Recurrent Neural Networks

Recurrent Neural Networks

Predicting the future is something you do all the time. In this chapter, we will discuss recurrent neural networks (RNNs)—a class of nets that can analyze time series data, e.g.,

- the number of daily active users on your website,
- the hourly temperature in your city,
- your home's daily power consumption,
- the trajectories of nearby cars,
- ... ...

Recurrent Neural Networks

More generally, RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

Recurrent Neural Networks

Plan:

- the fundamental concepts underlying RNNs;
- train RNN using backpropagation through time;
- compare with ARMA models;
- two main difficulties that RNNs face:
  - **unstable gradients** which can be alleviated using various techniques, including recurrent dropout and recurrent layer normalization.
  - **limited short-term memory**, which can be extended using LSTM and GRU cells.
- handle sequential data with other types of neural networks.

Recurrent Neural Networks

Feed Forward Neural Network: the activations flow only in one direction, from the input layer to the outputlayer.

Recurrent Neural Network: similar to a feed forward neural network, except it also has connections pointing backward.

Recurrent Neural Networks

Simple example: one neuron

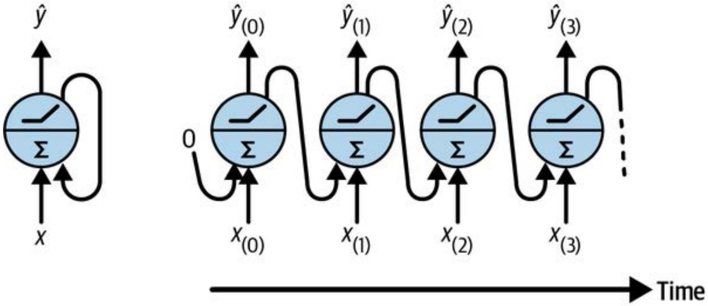


Figure: A recurrent neuron (left) unrolled through time (right)

Recurrent Neural Networks

create a layer of recurrent neurons

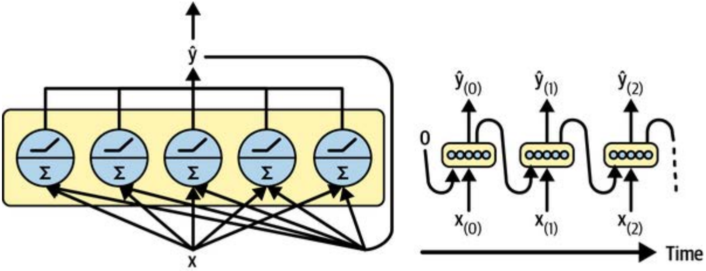


Figure: A layer of recurrent neurons (left) unrolled through time (right)

Recurrent Neural Networks

Each recurrent neuron has two sets of weights: one for the inputs $\boldsymbol{x}_(t)$ and the other for the outputs of the previous time step, $\hat{\boldsymbol{y}}_{(t-1)}$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices: $\boldsymbol{W}_{\boldsymbol{x}}$ and $\boldsymbol{W}_{\hat{\boldsymbol{y}}}$.

$$\hat{\boldsymbol{y}}_{(t)} = \phi(\boldsymbol{W}_{\boldsymbol{x}}\boldsymbol{x}_{(t)} + \boldsymbol{W}_{\hat{\boldsymbol{y}}}\hat{\boldsymbol{y}}_{(t-1)} + \boldsymbol{b})$$

Memory Cell

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of memory. A part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell).

Memory Cell

A cell's state at time step t, denoted $\boldsymbol{h}_{(t)}$ (the "$\boldsymbol{h}$" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\boldsymbol{h}_{(t)} = f(\boldsymbol{x}_{(t)}, \boldsymbol{h}_{(t-1)})$.

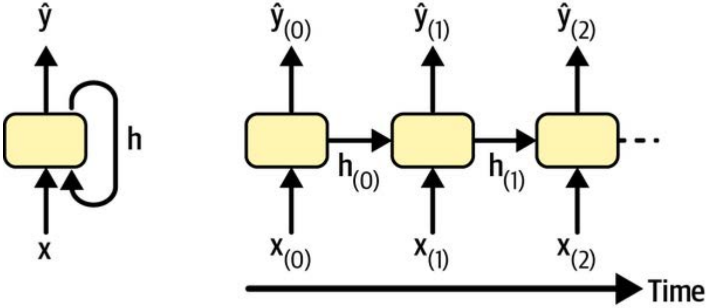
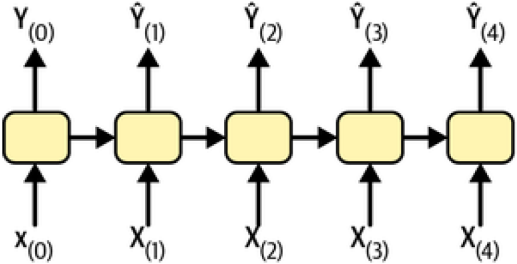


