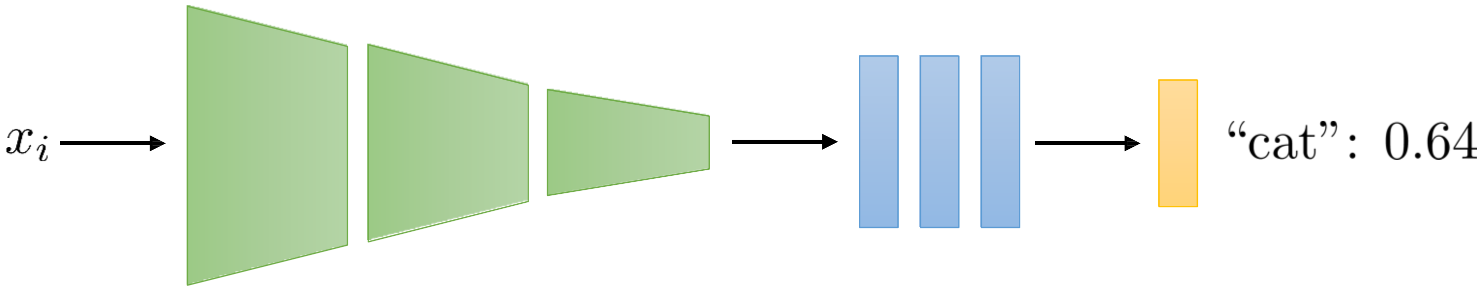
Hands-on Machine Learning

15. Recurrent Neural Networks

What if we have variable-size inputs?

Before:



Now:

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$

$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$

$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$

Examples:

classifying sentiment for a phrase (sequence of words)

predicting price of a stock (sequence of numbers)

classifying the activity in a video (sequence of images)

What if we have variable-size inputs?

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$

$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$

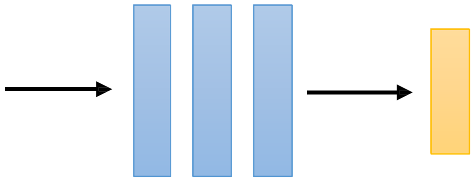
$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$

+ very simple, and can work if necessary

- doesn’t scale very well for very long sequences

Simple idea: zero-pad up to length of longest sequence

$$(x_{i,1}, x_{i,2}, x_{i,3}, 0, 0, 0)$$



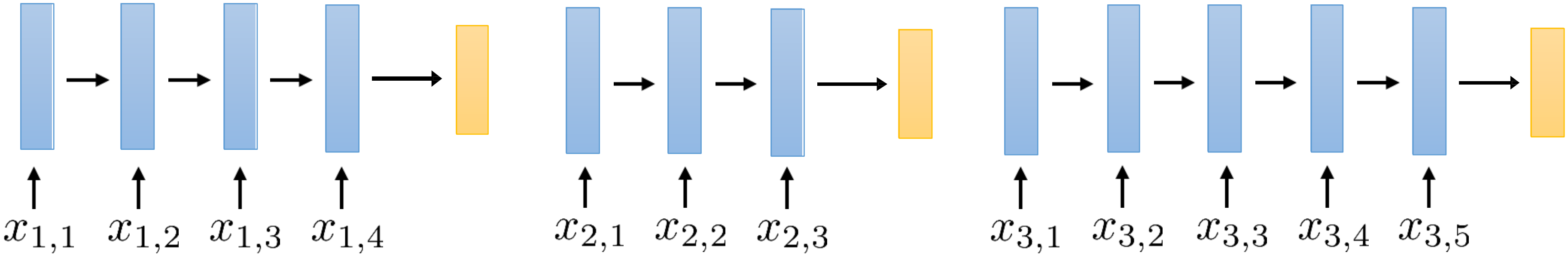
One Input per Layer

$$x_1 = (x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4})$$

$$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$$

$$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$$

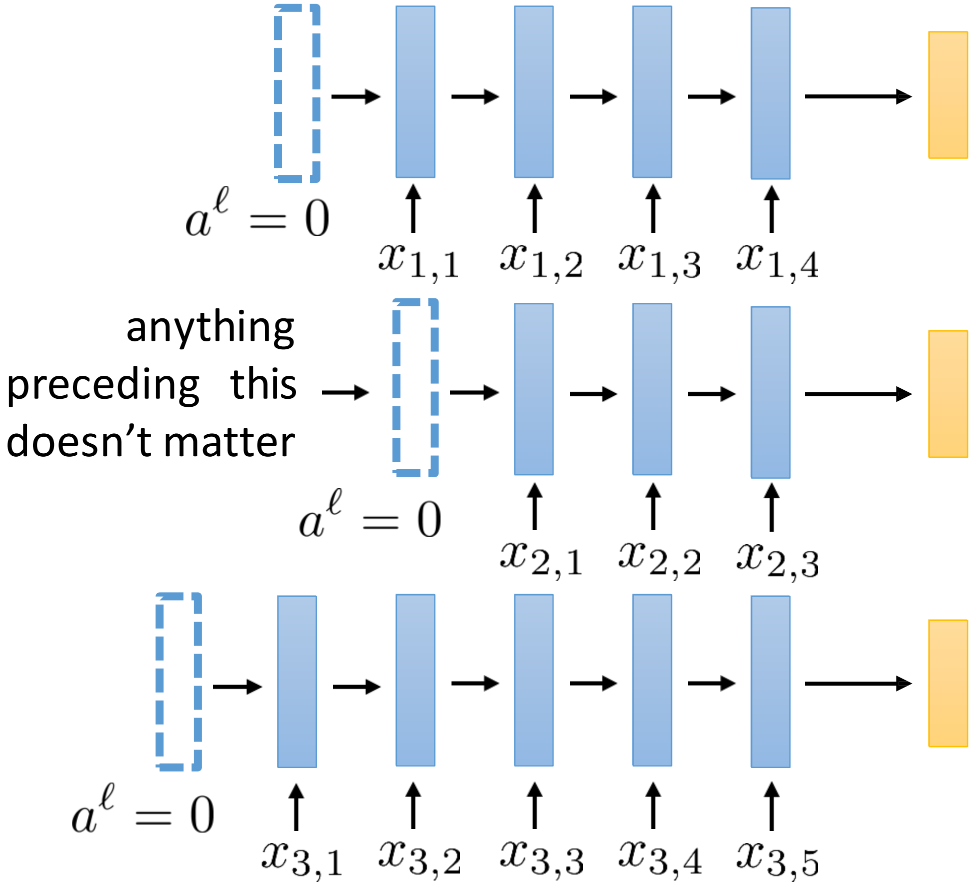
what happens to the missing layers?



