DI Squared

White Paper

Time-Series Forecasting

Time-series forecasting transforms historical patterns into actionable predictions by recognizing temporal structure – enabling organizations to anticipate demand, optimize resources, and mitigate risks across operations. The critical challenge lies in balancing model accuracy (predictive power) with interpretability (explainability), as high-performing neural networks often operate as "black boxes" while transparent statistical models may sacrifice precision.

This whitepaper explores both statistical frameworks (ARIMA/SARIMAX) and advanced machine learning approaches (N-BEATS, TCN, BlockRNN), providing practical guidance on selecting and deploying the right forecasting strategy for your business context.

Author: Sumana Reddy, Associate Consultant
Date: November 2025

Table of Contents

- The accuracy vs. interpretability trade-off: Neural networks excel at accuracy but lack transparency; statistical models are interpretable but limited in complexity.
- Stationarity is critical: Time series requires stable statistical properties; differencing transforms non-stationary data.
- SARIMAX for transparency: Achieved 3.04% MAPE on electricity data with clear, explainable seasonal components.
- DARTS: Unified framework for comparing N-BEATS, TCN, and BlockRNN models.
- Covariates improve performance: Past and future covariates can improve accuracy by 30-50%.
- Back-testing validates reliability: Rolling-window validation achieved ~12% average deviation – reasonable for production.
- Business impact first: Success depends on actionability, not just improving metrics.

Overview

In this white paper

Learn the different forecasting problems relating to interpretability of models, maintaining accuracy, and finding the right balance between them. In doing so, we will do a deep dive into Statistical/Traditional models as well as give an overview on how to use DARTS.

Implementation Guidance

- Three comparison tables showing how business applications in retail, financial services, manufacturing, and utilities should approach forecasting.
- Guidance on when to use each. model, including a decision matrix for SARIMAX vs. N-BEATS vs. BlockRNN vs. Linear Regression.
- DI Squared's 7-step framework from "start simple" through "monitor in production"

Metrics and Thresholds

- MAPE benchmarks (industry baseline 8-12%)
- Statistical validation tests (Ljung-Box, AIC/BIC)
- Actionable recommendations for model selection

Time Series: A Deep Dive

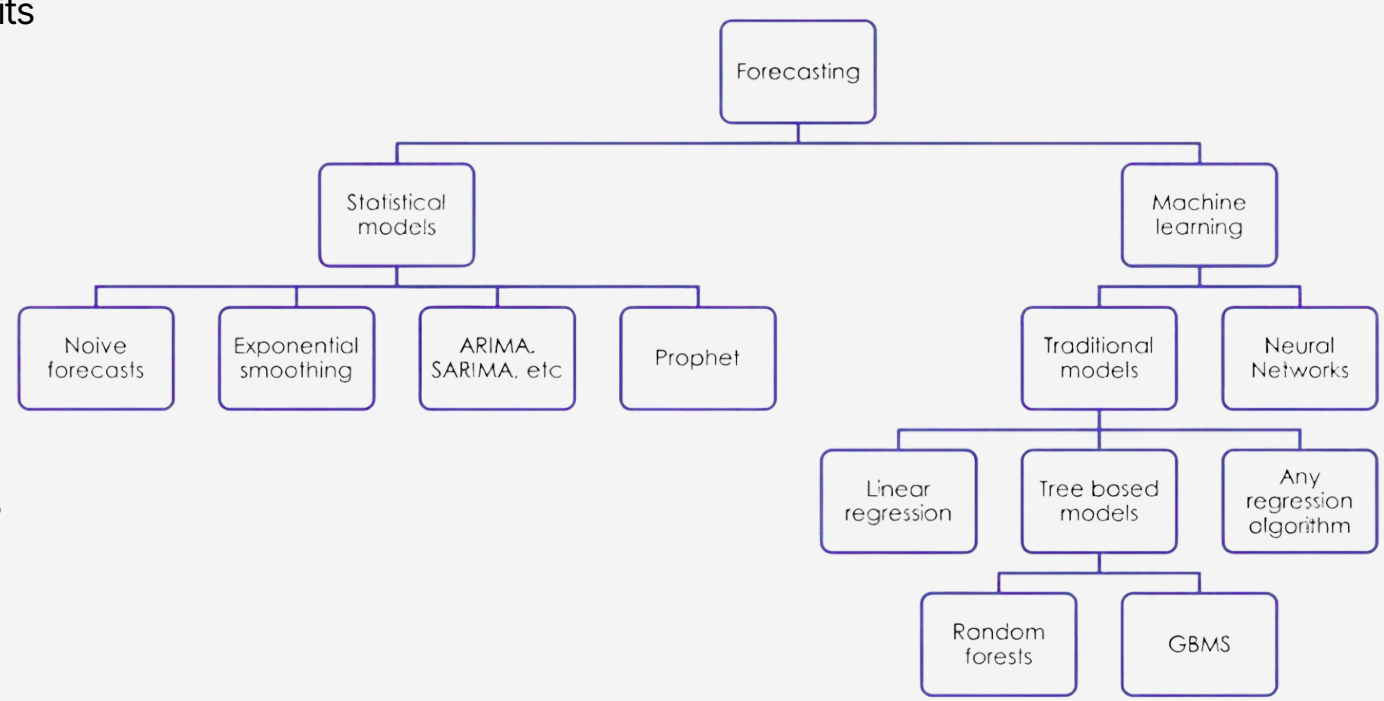
Types of Forecasting Models

Time series forecasting is the art of turning history into foresight. Instead of treating data as a static snapshot, it recognizes the temporal structure — the fact that yesterday's behavior often shapes tomorrow's outcome.

The power lies in identifying patterns that repeat, we know for a fact that time series is defined by its decomposition into the four components:

- Seasonality: Regular cycles (ice cream sales peaking in summer, tissue sales peaking in emergencies, etc.)
- Cyclical patterns: Irregular but recurring fluctuations (economic cycles)
- Noise: Random variations that don't follow patterns
- Modern businesses generate time series data constantly: website traffic, sales figures, sensor readings, user engagement metrics, and financial indicators.

For instance: If regular data analysis is like taking a photograph, time series analysis is like watching a movie. You're not just seeing what it is, but understanding how things change, trend, and cycle over time. This temporal dimension is the foundation for time series.



Time Series: A Deep Dive

Accuracy vs Interpretability

A central challenge in time series modeling is balancing predictive accuracy with interpretability. Advanced methods such as gradient-boosted trees and recurrent neural networks can capture complex nonlinear behaviors, interactions, and seasonal effects with high precision. However, domain experts often need models that provide transparent reasoning to validate forecasts, diagnose errors, and foster trust. This need for interpretability has grown increasingly important as machine learning plays a larger role in shaping high-stakes business decisions.

In forecasting, the ideal model would combine accuracy, interpretability, and robustness to handle complex real-world data.

Traditional statistical models such as Autoregressive Integrated Moving Average (ARIMA) and exponential smoothing (ETS) are valued for their clear mathematical foundations, but they tend to work well on simpler, stationary series. Their limited ability to incorporate exogenous variables often restricts accuracy.

This tradeoff shows up in many business scenarios.

Retailers need precise demand forecasts to avoid stockouts or overstocking but also need explainable outputs to justify inventory decisions to executives.

Financial institutions rely on forecasts of interest rates, credit demand, and risk exposure. Here, a black-box model might outperform in accuracy but could fail regulatory requirements for explainability.

Manufacturers and supply chain planners must anticipate capacity needs. When forecasts miss the mark, production schedules and supplier contracts are affected — leaders need clarity on why the model is projecting a spike or drop.

From this tension, two main directions have emerged.

One direction has emphasized preserving a statistical framework to maintain transparency and interpretability. Methods like state-space models and decomposition-based approaches (e.g., STL or Prophet) break time series into understandable components or introduce external factors in structured ways.

The second direction prioritizes maximum predictive performance, often at the cost of interpretability. Here, machine learning and neural network models dominate, leveraging large datasets and capturing nonlinear dependencies that classical models cannot.

These methods can achieve remarkable accuracy but are often criticized as “black boxes,” where the lack of interpretability hinders trust, adoption, and explainability in practice.

