Processing Sequences Using RNNs and CNNs

Deep Learning
Stijn Lievens and Sabine Devreese
2023-2024

Processing Sequences Using RNNs and CNNs

Recurrent Neural Networks

- **Recurrent neural networks (RNNs)** can be used to process sequences of arbitrary lengths.
  - In contrast, **feedforward neural networks** process fixed-size inputs.
- Examples of sequences:
  - time series data
  - sentences
  - audio samples
  - ....

Two Main Problems with RNNs

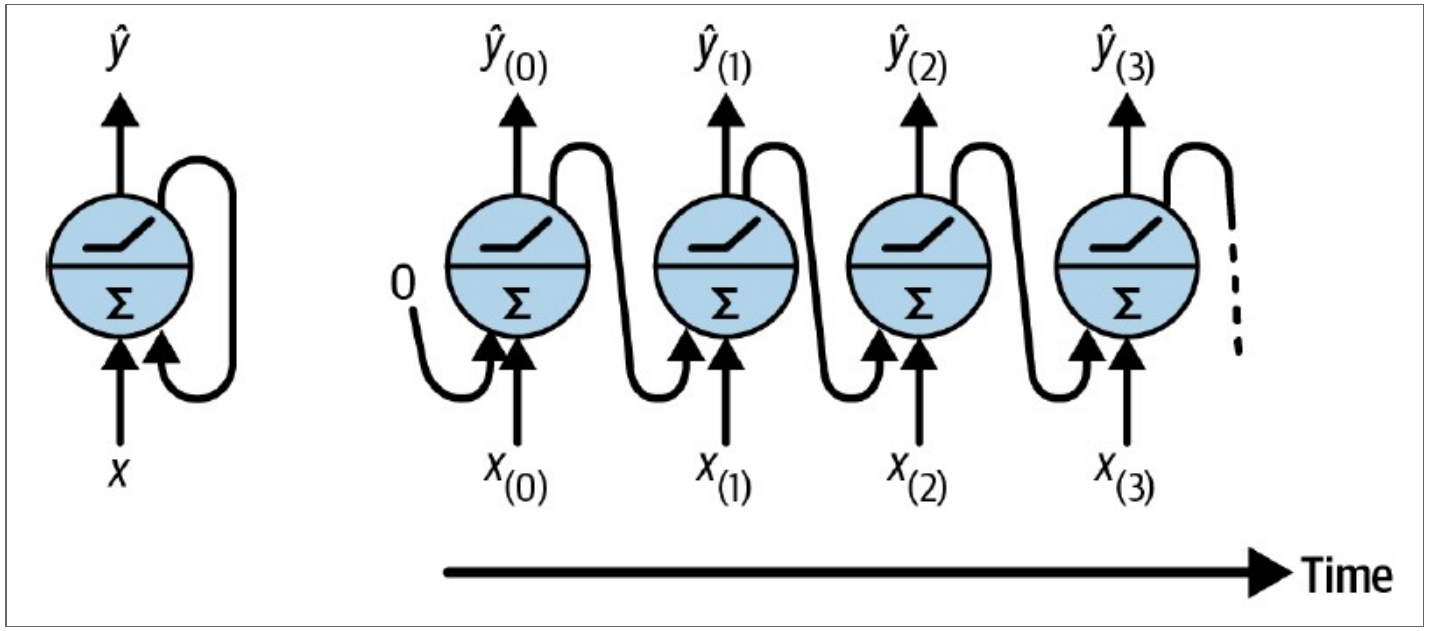
- *Unstable gradients*, which can be alleviated using **recurrent dropout** and **recurrent layer normalization**.
- *Limited short-term memory*, LSTM and GRU cells can help.

15.1 Recurrent Neurons and Layers

Simplest RNN

- At each time step t, a **recurrent neuron** receives both the input $\mathbf{x}_{(t)}$ and the output from the previous time step, $\hat{y}_{(t-1)}$.
  - At the first time step, there is no previous output, so it is typically set to zero.
- Such a network can be *unrolled through time*. See figure on the next slide.

Unrolling Through Time



Unrolling Through Time

Layer of Recurrent Neurons

- A layer of recurrent neurons is called a **recurrent layer**.
- At each time step t, every neuron receives both the input vector $\mathbf{x}_{(t)}$ and the output vector from the previous time step, $\hat{\mathbf{y}}_{(t-1)}$.
- Each recurrent neuron has *two sets of weights*:
  - One for the inputs $\mathbf{x}_{(t)}$.
  - One for the outputs of the previous time step, $\hat{\mathbf{y}}_{(t-1)}$.

Computation of a Recurrent Layer

- We can put all the weights in two matrices,
  - $\mathbf{W_x} \in \mathbb{R}^{n_{\text{inputs}} \times n_{\text{neurons}}}$ and $\mathbf{W}_{\hat{\mathbf{y}}} \in \mathbb{R}^{n_{\text{neurons}} \times n_{\text{neurons}}}$.
- Let $\mathbf{X}_{(t)} \in \mathbb{R}^{m \times n_{\text{inputs}}}$ be the input batch at time step t.
- Let $\hat{\mathbf{Y}}_{(t)} \in \mathbb{R}^{m \times n_{\text{neurons}}}$ be the output batch at time t.
- Let $\mathbf{b}$ be a vector of size n_{neurons} containing the biases of the neurons.
- Let ϕ be the activation function.
- The computation done by the recurrent layer is

$$\hat{\mathbf{Y}}_{(t)} = \phi\left(\mathbf{X}_{(t)}\mathbf{W_x} + \hat{\mathbf{Y}}_{(t-1)}\mathbf{W}_{\hat{\mathbf{y}}} + \mathbf{b}\right)$$

$\hat{\mathbf{Y}}_{(t)}$ Depends on ....

Notice how

- $\hat{\mathbf{Y}}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\hat{\mathbf{Y}}_{(t-1)}$.
- $\hat{\mathbf{Y}}_{(t-1)}$ is a function of $\mathbf{X}_{(t-1)}$ and $\hat{\mathbf{Y}}_{(t-2)}$
- $\hat{\mathbf{Y}}_{(t-2)}$ is a function of $\mathbf{X}_{(t-2)}$ and $\hat{\mathbf{Y}}_{(t-3)}$
- Etc.

In other words, $\hat{\mathbf{Y}}_{(t)}$ is a function of all the inputs from time step 0 to t: $\mathbf{X}_{(0)}$, $\mathbf{X}_{(1)}$, ..., $\mathbf{X}_{(t)}$.

Pseudocode RNN (Extra)

The pseudocode for a (simple) RNN is:

```
Y_t = 0 # initial "output"
for input_t in input_sequence:
    Y_t = activation(dot(input_t, W_x) + dot(Y_t, W_y) + b)
```

15.1.1 Memory Cells

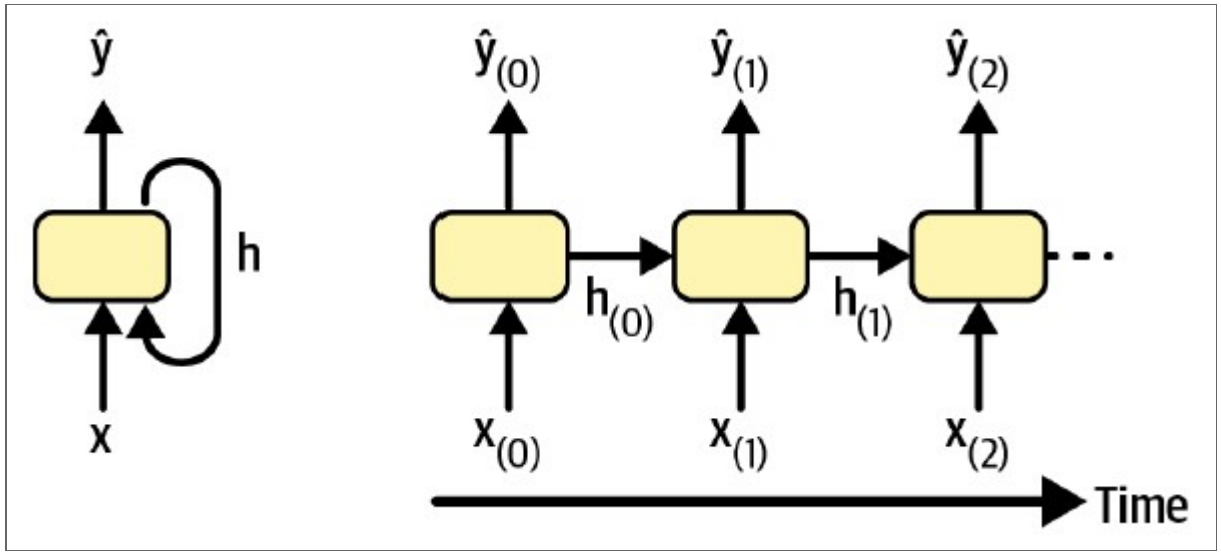
Memory Cells

- Since the output of a recurrent neuron depends on all inputs from the previous steps, it has some kind of *memory*.
  - The single recurrent neuron is a very basic memory cell. It can only learn short patterns.
- A cell may have a (hidden) state $\mathbf{h}_t$. In general

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1})$$

  and the output $\mathbf{y}_t$ is a function of $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$.
- Note: in the cells seen so far $\mathbf{h}_t = \mathbf{y}_t$, but this need not be the case.