Figure: A cell's hidden state and its output may be different

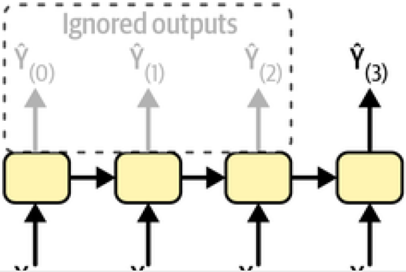
Input and Output Sequence

An RNN simultaneously take a sequence of inputs and produce a sequence of outputs. This type of sequence-to-sequence network is useful to forecast time series, such as your home's daily power consumption: you feed it the data over the last N days, and you train it to output the power consumption shifted by one day into the future.



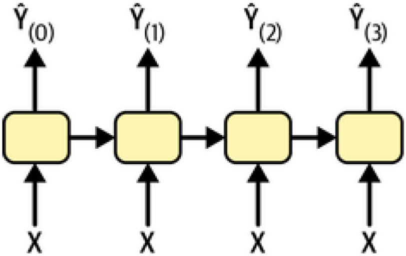
Input and Output Sequence

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one. This is a sequence-to-vector network. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., positive or negative).



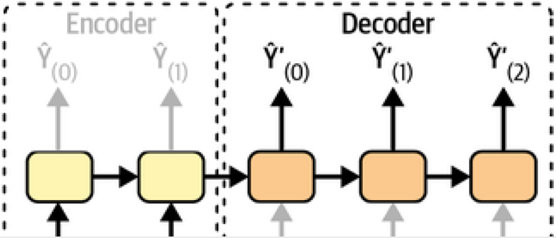
Input and Output Sequence

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence. This is a vector-to-sequence network. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.



Input and Output Sequence

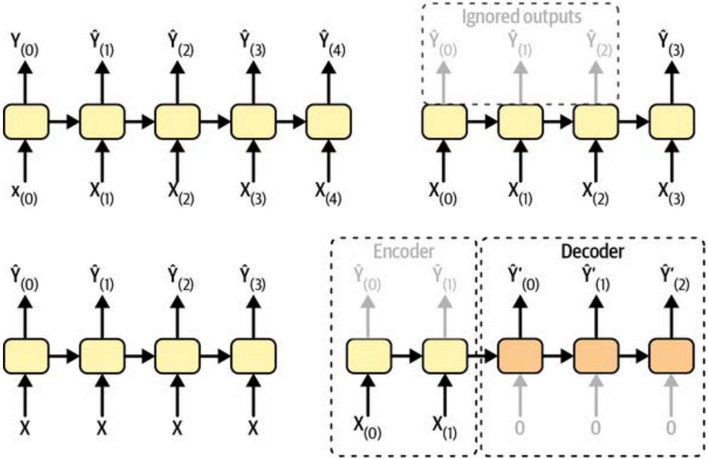
Lastly, you could have a sequence-to-vector network, called an **encoder**, followed by a vector-to-sequence network, called a **decoder**. For example, this could be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language.



Input and Output Sequence

This two-step model, called an encoder-decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN: the last words of a sentence can affect the first words of the translation, so you need to wait until you have seen the whole sentence before translating it.

Input and Output Sequence

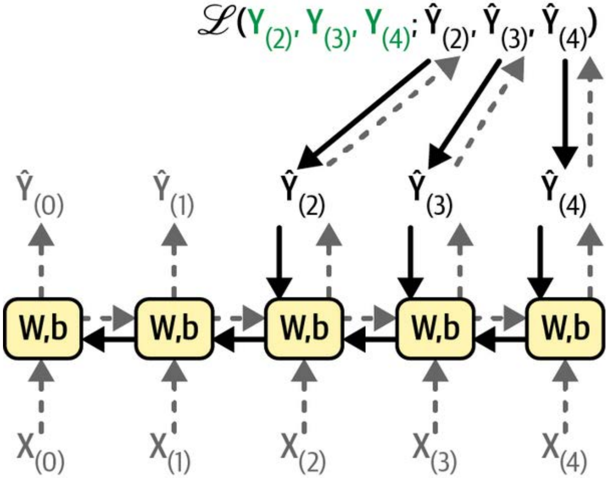


Training RNN

To train an RNN, the trick is to unroll it through time and then use regular backpropagation. This strategy is called backpropagation through time (BPTT).

- a first forward pass through the unrolled network;
- output sequence is evaluated using a loss function;
- compute the gradients of that loss through the unrolled network;
- a gradient descent step to update the parameters.

Training RNN



Training RNN

Note that this loss function may ignore some outputs. For example, in a sequence-to-vector RNN, all outputs are ignored except for the very last one. In this example, since the outputs $\hat{Y}_{(0)}$ and $\hat{Y}_{(1)}$ are not used to compute the loss, the gradients do not flow backward through them; they only flow through $\hat{Y}_{(2)}$, $\hat{Y}_{(3)}$, and $\hat{Y}_{(4)}$. Moreover, since the same parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ are used at each time step, their gradients will be tweaked multiple times during backprop.

