ML - Machine Learning

Time Series

Riccardo Berta

2026.01.28

Contents

1 Time Series

A **time series** is a sequence of observations collected at successive points in time, typically at regular intervals. Using functional notation, a time series can be written as:

$y(t), \quad t = 1, 2, \dots, T$

where y(t) denotes the value of the observed variable at time t. Common examples include the number of daily active users on a website, hourly temperature measurements in a city, a household's daily electricity consumption, or the trajectories of nearby vehicles. What distinguishes time series data is their **temporal ordering**: observations are not independent, and the order in which they occur is essential. In general, the value at time t depends on its past:

$y(t) = f\,(y(t-1), y(t-2), \dots) + \varepsilon(t)$

where

- $f(\cdot)$ captures **temporal patterns** such as trends, cycles, and recurring behaviors
- $\varepsilon(t)$ represents noise

By analyzing how the signal y(t) evolves over time, we can **learn patterns from the past** and use them to **forecast future values**, under the assumption that these patterns **persist**.

As a concrete example, consider the dataset reporting the daily number of passengers using buses and rail services operated by the Chicago Transit Authority. This dataset is publicly available from the Chicago Data Portal and provides a real-world illustration of time series data used for forecasting.

```python
import urllib.request
import zipfile
import os

# Define the URL of the dataset
url = "https://www.dropbox.com/s/wiww3a55sgifhb3/ridership.zip?dl=1"

# Create data directory if it does not exist
os.makedirs("./data", exist_ok=True)

# Prepare data if not already existing
if not os.path.exists("./data/ridership.zip"):
   print("Downloading dataset ... ")

   # Download the dataset
   u = urllib.request.urlopen(url)
   data = u.read()
   u.close()

   # Save the dataset to a local file
   with open("./data/ridership.zip", "wb") as f :
      f.write(data)

   # Extract the dataset
   with zipfile.ZipFile("./data/ridership.zip","r") as zip_ref:
      zip_ref.extractall("./data")

else:
   print("Dataset already exists. Skipping download.")
```

```
Dataset already exists. Skipping download.
```

We begin by **loading and cleaning the dataset**. Specifically, we read the CSV file, assign concise column names, sort the observations chronologically, remove the redundant total column, and eliminate any duplicate rows:

```python
import pandas as pd

# Load the dataset into a Pandas DataFrame
df = pd.read_csv("./data/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv",
                 parse_dates=["service_date"])

# Assign concise column names
df.columns = ["date", "day_type", "bus", "rail", "total"]

# Sort the DataFrame by date and set date as index
df = df.sort_values("date").set_index("date")

# Remove the redundant total column
df = df.drop("total", axis=1)

# Drop duplicate rows
df = df.drop_duplicates()
```

We now inspect the first few rows of the dataset in order to understand its structure and contents:

```python
# Inspect the first few rows of the dataset
df.head()
```

```
            day_type     bus    rail
date
2001-01-01         U  297192  126455
2001-01-02         W  780827  501952
2001-01-03         W  824923  536432
2001-01-04         W  870021  550011
2001-01-05         W  890426  557917
```

This dataset represents a **time series**, where observations are recorded at regular time intervals. More specifically, because multiple variables are observed at each time step, it is a **multivariate**

time series. The "day_type" column encodes the type of day: W for weekdays, A for Saturdays, and U for Sundays or public holidays.

1.1 Forecasting

The most common task in time series analysis is **forecasting**, which consists in predicting future values of a signal based solely on its past observations. Formally, given a time series y(t) observed up to time t, the goal of forecasting is to estimate one or more future values:

$\hat{y}(t+1), \hat{y}(t+2), \dots, \hat{y}(t+H)$

where H is the forecasting **horizon**.

In other words, forecasting aims to learn a function:

$\hat{y}(t+h) = f\,(y(t), y(t-1), \dots)$

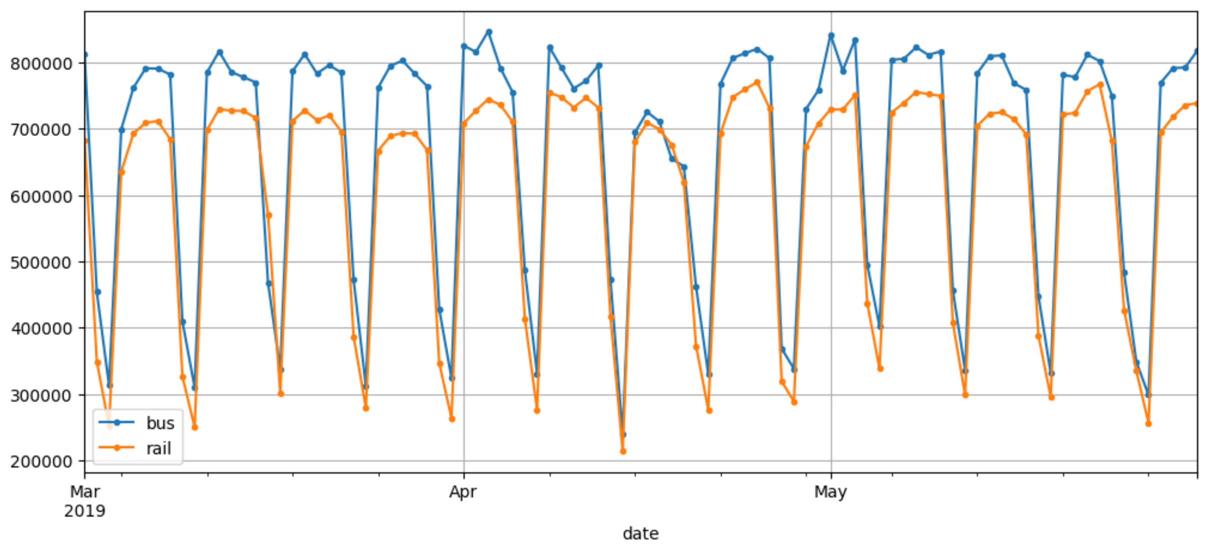
that **maps the historical evolution of the series to its future behavior**, under the assumption that the underlying temporal patterns remain stable.

In the concrete example, given the observed ridership up to today, we may want to forecast tomorrow's demand or the ridership over the next week. Such forecasts are essential for operational planning, resource allocation, and service optimization.

1.1.1 Seasonality and Naive Forecasting

To build some intuition, let's plot the bus and rail ridership over a few months in 2019 and observe their behavior over time.

```python
import matplotlib.pyplot as plt

# Plot bus and rail ridership from March to May 2019
df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(12, 5))
plt.show()
```



We can clearly observe a **recurring weekly pattern**, known as **seasonality**. In this case, the seasonal effect is so strong that even a very simple strategy, such as predicting tomorrow's ridership by copying the value observed one week earlier, already yields reasonably good results. This approach is known as **naive forecasting** and predicts future values by reusing past observations from the previous seasonal cycle. For a weekly seasonal pattern, the naive forecast can be written as

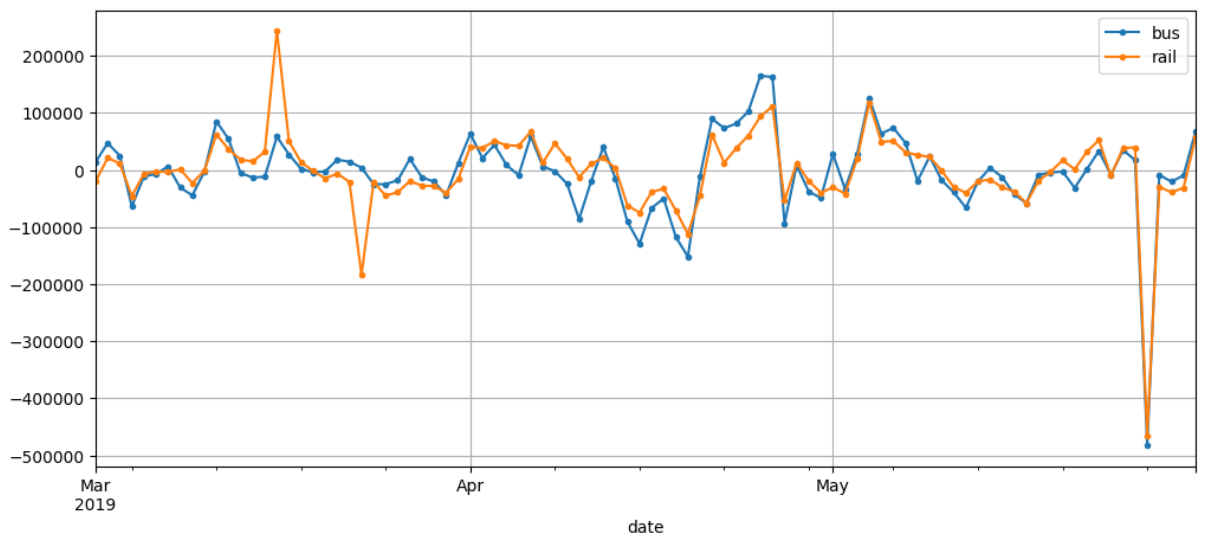
$$\hat{y}(t) = y(t-7)$$

Despite its simplicity, naive forecasting often provides a strong and informative **baseline** against which more sophisticated models can be compared. To make the weekly seasonality explicit, we can compute the **7-day difference**, defined as

$$\Delta_7 y(t) = y(t) - y(t-7)$$

If the series exhibits a strong weekly pattern, the **differenced signal** will fluctuate around zero, since values one week apart tend to be similar. In practice, this operation can be conveniently performed using the Pandas **diff()** method, which computes the difference between each value and the value observed a specified number of periods earlier:

```python
# Compute the difference betweem each value and the value observed 7 days earlier
diff_7 = df[["bus", "rail"]].diff(7)["2019-03":"2019-05"]

# Plot the differenced series to highlight seasonality
diff_7.plot(grid=True, marker=".", figsize=(12, 5))
plt.show()
```



Notice how closely the lagged time series follows the original one. When a time series shows a **strong correlation with a time-shifted version of itself**, we say that it exhibits **autocorrelation**. In this case, most of the differences remain quite small, indicating a strong weekly autocorrelation. An exception appears toward the end of May, where the differences become larger. This deviation may be due to a disruption of the usual pattern, such as a public holiday. To verify this, let's inspect the day types for that period:

```
# Check the day_type column for the end of May 2019
list(df.loc["2019-05-25":"2019-05-27"]["day_type"])
```

```
['A', 'U', 'U']
```

1.1.2 Performance Evaluation

The period with deviations corresponds to a long weekend, with Monday being the Memorial Day holiday. Such **calendar effects** can disrupt the usual seasonal pattern and may be explicitly modeled to improve forecasting accuracy. For now, however, we deliberately keep the model simple. To obtain a first **quantitative assessment** of our forecasting approach, we compute the **Mean Absolute Error** (**MAE**) over a three-month period. Given a sequence of true values and the corresponding forecasts, the MAE is defined as:

$$\text{MAE} = \frac{1}{T}\sum_{t=1}^{T}|y(t) - \hat{y}(t)|$$

This metric measures the average magnitude of the forecast errors in the original units of the data, providing an intuitive indication of how far the predictions are, on average, from the true observations:

```python
# Calculate the Mean Absolute Error (MAE) over the three-month period
mae_naive = diff_7.abs().mean();

# Display the MAE results
print("Mean Absolute Error (MAE) from March to May 2019:")
print(f"Bus: {mae_naive['bus']:.2f}")
print(f"Rail: {mae_naive['rail']:.2f}")
```

```
Mean Absolute Error (MAE) from March to May 2019:
Bus: 43915.61
Rail: 42143.27
```

At first glance, it is difficult to judge whether these errors are **large or small**. To put them into perspective, we normalize the forecast errors by dividing them by the corresponding target values. This metric is known as the **Mean Absolute Percentage Error** (**MAPE**) and is defined as:

$$\text{MAPE} = \frac{1}{T}\sum_{t=1}^{T}\left|\frac{y(t)-\hat{y}(t)}{y(t)}\right| \times 100$$

```python
# Get the target values for normalization
targets = df[["bus", "rail"]]["2019-03":"2019-05"]

# Calculate the Normalized Mean Absolute Error (NMAE)
mape_naive = (diff_7 / targets).abs().mean()

# Display the NMAE results
print("Mean Absolute Percentage Error (MAPE) from March to May 2019:")
print(f"Bus: {mape_naive['bus']:.2%}")
print(f"Rail: {mape_naive['rail']:.2%}")
```

```
Mean Absolute Percentage Error (MAPE) from March to May 2019:
Bus: 8.29%
Rail: 8.99%
```

An interesting observation is that, while the **MAE** appears slightly better for rail than for bus, the opposite holds when we consider **MAPE**. This happens because bus ridership volumes are generally larger than rail ridership volumes, which naturally leads to larger **absolute errors**:

$|y(t) - \hat{y}(t)|$

However, when forecast errors are expressed **relative to the true values**, the bus forecasts turn out to be slightly more accurate than the rail forecasts. For completeness, the **Mean Squared Error (MSE)** is defined as:

$$\text{MSE} = \frac{1}{T}\sum_{t=1}^{T}\left(y(t) - \hat{y}(t)\right)^2$$

MSE measures the average **squared** difference between the observed values and the corresponding forecasts over time. MSE **penalizes large forecast errors** more heavily than small ones, making it particularly sensitive to outliers and abrupt deviations. This property is useful when large errors are especially undesirable.

The **MAE**, **MAPE**, and **MSE** are among the most commonly used metrics for evaluating forecasting performance. As always, the choice of metric should be guided by the **scale of the data** and the **specific objectives** of the forecasting task.

1.1.3 Long-Term Patterns

By visually inspecting the time series, it is difficult to identify any clear **long-term trend** or **seasonal pattern** beyond the strong weekly seasonality. To further investigate this aspect, we can analyze the series at different temporal scales by **aggregating** observations over different time intervals.

Let $\mathcal{T}_k$ denote a generic time window (for example, a month, a quarter, or a year). An aggregated version of the time series can then be defined as:

$$y^{(\text{agg})}(k) = \frac{1}{|\mathcal{T}_k|}\sum_{t\in\mathcal{T}_k} y(t)$$

where $|\mathcal{T}_k|$ is the number of observations contained in the time window. By choosing different definitions of time window, we can construct monthly, quarterly, or yearly aggregated series, each emphasizing patterns at a different temporal scale.

In practice, this type of temporal aggregation is conveniently performed using the Pandas **resample()** method, which transforms a time series from one temporal resolution to another by **grouping observations into time-based windows** defined by a chosen resampling rule. The method requires a **DateTimeIndex** and produces a grouped representation of the data, which can then be summarized using aggregation operators such as the mean, sum, or median.

Aggregating a time series over long intervals compresses many observations into a single value. While this helps reveal broad trends, **it also hides information about how the series behaves within each interval**. In other words, aggregation tells us what happens on average, but not how the process evolves over time. **Rolling statistics** address this limitation by averaging over a moving window: instead of collapsing the data, they slide smoothly along the time axis. This preserves the temporal ordering of the observations while filtering out short-term noise, making it easier to visually track the gradual evolution of long-term trends and seasonal effects as they unfold

Given a window length m, the rolling average of the series is defined as:

$$\bar{y}^{(m)}(t) = \frac{1}{m}\sum_{i=0}^{m-1} y(t-i)$$

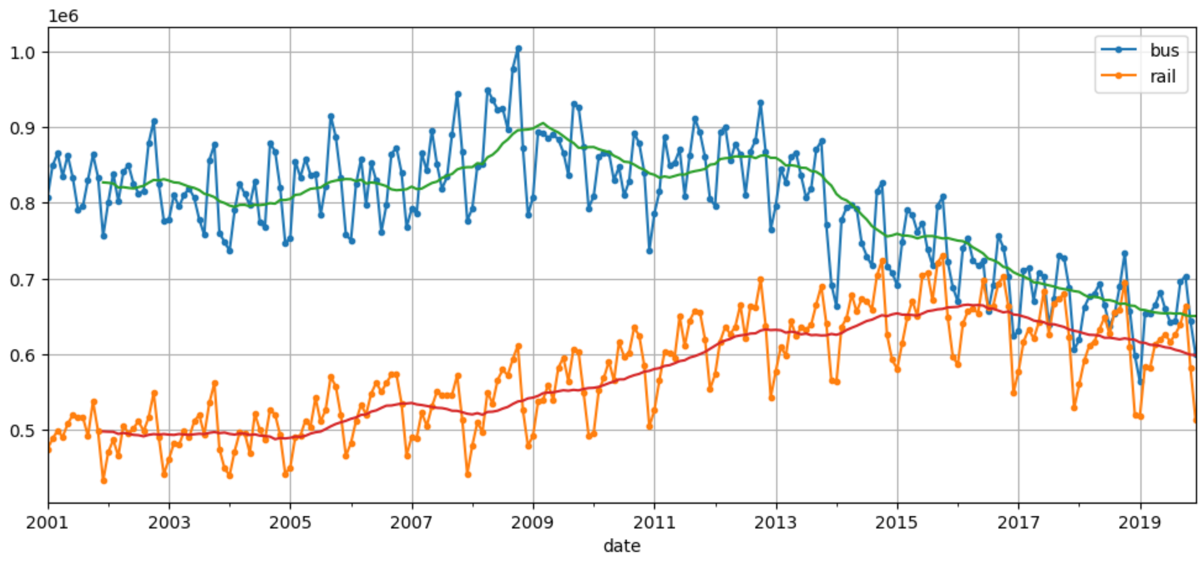
This operation replaces each observation with the average of the previous m values, producing a smoothed version of the series. Smaller values of m highlight medium-term fluctuations, while larger values emphasize long-term trends and low-frequency seasonal components.

In Pandas, rolling statistics are computed using the **rolling**() method, which applies a sliding window over the time series and computes the desired statistic for each position of the window.

By combining **aggregation** and **rolling averages**, we can study the time series at multiple temporal scales and more effectively assess the presence of **long-term trends** and **seasonal patterns** that may not be visible at the original resolution.

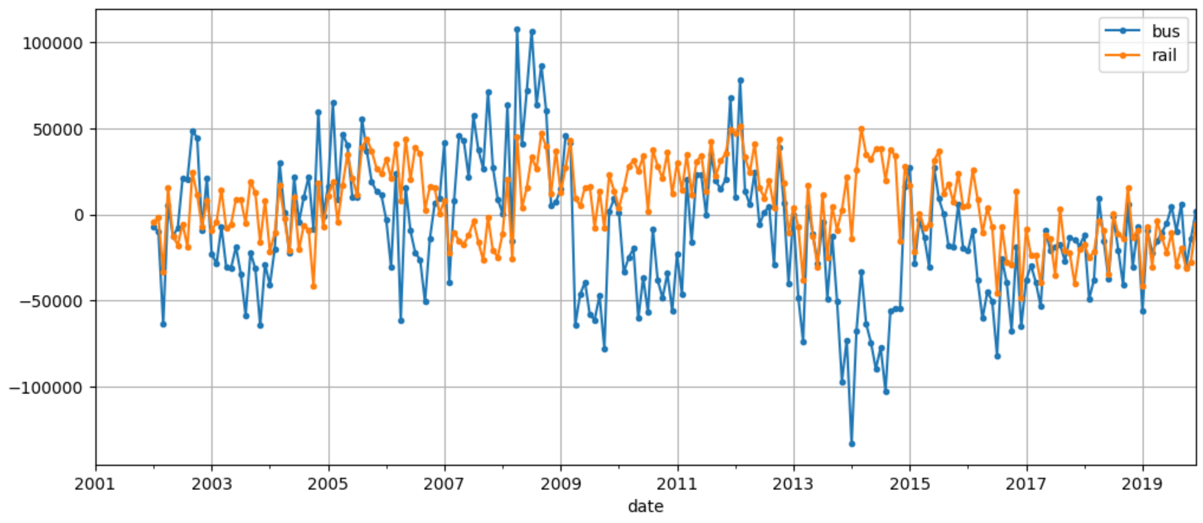
In the example, we can analyze data from 2001 to 2019 and visualize both the monthly aggregated series and its 12-month rolling average in order to assess the presence of **yearly seasonality** and **long-term trends** in the data:

```python
# Select a longer period for yearly seasonality analysis
period = slice("2001", "2019")

# Remove the day_type column for resampling
df_monthly = df.loc[ : , df.columns!='day_type']

# Compute the mean for each month
df_monthly = df_monthly.resample(rule='ME').mean()

# Compute the 12-month rolling average
rolling_average_12_months = df_monthly[period].rolling(window=12).mean()

# Plot the monthly data along with rolling average
fig, ax = plt.subplots(figsize=(12, 5))
df_monthly[period].plot(ax=ax, marker=".")
rolling_average_12_months.plot(ax=ax, grid=True, legend=False)
plt.show()
```



The data exhibit a degree of **yearly seasonality**, although it is noisier than the weekly pattern and more pronounced in the rail series than in the bus series. In particular, we observe peaks and troughs occurring at approximately the same times each year. To make this yearly pattern more explicit, let's examine the **12-month difference** of the series:

```python
# Compute the 12-month difference
diff_12 = df_monthly.diff(12)[period]

# Plot the differenced series to highlight seasonality
diff_12.plot(grid=True, marker=".", figsize=(12, 5))
plt.show()
```



After applying the 12-month differencing, the yearly seasonal pattern and long-term trend are largely removed, and the series now fluctuates around zero, making deviations and anomalies much more apparent.

