Chapter 13. Processing Sequences Using RNNs and CNNs

Predicting the future is something you do all the time, whether you are finishing a friend's sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs)—a class of nets that can predict the future (well, up to a point). RNNs can analyze time series data, such as 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, and more. Once an RNN learns past patterns in the data, it is able to use its knowledge to forecast the future, assuming, of course, that past patterns still hold in the future.

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, which makes them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter, we will first go through the fundamental concepts underlying RNNs and how to train them using backpropagation through time. Then, we will use them to forecast a time series. Along the way, we will look at the popular autoregressive moving average (ARMA) family of models, often used to forecast time series, and use them as baselines to compare with our RNNs. After that, we'll explore the two main difficulties that RNNs face:

- Unstable gradients (discussed in Chapter 11), which can be alleviated using various techniques, including *recurrent dropout* and *recurrent layer normalization*.
- A (very) limited short-term memory, which can be extended using long short-term memory (LSTM) and gated recurrent unit (GRU) cells.

RNNs are not the only types of neural networks capable of handling sequential data. For small sequences, a regular dense network can do the trick, and for very long sequences, such as audio samples or text, convolutional neural networks can actually work quite well too. We will discuss both of these possibilities, and we will finish this chapter by implementing a WaveNet—a CNN architecture capable of handling sequences of tens of thousands of time steps. But we can do even better! In Chapter 14, we will apply RNNs to natural language processing (NLP), and we will see how to boost them using attention mechanisms. Attention is at the core of transformers, which we will discover in Chapter 15: they are now the state of the art for sequence processing, NLP, and even computer vision. But before we get there, let's start with the simplest RNNs!

Recurrent Neurons and Layers

Up to now we have focused on feedforward neural networks, where the activations flow only

in one direction, from the input layer to the output layer. A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward.

Let's look at the simplest possible RNN, composed of one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in Figure 13-1 (left). At each *time step t* (also called a *frame*), this *recurrent neuron* receives the inputs $\mathbf{x}_{(t)}$ as well as its own output from the previous time step, $\hat{y}_{(t–1)}$. Since there is no previous output at the first time step, it is generally set to zero. We can represent this tiny network against the time axis, as shown in Figure 13-1 (right). This is called *unrolling the network through time* (it's the same recurrent neuron represented once per time step).

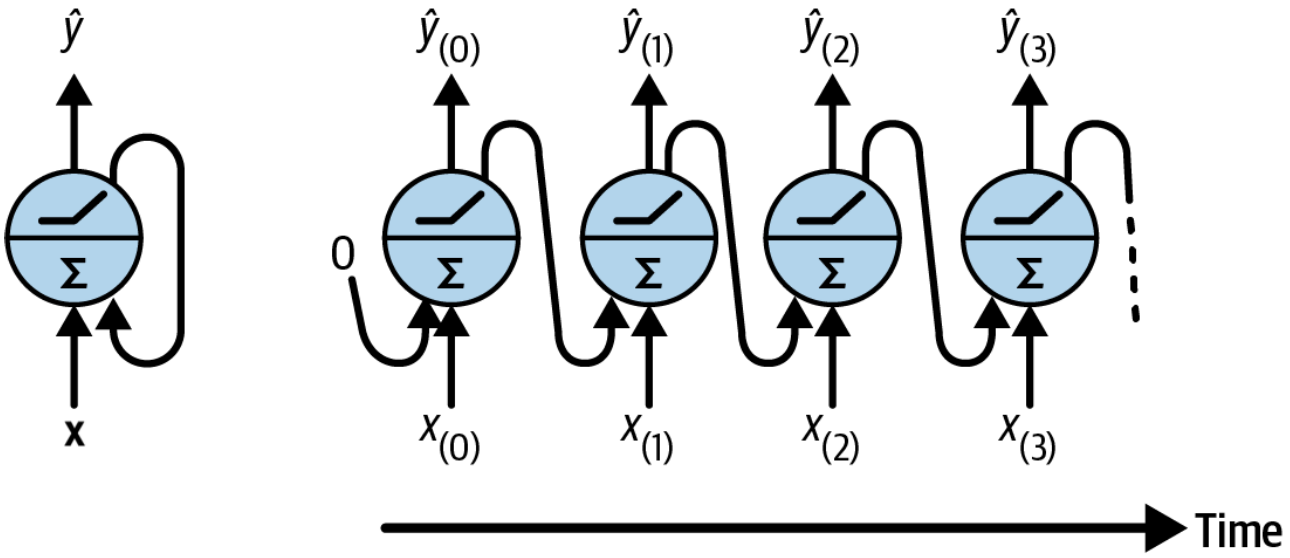


Figure 13-1. A recurrent neuron (left) unrolled through time (right)

You can easily create a layer of recurrent neurons. At each time step *t*, every neuron receives both the input vector $\mathbf{x}_{(t)}$ and the output vector from the previous time step $\hat{\mathbf{y}}_{(t–1)}$, as shown in Figure 13-2. Note that both the inputs and outputs are now vectors (when there was just a single neuron, the output was a scalar).

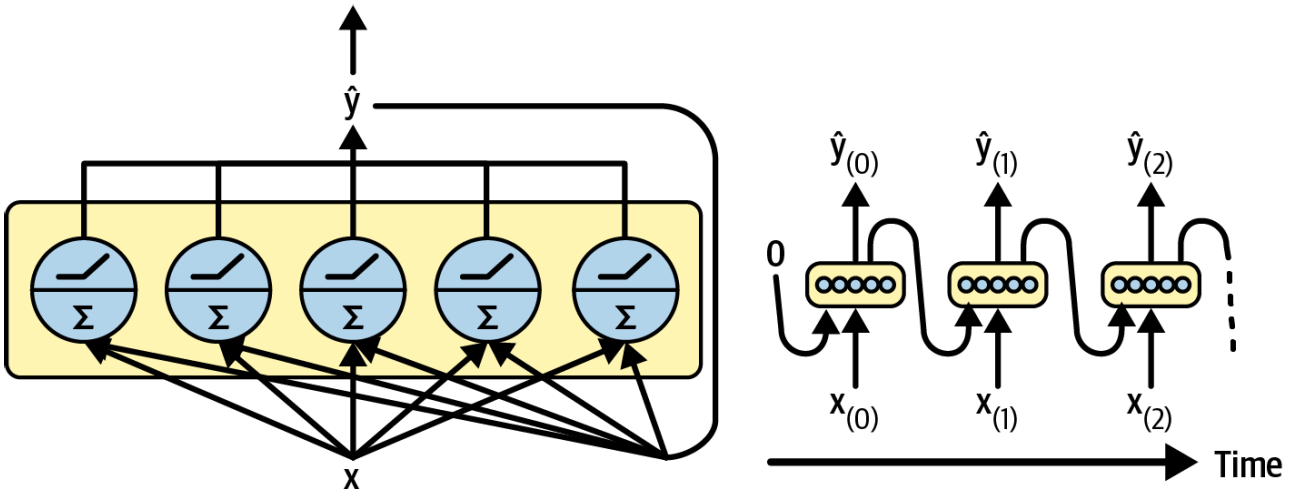


Figure 13-2. A layer of recurrent neurons (left) unrolled through time (right)

Each recurrent neuron has two sets of weights: one for the inputs $\mathbf{x}_{(t)}$ and the other for the outputs of the previous time step, $\hat{\mathbf{y}}_{(t–1)}$. Let's call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_{\hat{y}}$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight

the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices: $\mathbf{W}_x$ and $\mathbf{W}_{\hat{y}}$.

The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in Equation 13-1, where $\mathbf{b}$ is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU[1]).

Equation 13-1. Output of a recurrent layer for a single instance

Just as with feedforward neural networks, we can compute a recurrent layer's output in one shot for an entire mini-batch by placing all the inputs at time step t into an input matrix $\mathbf{X}_{(t)}$ (see Equation 13-2).

Equation 13-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\hat{\mathbf{y}}_{(t)} = \phi\left(\mathbf{W}_x{}^{\mathrm{T}}\mathbf{x}_{(t)} + \mathbf{W}_{\hat{y}}{}^{\mathrm{T}}\hat{\mathbf{y}}_{(t-1)} + \mathbf{b}\right)$$

In this equation:

- $\hat{\mathbf{Y}}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch, and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_{\hat{y}}$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_{\hat{y}}$ are often concatenated vertically into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of Equation 13-2).
- The notation $[\mathbf{X}_{(t)}\ \hat{\mathbf{Y}}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\hat{\mathbf{Y}}_{(t-1)}$.

Notice that $\hat{\mathbf{Y}}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\hat{\mathbf{Y}}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\hat{\mathbf{Y}}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\hat{\mathbf{Y}}_{(t-3)}$, and so on. This makes $\hat{\mathbf{Y}}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

NOTE

The idea of introducing backward connections (i.e., loops) in artificial neural networks dates back to the very origins of ANNs, but the first modern RNN architecture was proposed by Michael I. Jordan in 1986[2] At each time step, his RNN would look at the inputs for that time step, plus its own outputs from the previous time step. This is called *output feedback*.

Memory Cells

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*). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long, but this varies depending on the task). Later in this chapter, we will look at some more complex and powerful types of cells capable of learning longer patterns (roughly 10 times longer, but again, this depends on the task).

A cell's state at time step *t*, denoted $\mathbf{h}_{(t)}$ (the "h" stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$. Its output at time step *t*, denoted $\hat{\mathbf{y}}_{(t)}$, is also a function of the previous state and the current inputs, and typically it's just a function of the current state. In the case of the basic cells we have discussed so far, the output is just equal to the state, but in more complex cells this is not always the case, as shown in Figure 13-3.

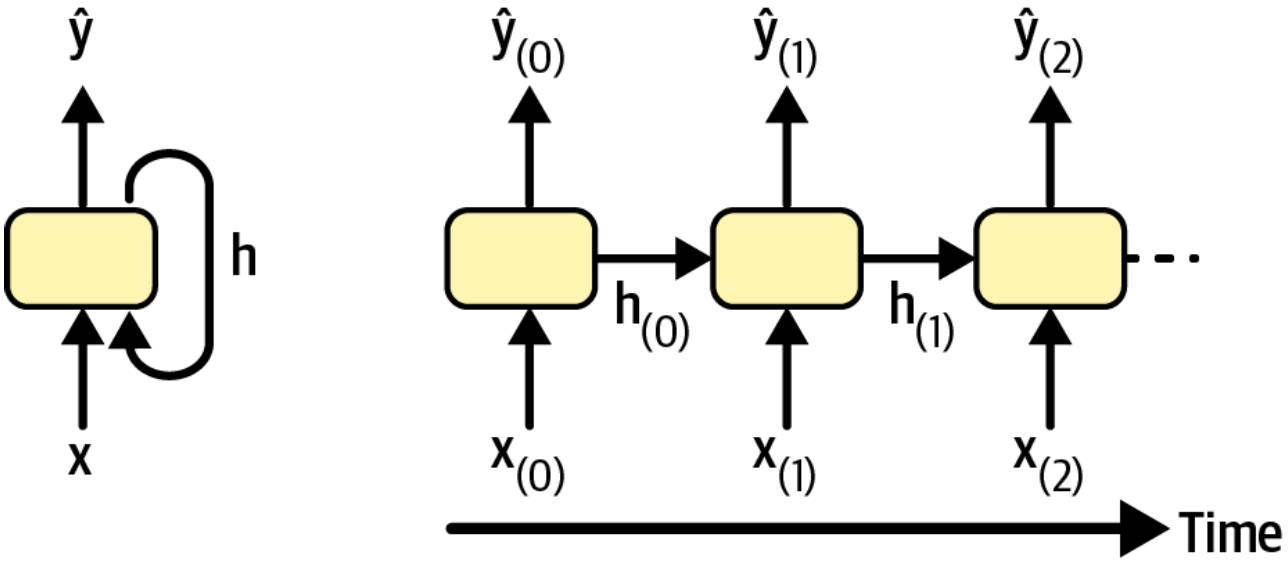


Figure 13-3. A cell's hidden state and its output may be different

NOTE

The first modern RNN that fed back the hidden state rather than the outputs was proposed by Jeffrey Elman in 1990.[3] This is called *state feedback*, and it's the most common approach today.

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see the top-left network in Figure 13-4). 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 (i.e., from *N* – 1 days ago to tomorrow).

ture (i.e., from $N - 1$ days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs and ignore all outputs except for the last one (see the top-right network in Figure 13-4). 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., from 0 [hate] to 1 [love]).

Conversely, you could feed the network the same input vector over and over again at each time step and let it output a sequence (see the bottom-left network of Figure 13-4). 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.