Output may Differ from State



Output may Differ from State

15.1.2 Input and Output Sequences

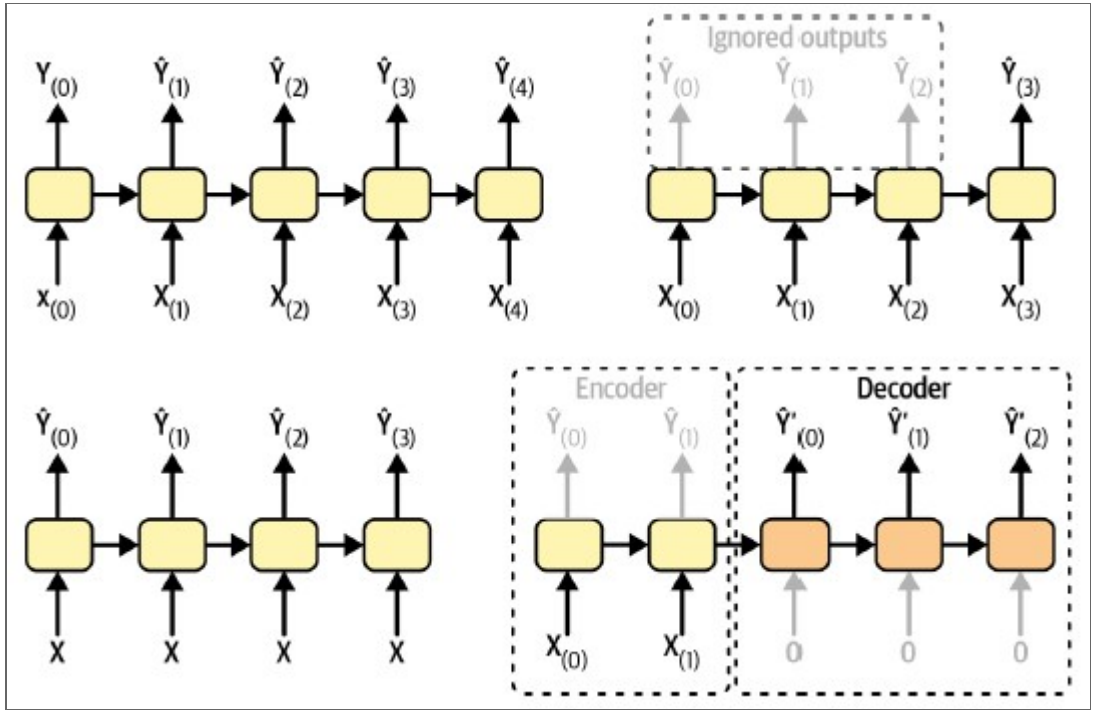
Input and Output Sequences

- A recurrent neural network can simultaneously take a sequence of inputs and produce a sequence of outputs. This is called a **sequence-to-sequence network**.
- If you ignore all the outputs except the last one you get a **sequence-to-vector network**.
  - E.g. sentiment analysis of a movie review.

Input and Output Sequences

- You can also feed the network the same input over and over again at each time step and let it produce a sequence. This is a **vector-to-sequence network**.
  - E.g. image captioning.
- Finally, you can have a sequence-to-vector network (the **encoder**) followed by a vector-to-sequence network (the **decoder**). This is called an **encoder-decoder network**.
  - E.g. machine translation.

Different Modes of Use for RNNs



Different Modes of Use for RNNs

15.2 Training RNNs

Training an RNN

- Training a RNN is done by unrolling the network, and then applying the (traditional) backpropagation algorithm.
  - This is called **backpropagation through time**.
- Since the same weights are used at each time step, they are updated multiple times.
  - The automatic difference engine of Tensorflow takes care of this.

15.3 Forecasting a Time Series

Use Case: Forecasting Passenger Numbers

- We use data from Chicago's Transit Authority, to predict the number of passengers that will ride on bus and rail the next day.

```
import pandas as pd
from pathlib import Path

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

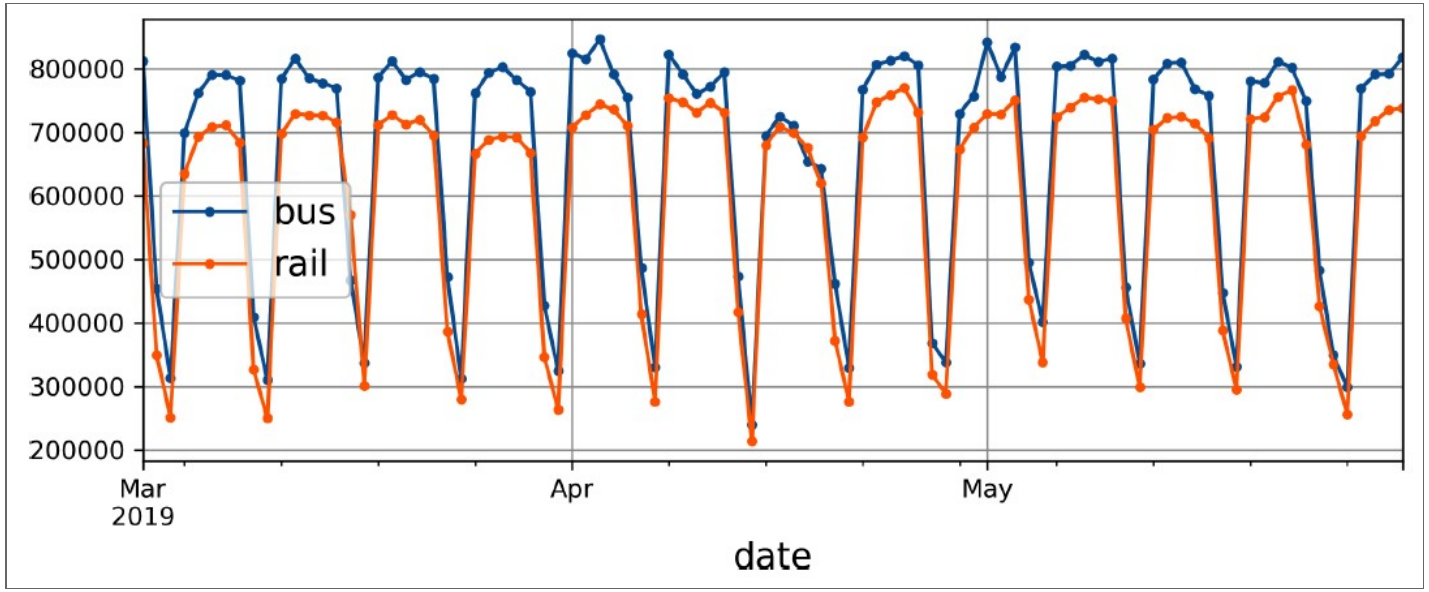
The Dataset

