Deep Learning

Lecture Topic:
Deep Learning for Time Series

Anxiao (Andrew) Jiang

Learning Objectives:

1. Understand RNN and LSTM

2. Understand how to use RNN for practical applications

Roadmap of this lecture:

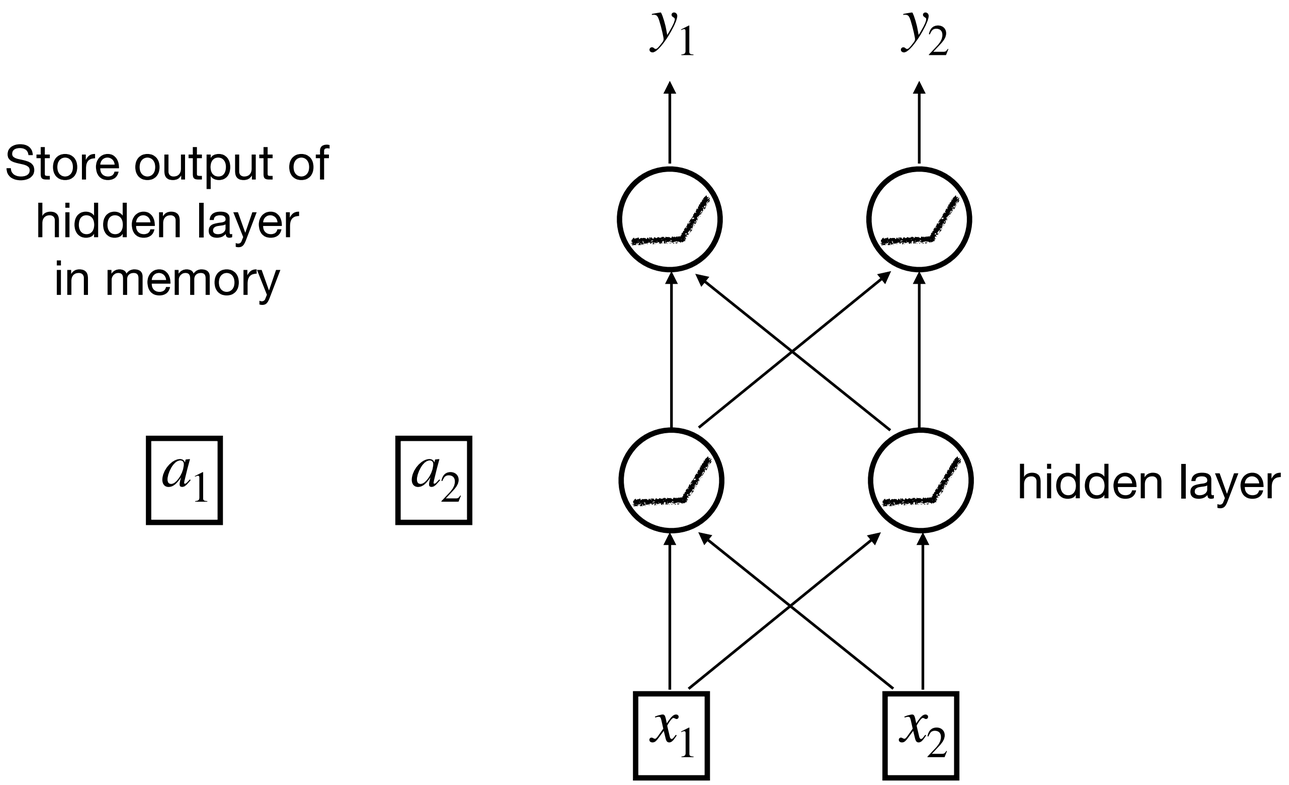
1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Recurrent Neural Network

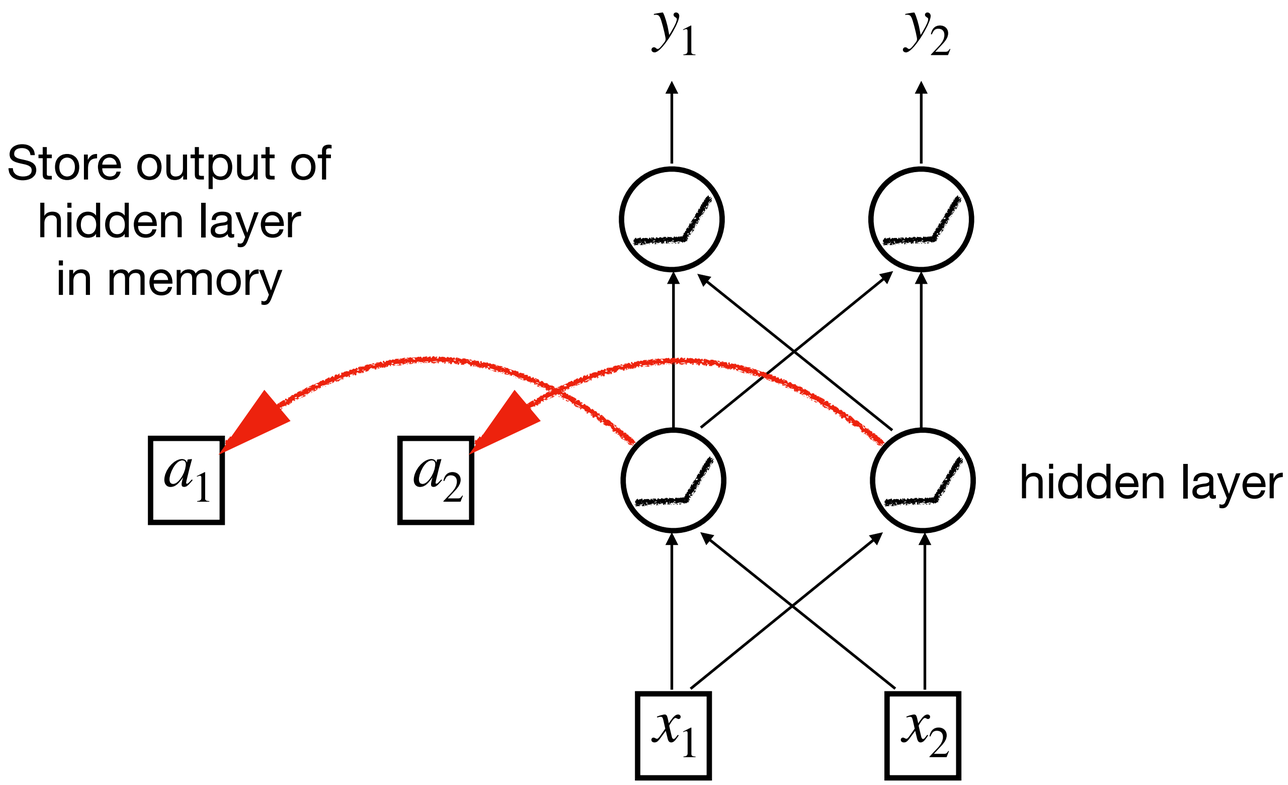
Adapted from lecture by Prof. Hung-yi Lee “Recurrent Neural Network”.

