In [1]:
```
#| default_exp models.nbeats
```

In [2]:
```
#| hide
%load_ext autoreload
%autoreload 2
```

NBEATS

The Neural Basis Expansion Analysis (`NBEATS`) is an `MLP` -based deep neural architecture with backward and forward residual links. The network has two variants: (1) in its interpretable configuration, `NBEATS` sequentially projects the signal into polynomials and harmonic basis to learn trend and seasonality components; (2) in its generic configuration, it substitutes the polynomial and harmonic basis for identity basis and larger network's depth. The Neural Basis Expansion Analysis with Exogenous (`NBEATSx`), incorporates projections to exogenous temporal variables available at the time of the prediction.

This method proved state-of-the-art performance on the M3, M4, and Tourism Competition datasets, improving accuracy by 3% over the `ESRNN` M4 competition winner.

References

-Boris N. Oreshkin, Dmitri Carpov, Nicolas Chapados, Yoshua Bengio (2019). "N-BEATS: Neural basis expansion analysis for interpretable time series forecasting".

Figure 1. Neural Basis Expansion Analysis.

In [3]:
```
#| export
from typing import Tuple, Optional

import numpy as np
import torch
import torch.nn as nn

from neuralforecast.losses.pytorch import MAE
from neuralforecast.common._base_windows import BaseWindows
```

In [4]:
```
#| hide
from fastcore.test import test_eq
from nbdev.showdoc import show_doc
from neuralforecast.utils import generate_series

import matplotlib.pyplot as plt
```

In [5]:
```
#| exporti
class IdentityBasis(nn.Module):
    def __init__(self, backcast_size: int, forecast_size: int,
```

```
                 out_features: int=1):
        super().__init__()
        self.out_features = out_features
        self.forecast_size = forecast_size
        self.backcast_size = backcast_size

    def forward(self, theta: torch.Tensor) -> Tuple[torch.Tensor, torch.Tens
        backcast = theta[:, :self.backcast_size]
        forecast = theta[:, self.backcast_size:]
        forecast = forecast.reshape(len(forecast), -1, self.out_features)
        return backcast, forecast

class TrendBasis(nn.Module):
    def __init__(self, degree_of_polynomial: int,
                 backcast_size: int, forecast_size: int,
                 out_features: int=1):
        super().__init__()
        self.out_features = out_features
        polynomial_size = degree_of_polynomial + 1
        self.backcast_basis = nn.Parameter(
            torch.tensor(np.concatenate([np.power(np.arange(backcast_size, d
                                    for i in range(polynomial_size)]), dtype
        self.forecast_basis = nn.Parameter(
            torch.tensor(np.concatenate([np.power(np.arange(forecast_size, d
                                    for i in range(polynomial_size)]), dtype

    def forward(self, theta: torch.Tensor) -> Tuple[torch.Tensor, torch.Tens
        polynomial_size = self.forecast_basis.shape[0] # [polynomial_size, L
        backcast_theta = theta[:, :polynomial_size]
        forecast_theta = theta[:, polynomial_size:]
        forecast_theta = forecast_theta.reshape(len(forecast_theta),polynomi
        backcast = torch.einsum('bp,pt->bt', backcast_theta, self.backcast_b
        forecast = torch.einsum('bpq,pt->btq', forecast_theta, self.forecast
        return backcast, forecast

class SeasonalityBasis(nn.Module):
    def __init__(self, harmonics: int,
                 backcast_size: int, forecast_size: int,
                 out_features: int=1):
        super().__init__()
        self.out_features = out_features
        frequency = np.append(np.zeros(1, dtype=float),
                                        np.arange(harmonics, harmonics / 2 *
                                                    dtype=float) / harmonics
        backcast_grid = -2 * np.pi * (
                np.arange(backcast_size, dtype=float)[:, None] / forecast_si
        forecast_grid = 2 * np.pi * (
                np.arange(forecast_size, dtype=float)[:, None] / forecast_si

        backcast_cos_template = torch.tensor(np.transpose(np.cos(backcast_gr
        backcast_sin_template = torch.tensor(np.transpose(np.sin(backcast_gr
        backcast_template = torch.cat([backcast_cos_template, backcast_sin_t

        forecast_cos_template = torch.tensor(np.transpose(np.cos(forecast_gr
        forecast_sin_template = torch.tensor(np.transpose(np.sin(forecast_gr
        forecast_template = torch.cat([forecast_cos_template, forecast_sin_t
```

```
        self.backcast_basis = nn.Parameter(backcast_template, requires_grad=
        self.forecast_basis = nn.Parameter(forecast_template, requires_grad=

    def forward(self, theta: torch.Tensor) -> Tuple[torch.Tensor, torch.Tens
        harmonic_size = self.forecast_basis.shape[0] # [harmonic_size, L+H]
        backcast_theta = theta[:, :harmonic_size]
        forecast_theta = theta[:, harmonic_size:]
        forecast_theta = forecast_theta.reshape(len(forecast_theta),harmonic
        backcast = torch.einsum('bp,pt->bt', backcast_theta, self.backcast_b
        forecast = torch.einsum('bpq,pt->btq', forecast_theta, self.forecast
        return backcast, forecast
```

In [6]:
```
#| exporti
ACTIVATIONS = ['ReLU',
               'Softplus',
               'Tanh',
               'SELU',
               'LeakyReLU',
               'PReLU',
               'Sigmoid']

class NBEATSBlock(nn.Module):
    """
    N-BEATS block which takes a basis function as an argument.
    """
    def __init__(self,
                 input_size: int,
                 n_theta: int,
                 mlp_units: list,
                 basis: nn.Module,
                 dropout_prob: float,
                 activation: str):
        """
        """
        super().__init__()

        self.dropout_prob = dropout_prob

        assert activation in ACTIVATIONS, f'{activation} is not in {ACTIVATI
        activ = getattr(nn, activation)()

        hidden_layers = [nn.Linear(in_features=input_size,
                                   out_features=mlp_units[0][0])]
        for layer in mlp_units:
            hidden_layers.append(nn.Linear(in_features=layer[0],
                                           out_features=layer[1]))
            hidden_layers.append(activ)

            if self.dropout_prob>0:
                raise NotImplementedError('dropout')
                #hidden_layers.append(nn.Dropout(p=self.dropout_prob))

        output_layer = [nn.Linear(in_features=mlp_units[-1][1], out_features
        layers = hidden_layers + output_layer
        self.layers = nn.Sequential(*layers)
```

```python
        self.basis = basis

    def forward(self, insample_y: torch.Tensor) -> Tuple[torch.Tensor, torch
        # Compute local projection weights and projection
        theta = self.layers(insample_y)
        backcast, forecast = self.basis(theta)
        return backcast, forecast
```