1.2 Moving Average Models

Moving Average models are a class of time series models where the present value is explained not by past values of the series itself, but by how the system has recently deviated from its own predictions, capturing the impact of short-lived random disturbances.

1.2.1 Autoregressive Moving Average (ARMA)

An alternative to naive forecasting is the **Autoregressive Moving Average (ARMA)** model, which generates forecasts by combining two complementary mechanisms. It predicts future values as a **weighted sum of past observations** and refines these predictions by incorporating a **weighted sum of recent forecast errors**:

$$\hat{y}(t) = \sum_{i=1}^{p} \alpha_i \, y(t-i) + \sum_{i=1}^{q} \beta_i \, \epsilon(t-i)$$

where the forecast error is defined as

$$\epsilon(t) = y(t) - \hat{y}(t)$$

The first term is the **autoregressive (AR)** component, which captures how past values of the series influence the current value. The second term is the **moving average (MA)** component, which corrects the forecast using information contained in past prediction errors. The hyperparameters p and q determine how many past observations and past errors are taken into account, respectively. In practice, there is no single "correct" choice for p and q. A common and effective strategy is to start with **small values**, fit the model to estimate the parameters α_i and β_i (typically using **maximum likelihood estimation**), and then examine the model's **diagnostics**, such as residual autocorrelation and error behavior. If the residuals resemble **white noise**, the model has successfully captured the relevant temporal structure of the data. If systematic patterns remain, the model is too simple and should be refined by increasing p and/or q. Conversely, if increasing model complexity does not lead to a meaningful improvement in the diagnostics, adding more parameters is unlikely to be beneficial and may instead result in **overfitting**, where the model begins to fit noise rather than signal.

This iterative procedure balances **model expressiveness**, the ability to capture temporal dependencies, and **parsimony**, the principle of keeping the model as simple as possible, ensuring that the chosen model explains the data well without introducing unnecessary parameters.

1.2.2 Autoregressive Integrated Moving Average (ARIMA)

The previous model works well when the time series is **stationary**, meaning that its statistical properties, such as mean and variance, do not change over time:

$\mathbb{E}[y(t)] = \mu, \qquad \text{Var}(y(t)) = \sigma^2 \quad \forall, t$

When a series contains **trends** or evolving patterns, this assumption is violated. In such cases, **differencing** provides a simple yet powerful way to remove non-stationary components. A **one-step difference** is defined as:

$\Delta y(t) = y(t) - y(t-1)$

This operation is analogous to a **first-order derivative** in discrete time: it measures how much the series changes from one time step to the next and removes any **linear trend**. For example, the series:

$y(t) = [3, 5, 7, 9, 11]$

grows linearly. Applying first differencing yields:

$\Delta y(t) = [2, 2, 2, 2]$

indicating a constant slope and showing that the original linear trend has been removed. If the series follows a **quadratic trend**, a single round of differencing is not sufficient. Consider the series:

$y(t) = [1, 4, 9, 16, 25, 36]$

After one differencing step we obtain:

$\Delta y(t) = [3, 5, 7, 9, 11]$

which is still trending. Applying differencing a **second time**:

$\Delta^2 y(t) = \Delta(\Delta y(t)) = y(t) - 2y(t-1) + y(t-2)$

yields:

$[2, 2, 2, 2]$

which removes the remaining structure. A convenient notation to express these operations is the **lag operator** L, which represents a shift backward in time:

$Ly(t) = y(t-1)$

Applying the operator multiple times produces larger shifts:

$L^2 y(t) = y(t-2), \quad L^3 y(t) = y(t-3)$, and so on.

In general,

$L^k y(t) = y(t-k)$

Using this notation, a one-step difference can be written compactly as:

$\Delta y(t) = y(t) - y(t-1) = y(t) - Ly(t) = (1-L)y(t)$

while a second difference becomes:

$\Delta^2 y(t) = (1-L)[(1-L)y(t)] = (1-L)^2 y(t) = y(t) - 2y(t-1) + y(t-2)$

More generally, applying differencing d times gives:

$\Delta^d y(t) = (1-L)^d y(t)$

This operation approximates the **d-th derivative** of the series and removes **polynomial trends up to degree d**. The hyperparameter d is called the **order of integration**. This idea lies at the core of the **Autoregressive Integrated Moving Average** (**ARIMA**) model. Before modeling temporal dependencies using an ARMA model, the series is first "flattened" through differencing so that its statistical properties remain stable over time:

$z(t) = \Delta^d y(t) = (1-L)^d y(t)$

An ARMA model is then applied to the differenced series:

$\hat{z}(t) = \sum_{i=1}^{p} \alpha_i \, z(t-i) + \sum_{i=1}^{q} \beta_i \, \epsilon(t-i)$

with prediction error

$\epsilon(t) = z(t) - \hat{z}(t)$

In summary, ARIMA is characterized by three hyperparameters: p and q, as in ARMA, and the additional parameter d, which controls the degree of differencing applied to the series before fitting the ARMA model.

1.2.3 Seasonal ARIMA (SARIMA)

A natural final extension of ARIMA is the **Seasonal ARIMA** (**SARIMA**) model, which explicitly accounts for **seasonal patterns**. The key idea is that many time series evolve on **two time scales simultaneously**:

- a **short-term scale**, capturing local dynamics from one time step to the next;
- a **seasonal scale**, capturing patterns that repeat every s time steps.

While ARIMA models only the short-term structure, SARIMA augments it by introducing additional autoregressive and moving-average components that operate at the **seasonal lag**. To do this, the model introduces three **seasonal hyperparameters** P, D, and Q, together with the **seasonal period** s. After applying both regular and seasonal differencing, the transformed series is defined as:

$$z(t) = \Delta^d \Delta_s^D y(t) = (1 - L)^d (1 - L_s)^D y(t)$$

An ARMA model is then applied to this stationary series:

$$\hat{z}(t) = \underbrace{\sum_{i=1}^{p} \alpha_i z(t-i)}_{\text{non-seasonal AR}} + \underbrace{\sum_{i=1}^{P} A_i z(t-i \cdot s)}_{\text{seasonal AR}} + \underbrace{\sum_{i=1}^{q} \beta_i \epsilon(t-i)}_{\text{non-seasonal MA}} + \underbrace{\sum_{i=1}^{Q} B_i \epsilon(t-i \cdot s)}_{\text{seasonal MA}}$$

with prediction error:

$$\epsilon(t) = z(t) - \hat{z}(t)$$

In this formulation, the non-seasonal terms model short-term dependencies, while the seasonal terms capture repeating patterns occurring every s time steps. Together, these components allow SARIMA to represent both **local dynamics** and **long-range seasonal structure** within a unified framework.

The **statsmodels** library is a Python package for classical statistical modeling and time series analysis. It provides implementations of ARMA, ARIMA, and SARIMA models, along with a wide range of statistical diagnostics and hypothesis tests. Unlike many machine learning libraries, statsmodels emphasizes **explicit model assumptions**, **parameter estimation**, and **interpretability**. We can exploit this library to fit a SARIMA model to the rail time series and generate forecasts that explicitly account for its seasonal behavior:

```python
from statsmodels.tsa.arima.model import ARIMA

# Select the rail time series for a specific period
origin, today = "2019-01-01", "2019-05-31"
rail_series = df.loc[origin:today]["rail"].asfreq("D")

# Define the ARIMA model parameters
order = (1, 0, 0)             # p=1, d=0, q=0
seasonal_order = (0, 1, 1, 7) # P=0, D=1, Q=1, and s=7

# Fit the SARIMA model to the rail time series
model = ARIMA(rail_series, order=order, seasonal_order=seasonal_order)
model = model.fit()
```

We can use the model to generate a forecast for the next day

```python
# Use the model to generate a forecast for the next day
y_pred = model.forecast()
print("SARIMA prediction: ", y_pred.values[0])

# Compare with the actual value
print("Actual value: ", df["rail"].loc["2019-06-01"])

# Compare with the naive prediction
print("Naive prediction: ", df["rail"].loc["2019-05-25"])
```

```
SARIMA prediction:  427758.6263023059
Actual value:  379044
Naive prediction:  426932
```

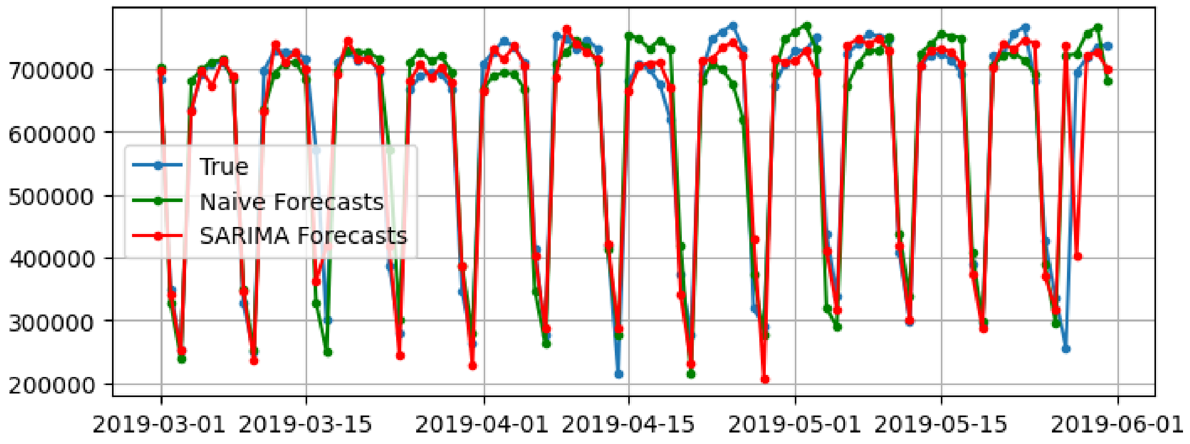
The forecast for this particular day is quite poor, similar or even worse than the naive baseline. However, judging a model based on a single prediction can be misleading. To properly assess its performance, we need to generate forecasts over a longer horizon and evaluate them using an aggregate metric, such as the Mean Absolute Percentage Error over the entire period:

```python
# Earliest date used for training
origin = "2019-01-01"

# Start and end of the evaluation period
start_date = "2019-03-01"
end_date = "2019-05-31"

# Create a daily date range for evaluation
time_period = pd.date_range(start_date, end_date)

# Extract the rail time series and enforce daily frequency
rail_series = df.loc[origin:end_date]["rail"].asfreq("D")

# List to store one-step-ahead predictions
y_preds = []

# Rolling (walk-forward) forecasting
for today in time_period.shift(-1):

    # Train the model using all data available up to "today"
    train_series = rail_series.loc[origin:today]
    # Fit the SARIMA model
    model = ARIMA(
        train_series,
        order=order,
        seasonal_order=seasonal_order
    )
    results = model.fit()

    # Forecast the next day (one-step ahead)
    y_pred = results.forecast(steps=1).iloc[0]

    # Store the prediction
    y_preds.append(y_pred)

# Convert predictions to a time-indexed Series
y_preds_sarima = pd.Series(y_preds, index=time_period)

# Naive forecast
y_preds_naive = rail_series.shift(7).loc[time_period]

# Targets for normalization
targets = rail_series.loc[time_period]

# Compute Naive MAPE
mape_naive = ((y_preds_naive - targets)/targets).abs().mean()

# Compute SARIMA MAPE
mape_sarima = ((y_preds_sarima - targets)/targets).abs().mean()

# Display the MAPE results
print(f"Naive MAPE: {mape_naive:.2%}")
print(f"SARIMA MAPE: {mape_sarima:.2%}")
```

```
Naive MAPE: 8.99%
SARIMA MAPE: 7.54%
```

Although the SARIMA model is not perfect, it consistently outperforms the naive baseline by a wide margin on average. To better understand this improvement, we can visualize the SARIMA forecasts alongside the actual observed values:

```
fig, ax = plt.subplots(figsize=(8, 3))

ax.plot(rail_series.loc[time_period].index,
        rail_series.loc[time_period].values,
        marker=".", label="True",
)

ax.plot(y_preds_naive.index,
        y_preds_naive.values,
        color="green", marker=".", label="Naive Forecasts"
)

ax.plot(y_preds_sarima.index,
        y_preds_sarima.values,
        color="r", marker=".", label="SARIMA Forecasts"
)

ax.grid(True)
ax.legend()
plt.show()
```



The plot shows that both models capture the strong weekly seasonality, but the SARIMA forecasts track the true series more closely than the naive baseline, especially around weekday peaks and weekend drops, resulting in a consistently lower average error.

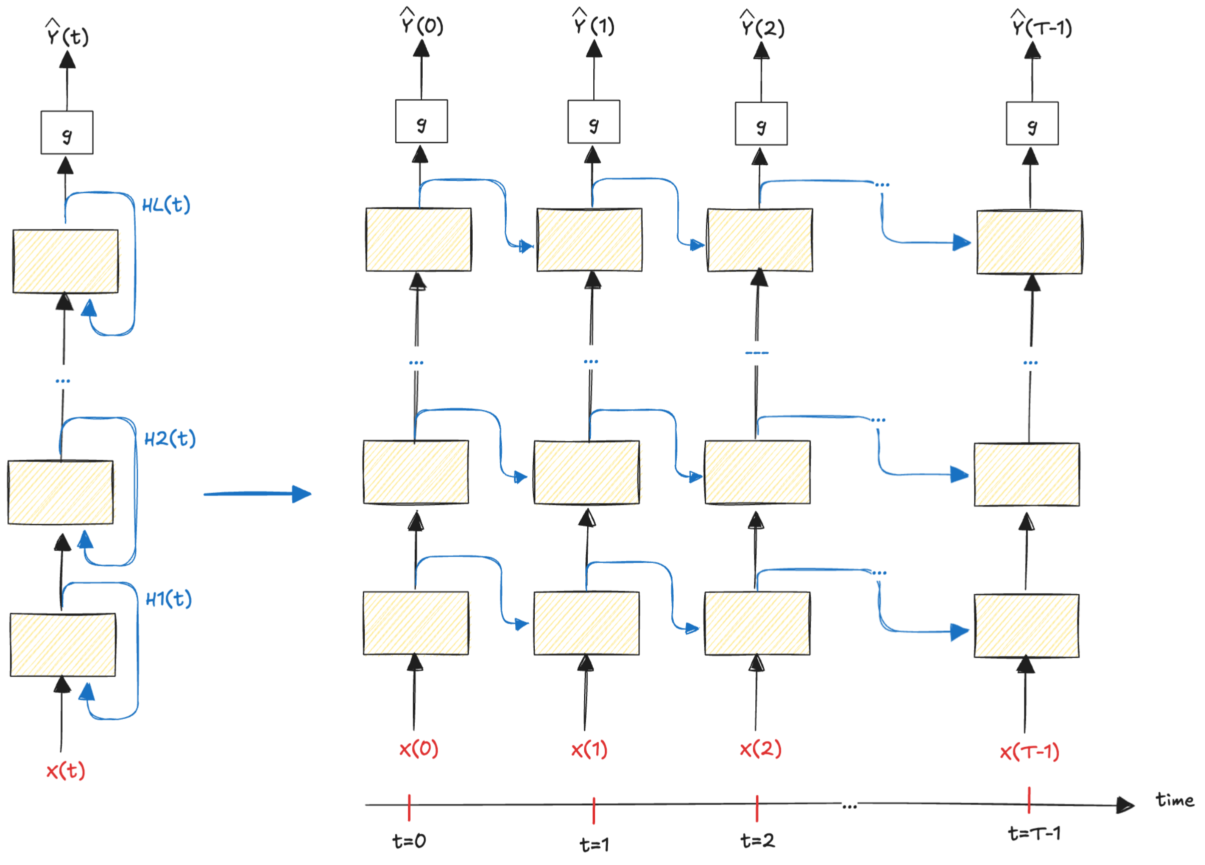
1.3 Recurrent Neural Network (RNN)

In a feedforward neural network, information flows strictly in one direction, from the input layer to the output layer. **Predictions are made independently**, without any memory of past inputs.

This makes feedforward networks poorly suited for time series, where the current value often depends on what happened before. A **Recurrent Neural Network (RNN)** addresses this limitation by introducing **feedback connections**: the network feeds part of its output back into itself. This creates an internal **state** that acts as a memory, allowing the model to carry information forward in time.

1.3.1 Recurrent Neuron

We begin with the simplest possible recurrent neural network: a single recurrent neuron. Unlike a standard feedforward neuron, which processes each input independently, a recurrent neuron maintains **an internal state that evolves over time**. This internal state allows the neuron to **retain information about past inputs** and to use it when processing new data:



At each time step t, the neuron receives two sources of information. The first is the current input vector:

$x(t) \in \mathbb{R}^d$

The second is the hidden state:

$h(t-1) \in \mathbb{R}$

which is carried over from the previous time step and acts as a **learned summary of the past inputs**. These two signals are combined to produce a new hidden state h(t), which is then fed back into the neuron and becomes part of the input at the next time step.

Because the hidden state is scalar in this setting, the neuron has two distinct sets of parameters: an input weight vector:

$w_x \in \mathbb{R}^d$

which processes the current input, and a recurrent weight

$w_h \in \mathbb{R}$

which controls how strongly past information influences the present. A bias term

$b \in \mathbb{R}$

is also included. The state update equation is therefore:

$h(t) = \phi\left(w_x^\top x(t) + w_h h(t-1) + b\right)$

where $\phi(\cdot)$ is a nonlinear activation function, such as ReLU. To fully define the recurrence, an initial hidden state must be specified, and is typically set to zero or treated as a learnable parameter.

The output of the neuron at time t is obtained by applying an output function g()) to the hidden state:

$\hat{y}(t) = g(h(t))$

This simple recurrent neuron already captures the key idea behind recurrent neural networks: **information flows not only forward from input to output, but also through time, allowing the model to build and update an internal memory as new data arrive**.