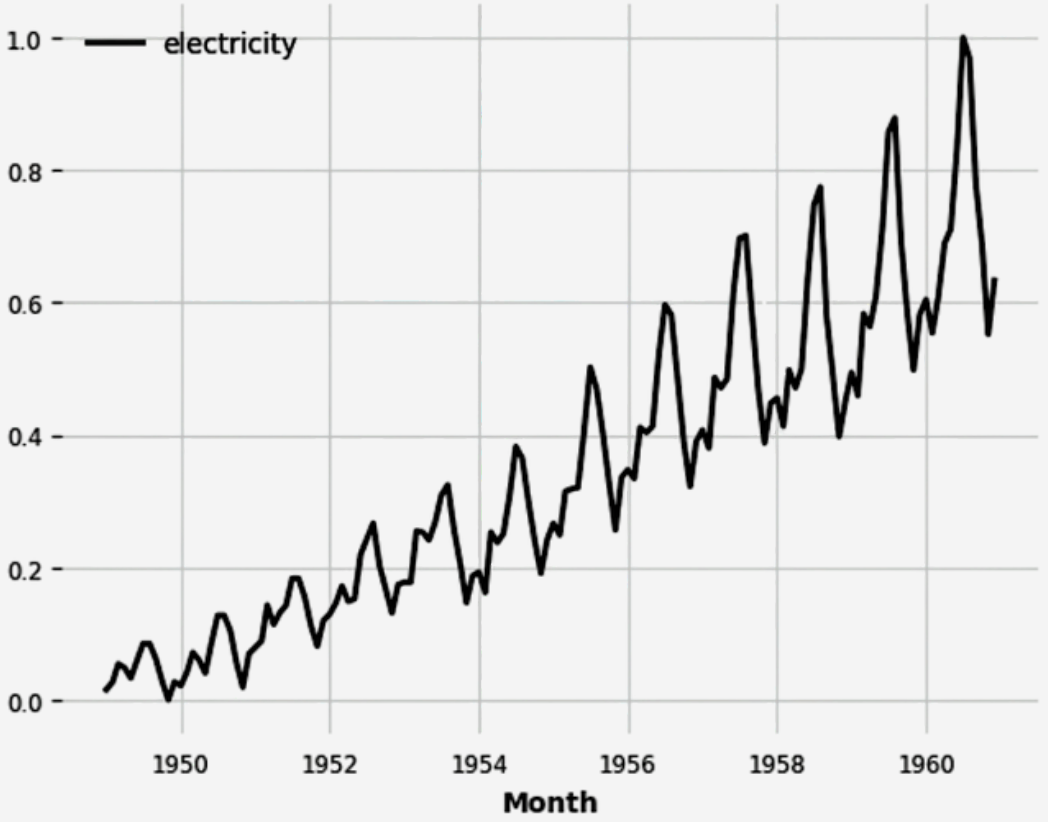
Time Series: A Deep Dive

Interpretability

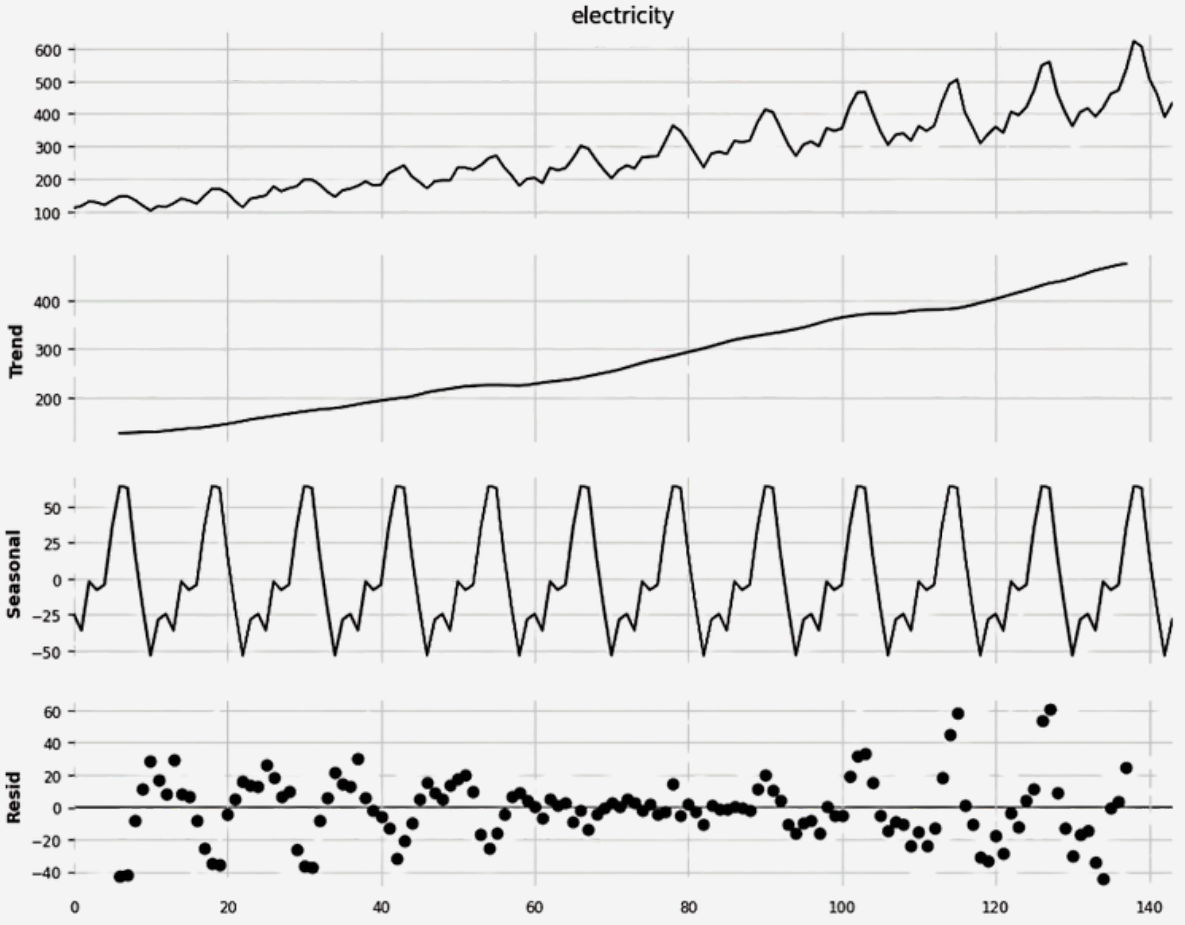
The foundation for statistical time series data analysis is stationarity (a shift in time doesn't cause a change in the shape of the distribution).

In addition to that, a good understanding of the seasonal components involved forms a basis for coming up with good parameters and exogenous features that can help in forecast accuracy.

Consider the electricity consumption dataset to the right and on the next page that describes the total electricity consumed over a period. The units are typically measured in kilowatt hours (kWh) or billion kilowatt hours (BkWh). Historical data provides monthly observations spanning multiple decades (1949 to 1960 for this example).



Examining the data, we see the trend and seasonality appear broadly as expected, but the residuals (the difference between the model's fitted value and the actual value) show inconsistent behavior over time—the magnitude of fluctuations in the middle of the series differs noticeably from the beginning and the end.

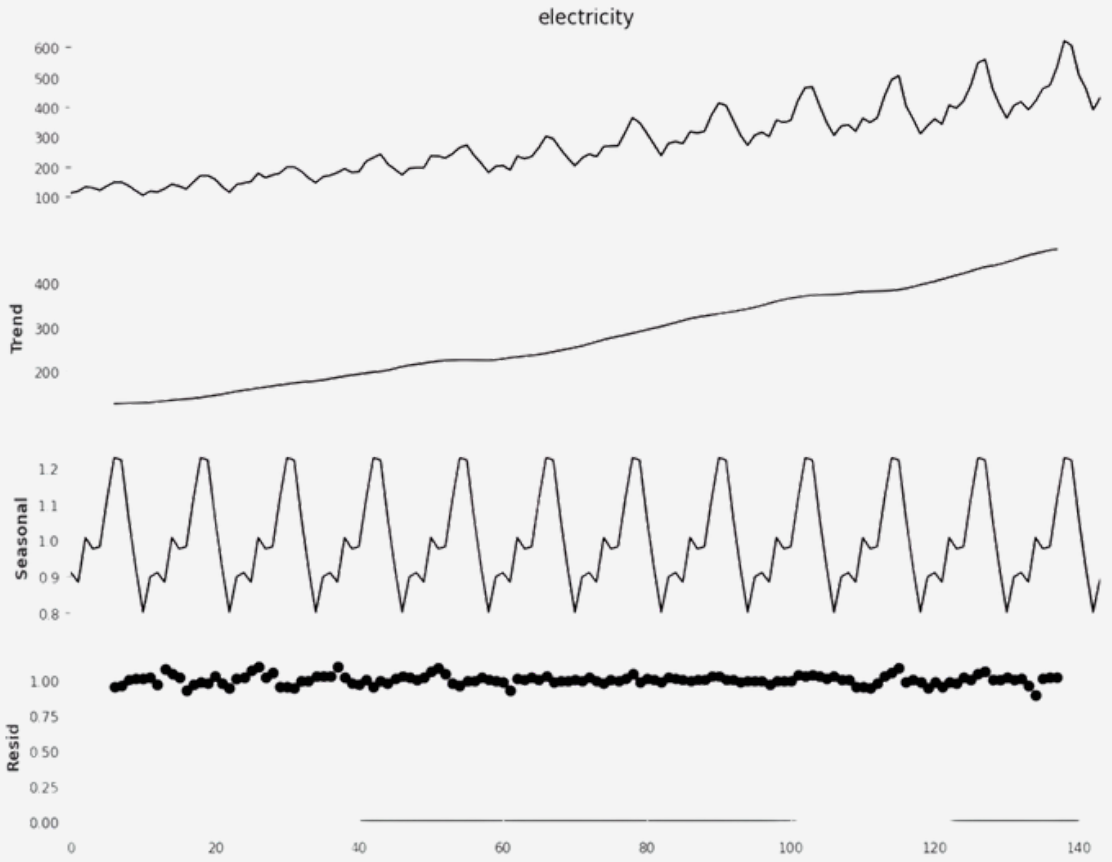


```
# decomposition
import statsmodels.api as sm
decomposition = sm.tsa.seasonal_decompose(series[“electricity”],period =12)
figure = decomposition.plot().set_size_inches(10, 8)
plt.show()
```