Forecasting a Time Series

Your task is to build a model capable of forecasting the number of passengers that will ride on bus and rail the next day. You have access to daily ridership data since 2001.

```
In  [6]: tf.keras.utils.get_file(
             "ridership.tgz",
             "https://github.com/ageron/data/raw/main/ridership.tgz",
             cache_dir=".",
             extract=True
         )

Out[6]: '.\\datasets\\ridership.tgz'
```

Forecasting a Time Series

The raw data is

	A	B	C	D	E
1	service_date	day_type	bus	rail_boarding	total_rides
2	01/01/2001	U	297192	126455	423647
3	01/02/2001	W	780827	501952	1282779
4	01/03/2001	W	824923	536432	1361355
5	01/04/2001	W	870021	550011	1420032
6	01/05/2001	W	890426	557917	1448343

We start by loading and cleaning up the data:

```
In [7]: import pandas as pd
        from pathlib import Path

        path = Path("datasets/ridership/CTA_-_Ridership_-_Daily_Boarding_Totals.csv")
        df = pd.read_csv(path, parse_dates=["service_date"])
        df.columns = ["date", "day_type", "bus", "rail", "total"]  # shorter names
        df = df.sort_values("date").set_index("date")
        df = df.drop("total", axis=1)  # no need for total, it's just bus + rail
        df = df.drop_duplicates()  # remove duplicated months (2011-10 and 2014-07)
```

Forecasting a Time Series

Let's check what the first few rows look like:

```
In [8]: df.head()
```

Out[8]:

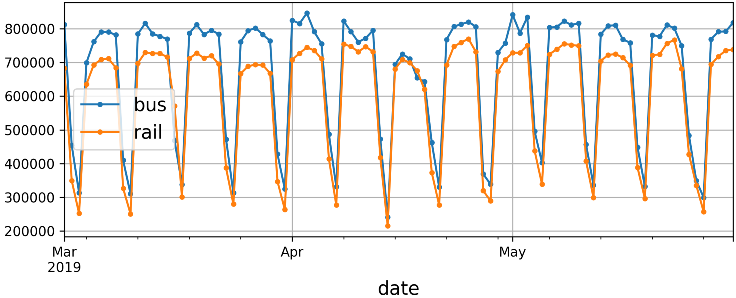
date	day_type	bus	rail
2001-01-01	U	297192	126455
2001-01-02	W	780827	501952
2001-01-03	W	824923	536432
2001-01-04	W	870021	550011
2001-01-05	W	890426	557917

On January 1st, 2001, 297,192 people boarded a bus, and 126,455 boarded a train. The day_type column contains **W** for Week-days, **A** for Saturdays, and **U** for Sundays or holidays.

Forecasting a Time Series

Let's plot the bus and rail ridership figures over a few months in 2019:

```
In [10]: import matplotlib.pyplot as plt

         df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
         save_fig("daily_ridership_plot")  # extra code – saves the figure for the book
         plt.show()
```

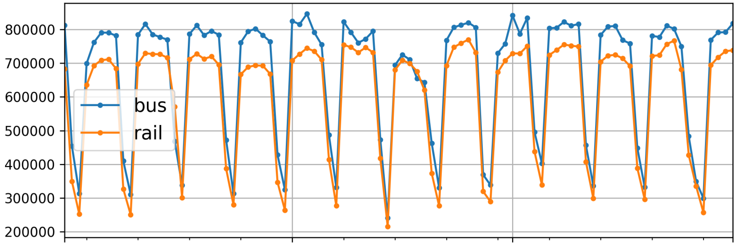


Forecasting a Time Series

- This is a time series: data with values at different time steps, usually at regular intervals.
- Since there are multiple values per time step, this is called a multivariate time series.
- If we only looked at the bus column, it would be a univariate time series, with a single value per time step.
- Predicting future values (i.e., forecasting) is the most typical task when dealing with time series.

Forecasting a Time Series

We can see that a similar pattern is clearly repeated every week. This is called a weekly seasonality. In fact, it's so strong in this case that forecasting tomorrow's ridership by just copying the values from a week earlier will yield reasonably good results. This is called naive forecasting : simply copying a past value to make our forecast. Naive fore-casting is often a great baseline, and it can even be tricky to beat in some cases.

