Traditional forecasting models like ARIMA and ETS treat each time series separately, creating individual models. These are local models. They may not suit large-scale forecasting because maintaining individual models is impractical. Also, using exogenous covariates in traditional models often requires careful crafting and tuning.

Neural networks (NNs) can address these issues. They learn a compressed representation of input data, allowing a single model for many time series. This enables learning complex interactions between various data sources. These are global models. They simplify forecasting by removing the need for separate models and handcrafted features. Neural networks have a long history dating back to the 1960s but have become popular for forecasting only recently in specialized labs. Examples include Amazon DeepAR (Salinas et al. 2019), Google TFT (Lim et al. 2021), and Nixtla NHITS (Challu et al. 2023).

This chapter starts with a brief introduction to forecasting with neural networks in Section 14.1. Then, it introduces a core neural network architecture, the Multilayer Perceptron (MLP), in Section 14.2. It then covers more complex neural network architectures in Section 14.3. The later sections (14.4-14.7) explain how to forecast with neural networks using the NeuralForecast library.

14.1 Neural forecasting

When using neural networks to forecast, we aim to learn a (compressed) representation of our input data (Bengio, Courville, and Vincent 2013). Mathematically, a neural network is a function f_\theta : X \mapsto Y, with X the input/feature space and Y the dependent variable space. We consider the setting with X = {y_{[0:t]}, x_{[0:t+h]}} and Y = {y_{[t+1:t+h]}}, where h is the forecast horizon, y is the target time series, and x are exogenous covariates. The forecasting task is to estimate the following conditional distribution:

P(y_{[t+1:t+h]} | y_{[0:t]}, x_{[0:t+h]}) = f_\theta (y_{[0:t]}, x_{[0:t+h]}) \tag{14.1}.

We can learn the function f_\theta by minimizing an optimization objective, commonly referred to as *loss function*. We can formalize the optimization objective:

L = f(Y, \hat{Y}_\theta) \tag{14.2} ,

in which \hat{Y}_\theta denotes the forecasts produced by our neural network f_\theta. A frequently used optimization objective when forecasting with neural networks is the *Mean Absolute Error (MAE)* (we will detail a few common optimization objectives in Section 14.5):

\text{MAE}=\frac{1}{N} \sum_{i=0}^N |Y_i - \hat{Y}_{i, \theta}| \tag{14.3} .

We obtain the best forecast when the optimization objective is minimized, which in the case of the MAE means the forecast \hat{Y}_{i, \theta} equals the actual observed value Y_i. The most used procedure to minimize the optimization objective is called *Gradient Descent* (GD). In GD, we iteratively update the learnable parameters of the neural network by using the derivative of the optimization objective with respect to the learnable parameters, i.e.

\theta_{i+1} \leftarrow \theta_i - \alpha \frac{\delta L}{\delta \hat{Y}_\theta} \tag{14.4},

where \alpha denotes the *learning rate*, a *hyperparameter* that allows us to adjust the rate of how fast the network learns the representation of the input data. We will discuss the optimization of these *hyperparameters* in Section 14.7. The key idea is that if we perform a sufficient amount of iterations of GD, we keep updating our parameters \theta of the neural network in a direction that minimizes the optimization objective.

To summarize,

- When forecasting with neural networks we aim to learn a (compressed) representation of our input data;
- We can learn this representation by minimizing an optimization objective, or *loss function*;
- The optimization objective is commonly iteratively minimized using *Gradient Descent (GD)*;
- The optimization procedure involves *hyperparameters*, which are tuning knobs that control the learning procedure.

14.2 Multilayer perceptron

A neural network can be thought of as a network of “neurons” which are organised in layers. The predictors (or inputs) form the

bottom layer, and the forecasts (or outputs) form the top layer. There may also be intermediate layers containing “hidden neurons”.

The simplest networks contain no hidden layers and are equivalent to linear regressions. Figure 14.1 shows the neural network version of a linear regression with four predictors. We refer to the coefficients of these predictors as “weights” or “parameters”. The forecasts are obtained by a linear combination of the inputs. The weights are selected in the neural network framework using Gradient Descent that minimises a loss function such as the Mean Absolute Error (MAE) or the Mean Squared Error (MSE). Of course, in this simple example, we can use linear regression which minimises MSE and is a much more efficient method of training the model.

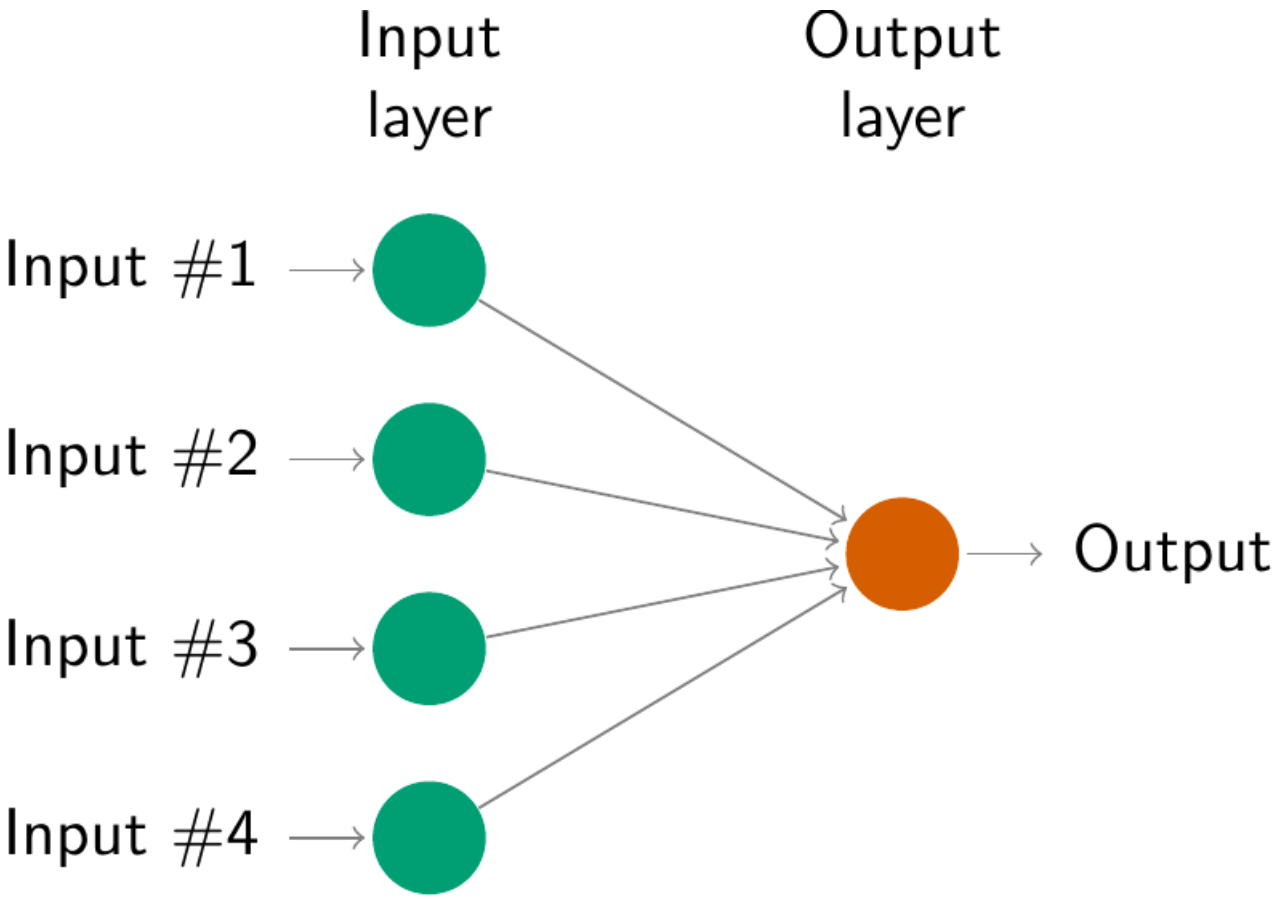

Figure 14.1: A simple neural network equivalent to a linear regression.

