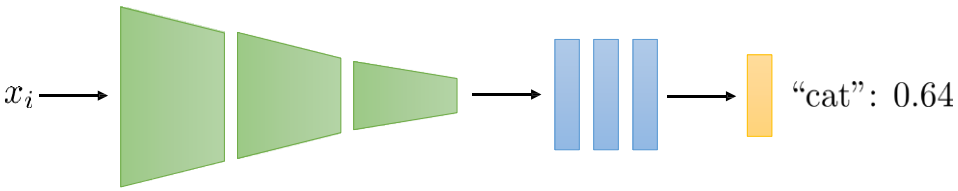
Hands-on Machine Learning

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?

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

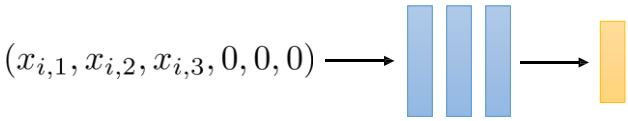
$x_2 = (x_{2,1}, x_{2,2}, x_{2,3})$

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

+ 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



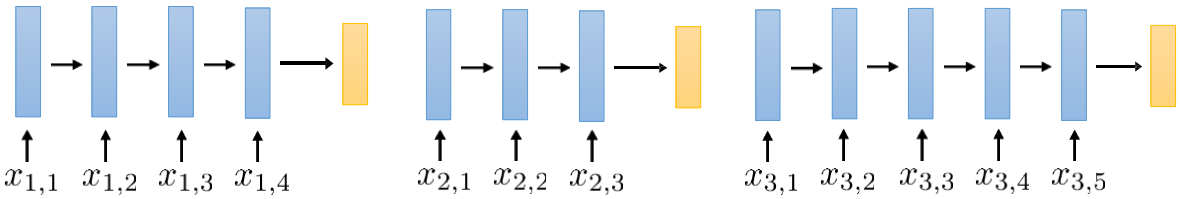
One Input per Layer

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

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

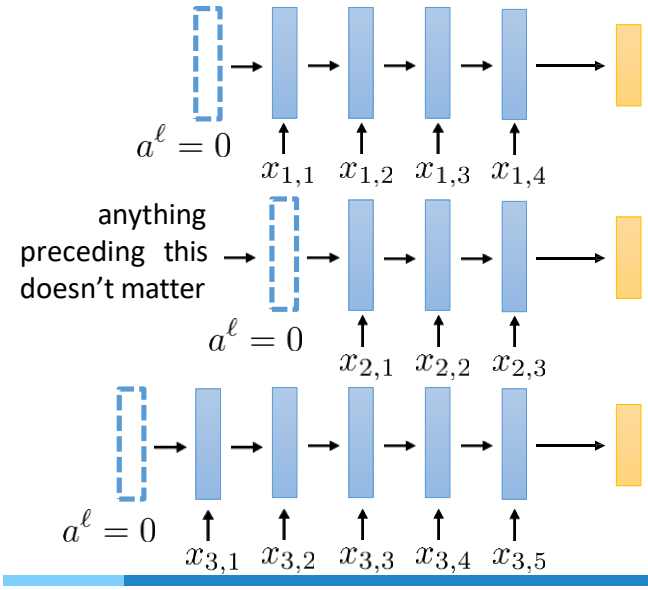
$x_3 = (x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, x_{3,5})$

what happens to the missing layers?



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

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:

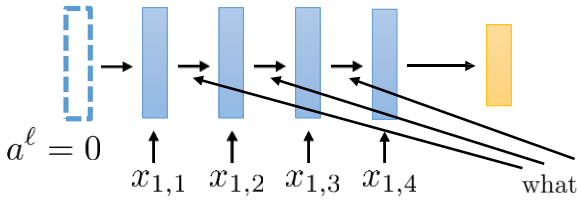
$$\bar{a}^{\ell-1} = \begin{bmatrix} a^{\ell-1} \\ x_{i,t} \end{bmatrix}$$

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

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

5

Sharing Weight Matrices



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

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

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

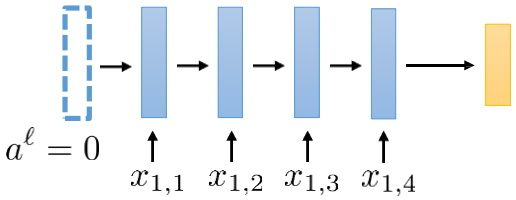
$b^{\ell_i} = b^{\ell_j}$ for all i, j

- 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.