each layer:

$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix} \qquad z^\ell = W^\ell \bar{a}^{\ell-1} + b^\ell \qquad a^\ell = \sigma(z^\ell)$$

Variable Layer Count



- The shorter the sequence, the fewer layers we have to evaluate
- But the total number of weight matrices increases with max sequence length!

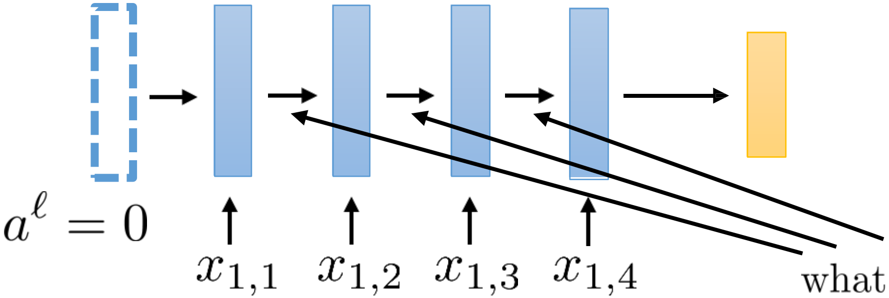
each layer:

$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix}$$

$$z^\ell = W^\ell \bar{a}^{\ell-1} + b^\ell$$

$$a^\ell = \sigma(z^\ell)$$

Sharing Weight Matrices



$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix} \qquad \begin{aligned} z^\ell &= W^\ell \bar{a}^{\ell-1} + b^\ell \\ a^\ell &= \sigma(z^\ell) \end{aligned}$$

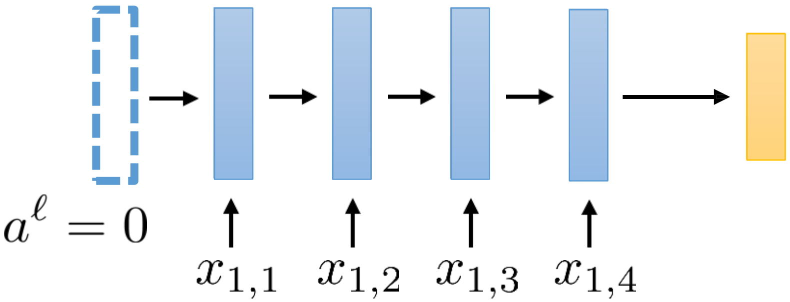
what if W^ℓ is *the same* for all these layers?

$$\text{i.e., } W^{\ell_i} = W^{\ell_j} \text{ for all } i, j$$
$$b^{\ell_i} = b^{\ell_j} \text{ for all } i, j$$

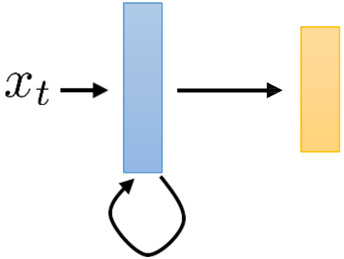
- We can have as many "layers" as we want!
- This is called a **recurrent** neural network (RNN).
- We could also call this a "variable-depth" network.

Recurrent Neural Networks

- What we just learned:



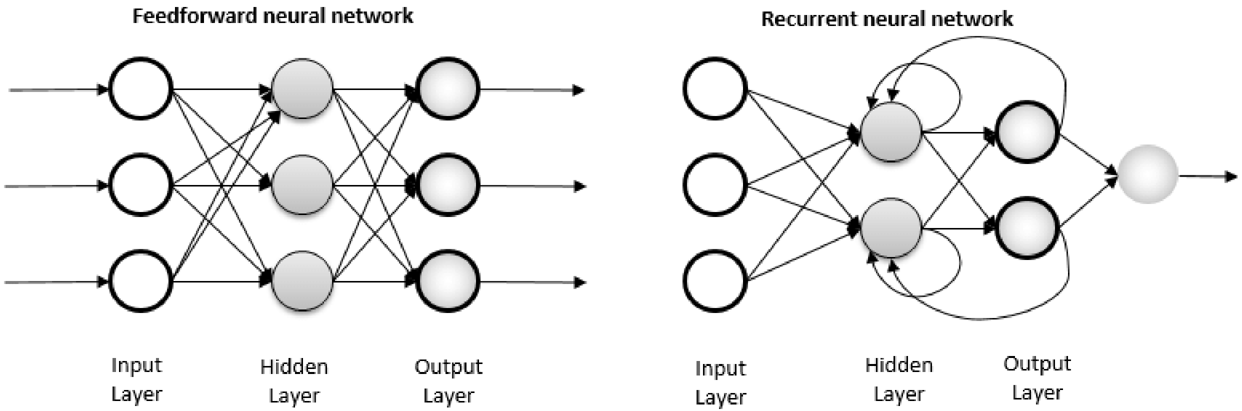
- What you often see in textbooks/classes:



- RNNs are just neural networks that share weights across multiple layers, take an input at each layer, and have a variable number of layers

Recurrent Neural Networks

- Recurrent neural networks (RNNs) are a class of artificial neural network commonly used for sequential data processing.
- RNNs can work on sequences of arbitrary lengths, rather than on fixed-sized inputs.

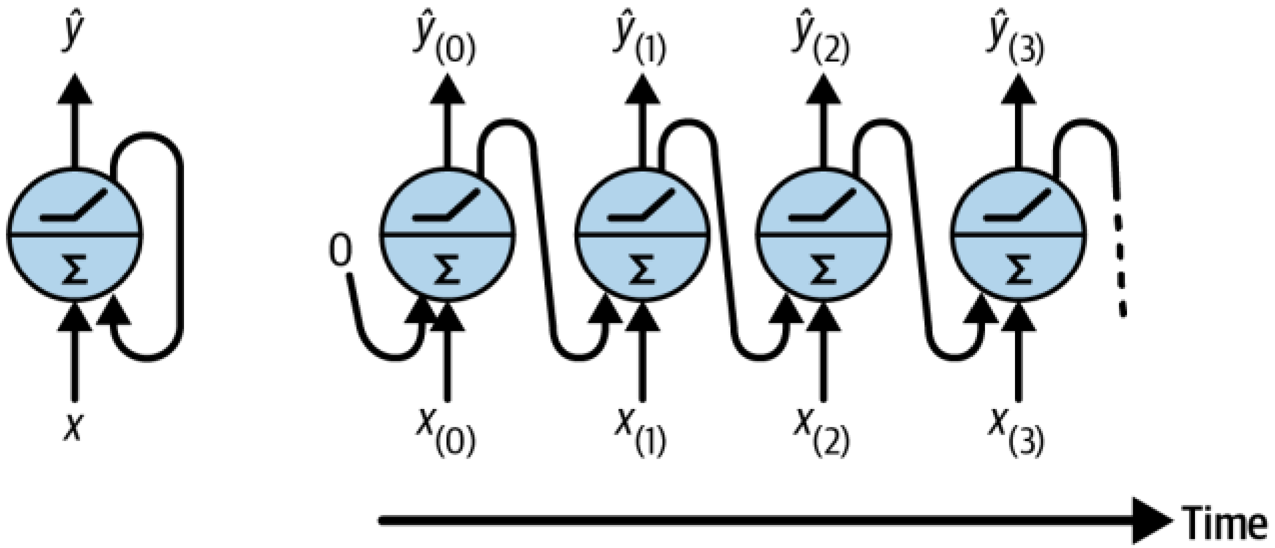


1.