Once we add an intermediate layer with hidden neurons, the neural network becomes non-linear. A simple example is shown in Figure 14.2.

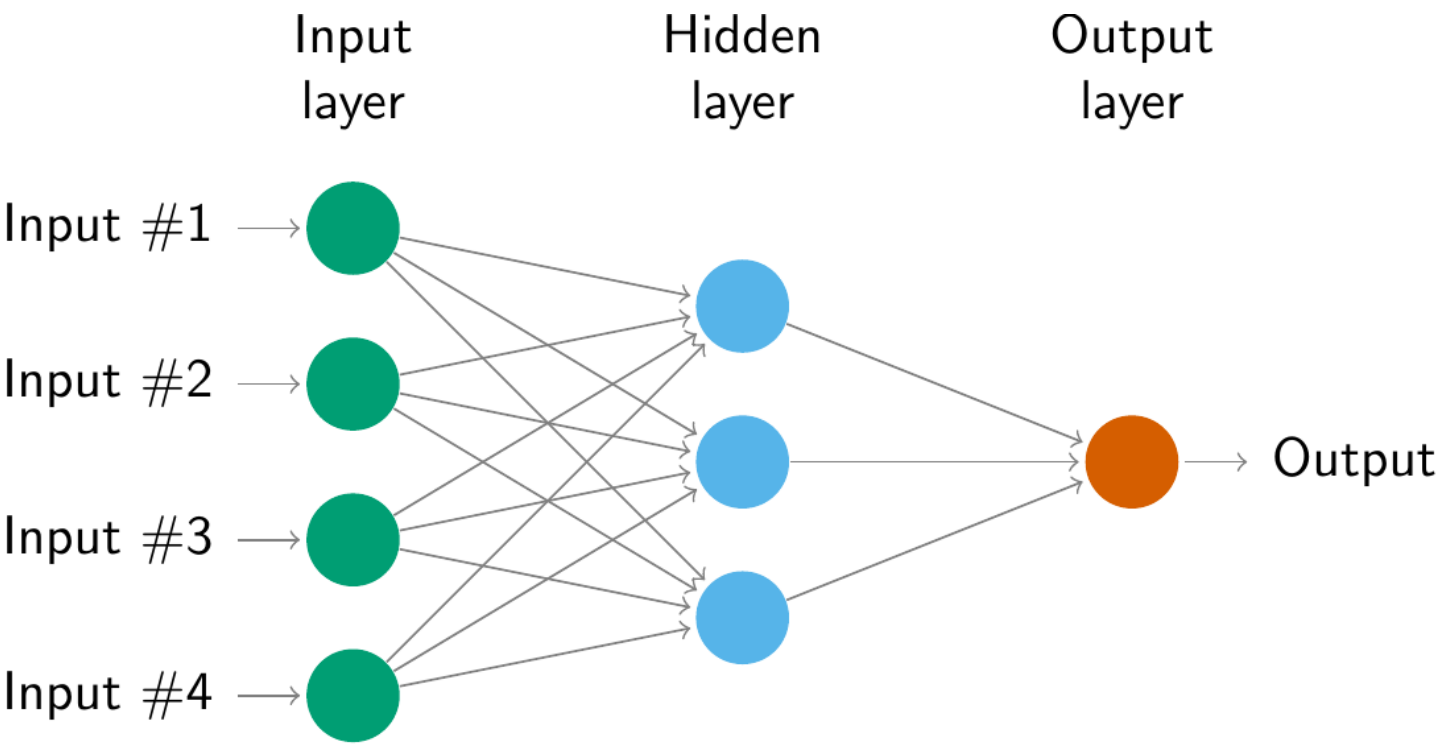

Figure 14.2: A neural network with four inputs and one hidden layer with three hidden neurons.

This is known as a *Multilayer Perceptron (MLP)*, where each layer of nodes receives inputs from the previous layers. The outputs of the nodes in one layer are inputs to the next layer. The inputs to each node are combined using a weighted linear combination. The result is then modified by a nonlinear function before being output. For example, the inputs into each hidden neuron in Figure 14.2 are combined linearly to give z_j = b_j + \sum_{i=1}^4 w_{i,j} x_i. In the hidden layer, this is then modified using a nonlinear function such as a sigmoid, s(z) = \frac{1}{1+e^{-z}}, to give the input for the next layer. This tends to reduce the effect of extreme input values, thus making the network somewhat robust to outliers.

The parameters b_1,b_2,b_3 and w_{1,1},\dots,w_{4,3} are “learned” (or estimated) from the data. The weights are initialized with random values, and are subsequently iteratively updated using Gradient Descent as described in Section 14.1. Consequently, there is an element of randomness in the predictions produced by a neural network.

The number of hidden layers and the number of nodes in each hidden layer are *hyperparameters* that must be specified in advance when trying to make predictions.

Example: Forecasting air passengers with an MLP

The AirPassengers dataset contains data on the number of US airline passengers from 1949 to 1960 for two different airlines. We split the data into a training and test set, where we use the last 12 months for testing. We use an MLP that takes in the last 24 months of observations (`input_size = 24`), and the task is to forecast the next 12 months (horizon `h=12`).

```
# We use the last 12 months for testing
test_size = 12
Y_train_df = AirPassengersPanel.query("ds.dt.year <= 1959")
Y_test_df = AirPassengersPanel.query("ds.dt.year > 1959").reset_index(
    drop=True
)  # 12 test

# Instantiate the MLP
model = MLP(h=12, input_size=24, scaler_type="robust")
# Instantiate the forecast
fcst = NeuralForecast(models=[model], freq="M")

# Fit the model to the training set
fcst.fit(df=Y_train_df)

# Generate predictions
forecasts = fcst.predict()
```

We can verify in Figure 14.3 that the MLP is able to predict the test set for both Airlines fairly accurately.

```
# Plot the predictions and the training data
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series(df=Y_train_df, forecasts_df=pd.concat([forecasts, Y_test_df["y"]], axis=1), ax=a

                  rm_legend=False)
for ax in fig.axes:
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='both')
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.suptitle("Number of passengers for different airlines")
fig.supylabel("Passengers")
fig.supxlabel("Month [1M]")
fig
```