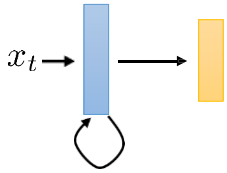
6

Recurrent Neural Networks

- What we just learned:



- What you often see in textbooks/classes:

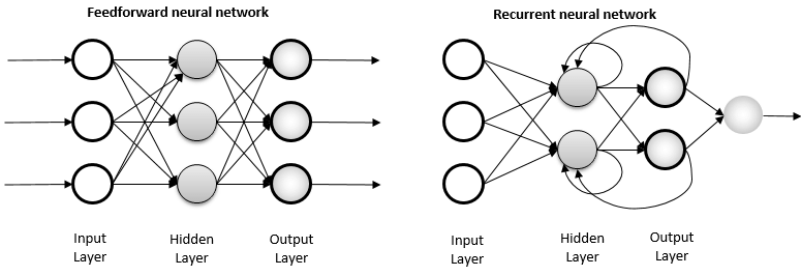


- RNNs are just neural networks that share weights across multiple layers, take an input at each layer, and have a variable number of layers

7

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.

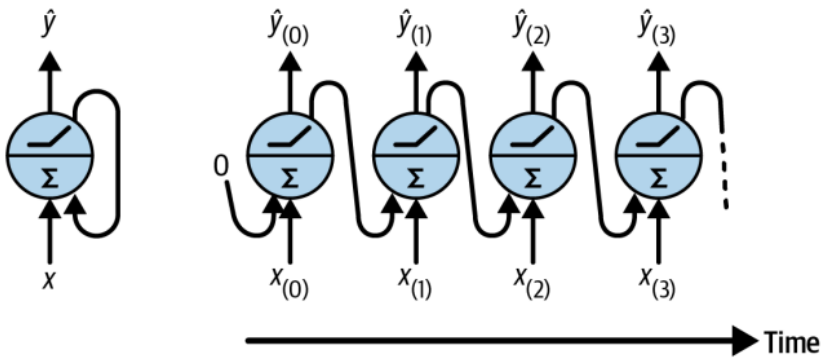


8

1. Recurrent Neurons and Layers

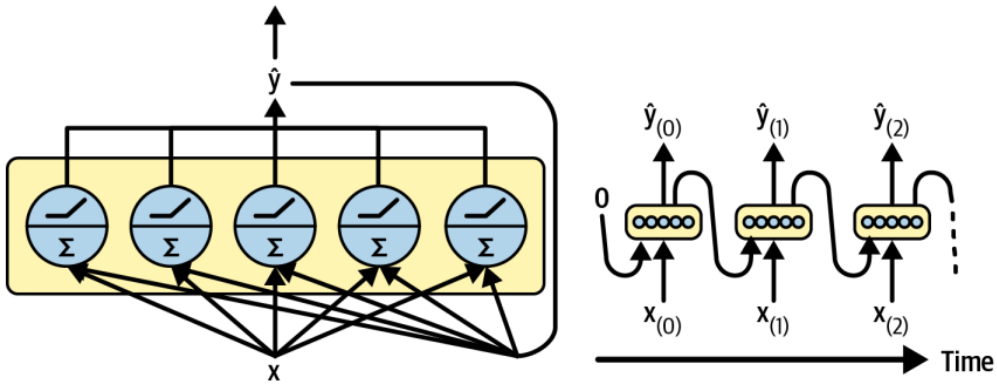
Recurrent Neuron

- 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

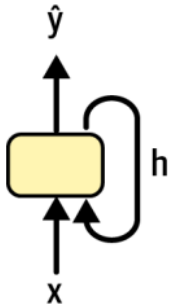
- Output of a recurrent layer: $\hat{\mathbf{y}}_{(t)} = \phi(\mathbf{W}_x^{\mathrm{T}}\mathbf{x}_{(t)} + \mathbf{W}_y^{\mathrm{T}}\hat{\mathbf{y}}_{(t-1)} + \mathbf{b})$



