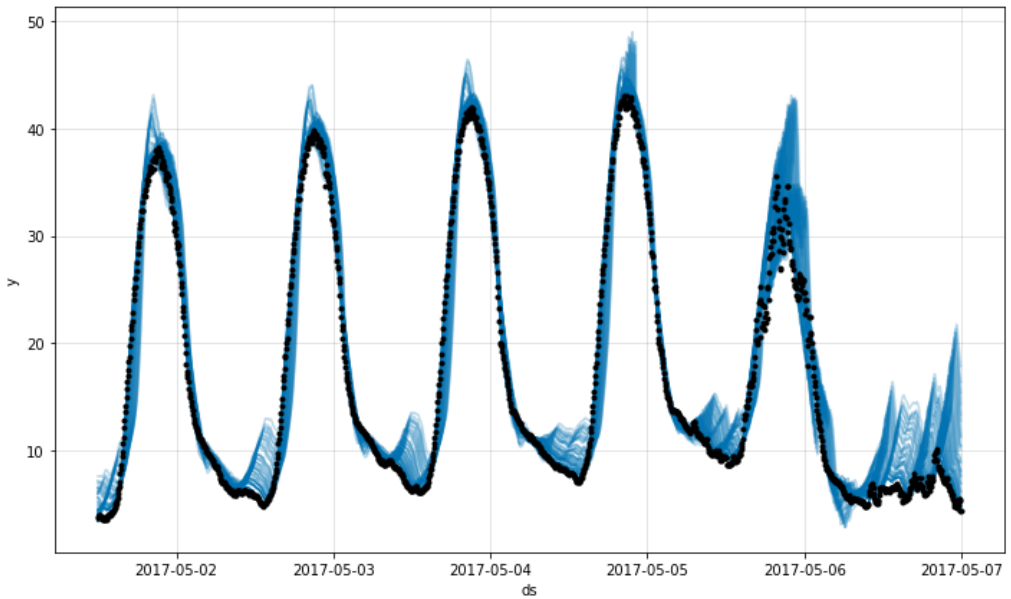


Figure: Results of LSTM on Yosemite dataset

# DeepAR

An example of DeepAR usage can be seen on snapshot figure 12. We run the analogous setup on Yosemite dataset and provide a plot with forecasts on figure 13.

```
m = DeepAR(
    n_lags=6 * 12,
    n_forecasts=3 * 12,
    batch_size=32,
    epochs=20,
    patience_early_stopping=10,
    early_stop=True,
    learning_rate=1e-3,
    auto_lr_find=False,
    hidden_size=10,
    rnn_layers=2,
    dropout=0.1,
)
metrics_DeepAR = m.fit(df, freq="5min")
```

Figure: Example of code for DeepAR model

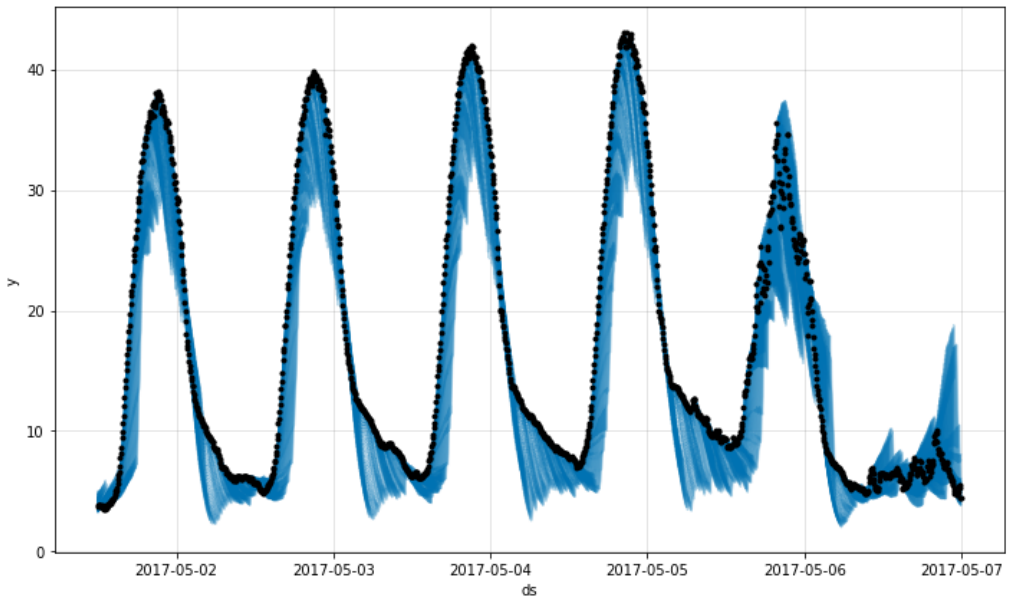


Figure: Results of DeepAR on Yosemite dataset

# NBeats

An example of NBeats usage can be seen on snapshot figure 10. We run the analogous setup on Yosemite dataset and provide a plot with forecasts on figure 11.

```
m = NBeats(
    n_lags=6 * 12,
    n_forecasts=3 * 12,
    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_NBeats = m.fit(df, freq="5min")
```