```
df.head()
```

yields

```
    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 March-May 2019



Daily Ridership in March-May 2019

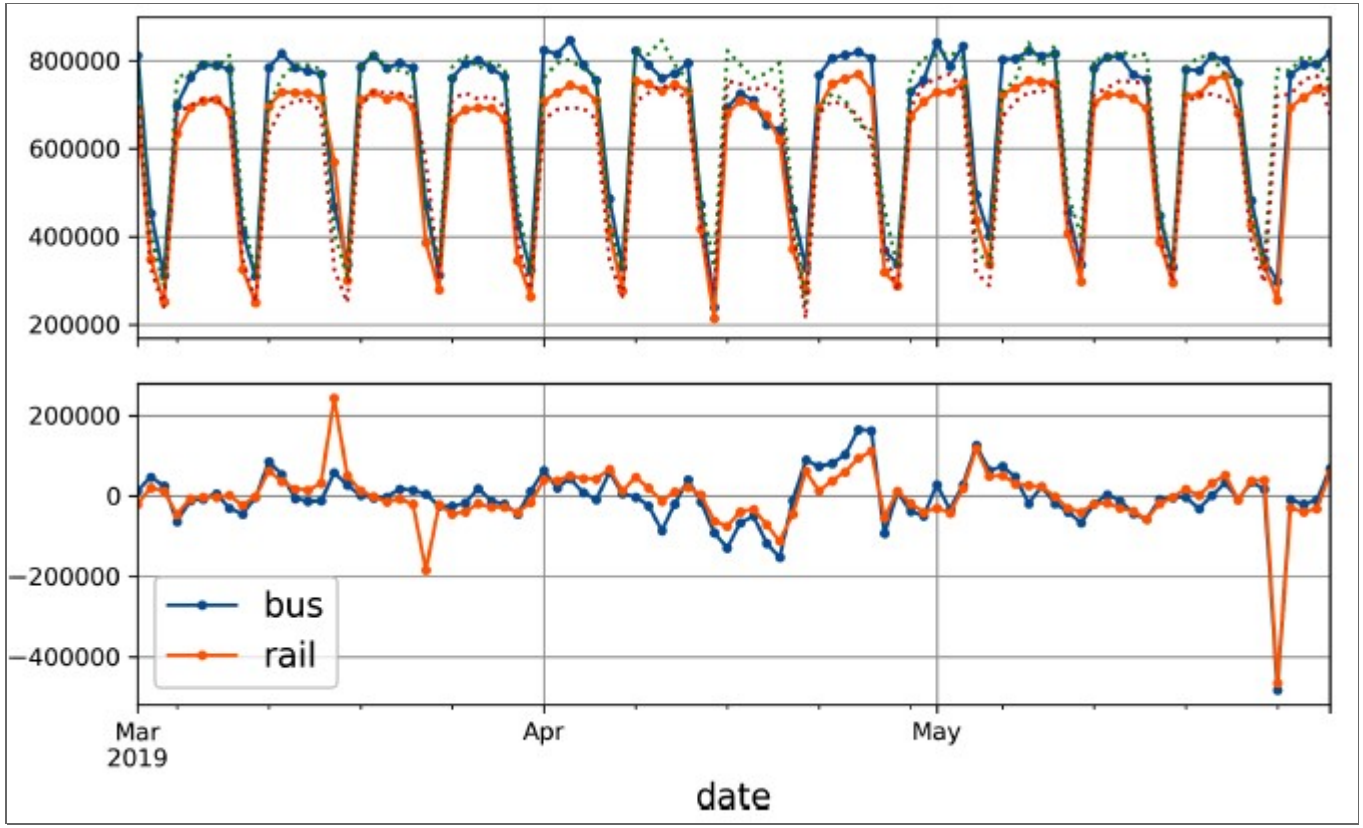
Time Series Data

- **Time series** data contains data with values at different time steps.
  - Usually, the time steps are equally spaced.
- A **univariate** time series contains a single value at each time step.
- A **multivariate** time series contains several values at each time step.
  - The ridership data is multivariate.

Time Series Forecasting

- Predicting future values of a time series is called **time series forecasting** and is the most typical task when dealing with time series.
- The ridership data shows a weekly **seasonality**.
  - A **naive forecast** can just copy the value from the previous week to make the forecast.

Visualizing the Naive Forecast



Visualizing the naive forecast and the difference

Code to Create Previous Plot

```
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
# lagged time series (shifted by 7 days)
df.shift(7).plot(ax=axs[0], grid=True, legend=False, linestyle=":")
# Plot 7-day difference time series
diff_7.plot(ax=axs[1], grid=True, marker=".")
axs[0].set_ylim([170_000, 900_000])  # extra code - beautifies the plot
plt.show()
```

Error Measures for Naive Forecast

- The **mean absolute error** (MAE) is the average absolute difference between the forecasts and the targets.
  - It's hard to tell whether this is good or bad.
- The **mean absolute percentage error** (MAPE) is the average absolute difference between the forecasts and the targets, divided by the absolute value of the targets, expressed as a percentage.
  - This is easier to interpret.

Python Code for Error Measures

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

yields

```
bus     43915.608696
rail    42143.271739
dtype: float64
```

while

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

yields

```
bus     0.082938
rail    0.089948
```

```
dtype: float64
```

Error Measures for Naive Forecast

- Note how the MAE for “bus” (43915) is higher than the MAE for “rail” (42143).
- This MAE means that we are off by about 43915 passengers on average for “bus”.
- The MAPE for “bus” is approximately 8.3%, which means that we are off by about 8.3% on average for “bus”, while we are off by about 9.0% on average for “rail”.
  - The MAPE thus shows that the naive forecast is better for “bus” than for “rail” (even though the MAE suggests otherwise).

15.3.1 The ARMA Model Family

ARMA Model Family

- ARMA stands for **autoregressive moving average**.
- An ARMA model predicts the future values as
  - an autoregressive linear combination of the p previous values
  - and a linear combination of the past q forecast errors.

  We don't go into the details of ARMA models, but they are a popular model family for time series forecasting.

15.3.2 Preparing the Data for Machine Learning Models

Goal of the Forecasting Model

- We try to build a model to predict tomorrow's ridership, based on the ridership of the previous 56 days.
- The inputs to the model will be sequences each containing 56 values for the time steps $t - 55$ up to t.
- For each input sequence, the output will be a single value, the predicted ridership at time step $t + 1$.

Training Data

- The training data will be every 56-day window from the past, and the target for each window is the value immediately after the window.
- Keras has a utility function to help us: `tf.keras.utils.timeseries_dataset_from_array`.
  - It takes a time series as input and produces a `tf.data.Dataset` object (see Chapter 13).
  - The `Dataset` will contain windows of the desired length, and the targets for each window.

timeseries_dataset_from_array

```
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)
```

gives

```
[(<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)>)]
```

Explanation

- `[0,1,2,3,4,5]` contains the following windows of length 3 and their targets:
  - `[0,1,2]` with target `3`
  - `[1,2,3]` with target `4`
  - `[2,3,4]` with target `5`
- The number of windows is not a multiple of the batch size, so the last batch is smaller.

The `window` Method of `Dataset`

- We can achieve the same with the `window` method of `Dataset`.
  - This method is more complex, but it is more flexible.
- `window` returns a dataset of window datasets (much like a list of lists).

The `window` Method: Example

```
for window_dataset in tf.data.Dataset.range(6).window(4, shift=1):
    for element in window_dataset:
        print(f"{element}", end=" ")
    print()
```

yields windows of length 4, each window shifted by 1:

```
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5
4 5
5
```

- Use `drop_remainder=True` to drop the windows that are too short.

The `flat_map` Method

- The `flat_map` method can be used to flatten a dataset of datasets.
- `flat_map` takes a function as argument that allows to transform each dataset before flattening.

The `flat_map` Method: Example

```
dataset = tf.data.Dataset.range(6).window(4,
  shift=1, drop_remainder=True)
dataset = dataset.flat_map(
  lambda window_dataset: window_dataset.batch(4))

for window_tensor in dataset:
    print(f"{window_tensor}")
```

yields

```
[0 1 2 3]
[1 2 3 4]
[2 3 4 5]
```

Creating a Small Helper Function

- Create a helper function to create a windowed dataset:

```
def to_windows(dataset, length):
    dataset = dataset.window(length, shift=1, drop_remainder=True)
    return dataset.flat_map(lambda window_ds: window_ds.batch(length))
```

Add the Targets

- Call `map` to create tuples with windows and targets:

```
dataset = to_windows(tf.data.Dataset.range(6), 4)
dataset = dataset.map(lambda window: (window[:-1], window[-1]))
list(dataset.batch(2))
```

yields

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

Creating Train/Val/Test Sets

```
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
```

- Note how we scale the values down by a factor of one million.
  - The values will be close to the $[0, 1]$ range, which is good for neural networks.