11

Memory Cell

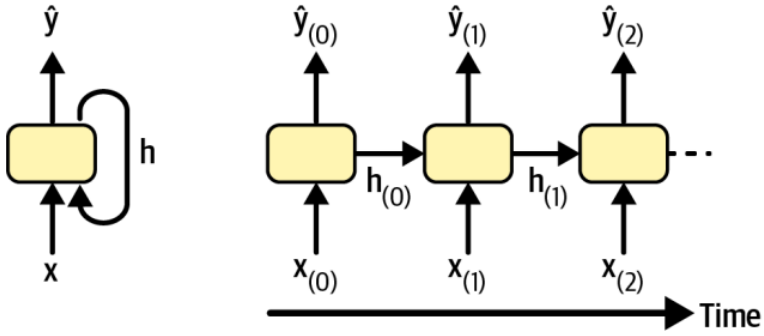
- Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of *memory*.
- A part of a neural network that preserves some state across time steps is called a *memory cell* (or simply a *cell*).
- A layer of recurrent neurons, 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: $\mathbf{h}_{(t)} = f(\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$



12

Hidden State of a Cell

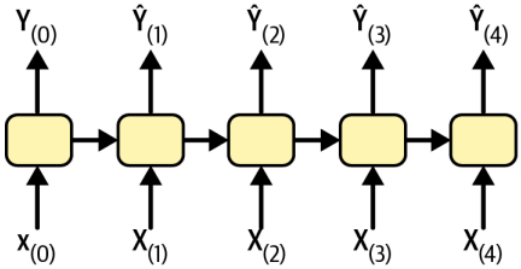
- Its output at time step t, denoted $\hat{\mathbf{y}}_{(t)}$, is also a function of the previous state and the current inputs.



13

Sequence-to-Sequence Network

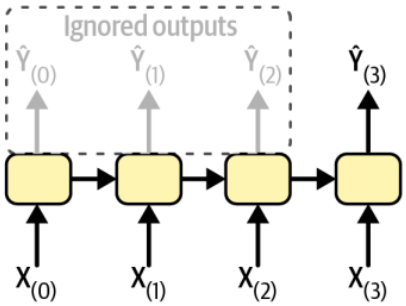
- A *sequence-to-sequence* RNN can simultaneously take a sequence of inputs and produce a sequence of outputs.
- Example: you feed it the data over the last N days, and you train it to output the series value shifted by one day into the future (i.e., from $N-1$ days ago to tomorrow).



14

Sequence-to-Vector Network

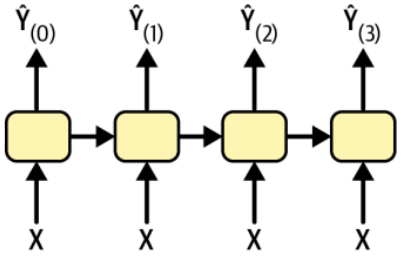
- *Sequence-to-vector* network: you could feed the network a sequence of inputs and ignore all outputs except for the last one.
- Example: you feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score.



15

Vector-to-Sequence Network

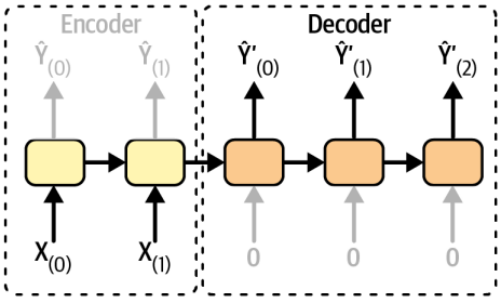
- *Vector-to-sequence* network: you could feed the network the same input vector over and over again at each time step and let it output a sequence.
- Example: the input could be an image (or the output of a CNN), and the output could be a caption for that image.



16

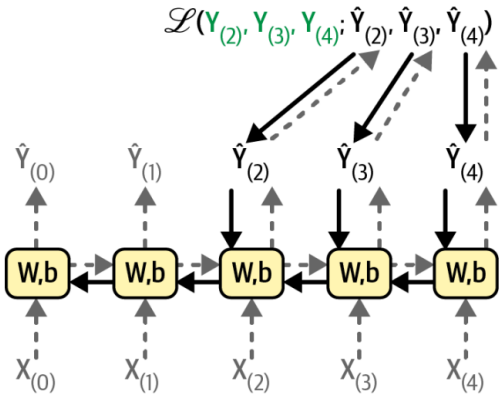
Encoder-Decoder Network

- 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.