Figure: Example of code for NBeats model

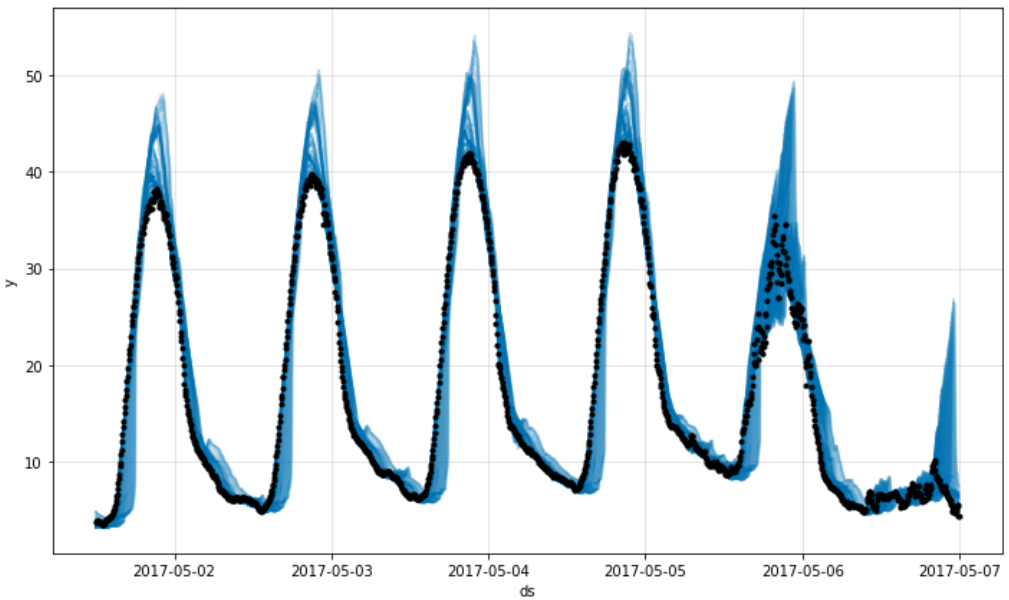


Figure: Results of NBeats on Yosemite dataset

# TFT

An example of TFT usage can be seen on snapshot figure 12. We run the analogous setup on Yosemite dataset and provide a plot with forecasts on figure 13.

```
m = TFT(
    n_lags=6 * 12,
    n_forecasts=3 * 12,
    epochs=12,
    learning_rate=0.03,
    hidden_size=16,
    attention_head_size=1,
    dropout=0.1,
    hidden_continuous_size=8,
    loss_func="QuantileLoss",
)
metrics_TFT = m.fit(df, freq="5min")
```