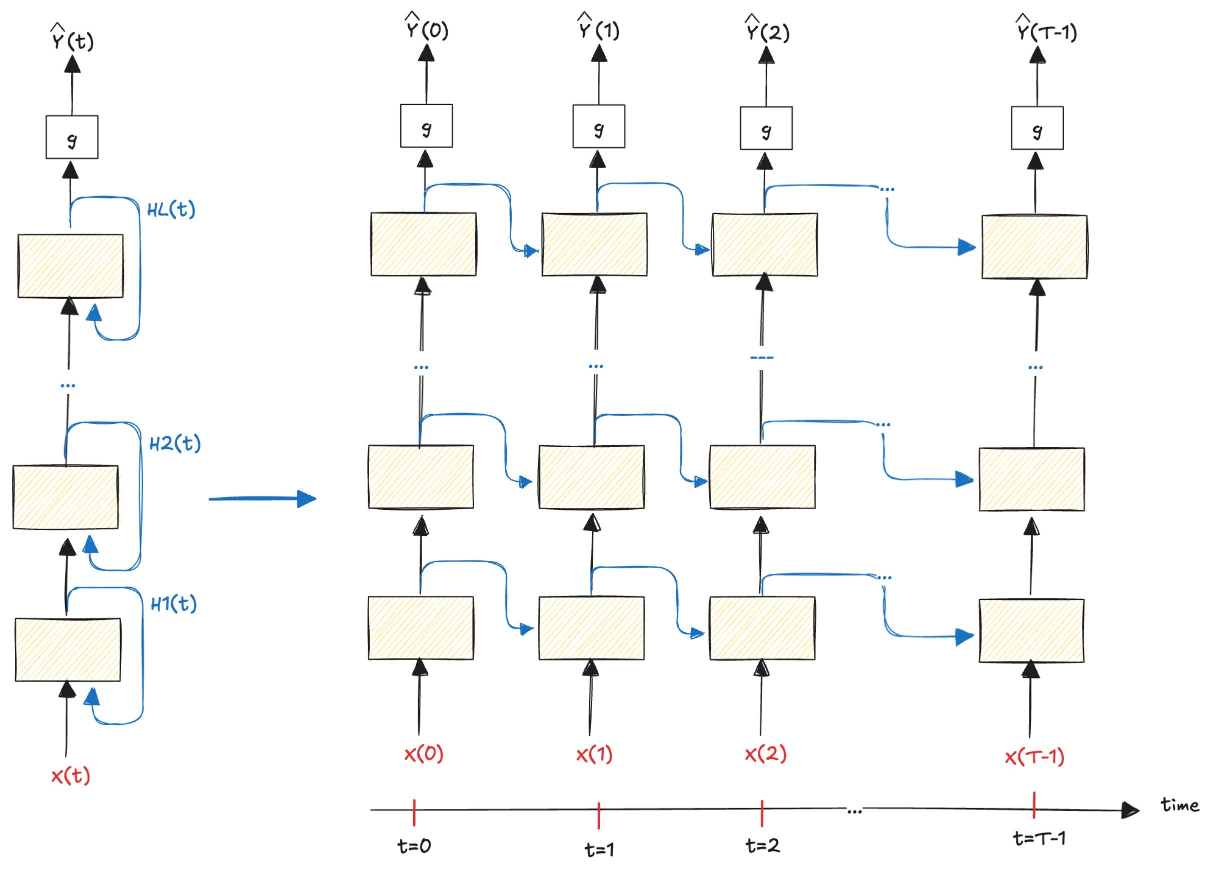
For example, in the context of daily ridership data, the hidden state can be interpreted as a compact summary of past travel patterns (weekly seasonality) which is then combined with the current day's observation to produce a prediction for the next day's ridership. We can implement this idea in Python:

```python
import numpy as np

# Phi activation function (ReLU)
def phi(z):
    return np.maximum(0.0, z)
```

```python
# g output function (identity)
def g(h):
    return h

def rnn_cell(x_t, h_prev, w_x, w_h, b):

    # Compute the linear combination of inputs and previous hidden state
    z = w_x @ x_t + w_h * h_prev + b

    # Apply activation function
    h_t = phi(z)

    # Compute output
    y_t = g(h_t)

    # Return the new hidden state and output
    return h_t, y_t
```

The recurrent neuron maintains an **internal state** that acts as a **compact memory** of the input sequence, enabling the network to retain, update, and exploit **information about the past**. As a result, observations are not processed in isolation: the model explicitly captures temporal dependencies, which makes it particularly well suited for sequential and time-series data. In this sense, the hidden state functions as a learned representation of past information that is continuously updated over time.

To better understand how this mechanism works, we can **conceptually unroll the recurrent connection along the time axis**. Consider an input sequence of length T, indexed by time steps t=0,1,...,T-1. At each time step, the recurrent neuron receives both the current input x(t) and its own hidden state h(t-1) from the previous time step. Since no past state exists at the initial time step t=0, the hidden state is typically initialized to zero.

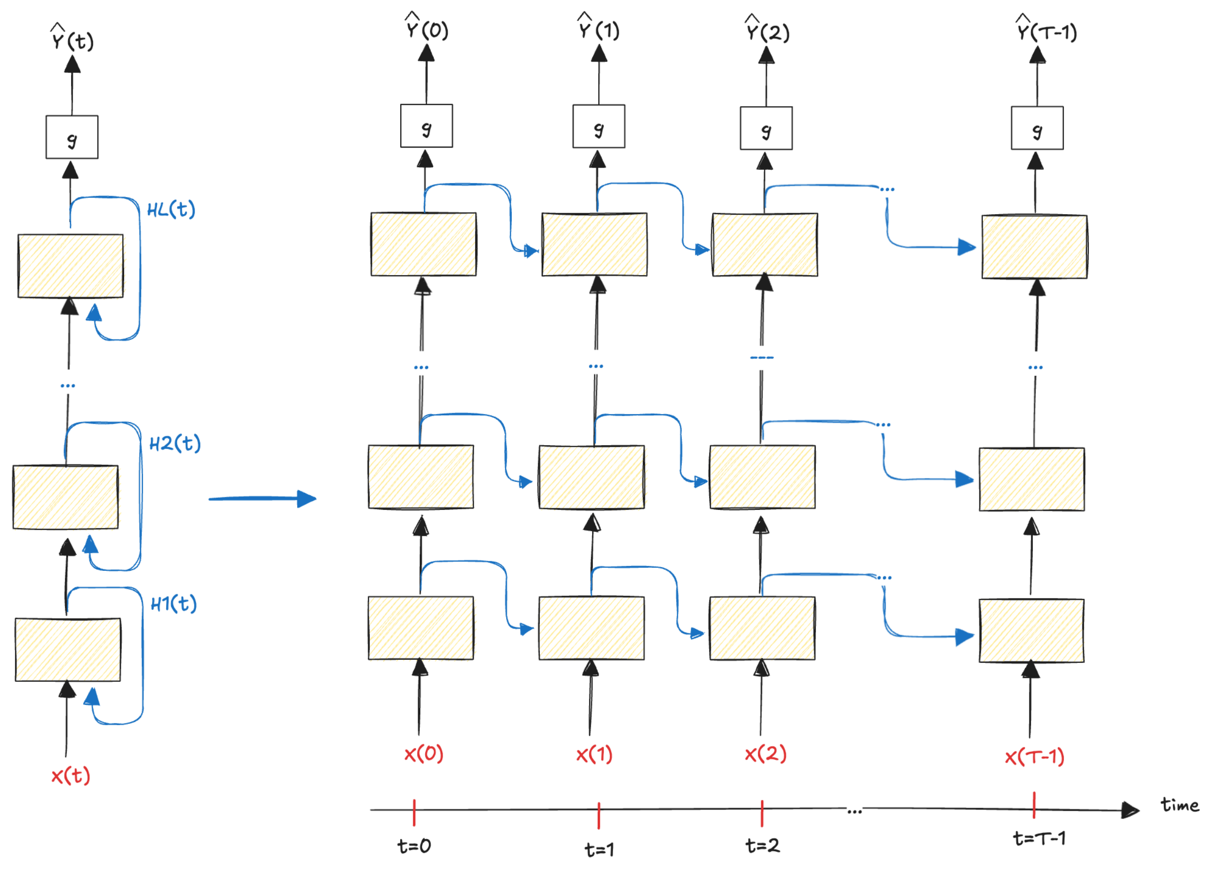


In this **unfolded view**, the recurrent neuron appears as **a small network replicated across time, with the same parameters shared at every step**. This representation, known as **unrolling** the network through time, makes explicit how information flows forward from one time step to the next and how the hidden state evolves over the entire sequence of length T We can unroll the previous Python implementation as follows:

```
def rnn_cell_unroll(X, w_x, w_h, b):

    # Get the number of time steps
    T = len(X)

    # Initialize the hidden state for the first time step
    h = 0.0

    # Lists to store hidden states and outputs
    H = []
    Y_hat = []

    # Unroll the RNN over time steps
    for t in range(T):

        # Compute the new hidden state and output
        h, y = rnn_cell(X[t], h, w_x, w_h, b)

        # Append to the lists
        H.append(h)
        Y_hat.append(y)

    # return the arrays of hidden states and outputs
    return np.array(H), np.array(Y_hat)
```

1.3.2 Recurrent Layer

A single recurrent neuron has very limited representational power. For this reason, practical recurrent neural networks use not just one recurrent neuron, but a **layer of recurrent neurons operating in parallel**. The layer consists of m recurrent neurons, all processing the **same input sequence**:



At each time step t, the input vector x(t) is provided simultaneously to all neurons in the layer. Each neuron maintains its own hidden activation, and together these activations form the hidden state vector:

$$h(t) = [h_0(t), h_1(t), \dots, h_{m-1}(t)]^\top \in \mathbb{R}^m$$

Each neuron combines the shared input with the **entire hidden state vector from the previous time step**, not just with its own past activation. As a result, every neuron has its own input weight vector and its own recurrent weight vector:

$$h_i(t) = \phi\left(w_{x,i}^\top x(t) + w_{h,i}^\top h(t-1) + b_i\right)$$

where:

$w_{x,i} \in \mathbb{R}^d$

$w_{h,i} \in \mathbb{R}^m$

$b_i \in \mathbb{R}$

This formulation shows that the hidden state represents a **distributed memory**: information about the past is not stored in a single unit, but is spread across multiple neurons, each capturing different aspects of the temporal structure (such as trends, periodic patterns, or short-term fluctuations).

The output at time t is obtained by applying a mapping function g to the entire hidden state vector:

$\hat{y}(t) = g(h(t))$

rather than to individual hidden units.

In practice, as for feedforward networks, recurrent neural networks are almost always trained using **mini-batches**, rather than processing a single sequence at a time. Thus, instead of working with a single input sequence:

$x(0), x(1), \dots, x(T-1)$

we work with a mini-batch of B sequences, each of length T:

$x^{(1)}(0), \dots, x^{(1)}(T-1)$
$x^{(2)}(0), \dots, x^{(2)}(T-1)$
...
$x^{(B)}(0), \dots, x^{(B)}(T-1)$

At a fixed time step t, the input to the recurrent layer can be represented as a matrix

$$X(t) = \begin{bmatrix} x^{(1)}(t) \\ x^{(2)}(t) \\ \vdots \\ x^{(B)}(t) \end{bmatrix} \in \mathbb{R}^{B \times d}$$

where each row corresponds to the input vector of one sequence at time t. Importantly, X(t) **does not represent a sequence by itself**; the **temporal structure arises from the ordered collection of such matrices over time**. Similarly, the hidden state at time t is represented as

$$H(t) = \begin{bmatrix} h^{(1)}(t) \\ h^{(2)}(t) \\ \vdots \\ h^{(B)}(t) \end{bmatrix} \in \mathbb{R}^{B \times m}$$

where each row is the hidden state of one sequence, and m is the hidden dimension of the recurrent layer.

At each time step, each sequence in the mini-batch **evolves independently**, but all sequences **share the same parameters**. For the b-th sequence in the mini-batch, the hidden state update is given by:

$$h^{(b)}(t) = \phi\big(W_x x^{(b)}(t) + W_h h^{(b)}(t-1) + b\big), \qquad b = 1, \dots, B$$

Explicitly, this corresponds to

$$
\begin{aligned}
h^{(1)}(t) &= \phi\big(W_x x^{(1)}(t) + W_h h^{(1)}(t-1) + b\big)\,,\\
h^{(2)}(t) &= \phi\big(W_x x^{(2)}(t) + W_h h^{(2)}(t-1) + b\big)\,,\\
&\vdots\\
h^{(B)}(t) &= \phi\big(W_x x^{(B)}(t) + W_h h^{(B)}(t-1) + b\big)\,.
\end{aligned}
$$

Stacking these equations for all sequences in the mini-batch leads to the vectorized formulation:

$$H(t) = \phi(X(t)W_x + H(t-1)W_h + b)$$

where

$$
\begin{aligned}
&W_x \in \mathbb{R}^{d\times m}\\
&W_h \in \mathbb{R}^{m\times m}\\
&b \in \mathbb{R}^{m}
\end{aligned}
$$

At each time step, the recurrent layer produces an output vector for each sequence in the mini-batch. If the output dimension is O, we can write:

$$\hat{Y}(t) = \begin{bmatrix} \hat{y}^{(1)}(t) \\ \hat{y}^{(2)}(t) \\ \vdots \\ \hat{y}^{(B)}(t) \end{bmatrix} \in \mathbb{R}^{B\times O}$$

The output is obtained by applying an output mapping to the hidden states:

$$\hat{Y}(t) = g(H(t))$$

In the simplest and most common case, g() is an affine transformation applied row-wise:

$$\hat{Y}(t) = H(t)W_y + b_y$$

where

$$W_y \in \mathbb{R}^{m\times O}$$

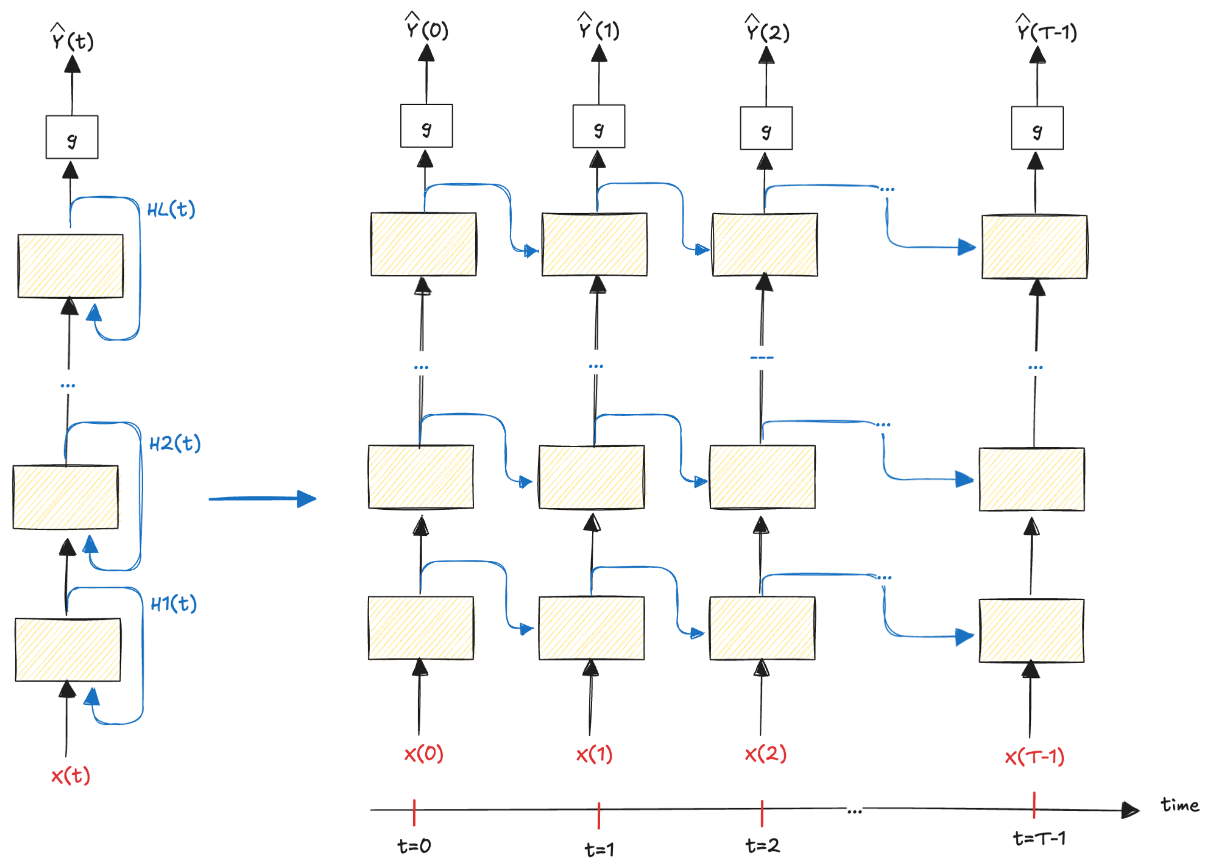
$$b_y \in \mathbb{R}^{O}$$

The same weight matrices are shared across all sequences in the mini-batch and across all time steps, ensuring that the model learns a single, consistent temporal transformation. Mini-batch processing significantly **improves computational efficiency** and **stabilizes training**, while fully preserving the temporal dynamics learned by the recurrent layer.

We can descrive this recurrent layer in Python as follows:

```python
import numpy as np

def init_rnn_layer(input_dim, hidden_dim, output_dim):

    params = {
        # Input weights:
        "W_x": np.random.normal(0.0, 0.1, size=(input_dim, hidden_dim)),

        # Recurrent weights:
        "W_h": np.random.normal(0.0, 0.1, size=(hidden_dim, hidden_dim)),

        # Bias for each recurrent neuron
        "b": np.zeros(hidden_dim),

        # Output weights:
        "W_y": np.random.normal(0.0, 0.1, size=(hidden_dim, output_dim)),

        # Output bias
        "b_y": np.zeros(output_dim)
    }

    return params
```

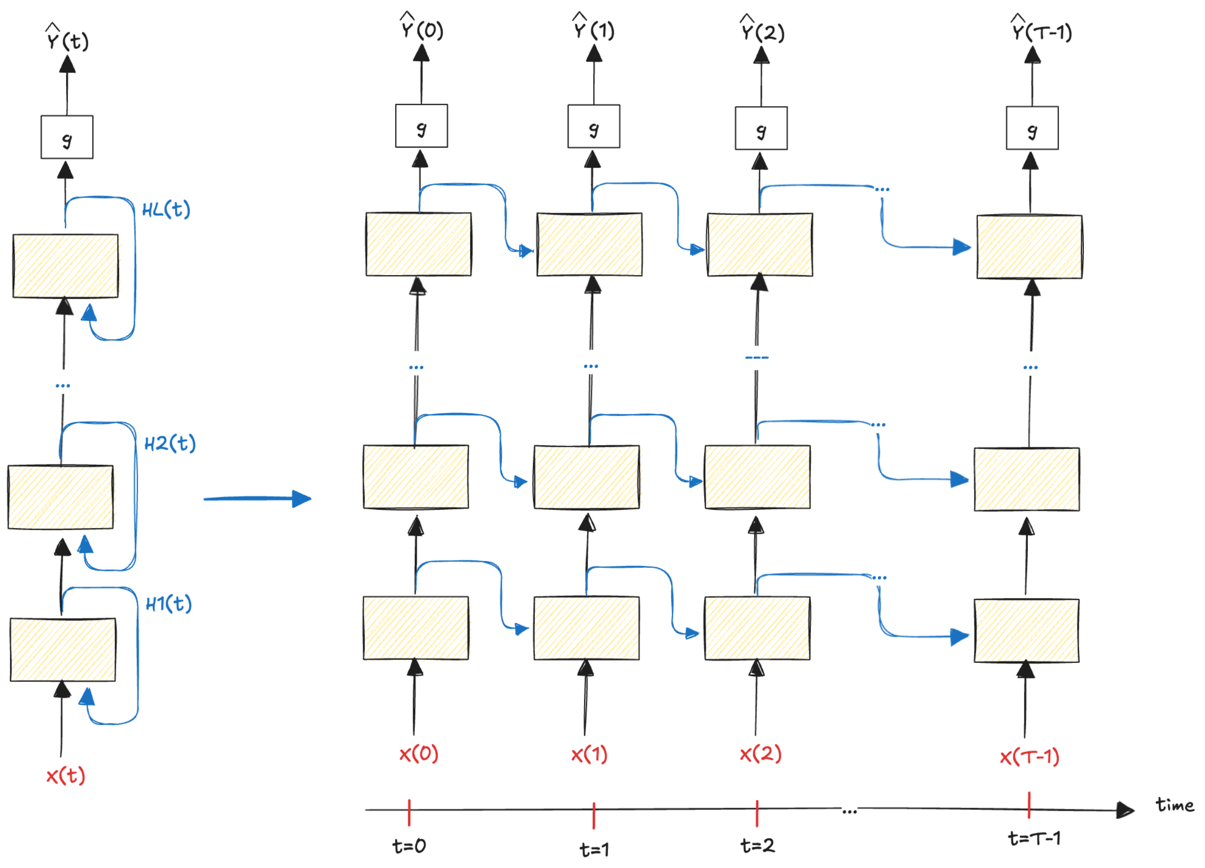
As with a single recurrent neuron, a recurrent layer can be **unrolled through time**. Each block represents the same recurrent layer evaluated at a different time step, with **shared parameters** across all copies. At time t, the layer receives the current input and the hidden state from the previous step, and produces a new hidden state and an output:



The same layer (with the same parameters) is replicated at each time step. This makes explicit how the hidden state evolves and how information flows forward in time, while emphasizing a crucial property of recurrent networks: **parameter sharing across time**. The weight matrices and the bias are reused at every time step, allowing the model to process sequences of arbitrary length with a fixed number of parameters.