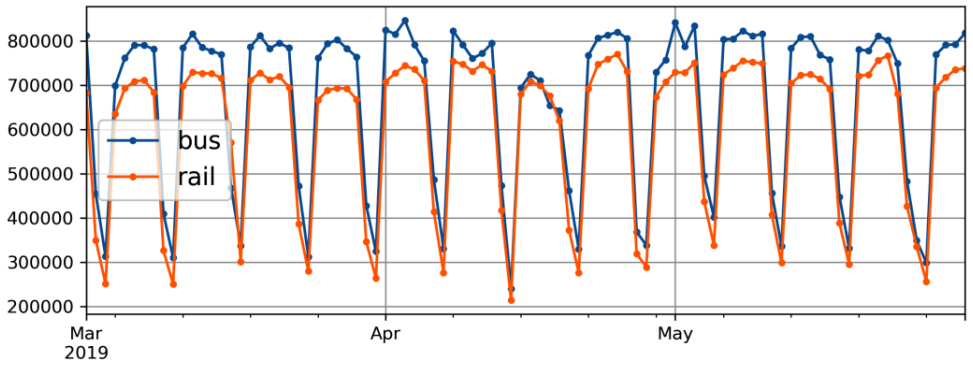
```
import pandas as pd
from pathlib import Path

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)
```

```
df.head()
```

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

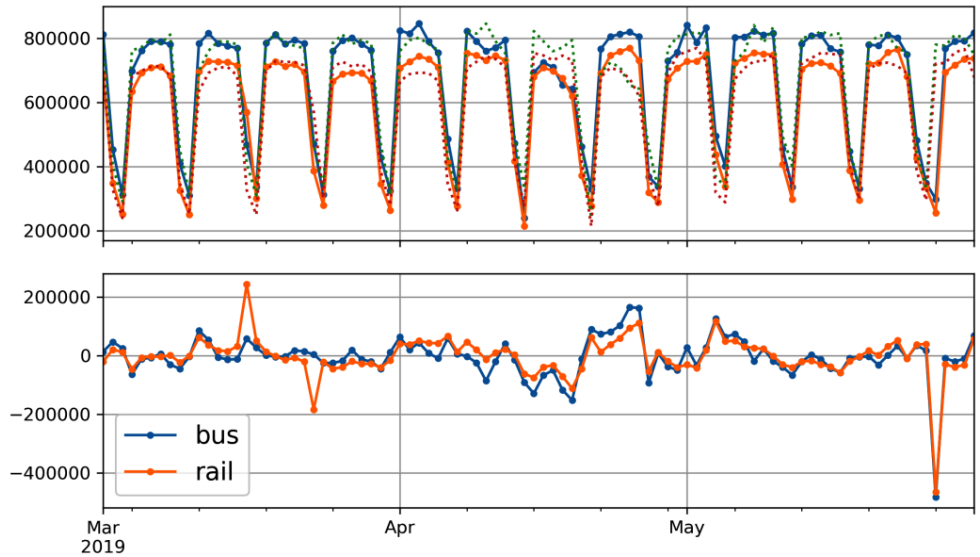
Daily Ridership in Chicago

- 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.