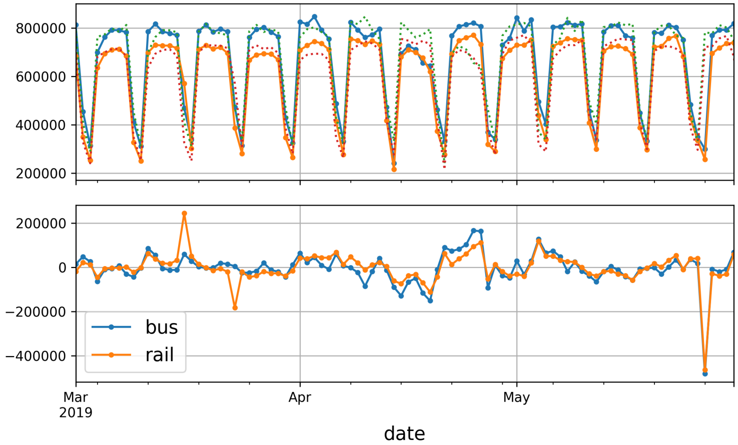


Forecasting a Time Series

To visualize these naive forecasts, let's overlay the two time series (for bus and rail) as well as the same time series lagged by one week using dotted lines. We'll also plot the difference between the two (i.e., the value at time t minus the value at time $t-7$); this is called differencing.

```
In [12]: diff_7 = df[["bus", "rail"]].diff(7)["2019-03":"2019-05"]

         fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 5))
         df.plot(ax=axs[0], legend=False, marker=".")  # original time series
         df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":")  # lagged
         diff_7.plot(ax=axs[1], grid=True, marker=".")  # 7-day difference time series
         axs[0].set_ylim([170_000, 900_000])  # extra code - beautifies the plot
         save_fig("differencing_plot")  # extra code - saves the figure
         plt.show()
```

Forecasting a Time Series



Forecasting a Time Series

Notice how closely the lagged time series track the actual time series. When a time series is correlated with a lagged version of itself, we say that the time series is autocorrelated . As you can see, most of the differences are fairly small, except at the end of May. Maybe there was a holiday at that time?

```
In  [13]: list(df.loc["2019-05-25":"2019-05-27"]["day_type"])
Out[13]: ['A', 'U', 'U']
```

Forecasting a Time Series

Let's just measure the mean absolute error (MAE) over the three-month period:

```
In  [14]: diff_7.abs().mean()

Out[14]: bus     43915.608696
         rail    42143.271739
         dtype: float64
```

It's hard to tell at a glance how good or bad this is:

```
In  [15]: targets = df[["bus", "rail"]]["2019-03":"2019-05"]
          (diff_7 / targets).abs().mean()

Out[15]: bus     0.082938
         rail    0.089948
         dtype: float64
```

Figure: mean absolute percentage error (MAPE)

Forecasting a Time Series

Note. The MAE, MAPE, and MSE are among the most common metrics you can use to evaluate your forecasts. As always, choosing the right metric depends on the task. For example, if your project suffers quadratically more from large errors than from small ones, then the MSE may be preferable, as it strongly penalizes large errors.

Forecasting a Time Series

Looking at the time series, there does not appear to be any significant monthly seasonality, but let's check whether there's any yearly seasonality.

```
In [16]: period = slice("2001", "2019")
         df_monthly = df.resample('M').mean()  # compute the mean for each month
         rolling_average_12_months = df_monthly[period].rolling(window=12).mean()

         fig, ax = plt.subplots(figsize=(8, 4))
         df_monthly[period].plot(ax=ax, marker=".")
         rolling_average_12_months.plot(ax=ax, grid=True, legend=False)
         save_fig("long_term_ridership_plot")  # extra code - saves the figure
         plt.show()
```

Forecasting a Time Series

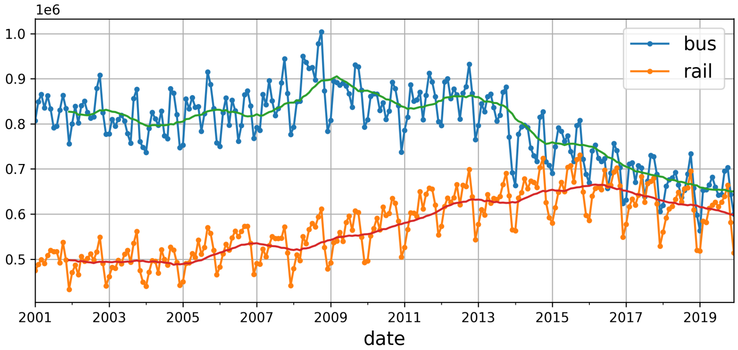
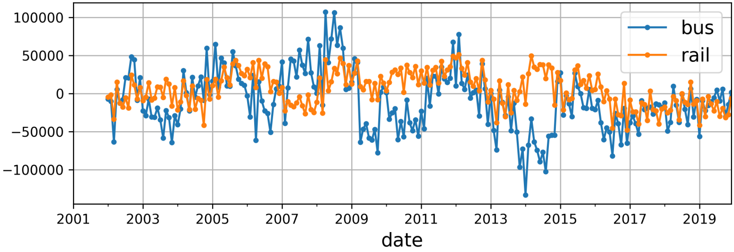


Figure: There's definitely some yearly seasonality as well, although it is noisier than the weekly seasonality and more visible for the rail series than the bus series.