Recurrent Neurons and Layers

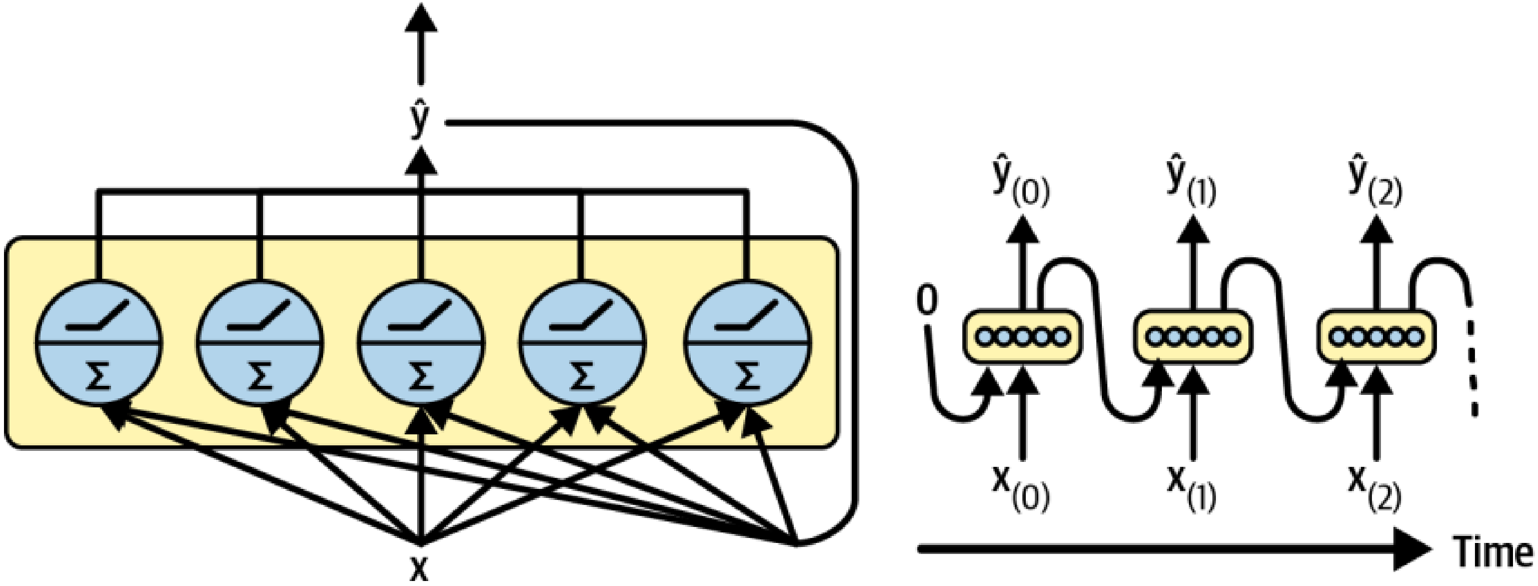
Recurrent Neuron

➢ At each *time step* t, the *recurrent neuron* receives the inputs $\mathrm{x}_{(t)}$ as well as its own output from the previous time step, $\hat{y}_{(t-1)}$.



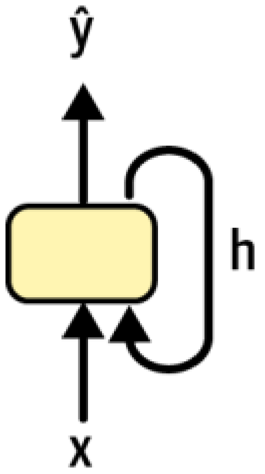
A layer of recurrent neurons

- Output of a recurrent layer: $\hat{\mathbf{y}}_{(t)} = \phi(\mathbf{W}_x^{\mathrm{T}}\mathbf{x}_{(t)} + \mathbf{W}_y^{\mathrm{T}}\hat{\mathbf{y}}_{(t-1)} + \mathbf{b})$



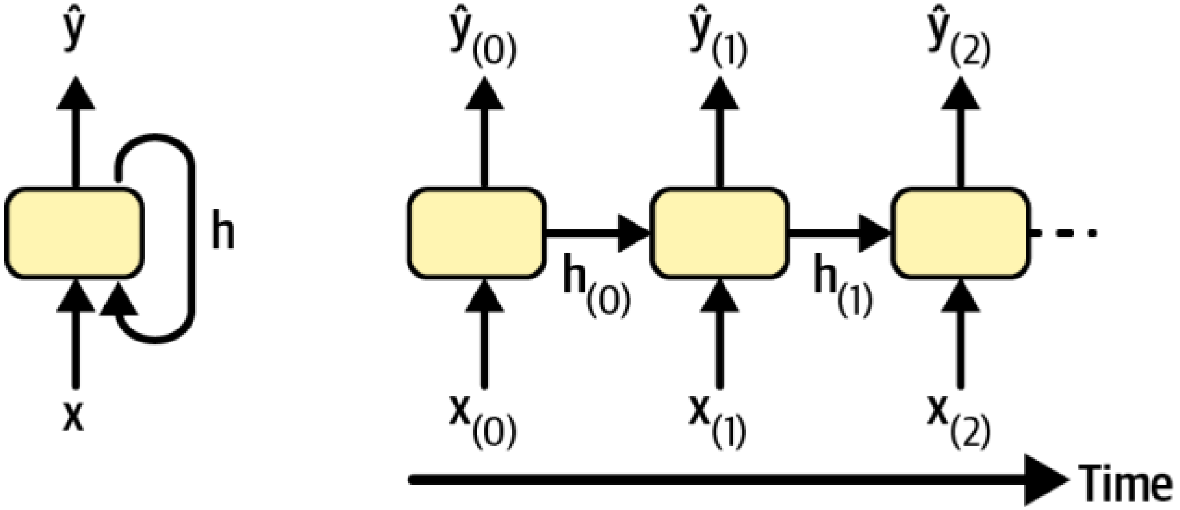
Memory Cell

- Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*.
- A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*).
- A layer of recurrent neurons, is a very basic cell, capable of learning only short patterns (typically about 10 steps long).
- A cell's state at time step t, denoted $\mathbf{h}_{(t)}$ is a function of inputs at that time step and its state at the previous time step: $\mathbf{h}_{(t)} = f(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$



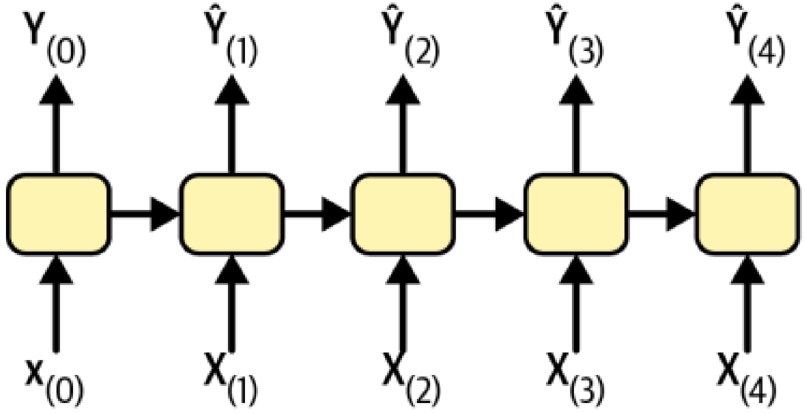
Hidden State of a Cell

- Its output at time step t, denoted $\hat{\mathbf{y}}_{(t)}$, is also a function of the previous state and the current inputs.