In [7]:
```python
#| export
class NBEATS(BaseWindows):
    """ NBEATS

    The Neural Basis Expansion Analysis for Time Series (NBEATS), is a simpl
    effective architecture, it is built with a deep stack of MLPs with the c
    residual connections. It has a generic and interpretable architecture de
    on the blocks it uses. Its interpretable architecture is recommended for
    data settings, as it regularizes its predictions through projections unt
    and trend basis well-suited for most forecasting tasks.

    **Parameters:**<br>
    `h`: int, forecast horizon.<br>
    `input_size`: int, considered autorregresive inputs (lags), y=[1,2,3,4]
    `n_harmonics`: int, Number of harmonic terms for seasonality stack type.
    `n_polynomials`: int, polynomial degree for trend stack. Note that len(n
    `stack_types`: List[str], List of stack types. Subset from ['seasonality
    `n_blocks`: List[int], Number of blocks for each stack. Note that len(n_
    `mlp_units`: List[List[int]], Structure of hidden layers for each stack
    `dropout_prob_theta`: float, Float between (0, 1). Dropout for N-BEATS b
    `shared_weights`: bool, If True, all blocks within each stack will share
    `activation`: str, activation from ['ReLU', 'Softplus', 'Tanh', 'SELU',
    `loss`: PyTorch module, instantiated train loss class from [losses colle
    `valid_loss`: PyTorch module=`loss`, instantiated valid loss class from
    `max_steps`: int=1000, maximum number of training steps.<br>
    `learning_rate`: float=1e-3, Learning rate between (0, 1).<br>
    `num_lr_decays`: int=3, Number of learning rate decays, evenly distribut
    `early_stop_patience_steps`: int=-1, Number of validation iterations bef
    `val_check_steps`: int=100, Number of training steps between every valid
    `batch_size`: int=32, number of different series in each batch.<br>
    `valid_batch_size`: int=None, number of different series in each validat
    `windows_batch_size`: int=1024, number of windows to sample in each trai
    `inference_windows_batch_size`: int=-1, number of windows to sample in e
    `start_padding_enabled`: bool=False, if True, the model will pad the tim
    `step_size`: int=1, step size between each window of temporal data.<br>
    `scaler_type`: str='identity', type of scaler for temporal inputs normal
    `random_seed`: int, random_seed for pytorch initializer and numpy genera
    `num_workers_loader`: int=os.cpu_count(), workers to be used by `TimeSer
    `drop_last_loader`: bool=False, if True `TimeSeriesDataLoader` drops las
    `alias`: str, optional,  Custom name of the model.<br>
    `**trainer_kwargs`: int,  keyword trainer arguments inherited from [PyTo

    **References:**<br>
    -[Boris N. Oreshkin, Dmitri Carpov, Nicolas Chapados, Yoshua Bengio (201
    "N-BEATS: Neural basis expansion analysis for interpretable time series
    """
    # Class attributes
    SAMPLING_TYPE = 'windows'
```

Loading [MathJax]/extensions/Safe.js

```
    def __init__(self,
                 h,
                 input_size,
                 n_harmonics: int = 2,
                 n_polynomials: int = 2,
                 stack_types: list = ['identity', 'trend', 'seasonality'],
                 n_blocks: list = [1, 1, 1],
                 mlp_units: list = 3 * [[512, 512]],
                 dropout_prob_theta: float = 0.,
                 activation: str = 'ReLU',
                 shared_weights: bool = False,
                 loss = MAE(),
                 valid_loss = None,
                 max_steps: int = 1000,
                 learning_rate: float = 1e-3,
                 num_lr_decays: int = 3,
                 early_stop_patience_steps: int =-1,
                 val_check_steps: int = 100,
                 batch_size: int = 32,
                 valid_batch_size: Optional[int] = None,
                 windows_batch_size: int = 1024,
                 inference_windows_batch_size: int = -1,
                 start_padding_enabled = False,
                 step_size: int = 1,
                 scaler_type: str ='identity',
                 random_seed: int = 1,
                 num_workers_loader: int = 0,
                 drop_last_loader: bool = False,
                 **trainer_kwargs):

        # Inherit BaseWindows class
        super(NBEATS, self).__init__(h=h,
                                     input_size=input_size,
                                     loss=loss,
                                     valid_loss=valid_loss,
                                     max_steps=max_steps,
                                     learning_rate=learning_rate,
                                     num_lr_decays=num_lr_decays,
                                     early_stop_patience_steps=early_stop_pa
                                     val_check_steps=val_check_steps,
                                     batch_size=batch_size,
                                     windows_batch_size=windows_batch_size,
                                     valid_batch_size=valid_batch_size,
                                     inference_windows_batch_size=inference_
                                     start_padding_enabled=start_padding_ena
                                     step_size=step_size,
                                     scaler_type=scaler_type,
                                     num_workers_loader=num_workers_loader,
                                     drop_last_loader=drop_last_loader,
                                     random_seed=random_seed,
                                     **trainer_kwargs)

        # Architecture
        blocks = self.create_stack(h=h,
                                   input_size=input_size,
```

```
                                   stack_types=stack_types,
                                   n_blocks=n_blocks,
                                   mlp_units=mlp_units,
                                   dropout_prob_theta=dropout_prob_theta,
                                   activation=activation,
                                   shared_weights=shared_weights,
                                   n_polynomials=n_polynomials,
                                   n_harmonics=n_harmonics)
        self.blocks = torch.nn.ModuleList(blocks)

    def create_stack(self, stack_types,
                     n_blocks,
                     input_size,
                     h,
                     mlp_units,
                     dropout_prob_theta,
                     activation, shared_weights,
                     n_polynomials, n_harmonics):

        block_list = []
        for i in range(len(stack_types)):
            for block_id in range(n_blocks[i]):

                # Shared weights
                if shared_weights and block_id>0:
                    nbeats_block = block_list[-1]
                else:
                    if stack_types[i] == 'seasonality':
                        n_theta = 2 * (self.loss.outputsize_multiplier + 1)
                                  int(np.ceil(n_harmonics / 2 * h) - (n_harm
                        basis = SeasonalityBasis(harmonics=n_harmonics,
                                                 backcast_size=input_size,fo
                                                 out_features=self.loss.outp

                    elif stack_types[i] == 'trend':
                        n_theta = (self.loss.outputsize_multiplier + 1) * (n
                        basis = TrendBasis(degree_of_polynomial=n_polynomial
                                           backcast_size=input_size,forecast
                                           out_features=self.loss.outputsize

                    elif stack_types[i] == 'identity':
                        n_theta = input_size + self.loss.outputsize_multipli
                        basis = IdentityBasis(backcast_size=input_size, fore
                                              out_features=self.loss.outputs
                    else:
                        raise ValueError(f'Block type {stack_types[i]} not f

                    nbeats_block = NBEATSBlock(input_size=input_size,
                                               n_theta=n_theta,
                                               mlp_units=mlp_units,
                                               basis=basis,
                                               dropout_prob=dropout_prob_the
                                               activation=activation)

                # Select type of evaluation and apply it to all layers of bl
                block_list.append(nbeats_block)
```

```
        return block_list

    def forward(self, windows_batch):

        # Parse windows_batch
        insample_y    = windows_batch['insample_y']
        insample_mask = windows_batch['insample_mask']

        # NBEATS' forward
        residuals = insample_y.flip(dims=(-1,)) # backcast init
        insample_mask = insample_mask.flip(dims=(-1,))

        forecast = insample_y[:, -1:, None] # Level with Naive1
        block_forecasts = [ forecast.repeat(1, self.h, 1) ]
        for i, block in enumerate(self.blocks):
            backcast, block_forecast = block(insample_y=residuals)
            residuals = (residuals - backcast) * insample_mask
            forecast = forecast + block_forecast

            if self.decompose_forecast:
                block_forecasts.append(block_forecast)

        # Adapting output's domain
        forecast = self.loss.domain_map(forecast)

        if self.decompose_forecast:
            # (n_batch, n_blocks, h, out_features)
            block_forecasts = torch.stack(block_forecasts)
            block_forecasts = block_forecasts.permute(1,0,2,3)
            block_forecasts = block_forecasts.squeeze(-1) # univariate outpu
            return block_forecasts
        else:
            return forecast
```