21

Autocorrelated Time Series



22

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()
```

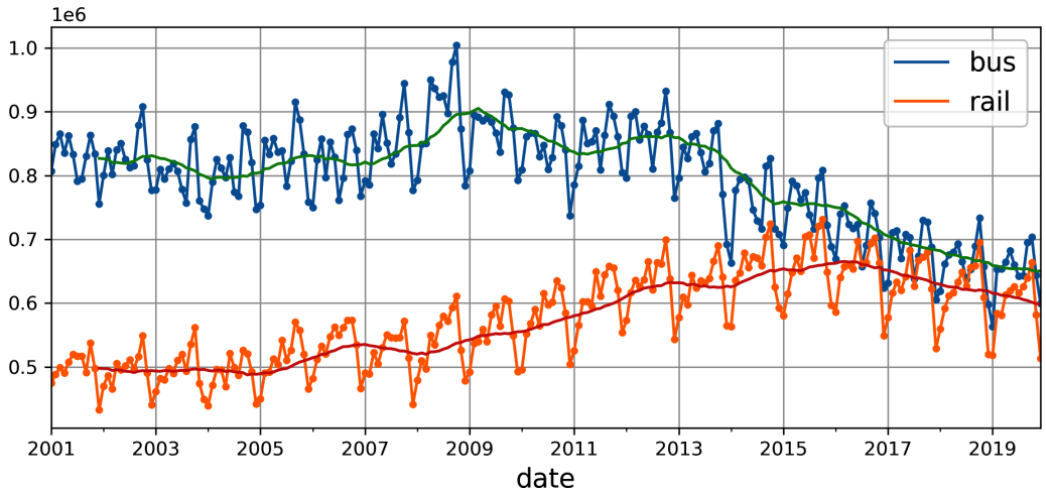
```
bus     43915.608696
rail    42143.271739
```

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

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

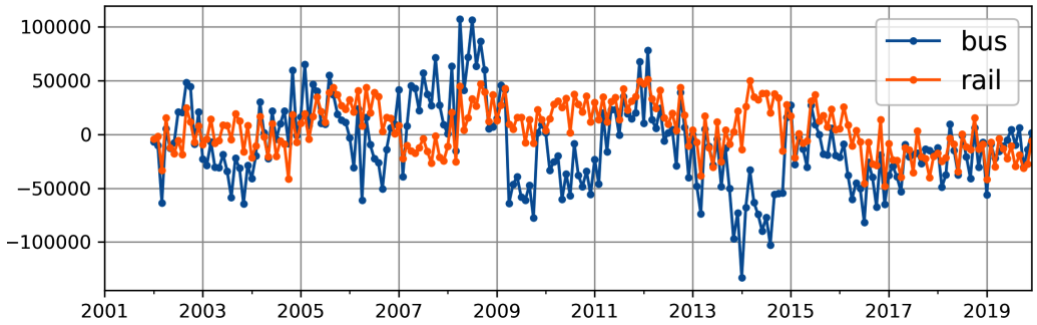
```
bus     0.082938
rail    0.089948
```


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.



25

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:

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

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

- *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.

26

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:

```
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).

3. Preparing the Data

Training Data

- **Goal**: forecast tomorrow's ridership based on the ridership of the past 8 weeks of data (56 days).
- The **inputs**: sequences 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 $(t+1)$.
- We use every 56-day window from the past as training data, and the target for each window will be the value immediately following it.
- `tf.keras.utils.timeseries_dataset_from_array()` is a utility function in Keras which takes a time series as input, and builds a `tf.data.Dataset` containing all the windows of the desired length, and their corresponding targets.

31

Preparing Training Data

```
import tensorflow as tf

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

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

32

Training/Validation/Test Split

- Scale down the data by a factor of one million, to ensure the values are near the 0–1 range, and split it into training period, a validation period, and a test period:

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

- When dealing with time series, you generally want to split across time.
  - In some cases you may be able to split along other dimensions, which will give you a longer time period to train on.
  - E.g., if you have data about the financial health of 10000 companies from 2001 to 2019, you might split this data across the different companies.

33

Preparing Data

```
seq_length = 56
train_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_train.to_numpy(),
    targets=rail_train[seq_length:],
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
valid_ds = tf.keras.utils.timeseries_dataset_from_array(
    rail_valid.to_numpy(),
    targets=rail_valid[seq_length:],
    sequence_length=seq_length,
    batch_size=32
)
```

34

4. Forecasting a Time Series

Forecasting Using a Linear Model

- We try a basic linear model with the Huber loss, which usually works better than minimizing the MAE directly. We also use early stopping.

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

- This model reaches a validation MAE, better than naive forecasting, but worse than the SARIMA model.

Forecasting Using a Simple RNN

- The most basic RNN, contains a single recurrent layer with just one recurrent neuron:

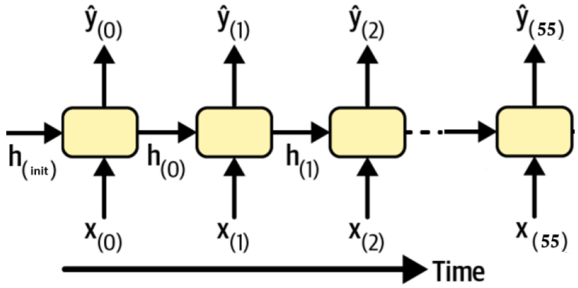
```
model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

- All recurrent layers in Keras expect 3D inputs of shape [*batch size*, *time steps*, *dimensionality*], where *dimensionality* is 1 for univariate time series and more for multivariate time series.
- The `input_shape` ignores the first dimension (i.e., the batch size).
- Recurrent layers can accept input sequences of any length, we can set the second dimension to `None`, which means "any size".

37

Forecasting Using a Simple RNN

- The activation function is `tanh` by default.
- Recurrent layers in Keras only return the final output. To make them return one output per time step, set `return_sequences=True`.
- We compile, train, and evaluate the model, and find that it's no good at all: its validation MAE is greater than 100,000!