Lastly, you could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder* (see the bottom-right network of Figure 13-4). 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. This two-step model, called an *encoder–decoder*,[4] works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented at the top left): 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. We will go through the implementation of an encoder–decoder in Chapter 14 (as you will see, it is a bit more complex than what Figure 13-4 suggests).

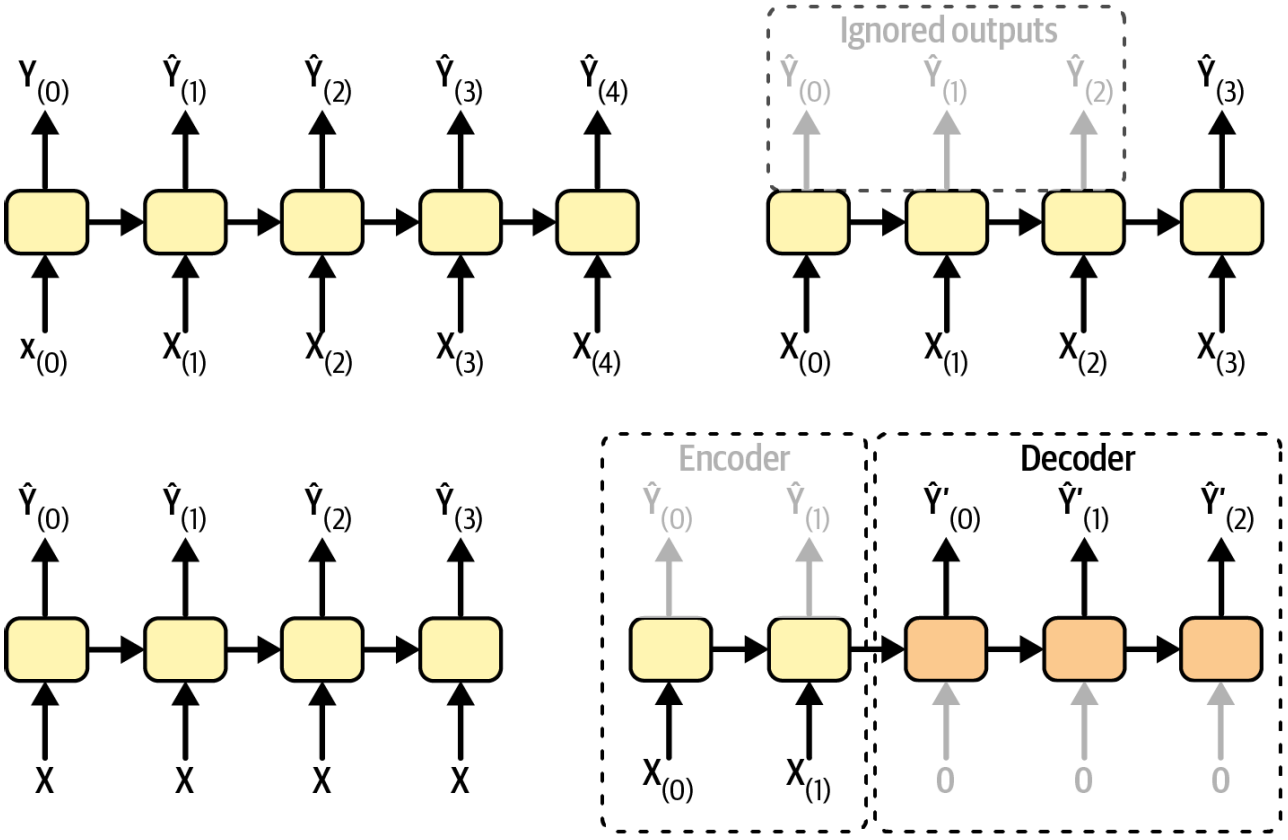


Figure 13-4. Sequence-to-sequence (top left), sequence-to-vector (top right), vector-to-sequence (bottom left), and encoder–decoder (bottom right) networks

This versatility sounds promising, but how do you train a recurrent neural network?

Training RNNs

To train an RNN, the trick is to unroll it through time (like we just did) and then use regular backpropagation (see Figure 13-5). This strategy is called *backpropagation through time* (BPTT).

Just like in regular backpropagation, there is a first forward pass through the unrolled network (represented by the dashed arrows). Then the output sequence is evaluated using a loss function $\mathscr{L}(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, \ldots, \mathbf{Y}_{(T)}; \hat{\mathbf{Y}}_{(0)}, \hat{\mathbf{Y}}_{(1)}, \ldots, \hat{\mathbf{Y}}_{(T)})$ (where $\mathbf{Y}_{(i)}$ is the i^{th} target, $\hat{\mathbf{Y}}_{(i)}$ is the i^{th} prediction, and T is the max time step). 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 Figure 13-5, the loss function is computed based on the last three outputs only. The gradients of that loss function are then propagated backward through the unrolled network (represented by the solid arrows). In this example, since the outputs $\hat{\mathbf{Y}}_{(0)}$ and $\hat{\mathbf{Y}}_{(1)}$ are not used to compute the loss, the gradients do not flow backward through them; they only flow through $\hat{\mathbf{Y}}_{(2)}$, $\hat{\mathbf{Y}}_{(3)}$, and $\hat{\mathbf{Y}}_{(4)}$. Moreover, since the same parameters $\mathbf{W}$ and $\mathbf{b}$ are used at each time step, their gradients will be tweaked multiple times during backprop. Once the backward phase is complete and all the gradients have been computed, BPTT can perform a gradient descent step to update the parameters (this is no different from regular backprop).

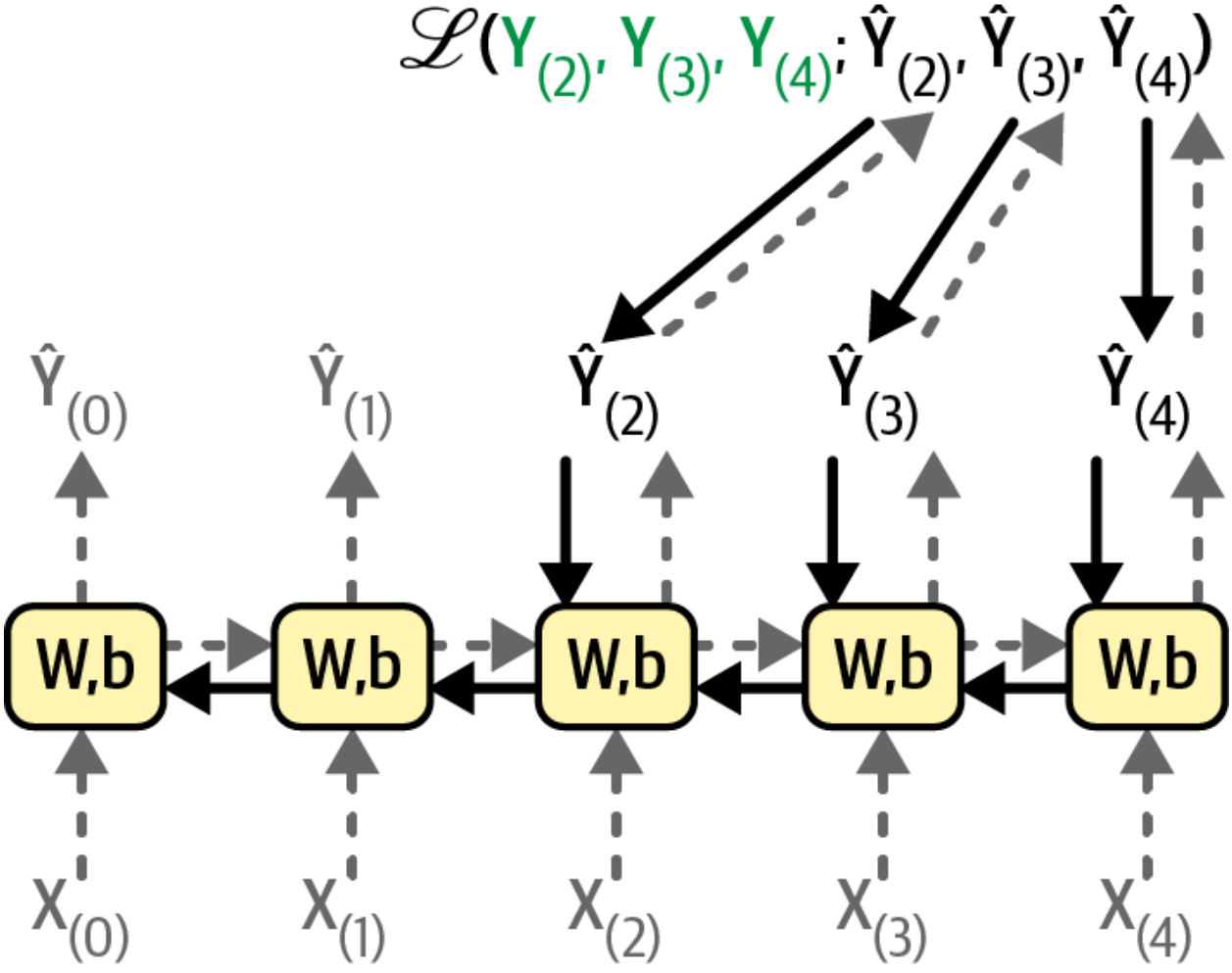


Figure 13-5. Backpropagation through time

Fortunately, PyTorch takes care of all of this complexity for you, as you will see. But before we get there, let's load a time series and start analyzing it using classical tools to better understand what we're dealing with, and to get some baseline metrics.

Forecasting a Time Series

All right! Let's pretend you've just been hired as a data scientist by Chicago's Transit Authority. Your first 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. Let's walk through how you would handle this. We'll start by loading and cleaning up the data:[5]

```
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)
```

We load the CSV file, set short column names, sort the rows by date, remove the redundant `total` column, and drop duplicate rows. Now let's check what the first few rows look like:

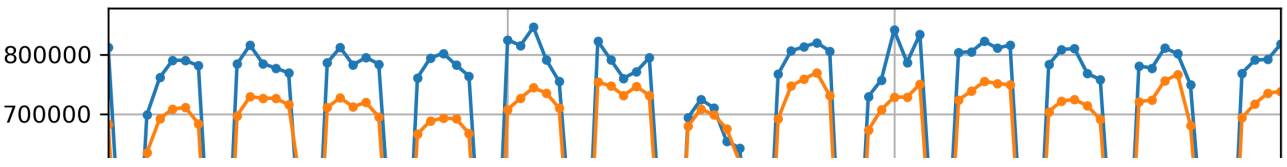
```
>>> df.head()
            day_type     bus    rail
date
2001-01-01        U  297192  126455
2001-01-02        W  780827  501952
2001-01-03        W  824923  536432
2001-01-04        W  870021  550011
2001-01-05        W  890426  557917
```

On January 1st, 2001, 297,192 people boarded a bus in Chicago, and 126,455 boarded a train. The `day_type` column contains `W` for **W**eekdays, `A` for S**a**turdays, and `U` for S**u**ndays or holidays.

Now let's plot the bus and rail ridership figures over a few months in 2019, to see what it looks like (see Figure 13-6):

```
import matplotlib.pyplot as plt

df["2019-03":"2019-05"].plot(grid=True, marker=".", figsize=(8, 3.5))
plt.show()
```



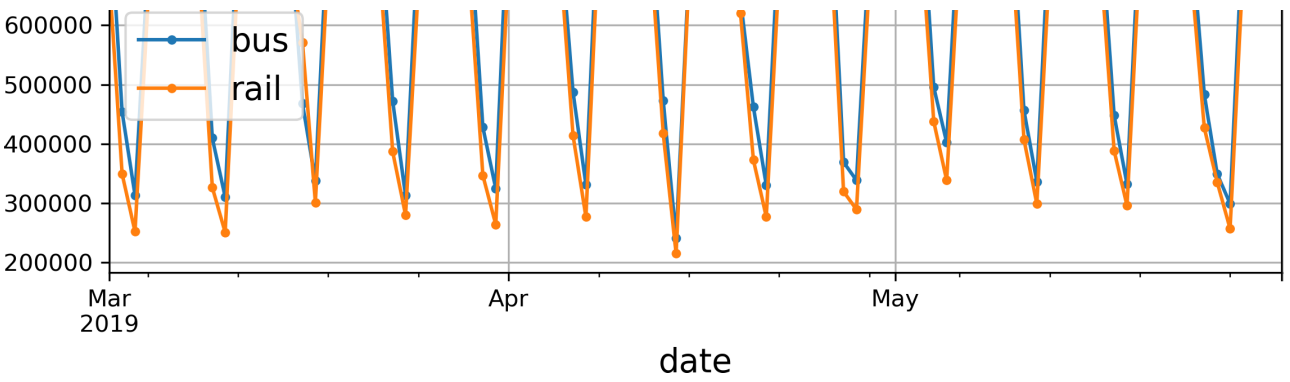


Figure 13-6. Daily ridership in Chicago

Note that Pandas includes both the start and end month in the range, so this plots the data from the 1st of March all the way up to the 31st of May. This is a *time series*: data with values at different time steps, usually at regular intervals. More specifically, 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, and this is what we will focus on in this chapter. Other tasks include imputation (filling in missing past values), classification, anomaly detection, and more.