Note on Validation and Test Sets

When dealing with time series, it is generally a good idea to split across time, rather than randomly.

Creating the Datasets for Training

```
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, # Shuffle the training windows!
    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
)
```

The data is now ready and we can train any regression model.

15.3.3 Forecasting Using a Linear Model

Using a Linear Model

We train a linear model with the Huber loss.

```
# Create a linear model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=[seq_length])
])
# Compile and train
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])
```

Validation MAE: ± 37866.

15.3.4 Forecasting Using a Simple RNN

Using a Simple RNN

- The linear model is better than naive forecasting, but it is worse than the SARIMA model (which we didn't cover in the slides).
  - The SARIMA model is a variant of the ARMA model family, with seasonality and differencing.
- The most basic RNN contains a single recurrent layer with a single neuron.

```
model = tf.keras.Sequential([
    # Note: input_shape!
    # Also note: default activation is tanh
    tf.keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Input to a Recurrent Layer

- Recurrent layers expect inputs of shape `[batch_size, time_steps, dimensionality]`.
  - For univariate time series, the dimensionality is 1.
- The `input_shape` argument never includes the batch size.
- Since recurrent layers can accept sequences of arbitrary length, the second dimension is `None`: `input_shape=[None, 1]`.
  - Note: the datasets actually contain data of shape `[batch_size, time_steps]`, but Keras adds the missing dimension automatically.

Working of Recurrent Layer

- For this model $h_{(\text{init})} = 0$ and this is passed to the single recurrent neuron along with the first input $x_{(0)}$.
- The neuron computes a weighted sum of these values, adds a bias term and applies the activation function.
- The result is the first output $\hat{y}_{(0)}$, which also equals the new state $h_{(0)}$.
- The new state $h_{(0)}$ is passed to the neuron along with the next input $x_{(1)}$ and the process repeats for all steps.
- The last value is $\hat{y}_{(55)}$, because there are 56 time steps.

`return_sequences` and default activation

By default, recurrent layers in Keras only return the final output vector. To make them return one output per time step, you must set `return_sequences=True`.

The default activation function for recurrent layer is the hyperbolic tangent (`tanh`*).*

Results of the Simple RNN

- The result of the simple RNN is extremely poor: MAE is more than 100000!
- There are two reasons for this:
  - A single recurrent neuron can only use the current input and the output value from the previous time step. It has very limited short-term memory. Moreover, this model only has 3 parameters: one weight for the input, one weight for the output of the previous time step, and one bias term. (The linear model had 57 parameters.)
  - The default activation function is tanh which outputs values between -1 and 1. The targets, however, contain values from 0 to around 1.4 .

Fixing Both Problems

- We can fix both problems by
  - Using a recurrent layer containing 32 neurons.
  - Adding a dense layer on top of it with a single neuron and no activation function.
- The recurrent layer will be able to carry much more information from one time step to the next, and the dense layer will project the final output from 32 dimensions down to 1, without any constraints on the value range.

Fixing Both Problems

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

This model can reach a validation MAE of 27703. This is the best model we've trained so far.

Inspecting the Model (Extra)

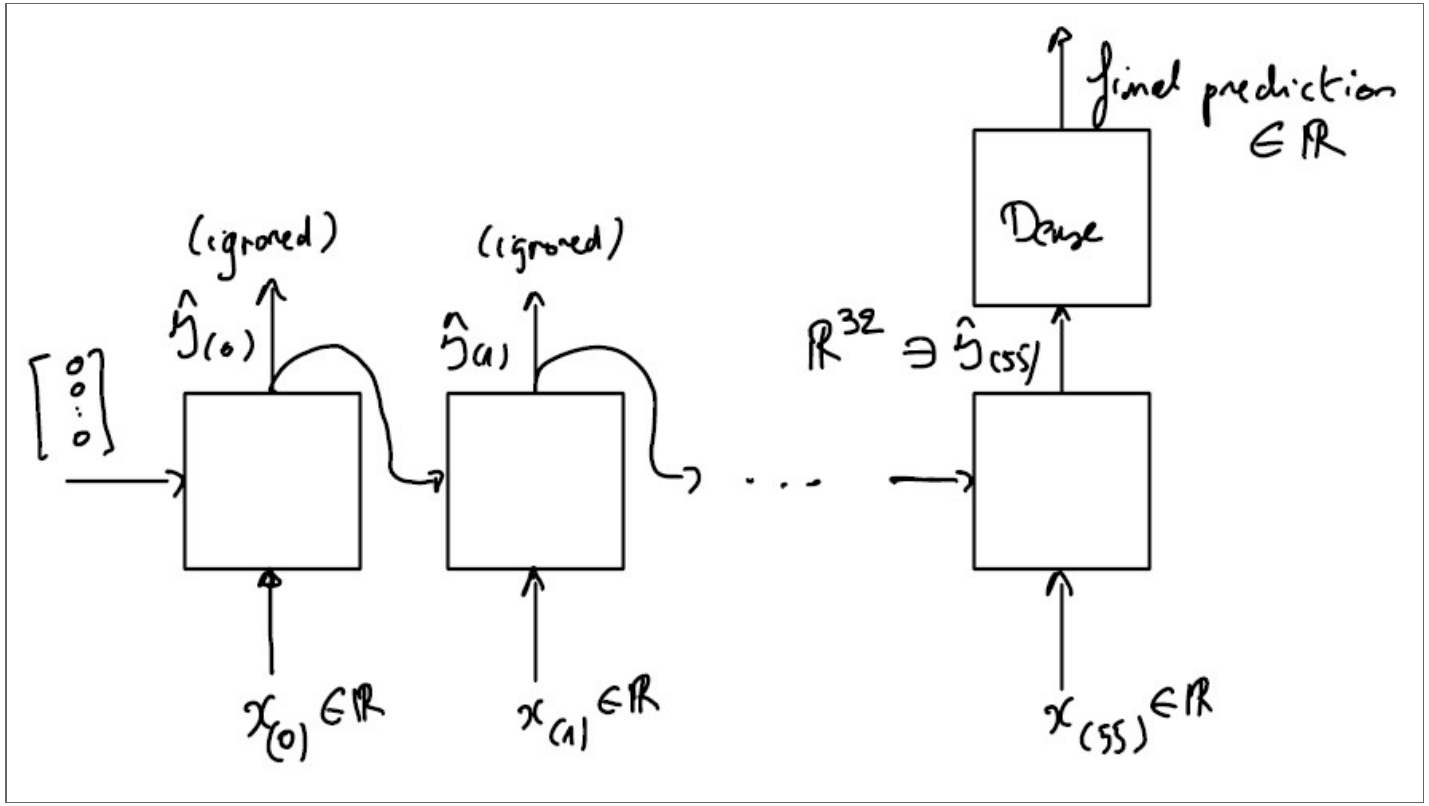
- The model summary tells us that the model has 1121 parameters:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 32)                1088

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 1,121
Trainable params: 1,121
Non-trainable params: 0
```

Inspecting the Model (Extra)

- The input sequence to the recurrent layer is univariate, and the output has 32 neurons, thus
  - $\mathbf{W_X} \in \mathbb{R}^{1\times 32}$ and $\mathbf{W_Y} \in \mathbb{R}^{32\times 32}$.
  - The bias has 32 neurons.
  - Thus: $32 + 32 \times 32 + 32 = 1088$ parameters.
- The dense layer takes as input a vector of length 32 and projects it down to a single dimension. Thus, it has 32 weights and one bias, for a total of 33 parameters.