Forecasting a Time Series

Let's check what we get if we plot the 12-month difference

```
In [17]: df_monthly.diff(12)[period].plot(grid=True, marker=".", figsize=(8, 3))
         save_fig("yearly_diff_plot")  # extra code – saves the figure
         plt.show()
```



Forecasting a Time Series

- Notice how differencing not only removed the yearly seasonality but also removed the long-term trends. For example, the linear downward trend present in the time series from 2016 to 2019 became a roughly constant negative value in the differenced time series.
- Differencing is a common technique used to remove trend and seasonality from a time series: it's easier to study a stationary time series, meaning one whose statistical properties remain constant over time, without any seasonality or trends.
- Once you're able to make accurate forecasts on the differenced time series, it's easy to turn them into forecasts for the actual time series by just adding back the past values that were previously subtracted.

Recall

Important concepts in time series analysis:

- seasonality;
- trend;
- differencing;
- moving average.

ARMA Model

We start with the autoregressive moving average (ARMA) model: it computes its forecasts using a simple weighted sum of lagged values and corrects these forecasts by adding a moving average.

$$\hat{\boldsymbol{y}}_{(t)} = \sum_{i=1}^{p} \alpha_i \boldsymbol{y}_{(t-i)} + \sum_{i=1}^{q} \theta_i \epsilon_{(t-i)}$$

with $\epsilon_{(t)} = \boldsymbol{y}_{(t)} - \hat{\boldsymbol{y}}_{(t)}$.

ARMA Model

- This model assumes that the time series is stationary. If it is not, then differencing may help.
- Using differencing over a single time step will produce an approximation of the derivative of the time series: indeed, it will give the slope of the series at each time step.
- This means that it will eliminate any linear trend, transforming it into a constant value. For example, if you apply one-step differencing to the series $[3, 5, 7, 9, 11]$, you get the differenced series $[2, 2, 2, 2]$.

ARMA Model

- If the original time series has a quadratic trend instead of a linear trend, then a single round of differencing will not be enough. For example, the series $[1, 4, 9, 16, 25, 36]$ becomes $[3, 5, 7, 9, 11]$ after one round of differencing.
- But if you run differencing for a second round, then you get $[2, 2, 2, 2]$. So, running two rounds of differencing will eliminate quadratic trends.
- More generally, running d consecutive rounds of differencing computes an approximation of the d-th order derivative of the time series, so it will eliminate polynomial trends up to degree d.
- This hyperparameter d is called the order of integration.

ARMA Model

Differencing is the central contribution of the autoregressive integrated moving average (ARIMA) model that runs d rounds of differencing to make the time series more stationary, then it applies a regular ARMA model. When making forecasts, it uses this ARMA model, then it adds back the terms that were subtracted by differencing.

ARMA Model

- One last member of the ARMA family is the seasonal ARIMA (SARIMA) model: it models the time series in the same way as ARIMA, but it additionally models a seasonal component for a given frequency (e.g., weekly), using the exact same ARIMA approach.
- It has a total of seven hyperparameters: the same p, d, and q hyperparameters as ARIMA, plus additional P, D, and Q hyperparameters to model the seasonal pattern, and lastly the period of the seasonal pattern, noted s.
- The hyperparameters P, D, and Q are just like p, d, and q, but they are used to model the time series at $t - s$, $t - 2s$, $t - 3s$, etc.

ARMA Model

We pretend today is the last day of May 2019, and we want to forecast the rail ridership for “tomorrow”, the 1st of June, 2019.

```
In [24]: from statsmodels.tsa.arima.model import ARIMA

         origin, today = "2019-01-01", "2019-05-31"
         rail_series = df.loc[origin:today]["rail"].asfreq("D")
         model = ARIMA(rail_series,
                       order=(1, 0, 0),
                       seasonal_order=(0, 1, 1, 7))
         model = model.fit()
         y_pred = model.forecast()  # returns 427,758.6
```

- asfreq("D") sets the time series' frequency to daily;
- order $= (1, 0, 0)$ means that $p = 1$, $d = 0$, $q = 0$;
- seasonal_order $= (0, 1, 1, 7)$ means that $P = 0$, $D = 1$, $Q = 1$, and $s = 7$.

ARMA Model

Pretty bad. Perhaps we were just unlucky that day.

```
In  [25]: y_pred[0]  # ARIMA forecast
Out[25]: 427758.62630005495

In  [26]: df["rail"].loc["2019-06-01"]  # target value
Out[26]: 379044

In  [27]: df["rail"].loc["2019-05-25"]  # naive forecast
Out[27]: 426932
```

ARMA Model

We run the same code in a loop to make forecasts for every day in March, April, and May, and compute the MAE over that period.