Looking at Figure 13-6, 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 forecasting is often a great baseline, and it can even be tricky to beat in some cases.

NOTE

In general, naive forecasting means copying the latest known value (e.g., forecasting that tomorrow will be the same as today). However, in our case, copying the value from the previous week works better, due to the strong weekly seasonality.

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 (i.e., shifted toward the right) 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* (see Figure 13-7):

```
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
plt.show()
```

Not too bad! 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? Let's check the `day_type` column:

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

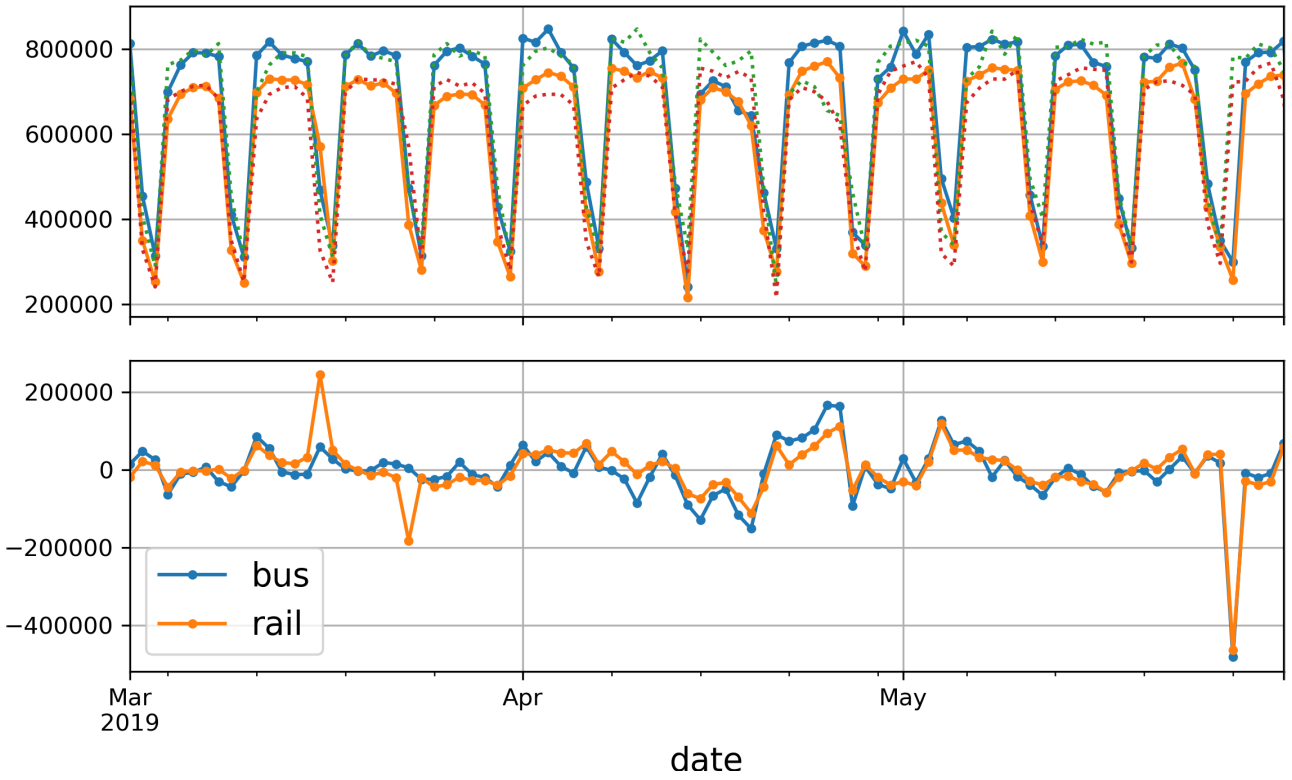


Figure 13-7. Time series overlaid with 7-day lagged time series (top), and difference between t and $t - 7$ (bottom)

Indeed, there was a long weekend back then: the Monday was the Memorial Day holiday. We could use this column to improve our forecasts, but for now let's just measure the mean absolute error over the three-month period we're arbitrarily focusing on—March, April, and May 2019—to get a rough idea:

```
>>> diff_7.abs().mean()
bus     43915.608696
rail    42143.271739
dtype: float64
```

Our naive forecasts get a mean absolute error (MAE) of about 43,916 bus riders, and about 42,143 rail riders. It's hard to tell at a glance how good or bad this is, so let's put the forecast errors into perspective by dividing them by the target values:

```
>>> targets = df[["bus", "rail"]]["2019-03":"2019-05"]
>>> (diff_7 / targets).abs().mean()
bus     0.082938
rail    0.089948
dtype: float64
```

What we just computed is called the *mean absolute percentage error* (MAPE). It looks like our naive forecasts give us a MAPE of roughly 8.3% for bus and 9.0% for rail. It's interesting to note that the MAE for the rail forecasts looks slightly better than the MAE for the bus forecasts, while the opposite is true for the MAPE. That's because the bus ridership is larger than the rail ridership, so naturally the forecast errors are also larger, but when we put the errors into perspective, it turns out that the bus forecasts are actually slightly better than the rail forecasts.

TIP

The MAE, MAPE, and mean squared error (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.

Looking at the time series, there doesn't appear to be any significant monthly seasonality, but let's check whether there's any yearly seasonality. We'll look at the data from 2001 to 2019. To reduce the risk of data snooping, we'll ignore more recent data for now. Let's also plot a 12-month rolling average for each series to visualize long-term trends (see Figure 13-8):

```
period = slice("2001", "2019")
df_monthly = df.select_dtypes(include="number").resample('ME').mean()
rolling_average_12_months = df_monthly.loc[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)
plt.show()
```

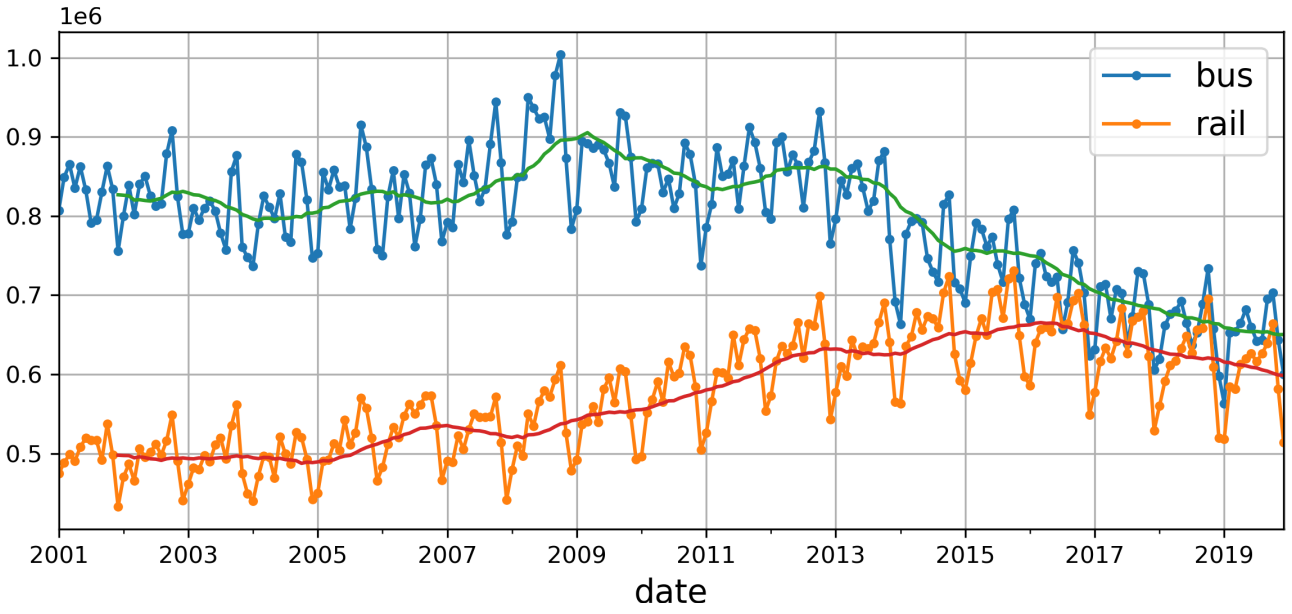


Figure 13-8. Yearly seasonality and long-term trends

Yep! 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: we see peaks and troughs at roughly the same dates each year. Let's check what we get if we plot the 12-month difference

roughly the same dates each year. Let's check what we get if we plot the 12-month difference (see Figure 13-9):

```
df_monthly.diff(12)[period].plot(grid=True, marker=".", figsize=(8, 3))
plt.show()
```

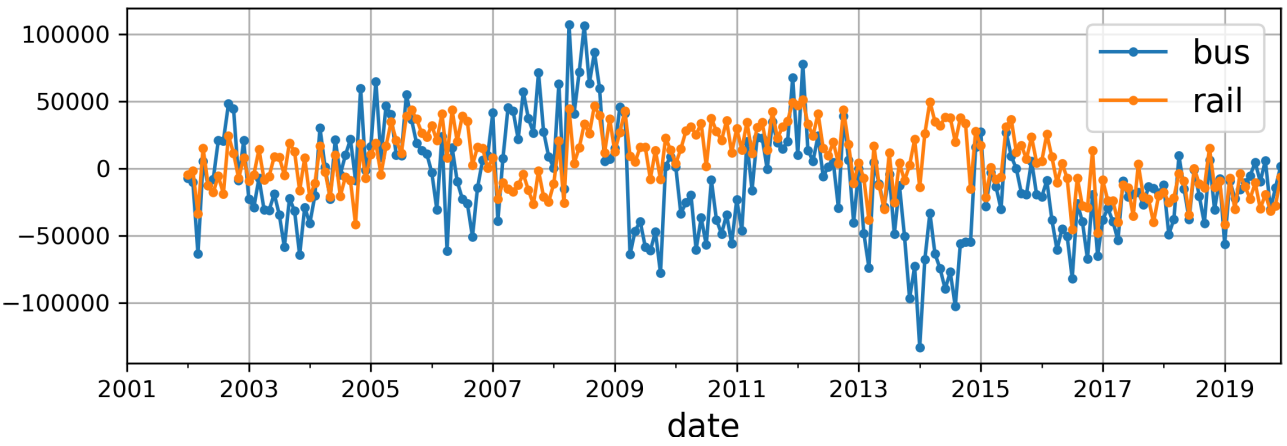


Figure 13-9. The 12-month difference

Notice how differencing not only removed the yearly seasonality, but it 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. In fact, 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 the same 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.

You may be thinking that we're only trying to predict tomorrow's ridership, so the long-term patterns matter much less than the short-term ones. You're right, but still, we may be able to improve performance slightly by taking long-term patterns into account. For example, daily bus ridership dropped by about 2,500 in October 2017, which represents about 570 fewer passengers each week, so if we were at the end of October 2017, it would make sense to forecast tomorrow's ridership by copying the value from last week, minus 570. Accounting for the trend will make your forecasts a bit more accurate on average.

Now that you're familiar with the ridership time series, as well as some of the most important concepts in time series analysis, including seasonality, trend, differencing, and moving averages, let's take a quick look at a very popular family of statistical models that are commonly used to analyze time series.

The ARMA Model Family

We'll start with the *autoregressive moving average* (ARMA) model, developed by Herman Wold in the 1930s: it computes its forecasts using a simple weighted sum of lagged values and corrects these forecasts by adding a moving average, very much like we just discussed. Specifically, the moving average component is computed using a weighted sum of the last few forecast errors. Equation 13-3 shows how the model makes its forecasts.

forecast errors. Equation 13-3 shows how the model makes its forecasts.

Equation 13-3. Forecasting using an ARMA model

In this equation:

- $\hat{y}_{(t)}$ is the model's forecast for time step t.
- $y_{(t)}$ is the time series' value at time step t.
- The first sum is the weighted sum of the past p values of the time series, using the learned weights α_i. The number p is a hyperparameter, and it determines how far back into the past the model should look. This sum is the *autoregressive* component of the model: it performs regression based on past values.
- The second sum is the weighted sum over the past q forecast errors $\varepsilon_{(t)}$, using the learned weights θ_i. The number q is a hyperparameter. This sum is the moving average component of the model.