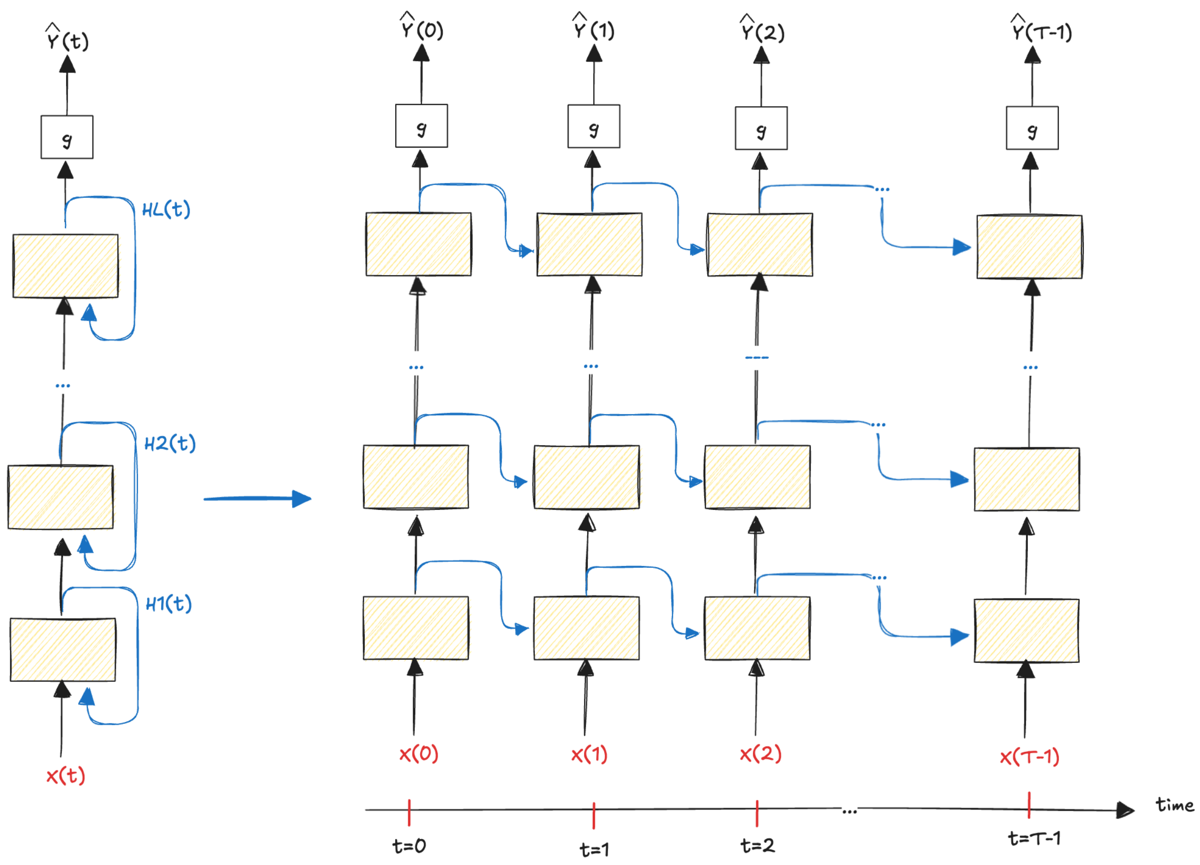
1.3.3 Recurrent Architectures

An RNN can process an entire sequence and produce a **corresponding sequence of outputs**. This architecture is known as a **sequence-to-sequence network**:



For example, the input consists of the last T observed values of household power consumption, and train it to predict the same sequence shifted **one day into the future**, effectively learning to forecast the next day's consumption at each time step.

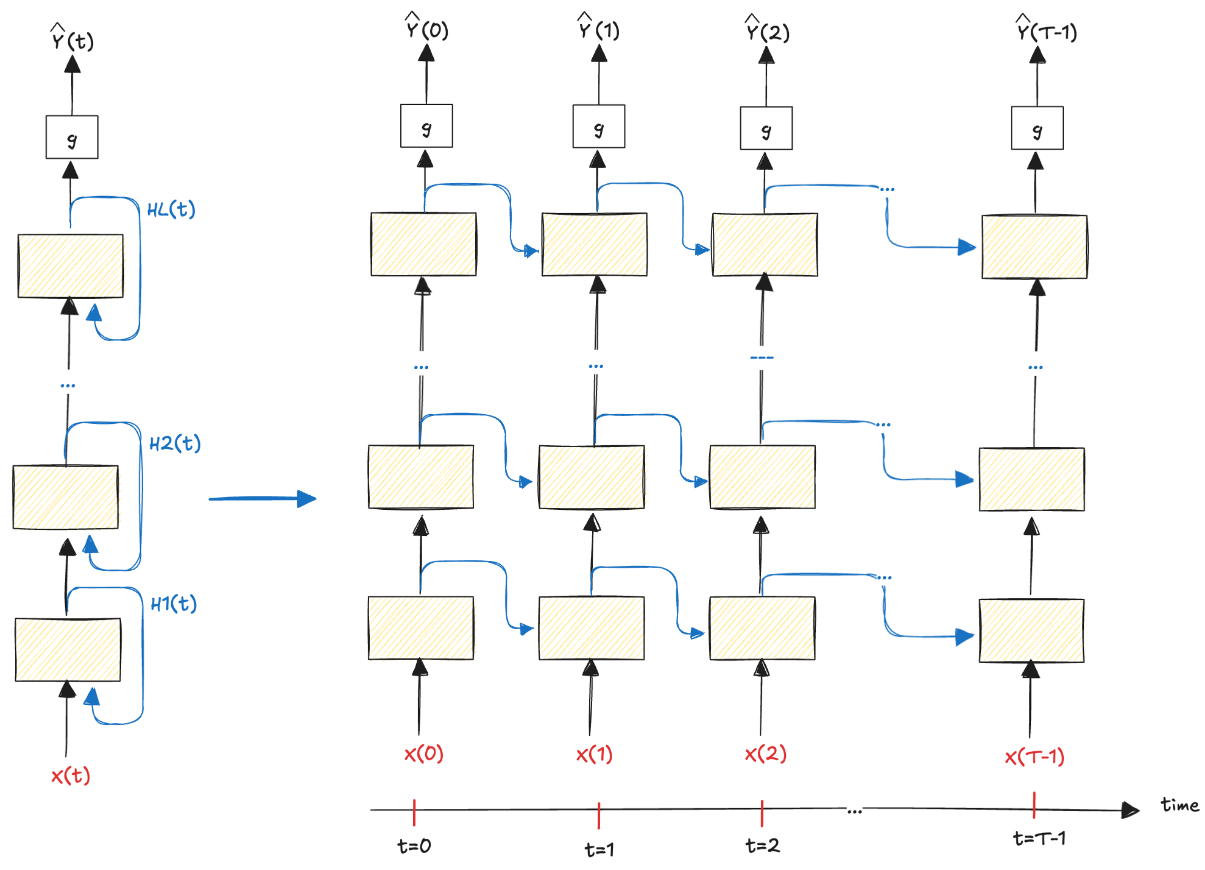
Alternatively, we can feed the network a **sequence of inputs** and use **only the final output**, obtaining a **sequence-to-vector** architecture:



In this case, the network compresses the entire input sequence into a single representation, which can then be used for tasks such classification. For example, we can input a sequence of words forming a movie review and have the network output a single sentiment score summarizing the overall opinion expressed in the text.

Although both architectures can be used to predict values one step into the future, **they differ in how supervision is applied**. In a sequence-to-sequence model, the network is trained to make accurate predictions at every time step, whereas in a sequence-to-vector model only the final prediction is supervised. As a result, sequence-to-sequence training provides a denser learning signal and encourages accurate intermediate predictions, while sequence-to-vector training forces the network to compress the entire sequence into a single representation optimized for the final output. The use case for sequence-to-sequence models is typically forecasting tasks, while sequence-to-vector models are often employed for classification or regression tasks based on sequential data.

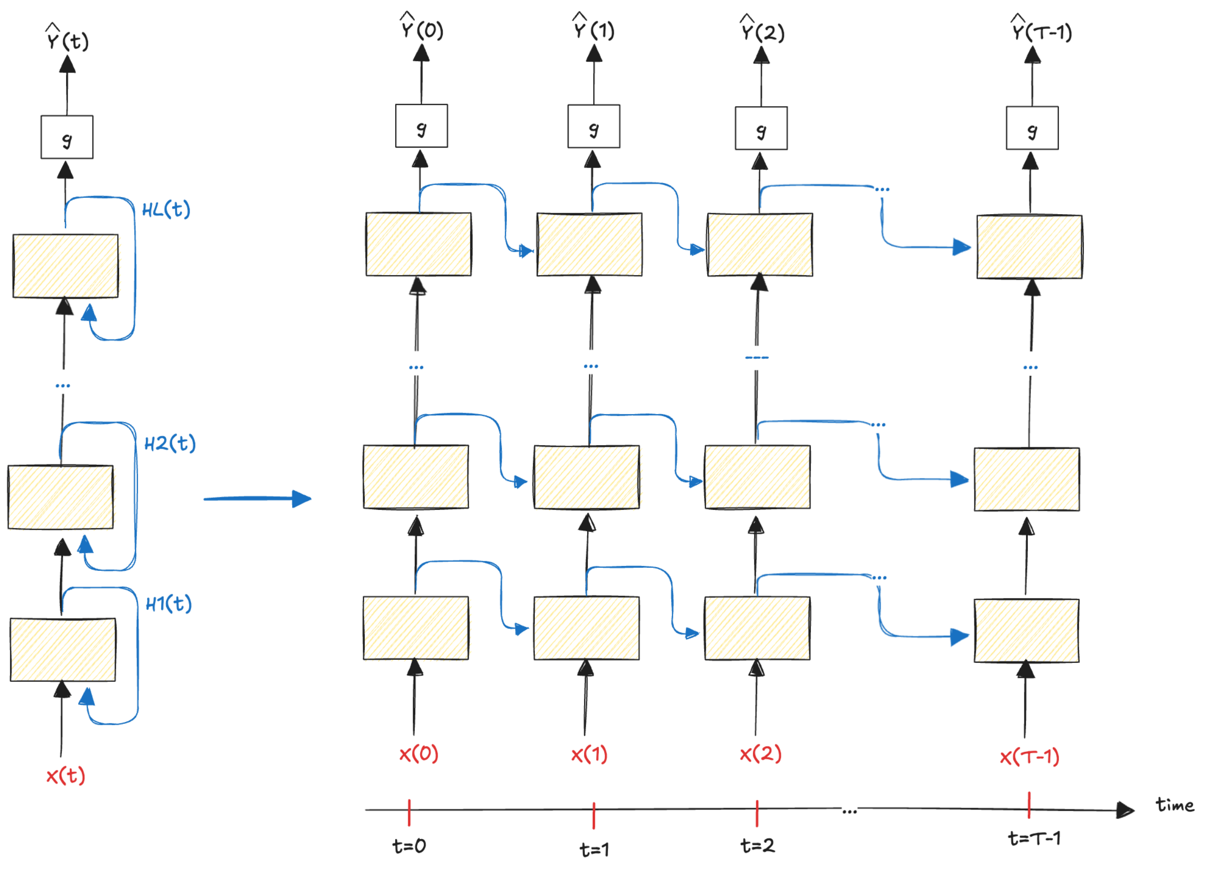
We can also provide the network with a **single input vector** and reuse it at every time step, allowing the network to generate a **sequence of outputs**. This architecture is known as a **vector-to-sequence** network:



In this setting, the same input vector is provided to the recurrent layer at each time step, while the hidden state evolves over time and produces a sequence of outputs. Alternatively, the input vector can be used to initialize the hidden state, with subsequent outputs generated purely through the recurrent dynamics; both formulations are commonly used in practice.

A common example is image captioning, where an image (or a feature vector extracted by a convolutional neural network) is given as input, and the recurrent network generates a sequence of words describing the image.

Lastly, we can consider an architecture composed of a sequence-to-vector network, called an **encoder**, followed by a vector-to-sequence network, called a **decoder**. This **encoder–decoder structure** allows the model to transform an input sequence into an output sequence of potentially different length.



The encoder processes the input sequence and compresses it into a single vector representation, given by its final hidden state. Intermediate encoder outputs are ignored. The final encoder hidden state is then used to initialize the hidden state of the decoder. The decoder generates an output sequence, one element at a time, using the encoded representation and its own recurrent dynamics. In practice, the decoder may also receive previous output tokens as inputs (**teacher forcing**). For example, this architecture can be used for translating a sentence from one language to another. We feed the network a sentence in one language, the encoder converts it into a single vector representation, and then the decoder decodes the vector into a sentence in another language.

1.3.4 Backpropagation Through Time

When working with sequential data, each prediction produced by a recurrent neural network depends not only on the current input, but also on the internal state built from all previous inputs. An intuitive way to think about this is to view the hidden state as a running summary of the past. If the network makes a poor prediction at time t, the problem may not lie only in the current input, but in how the hidden state was constructed at earlier time steps. Perhaps the network **failed to retain relevant information from the past**, or **assigned too much importance to irrelevant inputs**. In this sense, an error observed at time t **reveals that something went wrong earlier** in the sequence, when the hidden state was being built.

During training, we therefore want the network to **”learn from the future”**. When an error is observed at a later time step, we **propagate that error backward** through the sequence in order to adjust the parameters that were used at earlier steps. This allows the network to modify how it processes early inputs so that the hidden states they produce will be more useful for making accurate predictions later on.

Backpropagation Through Time (BPTT) formalizes this intuition. By unrolling the network through time, we obtain a feedforward computation network in which the same parameters appear at each time step. As in any feedforward network, we can then apply the chain rule to compute gradients. However, because the parameters are shared across time, the gradient with respect to each parameter is obtained by **summing its contributions across all time steps** in the unrolled network. In this way, the error at later outputs can be traced back through all intermediate hidden states, assigning responsibility to earlier computations. The network thus learns not only to predict well at each step, but to **build internal representations that support accurate predictions in the future**. Given an input sequence:

$X(0), X(1), \dots, X(T-1)$

the network performs a forward pass through the unrolled structure. At each time step, the hidden state is updated by combining the current input with the previous hidden state:

$H(t) = \phi(X(t)W_x + H(t-1)W_h + b)\,, \qquad H(-1) = H_0$

From the hidden state, the network produces an output. Here we explicitly include an linear implementeation of the g() function, defined by an output weight matrix and bias term:

$\hat{Y}(t) = g(H(t)) = H(t)W_y + b_y$

As the sequence is processed, this forward pass produces a sequence of hidden states:

$H(0), H(1), \dots, H(T-1)$

and a corresponding sequence of outputs

$\hat{Y}(0), \hat{Y}(1), \dots, \hat{Y}(T-1)$

Here the implementation of the forward pass through time:

```
# Phi activation function (ReLU)
def phi(Z):
    return np.maximum(0.0, Z)
```

```python
# g output function (linear)
def g(H, W_y, b_y):
    return H @ W_y + b_y

def rnn_layer_forward(X, params):

    # Get dimensions (T: time steps, B: batch size, D: input dimension)
    T, B, D = X.shape

    # Get hidden dimension
    hidden_dim = params["b"].shape[0]

    # Get output dimension
    output_dim = params["b_y"].shape[0]

    # Initialize hidden states
    H = np.zeros((T, B, hidden_dim))

    # Initialize outputs
    Y_hat = np.zeros((T, B, output_dim))

    # Set to zero the first previous hidden state
    H_prev = np.zeros((B, hidden_dim))

    # Unroll through time
    for t in range(T):

        # Update the hidden state
        H[t] = phi(X[t] @ params["W_x"] + H_prev @ params["W_h"] + params["b"])

        # Compute output
        Y_hat[t] = g(H[t], params["W_y"], params["b_y"])

        # Update previous hidden state
        H_prev = H[t]

    # Return hidden states and outputs
    return H, Y_hat
```

The predictions produced by the recurrent network should be **evaluated using a loss function**. Let

$Y(0), Y(1), \dots, Y(T-1)$

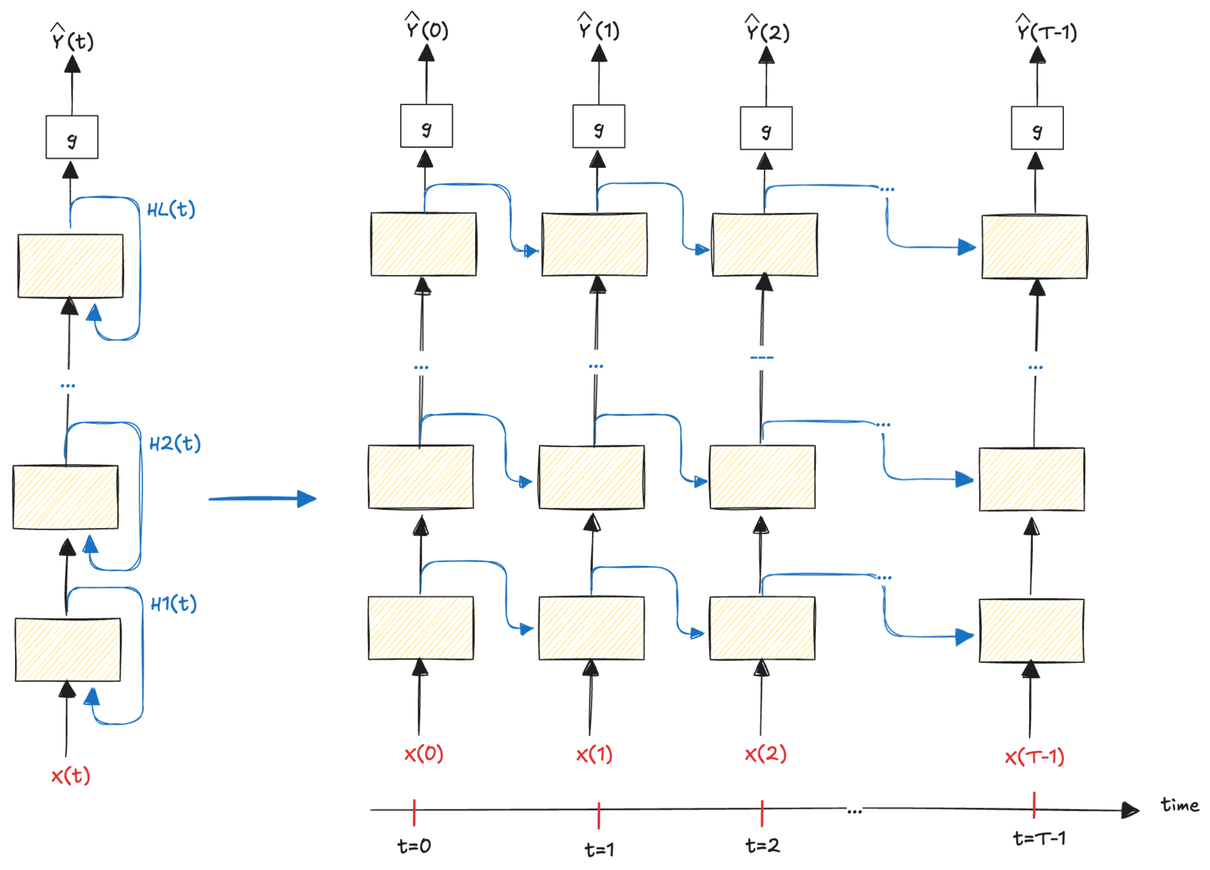
denote the **target sequence** associated with the input sequence. This sequence should represents **what we want the network to predict** at each time step. While the input sequence describes the information available to the model, the target sequence specifies the desired behavior of the model over time. In many problems, Y is naturally aligned with the input sequence. For example, in time-series forecasting, Y(t) may represent the future value of the signal relative to the input at time t. In a one-step-ahead forecasting task, this corresponds to

$$Y(t) = X(t+1)$$

so that each prediction is trained to anticipate what will happen next. More generally, Y can be interpreted as a **supervisory signal** that tells the network what information extracted from the past should be useful. Depending on the problem, the loss may involve all time steps (as in sequence-to-sequence problems) or only a subset of them (for example, only the final output in sequence-to-vector models). Let M denote the **number of time steps over which supervision is applied**. The training objective can then be written as:

$$\mathcal{L} = \sum_{t=T-M}^{T-1} \ell(Y(t), \hat{Y}(t))$$

where $\ell()$ is the loss evaluated at each time step (for example, mean squared error for regression or cross-entropy for classification) and M is the number of supervised time steps. In the following figure, for example, the early part of the sequence is used to build up a meaningful hidden state, while only the later part of the sequence is used to evaluate prediction quality:



To make this concrete, consider a regression task where the loss at each time step is the **squared error** between the predicted and target outputs. Since the network operates on mini-batches of B sequences and produces an O-dimensional output at each time step, the total number of scalar prediction errors over the supervised part of the sequence is MBO. We define the training objective as the sum of squared errors accumulated over the supervised time steps, the sequences in the mini-batch, and the output components:

$$\ell(Y(t), \hat{Y}(t)) = \sum_{b=1}^{B} \sum_{o=1}^{O} (\hat{Y}_{t,b,o} - Y_{t,b,o})^2$$

We can implement this loss computation in Python as follows:

```python
def sse_loss(Y_hat, Y, M):
    T = Y_hat.shape[0]
    residual = Y_hat[T-M:T] - Y[T-M:T]
    return np.sum(residual ** 2)
```

To train the model, we now need analyze **how the loss accumulated over time depends on parameters that are reused at every time step**, requiring a form of backpropagation that operates across time. Because the same parameters are reused at every time step, the gradient of the total loss is obtained by summing its contributions across all supervised time steps:

$$\frac{\partial \mathcal{L}}{\partial \omega} = \sum_{t=T-M}^{T-1} \frac{\partial \ell(Y(t), \hat{Y}(t))}{\partial \omega}$$

Applying the chain rule at a given time step t, we obtain:

$$\frac{\partial \ell(Y(t), \hat{Y}(t))}{\partial \omega} = \frac{\partial \ell(Y(t), \hat{Y}(t))}{\partial \hat{Y}(t)} \frac{\partial \hat{Y}(t)}{\partial H(t)} \frac{\partial H(t)}{\partial \omega}$$

This expression makes explicit how the effect of a parameter on the loss at time t can be decomposed into three parts: how the loss depends on the output, how the output depends on the hidden state, and how the hidden state depends on the parameter.