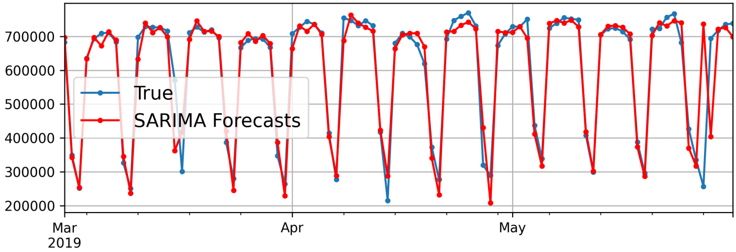
```
In [28]: origin, start_date, end_date = "2019-01-01", "2019-03-01", "2019-05-31"
         time_period = pd.date_range(start_date, end_date)
         rail_series = df.loc[origin:end_date]["rail"].asfreq("D")
         y_preds = []
         for today in time_period.shift(-1):
             model = ARIMA(rail_series[origin:today], # train on data up to "today"
                           order=(1, 0, 0),
                           seasonal_order=(0, 1, 1, 7))
             model = model.fit()  # note that we retrain the model every day!
             y_pred = model.forecast()[0]
             y_preds.append(y_pred)

         y_preds = pd.Series(y_preds, index=time_period)
         mae = (y_preds - rail_series[time_period]).abs().mean()  # returns 32,040.7

In [29]: mae

Out[29]: 32040.720106810877
```

The MAE is about 32,041, which is significantly lower than the MAE we got with naive forecasting (42,143).

ARMA Model

Displays the SARIMA forecasts

```
In [30]: fig, ax = plt.subplots(figsize=(8, 3))
         rail_series.loc[time_period].plot(label="True", ax=ax, marker=".", grid=True)
         ax.plot(y_preds, color="r", marker=".", label="SARIMA Forecasts")
         plt.legend()
         plt.show()
```



ARMA Model

At this point, you may be wondering how to pick good hyperparameters for the SARIMA model. There are several methods, but the simplest to understand and to get started with is the brute-force approach: just run a grid search. For each model you want to evaluate (i.e., each hyperparameter combination), you can run the preceding code example, changing only the hyperparameter values.

The model with the lowest MAE wins. Of course, you can replace the MAE with another metric if it better matches your business objective.

Preparing the Data

Now we have two baselines: naive forecasting and SARIMA. Let's try to use machine learning models to forecast this time series, starting with a basic linear model. Our goal will be to forecast tomorrow's ridership based on the ridership of the past 8 weeks of data (56 days).

But what will we use as training data? Well, that's the trick: we will use every 56-day window from the past as training data, and the target for each window will be the value immediately following it.

Preparing the Data

```
In [32]: import tensorflow as tf

         my_series = [0, 1, 2, 3, 4, 5]
         my_dataset = tf.keras.utils.timeseries_dataset_from_array(
             my_series,
             targets=my_series[3:],  # the targets are 3 steps into the future
             sequence_length=3,
             batch_size=2
         )
         list(my_dataset)

Out[32]: [(<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
           array([[0, 1, 2],
                  [1, 2, 3]])>,
           <tf.Tensor: shape=(2,), dtype=int32, numpy=array([3, 4])>),
          (<tf.Tensor: shape=(1, 3), dtype=int32, numpy=array([[2, 3, 4]])>,
           <tf.Tensor: shape=(1,), dtype=int32, numpy=array([5])>)]
```

The windows are [0, 1, 2], [1, 2, 3], and [2, 3, 4], and their respective targets are 3, 4, and 5.

Preparing the Data

We split the data into a training period, a validation period, and a test period. We focus on the rail ridership and scale it down by a factor of one million, to ensure the values are near the 0–1 range.

```
In [9]: rail_train = df["rail"]["2016-01":"2018-12"] / 1e6
        rail_valid = df["rail"]["2019-01":"2019-05"] / 1e6
        rail_test = df["rail"]["2019-06":] / 1e6
```

Now we are ready to build and train any model.

```
In [10]: seq_length = 56
         tf.random.set_seed(42)  # extra code – ensures reproducibility
         train_ds = tf.keras.utils.timeseries_dataset_from_array(
             rail_train.to_numpy(),
             targets=rail_train[seq_length:],
             sequence_length=seq_length,
             batch_size=32,
             shuffle=True,
             seed=42
         )
         valid_ds = tf.keras.utils.timeseries_dataset_from_array(
             rail_valid.to_numpy(),
             targets=rail_valid[seq_length:],
             sequence_length=seq_length,
             batch_size=32
         )
```

Linear Model

```
In [11]: tf.random.set_seed(42)
         model = tf.keras.Sequential([
             tf.keras.layers.Dense(1, input_shape=[seq_length])
         ])
         early_stopping_cb = tf.keras.callbacks.EarlyStopping(
             monitor="val_mae", patience=50, restore_best_weights=True)
         opt = tf.keras.optimizers.SGD(learning_rate=0.02, momentum=0.9)
         model.compile(loss=tf.keras.losses.Huber(), optimizer=opt, metrics=["mae"])
         history = model.fit(train_ds, validation_data=valid_ds, epochs=500,
                             callbacks=[early_stopping_cb])
```