Importantly, 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].

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*.

Differencing is the central contribution of the *autoregressive integrated moving average* (ARIMA) model, introduced in 1970 by George Box and Gwilym Jenkins in their book *Time Series Analysis* (Wiley). This model 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.

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.

Let's see how to fit a SARIMA model to the rail time series, and use it to make a forecast for tomorrow's ridership. We'll pretend today is the last day of May 2019, and we want to forecast the rail ridership for "tomorrow", the 1st of June, 2019. For this, we can use the `statsmodels` library, which contains many different statistical models, including the ARMA model and its variants, implemented by the `ARIMA` class:

```
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
```

In this code example:

- We start by importing the `ARIMA` class, then we take the rail ridership data from the start of 2019 up to "today", and we use `asfreq("D")` to set the time series' frequency to daily. This doesn't change the data at all in this case, since it's already daily, but without this the `ARIMA` class would have to guess the frequency, and it would display a warning.
- Next, we create an `ARIMA` instance, passing it all the data until "today", and we set the model hyperparameters: `order=(1, 0, 0)` means that $p = 1$, $d = 0$, and $q = 0$; and `seasonal_order=(0, 1, 1, 7)` means that $P = 0$, $D = 1$, $Q = 1$, and $s = 7$. Notice that the `statsmodels` API differs a bit from Scikit-Learn's API, since we pass the data to the model at construction time, instead of passing it to the `fit()` method.
- Next, we fit the model, and we use it to make a forecast for "tomorrow", the 1st of June, 2019.

The forecast is 427,759 passengers, when in fact there were 379,044. Yikes, we're 12.9% off—that's pretty bad. It's actually slightly worse than naive forecasting, which forecasts 426,932, off by 12.6%. But perhaps we were just unlucky that day? To check this, we can 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:

```
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().iloc[0]
    y_preds.append(y_pred)
```

```
    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
```

Ah, that's much better! The MAE is about 32,041, which is significantly lower than the MAE we got with naive forecasting (42,143). So although the model is not perfect, it still beats naive forecasting by a large margin, on average.

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. Good p, q, P, and Q values are usually fairly small (typically 0 to 2, sometimes up to 5 or 6), and d and D are typically 0 or 1, sometimes 2. As for s, it's just the main seasonal pattern's period: in our case it's 7 since there's a strong weekly seasonality. The model with the lowest MAE wins. Of course, you can replace the MAE with another metric if it better matches your business objective. And that's it!

TIP

There are other more principled approaches to selecting good hyperparameters, based on analyzing the *autocorrelation function* (ACF) and *partial autocorrelation function* (PACF),[6] or minimizing the AIC or BIC metrics (introduced in Chapter 8) to penalize models that use too many parameters and reduce the risk of overfitting the data, but grid search is a good place to start.

Preparing the Data for Machine Learning Models

Now that we have two baselines, naive forecasting and SARIMA, let's try to use the machine learning models we've covered so far 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). The inputs to our model will therefore be sequences (usually a single sequence per day once the model is in production), each containing 56 values from time steps $t – 55$ to t. For each input sequence, the model will output a single value: the forecast for time step $t + 1$.

But what will we use as training data? Well, here'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. To do that, we need to create a custom dataset that will chop a given time series into all possible windows of a given length, each with its corresponding target:

```
class TimeSeriesDataset(torch.utils.data.Dataset):
    def __init__(self, series, window_length):
        self.series = series
        self.window_length = window_length
```

```
    def __len__(self):
        return len(self.series) - self.window_length

    def __getitem__(self, idx):
        if idx >= len(self):
            raise IndexError("dataset index out of range")
        end = idx + self.window_length  # 1st index after window
        window = self.series[idx : end]
        target = self.series[end]
        return window, target
```

Let's test this class by applying it to a simple time series containing the numbers 0 to 5. We could represent this series in 1D using [0, 1, 2, 3, 4, 5], but the RNN modules expect each sequence to be 2D, with a shape of [*sequence length*, *dimensionality*]. For univariate time series, the dimensionality is simply 1, so we represent the time series as [[0], [1], [2], [3], [4], [5]]. In the code example below, the `TimeSeriesDataset` contains all the windows of length 3, each with its corresponding target (i.e., the first value after the window):

```
>>> my_series = torch.tensor([[0], [1], [2], [3], [4], [5]])
>>> my_dataset = TimeSeriesDataset(my_series, window_length=3)
>>> for window, target in my_dataset:
...     print("Window:", window, " Target:", target)
...
Window: tensor([[0], [1], [2]])  Target: tensor([3])
Window: tensor([[1], [2], [3]])  Target: tensor([4])
Window: tensor([[2], [3], [4]])  Target: tensor([5])
```

It looks like our `TimeSeriesDataset` class works fine! Now we can create a `DataLoader` for this tiny dataset, shuffling the windows and grouping them into batches of two:

```
>>> from torch.utils.data import DataLoader
>>> torch.manual_seed(0)
>>> my_loader = DataLoader(my_dataset, batch_size=2, shuffle=True)
>>> for X, y in my_loader:
...     print("X:", X, " y:", y)
...
X: tensor([[[0], [1], [2]], [[2], [3], [4]]])  y: tensor([[3], [5]])
X: tensor([[[1], [2], [3]]])  y: tensor([[4]])
```

The first batch contains the windows [[0], [1], [2]] and [[2], [3], [4]], along with their respective targets [3] and [5]; and the second batch contains only one window [[1], [2], [3]], with its target [4]. Indeed, when the length of a dataset is not a multiple of the batch size, the last batch is shorter.

OK, now that we have a way to convert a time series into a dataset that we can use to train ML models, let's go ahead and prepare our ridership dataset. First, we need to split our data into a training period, a validation period, and a test period. We will focus on the rail ridership for

training period, a validation period, and a test period. We will focus on the rail ridership for now. We will convert the data to 32-bit float tensors, and scale them down by a factor of one million to ensure the values end up near the 0–1 range; this plays nicely with the default weight initialization and learning rate. In this code, we use `df[["rail"]]` instead of `df["rail"]` to ensure the resulting tensor has a shape of [*series length*, 1] rather than [*series length*]:

```
rail_train = torch.FloatTensor(df[["rail"]]["2016-01":"2018-12"].values / 1e6)
rail_valid = torch.FloatTensor(df[["rail"]]["2019-01":"2019-05"].values / 1e6)
rail_test = torch.FloatTensor(df[["rail"]]["2019-06":].values / 1e6)
```

NOTE

When dealing with time series, you generally want to split across time. However, in some cases you may be able to split along other dimensions, which will give you a longer time period to train on. For example, if you have data about the financial health of 10,000 companies from 2001 to 2019, you might be able to split this data across the different companies. It's very likely that many of these companies will be strongly correlated, though (e.g., whole economic sectors may go up or down jointly), and if you have correlated companies across the training set and the test set, your test set will not be as useful, as its measure of the generalization error will be optimistically biased.

Next, let's use our `TimeSeriesDataset` class to create datasets for training, validation, and testing, and also create the corresponding data loaders. Since gradient descent expects the instances in the training set to be independent and identically distributed (IID), as we saw in Chapter 4, we must set `shuffle` to `True`—this will shuffle the windows, but not their contents:

```
window_length = 56
train_set = TimeSeriesDataset(rail_train, window_length)
train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
valid_set = TimeSeriesDataset(rail_valid, window_length)
valid_loader = DataLoader(valid_set, batch_size=32)
test_set = TimeSeriesDataset(rail_test, window_length)
test_loader = DataLoader(test_set, batch_size=32)
```

And now we're ready to build and train any regression model we want!

Forecasting Using a Linear Model

Let's try a basic linear model first. We will use the Huber loss, which usually works better than minimizing the MAE directly, as discussed in Chapter 9:

```
import torch.nn as nn
import torchmetrics
```

```
torch.manual_seed(42)
model = nn.Sequential(nn.Flatten(), nn.Linear(window_length, 1)).to(device)
loss_fn = nn.HuberLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
metric = torchmetrics.MeanAbsoluteError().to(device)
[...]  # train the PyTorch model (e.g., using train() function from Chapter 10)
```

Note that we must use a `nn.Flatten` layer before the `nn.Linear` layer, because the inputs have a shape of [*batch size*, *window length*, *dimensionality*], but the `nn.Linear` layer expects inputs of shape [*batch size*, *features*]. If you train this model, you will see that it reaches a validation MAE of 37,726 (your mileage may vary). That's better than naive forecasting, but worse than the SARIMA model.[7]

Can we do better with an RNN? Well, let's see!

Forecasting Using a Simple RNN

Let's implement a simple RNN containing a single recurrent layer (see Figure 13-2) plus a final `nn.Linear` layer that will take the last hidden state as input and output the model's forecast:

```
class SimpleRnnModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.memory_cell = nn.Sequential(
            nn.Linear(input_size + hidden_size, hidden_size),
            nn.Tanh()
        )
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        batch_size, window_length, dimensionality = X.shape
        X_time_first = X.transpose(0, 1)
        H = torch.zeros(batch_size, self.hidden_size, device=X.device)
        for X_t in X_time_first:
            XH = torch.cat((X_t, H), dim=1)
            H = self.memory_cell(XH)
        return self.output(H)

torch.manual_seed(42)
univar_model = SimpleRnnModel(input_size=1, hidden_size=32, output_size=1)
```

Let's go through this code:

- The constructor takes three arguments: the input size, the hidden size, and the output size. In our case, the input size is set to 1 when we create the model (on the very last line) since we are dealing with a univariate time series. The hidden size is the number of recurrent neurons. We set it to 32 in this example, but this is a hyperparameter you can tune. The out-

put size is 1 since we're only forecasting a single value.

- The constructor first creates the memory cell, which will be used once per time step: it's a sequential module composed of a `nn.Linear` layer and the tanh activation function. You can use another activation function here, but it's common to use the tanh activation function because it tends to be more stable than other activation functions in RNNs.
- Next, we create a `nn.Linear` layer that will be used to take the last hidden state and produce the final output. This is needed because the hidden state has one dimension per recurrent neuron (32 in this example), while the target has just one dimension since we're dealing with a univariate time series and we're only trying to forecast a single future value. Moreover, the tanh activation function only outputs values between –1 and +1, while the values we need to forecast occasionally exceed +1.
- The `forward()` method will be passed input batches produced by our data loader, so each batch will have a shape of [*batch size*, *window length*, *dimensionality*], with *dimensionality* = 1.
- The hidden state `H` is initialized to zeros: for each input window, there's one zero per recurrent neuron, so the hidden state's shape is [*batch size*, *hidden_size*].
- Next, we iterate over each time step. For this, we must swap the first two dimensions of `X` using `permute(0, 1)`. As a result, the input tensor `X_t` at each time step has a shape of [*batch size*, *dimensionality*].
- At each time step, we want to feed both the current inputs `X_t` and the hidden state `H` to the memory cell. For this, we must first concatenate `X_t` and `H` along the first dimension, resulting in a tensor `XH` of shape [*batch size*, *input_size* + *hidden size*]. Then we can pass `XH` to the memory cell to get the new hidden state.
- After the loop, `H` represents the final hidden state. We pass it through the output `nn.Linear` layer, and we get our final prediction of shape [*batch size*, *output size*].

In short, this model initializes the hidden state to zeros, then it goes through each time step and applies the memory cell to both the current inputs and the last hidden state, which gives it the new hidden state. It repeats this process until the last time step, then it passes the last hidden state through a linear layer to get the actual forecasts. All of this is performed simultaneously for every sequence in the batch.

So that's our first recurrent model! It's a sequence-to-vector model. Since there's a single output neuron in this case, the output vector for each input sequence has a size of 1.

Now if you move this model to the GPU, then train and evaluate it just like the previous one, you will find that its validation MAE reaches 30,659. That's the best model we've trained so far, and it even beats the SARIMA model; we're doing pretty well!

TIP

We've only normalized the time series, without removing trend and seasonality, and yet the model still performs well. This is convenient, as it makes it possible to quickly search for promising models without worrying too much about preprocessing. However, to get the best performance, you may want to try making the time series more stationary, for example, using differencing.