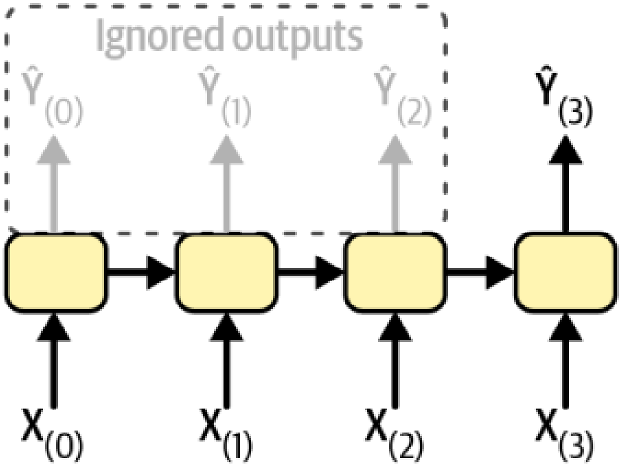
Sequence-to-Sequence Network

- A *sequence-to-sequence* RNN can simultaneously take a sequence of inputs and produce a sequence of outputs.
- Example: you feed it the data over the last N days, and you train it to output the series value shifted by one day into the future (i.e., from $N-1$ days ago to tomorrow).



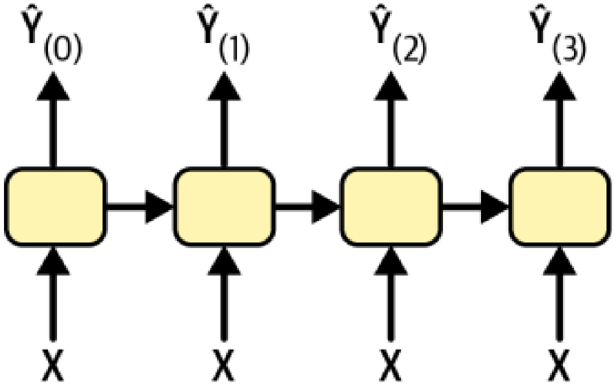
Sequence-to-Vector Network

- *Sequence-to-vector* network: you could feed the network a sequence of inputs and ignore all outputs except for the last one.
- Example: you feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score.



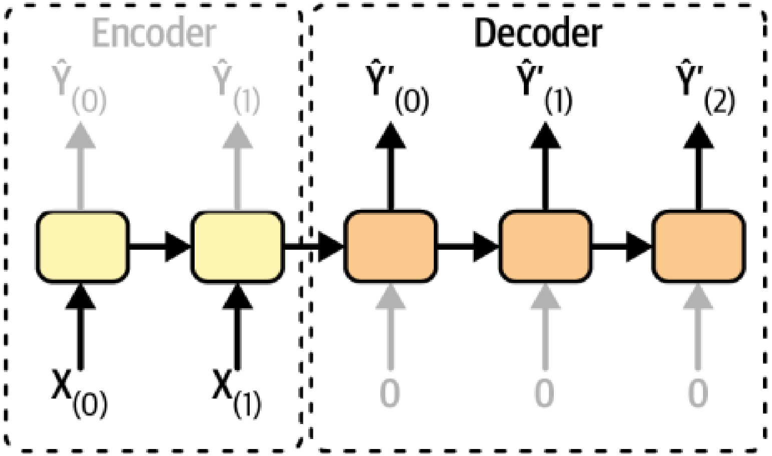
Vector-to-Sequence Network

- *Vector-to-sequence* network: you could feed the network the same input vector over and over again at each time step and let it output a sequence.
- Example: the input could be an image (or the output of a CNN), and the output could be a caption for that image.



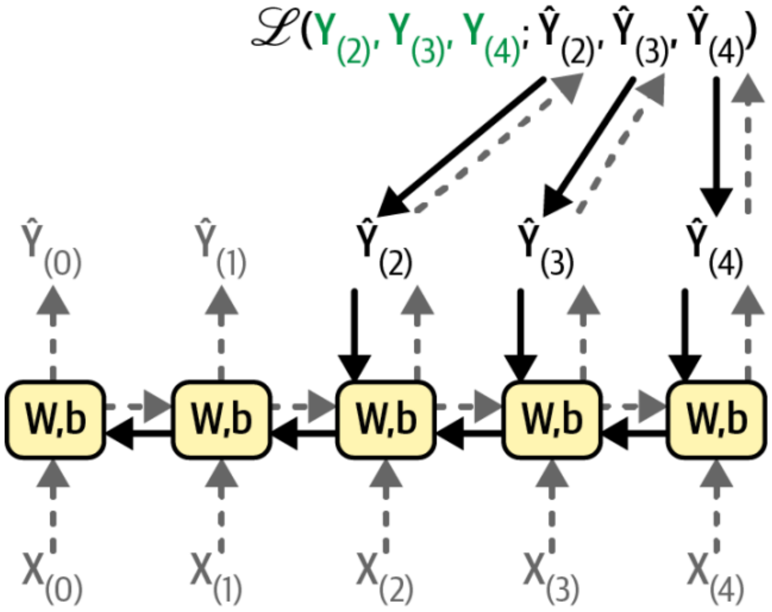
Encoder-Decoder Network

- You could have a sequence-to-vector network, called an *encoder*, followed by a vector-to-sequence network, called a *decoder*.
- Example*:* feed the network a sentence in English, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in French.