By default, satsmodels seasonal_decompose assumes an additive structure, whereas the data may follow a multiplicative relationship between its components. This is something we can test and compare.

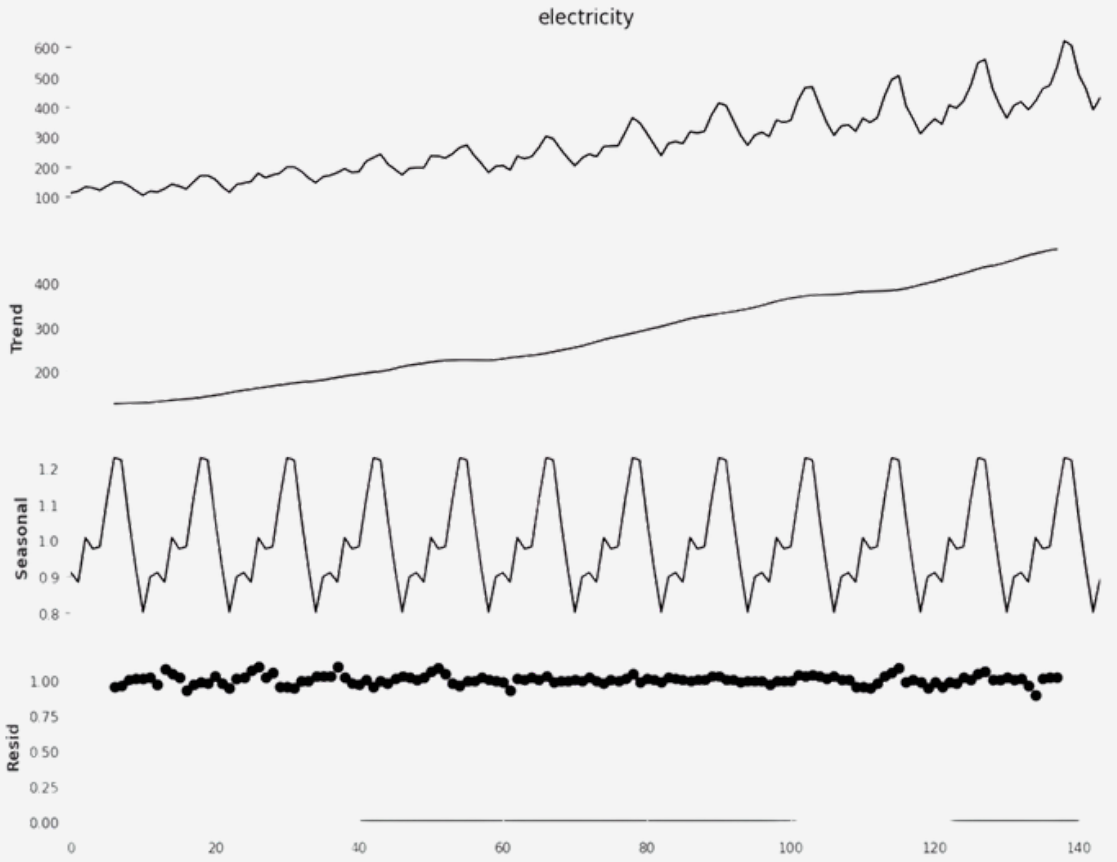
```
decomposition = sm.tsa.seasonal_decompose(series["electricity"],period =12, model = 'multiplicative')
figure = decomposition.plot().set_size_inches(10, 8)
plt.show()
```

The trend and seasonality patterns remain consistent, but the residuals now look more balanced around a constant average. This doesn't automatically mean the series is fully stable (stationary), but it does indicate the random fluctuations are more even over time, with no clear signs of increasing or decreasing variability. This matters because it increases confidence that the forecast isn't being distorted by uneven random fluctuations – making results more reliable and easier to interpret.



In general residuals are useful in checking whether a model has adequately captured the information in the data. A good forecasting method will yield residuals with the following properties:

- The residuals are uncorrelated. If there are correlations between residuals, then there is information left in the residuals which should be used in computing forecasts.
- The residuals have zero mean. If the residuals have a mean other than zero, then the forecasts are biased.
- The residuals have constant variance.
- The residuals are normally distributed.



Stationarity and Why it Matters

Most statistical forecasting methods work best with stationary data (statistical properties remain stable over time). If the series is non-stationary (e.g., has trends or shifting variability), the model must account for extra moving parts, making predictions less reliable. Stationarity also reduces the risk of concept drift, where the patterns we train on no longer match what happens in the future.

Without stationarity, it's like playing a game on uneven ground. If the field is sloped and bumpy, every move is tricky and unpredictable, but if we first level the field, the game's more straightforward.

That's what stationarity does: it levels the playing field so our forecasting models can play by clear, consistent rules.

Stationarity and Why it Matters

One of the simplest ways to turn a non-stationary time series into a stationary one is through differencing. We do this by subtracting each value from the previous value, much like lag.

Differencing at lag 1 is best thought of as discreet counterpart to differentiation: first derivative of a linear function is flat, first derivative of a quadratic function is linear etc.

So if we want to get rid of a trend behaving like polynomial of degree of n, we need to apply the differencing operator n times.

Test for Stationarity

The Augmented Dickey-Fuller (ADF) test checks whether a time series is stationary by testing for a unit root.

- Null hypothesis (H0): A unit root exists → the series is non-stationary.
- Alternative hypothesis (H1): No unit root → the series is stationary.

A unit root means a time series is highly persistent — its current value is heavily dependent on its past value — making the series non-stationary.

Consider the AR(1) equation:

$$X_t = \rho X_{t-1} + \varepsilon_t$$

A series with a unit root remembers every shock — if consumption jumps once, that effect carries forward, creating a drifting or trending pattern. A stationary series has no unit root and forgets shocks over time: if sales spike one month, they eventually drift back to normal; it self-corrects concept drift.

```
from statsmodels.tsa.stattools import adfuller
adf_result = adfuller(series.electricity.values)
print('ADF Statistic: %f' % adf_result[0])
print('p-value: %f' % adf_result[1])
print('Critical Values:')
for key, value in adf_result[4].items():
    print('\t%s: %.3f' % (key, value))
```

```
ADF Statistic: 0.815369
p-value: 0.991880
Critical Values:
        1%: -3.482
        5%: -2.884
        10%: -2.579
```

unit root = non-stationarity

The smaller the p-value (the p-value measures the probability of seeing the effect when the null hypothesis is true) the effect of the unit root is less likely to be seen, and the more likely it is our series is stationary. Here our p-value is 0.99, which is quite high. If we use a 5% Critical Value (CV), this series would be considered stationary. But as we just visually found an upward trend, it's common to adopt a stricter criterion (e.g. using 1% CV). Here, both the p-value, upward trend and seasonality strongly indicate the series is non-stationary, regardless of whether we use the 5% or 1% level.

```
from statsmodels.tsa.stattools import adfuller
adf_result = adfuller(series.electricity.values)
print('ADF Statistic: %f' % adf_result[0])
print('p-value: %f' % adf_result[1])
print('Critical Values:')
for key, value in adf_result[4].items():
    print('\t%s: %.3f' % (key, value))
```