PyTorch comes with an `nn.RNN` module that can greatly simplify the implementation of our `SimpleRnnModel`. The following implementation is (almost) equivalent to the previous one:

```
class SimpleRnnModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        outputs, last_state = self.rnn(X)
        return self.output(outputs[:, -1])
```

The code is much shorter than before. Let's go through it:

- In the constructor, we now create an `nn.RNN` module instead of building a memory cell. We specify the input size and the hidden size, just like we did earlier, and we also set `batch_first=True` because our input batches have the batch dimension first. If we didn't set `batch_first=True`, the `nn.RNN` module would assume that the time dimension comes first (i.e., it would expect the input batches to have a shape of [*window length*, *batch size*, *dimensionality*] instead of [*batch size*, *window length*, *dimensionality*]).
- The constructor also creates an output layer, exactly like in our previous implementation.
- In the `forward()` method, we pass the input batch directly to the `nn.RNN` module. This takes care of everything: internally, it initializes the hidden state with zeros, and it processes each time step using a simple memory cell based on a linear layer and an activation function (tanh by default), much like we did earlier.
- Note that the `nn.RNN` module returns two things:
  - `outputs` is a tensor containing the outputs of the top recurrent layer at every time step. Right now we have a single recurrent layer, but in the next section we will see that the `nn.RNN` module supports multiple recurrent layers. Since we are dealing with a simple RNN, the outputs are just the hidden states of the top recurrent layer at each time step. The `outputs` tensor has a shape of [*batch size*, *window length*, *hidden size*] (if we didn't set `batch_first=True`, then the first two dimensions would be swapped).
  - `last_state` contains the hidden state of each recurrent layer after the very last time step. Its shape is [*number of layers*, *batch size*, *hidden size*]. In our case, there's a single recurrent layer, so the size of the first dimension is 1.
- In the end, we take the last output (which is also the last state of the top recurrent layer) and we pass it through our `nn.Linear` output layer.

If you train this model, you will get a similar result as before, but generally much faster because the `nn.RNN` module is well optimized. In particular, when using an Nvidia GPU, the `nn.RNN` module leverages the cuDNN library which provides highly optimized implementations of various neural net architectures, including several RNN architectures.

NOTE

The `nn.RNN` module uses two bias parameters: one for the inputs, and the other for the hidden states. It just adds them up, so this really doesn't improve the model at all, but this extra parameter is required by the cuDNN library. This explains why you won't get exactly the same results as before.

Forecasting Using a Deep RNN

It is quite common to stack multiple layers of cells, as shown in Figure 13-10. This gives you a *deep RNN*.

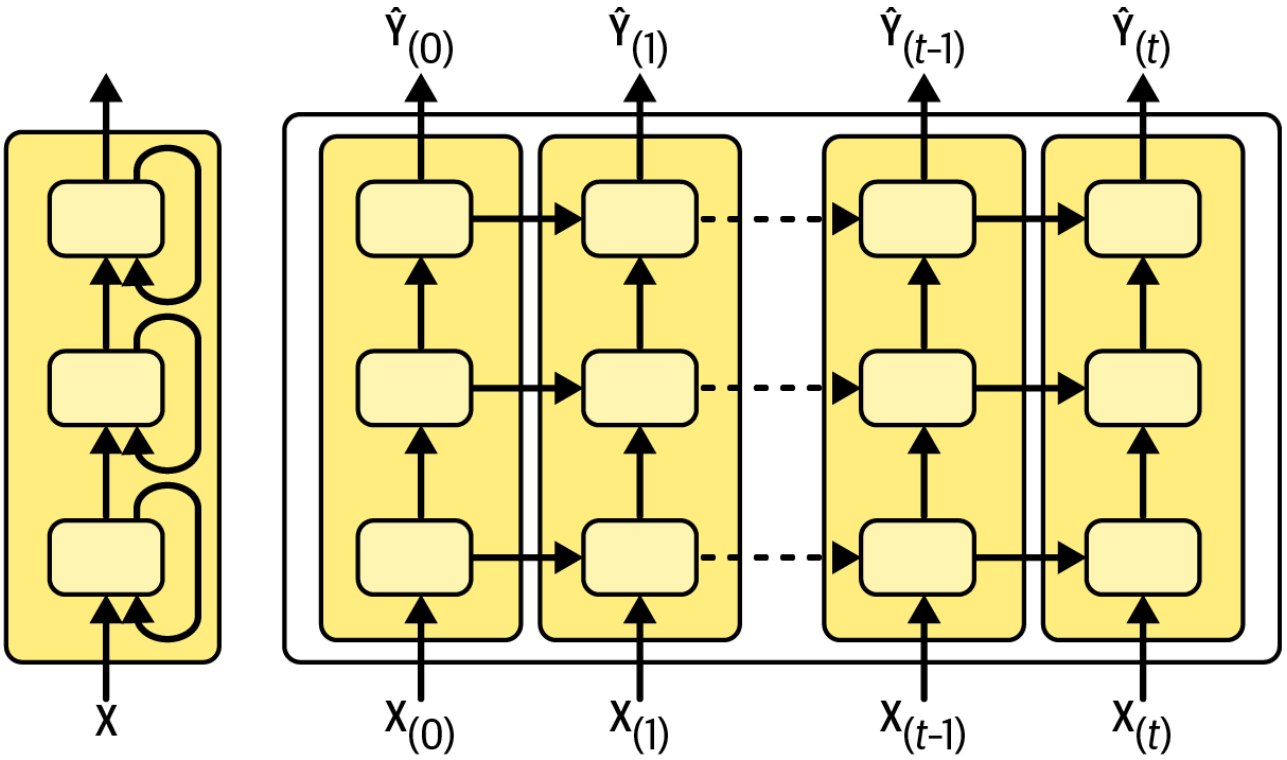


Figure 13-10. A deep RNN (left) unrolled through time (right)

Implementing a deep RNN with PyTorch is straightforward: just set the `num_layers` argument to the desired number of recurrent layers when creating the `nn.RNN` module. For example, if you set `num_layers=3` when creating the `nn.RNN` module in the previous model's constructor, you get a three-layer RNN (the rest of the code remains unchanged):

```
self.rnn = nn.RNN(input_size, hidden_size, num_layers=3, batch_first=True)
```

If you train and evaluate this model, you will find that it reaches an MAE of 29,273. That's our best model so far!

Forecasting Multivariate Time Series

An important quality of neural networks is their flexibility: in particular, they can deal with multivariate time series with almost no change to their architecture. For example, let's try to forecast the rail time series using both the rail and bus data as input. In fact, let's also throw in the day type! Since we can always know in advance whether tomorrow is going to be a week-

day, a weekend, or a holiday, we can shift the day type series one day into the future, so that the model is given tomorrow's day type as input. For simplicity, we'll do this processing using Pandas:

```
df_mulvar = df[["rail", "bus"]] / 1e6  # use both rail & bus 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, dtype=float)  # one-hot encode day type
```

Now `df_mulvar` is a DataFrame with five columns: the rail and bus data, plus three columns containing the one-hot encoding of the next day's type (recall that there are three possible day types, `W`, `A`, and `U`). Next, we can proceed much like we did earlier. First we split the data into three periods, scale it down by a factor of one million, and convert it to tensors:

```
mulvar_train = torch.FloatTensor(df_mulvar["2016-01":"2018-12"].values / 1e6)
mulvar_valid = torch.FloatTensor(df_mulvar["2019-01":"2019-05"].values / 1e6)
mulvar_test = torch.FloatTensor(df_mulvar["2019-06":].values / 1e6)
```

Then we need to create the PyTorch datasets. If we used the `TimeSeriesDataset` for this, the targets would include the next day's rail and bus ridership, as well as the one-hot encoding of the following day type. Since we only want to predict the rail ridership for now, we must tweak the `TimeSeriesDataset` to keep only the first value in the target, which is the rail ridership. One way to do this is to create a new `MulvarTimeSeriesDataset` class that extends the `TimeSeriesDataset` class and tweaks the `__getitem__()` method to filter the target:

```
class MulvarTimeSeriesDataset(TimeSeriesDataset):
    def __getitem__(self, idx):
        window, target = super().__getitem__(idx)
        return window, target[:1]
```

Next, we can create the datasets and the data loaders, much like we did earlier:

```
mulvar_train_set = MulvarTimeSeriesDataset(mulvar_train, window_length)
mulvar_train_loader = DataLoader(mulvar_train_set, batch_size=32, shuffle=True)
[...]  # create the datasets and data loaders for the validation and test sets
```

If you look at the batches produced by the data loaders, you will find that the input shape is [32, 56, 5], and the target shape is [32, 1]. Perfect!

So we can finally create the RNN:

```
torch.manual_seed(42)
mulvar_model = SimpleRnnModel(input_size=5, hidden_size=32, output_size=1)
mulvar_model = mulvar_model.to(device)
```

Notice that this model is identical to the `univar_model` RNN we built earlier, except `input_size=5`: at each time step, the model now receives five inputs instead of one. This model actually reaches a validation MAE of 23,227. Now we're making big progress!

In fact, it's not too hard to make the RNN forecast both the rail and bus ridership. You just need to return `target[:2]` instead of `target[:1]` in the `MulvarTimeSeriesDataset` class, and set `output_size=2` when creating the `SimpleRnnModel`, that's all there is to it!

As we discussed in Chapter 9, using a single model for multiple related tasks often 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). However, it depends on the task, and in this particular case the multitask RNN that forecasts both the bus and the rail ridership doesn't perform quite as well as dedicated models that forecast one or the other (using all five columns as input). Still, it reaches a validation MAE of 26,441 for rail and 26,178 for bus, which is still pretty good.

Forecasting Several Time Steps Ahead

So far we have only predicted the value at the next time step, but we could just as easily have predicted the value several steps ahead by changing the targets appropriately (e.g., to predict the ridership 2 weeks from now, we could just change the targets to be the value 14 days ahead instead of 1 day ahead). But what if we want to predict the next 14 values with a single model?

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 then use the model again to predict the following value, and so on, as in the following code:

```
n_steps = 14
univar_model.eval()
with torch.no_grad():
    X = rail_valid[:window_length].unsqueeze(dim=0).to(device)
    for step_ahead in range(n_steps):
        y_pred_one = univar_model(X)
        X = torch.cat([X, y_pred_one.unsqueeze(dim=0)], dim=1)

    Y_pred = X[0, -n_steps:, 0]
```

In this code, we take the rail ridership of the first 56 days of the validation period, we add a batch dimension of size 1 using the `unsqueeze()` method (since our `univar_model` expects 3D inputs), then we move the tensor to the GPU. Now the shape of `X` is [1, 56, 1]. Then we repeatedly use the model to forecast the next value, and we append each forecast to the input series, along the time axis (`dim=1`). Since each prediction has a shape of [1, 1], we must use `unsqueeze()` again to add a batch dimension of size 1 before we can concatenate it to `X`. In

the end, `X` has a shape of [1, 56 + 14, 1], and our final forecasts are the last 14 values of `X`. The resulting forecasts are plotted in Figure 13-11.

WARNING

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

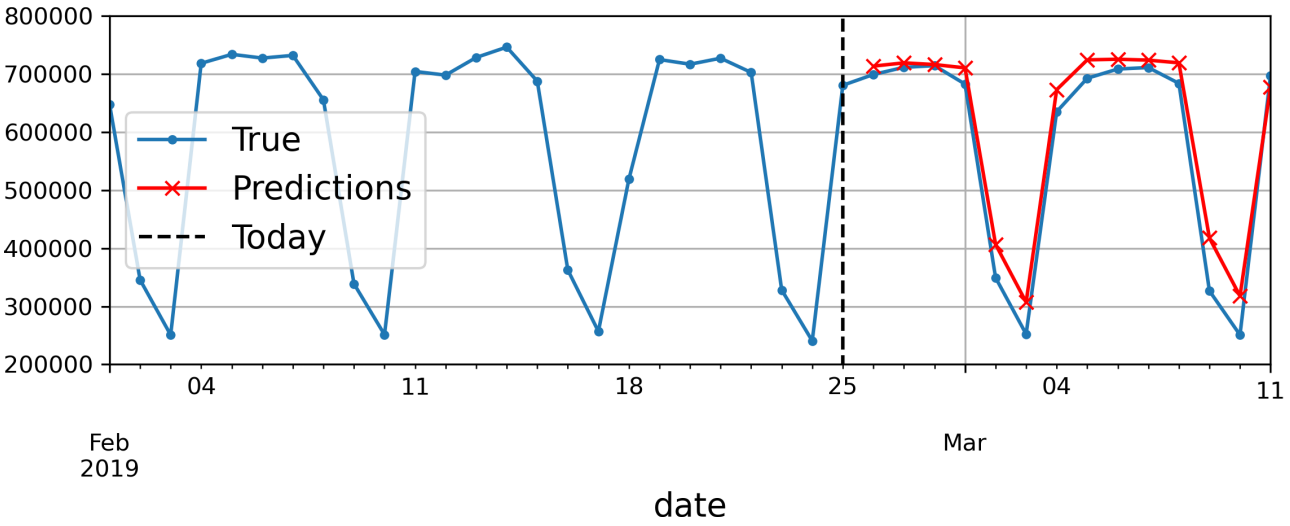


Figure 13-11. Forecasting 14 steps ahead, 1 step at a time

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. However, we first need to change the targets to be vectors containing the next 14 values. For this, we can create the following class:

```
class ForecastAheadDataset(TimeSeriesDataset):
    def __len__(self):
        return len(self.series) - self.window_length - 14 + 1

    def __getitem__(self, idx):
        end = idx + self.window_length  # 1st index after window
        window = self.series[idx : end]
        target = self.series[end : end + 14, 0]  # 0 = rail ridership
        return window, target
```

TIP

I've hardcoded the number 14, but in a real project you should make this configurable (e.g., just like the `window_length`).