38

What went wrong?

- The model only has a single recurrent neuron, so the only data it can use to make a prediction at each time step is the input value at the current time step and the output value from the previous time step.
  - The RNN's memory is extremely limited: it's just a single number, its previous output.
  - The model only has three parameters (two weights plus a bias term).

- The time series contains values from 0 to about 1.4, but since the default activation function is `tanh`, the recurrent layer can only output values between –1 and +1.
  - There's no way it can predict values between 1.0 and 1.4.

A Larger RNN Layer

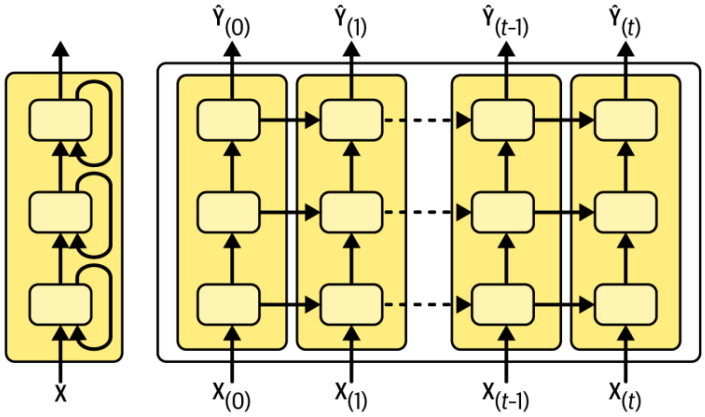
- Create a larger recurrent layer, with 32 recurrent neurons.
- A dense output layer to project the final output from 32 dimensions down to 1.

```
univar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 1]),
    tf.keras.layers.Dense(1)  # no activation function by default
])
```

- Its validation MAE reaches 27,703, better than the SARIMA model.
- We only normalized the time series, without removing trend and seasonality.
  - To get the best performance, you may want to try making the time series more stationary; e.g. using differencing.

Forecasting Using a Deep RNN

- To implement a deep RNN, stack `SimpleRNN` layers.

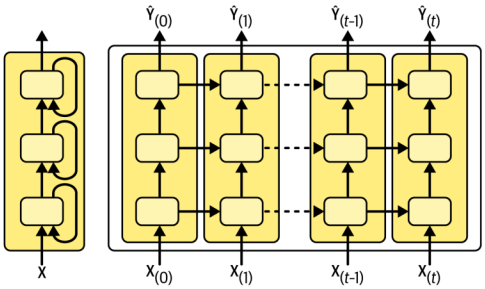


41

Forecasting Using a Deep RNN

```
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
])
```

- If you train and evaluate this model, you will find that it reaches an MAE of about 31,211.
  - This RNN is too large for our task.



42

Forecasting Multivariate Time Series

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

date	bus	rail	next_day_type_A	next_day_type_U	next_day_type_W
2001-01-01	0.297192	0.126455	False	False	True
2001-01-02	0.780827	0.501952	False	False	True
2001-01-03	0.824923	0.536432	False	False	True
2001-01-04	0.870021	0.550011	False	False	True
2001-01-05	0.890426	0.557917	True	False	False

43

Train/Validation/Test Split

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

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

44

Create the Model

```
mulvar_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(1)
])
```

- It reaches a validation MAE of 22,062.
- Using a single model for multiple related tasks often results in better performance than using a separate model for each task.
  - Features learned for one task may be useful for the other tasks.
  - Having to perform well across multiple tasks prevents the model from overfitting (it's a form of regularization).

5. Handling Long Sequences

The Unstable Gradients Problem in RNNs

- Many of the previous tricks can also be used for RNNs: good parameter initialization, faster optimizers, dropout, …
- Non-saturating activation functions (e.g., ReLU) may not help. Why?
  - 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 non-saturating 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.

47

The Unstable Gradients Problem in RNNs

- Batch normalization cannot be used as efficiently with RNNs as with deep feedforward nets.
  - You cannot use it between time steps, only between recurrent layers.
  - You can apply BN between layers by adding a `BatchNormalization` layer before each recurrent layer, but it will slow down training, and it may not help much.
- Another form of normalization often works better with RNNs is *layer normalization*.
  - It is very similar to batch normalization, but instead of normalizing across the batch dimension, layer normalization normalizes across the features dimension.