The first term is the derivative of the loss with respect to the output, and it is quite simple in the case of the squared errors used here. Over the supervised portion of the sequence the derivative is:

$$\frac{\partial \ell(Y(t), \hat{Y}(t))}{\partial \hat{Y}(t)} = 2(\hat{Y}(t) - Y(t))$$

whereas for t<T-M the derivative is zero. This term represents the **error signal injected at the output layer** at time t.

The second term is the derivative of the output with respect to the hidden state. Given the affine output mapping defined earlier, this derivative is simply the output weight matrix:

$$\frac{\partial \hat{Y}(t)}{\partial H(t)} = W_y$$

The nontrivial term in this expression is the derivative of the hidden state with respect to the parameters:

$$\frac{\partial H(t)}{\partial \omega}$$

which captures how a parameter influences the internal state of the network. Unlike in feed-forward networks, the hidden state is not a simple function of the parameters. Because of the recurrence:

$$H(t) = \phi(X(t)W_x + H(t-1)W_h + b)$$

The hidden state depends on the parameters both **directly** at time t (through their appearance in the affine transformation) and **indirectly** through the previous hidden state, which itself depend on th same parameters. To make this dependency explicit, it is convenient to introduce the pre-activation:

$$Z(t) = X(t)W_x + H(t-1)W_h + b$$

$$H(t) = \phi(Z(t))$$

Now, applying the chain rule:

$$\frac{\partial H(t)}{\partial \omega} = \frac{\partial H(t)}{\partial Z(t)} \frac{\partial Z(t)}{\partial \omega} + \frac{\partial H(t)}{\partial Z(t)} \frac{\partial Z(t)}{\partial H(t-1)} \frac{\partial H(t-1)}{\partial \omega} = ...$$

The first term captures the **direct contribution** of the parameter at time t, while the second term captures the **indirect contribution propagated from the past** through the recurrent connection. By recursively applying this relation backward in time, the effect of an error observed at a later time step is propagated to all earlier time steps that contributed to the construction of the hidden state:

$\displaystyle ... = \frac{\partial H(t)}{\partial Z(t)} \frac{\partial Z(t)}{\partial \omega} + \frac{\partial H(t)}{\partial Z(t)} \frac{\partial Z(t)}{\partial H(t-1)} \left(\frac{\partial H(t-1)}{\partial Z(t-1)} \frac{\partial Z(t-1)}{\partial \omega} + \frac{\partial H(t-1)}{\partial Z(t-1)} \frac{\partial Z(t-1)}{\partial H(t-2)} \frac{\partial H(t-2)}{\partial \omega} \right) = ... $

This expression shows that the total effect of a parameter on the hidden state at time t is obtained by summing its contributions from all previous time steps, each weighted by how changes in the hidden state propagate forward through time. However this mathematical expression is impractical to compute.

The key idea is rather than explicitly computing the derivative (which is cumbersome due to the temporal recursion), backpropagation proceeds by **propagating error signals backward** through the network. These signals measure **how sensitive the total loss is to intermediate variables** at each time step. We define the following error signals:

$$\delta_Y(t) = \frac{\partial \mathcal{L}}{\partial \hat{Y}(t)}$$

$$\delta_H(t) = \frac{\partial \mathcal{L}}{\partial H(t)}$$

$$\delta_Z(t) = \frac{\partial \mathcal{L}}{\partial Z(t)}$$

The first one is the **output error signal** and considering the sum of squared errors loss defined earlier, it is given by:

$$\delta_Y(t) = \frac{\partial \mathcal{L}}{\partial \hat{Y}(t)} = 2(\hat{Y}(t) - Y(t))$$

in t>=T-M and zero otherwise.

The seccond term is the **hidden-state error signal** and it receives two distinct contributions, which arise from the two different ways in which the hidden state influences the total loss. First, it **directly** affects the output at the same time step through the affine mapping:

$$\hat{Y}(t) = H(t)W_y + b_y$$

Applying the chain rule, this produces a contribution:

$$\left.\frac{\partial \mathcal{L}}{\partial H(t)}\right|_{\text{output}} = \frac{\partial \mathcal{L}}{\partial \hat{Y}(t)} \frac{\partial \hat{Y}(t)}{\partial H(t)} = \delta_Y(t) W_y^\top$$

Second, the hidden state also influences the loss **indirectly**, through its effect on all future hidden states. In particular, it appears in the pre-activation at the next time step:

$$Z(t+1) = X(t+1)W_x + H(t)W_h + b$$

and therefore affects the next hidden state and the chain rule yields the corresponding contribution:

$$\left.\frac{\partial \mathcal{L}}{\partial H(t)}\right|_{\text{future}} = \frac{\partial \mathcal{L}}{\partial Z(t+1)} \frac{\partial Z(t+1)}{\partial H(t)} = \delta_Z(t+1)\, W_h^\top$$

Combining the direct and indirect effects, the total hidden-state error signal becomes:

$$\delta_H(t) = \delta_Y(t) W_y^\top + \delta_Z(t+1) W_h^\top$$

which shows explicitly how errors are propagated from later to earlier time steps.

Finally, the error signal with respect to the pre-activation is obtained by propagating through the nonlinearity:

$$\delta_Z(t) = \frac{\partial \mathcal{L}}{\partial Z(t)} = \frac{\partial \mathcal{L}}{\partial H(t)} \frac{\partial H(t)}{\partial Z(t)} = \delta_H(t) \frac{\partial H(t)}{\partial Z(t)} = \delta_H(t) \frac{\partial \phi(Z(t))}{\partial Z(t)}$$

Because the activation function is applied component-wise:

$$H_i(t) = \phi(Z_i(t))$$

the partial derivative is a diagonal matrix:

$$\frac{\partial \phi(Z(t))}{\partial Z(t)} = \begin{bmatrix} \frac{\partial \phi(Z_1(t))}{\partial Z_1(t)} & 0 & \cdots & 0 \\ 0 & \frac{\partial \phi(Z_2(t))}{\partial Z_2(t)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial \phi(Z_m(t))}{\partial Z_m(t)} \end{bmatrix}$$

Thus, the pre-activation error signal can be computed by performing an element-wise multiplication (denoted by $\odot$) between the hidden-state error signal and the derivative of the activation function evaluated at the pre-activation:

$$\delta_Z(t) = \delta_H(t) \odot \phi'(Z(t))$$

In the particular case of the ReLU activation function, the derivative is given by:

$$\phi'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$

Putting everything together, we obtain the following backward recursion for t=T-1,T-2,...,0:

$$\delta_Y(t) = \begin{cases} 2(\hat{Y}(t) - Y(t)) & t \geq T - M \\ 0 & t < T - M \end{cases}$$

$$\delta_H(t) = \delta_Y(t) W_y^\top + \delta_Z(t+1) W_h^\top$$

$$\delta_Z(t) = \delta_H(t) \odot \mathbb{1}[Z(t) > 0]$$

Together, these equations define the complete backward recursion implemented by **Backpropagation Through Time** for a recurrent layer with ReLU activation and a sum-of-squared-errors loss. They make explicit how error signals are **injected at the outputs**, **propagated backward through the recurrent dynamics**, and **filtered by the activation function** at each time step.

Once the error signals have been computed, the gradients with respect to the model parameters can be obtained by differentiating. Because the same parameters are reused at every time step, their total gradient is obtained by **summing the contributions from all time steps**. The output parameters:

$$\omega \in \{W_y, b_y\}$$

appear only in the affine transformation, therefore:

$$\frac{\partial \mathcal{L}}{\partial \omega} = \sum_{t=0}^{T-1} \frac{\partial \mathcal{L}}{\partial \hat{Y}(t)} \frac{\partial \hat{Y}(t)}{\partial \omega} = \sum_{t=0}^{T-1} \delta_Y(t) \frac{\partial \hat{Y}(t)}{\partial \omega}$$

so, the gradients are obtained by multiplying the output error signal by the derivative of the output function:

$$\frac{\partial \hat{Y}(t)}{\partial W_y} = H(t) \ \rightarrow \ \frac{\partial \mathcal{L}}{\partial W_y} = \sum_{t=0}^{T-1} H(t)^\top \delta_Y(t)$$

$$\frac{\partial \hat{Y}(t)}{\partial b_y} = I \ \rightarrow \ \frac{\partial \mathcal{L}}{\partial b_y} = \sum_{t=0}^{T-1} \sum_{b=1}^{B} \delta_Y(t)_{b,:}$$

The parameters:

$$\omega \in \{b, W_x, W_h\}$$

appear only inside the pre-activation, therefore:

$$\frac{\partial \mathcal{L}}{\partial \omega} = \sum_{t=0}^{T-1} \frac{\partial \mathcal{L}}{\partial Z(t)} \frac{\partial Z(t)}{\partial \omega} = \sum_{t=0}^{T-1} \delta_Z(t) \frac{\partial Z(t)}{\partial \omega}$$

and tge gradients are obtained by multiplying the hidden error signal by the derivative of the affine map:

$$\frac{\partial Z(t)}{\partial b} = I \ \rightarrow \ \frac{\partial \mathcal{L}}{\partial b} = \sum_{t=0}^{T-1} \sum_{b=1}^{B} \delta_Z(t)_{b,:}$$

$$\frac{\partial Z(t)}{\partial W_x} = X(t) \rightarrow \frac{\partial \mathcal{L}}{\partial W_x} = \sum_{t=0}^{T-1} X(t)^\top \delta_Z(t)$$

$$\frac{\partial Z(t)}{\partial W_h} = H(t-1) \rightarrow \frac{\partial \mathcal{L}}{\partial W_h} = \sum_{t=0}^{T-1} H(t-1)^\top \delta_Z(t)$$

However, since H(-1) is not defined, we neet to initialize it to some value, typically zero. Here's how we can implement BPTT in Python for our recurrent layer:

```python
import numpy as np

def rnn_layer_backward(X, Y, H, Y_hat, params, M):

    # Dimensions
    T, B, D = X.shape
    H_dim = params["W_h"].shape[0]
    O_dim = params["W_y"].shape[1]

    # Gradient accumulators for shared parameters
    grads = {k: np.zeros_like(v, dtype=np.float64) for k, v in params.items()}

    # delta_Z(t+1) initialization
    delta_Z_future = np.zeros((B, H_dim), dtype=np.float64)

    for t in reversed(range(T)):

        # delta_Y
        if t ⩾ T - M:
            delta_Y = 2.0 * (Y_hat[t] - Y[t])
        else:
            delta_Y = np.zeros((B, O_dim), dtype=np.float64)

        #delta_H
        delta_H = delta_Y @ params["W_y"].T + delta_Z_future @ params["W_h"].T

        # delta_Z
        relu_mask = (H[t] > 0.0).astype(delta_H.dtype)
        delta_Z = delta_H * relu_mask

        # Accumulate gradients
        grads["W_y"] += H[t].T @ delta_Y
        grads["b_y"] += delta_Y.sum(axis=0)
        grads["b"]   += delta_Z.sum(axis=0)
        grads["W_x"] += X[t].T @ delta_Z
        grads["W_h"] += H[t-1].T @ delta_Z if t > 0 else 0

        # Prepare delta_Z for the next iteration
        delta_Z_future = delta_Z

    return grads
```

After computing the gradients, the model parameters are updated using Stochastic Gradient Descent, moving them in the direction of decreasing loss using the learning rate to control the step size:

$w \leftarrow w - \eta \nabla_w L$

```
def sgd_step(params, grads, lr=1e-3):
    for k in params:
        params[k] -= lr * grads[k]
```

Finally, we define a function that performs **a single training step**. Given an input sequence, it first runs a forward pass through time to compute the hidden states and the corresponding predictions at each time step. These predictions are then compared with the target sequence to evaluate the loss over the supervised time window. Next, BPTT is used to compute the gradients of the loss with respect to all model parameters, accounting for the fact that the same parameters are reused across time. The parameters are then updated with a single stochastic gradient descent step. The function returns the loss value, which provides a convenient scalar metric for monitoring training progress.

```
def rnn_layer_train_step(X, Y, params, M, lr=1e-3):

    # Forward pass through time
    H, Y_hat = rnn_layer_forward(X, params)

    # Compute loss over the last M time steps
    loss = sse_loss(Y_hat, Y, M)

    # Backward pass through time (BPTT)
    grads = rnn_layer_backward(X, Y, H, Y_hat, params, M)

    # Update parameters with SGD
    sgd_step(params, grads, lr)
    # Return scalar loss
    return loss
```

In practice, additional complications arise because **input sequences can be very long**. This creates challenges both from a **computational** standpoint, due to the large amount of memory required to store intermediate activations, and from an **optimization** standpoint, due to numerical instabilities in the gradient computation. Information originating from early time steps must pass through many successive matrix multiplications before influencing later outputs, and an equally long chain of matrix products appears when gradients are propagated backward through time. As a result, gradients may either **vanish** or **explode**, making learning difficult.

To mitigate these issues, it is common practice to truncate the unrolling of the network during backpropagation. Rather than propagating gradients all the way back to the beginning of the sequence, **the backward pass is limited to a fixed number of time steps**. This hyperparameter, known as the **truncation length**, controls how far into the past error signals are allowed to propagate.

A **larger truncation length** enables the model to capture longer-range temporal dependencies, but comes at the cost of increased memory usage and computational complexity. Conversely, a **smaller truncation length** reduces resource requirements and improves numerical stability, but may prevent the network from learning from distant past information. In practice, the truncation length is treated as a hyperparameter and chosen based on the characteristics of the data and the task, **balancing representational power against computational efficiency**.

1.3.5 Applying RNNs to Time Series Forecasting

We can use the RNN architecture to build a model for time-series forecasting, where the objective is to predict future values of a sequence based on its past observations. In this setting, we adopt **a sequence-to-sequence formulation**: the network receives a window of past ridership values as input and produces a sequence of **one-step-ahead predictions**, so that at each time step it learns to forecast the next value in the series.

To ensure a fair comparison with the previously discussed naive baseline and SARIMA model, we use the same evaluation period as a test set, while earlier data are split into training and validation periods. Before training the model, the ridership values are **normalized** by applying a simple rescaling. This brings the data into a numerically convenient range, which improves the stability of training, reduces the risk of excessively large gradients, and allows the optimization to focus on learning temporal patterns rather than compensating for large magnitudes. Importantly, this normalization does not change the temporal structure of the series and can be inverted after forecasting, so that predictions can be reported in the original units.

```
start_date = "2019-03-01"
end_date = "2019-05-31"

# Test data
rail_test = df["rail"][start_date:end_date] / 1e6

# Split other data into training and validation periods
rail_train = df["rail"]["2016-01-01":"2018-12-31"] / 1e6
rail_valid = df["rail"]["2019-01-01":"2019-02-28"] / 1e6
```

To apply the RNN model, the time series must first be transformed into **input–target sequences**. Each input sequence consists of the ridership values observed over the past T days, while the corresponding target sequence consists of the same window shifted one step into the future. In other words, for an input sequence:

$x(t), x(t+1), \ldots, x(t+T-1)$

the target sequence is:

$x(t+1), x(t+2), \ldots, x(t+T)$

In this formulation, the RNN processes a window of historical observations and learns to predict the next value at every time step, providing dense supervision across the entire sequence.

The choice of **T determines how much historical information the model can exploit**. It should be **large enough to capture the dominant temporal patterns** in the data, such as weekly seasonality, but **not so large as to introduce unnecessary model complexity or unstable gradient propagation** during training. In practice, a reasonable starting point for T is one or two full seasonal cycles (for example, T = 7 or T = 14 days for weekly patterns), and the optimal value can be selected based on validation performance.

The following function illustrates how a time series can be translated into a collection of sequence-to-sequence input–target pairs suitable for training an RNN:

```
import numpy as np

def make_sequences(series, T):

    arr = np.asarray(series)
    if arr.ndim == 1:
        arr = arr[:, None]

    N, D = arr.shape
    B = N - T

    X = np.zeros((T, B, D))
    Y = np.zeros((T, B, D))

    for b in range(B):
        X[:, b, :] = arr[b : b + T]
        Y[:, b, :] = arr[b + 1 : b + T + 1]

    return X, Y
```

We use this function to transform the time series into **training**, **validation**, and **test datasets**, each composed of **input sequences of past observations** and their corresponding **target values** to be predicted:

```
# Define sequence length
T = 14

# Create input-target sequences for training, validation, and test sets
X_train, Y_train = make_sequences(rail_train, T)
X_valid, Y_valid = make_sequences(rail_valid, T)
X_test,  Y_test  = make_sequences(rail_test,  T)
```

We use a recurrent neural network composed of **a single recurrent layer** followed by a linear output layer. Since the ridership series is one-dimensional, each time step provides a single scalar input and the input dimension is therefore **d=1**. At each time step, the recurrent layer updates a hidden state of dimension **m=20**, which acts as a distributed memory summarizing the relevant information extracted from past observations. The output dimension is **p=1**, because the model produces a scalar prediction at each time step.

In the sequence-to-sequence formulation adopted here, this forecast represents the next-day ridership value at each time step within the input window. All model parameters (input-to-hidden weights, recurrent weights, recurrent bias, output weights, and output bias) are initialized randomly and then learned during training.

```python
# Define RNN model dimensions
input_dim  = 1
hidden_dim = 20
output_dim = 1

# Initialize RNN layer parameters
params = init_rnn_layer(input_dim, hidden_dim, output_dim)
```

We implement a **mini-batch iterator** to train the RNN efficiently. Given a set of input sequences and their corresponding target sequences, the iterator first constructs an array of sample indices and **randomly shuffles** it in order to randomize the order of the sequences seen during training. Importantly, this shuffling is performed **across sequences**, not within sequences: the temporal ordering inside each individual sequence is preserved, since maintaining the correct time structure is essential for recurrent models.