This class inherits from the `TimeSeriesDataset` class and tweaks its `__len__()` and `__getitem__()` methods. The target is now a tensor containing the next 14 rail ridership values, rather than just the next value. We can once again create a training set, a validation set,

and a test set, based on the multivariate time series we built earlier:

```
ahead_train_set = ForecastAheadDataset(mulvar_train, window_length)
ahead_train_loader = DataLoader(ahead_train_set, batch_size=32, shuffle=True)
[...]  # create the datasets and data loaders for the validation and test sets
```

Lastly, we can create a simple RNN, just like the `mulvar_model`, but with `output_size=14`:

```
torch.manual_seed(42)
ahead_model = SimpleRnnModel(input_size=5, hidden_size=32, output_size=14)
ahead_model = ahead_model.to(device)
```

After training this model, you can predict the next 14 values at once, like this:

```
ahead_model.eval()
with torch.no_grad():
    window = mulvar_valid[:window_length]  # shape [56, 5]
    X = window.unsqueeze(dim=0)            # shape [1, 56, 5]
    Y_pred = ahead_model(X.to(device))     # shape [1, 14]
```

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 like the previous approach did. Now let's see whether a sequence-to-sequence model can do even better.

TIP

You can combine both approaches to forecast many steps ahead: use a model that forecasts the next 14 days in one shot, then append the forecasts to the inputs and run the model again to get forecasts for the following 14 days, and so on.[8]

Forecasting Using a 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. To be clear, 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 for each input window is not a vector anymore, but a sequence of the same length as the inputs, containing a 14-dimensional vector at each step. Given an input batch of shape [*batch size*, *window length*, *input size*], the output will have a shape of [*batch size*, *window length*, *output_size*]. This is no longer a sequence-to-vector RNN, it's a sequence-to-sequence (or *seq2seq*) RNN.

NOTE

It may be surprising that the targets contain values that appear in the inputs (except for the last time step). Isn't that cheating? Fortunately, not at all: at each time step, an RNN only knows about past time steps; it cannot look ahead. It is said to be a *causal* model.

You may be wondering why we would want to train a seq2seq model when we're really only interested in forecasting future values, which are output by our model at the very last time step. And you're right: after training, you can actually ignore all outputs except for the very last time step. The main 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 can both stabilize training and speed up convergence. Moreover, since the model must make predictions at each time step, it will see input sequences of varying lengths, which can reduce the risk of overfitting the model to the specific window length used during training. Well, at least that's the hope! Let's give this technique a try on the rail ridership time series. As usual, we first need to prepare the dataset:

```
class Seq2SeqDataset(ForecastAheadDataset):
    def __getitem__(self, idx):
        end = idx + self.window_length  # 1st index after window
        window = self.series[idx : end]
        target_period = self.series[idx + 1 : end + 14, 0]
        target = target_period.unfold(dimension=0, size=14, step=1)
        return window, target
```

Our new `Seq2SeqDataset` class inherits from the `ForecastAheadDataset` class and overrides the `__getitem__()` method: the input window is defined just like before, but the target is now a sequence of consecutive windows, shifted by one time step at each time step. The `unfold()` method is where the magic happens: it takes a tensor and produces sliding blocks from it. For this, it repeatedly slides along the given dimension by the given number of steps and extracts a block of the given size. For example:

```
>>> torch.tensor([0, 1, 2, 3, 4, 5]).unfold(dimension=0, size=4, step=1)
tensor([[0, 1, 2, 3],
        [1, 2, 3, 4],
        [2, 3, 4, 5]])
```

Once again we must create a training set, a validation set, and a test set, as well as the corresponding data loaders:

```
seq_train_set = Seq2SeqDataset(mulvar_train, window_length)
seq_train_loader = DataLoader(seq_train_set, batch_size=32, shuffle=True)
[...]  # create the datasets and data loaders for the validation and test sets
```

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

```
class Seq2SeqRnnModel(SimpleRnnModel):
    def forward(self, X):
        outputs, last_state = self.rnn(X)
        return self.output(outputs)

torch.manual_seed(42)
seq_model = Seq2SeqRnnModel(input_size=5, hidden_size=32, output_size=14)
seq_model = seq_model.to(device)
```

We inherit from the `SimpleRnnModel` class, and we override the `forward()` method. Instead of applying the linear `self.output` layer only to the outputs of the last time step, as we did before, we now apply it to the outputs of every time step. It may surprise you that this works at all. So far, we have only applied `nn.Linear` layers to 2D inputs of shape [*batch size*, *features*], but here the `outputs` tensor has a shape of [*batch size*, *window length*, *hidden size*]: it's 3D, not 2D! Luckily, this works fine as the `nn.Linear` layer will automatically be applied to each time step, so the model's predictions will have a shape of [*batch size*, *window length*, *output size*]: just what we need.

Under the hood, the `nn.Linear` layer relies on `torch.matmul()` for matrix multiplication. This function efficiently supports multiplying arrays of more than two dimensions. For example, you can multiply an array of shape [2, 3, 5, 7] with an array of shape [2, 3, 7, 11]. Indeed, these two arrays can both be seen as 2 × 3 grids of matrices, and `torch.matmul()` simply multiplies the corresponding matrices in both grids. Since multiplying a 5 × 7 matrix with a 7 × 11 matrix produces a 5 × 11 matrix, the final result is a 2 × 3 grid of 5 × 11 matrices, represented as a tensor of shape [2, 3, 5, 11]. Broadcasting is also supported; for example, you can multiply an array of shape [10, 56, 32] with an array of shape [32, 14]: each of the ten 56 × 32 matrices in the first array will be multiplied by the same 32 × 14 matrix in the second array, and you will get a tensor of shape [10, 56, 14]. That's what happens when you pass a 3D input to a `nn.Linear` layer.

TIP

Another way to get the exact same result is to replace the `nn.Linear` output layer with a `nn.Conv1d` layer using a kernel size of one (i.e., `Conv1d(32, 14, kernel_size=1)`). However, you would have to swap the last two dimensions of both the inputs and the outputs, treating the time dimension as a spatial dimension.

The training code is the same as usual. During training, all the model's outputs are used, but after training, only the outputs of the very last time step matter, and the rest can be ignored (as mentioned earlier). For example, we can forecast the rail ridership for the next 14 days like this:

```
seq_model.eval()
with torch.no_grad():
    some_window = mulvar_valid[:window_length]  # shape [56, 5]
    X = some_window.unsqueeze(dim=0)            # shape [1, 56, 5]
    Y_preds = seq_model(X.to(device))           # shape [1, 56, 14]
    Y_pred = Y_preds[:, -1]                     # shape [1, 14]
```

If you evaluate this model's forecasts for $t + 1$, you will find a validation MAE of 23,350, which is very good. Of course, the model is not as accurate for more distant forecasts. For example, the MAE for $t + 14$ is 35,315.

Simple RNNs can be quite good at forecasting time series or handling other kinds of sequences, but they do not perform as well on long time series or sequences. Let's discuss why and see what we can do about it.

Handling Long Sequences

To 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.

Fighting the Unstable Gradients Problem

Many of the tricks we used in deep nets to alleviate the unstable gradients problem can also be used for RNNs: good parameter initialization, faster optimizers, dropout, and so on. However, nonsaturating activation functions (e.g., ReLU) may not help as much here. In fact, they may actually lead the RNN to be even more unstable during training. Why? Well, suppose 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.

You can reduce this risk by using a smaller learning rate, or you can use a saturating activation function like the hyperbolic tangent (this explains why it's the default).

In much the same way, the gradients themselves can explode. If you notice that training is unstable, you may want to monitor the size of the gradients and perhaps use gradient clipping.

Moreover, batch normalization cannot be used as efficiently with RNNs as with deep feedforward nets. In fact, you cannot use it between time steps, only between recurrent layers. To be more precise, it is technically possible to add a BN layer to a memory cell so that it will be applied at each time step (both on the inputs for that time step and on the hidden state from the

previous step). However, this implies that the same BN 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 was demonstrated by César Laurent et al. in a 2015 paper.[9] The authors found that BN was slightly beneficial only when it was applied to the layer's inputs, not to the hidden states. In other words, it was slightly better than nothing when applied between recurrent layers (i.e., vertically in Figure 13-10), but not within recurrent layers (i.e., horizontally).

Layer norm (introduced in Chapter 11) tends to work a bit better than BN within recurrent layers. It is usually applied just before the activation function, at each time step. Sadly, PyTorch's `nn.RNN` module does not support LN, so you have to implement the RNN's loop manually (as we did earlier), and apply the `nn.LayerNorm` module at each iteration. This is not too hard, but you do lose the simplicity and speed of the `nn.RNN` module. For example, you can take the first version of our `SimpleRnnModel` class and add a `nn.LayerNorm` module to the memory cell, just before the tanh activation function:

```
self.memory_cell = nn.Sequential(
    nn.Linear(input_size + hidden_size, hidden_size),
    nn.LayerNorm(hidden_size),
    nn.Tanh()
)
```

WARNING

Layer norm does not always help; you just have to try it. In general, it works better in *gated RNNs* such as LSTM and GRU (discussed shortly). It is also more likely to help when the time series is preprocessed to remove any seasonality or trend.

Similarly, if you wish to apply dropout between each time step, you must write a custom RNN since the `nn.RNN` module does not support that. However, it does support adding a dropout layer after every recurrent layer: simply set the `dropout` hyperparameter to the desired dropout rate (it defaults to zero).

With these techniques, you can alleviate the unstable gradients problem and train an RNN much more efficiently. Now let's look at how to deal with the short-term memory problem.

TIP

When forecasting time series, it is often useful to have some error bars along with your predictions. For this, one approach is to use MC dropout (introduced in Chapter 11).

Tackling the Short-Term Memory Problem

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

tion is lost at each time step. After a while, the RNN's state contains virtually no trace of the first inputs. This can be a showstopper. Imagine Dory the fish[10] trying to translate a long sentence; by the time she's finished reading it, she has no clue how it started. To tackle this problem, various types of cells with long-term memory have been introduced. They have proven so successful that the basic cells are not used much anymore. Let's first look at the most popular of these long-term memory cells: the LSTM cell.

LSTM cells

The *long short-term memory* (LSTM) cell was proposed in 1997[11] by Sepp Hochreiter and Jürgen Schmidhuber and gradually improved over the years by several researchers, such as Alex Graves, Haşim Sak,[12] and Wojciech Zaremba.[13] You can simply use the `nn.LSTM` module instead of the `nn.RNN` module; it's a drop-in replacement, and it usually performs much better: training converges faster, and the model detects longer-term patterns in the data.

So how does this magic work? Well, the LSTM architecture is shown in Figure 13-12. If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c" stands for "cell"). You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.