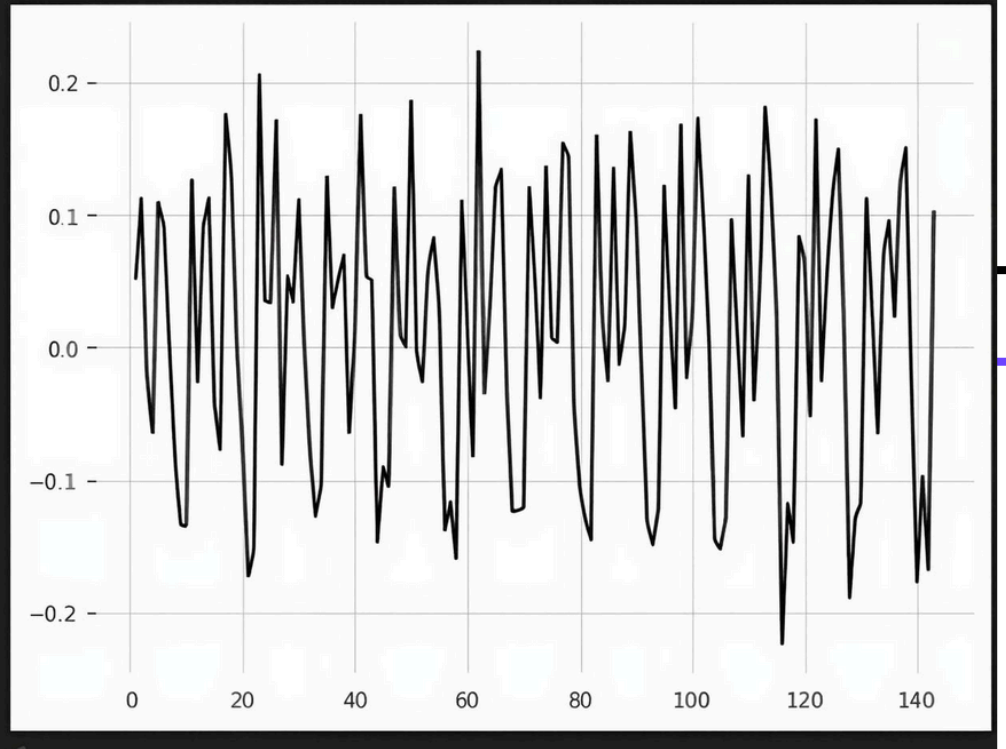
```
ADF Statistic: 0.815369
p-value: 0.991880
Critical Values:
        1%: -3.482
        5%: -2.884
        10%: -2.579
```

In addition to p-value, since the ADF statistic (0.815) > critical values, we fail to reject the null hypothesis, indicating our series is non-stationary.

Now that we know how strongly non-stationary our series is, let's work to stationarize the series.

```
series['electricity4'] = series['electricity'].apply(np.log).diff()
series.electricity4.plot()
plt.show()

result = adfuller(series.electricity4[10:])
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
```

✓ 0.1s




```
ADF Statistic: -3.086110
p-value: 0.027598
```

Modeling: A Quick Intro

Autoregressive Integrated Moving Average (ARIMA):

Works well for short-term forecasting on stationary or made stationary (via differencing) series.

ARIMA is the combination of 3 components:

- AR (AutoRegressive): Uses past values of the series.
- I (Integrated): Applies differencing to remove trends/non-stationarity.
- MA (Moving Average): Models the error terms as a linear combination of past errors.

The SARIMAX model expands on the ARIMA model by adding seasonal parameters to the AR, I, and MA components. The seasonal component repeats at a defined seasonal period (e.g., monthly, yearly). The X in SARIMAX only indicates a multi-variate model.

ARIMA(p, d, q)(P, D, Q, s)
Lowercase = non-seasonal parts,
Uppercase = seasonal parts,
s = season length.

SARIMAX works well for time series with strong seasonal patterns (monthly sales, daily traffic, quarterly demand), so we'll be using it for our example shown on the next page.

We're going to do a naive grid search using a "for" loop to find the best p,q, P, Q combinations.

The parameters that gave the lowest AIC are (p,q,P,Q) = (2,1,1,2)

```python
def optimize_SARIMA(endog: Union[pd.Series, list], order_list: list, d: int, D: int, s: int) -> pd.DataFrame:

    results = []

    for order in tqdm(order_list):
        try:
            model = SARIMAX(
                endog,
                order=(order[0], d, order[1]),
                seasonal_order=(order[2], D, order[3], s),
                simple_differencing=False).fit(disp=False)
        except:
            continue

        aic = model.aic
        results.append([order, aic])

    result_df = pd.DataFrame(results)
    result_df.columns = ['(p,q,P,Q)', 'AIC']

    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC', ascending=True).reset_index(drop=True)

    return result_df

ps, qs, Ps, Qs = range(0, 4, 1), range(0, 4, 1), range(0, 4, 1), range(0, 4, 1)
SARIMA_order_list = list(product(ps, qs, Ps, Qs))
train = series['electricity4'][:-12]

d = 1
D = 1
s = 12

SARIMA_result_df = optimize_SARIMA(train, SARIMA_order_list, d, D, s)
SARIMA_result_df
```

```
100%|██████████| 256/256 [10:11<00:00,  2.39s/it]
```

SARIMAX summary

```
SARIMA_model = SARIMAX(train, order=(2,1,1), seasonal_order=(1,1,2,12), simple_differencing=False)
SARIMA_model_fit = SARIMA_model.fit(disp=False)

print(SARIMA_model_fit.summary())
```

19] ✓ 5.5s

```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                          electricity4   No. Observations:                  132
Model:             SARIMAX(2, 1, 1)x(1, 1, [1, 2], 12)   Log Likelihood                190.307
Date:                                 Fri, 24 Oct 2025   AIC                           -366.614
Time:                                         15:11:28   BIC                           -347.160
Sample:                                              0   HQIC                          -358.714
                                                   - 132
Covariance Type:                                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.6923      0.128     -5.395      0.000      -0.944      -0.441
ar.L2          0.3031      0.127      2.386      0.017       0.054       0.552
ma.L1         -0.9989      1.422     -0.702      0.482      -3.786       1.789
ar.S.L12       0.9191      0.137      6.720      0.000       0.651       1.187
ma.S.L12      -1.8526      0.562     -3.296      0.001      -2.954      -0.751
ma.S.L24       0.9528      0.517      1.841      0.066      -0.061       1.967
sigma2         0.0014      0.003      0.536      0.592      -0.004       0.007
==========================================================================
Ljung-Box (L1) (Q):                   1.62   Jarque-Bera (JB):                 1.71
Prob(Q):                              0.20   Prob(JB):                         0.43
Heteroskedasticity (H):               0.45   Skew:                             0.20
Prob(H) (two-sided):                  0.01   Kurtosis:                         3.42
==========================================================================
```

Log Likelihood (measures how well a model explains the observed data) = 190.3

Higher is better (relative to other models). A higher value means the model fits the data better, because it thinks the observed outcomes were more likely under its assumptions.

AIC = -366.6, BIC = -347.2

Lower values of AIC and BIC indicate a better fit while balancing complexity. Our model fits very well, shown by how strong the negative AIC is.

Coefficients

AR terms

- *ar.L1 = -0.6923 (p < 0.001)* → Significant. Last lag has a strong negative effect.
- *ar.L2 = 0.3031 (p = 0.017)* → Significant. The 2nd lag has a positive effect.

MA term

- *ma.L1 = -0.9989 (p = 0.482)* → Not significant. The moving average component at lag 1 may not be important.

Seasonal Components

- *ar.S.L12 = 0.9191 (p < 0.001)* → Highly significant. The seasonal AR at lag 12 (yearly) strongly contributes.
- *ma.S.L12 = -1.8526 (p = 0.001)* → Significant. Seasonal MA at lag 12 has a big negative influence
- *ma.S.L24 = 0.9528 (p = 0.066)* → Borderline significant (~10% level).

sigma² = 0.0014 (p = 0.59)

- The estimated error variance is low, and not statistically different from zero → suggests residual noise is small.

Diagnostics

Ljung-Box test (p = 0.20): Residuals show no autocorrelation, which is good.
Jarque-Bera (p = 0.43): Residuals are normally distributed, which is good.