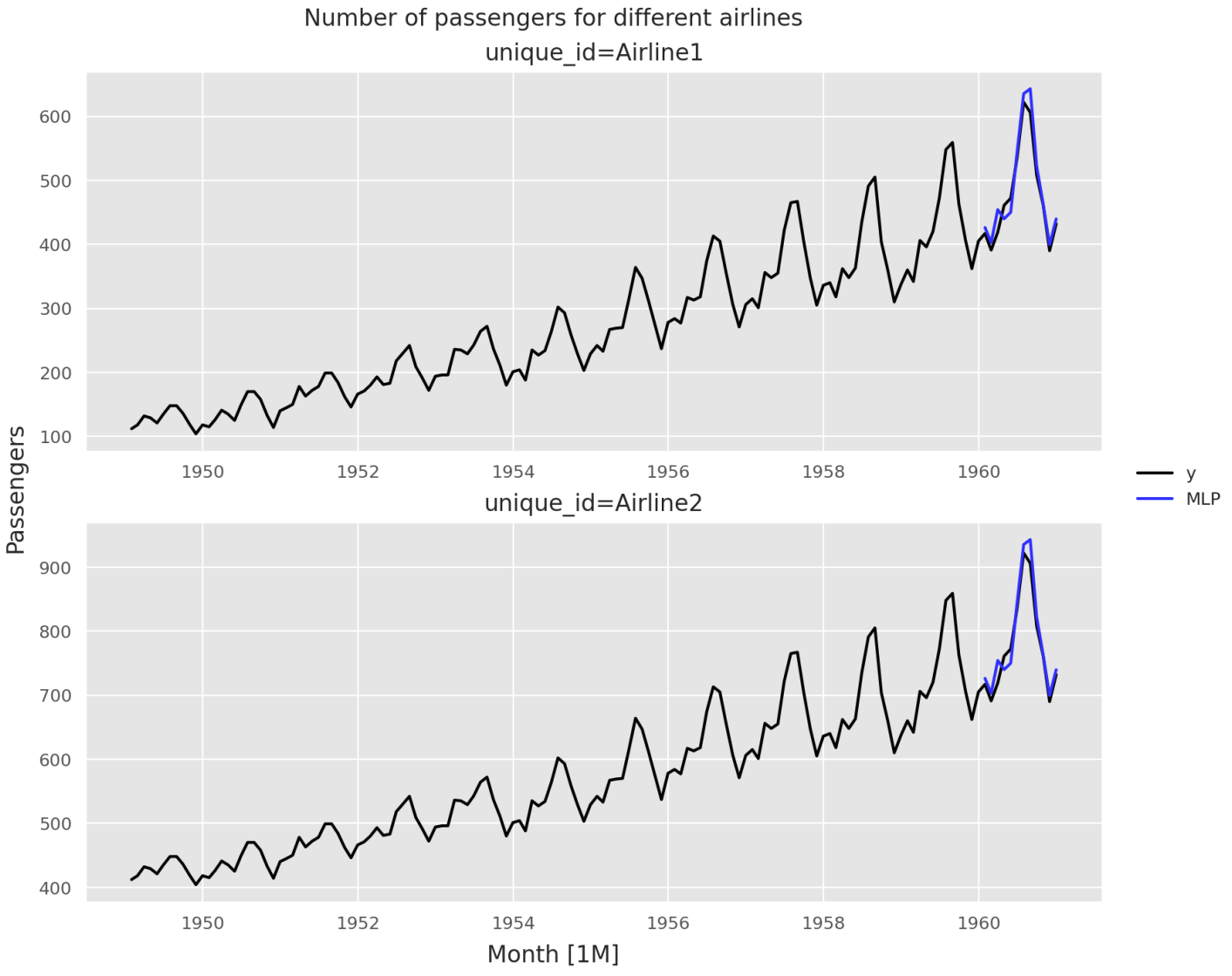

Figure 14.3: AirPassengers MLP example.

14.3 Modern neural network architectures

The *MLP* set out previously is one of the simplest neural network architectures. Nowadays, there is a vast amount of research into the best neural network architectures for forecasting. *NeuralForecast* provides access to many of these modern architectures, listed in Table 14.1.

In Table 14.1, we distinguish the following columns:

1. **Model**: The model name.
2. **Family**: The main neural network architecture underpinning the model. As you can see, there are many contemporary models that employ a variation of the *MLP* described above. Other common architecture families are *Convolutional Neural Network (CNN)*, *Transformers*, and *Recurrent Neural Network (RNN)*. It is beyond the scope of this book to explain all these building blocks in detail, and the interested reader is referred to the documentation of NeuralForecast for further explanation of each architecture.
3. **Univariate / Multivariate**: A multivariate model explicitly models the interactions between multiple time series in a dataset and will provide predictions for multiple time series concurrently. In contrast, a univariate model trained on multiple time series implicitly models interactions between multiple time series and provides predictions for single time series concurrently. Multivariate models are typically computationally expensive and empirically do not necessarily offer better forecasting performance compared to using a univariate model.
4. **Forecast Type**: Direct forecast models are models that produce all steps in the forecast horizon at once. In contrast, recursive forecast models predict one-step ahead, and subsequently use the prediction to compute the next step in the forecast horizon, and so forth. Direct forecast models typically suffer less from bias and variance propagation as compared to recursive forecast models, whereas recursive models can be computationally less expensive.
5. **Exogenous**: Whether the model accepts exogenous variables. This can be exogenous variables that contain information about the past and future (F), about the past only (*historical*, H), or that contain static information (*static*, S).

Table 14.1: Table of neural forecasting methods available in NeuralForecast.

Model	Family	Univariate / Multivariate	Forecast Type	Exogenous
Autoformer	Transformer	Univariate	Direct	F
BiTCN	CNN	Univariate	Direct	F/H/S
DeepAR	RNN	Univariate	Recursive	F/S
DeepNPTS	MLP	Univariate	Direct	F/H/S
DilatedRNN	RNN	Univariate	Recursive	F/H/S
FEDformer	Transformer	Univariate	Direct	F
GRU	RNN	Univariate	Recursive	F/H/S
HINT	Any	Both	Both	F/H/S
Informer	Transformer	Multivariate	Direct	F
iTransformer	Transformer	Multivariate	Direct	-
KAN	KAN	Univariate	Direct	-
LSTM	RNN	Univariate	Recursive	F/H/S
MLP	MLP	Univariate	Direct	F/H/S
MLPMultivariate	MLP	Multivariate	Direct	F/H/S
NBEATS	MLP	Univariate	Direct	-
NBEATSx	MLP	Univariate	Direct	F/H/S
NHITS	MLP	Univariate	Direct	F/H/S
NLinear	MLP	Univariate	Direct	-
PatchTST	Transformer	Univariate	Direct	-
RNN	RNN	Univariate	Recursive	F/H/S
SOFTS	MLP	Multivariate	Direct	-
StemGNN	GNN	Multivariate	Direct	-
TCN	CNN	Univariate	Recursive	F/H/S
TFT	Transformer	Univariate	Direct	F/H/S
TiDE	MLP	Univariate	Direct	F/H/S
TimeMixer	MLP	Multivariate	Direct	-
TimeLLM	LLM	Univariate	Direct	-
TimesNet	CNN	Univariate	Direct	F
TSMixer	MLP	Multivariate	Direct	-
TSMixerx	MLP	Multivariate	Direct	F/H/S
VanillaTransformer	Transformer	Univariate	Direct	F