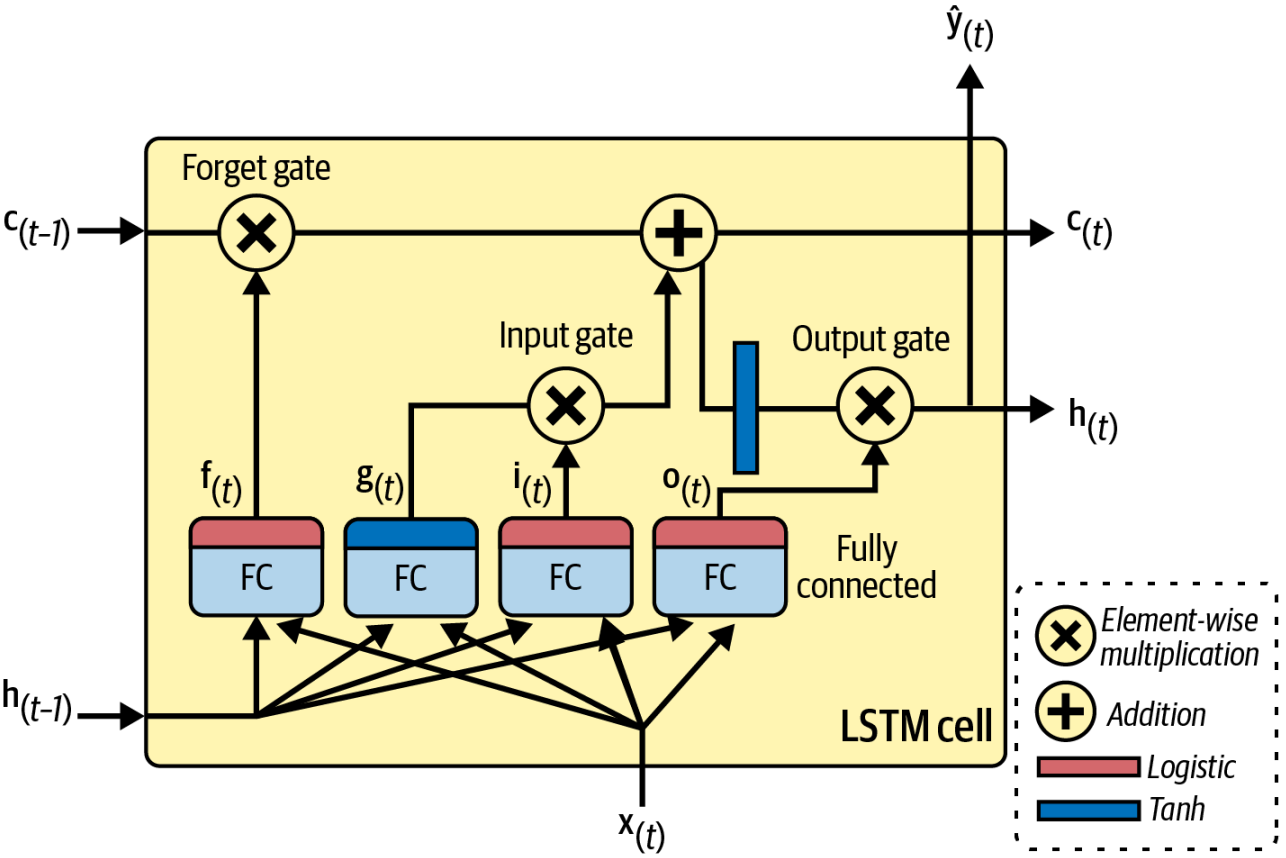


Figure 13-12. An LSTM cell

Now let's open the box! 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. As the long-term state $\mathbf{c}_{(t–1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $\mathbf{c}_{(t)}$ is sent straight out without any further transformation. So at each time step, some memories are dropped and some mem-

ories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and the result is filtered by the *output gate*. This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cell's output for this time step, $\mathbf{y}_{(t)}$). Now let's look at where new memories come from and how the gates work.

First, the current input vector $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t–1)}$ are fed to four different fully connected layers. They all serve a different purpose:

- The main layer is the one that outputs $\mathbf{g}_{(t)}$. It has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous (short-term) state $\mathbf{h}_{(t–1)}$. In a simple RNN cell, there is nothing other than this layer, and its output goes straight out to $\mathbf{y}_{(t)}$ and $\mathbf{h}_{(t)}$. But in an LSTM cell, this layer's output does not go straight out; instead its most important parts are stored in the long-term state (and the rest is dropped).
- The three other layers are *gate controllers*. Since they use the logistic activation function, the outputs range from 0 to 1. As you can see, the gate controllers' outputs are fed to element-wise multiplication operations: if they output 0s they close the gate, and if they output 1s they open it. Specifically:
  – The *forget gate* (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.
  – The *input gate* (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.
  – Finally, the *output gate* (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $\mathbf{h}_{(t)}$ and to $\mathbf{y}_{(t)}$.

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 (that's the role of the output gate), all while being fully differentiable. This explains why these cells have been amazingly successful at capturing long-term patterns in time series, long texts, audio recordings, and more.

Equation 13-4 summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

Equation 13-4. LSTM computations

In this equation:

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, and $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$

- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, and $\mathbf{b}_g$ are the bias terms for each of the four layers.

TIP

Try replacing `nn.RNN` with `nn.LSTM` in the previous models and see what performance you can reach on the ridership dataset (a bit of hyperparameter tuning may be required).

Just like for simple RNNs, if you want to add layer normalization or dropout at each time step, you must implement the recurrent loop manually. One option is to use Equation 13-4, but a simpler option is to use the `nn.LSTMCell` module, which runs a single time step. For example, here is a simple implementation:

```
class LstmModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.memory_cell = nn.LSTMCell(input_size, hidden_size)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        batch_size, window_length, dimensionality = X.shape
        X_time_first = X.transpose(0, 1)
        H = torch.zeros(batch_size, self.hidden_size, device=X.device)
        C = torch.zeros(batch_size, self.hidden_size, device=X.device)
        for X_t in X_time_first:
            H, C = self.memory_cell(X_t, (H, C))
        return self.output(H)
```

This is very similar to the first implementation of our `SimpleRnnModel`, but we are now using an `nn.LSTMCell` at each time step, and the hidden state is now split in two parts: the short-term `H` and the long-term `C`.

There are several variants of the LSTM cell. One particularly popular variant is the GRU cell, which we will look at now.

GRU cells

The *gated recurrent unit* (GRU) cell (see Figure 13-13) was proposed by Kyunghyun Cho et al. in a 2014 paper[14] that also introduced the encoder-decoder network we discussed earlier.

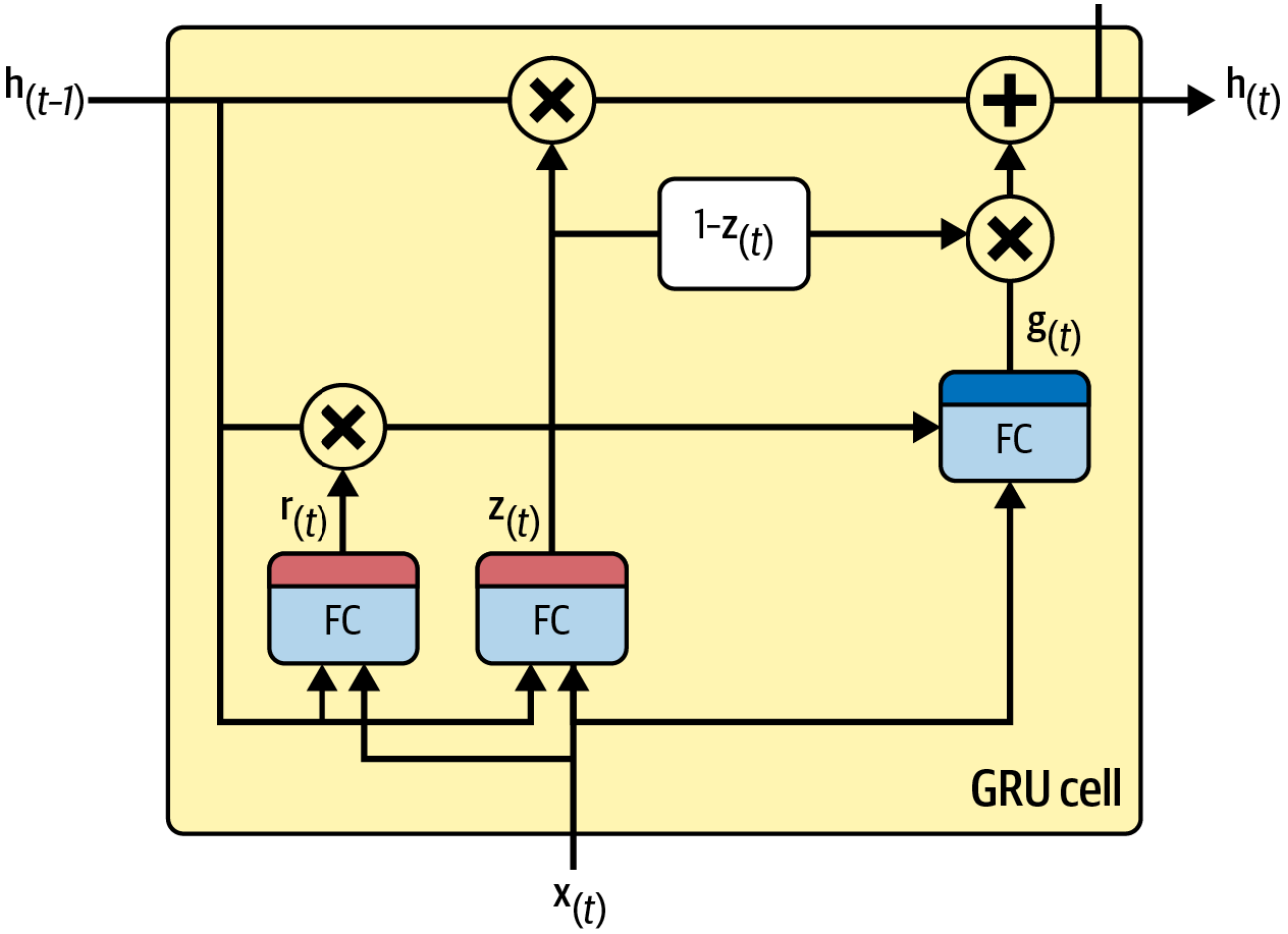


Figure 13-13. GRU cell

The GRU cell is a simplified version of the LSTM cell, and it often performs just as well.[15] These are the main simplifications:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$.
- A single gate controller $\mathbf{z}_{(t)}$ 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 (1 – 1 = 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 $\mathbf{r}_{(t)}$ that controls which part of the previous state will be shown to the main layer ($\mathbf{g}_{(t)}$).

Equation 13-5 summarizes how to compute the cell's state at each time step for a single instance.

Equation 13-5. GRU computations

PyTorch provides a `nn.GRU` layer; using it is just a matter of replacing `nn.RNN` or `nn.LSTM` with `nn.GRU`. It also provides a `nn.GRUCell` in case you want to create a custom RNN based on a GRU cell (just replace `nn.LSTMCell` with `nn.GRUCell` in the previous example, and get

rid of `C`).

LSTM and GRU are one of the main reasons behind the success of RNNs. Yet while they can tackle much longer sequences than simple RNNs, they 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.

Using 1D convolutional layers to process sequences

In Chapter 12, we saw that a 2D convolutional layer works by sliding several fairly small kernels (or filters) across an image, producing multiple 2D feature maps (one per kernel). Similarly, 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). If you use 10 kernels, then the layer's output will be composed of 10 1D sequences (all of the same length), or equivalently you can view this output as a single 10D sequence. This means that you can build a neural network composed of a mix of recurrent layers and 1D convolutional layers (or even 1D pooling layers). However, as mentioned earlier, you must swap the last two dimensions of the `nn.Conv1d` layer's inputs and outputs, since the `nn.Conv1d` layer expects inputs of shape [*batch size*, *input features*, *sequence length*], and produces outputs of shape [*batch size*, *output features*, *sequence length*].

WARNING

If you use a 1D convolutional layer with a stride of 1 and `"same"` padding, then the output sequence will have the same length as the input sequence. But if you use `"valid"` padding or a stride greater than 1, then the output sequence will be shorter than the input sequence, so make sure you adjust the targets accordingly.