Recurrent Neural Network (RNN)

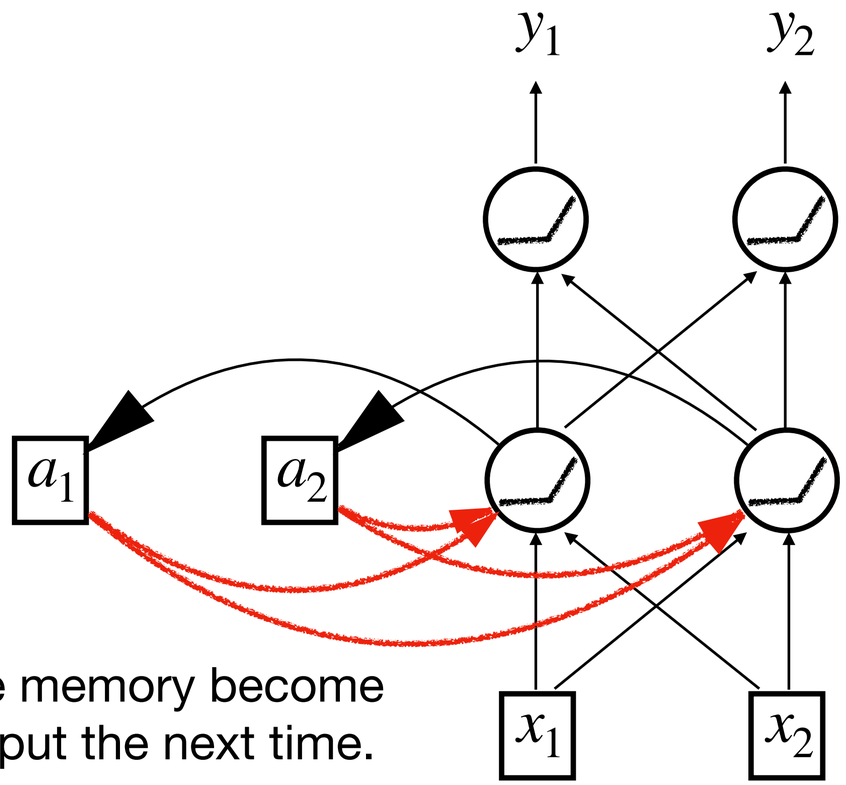
Store output of hidden layer in memory



Recurrent Neural Network (RNN)



Recurrent Neural Network (RNN)



Data in the memory become another input the next time.

Recurrent Neural Network (RNN)

When data are input as a sequence:

$(x_1^{(1)}, x_2^{(1)})$

$(x_1^{(2)}, x_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$

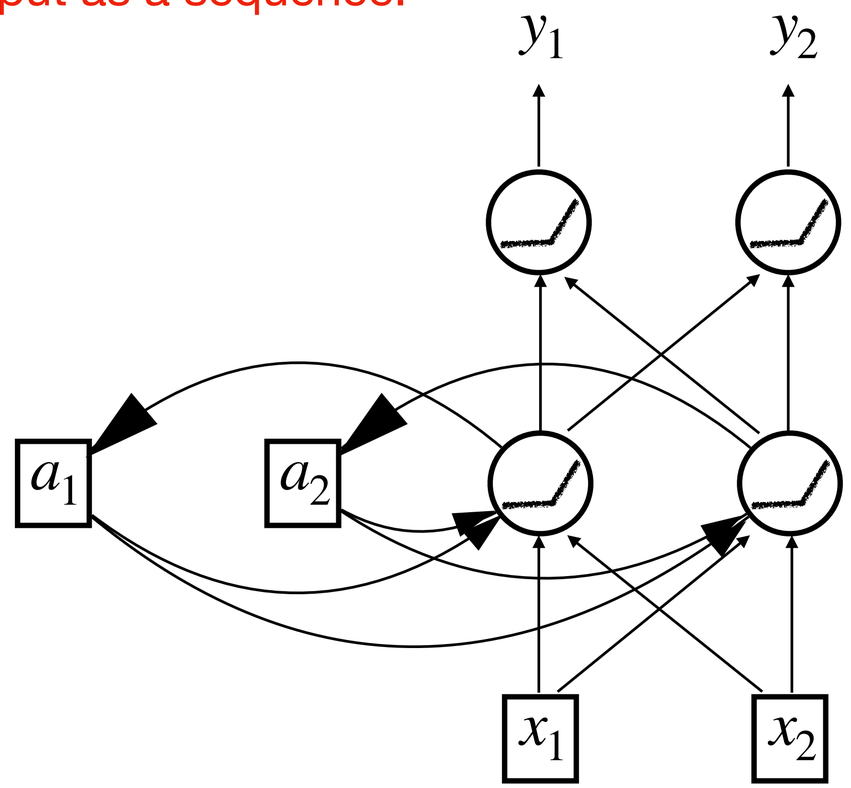
over n steps,
the output is also
a sequence:

$(y_1^{(1)}, y_2^{(1)})$

$(y_1^{(2)}, y_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$



Recurrent Neural Network (RNN)

When data are input as a sequence:

$(x_1^{(1)}, x_2^{(1)})$

$(x_1^{(2)}, x_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$

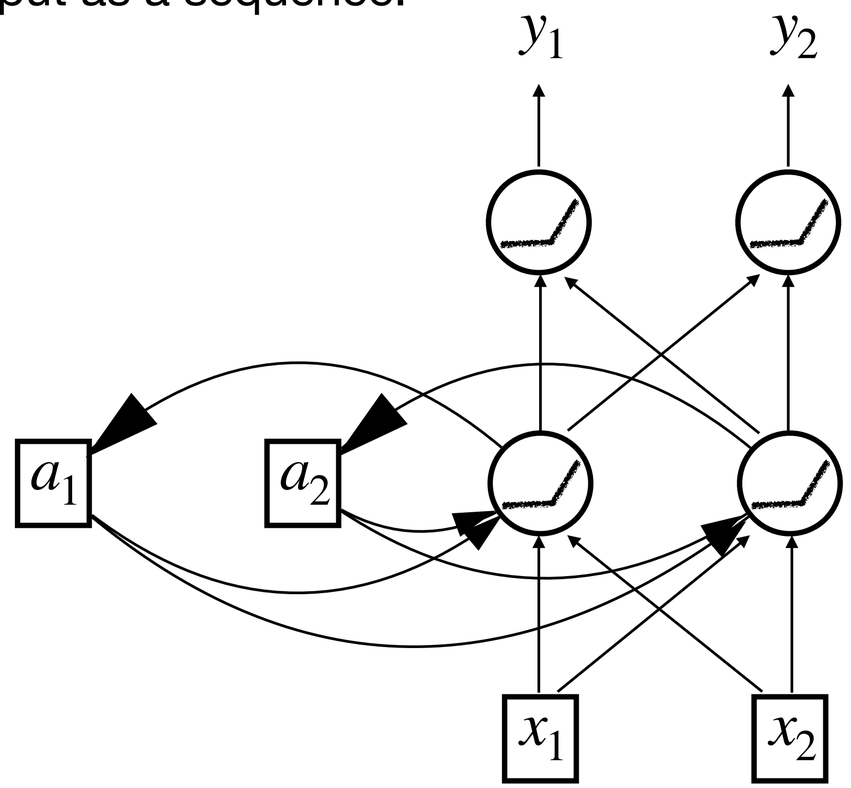
over n steps, the output is also a sequence:

$(y_1^{(1)}, y_2^{(1)})$

$(y_1^{(2)}, y_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$



Seen from the outside (of the RNN layer), the input is a matrix, and the output is also a matrix.