In [8]:
```
show_doc(NBEATS)
```

Out[8]:

NBEATS

```
NBEATS (h, input_size, n_harmonics:int=2, n_polynomials:i
nt=2,
         stack_types:list=['identity', 'trend', 'seasonali
ty'],
         n_blocks:list=[1, 1, 1], mlp_units:list=[[512, 51
2], [512, 512],
         [512, 512]], dropout_prob_theta:float=0.0, activa
tion:str='ReLU',
         shared_weights:bool=False, loss=MAE(), valid_loss
=None,
         max_steps:int=1000, learning_rate:float=0.001,
         num_lr_decays:int=3, early_stop_patience_steps:in
t=-1,
         val_check_steps:int=100, batch_size:int=32,
         valid_batch_size:Optional[int]=None, windows_batc
h_size:int=1024,
         inference_windows_batch_size:int=-1, start_paddin
g_enabled=False,
         step_size:int=1, scaler_type:str='identity', rand
om_seed:int=1,
         num_workers_loader:int=0, drop_last_loader:bool=F
alse,
         **trainer_kwargs)
```

NBEATS

The Neural Basis Expansion Analysis for Time Series (NBEATS), is a simple and yet effective architecture, it is built with a deep stack of MLPs with the doubly residual connections. It has a generic and interpretable architecture depending on the blocks it uses. Its interpretable architecture is recommended for scarce data settings, as it regularizes its predictions through projections unto harmonic and trend basis well-suited for most forecasting tasks.

Parameters:
`h`: int, forecast horizon.
`input_size`: int, considered autorregresive inputs (lags), y=[1,2,3,4] input_size=2 -> lags=[1,2].
`n_harmonics`: int, Number of harmonic terms for seasonality stack type. Note that len(n_harmonics) = len(stack_types). Note that it will only be used if a seasonality stack is used.

`n_polynomials` : int, polynomial degree for trend stack. Note that len(n_polynomials) = len(stack_types). Note that it will only be used if a trend stack is used.
`stack_types` : List[str], List of stack types. Subset from ['seasonality', 'trend', 'identity'].
`n_blocks` : List[int], Number of blocks for each stack. Note that len(n_blocks) = len(stack_types).
`mlp_units` : List[List[int]], Structure of hidden layers for each stack type. Each internal list should contain the number of units of each hidden layer. Note that len(n_hidden) = len(stack_types).
`dropout_prob_theta` : float, Float between (0, 1). Dropout for N-BEATS basis.
`shared_weights` : bool, If True, all blocks within each stack will share parameters.
`activation` : str, activation from ['ReLU', 'Softplus', 'Tanh', 'SELU', 'LeakyReLU', 'PReLU', 'Sigmoid'].
`loss` : PyTorch module, instantiated train loss class from losses collection.
`valid_loss` : PyTorch module= `loss` , instantiated valid loss class from losses collection.
`max_steps` : int=1000, maximum number of training steps.
`learning_rate` : float=1e-3, Learning rate between (0, 1).
`num_lr_decays` : int=3, Number of learning rate decays, evenly distributed across max_steps.
`early_stop_patience_steps` : int=-1, Number of validation iterations before early stopping.
`val_check_steps` : int=100, Number of training steps between every validation loss check.
`batch_size` : int=32, number of different series in each batch.
`valid_batch_size` : int=None, number of different series in each validation and test batch, if None uses batch_size.
`windows_batch_size` : int=1024, number of windows to sample in each training batch, default uses all.
`inference_windows_batch_size` : int=-1, number of windows to sample in each inference batch, -1 uses all.
`start_padding_enabled` : bool=False, if True, the model will pad the time series with zeros at the beginning, by input size.
`step_size` : int=1, step size between each window of temporal data.
`scaler_type` : str='identity', type of scaler for temporal inputs normalization see temporal scalers.
`random_seed` : int, random_seed for pytorch initializer and numpy generators.
`num_workers_loader` : int=os.cpu_count(), workers to be used by `TimeSeriesDataLoader` .
`drop_last_loader` : bool=False, if True `TimeSeriesDataLoader` drops last non-

full batch.
`alias` : str, optional, Custom name of the model.
`**trainer_kwargs` : int, keyword trainer arguments inherited from PyTorch Lighning's trainer.

References:
-Boris N. Oreshkin, Dmitri Carpov, Nicolas Chapados, Yoshua Bengio (2019). "N-BEATS: Neural basis expansion analysis for interpretable time series forecasting".

In [9]:
```
show_doc(NBEATS.fit, name='NBEATS.fit')
```

Out[9]:

NBEATS.fit

> ```
> NBEATS.fit (dataset, val_size=0, test_size=0, random_seed
> =None)
> ```

Fit.

The `fit` method, optimizes the neural network's weights using the initialization parameters (`learning_rate` , `windows_batch_size` , ...) and the `loss` function as defined during the initialization. Within `fit` we use a PyTorch Lightning `Trainer` that inherits the initialization's `self.trainer_kwargs` , to customize its inputs, see PL's trainer arguments.

The method is designed to be compatible with SKLearn-like classes and in particular to be compatible with the StatsForecast library.

By default the `model` is not saving training checkpoints to protect disk memory, to get them change `enable_checkpointing=True` in `__init__` .

Parameters:
`dataset` : NeuralForecast's `TimeSeriesDataset` , see documentation.
`val_size` : int, validation size for temporal cross-validation.
`random_seed` : int=None, random_seed for pytorch initializer and numpy generators, overwrites model.**init**'s.
`test_size` : int, test size for temporal cross-validation.

In [10]:
```
show_doc(NBEATS.predict, name='NBEATS.predict')
```

Out[10]:

NBEATS.predict

> NBEATS.predict (dataset, test_size=None, step_size=1, ran
> dom_seed=None,
>   **data_module_kwargs)

Predict.

Neural network prediction with PL's `Trainer` execution of `predict_step`.

Parameters:

`dataset`: NeuralForecast's `TimeSeriesDataset`, see documentation.
`test_size`: int=None, test size for temporal cross-validation.
`step_size`: int=1, Step size between each window.
`random_seed`: int=None, random_seed for pytorch initializer and numpy generators, overwrites model.**init**'s.
`**data_module_kwargs`: PL's TimeSeriesDataModule args, see documentation.