After shuffling, the dataset is partitioned into consecutive mini-batches of a specified size. For each mini-batch, the iterator yields a pair consisting of a batch of input sequences and the corresponding batch of target sequences, which can then be processed in parallel during the forward and backward passes.

```python
def iterate_minibatches(X, Y, batch_size, shuffle=True):
    """
    Iterate over mini-batches of sequences.

    X, Y: arrays of shape (T, B, D)
    Yields:
        X_batch, Y_batch of shape (T, B_batch, D)
    """
    T, B, D = X.shape

    indices = np.arange(B)
    if shuffle:
        np.random.shuffle(indices)

    for start in range(0, B, batch_size):
        batch_idx = indices[start:start + batch_size]
        yield X[:, batch_idx, :], Y[:, batch_idx, :]
```

Finally we can implement the **training loop**. The model is trained for a fixed number of epochs, where each epoch corresponds to one full pass over the training dataset. At each epoch, the training data are randomly partitioned into mini-batches of fixed size. For each mini-batch, the input and target tensors are transposed into the format expected by the RNN. A forward pass through the unrolled network is then performed to compute predictions at all time steps, the loss is evaluated over the supervised portion of the sequence (corresponding to the last M time steps), and the model parameters are updated by applying BPTT followed by a stochastic gradient descent step. In the sequence-to-sequence setting considered here, the network produces a prediction at every time step, and supervision can be applied either at all time steps or only over the final part of the sequence, depending on the choice of M.

```python
epochs = 250
batch_size = 32
lr = 1e-3
M = T  # full sequence-to-sequence supervision

epoch_mean_losses = []

for epoch in range(epochs):

    epoch_losses = []

    for Xb, Yb in iterate_minibatches(X_train, Y_train, batch_size, shuffle=True):

        loss = rnn_layer_train_step(Xb, Yb, params, M, lr=lr)
        epoch_losses.append(loss)

    epoch_loss = float(np.mean(epoch_losses))
    epoch_mean_losses.append(epoch_loss)

    print(f"\rEpoch {epoch+1:02d}/{epochs} | Train loss: {epoch_loss:.6f}", end="",
    ↪  flush=True)
```

```
Epoch 500/500 | Train loss: 3.341283
```

We evaluate the trained RNN on the held-out test period by generating one-step-ahead forecasts for every test time and comparing them with the corresponding ground truth. Forecasting is performed in a rolling fashion: for each test time t we build an input window containing the T observations immediately preceding t, feed that window through the trained network, and retain the model's last output \hat y(t) as the one-step forecast. Because each forecast uses the preceding T days, the very first test forecast requires the T days immediately before the test start; forecasting therefore cannot start from the first test day without historical context. This procedure guarantees that every prediction is produced using only information that would have been available at decision time.

```python
# Full rail series up to end_date (includes the test period, but we will never index
↪  into the future)
rail_full = (df["rail"].loc[:end_date] / 1e6).values

# Number of test points
n_test = len(rail_test)

# Index of the first test point within rail_full
test_start_idx = len(rail_full) - n_test

y_preds_rnn = []

for k in range(n_test):
    idx = test_start_idx + k  # index of the value we want to predict

    # past window of length T immediately before idx
    x_input = rail_full[idx - T: idx]
    x_input = x_input.reshape(T, 1, 1)

    _, Y_hat = rnn_layer_forward(x_input, params)
    y_preds_rnn.append(float(Y_hat[-1, 0, 0]))
```

We compute the MAE between the RNN forecasts and the true ridership values on the test set in order to assess the model's predictive accuracy and to compare its performance with the previously discussed naive baseline and SARIMA models. All forecasts are first mapped back to the original (**unnormalized**) scale before computing error statistics:

```python
# Use the test index
test_index = df.loc[start_date:end_date].index

# Convert predictions to a Pandas Series and unnormalize
y_preds_rnn = pd.Series(y_preds_rnn, index=test_index)
y_preds_rnn_unscaled = y_preds_rnn * 1e6

# True values (unscaled) as a Pandas Series
y_true_unscaled = df.loc[start_date:end_date, "rail"]

# Compute RNN MAPE
mape_rnn = ((y_preds_rnn_unscaled - y_true_unscaled) / y_true_unscaled).abs().mean()

# Display the MAPE results
print(f"Naive MAPE:  {mape_naive:.2%}")
print(f"SARIMA MAPE: {mape_sarima:.2%}")
print(f"RNN MAPE:    {mape_rnn:.2%}")
```

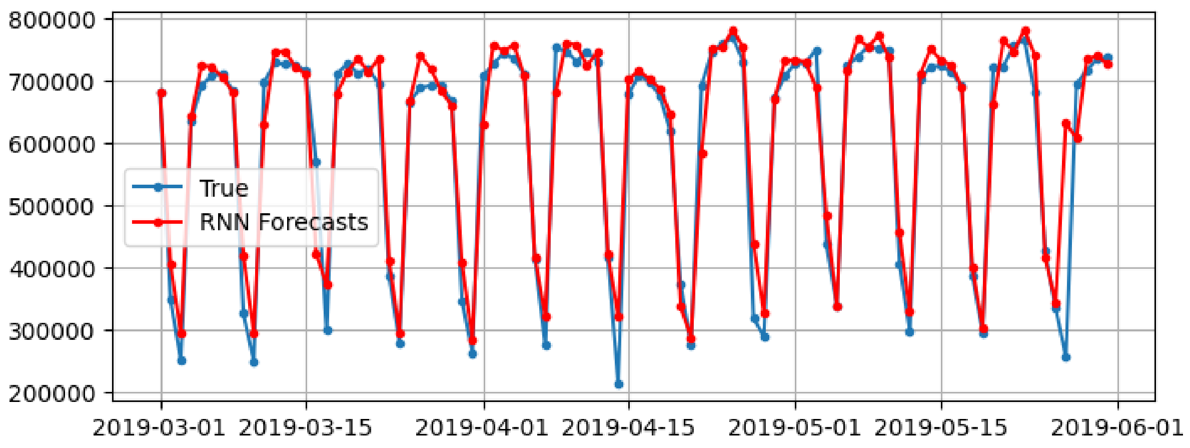
```
Naive MAPE:  8.99%
SARIMA MAPE: 7.54%
RNN MAPE:    7.51%
```

The rail ridership time series exhibits a very strong and highly regular weekly seasonality, combined with smooth dynamics and limited nonlinear structure. In such settings, **simple models that explicitly encode this regularity are often extremely difficult to outperform**. The naive forecasting strategy, which directly copies the value from one week earlier, effectively hard-codes the correct inductive bias about the data-generating process. In contrast, the RNN must **learn** this behavior implicitly from data through gradient-based optimization. Even though the RNN is, in principle, a far more expressive model, this makes **the learning problem substantially harder**. Moreover, in this experiment, we focus on demonstrating the modeling and training procedure, and **we do not explicitly address potential overfitting**; in practice, techniques such as validation-based early stopping, regularization, or dropout would be required to control model complexity and improve generalization.

It is worth noting that the performance of the RNN **depends on the choice of the sequence length T and on the loss window M** used during training. Varying T changes how much historical context is provided to the model, while varying M determines how many of the model's predictions are directly supervised. In principle, increasing T allows the network to access longer-term information, whereas reducing M can focus learning on the most recent predictions. However, in practice, these choices often involve a **delicate trade-off between representational capacity, optimization stability, and effective gradient propagation**. Exploring different values of T and M can therefore provide useful insight into the model's behavior.

We can plot the forecasts produced by the RNN alongside the actual observed values to visually assess its performance:

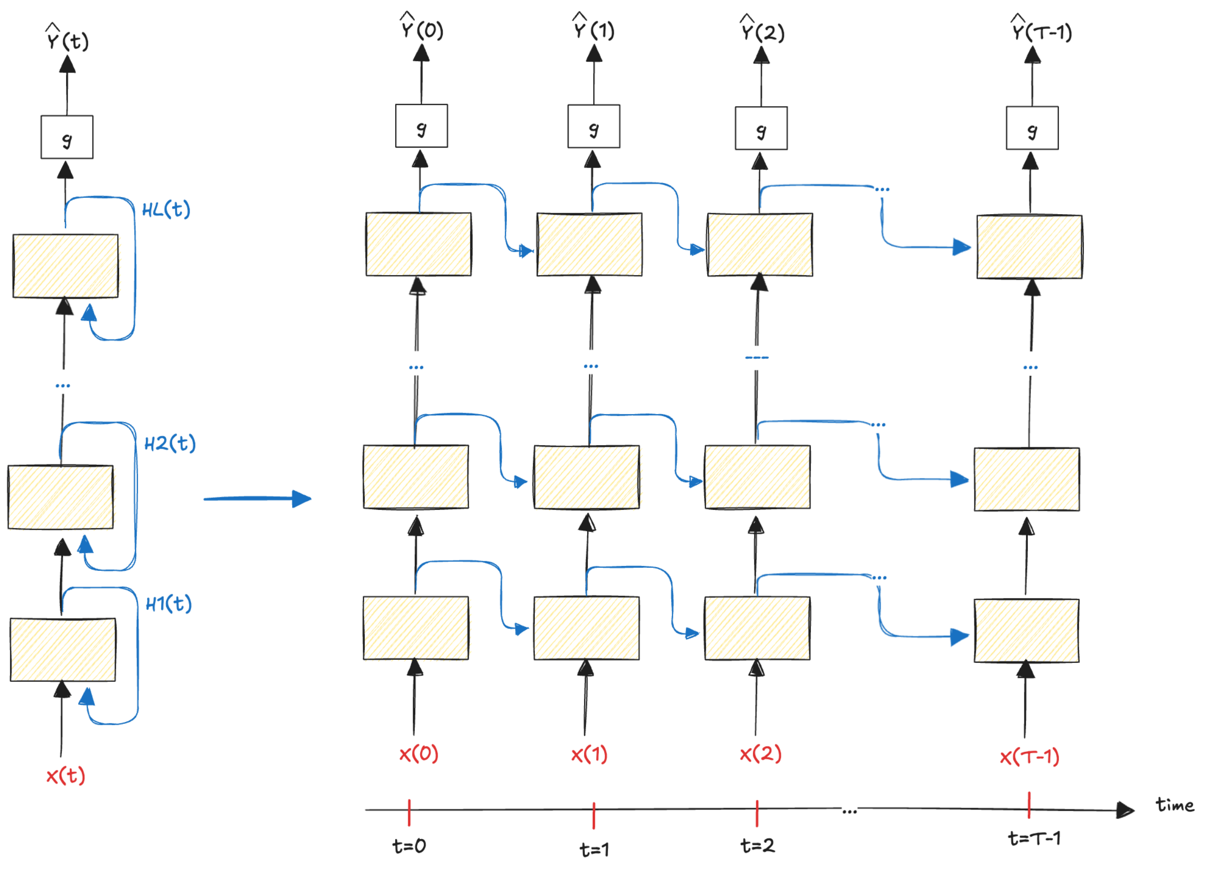
```
fig, ax = plt.subplots(figsize=(8, 3))

ax.plot(rail_series.loc[time_period].index,
        rail_series.loc[time_period].values,
        marker=".", label="True",
)

ax.plot(y_preds_rnn_unscaled.index,
        y_preds_rnn_unscaled.values,
        color="red", marker=".", label="RNN Forecasts"
)

ax.grid(True)
ax.legend()
plt.show()
```



From a modeling perspective, the network used in this experiment is a **vanilla recurrent neural network**. Its practical ability to exploit long-term dependencies is limited by the well-known **vanishing gradient problem**. As gradients are propagated backward through time, they are repeatedly multiplied by the recurrent weight matrix and by the derivative of the activation function, which causes their magnitude to **shrink rapidly**. As a consequence, the effective temporal memory of the network is often much shorter than the nominal sequence length, making it **difficult for a simple RNN to reliably capture seasonal patterns** that extend over many time steps.

1.3.6 Deep networks

It is common to increase the representational capacity of recurrent networks by **stacking multiple recurrent layers**. In a stacked RNN, each layer contains its own hidden state and is unrolled through time in the same way as a single-layer model:



At a given time step, the first recurrent layer processes the input and produces a hidden representation. This representation is then fed as input to the second recurrent layer, producing a second hidden representation, and so on up to the top layer. In other words, information flows forward in time within each layer and upward across layers at each time step, while the parameters within each layer are shared across all time steps. Formally, the computation can be written as:

$$H^{(\ell)}(t) = \phi(H^{(\ell-1)}(t)W_x^{(\ell)} + H^{(\ell)}(t-1)W_h^{(\ell)} + b^{(\ell)})$$

with the output typically computed from the top-layer state:

$$\hat{Y}(t) = g(H^{(L)}(t))$$

This stacked structure allows higher layers to build **progressively more abstract temporal features**, often improving performance when the underlying dynamics are complex.

We now implement the stacked recurrent neural network using **PyTorch**, which provides built-in support for this architecture through the **nn.RNN module**, where the number of stacked layers is controlled by a single parameter:

```
import torch
import torch.nn as nn

# Input size, just the rail ridership
input_size  = 1

# Number of hidden units in each RNN layer
hidden_size = 32

# Number of recurrent layers
num_layers  = 3

# Output size. just the rail ridership prediction
output_size = 1

# Define the recurrent layers
deep_rnn = nn.RNN(input_size=input_size,
             hidden_size=hidden_size,
             num_layers=num_layers,
             nonlinearity="tanh",
             batch_first=True       # inputs: (B, T, D)
)

# Define the fully connected layer
fc = nn.Linear(hidden_size, output_size)

#Define the optimizer
optimizer = torch.optim.Adam(list(deep_rnn.parameters()) +
                             list(fc.parameters()),
                             lr=1e-3)

# Define the forward function
def forward(x):
    out, _ = deep_rnn(x)
    y_hat = fc(out)
    return y_hat
```

We need to write the loss we already defined using the PyTorch tensors instead of NumPy arrays:

```python
def mse_loss_torch(y_hat, y, M):
    return 0.5 * ((y_hat[:, -M:, :] - y[:, -M:, :]) ** 2).sum()
```

We now define a function that performs a single training step for the deep network: given a mini-batch of input–target sequences, it executes a forward pass, computes the loss over the supervised time steps, propagates gradients backward through time, and updates the model parameters:

```python
def train_step(Xb, Yb, deep, fc, forward, M):

    # Set the model to training mode
    deep.train()
    fc.train()

    # Prepare mini-batch tensors
    xb = torch.as_tensor(Xb, dtype=torch.float32)
    yb = torch.as_tensor(Yb, dtype=torch.float32)

    # Clear previous gradients
    optimizer.zero_grad()

    # Forward pass
    y_hat = forward(xb)

    # Compute loss over the last M time steps
    loss = mse_loss_torch(y_hat, yb, M)

    # Backward pass
    loss.backward()

    # Update parameters
    optimizer.step()

    # Return scalar loss
    return float(loss.detach())
```

We can now train the stacked RNN by iterating over the training data for multiple epochs and applying the training step to each mini-batch, monitoring the average training loss at the end of each epoch:

```python
# Number of training epochs
epochs = 80

# Minibatch size
batch_size = 32

# Loss window
M = T

# List to store losses
losses = []

# Training loop
for epoch in range(1, epochs + 1):

    for Xb, Yb in iterate_minibatches(X_train, Y_train, batch_size):
        loss = train_step(Xb, Yb, deep_rnn, fc, forward, M)
        losses.append(loss)

    # Print epoch loss
    print(f"\rEpoch {epoch+1:02d}/{epochs} | Train loss: {np.mean(losses):.6f}",
    ↪  end="", flush=True)
```

```
Epoch 81/80 | Train loss: 2.7165363
```

We now evaluate the trained deep RNN on the test set by rolling the model forward to generate one-step-ahead forecasts at each test time step and computing the MAE on the original ridership scale, in direct comparison with the naive, SARIMA and simple RNN baselines:

```python
# Predictions list
y_preds_deep_rnn = []

# Prepare the model for evaluation
deep_rnn.eval()
fc.eval()

with torch.no_grad():
    for t in range(len(rail_test)):

        # Index of current test time in rail_full
        idx = len(rail_full) - len(rail_test) + t
        # # Extract last T observations before time t
        x_input = rail_full[idx - T:idx]

        # To tensor (B=1, T, 1)
        x_tensor = torch.tensor(x_input.reshape(1, T, 1), dtype=torch.float32)

        # Forward pass
        Y_hat = forward(x_tensor)

        # One-step-ahead prediction
        y_pred = Y_hat[0, -1, 0].item()
        y_preds_deep_rnn.append(y_pred)

# Convert to numpy and unnormalize to original units
y_preds_deep_rnn = np.array(y_preds_deep_rnn)
y_preds_deep_rnn_unscaled = y_preds_deep_rnn * 1e6

# Compute MAE
mae_deep_rnn = np.mean(np.abs(y_preds_deep_rnn_unscaled - y_true_unscaled))

# Print MAE
print("Naive MAE :", mae_naive)
print("SARIMA MAE:", mae_sarima)
print("RNN MAE   :", mae_rnn)
print("Deep RNN MAE:", mae_deep_rnn)
```

```
Naive MAE : 42143.27173913043
SARIMA MAE: 32040.720095473138
RNN MAE   : 36396.27665014888
Deep RNN MAE: 39366.48723723577
```

Stacking multiple recurrent layers has performances comparable to SARIMA and simple RNN, highlighting that increased model complexity alone is not sufficient to surpass well-specified statistical models when the underlying data structure is simple and highly regular.

Overall, these results reinforce a key lesson in time-series forecasting: **model choice should be guided by the structure of the data rather than by model complexity alone**. While deep learning models offer great flexibility, classical approaches that explicitly encode known regularities can remain superior in settings dominated by strong seasonality and stable dynamics.

1.4 Extending to Multivariate and Multi-Step Forecasting

So far, we have focused on forecasting a univariate time series using one-step-ahead predictions. While this setting is useful for introducing recurrent neural networks and for understanding their strengths and limitations, many **real-world forecasting problems** are more complex. In practice, predictions often depend on **multiple correlated time series**, and the goal is frequently to **forecast several future time steps simultaneously** rather than a single value.

1.4.1 Multivariate Input

A **multivariate time series** consists of **multiple interrelated sequences** observed over time. At each time step, the data are represented by a **vector** of observations, where each component corresponds to a different variable or feature. These variables often capture **complementary aspects** of the underlying process and may provide additional predictive power when **modeled jointly**.

For example, in the context of the public transportation ridership example, we may observe separate time series for bus and rail usage. While each series reflects distinct travel behaviors, they are clearly related and may influence one another over time.

From a modeling perspective, multivariate time series can be handled with minimal changes to the recurrent network architecture. The main difference lies in the input representation: **instead of feeding a single scalar at each time step, the network receives a vector of features**.

To illustrate this setting, we consider forecasting the rail ridership time series while using both bus and rail ridership as inputs to the model. This allows the RNN to **exploit cross-dependencies** between variables when generating forecasts.

```
df_mulvar = df[["bus", "rail"]] / 1e6  # use both bus & rail series as input

df_mulvar.head()
```

```
                 bus      rail
date
2001-01-01  0.297192  0.126455
2001-01-02  0.780827  0.501952
2001-01-03  0.824923  0.536432
2001-01-04  0.870021  0.550011
2001-01-05  0.890426  0.557917
```

We split the multivariate time series into training, validation, and test sets using the same chronological boundaries adopted in the univariate experiments, ensuring a fair and consistent evaluation of the multivariate forecasting model:

```
mulvar_train = df_mulvar["2016-01":"2018-12-31"]
mulvar_valid = df_mulvar["2019-01-01":"2019-02-28"]
mulvar_test = df_mulvar["2019-03-01":"2019-05-31"]
```

```
# Define sequence length
T = 14

# Create input-target sequences for training, validation, and test sets
X_train_mulvar, Y_train_mulvar = make_sequences(mulvar_train, T)
X_valid_mulvar, Y_valid_mulvar = make_sequences(mulvar_valid, T)
X_test_mulvar,  Y_test_mulvar  = make_sequences(mulvar_test,  T)
```

We now define a **stacked recurrent neural network for multivariate forecasting**, where each time step is represented by a vector of inputs (bus and rail ridership), and the network produces a sequence of predictions for the target variable through a linear readout applied to the hidden states:

```
# Multivariate input: [bus, rail]
input_size  = 2

# Number of hidden units in each RNN layer
hidden_size = 32

# Number of recurrent layers
num_layers = 3

# Output size: we predict rail only
output_size = 1

# Define the recurrent layers
deep_multi_rnn = nn.RNN(
    input_size=input_size,
    hidden_size=hidden_size,
    num_layers=num_layers,
    nonlinearity="tanh",
    batch_first=True        # (B, T, D)
)
```

```
# Define the fully connected layer
fc = nn.Linear(hidden_size, output_size)

#Define the optimizer
optimizer = torch.optim.Adam(list(deep_multi_rnn.parameters()) +
                             list(fc.parameters()),
                             lr=1e-3)

# Define the forward function
def forward_multi(x):
    out, _ = deep_multi_rnn(x)       # (B, T, H)
    out_last = out[:, -1, :]         # (B, H)
    y_hat = fc(out_last)             # (B, 1)
    return y_hat
```

Notice that the only difference with respect to the univariate model is the **input dimensionality**. At each time step, the model now receives two input features instead of one: one for bus ridership and one for rail ridership. In that case we ooutput only the rail ridership forecast, however the model could easily be extended to **produce forecasts for both series simultaneously** by adjusting the output layer accordingly. The train step is equivalent to the previous one, since the architecture is the same and

```
# Number of training epochs
epochs = 80

# Minibatch size
batch_size = 32

# Loss window
M = T

# List to store losses
losses = []

# Training loop
for epoch in range(1, epochs + 1):

    for Xb, Yb in iterate_minibatches(X_train_mulvar, Y_train_mulvar, batch_size):
        loss = train_step(Xb, Yb, deep_multi_rnn, fc, forward_multi, M)
        losses.append(loss)
    # Print epoch loss (overwrite same line)
    print(
        f"\rEpoch {epoch:03d}/{epochs} | Train loss: {np.mean(losses):.6f}", end="",
         ↪  flush=True
    )
```

In general, using a single model for multiple related tasks often can results in better performance than using a separate model for each task, since features learned for one task may be useful for the other tasks, and also because having to perform well across multiple tasks prevents the model from overfitting (it's a form of regularization). Not in that case.