This model reaches a validation MAE of about 37591. That's better than naive forecasting but worse than the SARIMA model.

```
In [12]: # extra code – evaluates the model
         valid_loss, valid_mae = model.evaluate(valid_ds)
         valid_mae * 1e6

         3/3 [==============================] - 0s 43ms/step - loss: 0.0021 - mae: 0.0376

Out[12]: 37591.252475976944
```

Simple RNN

```
In [13]: tf.random.set_seed(42)  # extra code – ensures reproducibility
         model = tf.keras.Sequential([
             tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
         ])
```

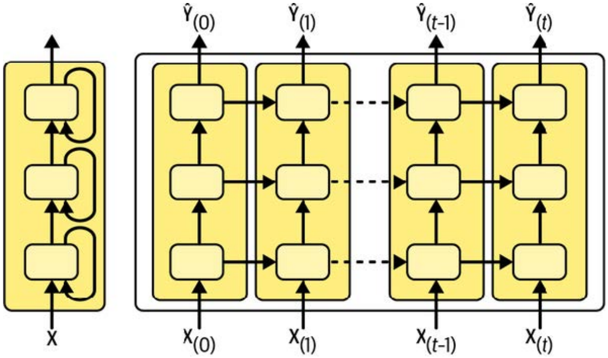
It's no good at all: its validation MAE is greater than 100,000! (102784.6559882164)

```
In [16]: tf.random.set_seed(42)  # extra code – ensures reproducibility
         univar_model = tf.keras.Sequential([
             tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
             tf.keras.layers.Dense(1)  # no activation function by default
         ])
```

That's the best model we've trained so far (30813.174322247505), and it even beats the SARIMA model

Deep RNN

Just stack recurrent layers!



Deep RNN

```
In [18]: tf.random.set_seed(42)  # extra code – ensures reproducibility
         deep_model = tf.keras.Sequential([
             tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
             tf.keras.layers.SimpleRNN(32, return_sequences=True),
             tf.keras.layers.SimpleRNN(32),
             tf.keras.layers.Dense(1)
         ])
```

It reaches an MAE of about 30422.

Multivariate Time Series

A great quality of neural networks is their flexibility: in particular, they can deal with multivariate time series with almost no change to their architecture.

In [20]:
```
df_mulvar = df[["bus", "rail"]] / 1e6  # use both bus & rail series as input
df_mulvar["next_day_type"] = df["day_type"].shift(-1)  # we know tomorrow's type
df_mulvar = pd.get_dummies(df_mulvar)  # one-hot encode the day type
```

In [21]:
```
mulvar_train = df_mulvar["2016-01":"2018-12"]
mulvar_valid = df_mulvar["2019-01":"2019-05"]
mulvar_test = df_mulvar["2019-06":]
```

In [22]:
```
tf.random.set_seed(42)  # extra code - ensures reproducibility

train_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),  # use all 5 columns as input
    targets=mulvar_train["rail"][seq_length:],  # forecast only the rail series
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
valid_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid["rail"][seq_length:],
    sequence_length=seq_length,
    batch_size=32
```

Deep RNN

```
In  [23]: tf.random.set_seed(42)  # extra code - ensures reproducibility
          mulvar_model = tf.keras.Sequential([
              tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
              tf.keras.layers.Dense(1)
          ])
```

At each time step, the model now receives five inputs instead of one. This model reaches a validation MAE of 23537.

Deep RNN

It's not too hard to make the RNN forecast for both the bus and rail ridership.

```
In [56]: # extra code – build and train a multitask RNN that forecasts both bus and rail

tf.random.set_seed(42)

seq_length = 56
train_multask_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=mulvar_train[["bus", "rail"]][seq_length:],  # 2 targets per day
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
valid_multask_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid[["bus", "rail"]][seq_length:],
    sequence_length=seq_length,
    batch_size=32
)

tf.random.set_seed(42)
multask_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(2)
])
```

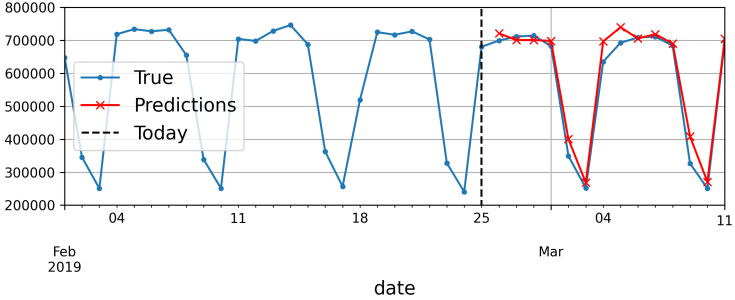
It reaches a validation MAE of 25,330 for rail and 26,369for bus, which is pretty good.