We turn to the `AirPassengers` dataset once again. Now, we train it using two more modern architectures, for example `NHITS` and `RNN`. We find that the results are comparable to the results we obtained using the relatively simple MLP architecture, which is due to the relative simplicity of the AirPassengers dataset (i.e., it doesn't require a very sophisticated model to get near perfect forecasting performance on this dataset).

```
test_size = 12
Y_train_df = AirPassengersPanel.query("ds.dt.year <= 1959")
Y_test_df = AirPassengersPanel.query("ds.dt.year > 1959").reset_index(
    drop=True
)  # 12 test

# Configure NHITS and RNN
models = [
    NHITS(h=12, input_size=24, scaler_type="robust"),
    RNN(h=12, input_size=24, scaler_type="robust"),
]

# Instantiate the NeuralForecast class
nf = NeuralForecast(models=models, freq="M")

# Fit the model to the training set
nf.fit(df=Y_train_df)

# Generate predictions
forecasts = nf.predict()
```

```
# Plot the predictions and the training data
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series(df=Y_train_df, forecasts_df=pd.concat([forecasts, Y_test_df["y"]], axis=1),
                  ax=axes, rm_legend=False)
for ax in fig.axes:
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='both')
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.suptitle("Number of passengers for different airlines")
fig.supylabel("Passengers")
fig.supxlabel("Month [1M]")
fig
```

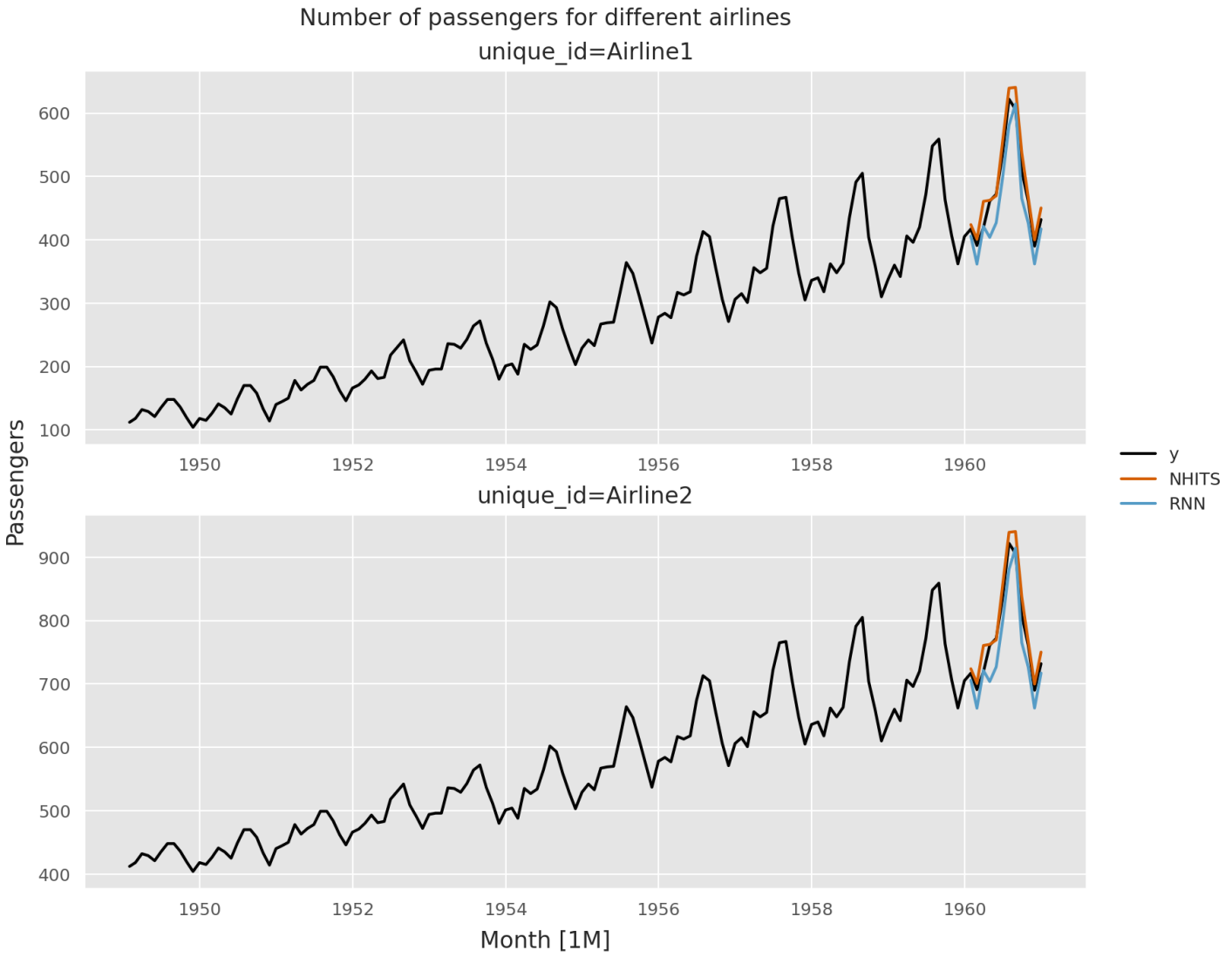

Figure 14.4: AirPassengers NHITS and RNN example.

14.4 Scaling the data

In the AirPassengers example, we included `scaler_type='robust'`. This setting controls how data gets scaled before entering the neural network. Scaling becomes necessary because input ranges can vary widely, which may cause performance issues during training with gradient descent. Common scaling options include:

- `identity`: no scaling
- `standard`: scaled by mean and standard deviation
- `robust`: a variation of standard scaling that handles outliers better. It removes the median and scales the data according to interquantile range (IQR). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Each technique scales time series and exogenous covariates based on the t values observed in the input window.

Let's look at our MLP example when using `identity` scaling (i.e., no scaling at all). We can visually verify that the forecast has worsened, and it seems there is a bias present in the forecast.

```
test_size = 12
Y_train_df = AirPassengersPanel.query("ds.dt.year <= 1959")
Y_test_df = AirPassengersPanel.query("ds.dt.year > 1959").reset_index(
    drop=True
)  # 12 test

# Configure the model
model = MLP(h=12, input_size=24, scaler_type="identity")

# Instantiate the forecast
nf = NeuralForecast(models=[model], freq="M")

# Fit the model to the training set
nf.fit(df=Y_train_df)

# Generate predictions
forecasts = nf.predict()
```

```
# Plot the predictions and the training data
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series(df=Y_train_df, forecasts_df=pd.concat([forecasts, Y_test_df["y"]], axis=1),
                  ax=axes, rm_legend=False)
for ax in fig.axes:
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='both')
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.suptitle("Number of passengers for different airlines")
fig.supylabel("Passengers")
fig.supxlabel("Month [1M]")
fig
```