Inspecting the Model (Extra)



Picture of the model

Remark

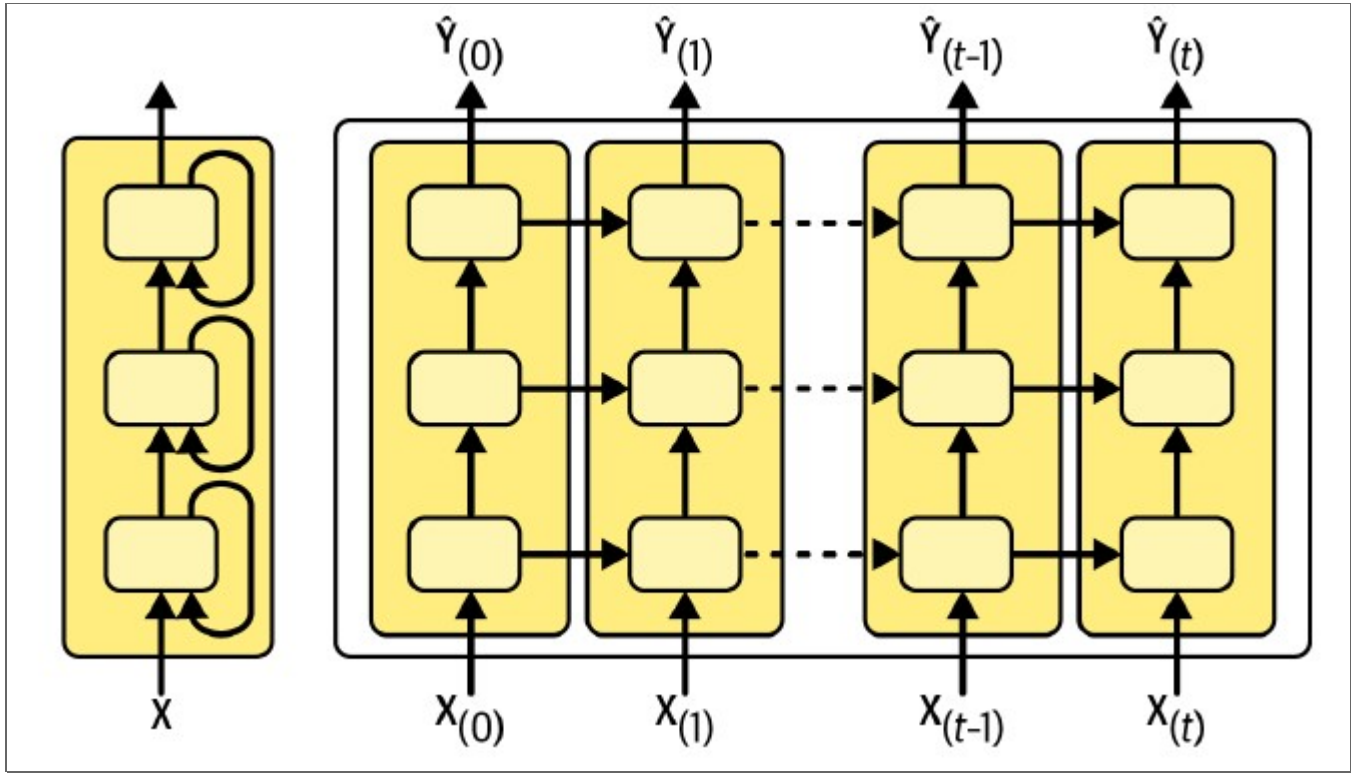
- We’ve only normalized the time series by dividing by 1 million.
- We didn’t remove trend nor seasonality.
- We can probably get even better performance by removing trend and seasonality, e.g. by using differencing.

15.3.5 Forecasting Using a Deep RNN

Using a Deep RNN

- In a **deep RNN** multiple recurrent layers are stacked on top of each other.
- This is easy in Keras, just add more recurrent layers to the model.
  - Do not forget to use `return_sequences=True` for all recurrent layers (except the last one).

A Deep RNN



A Deep RNN

Implementing 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)])
```

- In this code, we stacked three recurrent layers.
  - The first two are sequence-to-sequence layers, so they return sequences.
  - The last one is a sequence-to-vector layer, so it returns a single vector.
- Again, we add a `Dense` layer on top of the recurrent layers to project the outputs down to a single value.

Results

- This RNN performs worse than the previous one: validation MAE is about 31211.
- It is probably too large for the task at hand.
  - First recurrent layer has: $\mathbf{32 + 32 \times 32 + 32 = 1088}$ parameters (as before)
  - The second and third recurrent layer each have: $\mathbf{32 \times 32(W_X) + 32 \times 32(W_Y) + 32(b) = 2080}$ parameters.
  - The final dense layer still has 33 parameters:
  - Total number of parameters: $\mathbf{1088 + 2080 + 2080 + 33 = 5281}$.

15.3.6 Forecasting Multivariate Time Series

Forecasting Multivariate Time Series

- We can forecast multivariate time series with almost no change to the network architecture.
- Example: forecast the rail ridership using
  - the bus and rail data as input
  - and the day type as well. (This is OK, since we know in advance whether tomorrow is going to be a weekday, weekend or holiday.) We shift the day type series one day so that the model gets tomorrow's day type as input.
  - The day type will be one-hot encoded.

Preprocessing using Pandas

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

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

Creating `tf.data.Dataset`s

```
train_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),  # use all 5 columns as input
    # forecast only the rail series
    targets=mulvar_train["rail"][seq_length:],
    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
)
```

Creating the RNN

```
mulvar_model = tf.keras.Sequential([
    # input_shape denotes a multivariate series as input
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(1)
])
```

- The input shape is now `[None, 5]` because at each time step the model receives 5 inputs (two real values, and three binary values).
- This model reaches a validation of MAE of 22026. The best model so far!

Predicting Multiple Outputs

- You can easily adapt the RNN to forecast both the bus and rail ridership.
  - Just change the targets to `mulvar_train["bus", "rail"][seq_length:]` for training (and likewise for validation).
  - Give the `Dense` layer two neurons instead of one.
- In general, using a single model for multiple related tasks can help with performance, but in this case it doesn't.

15.3.7 Forecasting Several Time Steps Ahead

Predicting Several Time Steps Ahead

- Suppose we not only want to predict tomorrow's ridership, but the next 14 days.

Approach 1: Use Model to Predict One Day Ahead

- We can take a model like the `univar_model` RNN that we trained and make it predict the next value.
- This value is then added to the input sequence (acting as if the predicted value actually occurred) and the model is used to predict the next value.
- We can repeat this process for the desired number of time steps.

Approach 1: Code

```
import numpy as np

# X.shape = (1,56,1)
X = rail_valid.to_numpy()[np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = univar_model.predict(X)
    X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)
    # note: X.shape is (1,57,1), (1,58,1) ....
```

Approach 1: Remark

If the model makes an error at one time step, the forecasts for the following time steps are impacted as well. Errors tend to accumulate. It is best to use this technique only for a small number of time steps.

Approach 2: Predict 14 Values at Once

- We can still use a sequence-to-vector model, but the targets will be 14 values instead of one.
- Thus, targets need to be vectors containing the next 14 values.
- We use `timeseries_dataset_from_array` but do not create targets (`targets=None`).
  - The sequences are longer: `seq_length + 14`.
  - Next, we use `map` to split each batch into inputs and targets

Approach 2: Code

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

# mulvar_train is the multivariate training set from before

ahead_train_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=None,
    sequence_length=seq_length + 14,
    batch_size=32,
    shuffle=True,
    seed=42
).map(split_inputs_and_targets)
ahead_val_ds =  # similar but without shuffling
```

Approach 2: Code

Building the model is easy:

```
ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    # Dense layer has 14 units, since we want to predict 14 values
    tf.keras.layers.Dense(14)
])
```

Predicting the next 14 values is done like this:

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

15.3.8 Forecasting Using a Sequence-to-Sequence Model

Using a Seq-to-Seq Model

- Instead of predicting the next 14 values only at the last time step, we can train a model to predict the next 14 values *at each and every time step*!
  - Thus, we use a sequence-to-sequence RNN (instead of a sequence-to-vector RNN).
- Many more gradients will flow through the model, and they won't have to flow through as many time steps.
  - This will both stabilize and speed up training.

Using a Seq-to-Seq Model

- To make this very clear:
  - At time step 0, the model will output a vector containing the forecast for time steps 1 to 14.
  - At time step 1, it will output a vector containing the forecast for time steps 2 to 15.
  - And so on.
- Thus, *the target is now a sequence*.
  - The target sequence has the same length as the input sequence.
  - At each time step, the value of the target sequence is a 14-dimensional vector.