Modeling summary

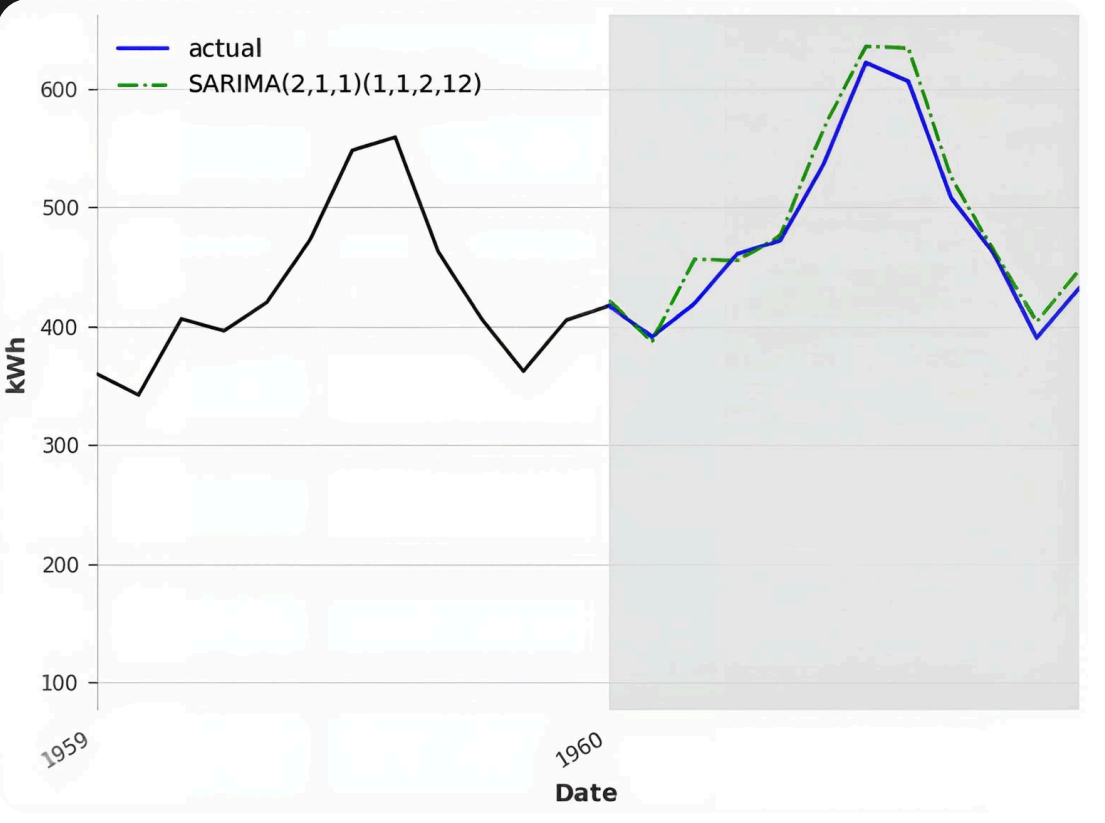
The model captures both the trend and strong yearly seasonality (lag 12). Most seasonal terms are significant; confirming seasonality is key. Diagnostics (Ljung-Box, JB) suggest residuals are well-behaved, and the model is statistically sound.

Slight heteroskedasticity means variance isn't constant, but it may not hurt forecasts much. Our SARIMAX model fits the data well, confirms strong seasonal structure at 12 months, and leaves mostly white noise in residuals.

Let's look at mape to judge how good the predictions are:

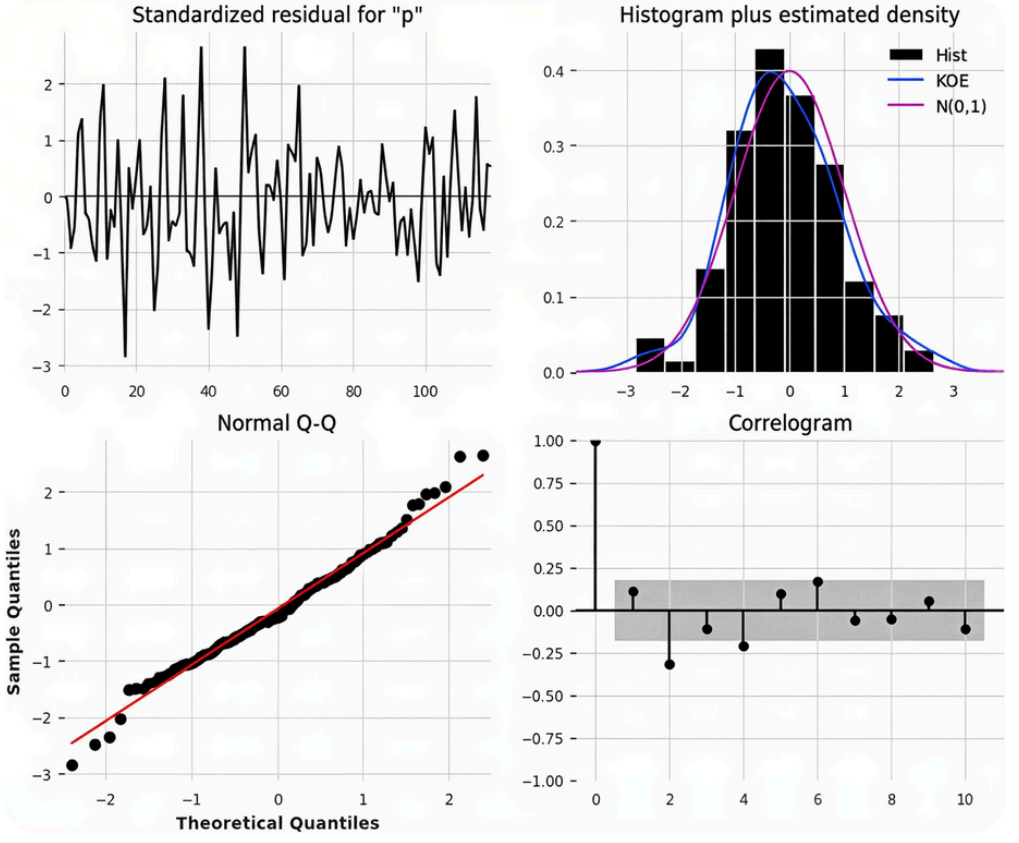
```python
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

mape_SARIMA = mape(test['electricity'], test['SARIMA_pred'])
print(f'MAPE SARIMA: {mape_SARIMA}')
```

```
MAPE SARIMA: 3.0383836601925447
```



Residuals

Before we finalize the model, it's important to check the residuals, to ensure our model is not ignoring any patterns that are yet to be forecasted.



- The top-left plot shows the residuals over time.
- The top-right plot shows the distribution of the residuals, which is close to a normal distribution, which is ideal.
- The bottom-left plot (Q-Q plot) displays a line that is fairly straight, indicating that the residuals are normally distributed.
- The bottom-right plot shows the autocorrelation of the residuals, which is close to 0 for all lags, indicating that the residuals are uncorrelated.

Therefore, from a qualitative standpoint, it seems that our residuals are close to “white noise”, indicating that the model errors are random, and are not meaningful.

So far, we’ve focused on modeling Interpretability and the different methods we can use to ensure we are not creating black boxes for those relying on the results. The other side of the time series “equation” that we need to balance is Accuracy.