Recurrent Neural Network (RNN)

When data are input as a sequence:

$(x_1^{(1)}, x_2^{(1)})$

$(x_1^{(2)}, x_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$

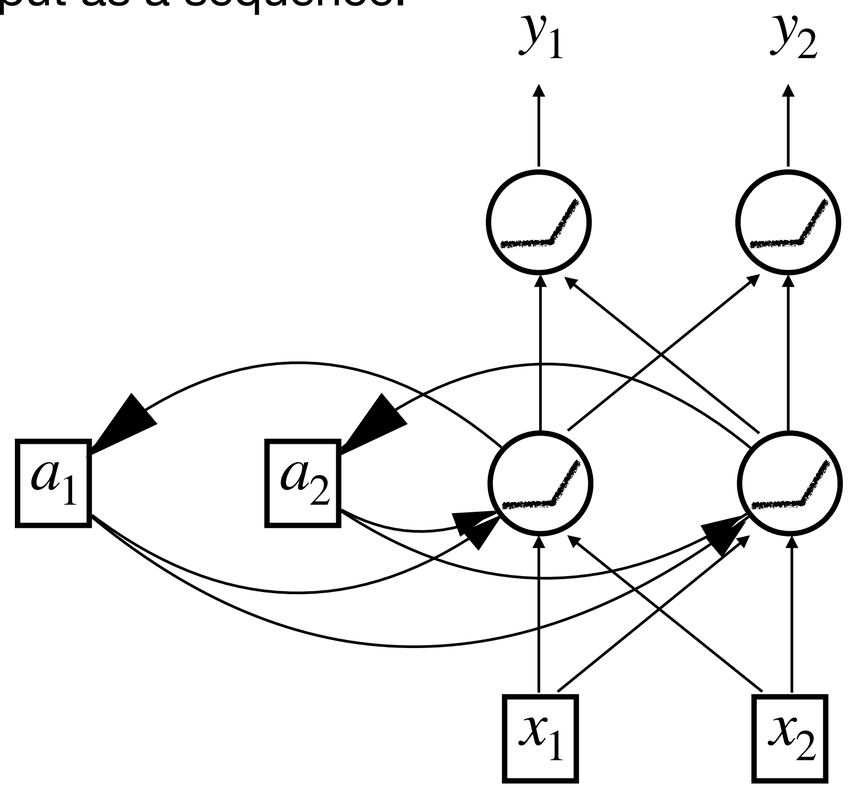
over n steps, the output is also a sequence:

$(y_1^{(1)}, y_2^{(1)})$

$(y_1^{(2)}, y_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$



The output at any time step depends not only on the current input, but also the previous inputs in the sequence.

Recurrent Neural Network (RNN)

When data are input as a sequence:

$(x_1^{(1)}, x_2^{(1)})$

$(x_1^{(2)}, x_2^{(2)})$

$\vdots$

$(x_1^{(n)}, x_2^{(n)})$

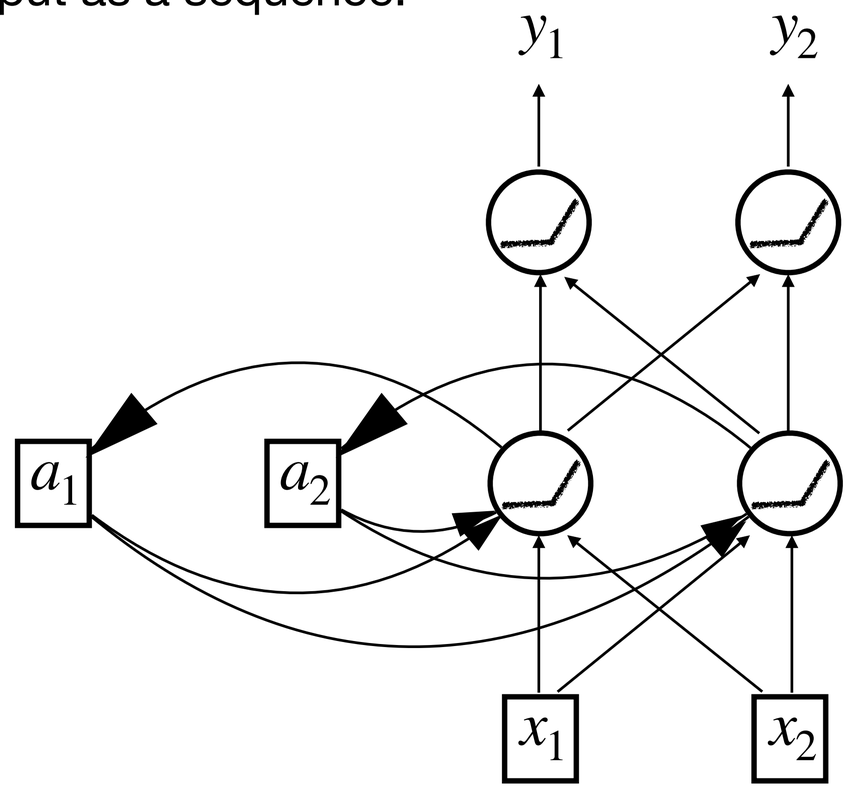
over n steps, the output is also a sequence:

$(y_1^{(1)}, y_2^{(1)})$

$(y_1^{(2)}, y_2^{(2)})$

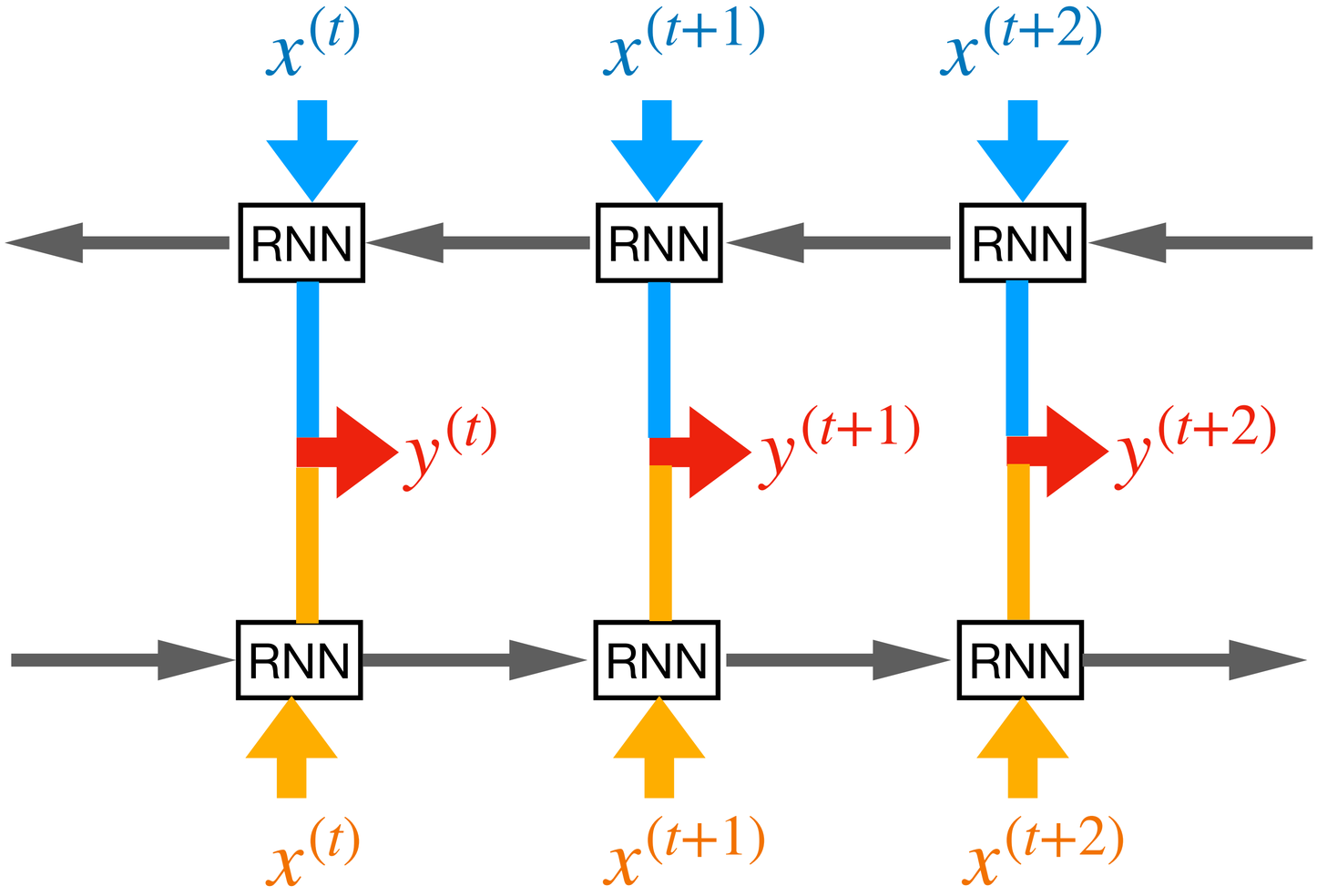
$\vdots$

$(x_1^{(n)}, x_2^{(n)})$

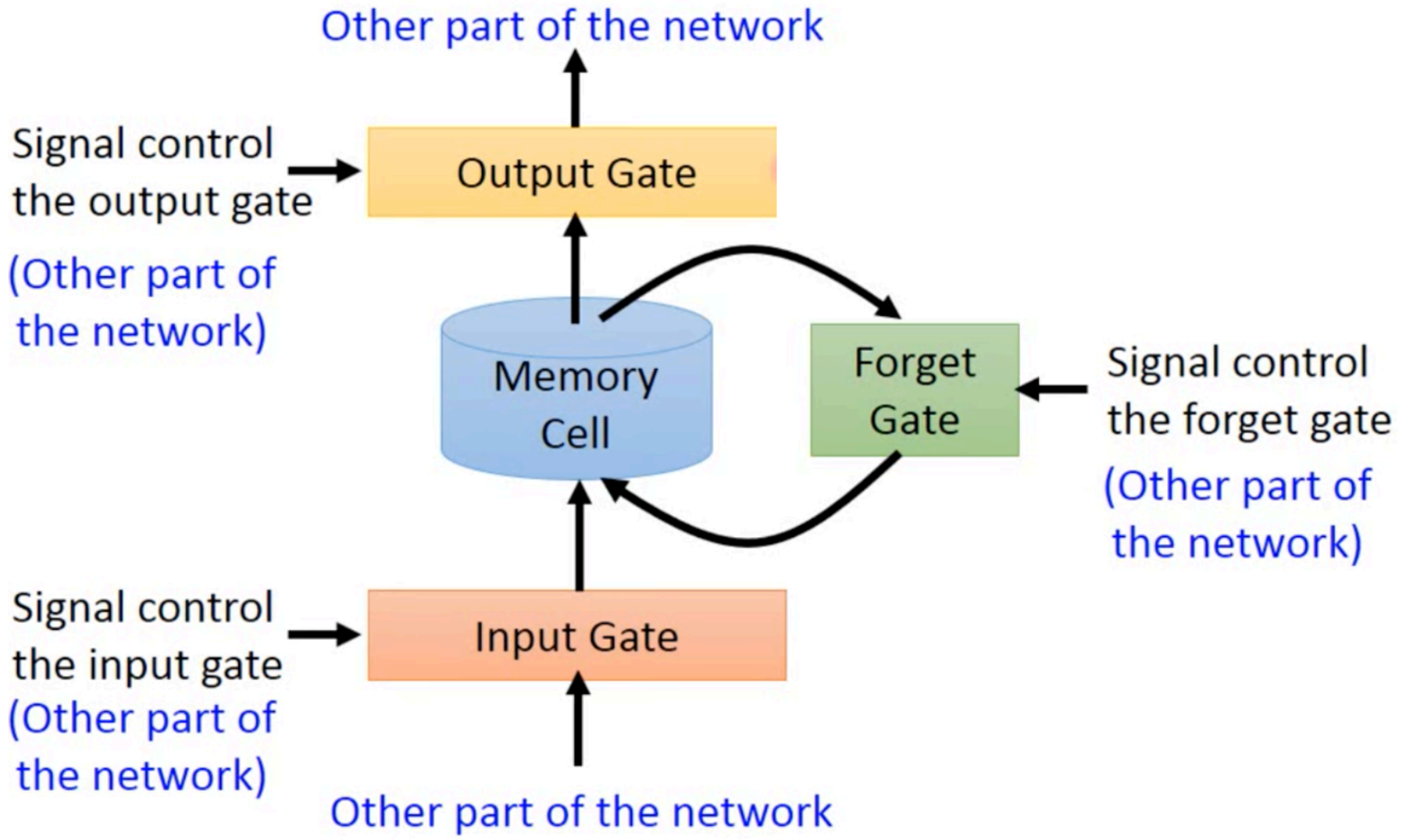


Multiple RNN layers can be stacked.