Training RNNs

➢ *Backpropagation through time* (BPTT): unroll the RNN through time and then use regular backpropagation.



2.

Time Series and ARMA Model Family

Chicago's Transit Authority Dataset

➢ Task: build a model to forecast the number of passengers that will ride on bus and rail the next day.

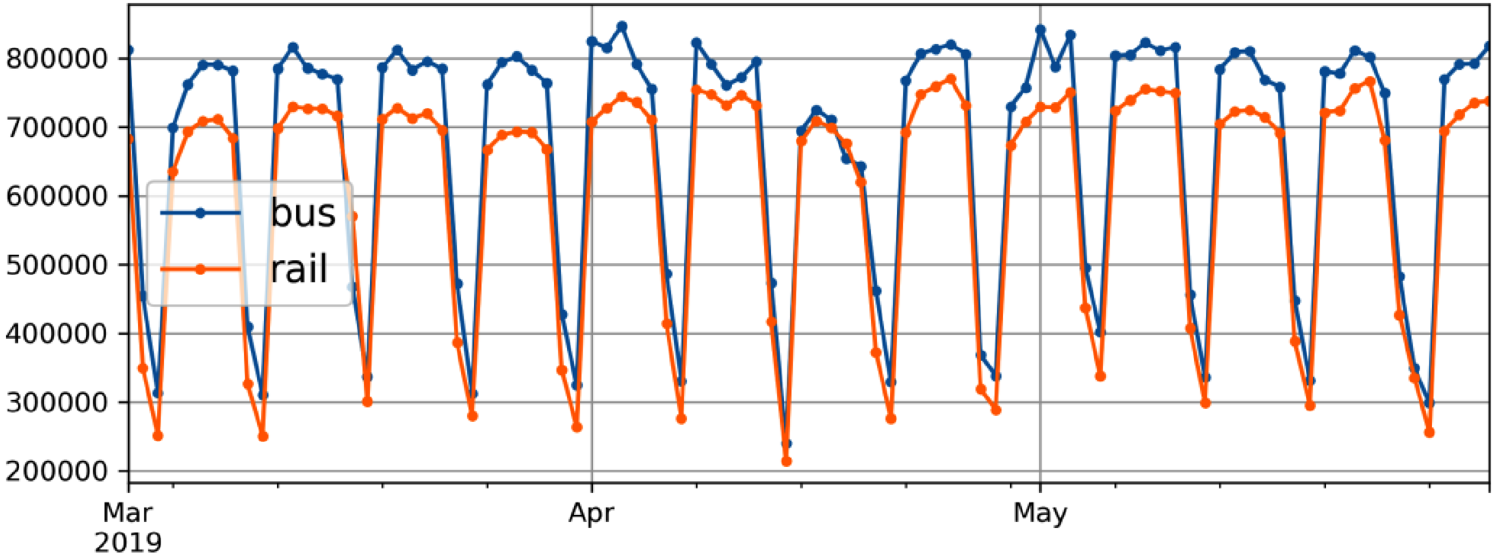
```python
import pandas as pd
from pathlib import Path

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)
```

```python
df.head()
```

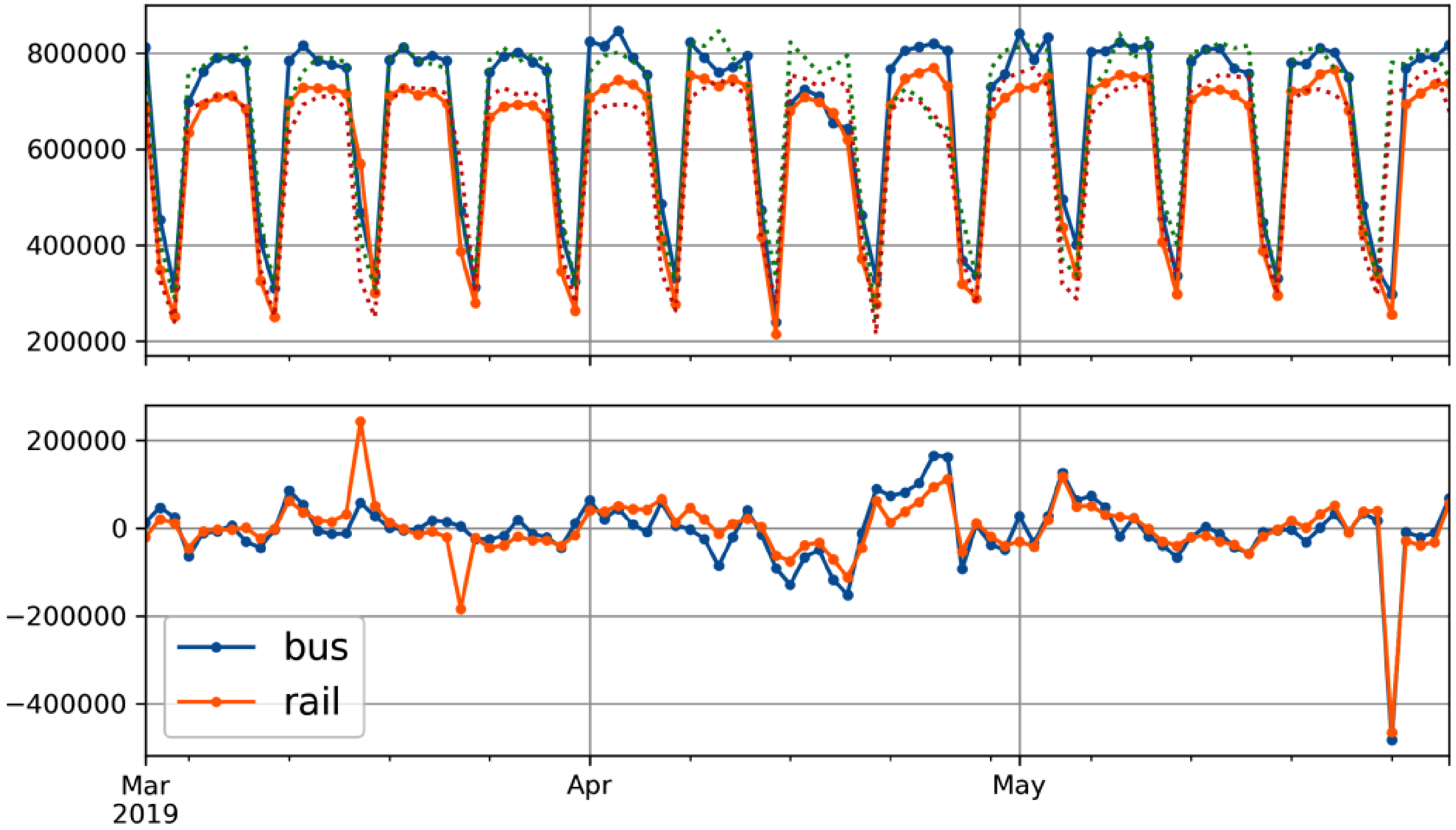
	day_type	bus	rail
date			
2001-01-01	U	297192	126455
2001-01-02	W	780827	501952
2001-01-03	W	824923	536432
2001-01-04	W	870021	550011
2001-01-05	W	890426	557917