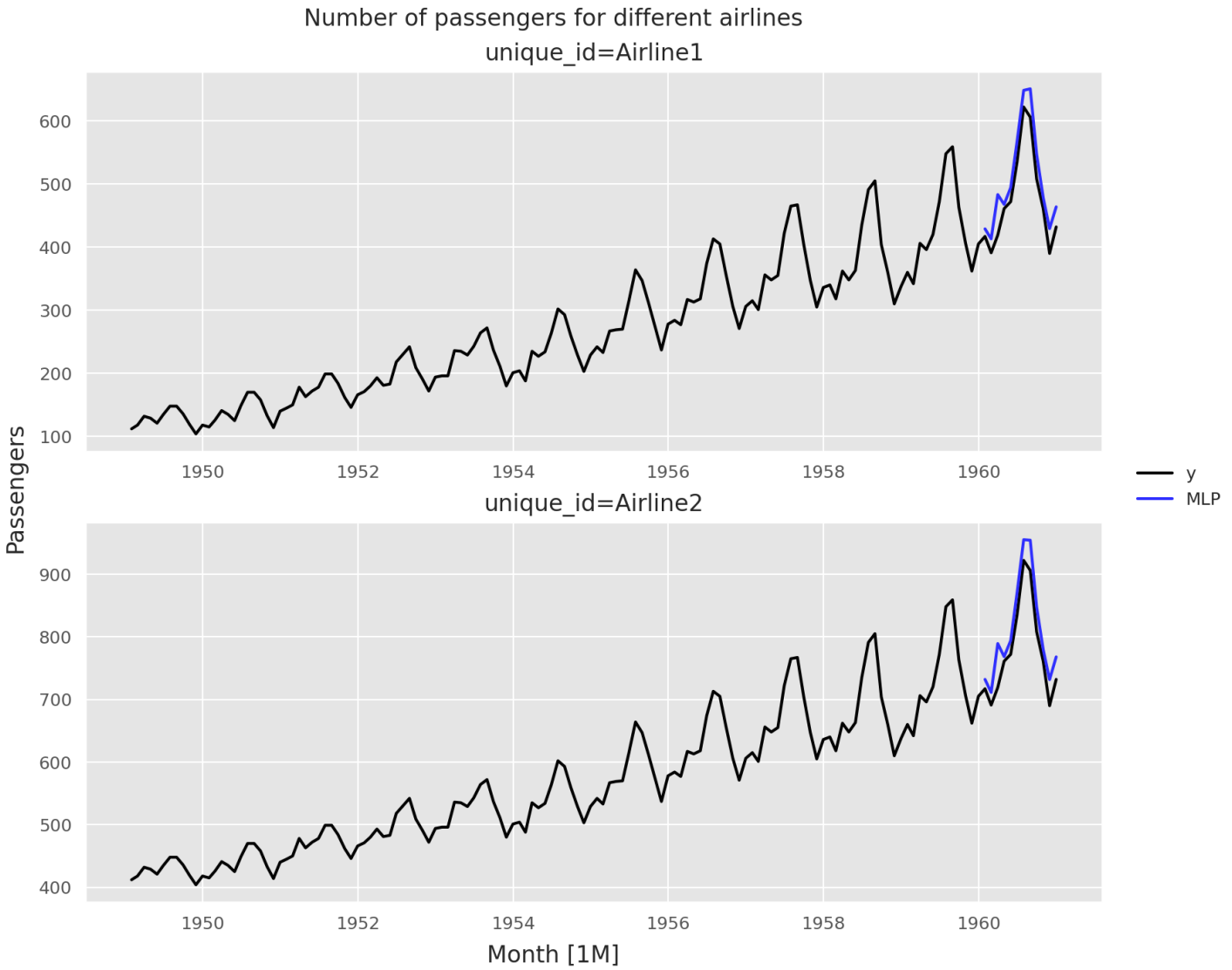

Figure 14.5: AirPassengers MLP example using identity scaling.

14.5 Optimization objectives

When forecasting with neural networks, we aim to learn a function f_\theta : X \mapsto Y, using a ground truth Y, such that we can generate forecasts \hat{Y}, i.e. f(X) = \hat{Y}. Ideally, there is no difference between the ground truth and the forecast: a perfect forecast without forecast error. Of course, with real data, this is impossible.

We try to minimise the difference between the ground truth and the forecast by using a *loss function*, which is our *optimization objective*. For example, we can define the *Mean Absolute Error*: \text{MAE}=\frac{1}{N} \sum_{i=0}^N |Y_i - \hat{Y}_i|

A perfect forecast has a zero MAE, and a very bad forecast a high MAE. Consequently, minimizing the MAE by adjusting the weights of our neural network will improve the forecasting performance.

There are many different optimization objectives. By default, *NeuralForecast* uses MAE as loss function. However, we can also use the *Mean Squared Error* (MSE): \text{MSE}=\frac{1}{N} \sum_{i=0}^N (Y_i - \hat{Y}_i)^2

The MSE is more susceptible to outliers as a result of the quadratic term and should commonly be used with scaled data. Let's see how our AirPassengers example looks like when using MSE as loss function.

Note that minimizing MSE leads to forecasts equal to the mean of the forecast distribution, while minimizing MAE leads to forecasts equal to the median of the forecast distribution.

```
test_size = 12
Y_train_df = AirPassengersPanel.query("ds.dt.year <= 1959")
Y_test_df = AirPassengersPanel.query("ds.dt.year > 1959").reset_index(
    drop=True
)  # 12 test

# Instantiate NHITS
model = NHITS(h=12, input_size=24, loss=MSE(), scaler_type="robust")

# Instantiate the forecast
nf = NeuralForecast(models=[model], freq="M")

# Fit the model to the training set
nf.fit(df=Y_train_df)

# Generate predictions
forecasts = nf.predict()
```

```
# Plot the predictions and the training data
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series(df=Y_train_df, forecasts_df=pd.concat([forecasts, Y_test_df["y"]], axis=1),
                  ax=axes, rm_legend=False)
for ax in fig.axes:
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='both')
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.suptitle("Number of passengers for different airlines")
fig.supylabel("Passengers")
fig.supxlabel("Month [1M]")
fig
```