1.4.2 Forecasting Several Steps Ahead

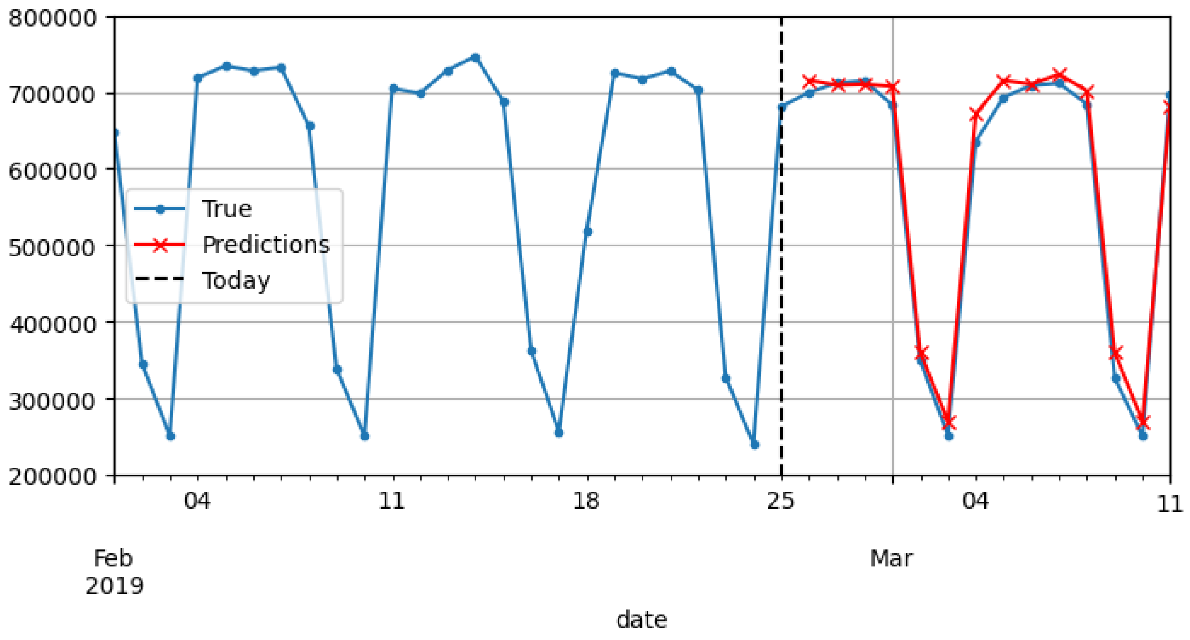
So far we have only predicted the value at the next time step, and we could just as easily predict the value several steps ahead by changing the target appropriately (for example, to predict the ridership 2 weeks from now, we could just change the target to be the value 14 days ahead instead of 1 day ahead). But what if we want to predict the next N values?

The first option is called **recursive multi-step forecasting**: take the sequece-to-vector model, make it predict the next value, add that value to the inputs, and acting as if the predicted value had actually occurred, finally, use the model again to predict the following value, and so on:

```
import numpy as np

X = rail_valid.to_numpy()[np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)
```

We can plot the resulting forecasts, for example for the rail:

```
# Forecasts from 2019-02-26 (the 57th day of 2019) to 2019-03-11 (14 days in total)
Y_pred = pd.Series(X[0, -14:, 0], index=pd.date_range("2019-02-26", "2019-03-11"))

fig, ax = plt.subplots(figsize=(8, 3.5))
(rail_valid * 1e6)["2019-02-01":"2019-03-11"].plot(label="True", marker=".", ax=ax)
(Y_pred * 1e6).plot(label="Predictions", grid=True, marker="x", color="r", ax=ax)
ax.vlines("2019-02-25", 0, 1e6, color="k", linestyle="--", label="Today")
ax.set_ylim([200_000, 800_000])
plt.legend(loc="center left")

plt.show()
```



Notice that if the model makes an error at one time step, then the forecasts for the following time steps are impacted as well, so the **errors tend to accumulate**. So, it's preferable to use this technique only for a small number of steps.

The second option is the **direct multi-step forecasting**: we can still use a sequence-to-vector model, but we train the RNN to predict the next N values in one shot:

However, we first **need to change the dataset in order to have targets as vectors** containing the next N values. To do this, we can use again timeseries_dataset_from_array(), but this time asking it to create datasets with longer sequences (seq_length + N) and without targets (targets=None). Then we can use map() method to apply a custom function to each batch of sequences, splitting them into inputs (seq_length) and targets (N).

```python
N = 14

def split_inputs_and_targets(mulvar_series, ahead=14, target_col=1):
    return mulvar_series[:, :-ahead], mulvar_series[:, -ahead:, target_col]

ahead_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    batch_size=32,
    shuffle=True,
).map(split_inputs_and_targets)

ahead_valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    batch_size=32
).map(split_inputs_and_targets)
```

Now we just need the output layer to have N units instead of 1:

```python
ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 2]),
    tf.keras.layers.Dense(14)
])
```

```python
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
ahead_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

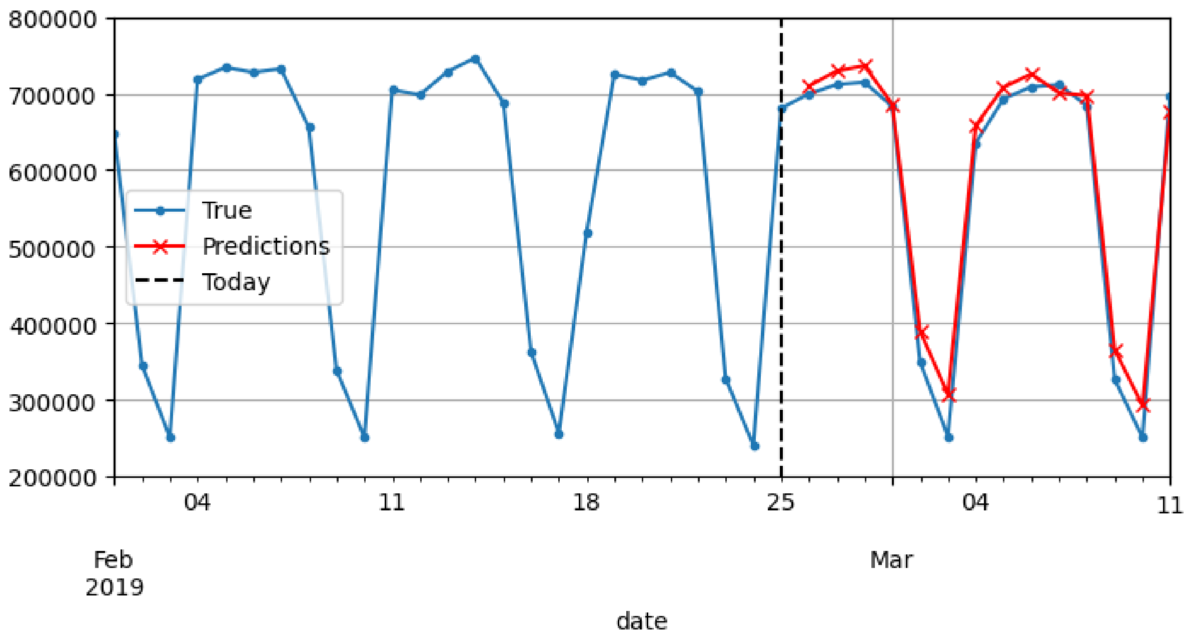
```python
history = ahead_model.fit(ahead_train_ds, validation_data=ahead_valid_ds, epochs=500,
 ↪  callbacks=[early_stopping_cb])
```

```python
valid_loss, valid_mae = ahead_model.evaluate(ahead_valid_ds)
print("Validantion MAE: ", valid_mae * 1e6)
```

After training this model, we can predict the next N values at once:

```
X = mulvar_valid.to_numpy()[np.newaxis, :seq_length]  # shape [1, 56, 5]
Y_pred = ahead_model.predict(X)  # shape [1, 14]

print(Y_pred)
```

```
Y_pred = pd.Series(Y_pred[0], index=pd.date_range("2019-02-26", "2019-03-11"))
fig, ax = plt.subplots(figsize=(8, 3.5))
(mulvar_valid.rail * 1e6)["2019-02-01":"2019-03-11"].plot(label="True", marker=".",
 ↪  ax=ax)
(Y_pred * 1e6).plot(label="Predictions", grid=True, marker="x", color="r", ax=ax)
ax.vlines("2019-02-25", 0, 1e6, color="k", linestyle="--", label="Today")
ax.set_ylim([200_000, 800_000])
plt.legend(loc="center left")

plt.show()
```



The forecasts of this approach for the next day are obviously better than its forecasts for N days into the future, but it doesn't accumulate errors like the previous one.

The third option is **using a sequence-to-sequence model**: instead of training the model to forecast the next N values only at the very last time step, we can train it to forecast the next N values at each and every time step.

The advantage of this technique is that the loss will contain a term for the output of the RNN at each and every time step, not just for the output at the last time step. This means there will be many more error gradients flowing through the model, and they won't have to flow through time as much since they will come from the output of each time step, not just the last one. This will both stabilize and speed up training.

To be clear, at time step 0 the model will output a vector containing the forecasts for time steps 1 to N, then at time step 1 the model will forecast time steps 2 to N+1, and so on. In other words, the targets are sequences of consecutive windows, shifted by one time step at each time step. The target is not a vector anymore, but a sequence of the same length as the inputs, containing a N-dimensional vector at each step.

Preparing the datasets is not trivial, since each instance has a window as input and a sequence of windows as output. A way to get the result is to use the window() method that returns a dataset of window datasets:

```
data = tf.data.Dataset.range(7)

dummy_dataset = data.window(3, shift=1)

for window in dummy_dataset:
    for element in window:
        print(f"{element}", end=" ")
    print()
```

In the example, the dataset contains six windows, each shifted by one step compared to the previous one, and the last three windows are smaller because they've reached the end of the series. In general, we to get rid of these smaller windows by passing drop_remainder=True to the method.

However, window() returns a nested dataset (a list of lists), useful when we want to transform each window by calling its methods, but we cannot use it directly for training: our model expects tensors as not datasets. Therefore, we can use flat_map() method, which converts a nested dataset into a flat one:

```
dummy_dataset = data.window(3, shift=1, drop_remainder=True)
dummy_dataset = dummy_dataset.flat_map(lambda window: window.batch(3))
list(dummy_dataset)
```

```
[<tf.Tensor: shape=(3,), dtype=int64, numpy=array([0, 1, 2])>,
 <tf.Tensor: shape=(3,), dtype=int64, numpy=array([1, 2, 3])>,
 <tf.Tensor: shape=(3,), dtype=int64, numpy=array([2, 3, 4])>,
 <tf.Tensor: shape=(3,), dtype=int64, numpy=array([3, 4, 5])>,
 <tf.Tensor: shape=(3,), dtype=int64, numpy=array([4, 5, 6])>]
```

We now have a dataset containing consecutive windows represented as tensors. We can repeat the same procedure to get windows of consecutive windows:

```
dummy_dataset = dummy_dataset.window(4, shift=1, drop_remainder=True)
dummy_dataset = dummy_dataset.flat_map(lambda window: window.batch(4))
list(dummy_dataset)
```

```
[<tf.Tensor: shape=(4, 3), dtype=int64, numpy=
 array([[0, 1, 2],
        [1, 2, 3],
        [2, 3, 4],
        [3, 4, 5]])>,
 <tf.Tensor: shape=(4, 3), dtype=int64, numpy=
 array([[1, 2, 3],
        [2, 3, 4],
        [3, 4, 5],
        [4, 5, 6]])>]
```

Finally, we can use the map() method to split these windows of windows intocinputs and targets:

```
dummy_dataset = dummy_dataset.map(lambda S: (S[:, 0], S[:, 1:]))
list(dummy_dataset)
```

```
[(<tf.Tensor: shape=(4,), dtype=int64, numpy=array([0, 1, 2, 3])>,
  <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
  array([[1, 2],
         [2, 3],
         [3, 4],
         [4, 5]])>),
 (<tf.Tensor: shape=(4,), dtype=int64, numpy=array([1, 2, 3, 4])>,
  <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
  array([[2, 3],
         [3, 4],
         [4, 5],
         [5, 6]])>)]
```

Now the dataset contains sequences of length 4 as inputs, and the targets are sequences containing the next two steps, for each time step. For example, the first input sequence is [0, 1, 2, 3] has a corresponding target as [[1, 2], [2, 3], [3, 4], [4, 5]], which are the next two values for each time step.

Now we can create an utility function to prepare the datasets for sequence-to-sequence models:

```
def to_seq2seq_dataset(series, seq_length=56, ahead=14, target_col=1, batch_size=32):
    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(ahead + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda window: window.batch(ahead + 1))
    ds = ds.window(seq_length, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda window: window.batch(seq_length))
    ds = ds.map(lambda S: (S[:, 0], S[:, 1:, 1]))
    return ds.batch(batch_size)
```

Now we can create the datasets for our running example:

```
seq2seq_train = to_seq2seq_dataset(mulvar_train)
seq2seq_valid = to_seq2seq_dataset(mulvar_valid)
```

And lastly, we can build the sequence-to-sequence model

```
seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 2]),
    tf.keras.layers.Dense(14)
])
```

It is almost identical to previous model: the only difference is that we set return_sequences=True in the RNN layer. This way, it will output a sequence of vectors, instead of outputting a single vector at the last time step:

```
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
seq2seq_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

```
history = seq2seq_model.fit(seq2seq_train, validation_data=seq2seq_valid, epochs=500,
 ↪  callbacks=[early_stopping_cb])
```

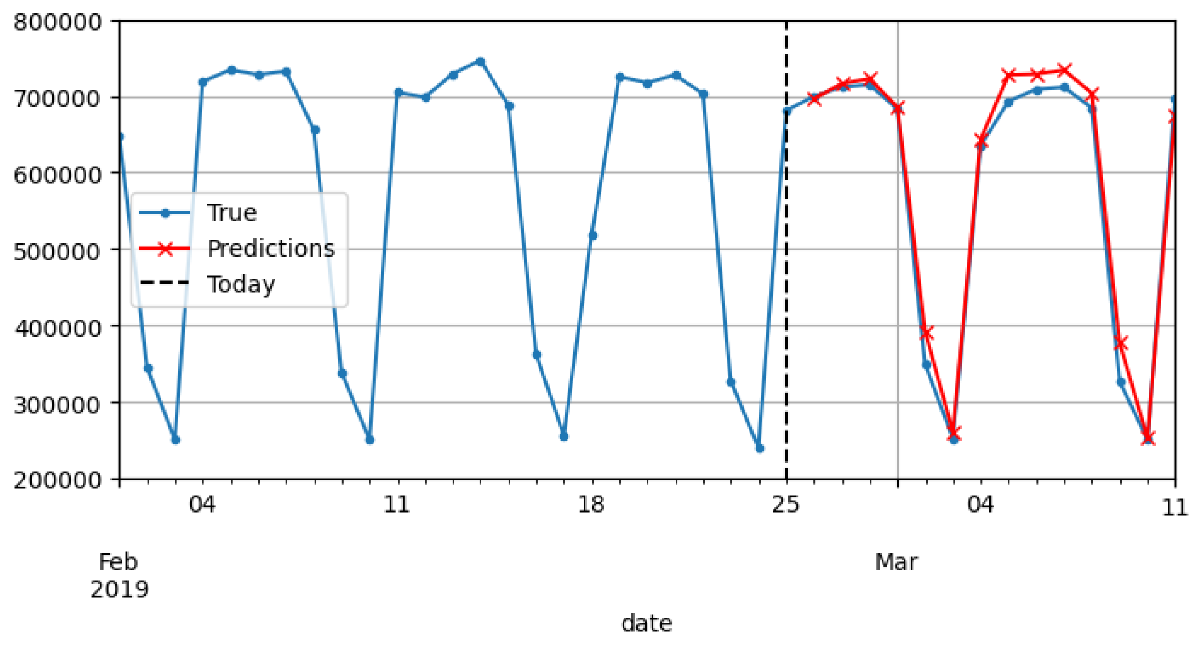
We can evaluate the model forecasts MAE for each steps:

```
Y_pred_valid = seq2seq_model.predict(seq2seq_valid)
for ahead in range(14):
    preds = pd.Series(Y_pred_valid[:-1, -1, ahead], index=mulvar_valid.index[56 +
 ↪  ahead : -14 + ahead])
    mae = (preds - mulvar_valid["rail"]).abs().mean() * 1e6
    print(f"MAE for +{ahead + 1}: {mae:,.0f}")
```

```
X = mulvar_valid.to_numpy()[np.newaxis, :seq_length]  # shape [1, 56, 5]
Y_pred = seq2seq_model.predict(X)[0, -1]  # shape [1, 14]
```

```
1/1 [==============================] - 0s 19ms/step
```

```
Y_pred = pd.Series(Y_pred, index=pd.date_range("2019-02-26", "2019-03-11"))
fig, ax = plt.subplots(figsize=(8, 3.5))
(mulvar_valid.rail * 1e6)["2019-02-01":"2019-03-11"].plot(label="True", marker=".",
 ↪  ax=ax)
(Y_pred * 1e6).plot(label="Predictions", grid=True, marker="x", color="r", ax=ax)
ax.vlines("2019-02-25", 0, 1e6, color="k", linestyle="--", label="Today")
ax.set_ylim([200_000, 800_000])
plt.legend(loc="center left")

plt.show()
```



We can combine different approaches to forecasting multiple steps ahead. For example, we can train a model that forecasts N days ahead, then take its output and append it to the inputs, then run the model again to get forecasts for the following N days, and possibly repeat the process. However, simple RNNs can be quite good at forecasting short time series, but they do not perform as well on long time series.

1.5 Handling Long Sequences

xxxxxTo train an RNN on long sequences, we must run it over many time steps, making the unrolled RNN a very deep network. Just like any deep neural network it may suffer from the unstable gradients problem, discussed in Chapter 11: it may take forever to train, or training may be unstable. Moreover, when an RNN processes a long sequence, it will gradually forget the first inputs in the sequence. Let's look at both these problems, starting with the unstable gradients problem.xxx

1.5.1 Unstable Gradients Problem

On long sequences, we must run the network over many time steps, **making the unrolled RNN a very deep network**. Like any deep neural network, it may suffer from the unstable gradients problem: it may take forever to train, or training may be unstable. Many of the tricks used in deep nets can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on.

However, **non-saturating activation functions (e.g., ReLU) may not help**. Suppose that gradient descent updates the weights in a way that increases the outputs slightly at the first time step. Because the same weights are used at every time step, the outputs at the second time step may also be slightly increased, and those at the third, and so on until the outputs explode, and a nonsaturating activation function does not prevent that. We can reduce this risk by using a smaller learning rate or we can use a saturating activation function (e.g. hyperbolic tangent). In the same way, gradients can explode. If we notice that training is unstable, we should monitor the size of the gradients and perhaps use gradient clipping.

Moreover, **batch normalization cannot be used as efficiently with RNNs**. We cannot use it between time steps, only between recurrent layers. To be more precise, it is technically possible to add a batch normalization layer to a memory cell, so that it will be applied at each time step. However, the same batch normalizatoin layer will be used at each time step, with the same parameters, regardless of the actual scale and offset of the inputs and hidden state. In practice, this does not yield good results: as demonstrated in Batch Normalized Recurrent Neural

Networks. It was slightly better than nothing when applied between recurrent layers, but not within recurrent layers.

Layer normalization often works better with RNNs: it is similar to batch normalization, but instead of normalizing across the batch dimension, it normalizes across the features dimension. One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing, and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set. In an RNN, it is typically used right after the linear combination of the inputs and the hidden states.

We need to define a custom memory cell, which is just like a regular layer, except it will apply layer normalization at each time step, and its call() method takes two arguments: the inputs at the current time step and the hidden states from the previous time step.

```
class LNSimpleRNNCell(tf.keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units

        # create a SimpleRNNCell with no activation function (because we
        # want to perform layer normalization after the linear operation but
        # before the activation function)
        self.simple_rnn_cell = tf.keras.layers.SimpleRNNCell(units, activation=None)
        self.layer_norm = tf.keras.layers.LayerNormalization()
        self.activation = tf.keras.activations.get(activation)

    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

Notice that the states argument is a list containing one or more tensors. In a simple RNN cell it contains a single tensor equal to the outputs of the previous time step, but other cells may have multiple state tensors. A cell must also have a state_size attribute and an output_size attribute. In a simple RNN, both are simply equal to the number of units.