Forecasting Several Time Steps

The first option is to take the univar_model RNN we trained earlier for the rail time series, make it predict the next value, and add that value to the inputs, acting as if the predicted value had actually occurred; we would then use the model again to predict the following value, and so on.

```
In [36]: import numpy as np

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

Forecasting Several Time Steps



Forecasting Several Time Steps

The second option is to train an RNN to predict the next 14 values in one shot. We can still use a sequence-to-vector model, but it will output 14 values instead of 1.

In [38]:
```
tf.random.set_seed(42)  # extra code – ensures reproducibility

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,
    seed=42
).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)
```

Forecasting Several Time Steps

```
In [39]: tf.random.set_seed(42)

         ahead_model = tf.keras.Sequential([
             tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
             tf.keras.layers.Dense(14)
         ])
```

It reaches a prediction MAE of 34203.
This approach works quite well. Its forecasts for the next day are obviously better than its forecasts for 14 days into the future, but it doesn't accumulate errors as the previous approach did.

Sequence-to-Sequence Model

Instead of training the model to forecast the next 14 values only at the very last time step, we can train it to forecast the next 14 values at each and every time step. In other words, we can turn this sequence-to-vector RNN into a sequence-to-sequence RNN.

Sequence-to-Sequence Model

At time step 0 the model will output a vector containing the forecasts for time steps 1 to 14, then at time step 1 the model will forecast time steps 2 to 15, 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 any more, but a sequence of the same length as the inputs, containing a 14-dimensional vector at each step.

Sequence-to-Sequence Model

Preparing the datasets is not trivial. For example, let's turn the series of numbers 0 to 6 into a dataset containing sequences of 4 consecutive windows, each of length 3:

```
In [43]: my_series = tf.data.Dataset.range(7)
         dataset = to_windows(to_windows(my_series, 3), 4)
         list(dataset)

Out[43]: [<tf.Tensor: shape=(4, 3), dtype=int64, numpy=
          array([[0, 1, 2],
                 [1, 2, 3],
                 [2, 3, 4],
                 [3, 4, 5]], dtype=int64)>,
          <tf.Tensor: shape=(4, 3), dtype=int64, numpy=
          array([[1, 2, 3],
                 [2, 3, 4],
                 [3, 4, 5],
                 [4, 5, 6]], dtype=int64)>]
```

Sequence-to-Sequence Model

Now we can use the map() method to split these windows of windows into inputs and targets:

```
In  [44]: dataset = dataset.map(lambda S: (S[:, 0], S[:, 1:]))
          list(dataset)

Out[44]: [(<tf.Tensor: shape=(4,), dtype=int64, numpy=array([0, 1, 2, 3], dtype=int64)>,
           <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
           array([[1, 2],
                  [2, 3],
                  [3, 4],
                  [4, 5]], dtype=int64)>),
          (<tf.Tensor: shape=(4,), dtype=int64, numpy=array([1, 2, 3, 4], dtype=int64)>,
           <tf.Tensor: shape=(4, 2), dtype=int64, numpy=
           array([[2, 3],
                  [3, 4],
                  [4, 5],
                  [5, 6]], dtype=int64)>)]
```

Sequence-to-Sequence Model

```
In [45]: def to_seq2seq_dataset(series, seq_length=56, ahead=14, target_col=1,
                                batch_size=32, shuffle=False, seed=None):
             ds = to_windows(tf.data.Dataset.from_tensor_slices(series), ahead + 1)
             ds = to_windows(ds, seq_length).map(lambda S: (S[:, 0], S[:, 1:, 1]))
             if shuffle:
                 ds = ds.shuffle(8 * batch_size, seed=seed)
             return ds.batch(batch_size)
```

```
In [46]: seq2seq_train = to_seq2seq_dataset(mulvar_train, shuffle=True, seed=42)
         seq2seq_valid = to_seq2seq_dataset(mulvar_valid)
```

```
In [47]: tf.random.set_seed(42)  # extra code - ensures reproducibility
         seq2seq_model = tf.keras.Sequential([
             tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 5]),
             tf.keras.layers.Dense(14)
             # equivalent: tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(14))
             # also equivalent: tf.keras.layers.Conv1D(14, kernel_size=1)
         ])
```

Sequence-to-Sequence Model

```
In [50]: 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}")
```

```
3/3 [==============================] - 0s 5ms/step
MAE for +1: 24,501
MAE for +2: 27,870
MAE for +3: 29,396
MAE for +4: 32,048
MAE for +5: 32,972
MAE for +6: 34,170
MAE for +7: 35,778
MAE for +8: 36,076
MAE for +9: 33,626
MAE for +10: 32,891
MAE for +11: 38,944
MAE for +12: 38,702
MAE for +13: 37,223
MAE for +14: 35,361
```