In [11]:

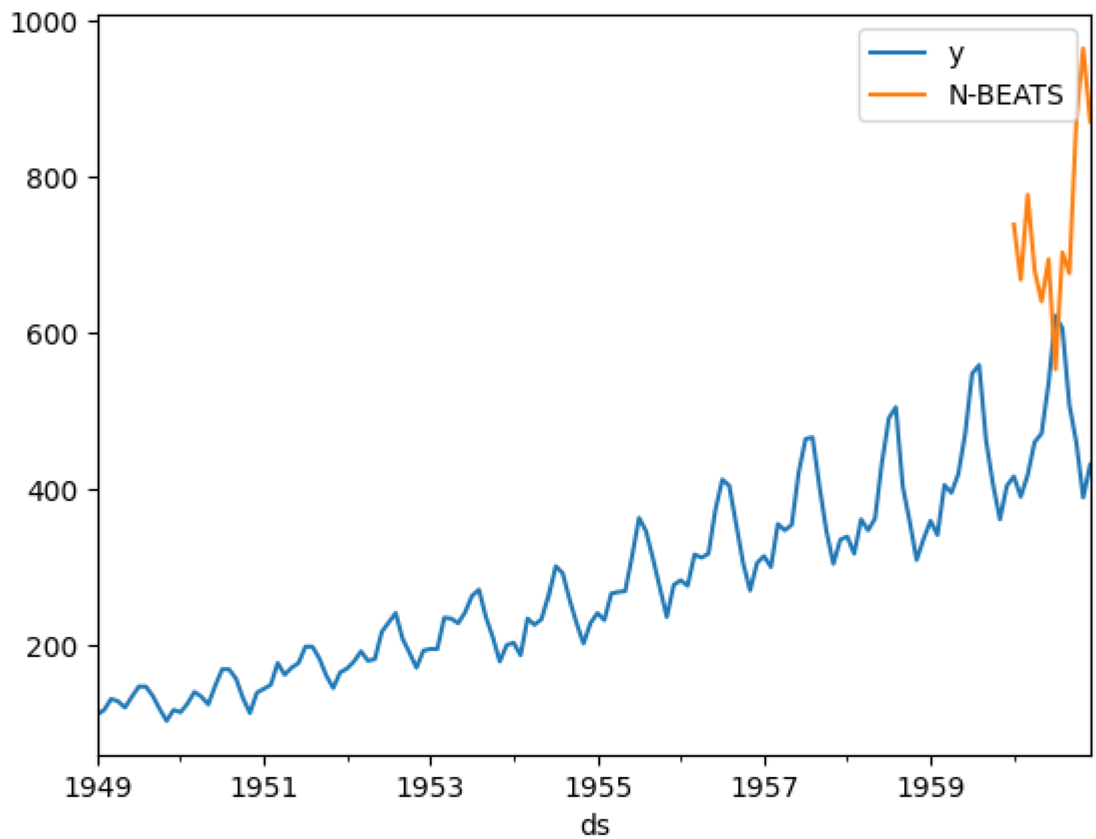
```
#| hide
import logging
import warnings
logging.getLogger("pytorch_lightning").setLevel(logging.ERROR)
warnings.filterwarnings("ignore")
```

In [12]:

```
#| hide
import pandas as pd
import matplotlib.pyplot as plt

import pytorch_lightning as pl

from neuralforecast.utils import AirPassengersDF as Y_df
from neuralforecast.tsdataset import TimeSeriesDataset, TimeSeriesLoader

Y_train_df = Y_df[Y_df.ds<Y_df['ds'].values[-12]] # 132 train
Y_test_df = Y_df[Y_df.ds>=Y_df['ds'].values[-12]]   # 12 test

dataset, *_ = TimeSeriesDataset.from_df(df = Y_train_df)
nbeats = NBEATS(h=12, input_size=24, windows_batch_size=None,
                stack_types=['identity', 'trend', 'seasonality'], max_steps=
nbeats.fit(dataset=dataset)
y_hat = nbeats.predict(dataset=dataset)
Y_test_df['N-BEATS'] = y_hat

pd.concat([Y_train_df, Y_test_df]).drop('unique_id', axis=1).set_index('ds')
```

```
Seed set to 1
2023-11-02 02:21:01.753880: I tensorflow/core/util/port.cc:111] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-11-02 02:21:01.788232: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could n
ot find cuda drivers on your machine, GPU will not be used.
2023-11-02 02:21:01.929456: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_dnn.cc:9342] Unable to register cuDNN factory: Attempting to register fa
ctory for plugin cuDNN when one has already been registered
2023-11-02 02:21:01.929492: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_fft.cc:609] Unable to register cuFFT factory: Attempting to register fac
tory for plugin cuFFT when one has already been registered
2023-11-02 02:21:01.930433: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_blas.cc:1518] Unable to register cuBLAS factory: Attempting to register
factory for plugin cuBLAS when one has already been registered
2023-11-02 02:21:02.015786: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operation
s, rebuild TensorFlow with the appropriate compiler flags.
2023-11-02 02:21:02.979199: W tensorflow/compiler/tf2tensorrt/utils/py_util
s.cc:38] TF-TRT Warning: Could not find TensorRT
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

Out[12]:
```
<Axes: xlabel='ds'>
```



In [13]:

```
#| hide
#test we recover the same forecast
y_hat2 = nbeats.predict(dataset=dataset)
test_eq(y_hat, y_hat2)
```

```
Predicting: |
| 0/? [00:00…
```

In [14]:

```
#| hide
#test no leakage with test_size
dataset, *_ = TimeSeriesDataset.from_df(Y_df)
model = NBEATS(input_size=24, h=12,
               windows_batch_size=None, max_steps=1)
model.fit(dataset=dataset, test_size=12)
y_hat_test = model.predict(dataset=dataset, step_size=1)
np.testing.assert_almost_equal(y_hat, y_hat_test, decimal=4)
#test we recover the same forecast
y_hat_test2 = model.predict(dataset=dataset, step_size=1)
test_eq(y_hat_test, y_hat_test2)
```