Daily Ridership in Chicago

- Weekly *seasonality*: a similar pattern is clearly repeated every week.
- *Naive forecasting*: simply copying a past value to make our forecast.
  - It is often a great baseline.



Autocorrelated Time Series



Error of Naive Forecast

- Measure the mean absolute error (MAE):

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

```
diff_7.abs().mean()
```

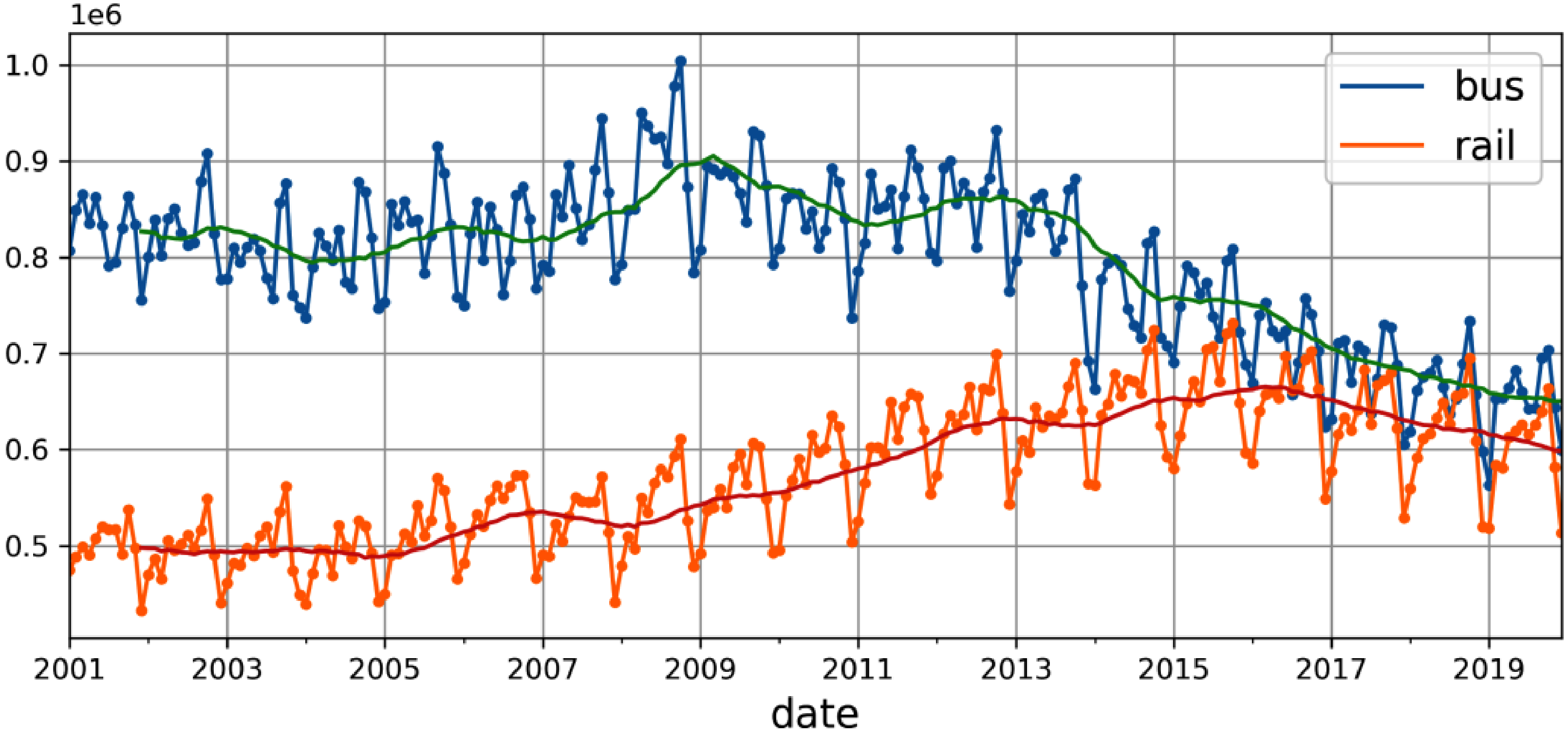
```
bus      43915.608696
rail     42143.271739
```