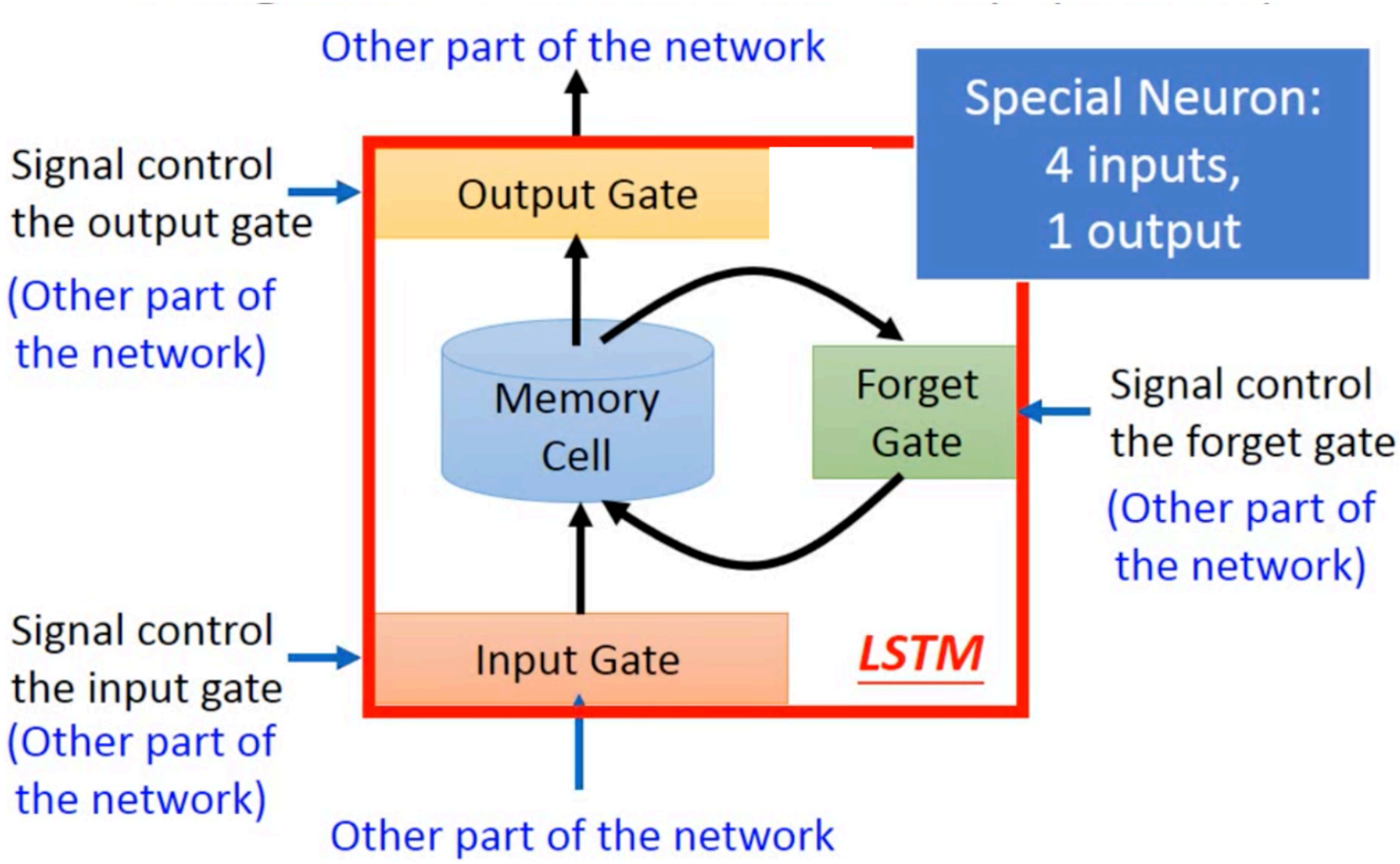
Bidirectional RNN



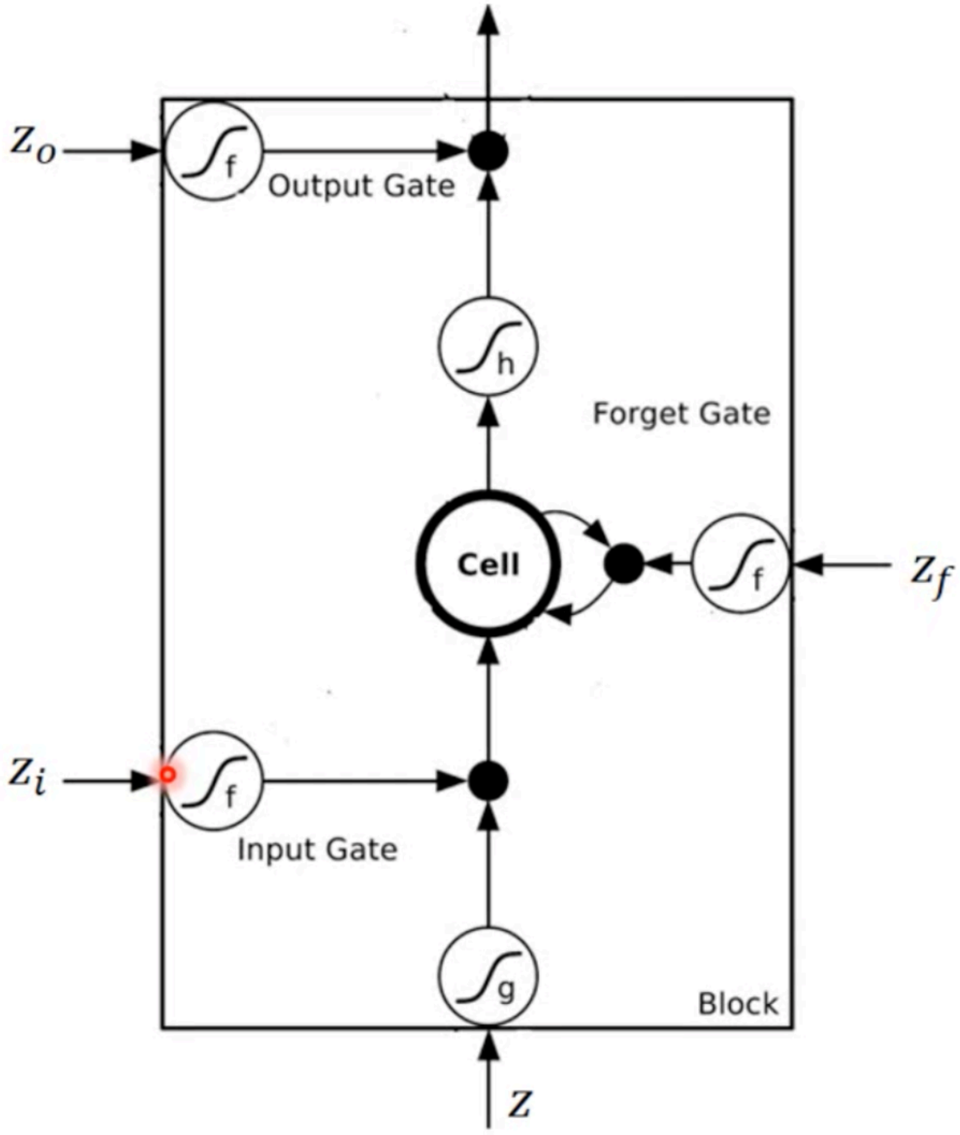
Long Short-Term Memory (LSTM): a cell

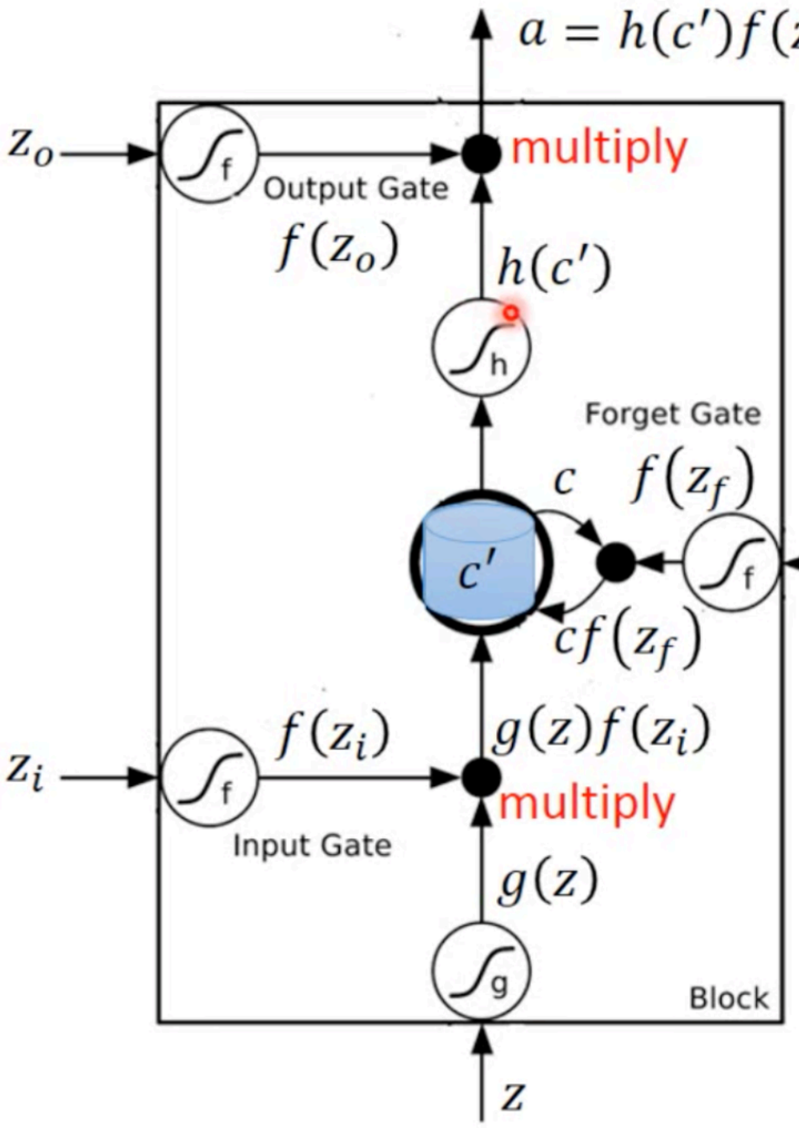


Long Short-Term Memory (LSTM): a cell



Long Short-Term Memory (LSTM): a cell





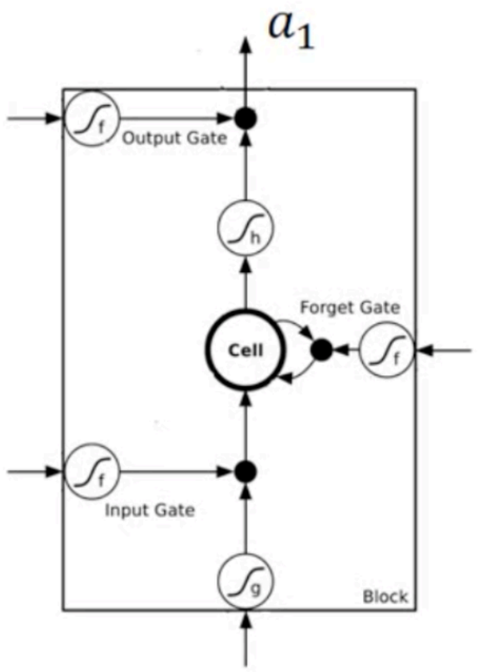
Activation function f is usually a sigmoid function