Preparing the Data

- Preparing the data is the hardest part.
- Remember the `to_windows` function we created earlier?

```
def to_windows(dataset, length):
    dataset = dataset.window(length, shift=1, drop_remainder=True)
    return dataset.flat_map(lambda window_ds: window_ds.batch(length))
```

Preparing the Data

- Simple use case: series `[0,1,2,3,4,5,6]`.
  - We will use use sequences of length 4.
  - At each time step, we try to predict the next two values.
- Thus, for `[0,1,2,3]` the target should be `[[1,2],[2,3],[3,4],[4,5]]`.
- for `[1,2,3,4]` the target should be `[[2,3],[3,4],[4,5],[5,6]]`.

Preparing the Data

- Calling `to_windows` twice results in the following:

```
my_series = tf.data.Dataset.range(7)
dataset = to_windows(to_windows(my_series, 3), 4)
list(dataset)
```

yields

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

Preparing the Data

- Next, use `map` to split these windows of windows into inputs and targets:

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

yields

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

Note on this Data

Even though the targets contain data also present in the input, the RNN cannot cheat. An RNN only knows about past time steps, and it cannot look into the future. It is a ***causal*** *model.*

Utility Function

- Let's put this into a utility function (that also takes care of batching and optionally shuffling):

```
def to_seq2seq_dataset(series, seq_length=56, ahead=14, target_col=1,
                       batch_size=32, shuffle=False, seed=None):
  ds = to_windows(tf.data.Dataset.from_tensor_slices(series), ahead + 1)
  # In the book the next line is incorrect. I changed 1 to target_col.
  ds = to_windows(ds, seq_length).map(
          lambda S: (S[:, 0], S[:, 1:, target_col]))
  if shuffle:
      ds = ds.shuffle(8 * batch_size, seed=seed)
  return ds.batch(batch_size)
```

Creating Train and Validation Datasets

We can now easily create the training and validation datasets:

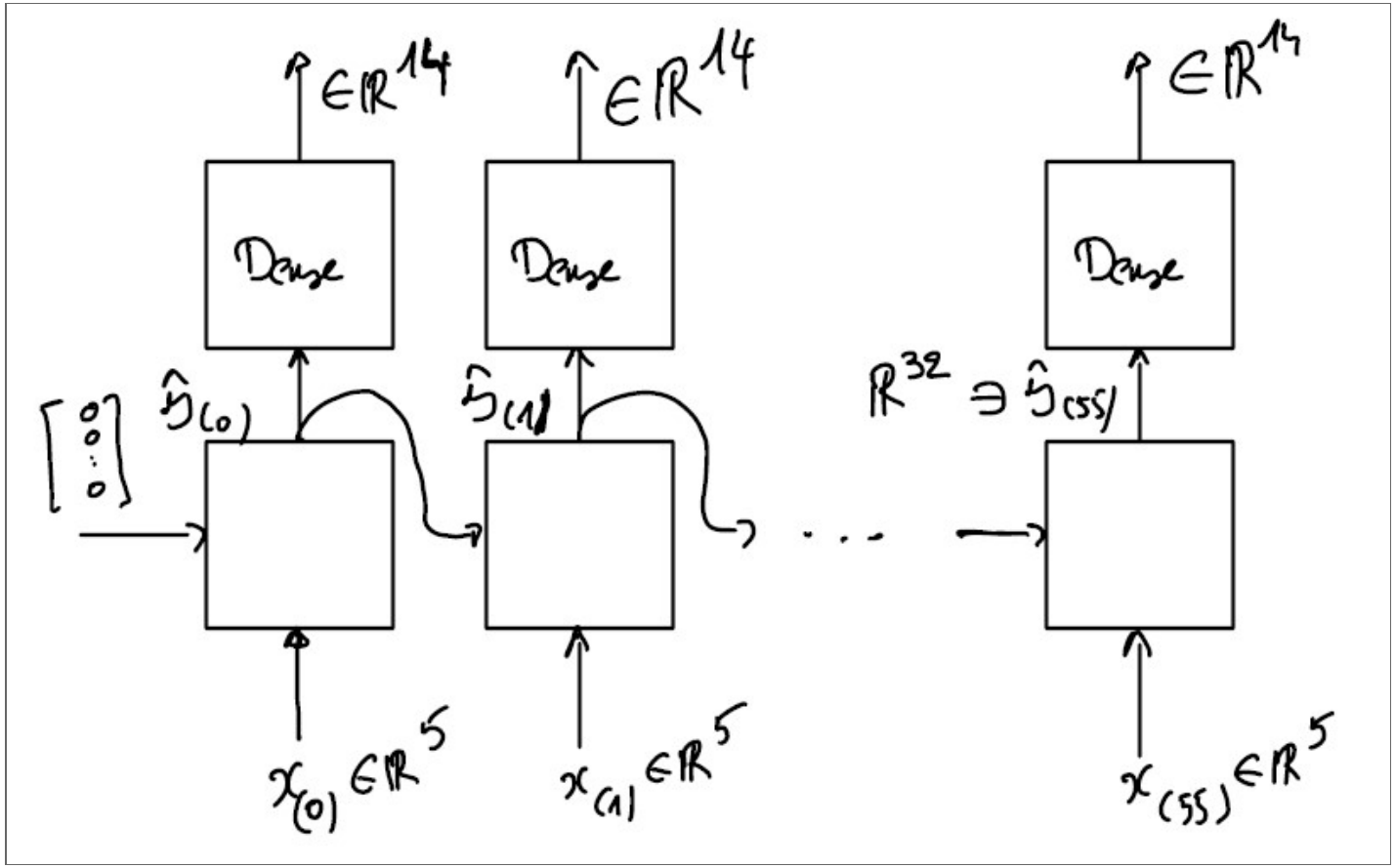
```
seq2seq_train = to_seq2seq_dataset(mulvar_train, shuffle=True, seed=42)
seq2seq_valid = to_seq2seq_dataset(mulvar_valid)
```

Building the Model

Building the model is also easy:

```
seq2seq_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True,
                              input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
    # equivalent:
    # tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(14))
    # also equivalent:
    # tf.keras.layers.Conv1D(14, kernel_size=1)
])
```

Picture of the Model (Extra)



Picture of the model

Notes on the Model

- We use `return_sequences=True` because we want to predict a sequence. The output at each time step will be a vector of size 32.
- The `Dense` layer will be applied at each time step, taking the 32-dimensional vector and outputting a 14-dimensional vector.
  - An equivalent alternative is to use `Conv1D(filters=14, kernel_size=1)`. (See following slides.)