```
Seed set to 1
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

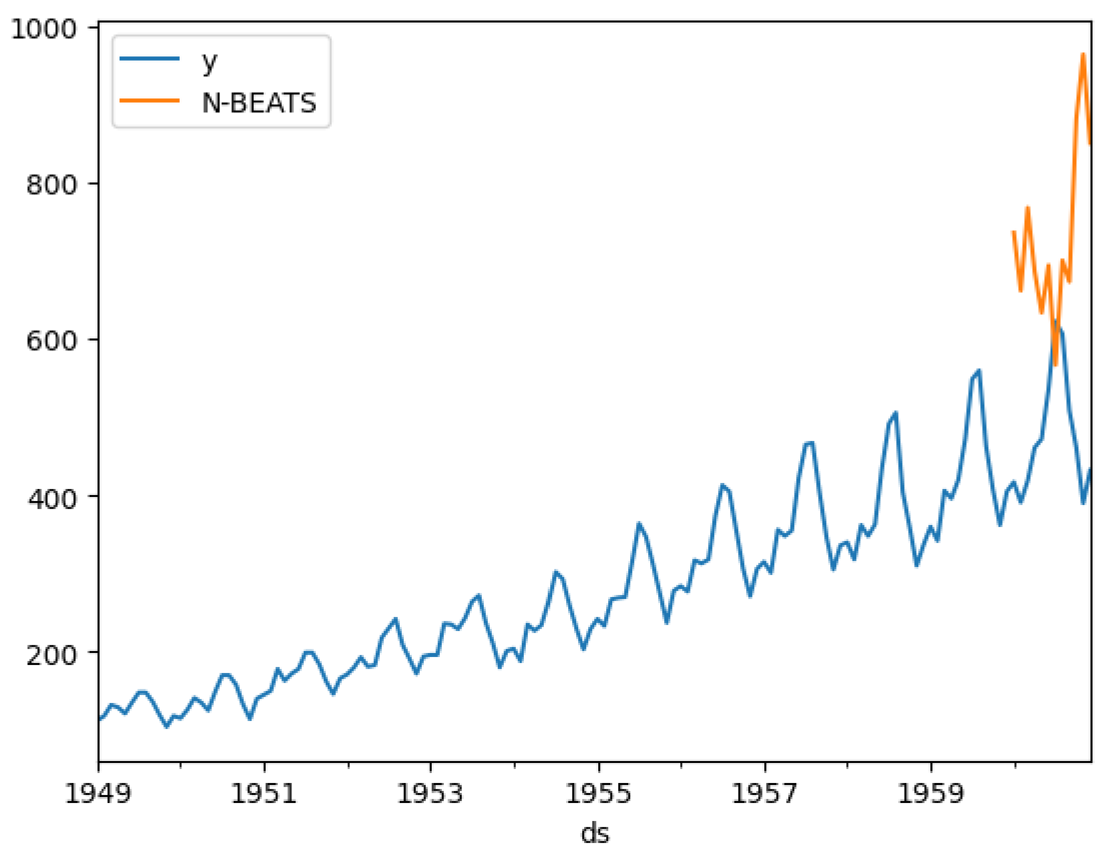
```
Predicting: |
| 0/? [00:00…
```

In [15]:

```
#| hide
# test validation step
dataset, *_ = TimeSeriesDataset.from_df(Y_train_df)
model = NBEATS(input_size=24, h=12, windows_batch_size=None, max_steps=1)
model.fit(dataset=dataset, val_size=12)
y_hat_w_val = model.predict(dataset=dataset)
Y_test_df['N-BEATS'] = y_hat_w_val

pd.concat([Y_train_df, Y_test_df]).drop('unique_id', axis=1).set_index('ds')
```

```
Seed set to 1
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

Out[15]: <Axes: xlabel='ds'>



In [16]:

```
#| hide
# test no leakage with test_size and val_size
dataset, *_ = TimeSeriesDataset.from_df(Y_train_df)
model = NBEATS(input_size=24, h=12, windows_batch_size=None, max_steps=1)
model.fit(dataset=dataset, val_size=12)
y_hat_w_val = model.predict(dataset=dataset)
```

Loading [MathJax]/extensions/Safe.js

```
dataset, *_ = TimeSeriesDataset.from_df(Y_df)
model = NBEATS(input_size=24, h=12, windows_batch_size=None, max_steps=1)
model.fit(dataset=dataset, val_size=12, test_size=12)

y_hat_test_w_val = model.predict(dataset=dataset, step_size=1)

np.testing.assert_almost_equal(y_hat_test_w_val, y_hat_w_val, decimal=4)
```

```
Seed set to 1
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
Seed set to 1
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

In [17]:

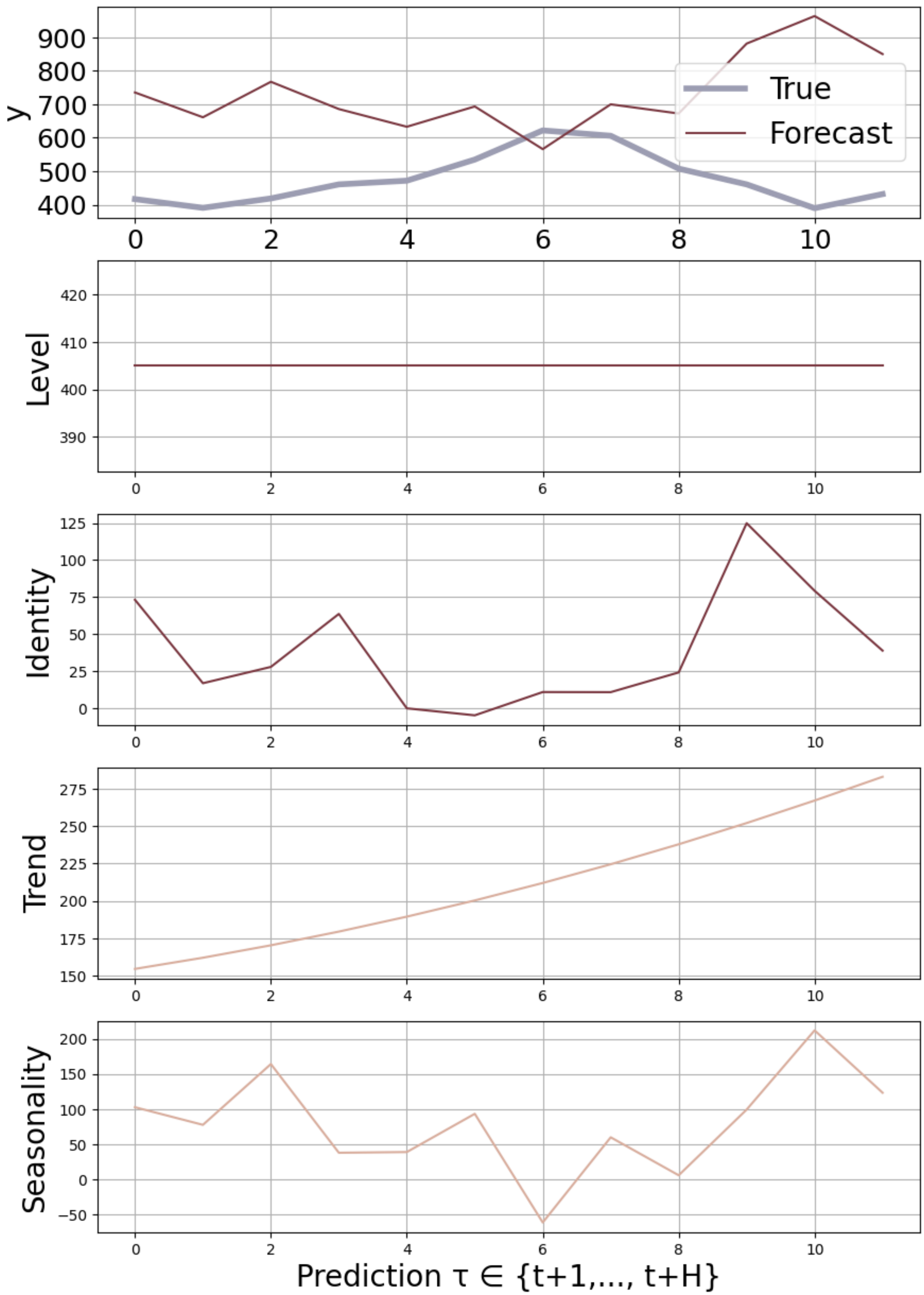
```
#| hide
# qualitative decomposition evaluation
y_hat = model.decompose(dataset=dataset)

fig, ax = plt.subplots(5, 1, figsize=(10, 15))

ax[0].plot(Y_test_df['y'].values, label='True', color="#9C9DB2", linewidth=4
ax[0].plot(y_hat.sum(axis=1).flatten(), label='Forecast', color="#7B3841")
ax[0].grid()
ax[0].legend(prop={'size': 20})
for label in (ax[0].get_xticklabels() + ax[0].get_yticklabels()):
    label.set_fontsize(18)
ax[0].set_ylabel('y', fontsize=20)

ax[1].plot(y_hat[0,0], label='level', color="#7B3841")
ax[1].grid()
ax[1].set_ylabel('Level', fontsize=20)

ax[2].plot(y_hat[0,1], label='stack1', color="#7B3841")
ax[2].grid()
ax[2].set_ylabel('Identity', fontsize=20)

ax[3].plot(y_hat[0,2], label='stack2', color="#D9AE9E")
ax[3].grid()
ax[3].set_ylabel('Trend', fontsize=20)

ax[4].plot(y_hat[0,3], label='stack3', color="#D9AE9E")
d()
```

```
ax[4].set_ylabel('Seasonality', fontsize=20)

ax[4].set_xlabel('Prediction \u03C4 \u2208 {t+1,..., t+H}', fontsize=20)
```

```
Predicting: |
| 0/? [00:00…
```