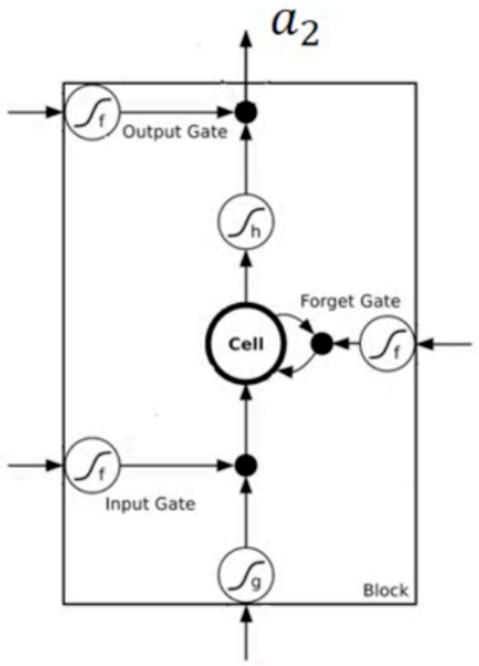
Between 0 and 1

Mimic open and close gate

$$c' = g(z)f(z_i) + cf(z_f)$$

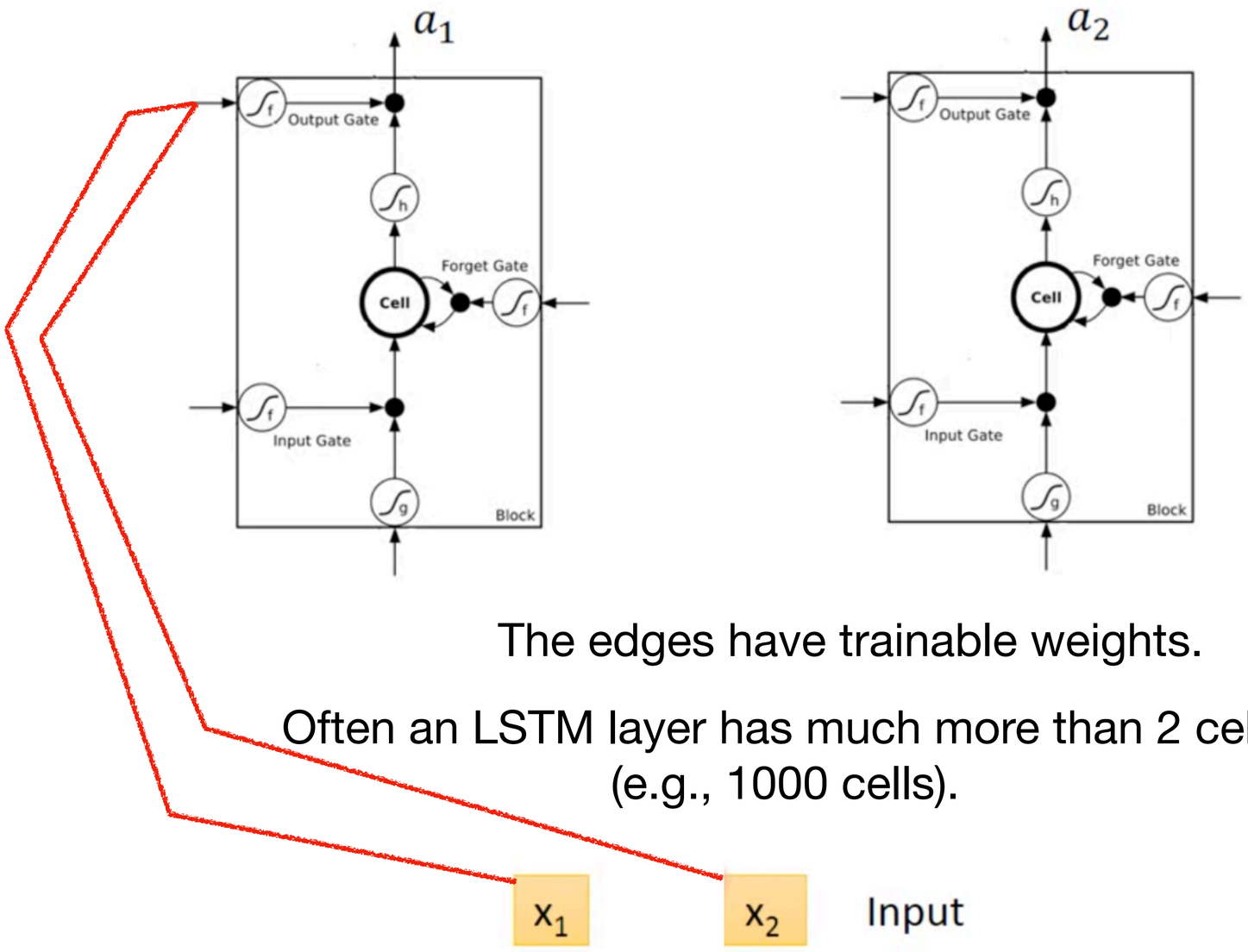
LSTM layer: use LSTM cells to replace normal neurons





x_1 x_2 Input

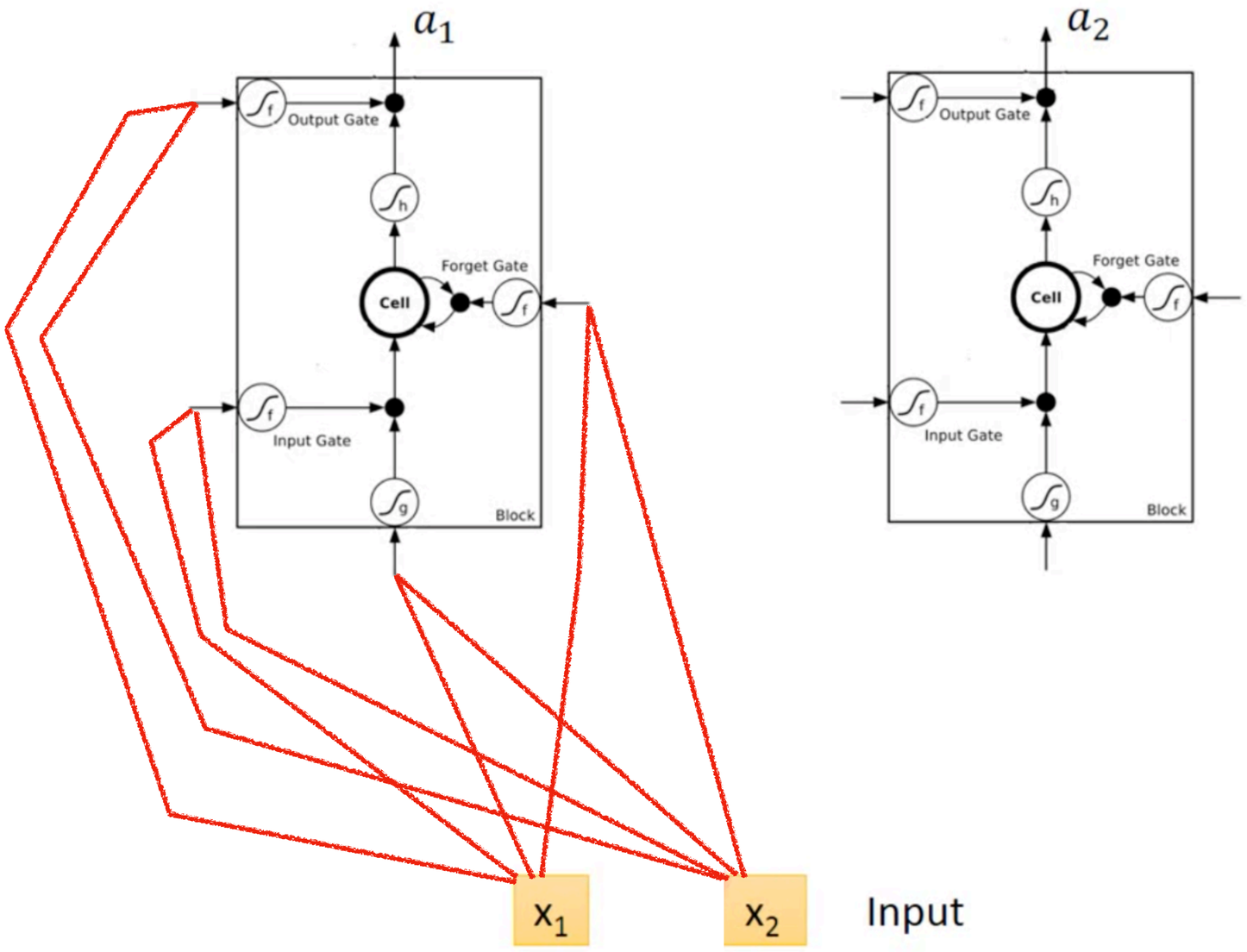
LSTM layer: use LSTM cells to replace normal neurons