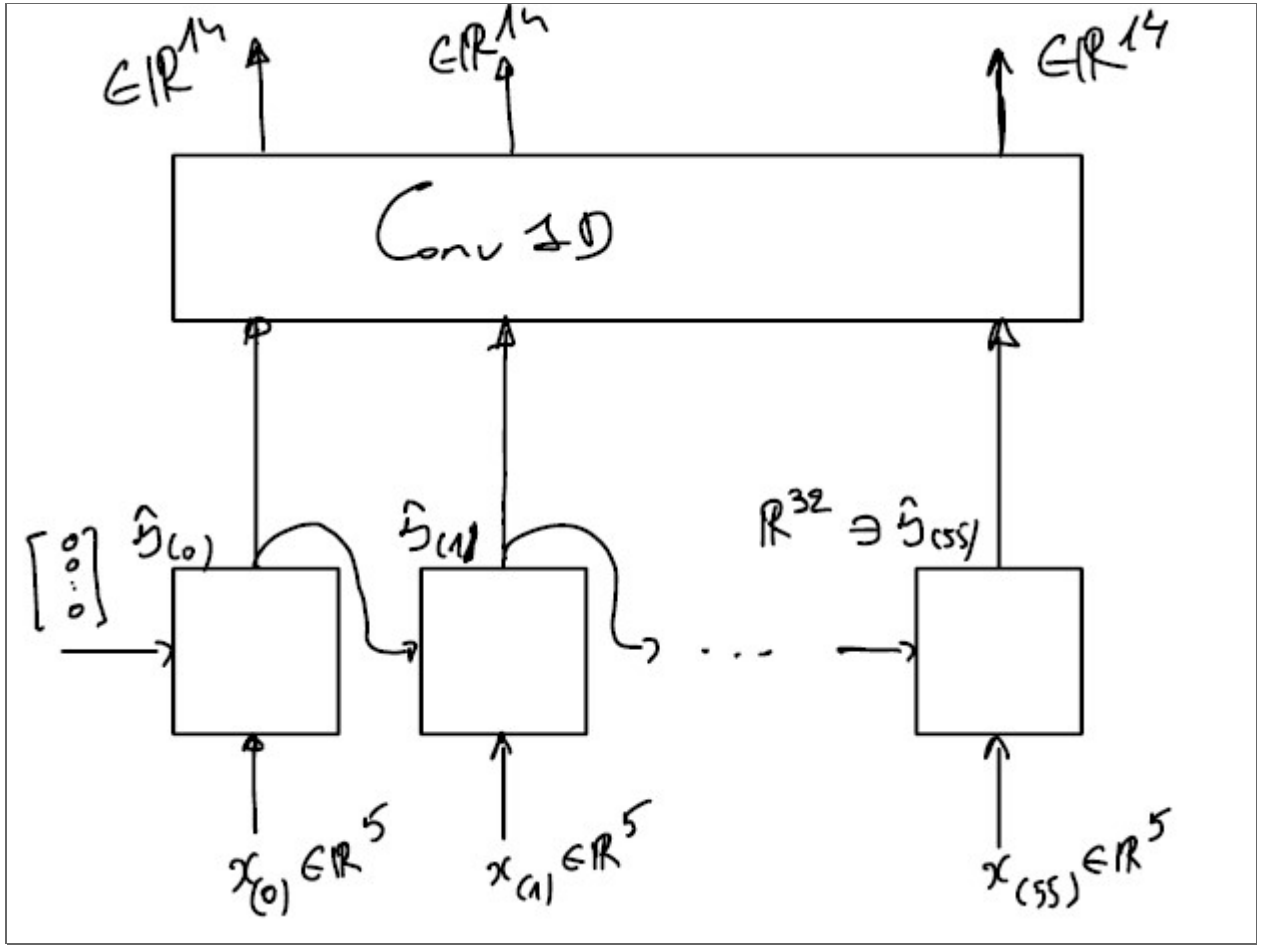
`TimeDistributed` Layer

- The `TimeDistributed` layer lets you apply any vector-to-vector layer to every vector in the input sequence, at every time step.
  - This is done efficiently.
- For the `Dense` layer this is not needed, since it already supports sequences as inputs.
  - If the input to a `Dense` layer has shape `(batch_size, d0, d1)` then the output will have shape `(batch_size, d0, units)`, where `units` is the number of neurons in the `Dense` layer.

`Conv1D` Layer instead of `Dense`

- The `Dense` layer has= $32 \times 14 + 14$ parameters.
- A `Conv1D` layer computes a convolution over a single (time) dimension.
  - `kernel_size` is the number of time steps that the convolutional kernel will look at while considering all the channels.
  - `filters` is the number of output channels.
  - So, if we `kernel_size` equal to 1, and `filters` equal to 14, the number of parameters will be $32 \times 14 + 14$.
  - Even more, the results will be identical as that of the `Dense` layer because each filter computes a weighted sum of the 32 inputs, and adds a bias term.

Model with Conv1D Layer (Extra)



Picture of the model

Using the Seq-to-Seq Model

- Training the seq-to-seq model is the same as usual. During training, all the model's outputs are used.
- However, at inference time, only the very last time step matters (and the rest can be ignored).

```
X = mulvar_valid.to_numpy()[np.newaxis, :seq_length]
 # only the last time step's output is used
y_pred_14 = seq2seq_model.predict(X)[0, -1]
```

Evaluating this Model

- For timestep $t+1$ the validation MAE for this model is 25519, for $t+2$ it is 26274 and it gradually gets worse until for $t+14$ we get a validation MAE of 34322.

15.4 Handling Long Sequences

Long Sequences

- When you train an RNN on long sequences, the unrolled network becomes a very deep network.
  - Just, like any deep network if may suffer from unstable gradients.
- Moreover, when an RNN processes a very long sequence, it gradually forgets the first inputs in the sequence.
  - This is called the **short-term memory problem**.

15.4.1 Fighting the Unstable Gradients Problem

Techniques to Fight Unstable Gradients

- Much of the same techniques that we used to fight unstable gradients in deep networks can be used to fight unstable gradients in RNNs.
  - Good parameter initialization.
  - Faster optimizers.
  - Dropout
- Nonsaturing activation functions (like ReLU) are not helpful.
  - This is why `tanh` is the default activation function for RNNs.

Layer Normalization

- Batch normalization doesn't help much with RNNs.
- **Layer normalization**, however, can be used with RNNs.
  - Layer normalization normalizes across the *features* dimension (instead of the batch dimension for batch norm).
  - Layer normalization behaves the same during training and testing. No need to estimate the feature statistics across all instances in the training set like BN does.
  - Layer norm learns an offset and a scale for each input feature.
  - In an RNN layer norm is typically applied after taking the linear combination of the inputs and the hidden states (but before applying the activation function).

Layer Normalization: Example (Extra)

Suppose we have a batch of 2 instances, each with 3 features:

```
[[1, 2, 3],
 [4, 5, 6]]
```

Compute the mean and standard deviation of the feature values for each instance:

```
mean = [[2], [5]], sigma = [[0.8165], [0.8165]]
```

The normalized features are:

```
[[-1.23, 0, 1.23],
 [-1.23, 0, 1.23]]
```

Layer Normalization: Example (Extra)

Finally, the layer normalization has a learnable offset β and scale parameter γ for each feature. The final result is:

```
[[-1.23, 0, 1.23],
 [-1.23, 0, 1.23]]
*
[scale1, scale2, scale3]
+
[offset1, offset2, offset3]
```

Layer Normalization: Code (Extra)

The next example (not in the book) shows how to use layer normalization in Keras:

```
import tensorflow as tf
X = tf.constant(np.arange(1,7).reshape(2, 3), dtype=tf.float32)
norm_layer = tf.keras.layers.LayerNormalization()
norm_layer(X)
```

gives

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[-1.2238274,  0.       ,  1.2238274],
       [-1.2238274,  0.       ,  1.2238274]], dtype=float32)>
```

Layer Normalization: Code (Extra)

Note that the learnable scale γ and offset β are initialized to all 1 and and all 0 respectively.