Out[17]:

```
Text(0.5, 0, 'Prediction τ ∈ {t+1,..., t+H}')
```



Usage Example

Loading [MathJax]/extensions/Safe.js

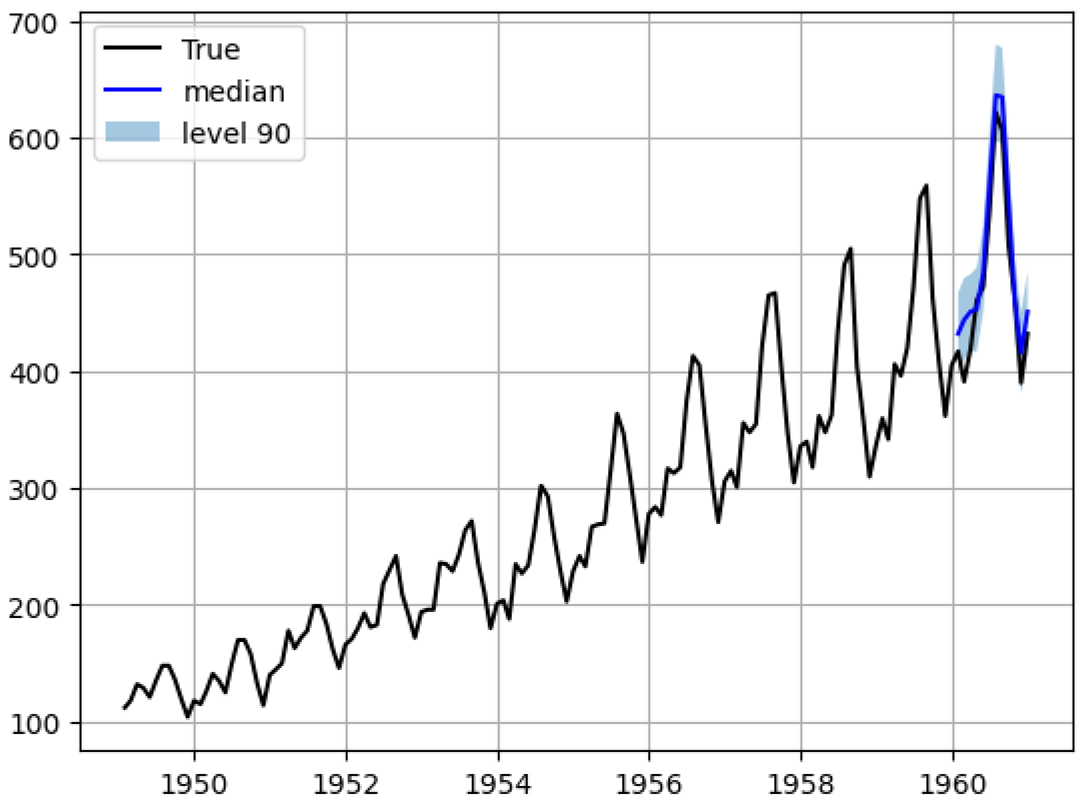
In [18]:

```python
#| eval: false
import numpy as np
import pandas as pd
import pytorch_lightning as pl
import matplotlib.pyplot as plt

from neuralforecast import NeuralForecast
from neuralforecast.models import NBEATS
from neuralforecast.losses.pytorch import MQLoss, DistributionLoss
from neuralforecast.tsdataset import TimeSeriesDataset
from neuralforecast.utils import AirPassengers, AirPassengersPanel, AirPasse

Y_train_df = AirPassengersPanel[AirPassengersPanel.ds<AirPassengersPanel['ds
Y_test_df = AirPassengersPanel[AirPassengersPanel.ds>=AirPassengersPanel['ds

model = NBEATS(h=12, input_size=24,
               loss=DistributionLoss(distribution='Poisson', level=[80, 90])
               stack_types = ['identity', 'trend', 'seasonality'],
               max_steps=100,
               val_check_steps=10,
               early_stop_patience_steps=2)

fcst = NeuralForecast(
    models=[model],
    freq='M'
)
fcst.fit(df=Y_train_df, static_df=AirPassengersStatic, val_size=12)
forecasts = fcst.predict(futr_df=Y_test_df)

# Plot quantile predictions
Y_hat_df = forecasts.reset_index(drop=False).drop(columns=['unique_id','ds']
plot_df = pd.concat([Y_test_df, Y_hat_df], axis=1)
plot_df = pd.concat([Y_train_df, plot_df])

plot_df = plot_df[plot_df.unique_id=='Airline1'].drop('unique_id', axis=1)
plt.plot(plot_df['ds'], plot_df['y'], c='black', label='True')
plt.plot(plot_df['ds'], plot_df['NBEATS-median'], c='blue', label='median')
plt.fill_between(x=plot_df['ds'][-12:],
                 y1=plot_df['NBEATS-lo-90'][-12:].values,
                 y2=plot_df['NBEATS-hi-90'][-12:].values,
                 alpha=0.4, label='level 90')
plt.grid()
plt.legend()
plt.plot()
```

```
Seed set to 1
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
```

```
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

Out[18]: []



In [19]:

```
from neuralforecast.losses.numpy import mae, mse

y_true = Y_test_df.y.values
y_hat = Y_hat_df['NBEATS-median'].values


print('MAE: ', mae(y_hat, y_true))
print('MSE: ', mse(y_hat, y_true))
```

```
MAE:  24.979166666666668
MSE:  842.46875
```

Exchange rate

In [9]:

```
import pandas as pd
from neuralforecast.core import NeuralForecast
```

Loading [MathJax]/extensions/Safe.js

```python
import numpy as np
import pytorch_lightning as pl
import matplotlib.pyplot as plt

from neuralforecast.models import NBEATS
from neuralforecast.losses.pytorch import MQLoss, DistributionLoss

Y_df = pd.read_csv("raw_data/df_Exchange.csv")

Y_df['ds'] = pd.to_datetime(Y_df['ds'])

# For this excercise we are going to take 20% of the DataSet
n_time = len(Y_df.ds.unique())
val_size = int(.1 * n_time)
test_size = int(.2 * n_time)

Y_df.groupby('unique_id').head(2)
```

Out[9]:

	unique_id	ds	y
0	0	1990-01-01	0.606785
1	0	1990-01-02	0.570900
7588	1	1990-01-01	-0.361671
7589	1	1990-01-02	-0.367639
15176	2	1990-01-01	0.735367
15177	2	1990-01-02	0.729629
22764	3	1990-01-01	-1.164373
22765	3	1990-01-02	-1.170907
30352	4	1990-01-01	2.851890
30353	4	1990-01-02	2.851890
37940	5	1990-01-01	-1.861369
37941	5	1990-01-02	-1.838665
45528	6	1990-01-01	-1.820047
45529	6	1990-01-02	-1.847258
53116	OT	1990-01-01	-0.124081
53117	OT	1990-01-02	-0.113588

In [10]:

```python
horizon = 96

model = NBEATS(h=horizon, input_size=5*horizon,
               stack_types = ['identity', 'trend', 'seasonality'],
               batch_size = 8,
               windows_batch_size = 256,
               max_steps = 100)
```

```
Seed set to 1
```

In [11]:
```python
nf = NeuralForecast(
    models=[model],
    freq='D')

Y_hat_df = nf.cross_validation(df=Y_df, val_size=val_size,
                               test_size=test_size, n_windows=None)
```

```
2023-11-04 11:18:19.548041: I tensorflow/core/util/port.cc:111] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-11-04 11:18:19.690180: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could n
ot find cuda drivers on your machine, GPU will not be used.
2023-11-04 11:18:20.297584: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_dnn.cc:9342] Unable to register cuDNN factory: Attempting to register fa
ctory for plugin cuDNN when one has already been registered
2023-11-04 11:18:20.297645: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_fft.cc:609] Unable to register cuFFT factory: Attempting to register fac
tory for plugin cuFFT when one has already been registered
2023-11-04 11:18:20.303500: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_blas.cc:1518] Unable to register cuBLAS factory: Attempting to register
factory for plugin cuBLAS when one has already been registered
2023-11-04 11:18:20.624025: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operation
s, rebuild TensorFlow with the appropriate compiler flags.
2023-11-04 11:18:22.910326: W tensorflow/compiler/tf2tensorrt/utils/py_util
s.cc:38] TF-TRT Warning: Could not find TensorRT
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

In [12]:
```python
Y_hat_df.to_csv('results/Exchange_rate/NBEATS.csv')
```

In [13]:
```python
from neuralforecast.losses.numpy import mae, mse

print('MAE: ', mae(Y_hat_df['y'], Y_hat_df['NBEATS']))
print('MSE: ', mse(Y_hat_df['y'], Y_hat_df['NBEATS']))
```

```
MAE:  0.20772644794136158
MSE:  0.08869165061559935
```

Ettm2

In [10]:
```python
import pandas as pd
from neuralforecast.core import NeuralForecast
import numpy as np
```

Loading [MathJax]/extensions/Safe.js

```python
import pytorch_lightning as pl
import matplotlib.pyplot as plt

from neuralforecast.models import NBEATS
from neuralforecast.losses.pytorch import MQLoss, DistributionLoss

Y_df = pd.read_csv("raw_data/df_Ettm2.csv")

Y_df['ds'] = pd.to_datetime(Y_df['ds'])

# For this excercise we are going to take 20% of the DataSet
n_time = len(Y_df.ds.unique())
val_size = int(.2 * n_time)
test_size = int(.2 * n_time)

Y_df.groupby('unique_id').head(2)
```

Out[10]:

	unique_id	ds	y
0	HUFL	2016-07-01 00:00:00	-0.041413
1	HUFL	2016-07-01 00:15:00	-0.185467
57600	HULL	2016-07-01 00:00:00	0.040104
57601	HULL	2016-07-01 00:15:00	-0.214450
115200	LUFL	2016-07-01 00:00:00	0.695804
115201	LUFL	2016-07-01 00:15:00	0.434685
172800	LULL	2016-07-01 00:00:00	0.434430
172801	LULL	2016-07-01 00:15:00	0.428168
230400	MUFL	2016-07-01 00:00:00	-0.599211
230401	MUFL	2016-07-01 00:15:00	-0.658068
288000	MULL	2016-07-01 00:00:00	-0.393536
288001	MULL	2016-07-01 00:15:00	-0.659338
345600	OT	2016-07-01 00:00:00	1.018032
345601	OT	2016-07-01 00:15:00	0.980124

In [11]:

```python
horizon = 96

model = NBEATS(h=horizon, input_size=5*horizon,
               stack_types = ['identity', 'trend', 'seasonality'],
               batch_size = 7,
               windows_batch_size = 256,
               max_steps = 100,
              val_check_steps=10)


nf = NeuralForecast(
    models=[model],
    freq='15min')
```

```
Y_hat_df = nf.cross_validation(df=Y_df, val_size=val_size,
                               test_size=test_size, n_windows=None)
```

```
Seed set to 1
2023-11-02 02:40:11.197799: I tensorflow/core/util/port.cc:111] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-11-02 02:40:11.199229: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could n
ot find cuda drivers on your machine, GPU will not be used.
2023-11-02 02:40:11.217067: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_dnn.cc:9342] Unable to register cuDNN factory: Attempting to register fa
ctory for plugin cuDNN when one has already been registered
2023-11-02 02:40:11.217086: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_fft.cc:609] Unable to register cuFFT factory: Attempting to register fac
tory for plugin cuFFT when one has already been registered
2023-11-02 02:40:11.217100: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_blas.cc:1518] Unable to register cuBLAS factory: Attempting to register
factory for plugin cuBLAS when one has already been registered
2023-11-02 02:40:11.221761: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operation
s, rebuild TensorFlow with the appropriate compiler flags.
2023-11-02 02:40:11.779149: W tensorflow/compiler/tf2tensorrt/utils/py_util
s.cc:38] TF-TRT Warning: Could not find TensorRT
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

In [12]:
```
from neuralforecast.losses.numpy import mae, mse

print('MAE: ', mae(Y_hat_df['y'], Y_hat_df['NBEATS']))
print('MSE: ', mse(Y_hat_df['y'], Y_hat_df['NBEATS']))
```