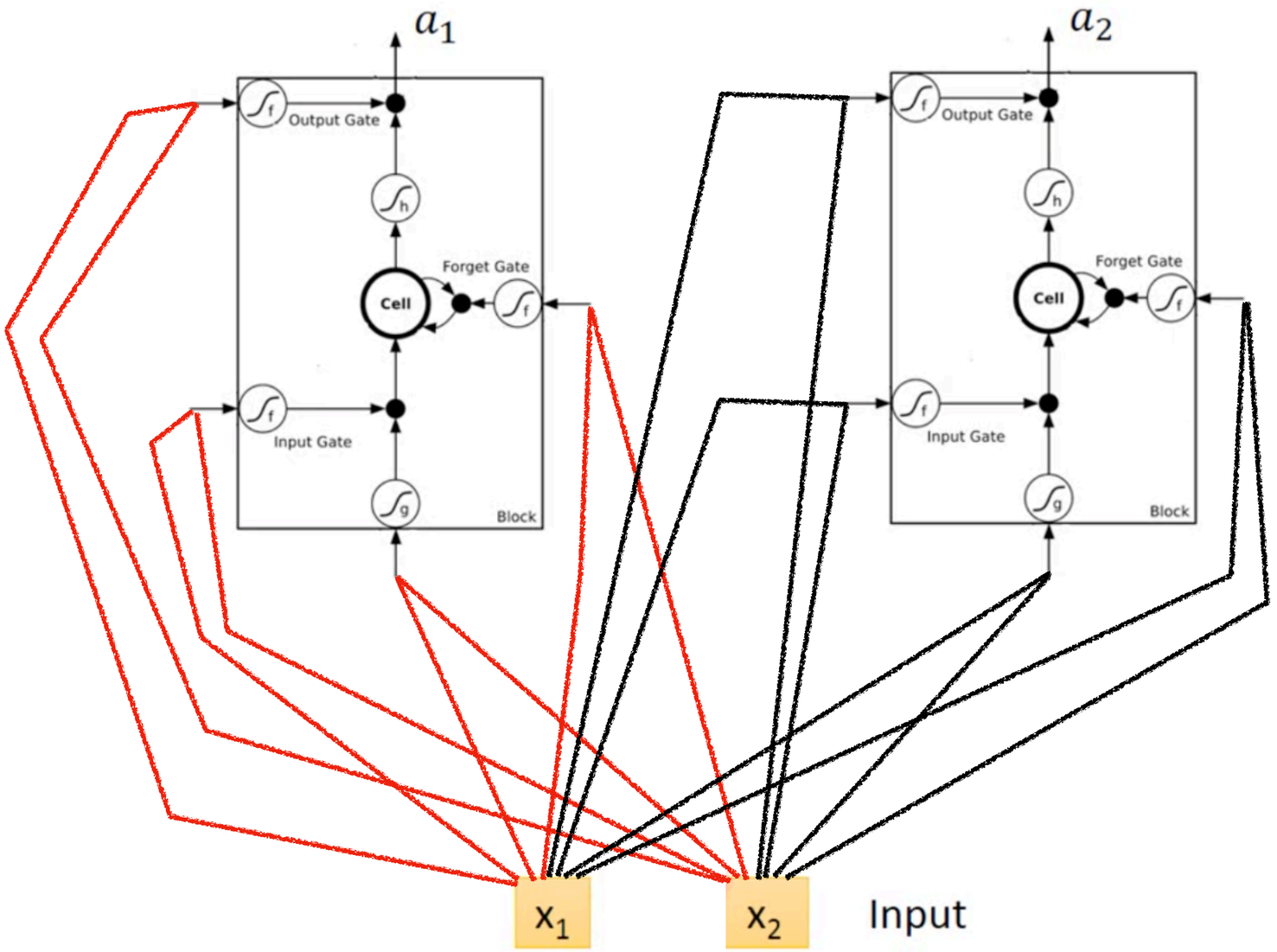
The edges have trainable weights.

Often an LSTM layer has much more than 2 cells (e.g., 1000 cells).