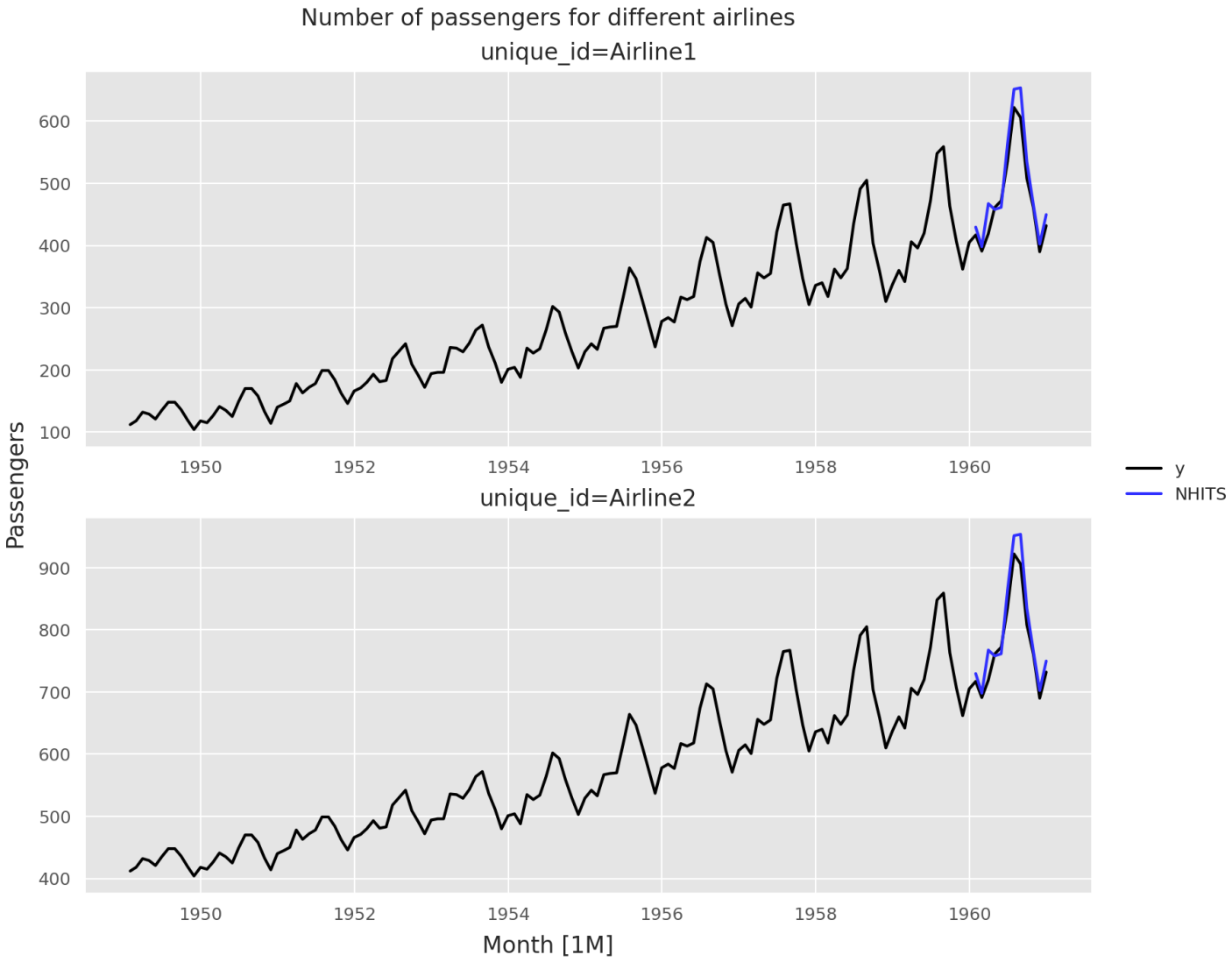

Figure 14.6: AirPassengers NHITS trained with MSE example.

Probabilistic optimization objectives

In the previous example, we showed how to generate *point predictions*. By changing the optimization objective however, we can also generate probabilisic forecasts, in order to model the full conditional distribution of our forecasting task (rather than only the median (using MAE as loss function) or the mean (using MSE as loss function)).

For example, we can assume that our forecast distribution follows a Gaussian ('Normal') distribution by specifying a `DistributionLoss`. As can be seen, this brings the benefit that we now have a notion of uncertainty around the forecast, indicated by the 80- and 90- percentile shaded regions in the plot.

```
test_size = 12
Y_train_df = AirPassengersPanel.query("ds.dt.year <= 1959")
Y_test_df = AirPassengersPanel.query("ds.dt.year > 1959").reset_index(
    drop=True
)  # 12 test

# Instantiate NHITS
model = NHITS(
    h=12,
    input_size=24,
    loss=DistributionLoss(distribution="Normal"),
    scaler_type="robust",
)

# Instantiate the forecast
nf = NeuralForecast(models=[model], freq="M")

# Fit the model to the training set
nf.fit(df=Y_train_df)

# Generate predictions
forecasts = nf.predict()
```

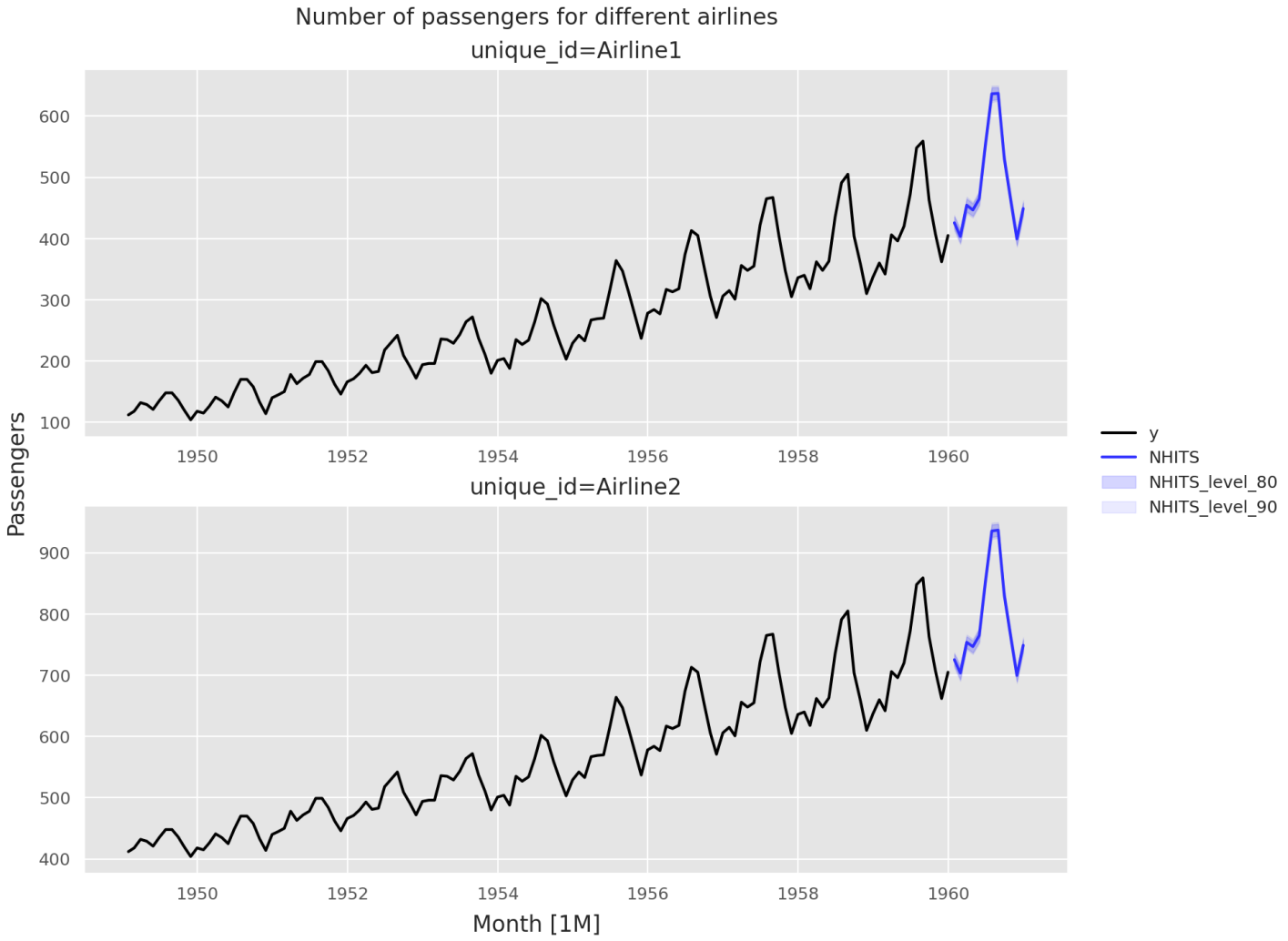

Figure 14.7: AirPassengers NHITS trained with Gaussian DistributionLoss.