For example, the following model is composed of a 1D convolutional layer, followed by a GRU layer, and lastly a linear output layer, all of which input and output batches of sequences (i.e., 3D tensors). The `nn.Conv1d` layer downsamples the input sequences by a factor of 2, using a stride of 2. The kernel size is as large as the stride (larger, in fact), so all inputs will be used to compute the layer's output, and therefore the model can learn to preserve the most useful information, dropping only the unimportant details. In the `forward()` method, we just chain the layers, but we permute the last two dimensions before and after the `nn.Conv1d` layer, and we ignore the hidden states returned by the `nn.GRU` layer:

```
class DownsamplingModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.conv = nn.Conv1d(input_size, hidden_size, kernel_size=4, stride=2)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, X):
```

```
        def forward(self, X):
            Z = X.permute(0, 2, 1)  # treat time as a spatial dimension
            Z = self.conv(Z)
            Z = Z.permute(0, 2, 1)  # swap back time & features dimensions
            Z = torch.relu(Z)
            Z, _states = self.gru(Z)
            return self.linear(Z)

torch.manual_seed(42)
dseq_model = DownsamplingModel(input_size=5, hidden_size=32, output_size=14)
dseq_model = dseq_model.to(device)
```

By shortening the time series, the convolutional layer helps the `GRU` layer detect longer patterns, so we can afford to double the window length to 112 days. Note that we must also crop off the first three time steps from 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, therefore the first forecasts must 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. For all this, we need a new `Dataset` class, so let's create a subclass of the `Seq2SeqDataset` class:

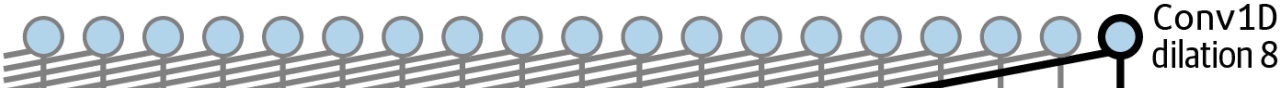
```
class DownsampledDataset(Seq2SeqDataset):
    def __getitem__(self, idx):
        window, target = super().__getitem__(idx)
        return window, target[3::2]  # crop the first 3 targets and downsample

window_length = 112
dseq_train_set = DownsampledDataset(rail_train, window_length)
dseq_train_loader = DataLoader(dseq_train_set, batch_size=32, shuffle=True)
[...]  # create the datasets and data loaders for the validation and test sets
```

And now the model can be trained as usual. We've successfully mixed convolutional layers and recurrent layers. But what if we used only 1D convolutional layers and dropped the recurrent layers entirely?

WaveNet

In a 2016 paper,[16] Aaron van den Oord and other DeepMind researchers introduced a novel architecture called *WaveNet*. They 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 (see Figure 13-14). 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.



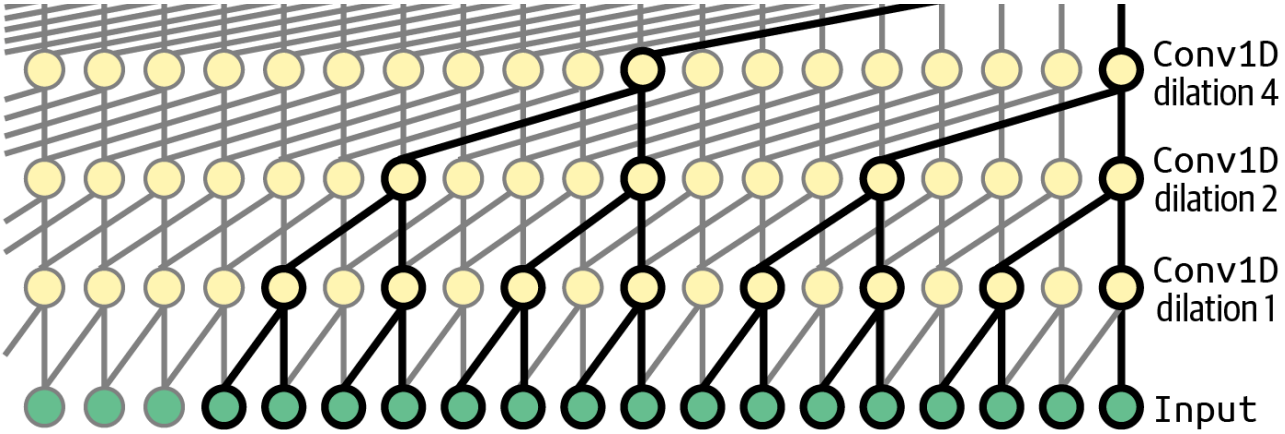


Figure 13-14. WaveNet architecture

The authors of the paper actually stacked 10 convolutional layers with dilation rates of 1, 2, 4, 8, …, 256, 512, then they stacked another group of 10 identical layers (also with dilation rates 1, 2, 4, 8, …, 256, 512), then again another identical group of 10 layers. They justified this architecture by pointing out that a single stack of 10 convolutional layers with these dilation rates will act like a super-efficient convolutional layer with a kernel of size 1,024 (except way faster, more powerful, and using significantly fewer parameters). They also left-padded 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. Padding on the left rather than on both sides is important, as it ensures that the convolutional layer does not peek into the future when making predictions. This makes it a causal model.

Let's implement a simplified WaveNet to tackle the same sequences as earlier.[17] We will start by creating a custom `CausalConv1d` module that acts just like a `nn.Conv1d` module, except the inputs get padded on the left side by the appropriate amount to ensure the sequence preserves the same length:

```
mport torch.nn.functional as F

class CausalConv1d(nn.Conv1d):
    def forward(self, X):
        padding = (self.kernel_size[0] - 1) * self.dilation[0]
        X = F.pad(X, (padding, 0))
        return super().forward(X)
```

In this code, we inherit from the `nn.Conv1d` class and we override the `forward()` method. In it, we calculate the size of the left-padding we need, and we pad the sequences using the `pad()` function before calling the base class's `forward()` method. The `pad()` function takes two arguments: the tensor to pad (`X`), and a tuple of ints that indicates how much to pad to the left and right in the last dimension (i.e., the time dimension).

Now we're ready to build the Wavenet model itself:

```
class WavenetModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
```

```
        layers = []
        for dilation in (1, 2, 4, 8) * 2:
            conv = CausalConv1d(input_size, hidden_size, kernel_size=2,
                                dilation=dilation)
            layers += [conv, nn.ReLU()]
            input_size = hidden_size
        self.convs = nn.Sequential(*layers)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, X):
        Z = X.permute(0, 2, 1)
        Z = self.convs(Z)
        Z = Z.permute(0, 2, 1)
        return self.output(Z)

torch.manual_seed(42)
wavenet_model = WavenetModel(input_size=5, hidden_size=32, output_size=14)
wavenet_model = wavenet_model.to(device)
```

In the constructor, we create eight `CausalConv1d` layers with various dilation rates (1, 2, 4, 8, then again 1, 2, 4, 8), each followed by the ReLU activation function. We chain all these modules in a `nn.Sequential` module `self.convs`. We also create the output `nn.Linear` layer. In the forward method, we permute the last two dimensions of the inputs, as we did earlier, we then pass them through the convolutional layers, then we permute the last two dimensions back to their original order, and we pass the result through the output layer. Thanks to the causal padding, every convolutional layer outputs a sequence of the same length as its input sequence, so the targets we use during training can be the full 112-day sequences; no need to crop them or downsample them. Thus, we can train the model using the data loaders we built for the `Seq2SeqModel` (i.e., `seq_train_loader` and `seq_valid_loader`).

The models we've discussed in this section offer similar performance for the ridership forecasting task, but they may vary significantly depending on the task and the amount of available data. In the WaveNet paper, the authors achieved state-of-the-art performance on various audio tasks (hence the name of the architecture), including text-to-speech tasks, producing very realistic voices across several languages. They also used the model to generate music, one audio sample at a time. This feat is all the more impressive when you realize that a single second of audio can contain tens of thousands of time steps—even LSTMs and GRUs cannot handle such long sequences.

WARNING

If you evaluate our best Chicago ridership models on the test period, starting in 2020, you will find that they perform much worse than expected! Why is that? Well, that's when the Covid-19 pandemic started, which greatly affected public transportation. As mentioned earlier, these models will only work well if the patterns they learned from the past continue in the future. In any case, before deploying a model to production, verify that it works well on recent data. And once it's in production, make sure to monitor its performance regularly.

With that, you can now tackle all sorts of time series! In Chapter 14, we will continue to explore RNNs, and we will see how they can tackle various NLP tasks as well.

Exercises

1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?
2. How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?
3. How can you build a deep sequence-to-sequence RNN in PyTorch?
4. Suppose you have a daily univariate time series, and you want to forecast the next seven days using an RNN. Which architecture should you use?
5. What are the main difficulties when training RNNs? How can you handle them?
6. Can you sketch the LSTM cell's architecture?
7. Why would you want to use 1D convolutional layers in an RNN?
8. Which neural network architecture could you use to classify videos?
9. Try to tweak the `Seq2SeqModel` model to forecast both rail and bus ridership for the next 14 days. The model will now need to predict 28 values instead of 14.
10. 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). 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. Then use this 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. Also make sure to check out Google's Coconet model, which was used for a nice Google doodle about Bach.
11. Train a classification model for the QuickDraw dataset, which contains millions of sketches of various objects. Start by downloading the simplified data for a few classes (e.g., *ant.ndjson*, *axe.ndjson*, and *bat.ndjson*). Each NDJSON file contains one JSON object per line, which you can parse using Python's `json.loads()` function. This will give you a list of sketches, where each sketch is represented as a Python dictionary. In each dictionary, the `"drawing"` entry contains a list of pen strokes. You can convert this list to a 3D float tensor where the dimensions are [*strokes*, *x coordinates*, *y coordinates*]. Since an RNN takes a single sequence as input, you will need to concatenate all the strokes for each sketch into a single sequence. It's best to add an extra feature to allow the RNN to know how far along each stroke it currently is (e.g., from 0 to 1). In other words, the model will receive a sequence where each time step has three features: the x and y coordinates of the pen, and the progress ratio along the current stroke.
12. Create a dataset containing short audio recordings of you saying "yes" or "no", and train a binary classification RNN on it. For example, you could:
    a. Use an audio recording software such as Audacity to record yourself saying "yes" as

many times as your patience allows, with short pauses between each word. Create a similar recording for the word "no". Try to cover the various ways you might realistically pronounce these words in real life.

b. Load each WAV file using the `torchaudio.load()` function from the TorchAudio library. This will return a tensor containing the audio, as well as an integer indicating the number of samples per second. The audio tensor has a shape of [*channels*, *samples*]: one channel for mono, two for stereo. Convert stereo to mono by averaging over the channels dimension.

c. Chop each recording into individual words by splitting at the silences. You can do this using the `torchaudio.transforms.Vad` transform (Voice Activity Detection).

d. Since the sequences are so long, it's hard to directly train an RNN on them, so it helps to convert the audio to a spectrogram first. For this, you can use the `torchaudio.transforms.MelSpectrogram` transform, which is well suited for voice. The output is a dramatically shorter sequence, with many more channels.

e. Now try building and training a binary classification RNN on your yes/no dataset! Consider sharing your dataset and model with the world (e.g., via the Hugging Face Hub).

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

[1] Note that many researchers prefer to use the hyperbolic tangent (tanh) activation function in RNNs rather than the ReLU activation function. For example, see Vu Pham et al.'s 2013 paper "Dropout Improves Recurrent Neural Networks for Handwriting Recognition". ReLU-based RNNs are also possible, as shown in Quoc V. Le et al.'s 2015 paper "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units".

[2] Michael I. Jordan, "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine", *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (1986).

[3] Jeffrey L. Elman, "Finding Structure in Time", *Cognitive Science*, Volume 14, Issue 2 (1990)

[4] Nal Kalchbrenner and Phil Blunsom, "Recurrent Continuous Translation Models", *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (2013): 1700–1709.

[5] The latest data from the Chicago Transit Authority is available at the Chicago Data Portal.

[6] For more details on the ACF-PACF approach, check out this very nice post by Jason Brownlee.

[7] Note that the validation period starts on the 1st of January 2019, so the first prediction is for the 26th of February 2019, eight weeks later. When we evaluated the baseline models, we used predictions starting on the 1st of March instead, but this should be close enough.

[8] We cannot use the `ahead_model` for this because it needs both the rail and bus ridership as input, but it only forecasts the rail ridership.

[9] César Laurent et al., "Batch Normalized Recurrent Neural Networks", *Proceedings of the IEEE*

International Conference on Acoustics, Speech, and Signal Processing (2016): 2657–2661.

[10] A character from the animated movies *Finding Nemo* and *Finding Dory* who has short-term memory loss.

[11] Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory", *Neural Computation* 9, no. 8 (1997): 1735–1780.

[12] Haşim Sak et al., "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition", arXiv preprint arXiv:1402.1128 (2014).

[13] Wojciech Zaremba et al., "Recurrent Neural Network Regularization", arXiv preprint arXiv:1409.2329 (2014).

[14] Kyunghyun Cho et al., "Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation", *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014): 1724–1734.

[15] See Klaus Greff et al., "LSTM: A Search Space Odyssey", *IEEE Transactions on Neural Networks and Learning Systems* 28, no. 10 (2017): 2222–2232. This paper seems to show that all LSTM variants perform roughly the same.

[16] Aaron van den Oord et al., "WaveNet: A Generative Model for Raw Audio", arXiv preprint arXiv:1609.03499 (2016).

[17] The complete WaveNet uses a few more tricks, such as skip connections like in a ResNet, and *gated activation units* similar to those found in a GRU cell.