```
MAE:  0.26536146140480926
MSE:  0.1812076133858475
```

In [29]:
```
Y_hat_df.to_csv('results/Ettm2/NBEATS.csv')
```

Weather

In [8]:
```
import pandas as pd
from neuralforecast.core import NeuralForecast
import numpy as np
import pytorch_lightning as pl
import matplotlib.pyplot as plt

from neuralforecast.models import NBEATS
```

```
from neuralforecast.losses.pytorch import MQLoss, DistributionLoss

Y_df = pd.read_csv("raw_data/df_Weather.csv")

Y_df['ds'] = pd.to_datetime(Y_df['ds'])

# For this excercise we are going to take 20% of the DataSet
n_time = len(Y_df.ds.unique())
val_size = int(.1 * n_time)
test_size = int(.2 * n_time)

Y_df.groupby('unique_id').head(2)
```

Out[8]:

	unique_id	ds	y
0	H2OC (mmol/mol)	2020-01-01 00:10:00	-0.999107
1	H2OC (mmol/mol)	2020-01-01 00:20:00	-1.008072
52695	OT	2020-01-01 00:10:00	0.044395
52696	OT	2020-01-01 00:20:00	0.044134
105390	PAR (�mol/m�/s)	2020-01-01 00:10:00	-0.679493
105391	PAR (�mol/m�/s)	2020-01-01 00:20:00	-0.679493
158085	SWDR (W/m�)	2020-01-01 00:10:00	-0.672767
158086	SWDR (W/m�)	2020-01-01 00:20:00	-0.672767
210780	T (degC)	2020-01-01 00:10:00	-1.459980
210781	T (degC)	2020-01-01 00:20:00	-1.454798
263475	Tdew (degC)	2020-01-01 00:10:00	-1.052596
263476	Tdew (degC)	2020-01-01 00:20:00	-1.069612
316170	Tlog (degC)	2020-01-01 00:10:00	-1.424132
316171	Tlog (degC)	2020-01-01 00:20:00	-1.416612
368865	Tpot (K)	2020-01-01 00:10:00	-1.607935
368866	Tpot (K)	2020-01-01 00:20:00	-1.602882
421560	VPact (mbar)	2020-01-01 00:10:00	-0.979132
421561	VPact (mbar)	2020-01-01 00:20:00	-0.990506
474255	VPdef (mbar)	2020-01-01 00:10:00	-0.838497
474256	VPdef (mbar)	2020-01-01 00:20:00	-0.828332
526950	VPmax (mbar)	2020-01-01 00:10:00	-1.141181
526951	VPmax (mbar)	2020-01-01 00:20:00	-1.138714
579645	max. PAR (�mol/m�/s)	2020-01-01 00:10:00	-0.588296
579646	max. PAR (�mol/m�/s)	2020-01-01 00:20:00	-0.588296
632340	max. wv (m/s)	2020-01-01 00:10:00	-0.832381
632341	max. wv (m/s)	2020-01-01 00:20:00	-1.125140
685035	p (mbar)	2020-01-01 00:10:00	2.114257
685036	p (mbar)	2020-01-01 00:20:00	2.099194
737730	rain (mm)	2020-01-01 00:10:00	-0.093506
737731	rain (mm)	2020-01-01 00:20:00	-0.093506
790425	raining (s)	2020-01-01 00:10:00	-0.221050
790426	raining (s)	2020-01-01 00:20:00	-0.221050
843120	rh (%)	2020-01-01 00:10:00	0.990128

	unique_id	ds	y
843121	rh (%)	2020-01-01 00:20:00	0.942141
895815	rho (g/m**3)	2020-01-01 00:10:00	1.940406
895816	rho (g/m**3)	2020-01-01 00:20:00	1.932788
948510	sh (g/kg)	2020-01-01 00:10:00	-0.998513
948511	sh (g/kg)	2020-01-01 00:20:00	-1.009228
1001205	wd (deg)	2020-01-01 00:10:00	0.555571
1001206	wd (deg)	2020-01-01 00:20:00	0.354339
1053900	wv (m/s)	2020-01-01 00:10:00	-0.017801
1053901	wv (m/s)	2020-01-01 00:20:00	-0.029125

In [9]:

```
horizon = 96

model = NBEATS(h=horizon, input_size=5*horizon,
               stack_types = ['identity', 'trend', 'seasonality'],
               batch_size = 21,
               windows_batch_size = 256,
               max_steps = 100,
              val_check_steps=10)


nf = NeuralForecast(
    models=[model],
    freq='10min')

Y_hat_df = nf.cross_validation(df=Y_df, val_size=val_size,
                               test_size=test_size, n_windows=None)
```

Loading [MathJax]/extensions/Safe.js

```
Seed set to 1
2023-11-02 18:41:08.083774: I tensorflow/core/util/port.cc:111] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-11-02 18:41:08.119009: I tensorflow/tsl/cuda/cudart_stub.cc:28] Could n
ot find cuda drivers on your machine, GPU will not be used.
2023-11-02 18:41:08.287268: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_dnn.cc:9342] Unable to register cuDNN factory: Attempting to register fa
ctory for plugin cuDNN when one has already been registered
2023-11-02 18:41:08.287296: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_fft.cc:609] Unable to register cuFFT factory: Attempting to register fac
tory for plugin cuFFT when one has already been registered
2023-11-02 18:41:08.288277: E tensorflow/compiler/xla/stream_executor/cuda/c
uda_blas.cc:1518] Unable to register cuBLAS factory: Attempting to register
factory for plugin cuBLAS when one has already been registered
2023-11-02 18:41:08.364838: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operation
s, rebuild TensorFlow with the appropriate compiler flags.
2023-11-02 18:41:09.325051: W tensorflow/compiler/tf2tensorrt/utils/py_util
s.cc:38] TF-TRT Warning: Could not find TensorRT
Sanity Checking: |
| 0/? [00:00…
Training: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Validation: |
| 0/? [00:00…
Predicting: |
| 0/? [00:00…
```

In [10]:

```
from neuralforecast.losses.numpy import mae, mse

print('MAE: ', mae(Y_hat_df['y'], Y_hat_df['NBEATS']))
print('MSE: ', mse(Y_hat_df['y'], Y_hat_df['NBEATS']))
```

```
MAE:  0.20784774329722125
MSE:  0.1783033362809145
```

In [11]:
```python
from neuralforecast.losses.numpy import mae, mse

print('MAE: ', mae(Y_hat_df['y'], Y_hat_df['NBEATS']))
print('MSE: ', mse(Y_hat_df['y'], Y_hat_df['NBEATS']))
```

```
MAE:  0.2148191584760335
MSE:  0.18219068222659401
```

In [12]:
```python
data = {'NBEATS_MSE': mse(Y_hat_df['y'], Y_hat_df['NBEATS']),
        'NBEATS_MAE': mae(Y_hat_df['y'], Y_hat_df['NBEATS'])}

df = pd.DataFrame(data, index=['Weather'])
df.to_csv('results/Weather/df_NBEATS.csv')
```

In [12]:
```python
#Y_hat_df.to_csv('results/Weather/NBEATS.csv')
```