Figure: Example of code for TFT model

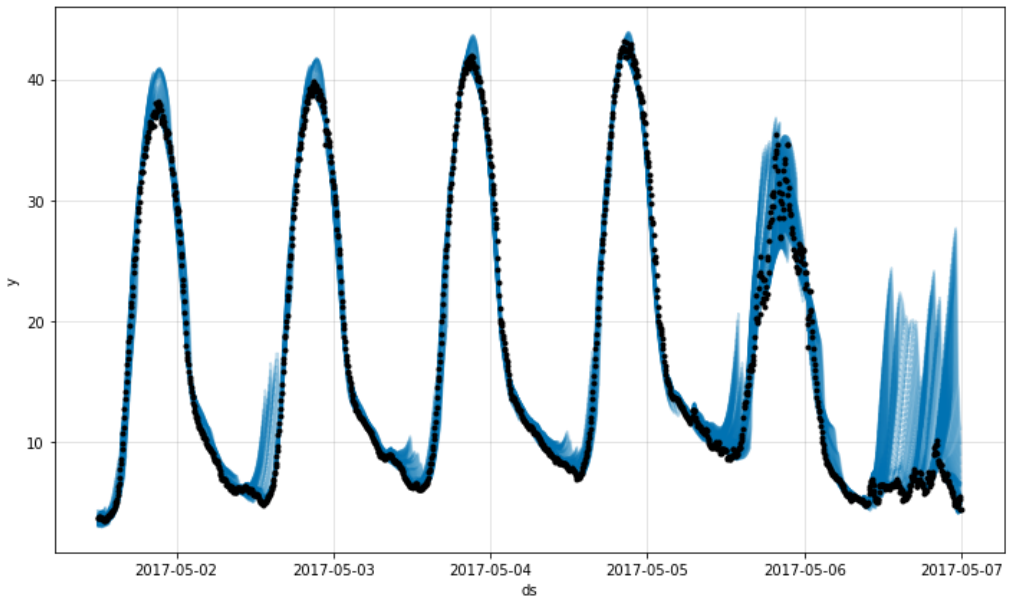


Figure: Results of TFT on Yosemite dataset

# Hyperparameter tuning

- For hyperparameter tuning we used **Ray Tune library**
- We provide a module with **automatic hyperparameter search**, as well as a possibility to use **custom sets of hyperparameters** to tune over
- This will be useful for **both experienced and new users**

```
freq = '5min'
best_params, results_df = tune_hyperparameters('NP',
                                               df,
                                               freq)
```

```
best_params
```

```
{'growth': 'linear',
 'n_changepoints': 10,
 'changepoints_range': 0.5,
 'trend_reg': 10.0,
 'yearly_seasonality': False,
 'weekly_seasonality': True,
 'daily_seasonality': True,
 'seasonality_mode': 'additive',
 'seasonality_reg': 0.0,
 'n_lags': 30,
 'd_hidden': 8,
 'num_hidden_layers': 2,
 'ar_sparsity': 0.8,
 'learning_rate': 0.024150905458487568,
 'loss_func': 'Huber',
 'normalize': 'auto'}
```

Figure: Example of automatic hyperparameter tuning with Ray Tune on NeuralProphet

```
config = {'n_lags': tune.grid_search([10, 20, 30]),
          'learning_rate': tune.loguniform(1e-4, 1e-1),
          'num_hidden_layers': tune.choice([2, 8, 16])}
```

```
freq = '5min'
best_params, results_df = tune_hyperparameters('NP',
                                               df,
                                               freq,
                                               mode = 'manual',
                                               num_samples = 3,
                                              config = config)
```

```
best_params
```

```
{'n_lags': 10, 'learning_rate': 0.07003935175029312, 'num_hidden_layers': 2}
```

Figure: Example of manual hyperparameter tuning with Ray Tune on NeuralProphet

# Example of hyperparameter tuning

On figure 16 we provide a resulting metrics of two NP models: with default parameters, and with tuned parameters. We ran them on Yosemite dataset.

| | | | |
|---|---|---|---|
| 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: Left is metrics of default model, and right is metrics on tuned model

# Libra evaluation

We implemented the code to run evaluation on sets of dataset, following the LIBRA framework.

- Implemented **specific metrics** functions
- **Wrapped the running cycle** and saving results into **one module**
- Created example to show how to **process and output** the results in a table

| 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 |

Figure: Results of LIBRA evaluation on sample from Economics usecase

# Tests

NeuralProphet has **precommit hooks** with integration tests, as well as debugging functional.

We **added integration tests** that cover **all additionally added functional**.

```python
def test_TFT(self):
    log.info("TEST TFT")
    df = pd.read_csv(YOS_FILE, nrows=NROWS)
    m = TFT(
        n_lags=10,
        n_forecasts=3,
        epochs=EPOCHS,
    )
    metrics = m.fit(df, freq="5min")
    future = m.make_future_dataframe(df, periods=12 * 24, n_historic_predictions=12 * 24)
    forecast = m.predict(future)

def test_ray_tune(self):
    log.info("TEST Ray")
    df = pd.read_csv(YOS_FILE, nrows=NROWS)
    freq = '5min'
    from ray import tune

    config = {'n_lags': tune.grid_search([10, 30]),
              'num_hidden_layers': tune.choice([2, 4])}
    best_params, results_df = tune_hyperparameters('NP', df, freq, mode='manual', num_samples=3, config=config)

def test_Libra(self):
    log.info("TEST Libra")
    usecase = 'economics'
    datasets, frequencies = get_datasets(usecase)
    methods = ['onestep', 'multistep']
    metrics = libra(n_datasets=1, datasets=datasets, frequencies=frequencies,
                    method=methods[0], n_epochs=1, usecase=usecase)
```

Figure: Example of integrations tests we added

# Documentation

NeuralProphet has existing documentation .md files, which are automatically compiled once pushed to original library's master. We created **additional pages** to cover our **new functional**. Additionally, we **integrated** links to this pages into **main NeuralProphet docs**.
We also created **example notebooks**, which are possible to be run in **Colab**, with all necessary preinstallation code required.

# Conclusions

We implemented all initially set aims: refactored NeuralProphet, added new models, added hyperparameter tuning and LIBRA benchmarking. We support our implementation with necessary tests and documentation.
Prospects for future work:

- Add new functionality using callbacks from PyTorch Lightning
- Refactor the code even further for full GPU compatability
- Add improved hyperparameter initialization for NeuralProphet

# References


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

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

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

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

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

Triebe, Oskar J., Nikolay Laptev, and Ram Rajagopal (2019). "AR-net: A simple auto-regressive neural network for time-series". In: *arXiv* d, pp. 1–12. ISSN: 23318422. arXiv: 1911.12436.

Salinas, David et al. (2020). "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3, 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.

Bauer, André et al. (2021). "Libra : A Benchmark for Time Series Forecasting Methods Libra : A Benchmark for Time Series Forecasting Methods". In: April.

# The End