```
custom_ln_model = tf.keras.Sequential([
    tf.keras.layers.RNN(LNSimpleRNNCell(32), return_sequences=True,
 ↪  input_shape=[None, 2]),
    tf.keras.layers.Dense(14)
])
```

```
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
custom_ln_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

Just training for some epochs to show that it works:

```
history = custom_ln_model.fit(seq2seq_train, validation_data=seq2seq_valid,
 ↪  epochs=500, callbacks=[early_stopping_cb])
```

```
Y_pred_valid = custom_ln_model.predict(seq2seq_valid)
for ahead in range(14):
    preds = pd.Series(Y_pred_valid[:-1, -1, ahead], index=mulvar_valid.index[56 +
 ↪  ahead : -14 + ahead])
    mae = (preds - mulvar_valid["rail"]).abs().mean() * 1e6
    print(f"MAE for +{ahead + 1}: {mae:,.0f}")
```

Similarly, we can create a custom cell to apply dropout between each time step. However, most recurrent layers in Keras have dropout hyperparameters. With these techniques, we can alleviate the unstable gradients problem and train an RNN much more efficiently.

1.6 Short-Term Memory Problem

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in ”the clouds are in the” we don’t need any further context, it’s pretty obvious the next word is going to be ”sky”. In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information. But there are also cases where we need **more context**. Consider trying to predict the last word in the text ”I grew up in France... I speak fluent”. Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It’s entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.

Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. To tackle this problem, various types of cells with **long-term memory** have been introduced. They proven so successful that the basic cells are not used anymore.

The **long short-term memory (LSTM)** cell looks exactly like a regular cell from the input-outpurt point of view (), however its state is split into two vectors: h_t (the short-term state) and c_t (the long-term state). The key idea is that **the network can learn what to store in the long-term state, what to throw away, and what to read from it**.

The long-term cell state c_t is like a conveyor belt: it runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged:

The LSTM cell has the ability to remove or add information to the long-term state, carefully regulated by structures called gates. They are composed of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. An LSTM cell has three of these gates, to protect and control the cell long-term state.

The first step is to decide what information we're going to throw away from the long-term cell state. This decision is made by a sigmoid layer called **the forget gate layer**:

$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

In the language model exampe, the cell log-term state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

The next step is to decide what new information we're going to store in the cell long-term state:

$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

$\hat{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

This has two parts. First, a sigmoid layer called the **input gate layer** decides which values we'll update. Next, a tanh layer creates a vector of new candidate values $\hat{c}_t$, that could be added to the state. In the next step, we'll combine these two to create an update to the state. In the example, we want to add the gender of the new subject to the cell state, to replace the old one we are forgetting.

It's now time to update the old long-term state C_{t-1} into the new long-term state C_t. The previous steps already decided what to do, we just need to actually do it:

$c_t = f_t * C_{t-1} + i_t * \hat{c}_t$

We multiply the old state by f_t (forgetting things we decided to forget earlier), then we add $\hat{c}_t$ (the new candidate values), scaled by how much we decided to update each state value. In the example, this is where we actually drop the information about the old subject gender and add the new information.

Finally, we need to decide what we're going to output using the **output gate**. This output will be based on our long-term state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the long-term state we're going to output. Then, we put the long-term state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to:

$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$

$h_t = o_t * \tanh(c_t)$

In the example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next. In short, an LSTM cell can learn to recognize an important input (that's the role of the input gate), store it in the long-term state, preserve it for as long as it is needed (that's the role of the forget gate), and extract it whenever it is needed.

In Keras, we can simply use the LSTM layer instead of the SimpleRNN layer:

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 2]),
    tf.keras.layers.Dense(14)
])
```

```
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
lstm_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

Just training for 5 epochs to show that it works:

```
history = lstm_model.fit(seq2seq_train, validation_data=seq2seq_valid, epochs=5,
 ↪  callbacks=[early_stopping_cb])
```

There are several variants of the LSTM cell. One particularly popular variant is the **Gated Recurrent Unit (GRU) cell**. It is a simplified version of the LSTM cell, and it seems to perform just as well:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\hat{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \hat{h}_t$$

Both state vectors are merged into a single vector h_t. A single gate controller z controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open (1) and the input gate is closed (0). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. There is no output gate: the full state vector is output at every time step. However, there is a new gate controller r that controls which part of the previous state will be shown to the main layer.

Keras provides a tf.keras.layers.GRU layer: using it is just a matter of replacing SimpleRNN or LSTM with GRU. It also provides a tf.keras.layers.GRUCell, in case you want to create a custom cell based on a GRU cell.

```
gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(32, return_sequences=True, input_shape=[None, 2]),
    tf.keras.layers.Dense(14)
])
```

```
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
gru_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

Just training for 5 epochs to show that it works:

```
history = gru_model.fit(seq2seq_train, validation_data=seq2seq_valid, epochs=5,
 ↪  callbacks=[early_stopping_cb])
```

These are only two of the most notable LSTM variants.

LSTM and GRU cells still have a **fairly limited short-term memory**, and they have a hard time learning long-term patterns in sequences of 100 time steps or more (such as audio samples, long time series, or long sentences). One way to solve this is to shorten the input sequences, for example, using **1D convolutional layers**.

Similarly two 2D convolutional, a 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel. Each kernel will learn to detect a single very short sequential pattern (no longer than the kernel size).

This means that we can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). By shortening the sequences the convolutional layer may help the LSTM/GRU layers detect longer patterns.

For example, the following model is the same as earlier, except it starts with a 1D convolutional layer that **downsamples the input sequence** by a factor of 2 (using a stride of 2). The kernel size is larger than the stride, so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the useful information, dropping only the unimportant details.

```
conv_rnn_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32, kernel_size=4, strides=2, activation="relu",
 ↪  input_shape=[None, 2]),
    tf.keras.layers.GRU(32, return_sequences=True),
    tf.keras.layers.Dense(14)
])
```

Notice that we must also crop off the first three time steps in the targets: indeed, the kernel's size is 4, so the first output of the convolutional layer will be based on the input time steps 0 to 3, and the first forecasts will be for time steps 4 to 17 (instead of time steps 1 to 14). Moreover, we must downsample the targets by a factor of 2, because of the stride:

```
longer_train = to_seq2seq_dataset(mulvar_train, seq_length=112)
longer_valid = to_seq2seq_dataset(mulvar_valid, seq_length=112)

downsampled_train = longer_train.map(lambda X, Y: (X, Y[:, 3::2]))
downsampled_valid = longer_valid.map(lambda X, Y: (X, Y[:, 3::2]))
```

Just training for 5 epochs to show that it works:

```
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
conv_rnn_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

```
history = conv_rnn_model.fit(downsampled_train, validation_data=downsampled_valid,
 ↪  epochs=5, callbacks=[early_stopping_cb])
```

If we train and evaluate this model, we will find that it outperforms the previous model (by a small margin). In fact, it is actually possible to use only 1D convolutional layers and drop the recurrent layers entirely! This is the **WaveNet architecuture**. It stacked 1D convolutional layers, doubling the dilation rate (how spread apart each neuron's inputs are) at every layer: the first convolutional layer gets a glimpse of just two time steps at a time, while the next one sees four time steps (its receptive field is four time steps long), the next one sees eight time steps, and so on.

This way, the lower layers learn short-term patterns, while the higher layers learn long-term patterns. Thanks to the doubling dilation rate, the network can process extremely large sequences very efficiently. The architecute also left-pads the input sequences with a number of zeros equal to the dilation rate before every layer, to preserve the same sequence length throughout the network.

Here is how to implement a simplified WaveNet to tackle the same sequences as earlier:

```
wavenet_model = tf.keras.Sequential()

# the model starts with an explicit input layer
wavenet_model.add(tf.keras.layers.InputLayer(input_shape=[None, 2]))

# then it continues with 1D convolutional layers using
# growing dilation rates: 1, 2, 4, 8, two times
# Thanks to the causal padding, every convolutional layer outputs a sequence of the
# ame length as its input sequence, so the targets we use during training can
# be the full sequences: no need to crop them or downsample them.

for rate in (1, 2, 4, 8) * 2:
    wavenet_model.add(tf.keras.layers.Conv1D(filters=32, kernel_size=2,
 ↪  padding="causal",
                                             activation="relu", dilation_rate=rate))

# finally, we add the output layer: a convolutional layer with 14 filters of size 1
 ↪  and
# without any activation function.
wavenet_model.add(tf.keras.layers.Conv1D(filters=14, kernel_size=1))
```

Just training for 5 epochs to show that it works:

```
opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
wavenet_model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
```

```
history = wavenet_model.fit(longer_train, validation_data=longer_valid, epochs=5,
 ↪  callbacks=[early_stopping_cb])
```

The models discussed offer similar performance for the simple ridership forecasting task, but they may vary significantly depending on the task and the amount of available data.

1.7 Exercise

1 - Download the Bach chorales dataset and unzip it. It is composed of 382 chorales composed by Johann Sebastian Bach. Each chorale is 100 to 640 time steps long, and each time step contains 4 integers, where each integer corresponds to a note's index on a piano (except for the value 0, which means that no note is played).

```
url =
 ↪  "https://www.dropbox.com/scl/fi/naevqylv41x21zdiol1rq/jsb_chorales.zip?rlkey=zhsumvz5mfmezu1vs5
 ↪  # dl=1 is important
u = urllib.request.urlopen(url)
data = u.read()
u.close()
with open("./data/jsb_chorales.zip", "wb") as f :
   f.write(data)
```

```
import zipfile

with zipfile.ZipFile("./data/jsb_chorales.zip","r") as zip_ref:
    zip_ref.extractall("./data")
```

```
from pathlib import Path
jsb_chorales_dir = Path("data/jsb_chorales")

train_files = sorted(jsb_chorales_dir.glob("train/chorale_*.csv"))
valid_files = sorted(jsb_chorales_dir.glob("valid/chorale_*.csv"))
test_files = sorted(jsb_chorales_dir.glob("test/chorale_*.csv"))
```

```
def load_chorales(filepaths):
    return [pd.read_csv(filepath).values.tolist() for filepath in filepaths]

train_chorales = load_chorales(train_files)
valid_chorales = load_chorales(valid_files)
test_chorales = load_chorales(test_files)
```

```python
train_chorales[0]
```

```python
notes = set()
for chorales in (train_chorales, valid_chorales, test_chorales):
    for chorale in chorales:
        for chord in chorale:
            notes |= set(chord)

n_notes = len(notes)
min_note = min(notes - {0})
max_note = max(notes)

assert min_note == 36
assert max_note == 81
```

Notice that values range from 36 (C1 = C on octave 1) to 81 (A5 = A on octave 5), plus 0 for silence. Let's write some functions to listen to chorales:

```python
def notes_to_frequencies(notes):
    # Frequency doubles when you go up one octave; there are 12 semi-tones
    # per octave; Note A on octave 4 is 440 Hz, and it is note number 69.
    return 2 ** ((np.array(notes) - 69) / 12) * 440
```

```python
def frequencies_to_samples(frequencies, tempo, sample_rate):
    note_duration = 60 / tempo # the tempo is measured in beats per minutes
    # To reduce click sound at every beat, we round the frequencies to try to
    # get the samples close to zero at the end of each note.
    frequencies = (note_duration * frequencies).round() / note_duration
    n_samples = int(note_duration * sample_rate)
    time = np.linspace(0, note_duration, n_samples)
    sine_waves = np.sin(2 * np.pi * frequencies.reshape(-1, 1) * time)
    # Removing all notes with frequencies ≤ 9 Hz (includes note 0 = silence)
    sine_waves *= (frequencies > 9.).reshape(-1, 1)
    return sine_waves.reshape(-1)
```

```python
def chords_to_samples(chords, tempo, sample_rate):
    freqs = notes_to_frequencies(chords)
    freqs = np.r_[freqs, freqs[-1:]] # make last note a bit longer
    merged = np.mean([frequencies_to_samples(melody, tempo, sample_rate)
                     for melody in freqs.T], axis=0)
    n_fade_out_samples = sample_rate * 60 // tempo # fade out last note
    fade_out = np.linspace(1., 0., n_fade_out_samples)**2
    merged[-n_fade_out_samples:] *= fade_out
    return merged
```

```
from IPython.display import Audio

def play_chords(chords, tempo=160, amplitude=0.1, sample_rate=44100, filepath=None):
    samples = amplitude * chords_to_samples(chords, tempo, sample_rate)
    if filepath:
        from scipy.io import wavfile
        samples = (2**15 * samples).astype(np.int16)
        wavfile.write(filepath, sample_rate, samples)
        return display(Audio(filepath))
    else:
        return display(Audio(samples, rate=sample_rate))
```

Now let's listen to a few chorales:

```
for index in range(3):
    play_chords(train_chorales[index])
```

```
<IPython.lib.display.Audio object>

<IPython.lib.display.Audio object>

<IPython.lib.display.Audio object>
```

2 - Train a model (recurrent, convolutional, or both) that can predict the next time step (four notes), given a sequence of time steps from a chorale.

We need to train a model that can predict the next chord given all the previous chords. If we try to predict the next chord in one shot (all 4 notes at once), we gets notes that don't go very well together (believe me, I tried). It's much better (and simpler) to predict one note at a time. So we will need to preprocess every chorale, turning each chord into an arpegio (i.e., a sequence of notes rather than notes played simultaneuously). So each chorale will be a long sequence of notes (rather than chords), and we can just train a model that can predict the next note given all the previous notes. We will also shift the values so that they range from 0 to 46, where 0 represents silence, and values 1 to 46 represent notes 36 (C1) to 81 (A5). And we will train the model on windows of 128 notes (i.e., 32 chords).

In this simple case, the dataset fits in memory, so we could preprocess the chorales in RAM using any Python code we like, but we will demonstrate here how to do all the preprocessing using tf.data, in order to be able to manage also dataset that dosn't fit in memory.

```python
def create_target(batch):
    X = batch[:, :-1]
    Y = batch[:, 1:] # predict next note in each arpegio, at each step
    return X, Y
```

```python
def preprocess(window):
    window = tf.where(window == 0, window, window - min_note + 1) # shift values
    return tf.reshape(window, [-1]) # convert to arpegio
```

```python
def bach_dataset(chorales, batch_size=32, shuffle_buffer_size=None,
                 window_size=32, window_shift=16, cache=True):
    def batch_window(window):
        return window.batch(window_size + 1)

    def to_windows(chorale):
        dataset = tf.data.Dataset.from_tensor_slices(chorale)
        dataset = dataset.window(window_size + 1, window_shift, drop_remainder=True)
        return dataset.flat_map(batch_window)

    chorales = tf.ragged.constant(chorales, ragged_rank=1)
    dataset = tf.data.Dataset.from_tensor_slices(chorales)
    dataset = dataset.flat_map(to_windows).map(preprocess)
    if cache:
        dataset = dataset.cache()
    if shuffle_buffer_size:
        dataset = dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(create_target)
    return dataset.prefetch(1)
```

Now let's create the training set, the validation set and the test set:

```
train_set = bach_dataset(train_chorales, shuffle_buffer_size=1000)
valid_set = bach_dataset(valid_chorales)
test_set = bach_dataset(test_chorales)
```

We will use a sequence-to-sequence approach, where we feed a window to the neural net, and it tries to predict that same window shifted one time step into the future. We could feed the note values directly to the model, as floats, but this would probably not give good results. The relationships between notes are not that simple: for example, if you replace a C3 with a C4, the melody will still sound fine, even though these notes are 12 semi-tones apart (i.e., one octave). Conversely, if you replace a C3 with a C#3, it's very likely that the chord will sound horrible, despite these notes being just next to each other. So we will use an Embedding layer to convert each note to a small vector representation (we will see embeddings in another notebook). We will use 5-dimensional embeddings, so the output of this first layer will have a shape of [batch_size, window_size, 5]. We will then feed this data to a small WaveNet-like neural network, composed of a stack of 4 Conv1D layers with doubling dilation rates. We will intersperse these layers with BatchNormalization layers for faster better convergence. Then one LSTM layer to try to capture long-term patterns. And finally a Dense layer to produce the final note probabilities. It will predict one probability for each chorale in the batch, for each time step, and for each possible note (including silence). So the output shape will be [batch_size, window_size, 47].

```
n_embedding_dims = 5

model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_notes, output_dim=n_embedding_dims,
                              input_shape=[None]),
    tf.keras.layers.Conv1D(32, kernel_size=2, padding="causal", activation="relu"),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv1D(48, kernel_size=2, padding="causal", activation="relu",
 ↪  dilation_rate=2),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv1D(64, kernel_size=2, padding="causal", activation="relu",
 ↪  dilation_rate=4),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv1D(96, kernel_size=2, padding="causal", activation="relu",
 ↪  dilation_rate=8),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.LSTM(256, return_sequences=True),
    tf.keras.layers.Dense(n_notes, activation="softmax")
])
```

```
model.summary()
```

Now we're ready to compile and train the model!

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=1e-3)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
 ↪  metrics=["accuracy"])
```

```
model.fit(train_set, epochs=20, validation_data=valid_set)
```

We have not done much hyperparameter search, so feel free to iterate on this model and try to optimize it. For example, we could try removing the LSTM layer and replacing it with Conv1D layers. You could also play with the number of layers, the learning rate, the optimizer, and so on. Once we're satisfied with the performance of the model on the validation set, we can save it and evaluate it one last time on the test set:

```
model.save("./data/bach_model", save_format="tf")
```

```
model.evaluate(test_set)
```

3 - Use the model to generate Bach-like music, one note at a time: you can do this by giving the model the start of a chorale and asking it to predict the next time step, then appending these time steps to the input sequence and asking the model for the next note, and so on.

There's no real need for a test set in this exercise, since we will perform the final evaluation by just listening to the music produced by the model. So if we want, you can add the test set to the train set, and train the model again, hopefully getting a slightly better model.

Now let's write a function that will generate a new chorale. We will give it a few seed chords, it will convert them to arpegios (the format expected by the model), and use the model to predict the next note, then the next, and so on. In the end, it will group the notes 4 by 4 to create chords again, and return the resulting chorale.

```
def generate_chorale(model, seed_chords, length):
    arpegio = preprocess(tf.constant(seed_chords, dtype=tf.int64))
    arpegio = tf.reshape(arpegio, [1, -1])
    for chord in range(length):
        for note in range(4):
            next_note = model.predict(arpegio, verbose=0).argmax(axis=-1)[:1, -1:]
            arpegio = tf.concat([arpegio, next_note], axis=1)
    arpegio = tf.where(arpegio == 0, arpegio, arpegio + min_note - 1)
    return tf.reshape(arpegio, shape=[-1, 4])
```

To test this function, we need some seed chords. Let's use the first 8 chords of one of the test chorales:

```
seed_chords = test_chorales[2][:8]
play_chords(seed_chords, amplitude=0.2)
```

```
<IPython.lib.display.Audio object>
```

Now we are ready to generate our first chorale. Let's ask the function to generate 56 more chords, for a total of 64 chords, i.e., 16 bars (assuming 4 chords per bar, i.e., a 4/4 signature):

```
new_chorale = generate_chorale(model, seed_chords, 56)
play_chords(new_chorale)
```

This approach has one major flaw: it is often too conservative. The model will not take any risk, it will always choose the note with the highest score, and since repeating the previous note generally sounds good enough, it's the least risky option, so the algorithm will tend to make notes last longer and longer. Pretty boring. Plus, if you run the model multiple times, it will always generate the same melody. So, instead of always picking the note with the highest score, we will pick the next note randomly, according to the predicted probabilities. For example, if the model predicts a C3 with 75% probability, and a G3 with a 25% probability, then we will pick one of these two notes randomly, with these probabilities. We will also add a "temperature" parameter that will control how "hot" we want the system to feel. A high temperature will bring the predicted probabilities closer together, reducing the probability of the likely notes and increasing the probability of the unlikely ones.

```
def generate_chorale_hot(model, seed_chords, length, temperature=1):
    arpegio = preprocess(tf.constant(seed_chords, dtype=tf.int64))
    arpegio = tf.reshape(arpegio, [1, -1])
    for chord in range(length):
        for note in range(4):
            next_note_probas = model.predict(arpegio)[0, -1:]
            rescaled_logits = tf.math.log(next_note_probas) / temperature
            next_note = tf.random.categorical(rescaled_logits, num_samples=1)
            arpegio = tf.concat([arpegio, next_note], axis=1)
    arpegio = tf.where(arpegio == 0, arpegio, arpegio + min_note - 1)
    return tf.reshape(arpegio, shape=[-1, 4])
```

Let's generate 3 chorales: one cold, one medium, and one hot and saves each chorale to a separate file. You can run these cells over an over again until you generate a masterpiece!

```
new_chorale_cold = generate_chorale_hot(model, seed_chords, 56, temperature=0.8)
play_chords(new_chorale_cold, filepath="./data/bach_cold.wav")
```

```
new_chorale_medium = generate_chorale_hot(model, seed_chords, 56, temperature=1.0)
play_chords(new_chorale_medium, filepath="./data/bach_medium.wav")
```

```
new_chorale_hot = generate_chorale_hot(model, seed_chords, 56, temperature=1.5)
play_chords(new_chorale_hot, filepath="./data/bach_hot.wav")
```