- Measure the *mean absolute percentage error* (MAPE):

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

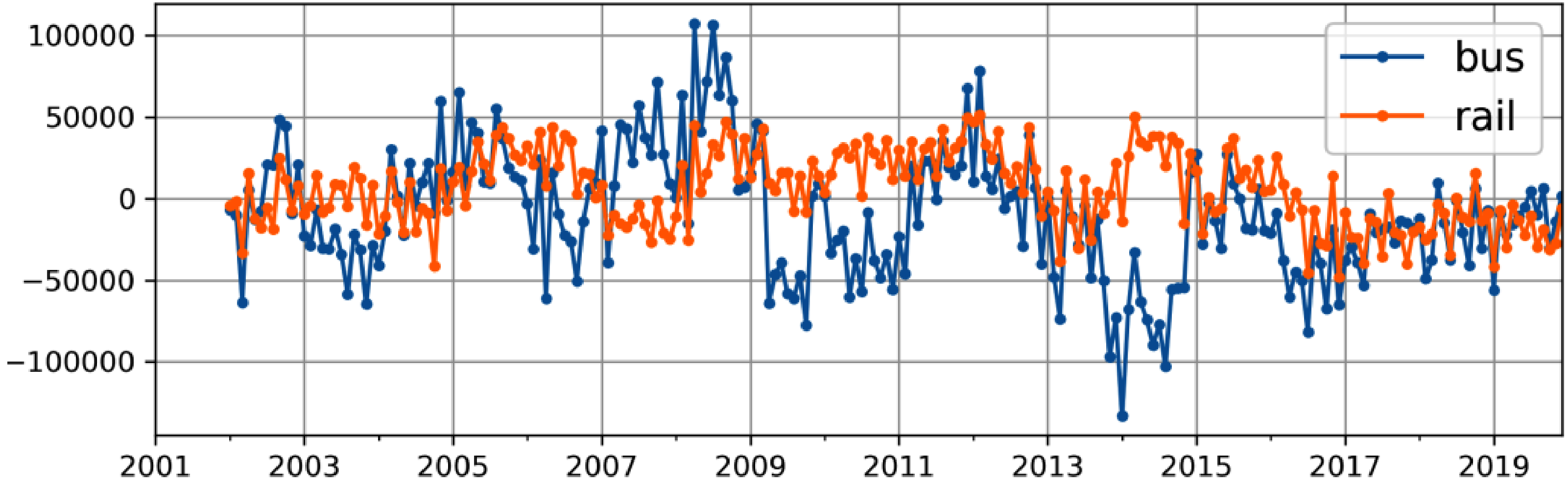
```
bus      0.082938
rail     0.089948
```

Yearly Seasonality



12-month Difference

- Differencing removes the yearly seasonality and long-term trends.
- For *stationary* time series, statistical properties remain constant over time, without any seasonality or trends.



The ARMA Model Family

- The *autoregressive moving average* (ARMA) model forecasts using a simple weighted sum of lagged values and corrects these forecasts by adding a moving average of the last few forecast errors:

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

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

- *Autoregressive* component: the weighted sum of the past p values of the time series, using the learned weights α_i.
- *Moving average* component: the weighted sum over the past q forecast errors $\epsilon_{(t)}$, using the learned weights θ_i.

Stationary Assumption

- The ARMA model assumes that the time series is stationary.
  - If it is not, then differencing may help.
- One round of differencing eliminates any linear trend:

$$[3, 5, 7, 9, 11] \xrightarrow{\text{differencing}} [2, 2, 2, 2]$$

- Two rounds of differencing eliminates quadratic trends:

$$[1, 4, 9, 16, 25, 36] \xrightarrow{\text{differencing}} [3, 5, 7, 9, 11] \xrightarrow{\text{differencing}} [2, 2, 2, 2]$$

- d consecutive rounds of differencing computes an approximation of the d^{th} order derivative of the time series, and eliminate polynomial trends up to degree d.

ARIMA and SARIMA Models

- The *autoregressive integrated moving average* (ARIMA) 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.
- The *seasonal ARIMA* (SARIMA) model: similar to ARIMA, but adds 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
  - P, D, and Q hyperparameters to model the seasonal pattern
    - 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.
  - s: the period of the seasonal pattern.

Forecasting Using `ARIMA` Class

- Assume today is the last day of May 2019, and we want to forecast the rail ridership for “tomorrow”, the 1st of June, 2019.
- Use `ARIMA` class from `statsmodels` library:

```python
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),  #(p, d, q)
              seasonal_order=(0, 1, 1, 7))  #(P, D, Q, s)
model = model.fit()
y_pred = model.forecast()  # returns 427,758.6
```

- The MAE for a 3-month period forecast using SARIMA is 32,041, which is significantly lower than the MAE we got with naive forecasting (42,143).