The prediction intervals shown in Figure 14.7 are generated by NHITS, which directly predicts future values and their variances in a single pass since we are using `DistributionLoss(distribution="Normal")`. However, this approach can lead to underestimated prediction intervals when the model overfits the training data. Since the model learns distribution parameters based on the training data, narrow intervals may result from the model capturing patterns too closely without properly accounting for future uncertainty.

Unlike some traditional statistical models, where uncertainty increases over time through recursive forecasting, neural models like

NHITS rely on what they learn about the data's distribution during training. When the training data has relatively low variability, this can further contribute to narrow prediction intervals.

14.6 Exogenous variables

A key benefit of neural networks is that we can add all sorts of different inputs to the forecasting model. For example, rather than only the values of the time series themselves, we can also provide the model with other sources of information that may be beneficial for our forecast. These *exogenous covariates* can be typically categorized into three distinct types:

1. *Static covariates*: covariates that are static throughout the entire forecast horizon. For example, the identifiers `Airline1` and `Airline2` in our AirPassengers dataset.
2. *Historic covariates*: covariates for which only the historic data is available. For example, the outside temperature can be a historic covariate for a task of forecasting ice cream demand: we don't know the future outside temperature.
3. *Future covariates*: covariates for which we know their future values. For example, the day of the week can be used as a future covariate when trying to predict retail demand: we know the day of the week in advance when making forecasts for a particular day.

In NeuralForecast, each of these three types can be separately specified.

```
# Read the data
df = pd.read_csv("../data/EPF_FR_BE.csv", parse_dates=[1])
static_df = pd.read_csv("../data/EPF_FR_BE_static.csv")
futr_df = pd.read_csv("../data/EPF_FR_BE_futr.csv", parse_dates=[1])


# Specify the model
horizon = 24  # day-ahead daily forecast
model = BiTCN(
    h=horizon,
    input_size=5 * horizon,
    futr_exog_list=["gen_forecast", "week_day"],  # <- Future exogenous variables
    hist_exog_list=["system_load"],  # <- Historical exogenous variables
    stat_exog_list=["market_0", "market_1"],  # <- Static exogenous variables
    scaler_type="robust",
)

# Instantiate the forecast
nf = NeuralForecast(models=[model], freq="H")

# Fit the model to the training set
nf.fit(df=df, static_df=static_df)

# Generate predictions
forecasts = nf.predict(futr_df=futr_df)
```

```
# Plot the predictions and the training data
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series(df=df, forecasts_df=forecasts, max_insample_length=100, ax=axes, rm_legend=False

for ax in fig.axes:
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='both')
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.suptitle("Electricity price for different markets")
fig.supylabel("Price")
fig.supxlabel("Hour [1H]")
fig
```

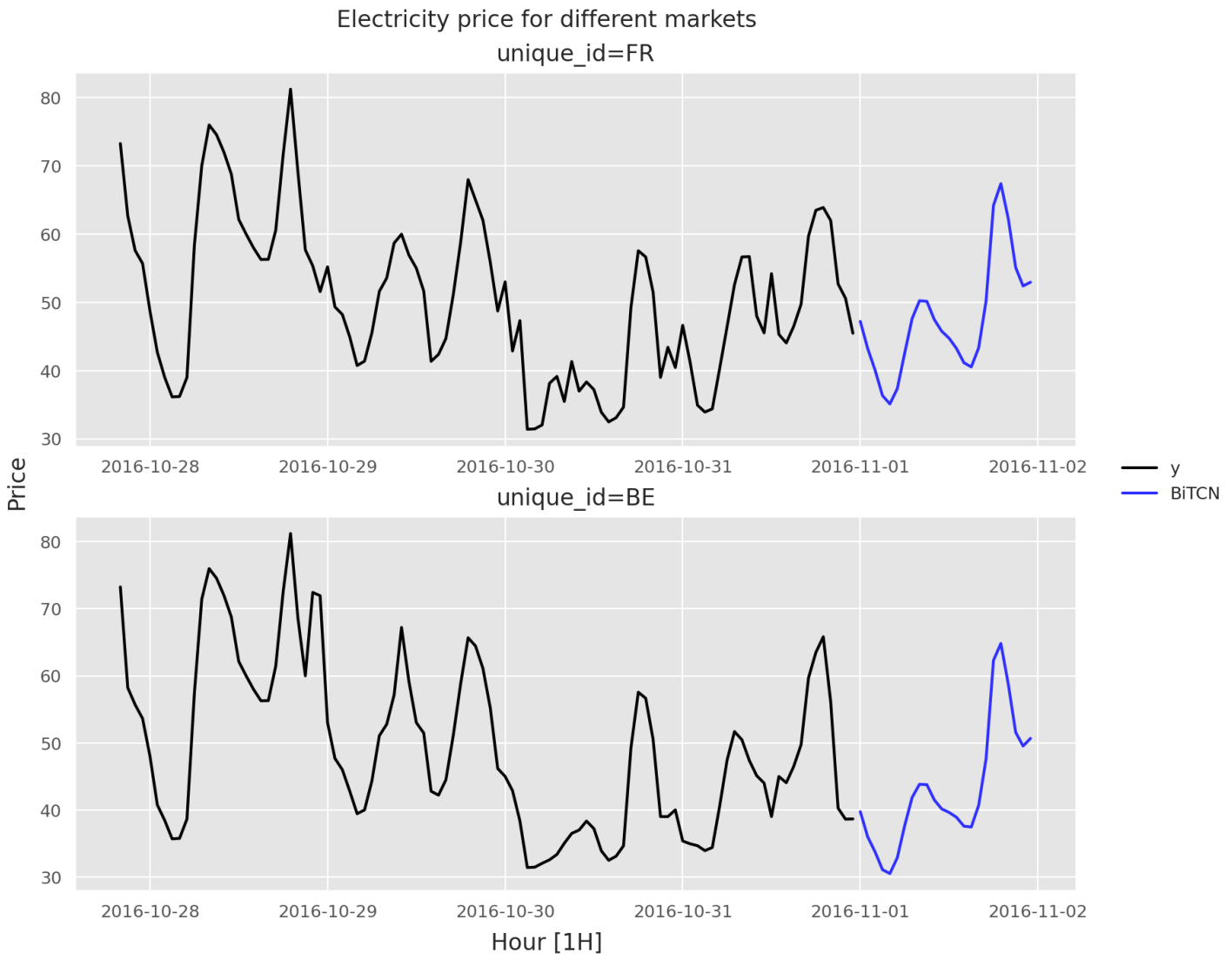

Figure 14.8: BiTCN trained with exogenous variables.

14.7 Hyperparameter optimization

Forecasting with neural networks often requires a practitioner to choose certain settings for training the forecasting model. These settings are commonly referred to as hyperparameters. We've seen a few of them already:

- the number of hidden layers and number of nodes in an MLP;
- the optimization objective;

- the input size (i.e., the amount of historic timesteps from a timeseries that the neural network considers for creating the forecast);
- the scaler;
- and so forth.