48

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 first inputs.
- To tackle this problem, the *long short-term memory* (LSTM) cell was proposed in 1997.
- The LSTM cell can be used very much like a basic cell, except it will perform much better; training will converge faster, and it will detect longer-term patterns in the data:

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```


LSTM Cell

$\mathbf{h}_{(t)}$: short-term state

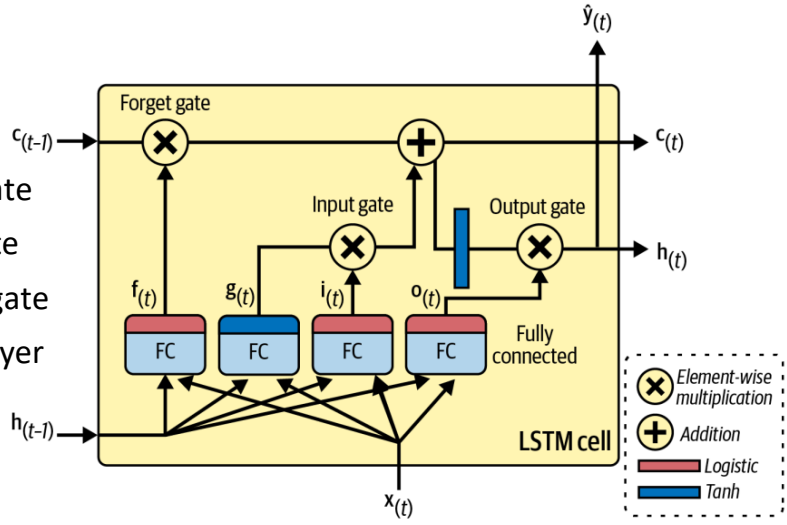
$\mathbf{c}_{(t)}$: long-term state

$\mathbf{f}_{(t)}$: controller of forget gate

$\mathbf{i}_{(t)}$: controller of input gate

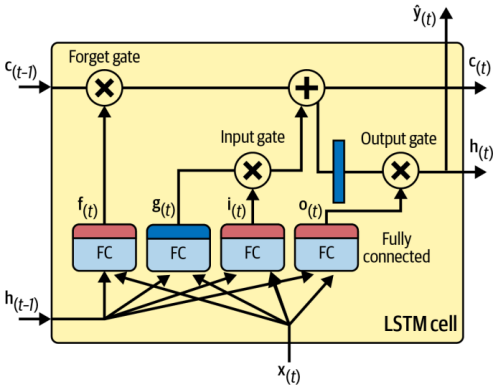
$\mathbf{o}_{(t)}$: controller of output gate

$\mathbf{g}_{(t)}$: output of the main layer



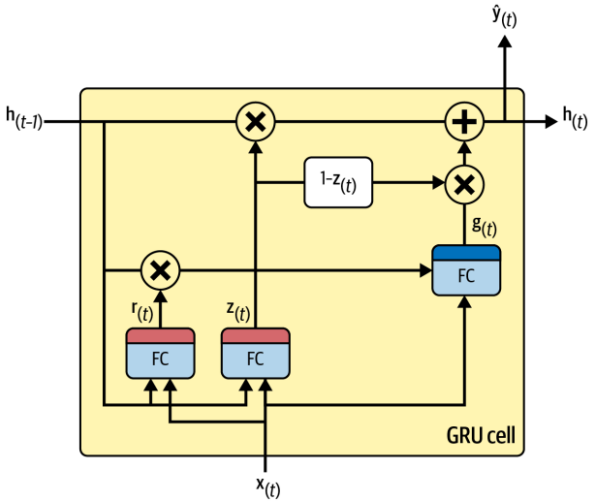
Where new memories come from?

- The main FC layer has the usual role of analyzing the current inputs $\mathbf{x}_{(t)}$ and the previous short-term state $\mathbf{h}_{(t-1)}$.
- The 3 other layers are *gate controllers*:
  - *forget gate* controls which parts of the long-term state should be erased.
  - *input gate* controls which parts of $\mathbf{g}_{(t)}$. should be added to the long-term state.
  - *output gate* controls which parts of the long-term state should be read and output at this time step.



GRU Cell

- The *gated recurrent unit* (GRU) cell is a simplified version of the LSTM cell:
  - 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.
  - There is no output gate.




```
gru_model = tf.keras.Sequential([
    tf.keras.layers.GRU(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```