```
norm_layer.weights
```

yields

```
[<tf.Variable 'layer_normalization/gamma:0'
 shape=(3,) dtype=float32, numpy=array([1., 1., 1.], dtype=float32)>,
 <tf.Variable 'layer_normalization/beta:0'
 shape=(3,) dtype=float32, numpy=array([0., 0., 0.], dtype=float32)>]
```

15.4.2 Tackling the Short-Term Memory Problem

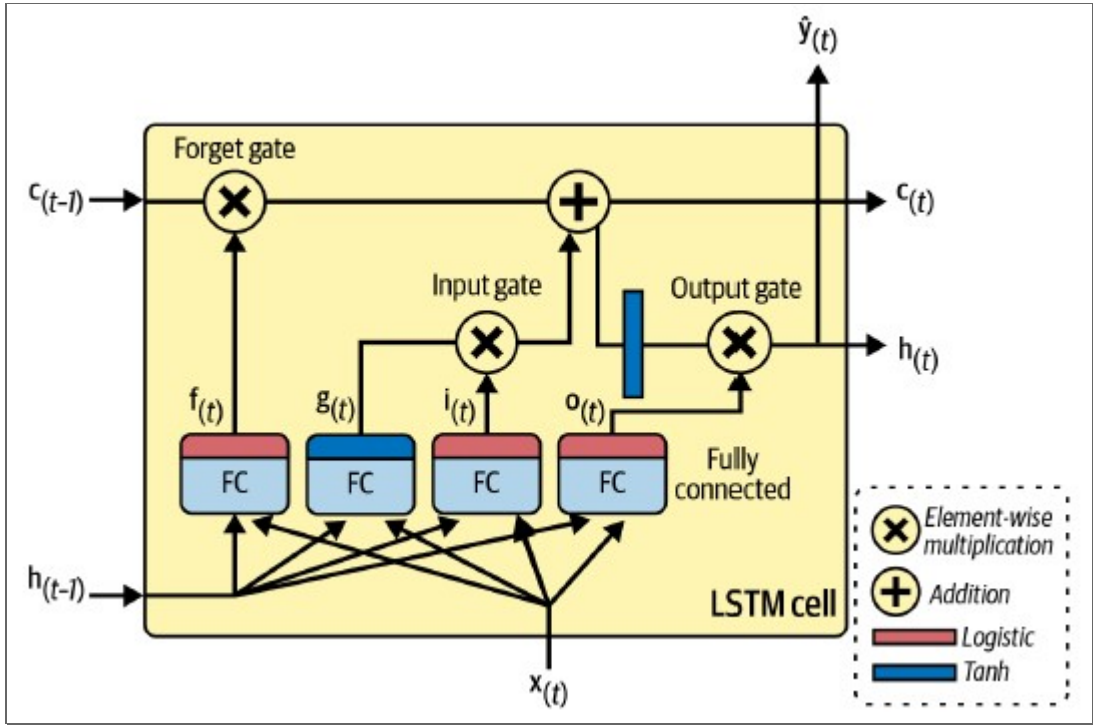
New Cell Types

- In order to tackle the **short-term memory problem**, new cell types were introduced:
  - **LTSM** (Long Short-Term Memory) cell
  - **GRU** (Gated Recurrent Unit) cell
- Both these cells are more complicated than the basic RNN cell, but can be used much in the same way.
- Even though these cells are better than the basic RNN cell at remembering information, they still have trouble handling very long sequences (more than 100 time steps).

LSTM Cell

- The *state* of an LSTM-cell is split into two vectors:
  - $\mathbf{h}_t$ can be thought of as the *short-term* state.
  - $\mathbf{c}_t$ can be thought of as the *long-term* state.
- The cell takes, as input, the input vector $\mathbf{x}_t$ and the short-term state $\mathbf{h}_{t-1}$ as well as the long-term state $\mathbf{c}_{t-1}$.
- The cell produces:
  - The output for time step t: $\hat{\mathbf{y}}_t$.
  - Both state vectors: $\mathbf{h}_t$ and $\mathbf{c}_t$.
- All three vectors $\hat{\mathbf{y}}_t$, $\mathbf{h}_t$ and $\mathbf{c}_t$ have the same number of components, defined when creating the LSTM cell.

LSTM Cell



LSTM Cell

LSTM Cell Calculations

- The LSTM cell contains 4 fully connected layers.
- Each of these takes the current (batch of) inputs $\mathbf{X}_t$ `(batch_size, num_features)` and the (previous) short-term state $\mathbf{H}_{t-1}$ `(batch_size, units)` as inputs.
  - $\mathbf{X}_t$ and $\mathbf{H}_{t-1}$ are concatenated to yield `(batch_size, num_features + units)`
  - Each of the 4 layers has `units` neurons.
  - The f, i and o layers use `sigmoid` activation functions, while the g layer uses `tanh`.

LSTM Cell Calculations (Ctd.)

The following calculations are performed on each time step:

- $\mathbf{H}_{t-1}$ and $\mathbf{C}_{t-1}$ are extracted from the "state"
- The 4 different layers are applied to the concatenation of $\mathbf{X}_t$ and $\mathbf{H}_{t-1}$ to yield tensors $\mathbf{F}_t$, $\mathbf{I}_t$, $\mathbf{O}_t$ and $\mathbf{G}_t$ each of size `(batch_size, units)`.
- $\mathbf{C}_t = \mathbf{F}_t \otimes \mathbf{C}_{t-1} + \mathbf{I}_t \otimes \mathbf{G}_t$
- $\mathbf{H}_t = \mathbf{O}_t \otimes \tanh(\mathbf{C}_t)$
- $\hat{\mathbf{Y}}_t = \mathbf{H}_t$

GRU Cell

- The GRU (Gated Recurrent Unit) cell is a simplified version of the LSTM cell.
  - There is only a single state vector $\mathbf{h}_t$.
  - There are fewer “gates” inside the cell.

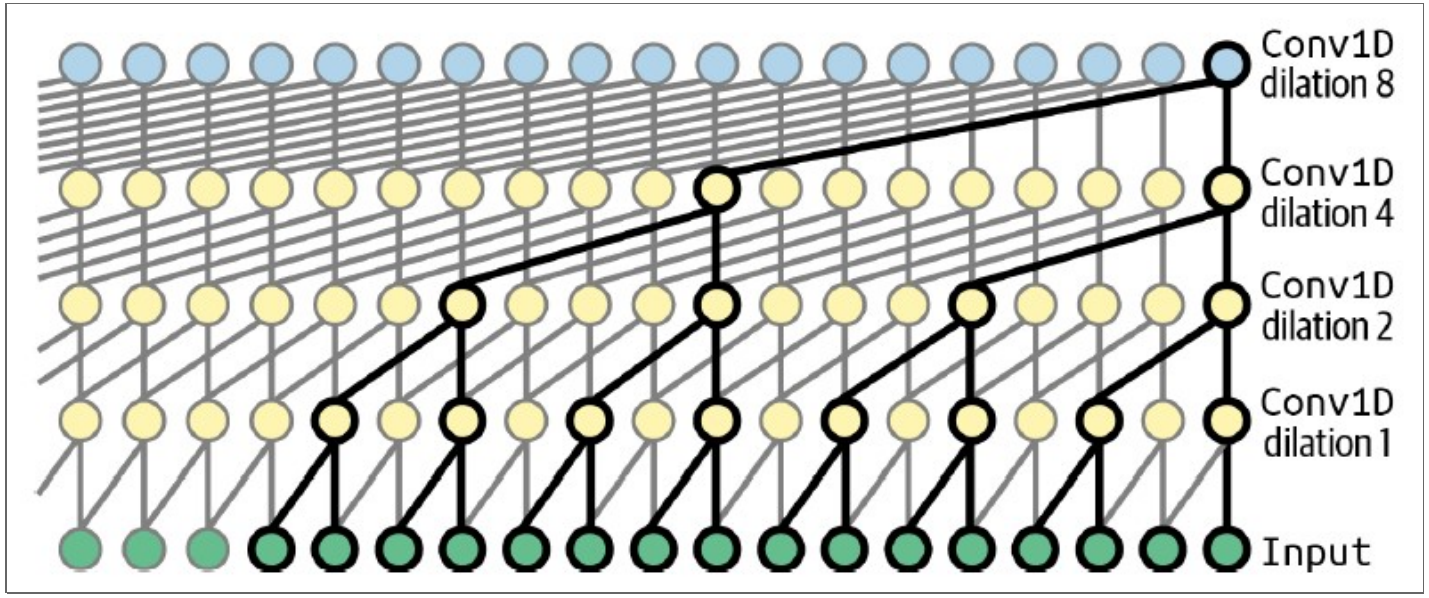
Using 1D Convolutional Layers

- A 1D convolutional layer slides several kernels across a sequence, producing a 1D feature map per kernel.
- 1D convolutional layers can also be used to process sequences.
  - They can also be used in combination with RNNs.

WaveNet

- **WaveNet** is an architecture to process very long sequences.
  - It is composed of a large stack of 1D convolutional layers.
  - Each convolutional layer uses a **dilation rate** that is double that of the previous layer.
  - Wavenet uses **causal padding** to ensure that the network can't look into the future. Causal padding is similar to "same" padding but all the zeroes are put on the left of the sequence.

WaveNet



WaveNet

Wavenet: Implementation

- We can implement two stacks of layers as shown on the previous slide, and finally add a last `Conv1D` layer to produce the final predictions as follows:

```
wavenet_model = tf.keras.Sequential()
wavenet_model.add(tf.keras.layers.InputLayer(input_shape=[None, 5]))
for rate in (1, 2, 4, 8) * 2:
  wavenet_model.add(tf.keras.layers.Conv1D(
    # Note: causal padding and dilation rate
    filters=32, kernel_size=2, padding="causal", activation="relu",
    dilation_rate=rate))
# Add final Conv1D layer to produce the predictions
wavenet_model.add(tf.keras.layers.Conv1D(filters=14, kernel_size=1))
```