Ideally, a practitioner doesn't have to choose these settings herself, but rather have an optimization algorithm pick them. To this end, we can perform *hyperparameter optimization*. *NeuralForecast* can perform the hyperparameter optimization automatically based on a default configuration. To do so, it makes use of *Auto* models, which are versions of the models from Table 14.1 that allow to automatically optimize hyperparameters of the neural network. For a specified number of iterations, an *Auto* model will intelligently select a set of hyperparameters from the eligible range and evaluate these on a validation set. At the end of the training process, the best performing hyperparameters are returned and can be used for the downstream forecasting application.

```
test_size = 12
Y_train_df = AirPassengersPanel.query("ds.dt.year <= 1959")
Y_test_df = AirPassengersPanel.query("ds.dt.year > 1959").reset_index(
    drop=True
)  # 12 test

# Get the default hyperparameter range for NHITS. Update random_seed and n_pool_kernel_size with a

nhits_config = AutoNHITS.get_default_config(h=12, backend="ray")
nhits_config["random_seed"] = tune.randint(1, 10)
nhits_config["n_pool_kernel_size"] = tune.choice([[2, 2, 2], [16, 8, 1]])

# Instantiate the AutoNHITS model with the chosen configuration.
# We optimize for 10 steps, i.e. we evaluate 10 different hyperparameter configurations.
model = AutoNHITS(h=12, num_samples=1)

# Instantiate the forecast
nf = NeuralForecast(models=[model], freq="M")

# Fit the model to the training set
nf.fit(df=Y_train_df)

# Generate predictions
forecasts = nf.predict()
```

```
# Plot the predictions and the training data
_, axes = plt.subplots(nrows=2, ncols=1)
fig = plot_series(df=Y_train_df, forecasts_df=pd.concat([forecasts, Y_test_df["y"]], axis=1), ax=a

            rm_legend=False)
for ax in fig.axes:
    ax.set_xlabel("")
    ax.set_ylabel("")
    ax.tick_params(axis='both')
    for label in ax.get_xticklabels():
        label.set_rotation(0)
fig.suptitle("Number of passengers for different airlines")
fig.supylabel("Passengers")
fig.supxlabel("Month [1M]")
fig
```

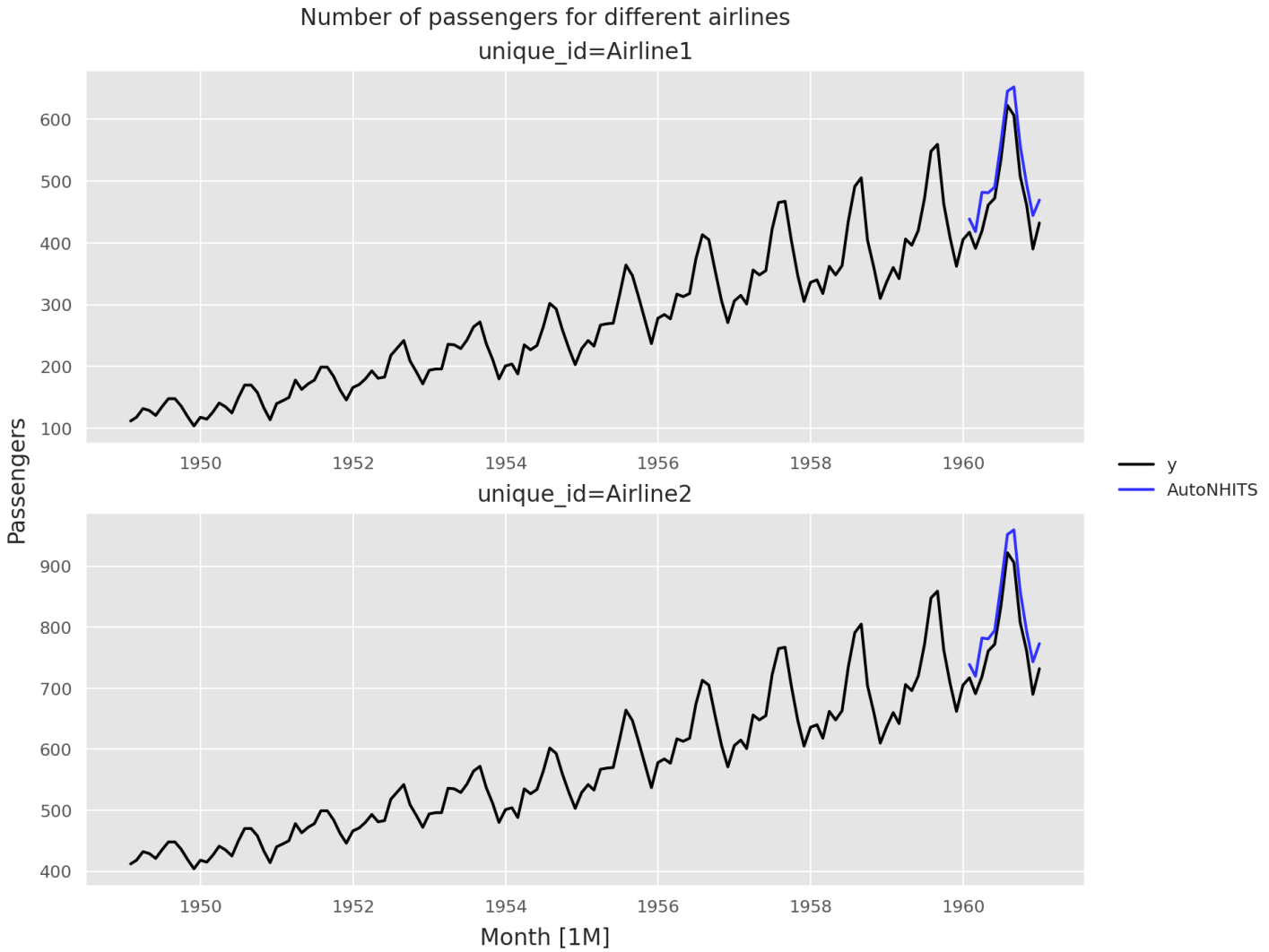

Figure 14.9: AirPassengers NHITS with automatic hyperparameter optimization.

14.8 Exercises

1. Use the LSTM model to forecast 24-steps ahead for all the series in the `pedestrian` data set.
2. Consider the weekly data on US finished motor gasoline products supplied (millions of barrels per day) (series `us_gasoline`).
    a. Use the MLP model to forecast the next 13 weeks of data.
    b. Repeat the previous exercise using the AutoMLP and the AutoNHITS models from NeuralForecast.
    c. Compare the accuracy of the models you generated.
    d. Compare the accuracy of the models against the models of Chapter 12, Exercise 2.

14.9 Further reading

- Bengio, Courville, and Vincent (2013) provides an overview of representation learning.
- Mariet and Kuznetsov (2019) for a more formal introduction to sequence-to-sequence modeling for time series.
- Hewamalage, Bergmeir, and Bandara (2021) for a review on recurrent neural networks.

14.10 Used modules and classes

NeuralForecast

- `NeuralForecast` class - Core forecasting engine for neural network models
- `MLP` model - Basic multilayer perceptron architecture
- `NHITS` model - Neural hierarchical interpolation for time series
- `RNN` model - Recurrent neural network architecture
- `AutoNHITS` model - Automatic hyperparameter optimization for NHITS

UtilsForecast

- `plot_series` function - For visualizing time series data and forecasts

DatasetsForecast

- `AirPassengersPanel` dataset - Classic airline passengers dataset for examples

← Chapter 13 Some practical forecasting issues

Chapter 15 Foundation forecasting models →