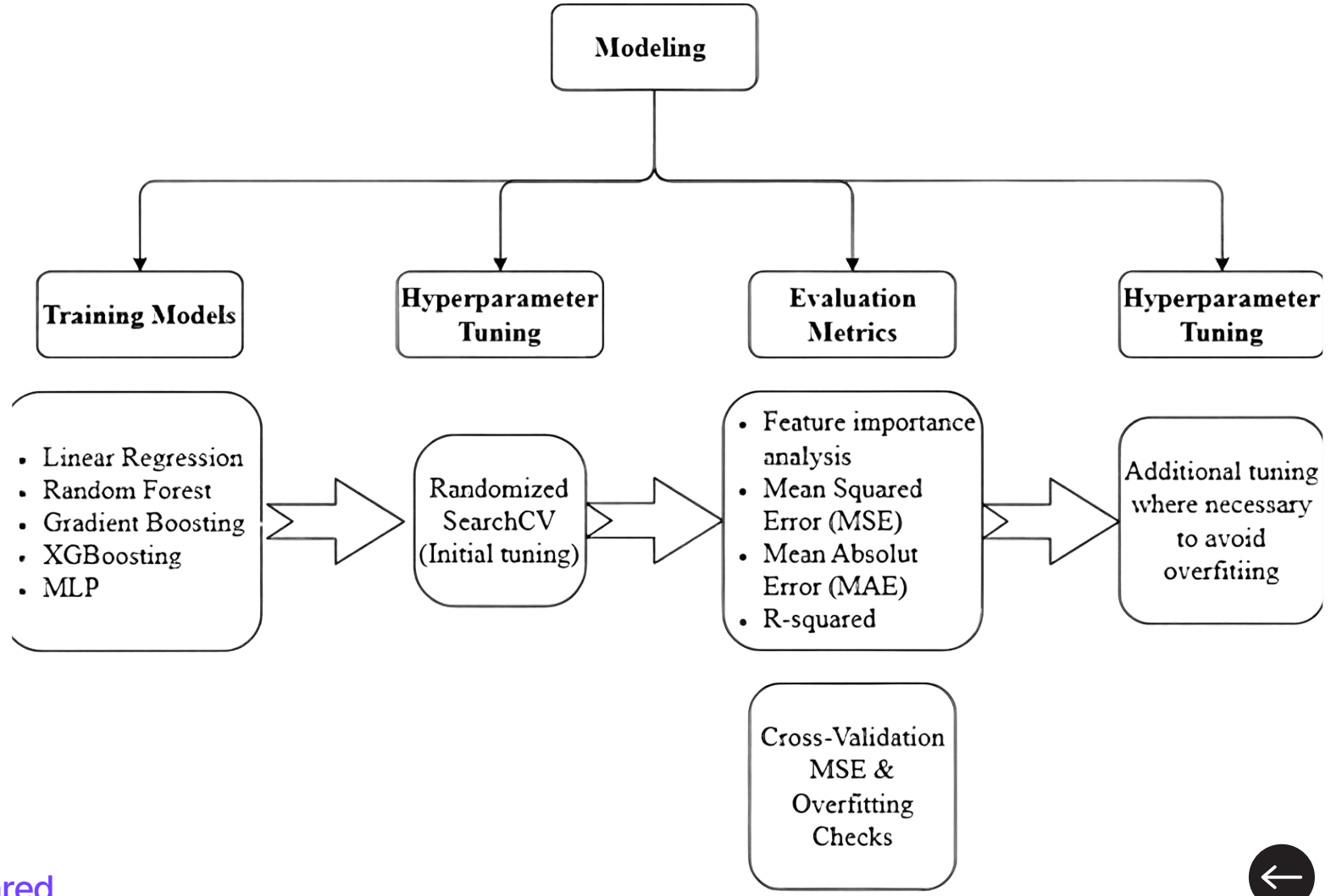
Time Series Forecasting Accuracy

Accurate time series forecasting is critical for planning and decision-making in areas like energy demand, sales, finance, and operations. In this section, we explore how the DARTS library in Python makes it easier to build, compare, and evaluate a range of modern forecasting models. In addition, we'll also see how incorporating covariates and performing rigorous back-testing generates both reliable and business ready forecasts.

DARTS (Differentiable Architecture Search) is a library in Python that specializes in time series forecasting. It has a range of forecasting models that can be leveraged in a single place, making it super easy and efficient to use.

Time Series Forecasting Accuracy

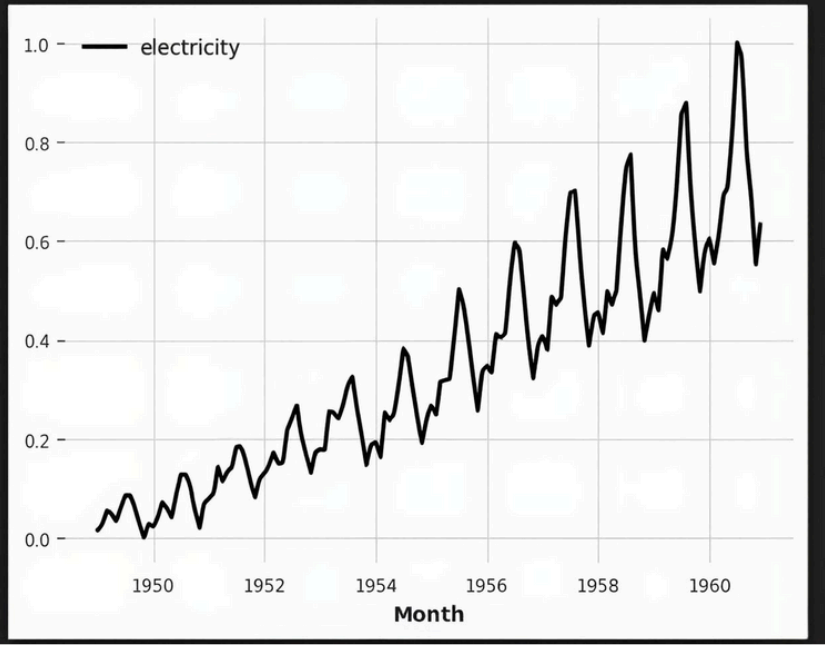
In general, a machine learning model architecture can broadly be set up in the following stages:



Let's start with train/validation set split for our electricity consumption dataset:

```
scaler_electricity = Scaler()
series_electricity_scaled = scaler_electricity.fit_transform(series_electricity)
series_electricity_scaled.plot(label="electricity")


#train validation split
#last 36 months as validation set
train_electricity, val_electricity = series_electricity_scaled[:-36], series_electricity_scaled[-36
```

✓ 0.5s



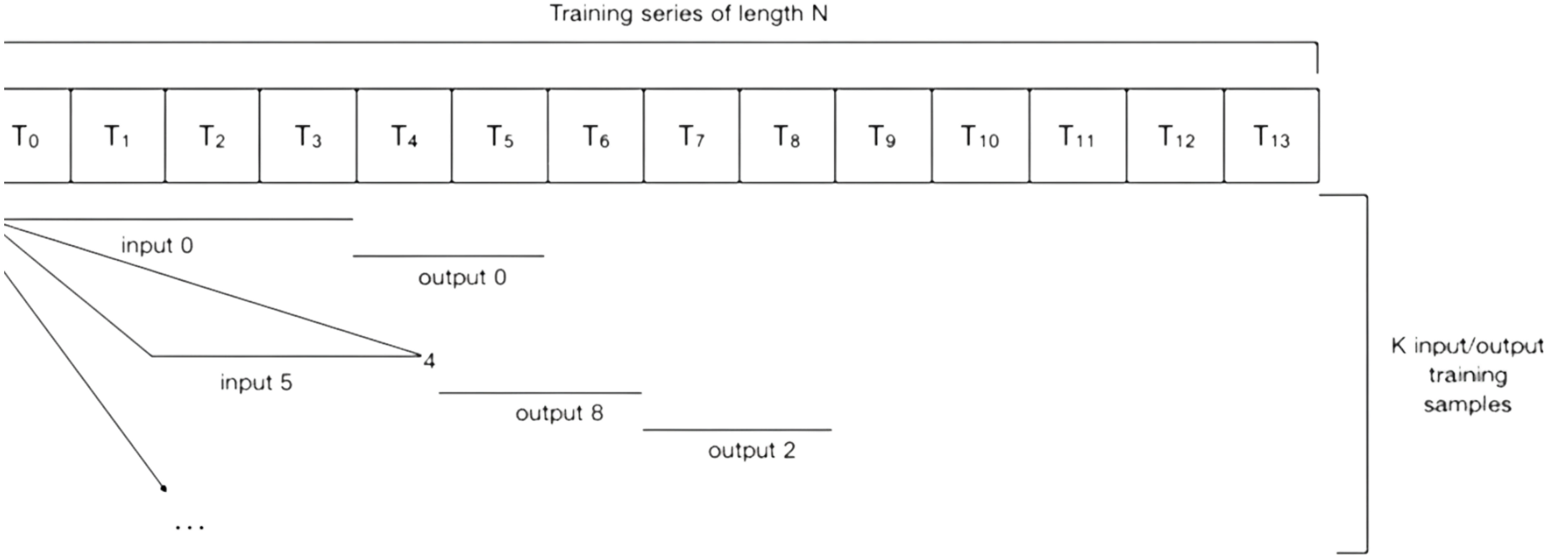
For simplicity, we can narrow down our models to the following:

- LinearRegressionModel
- BlockRNNModel
- Temporal Convolutional Networks (TCNModel)
- N-Beats (NBEATSModel)
- TiDEModel

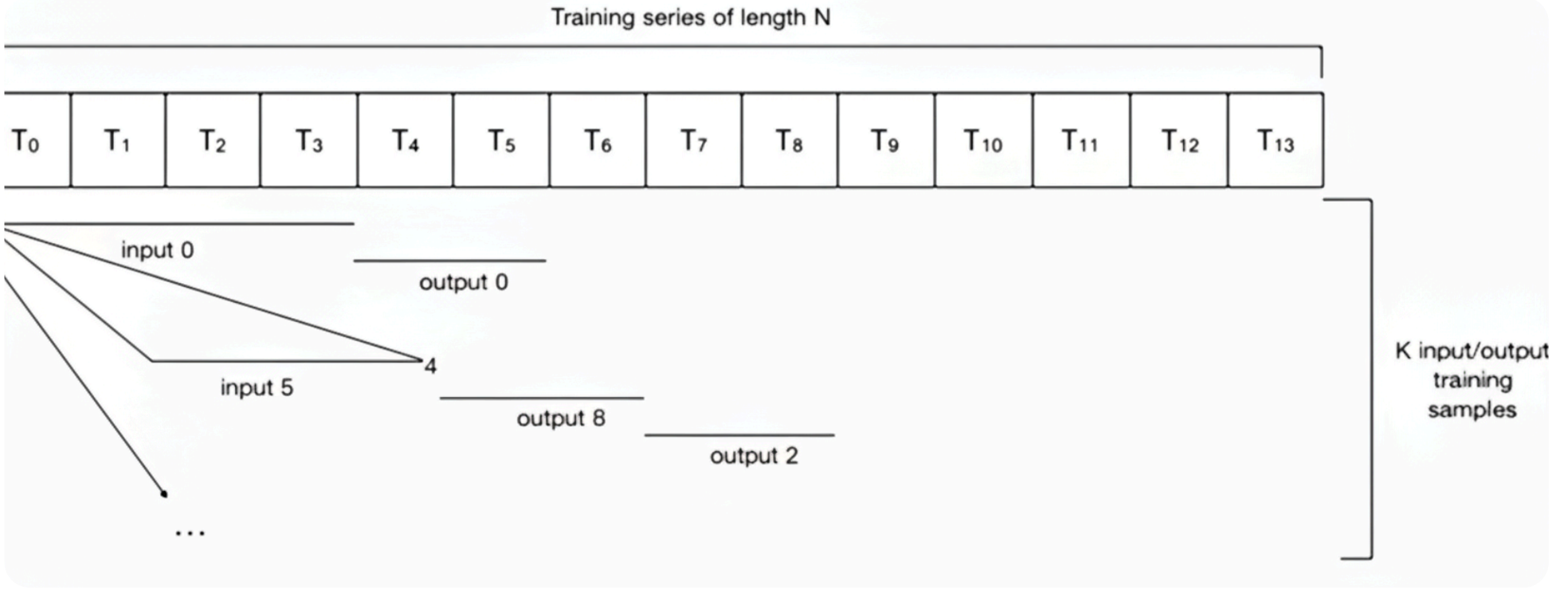
All the above are global forecasting models, which means that they can seamlessly be used with a time series of more than one dimension; the target series can contain one or several dimensions. A time series with several dimensions is just a regular time series where the values at each time stamp are vectors instead of scalars.

In addition to a single time series, we may be able to leverage some additional exogenous features (also called covariates) that could help with accuracy of prediction. We further differentiate covariate series, depending on whether they can be known in advance or not:

- **Past Covariates** denote time series whose past values are known at the time of the prediction. *These are usually things that must be measured or observed.*
- **Future Covariates** denote time series whose future values are already known at the time of prediction for the span of the forecast horizon. *These can, for instance, represent known future holidays or weather forecasts.*



Models such as BlockRNN, N-Beats, TCN, and Transformers are built on a block-based architecture. This means they process segments of the time series as input and generate corresponding segments of future values as output. The input size equals the number of target series dimensions combined with all covariate dimensions, while the output size corresponds only to the dimensions of the target series.



Let's look at some important parameters when building our models:

- **input_chunk_length:** this is the length of the lookback window of the model; each output will be computed by the model by reading the previous input_chunk_length points.
- **output_chunk_length:** this is the length of the outputs (forecasts) produced by the internal model. However, the predict() method of the "outer" DARTS model (e.g., the one of NBEATSModel, TCNModel, etc) can be called for a longer time horizon.

In these cases, if predict() is called for a horizon longer than output_chunk_length, the internal model will simply be called repeatedly, feeding on its own previous outputs in an auto-regressive fashion[CM1] (if the forecast horizon exceeds the model's output_chunk_length, the model extends predictions by recursively feeding its own prior outputs back as inputs, also called auto-regression).

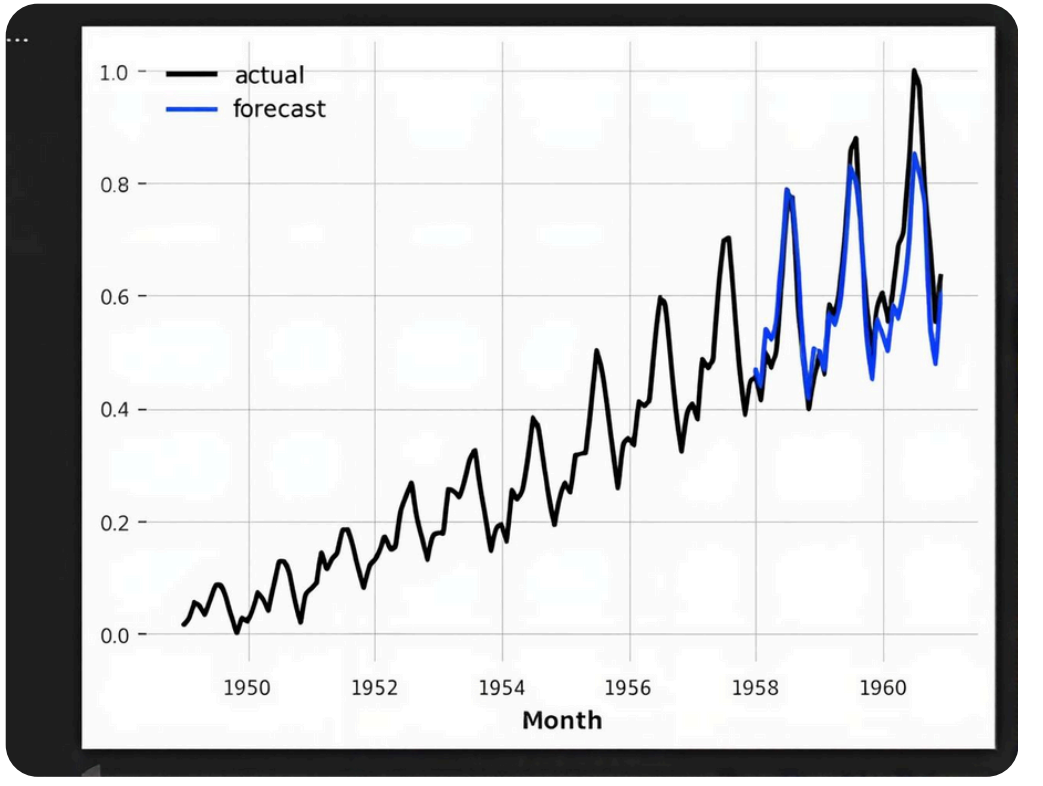
If past_covariates are used, it requires these covariates to be known for a set amount of time in advance.

Single Series Forecast

We'll build an N-BEATS model that has a lookback window of 24 points (input_chunk_length=24) and predicts the next 12 points (output_chunk_length=12). We chose these values so it'll make our model produce successive predictions for one year at a time, looking at the past two years.

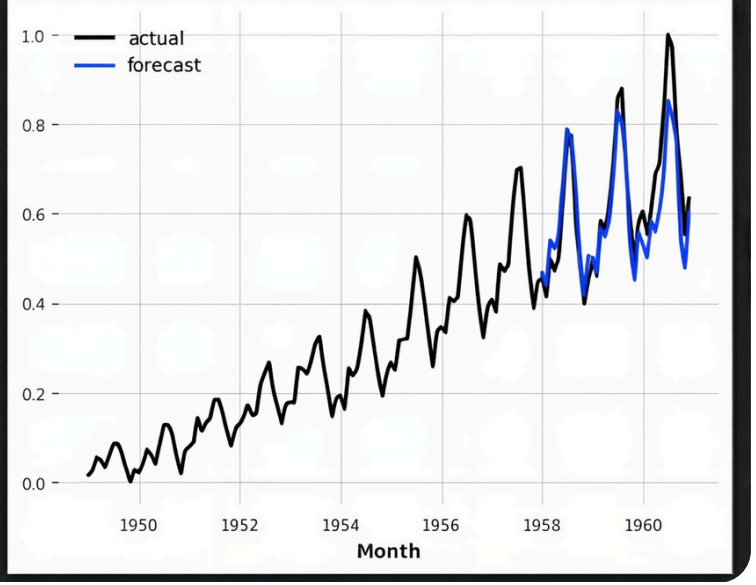
```
model_electricity = NBEATSModel(
    input_chunk_length=24,
    output_chunk_length=12,
    n_epochs=200,
    random_state=0,
    **generate_torch_kwargs(),
)

model_electricity.fit(train_electricity)


pred = model_electricity.predict(n=36)

series_electricity_scaled.plot(label="actual")
pred.plot(label="forecast")
plt.legend()
print(f"MAPE = {mape(series_electricity_scaled, pred):.2f}%")
```

```
[21] ✓ 1m 34.4s

Epoch 199: 100%

MAPE = 8.02%
```



```
model_electricity = NBEATSModel(
    input_chunk_length=24,
    output_chunk_length=12,
    n_epochs=200,
    random_state=0,
    **generate_torch_kwargs(),
)

model_electricity.fit(train_electricity)


pred = model_electricity.predict(n=36)

series_electricity_scaled.plot(label="actual")
pred.plot(label="forecast")
plt.legend()
print(f"MAPE = {mape(series_electricity_scaled, pred):.2f}%")
```

```
21]  ✓ 1m 34.4s

Epoch 199: 100%

MAPE = 8.02%
```



Training with Covariates

Behind the Scenes

To train the internal neural network, DARTS first makes a dataset of inputs/outputs examples from the provided time series (in this case: series_electricity_scaled). There are several ways this can be done, and DARTS contains a few different dataset implementations in the darts.utils.data package.

By default, NBEATSModel will instantiate a darts.utils.data.SequentialTorchTrainingDataset, which simply builds all the consecutive pairs of input/output sub-sequences of lengths input_chunk_length and output_chunk_length) existing in the series.

Training with Covariates

Build your own basic covariates (also called exogenous variables) from the date field.

```
# build year and month series:
electricity_year = datetime_attribute_timeseries(series_electricity_scaled, attribute="year")
electricity_month = datetime_attribute_timeseries(series_electricity_scaled, attribute="month")

# stack year and month to obtain series of 2 dimensions (year and month):
electricity_covariates = electricity_year.stack(electricity_month)

# split in train/validation sets:
electricity_train_covariates, electricity_val_covariates = electricity_covariates[:-36], electricity_covariates[-36:]

# scale them between 0 and 1:
scaler_covariates = Scaler()
electricity_train_covariates = scaler_covariates.fit_transform([
    electricity_train_covariates,
])
electricity_val_covariates = scaler_covariates.transform([
electricity_val_covariates,
])
```

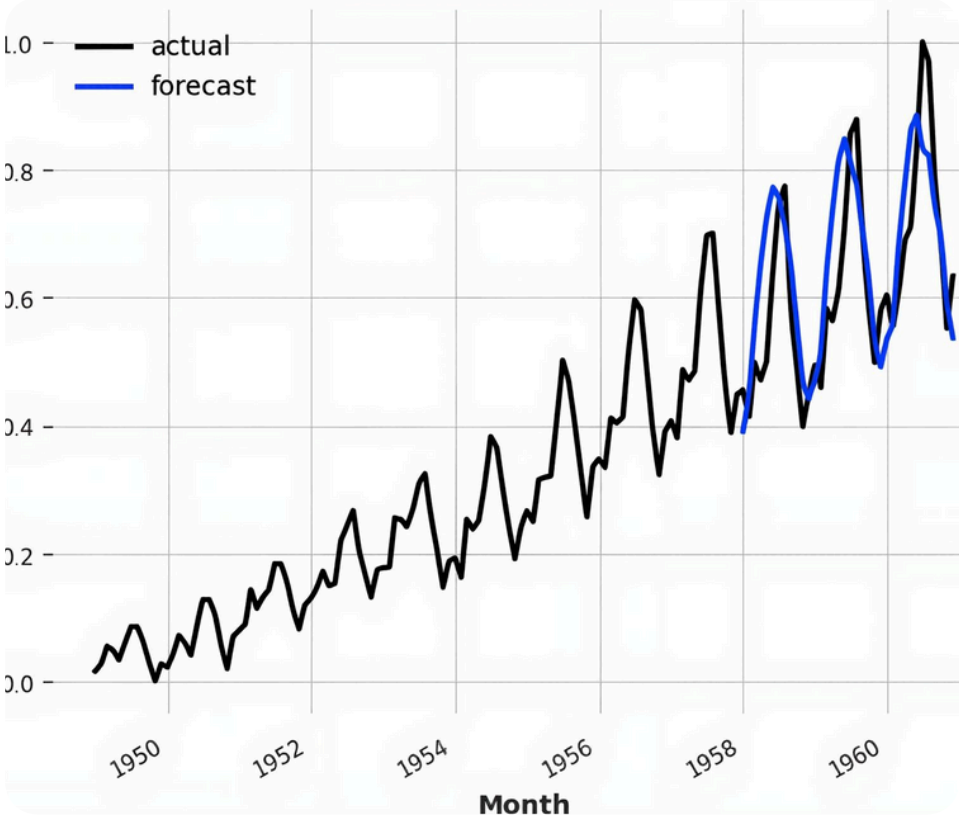
Training with Covariates

```
model_name = "BlockRNN_test"
model_pastcov = BlockRNNModel(
    model="LSTM",
    input_chunk_length=24,
    output_chunk_length=12,
    n_epochs=100,
    random_state=0,
    model_name=model_name,
    save_checkpoints=True,  # store model states: latest and best performing of validation set
    force_reset=True,
    **generate_torch_kwargs(),
)

#past covariates
model_pastcov.fit(
    series=[train_electricity],
    past_covariates=[electricity_covariates],
    val_series=[val_electricity],
    val_past_covariates=[electricity_covariates],
)
```

[24] ✓ 9.3s

```
Epoch 99: 100%
```

Looking at the chart, we see that the forecast tracks the actual data closely, capturing both the seasonal spikes and upward trend. The key takeaway is that the model is not just memorizing patterns. By feeding in relevant covariates, it learns relationships that help stabilize and maintain accuracy further into the horizon.



Back-testing with covariates

We can also back-test the models using covariates. Say for instance we are interested in evaluating the running accuracy with a horizon of 12 months starting at the beginning of the validation series.

- We start at the beginning of the validation series (start=val_electricity.start_time()).
- Each prediction will have length forecast_horizon=12.
- The next prediction will start stride=12 points after the previous one
- We keep all predicted values from each forecast (last_points_only=False)
- Then we continue until we run out of input data.

```python
### Backtesting
backtest_pastcov = model_pastcov.historical_forecasts(
    series_electricity_scaled,
    past_coveriates=electricity_covariates,
    start=val_electricity.start_time(),
    forecast_horizon=12,
    stride=12,
    last_points_only=False,
    retrain=False,
    verbose=True,
)
backtest_pastcov = concatenate(backtest_pastcov)
print(
    f"MAPE (BlockRNNModel with past coveriates) = {mape(series_electricity_scaled, backtest_pastcov):.2f}%"
)
```

✓ 0.1s

MAPE (BlockRNNModel with past covariates) = 12.09%

Generating TimeSeries: 100% 3/3 [00:00<00:00, 187.03it/s]

This means that on average, the model forecasts differ from actual values by about 12%, which indicates a reasonably strong fit given the complexity of real-world time series data.

So far, we have seen how:

- Simple model configuration controls how far back the model looks and how far forward it predicts.
- Covariates add critical context, boosting the accuracy and stability of forecasts.
- Back-testing provides a grounded way for us to confidently evaluate performance across the full horizon, giving us a strong degree of confidence in the model's reliability.

Nuances and Considerations

Building an ML model vs deploying them in production often comes with a few reality checks:

- Data Quality: Predictions (forecasts) are only as good as the input data. Missing values, anomalies, feature engineering, and concept drift require constant monitoring.
- Model Complexity vs Business Value: Sometimes a simple interpretable model may deliver more value with far less effort compared to a deep learning approach – we don't need a power drill to hang a small picture frame; a simple hammer and nail will do. Choosing the appropriate approach for the use case makes the most difference.
- Scalability: Must be able to integrate into existing data platforms and scale across multiple business units, products, and regions.
- Evaluation and Accuracy: ML pipeline robustness, explainability, and the ability for business to act on predictions matters as much as improving ML evaluation metrics.

Time series forecasting is never about chasing a perfect model – rather, focus on finding the balance between accuracy, interpretability and practical utility.

- **Start simple:** Begin with basic models like regression to understand the data better and establish a baseline.
- **Add complexity only if needed:** Move to more complex models, when the use case requires capturing long-term, non-linear dependencies.
- **Leverage covariates where possible:** They can dramatically improve performance, especially for real-world applications with external drivers.
- **Keep your use case in mind:** Sometimes slightly lower accuracy, but higher interpretability may be more valuable than a black-box model.

We're people people.
Throw some time on our calendar.

DI Squared

Let's get to work today, together.

To discover how we can help you do more with data, book a complimentary 1:1 with one of our data specialists today.

Start with a free 1:1

About DI Squared

DI Squared empowers organizations to extract the full value of their data through enhanced data platforms and processes.

We engineer flexible, reliable data infrastructure – from data lakes and pipelines to dashboards and augmented analytics – ensuring long-term scalability.

To learn more, visit www.disqr.com

About DI Squared
DI Squared empowers organizations to extract the full value of their data by engineering data platforms and processes. We engineer flexible, reliable data analytics infrastructure—from data lakes and pipelines to dashboards and augmented analytics—ensuring long-term scalability. To learn more, visit www.disqr.com