LSTM layer: use LSTM cells to replace normal neurons

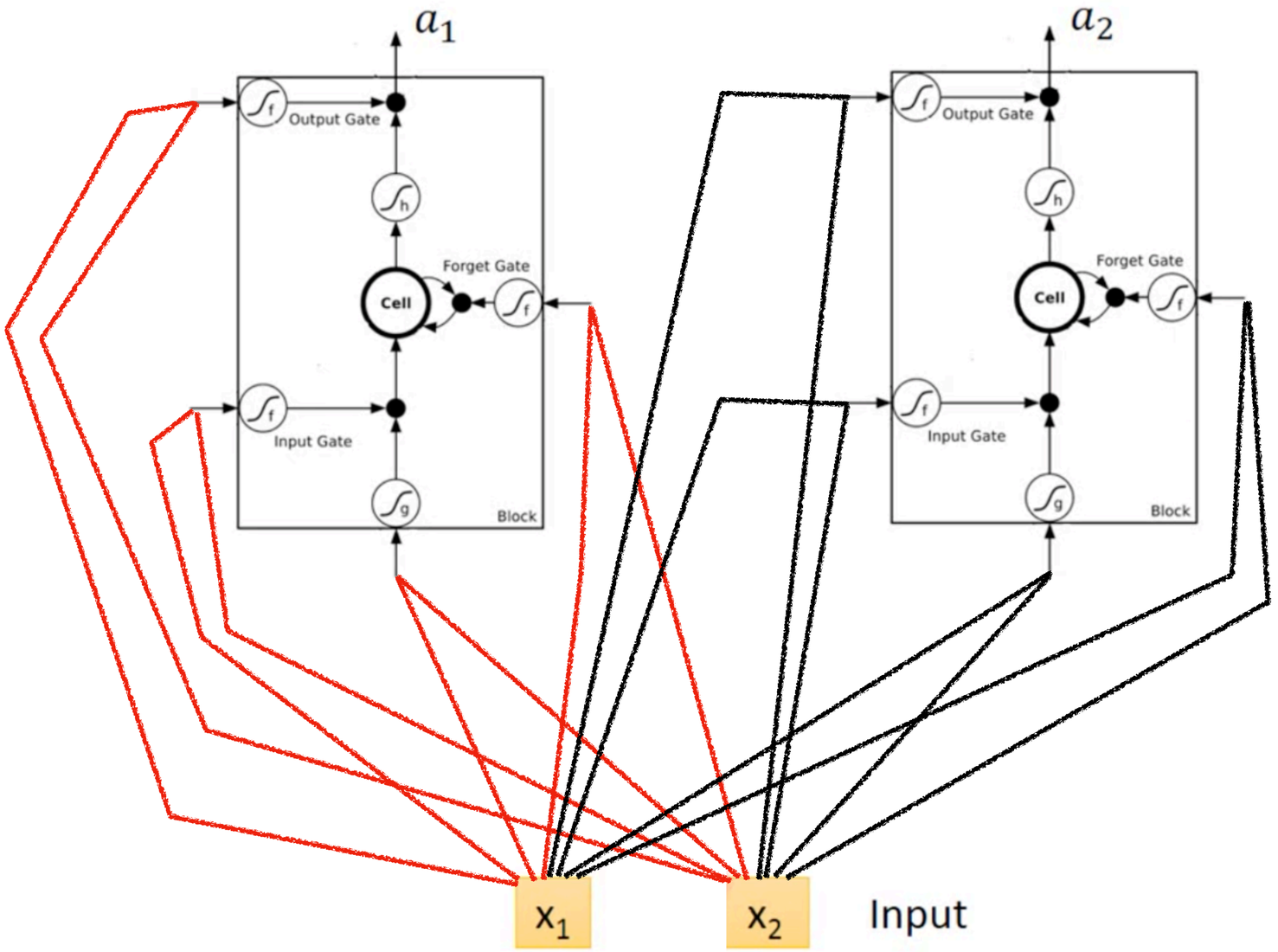


LSTM layer: use LSTM cells to replace normal neurons

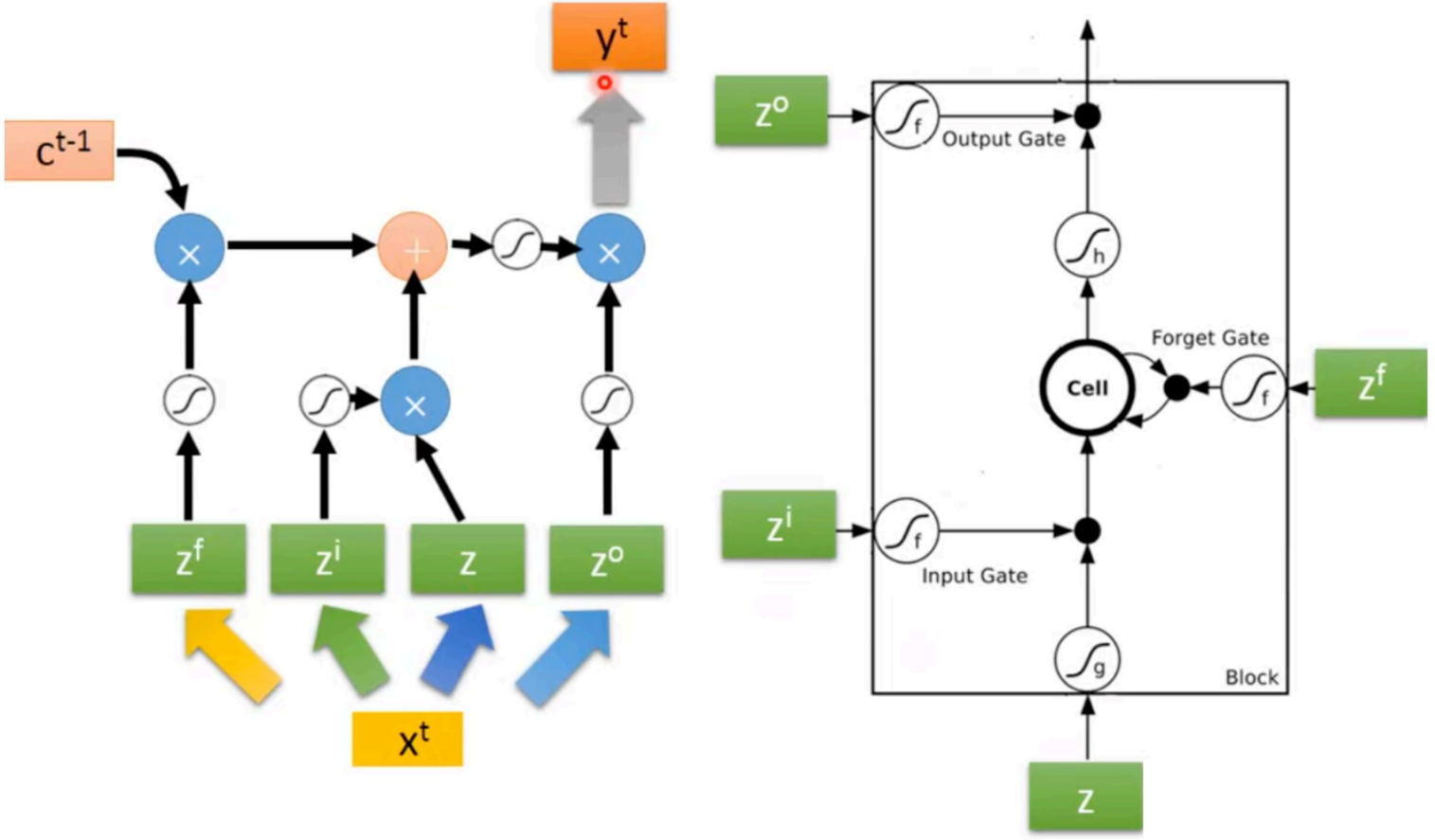


All edges have different weights.

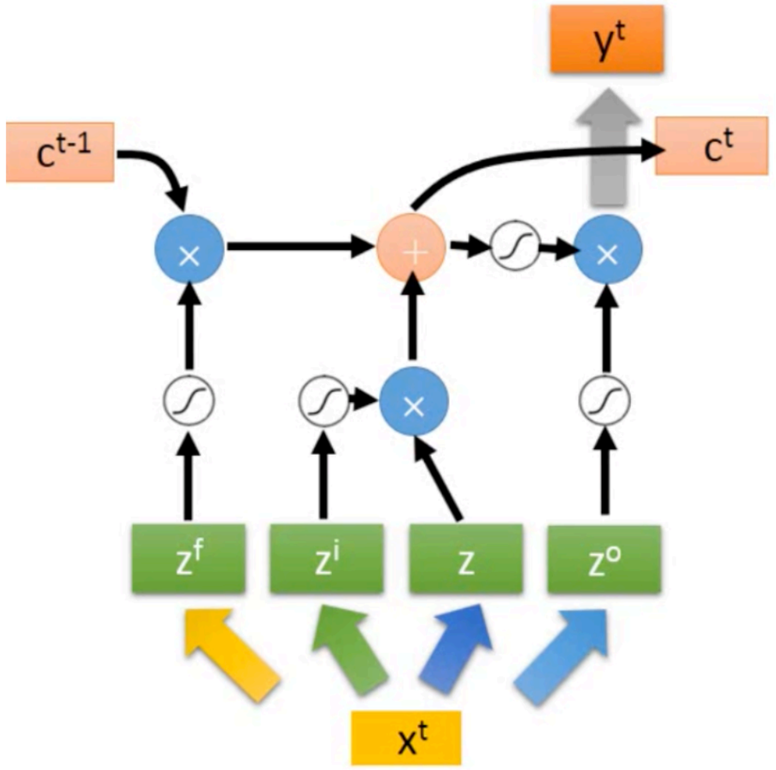
4 times “usual number of” parameters.



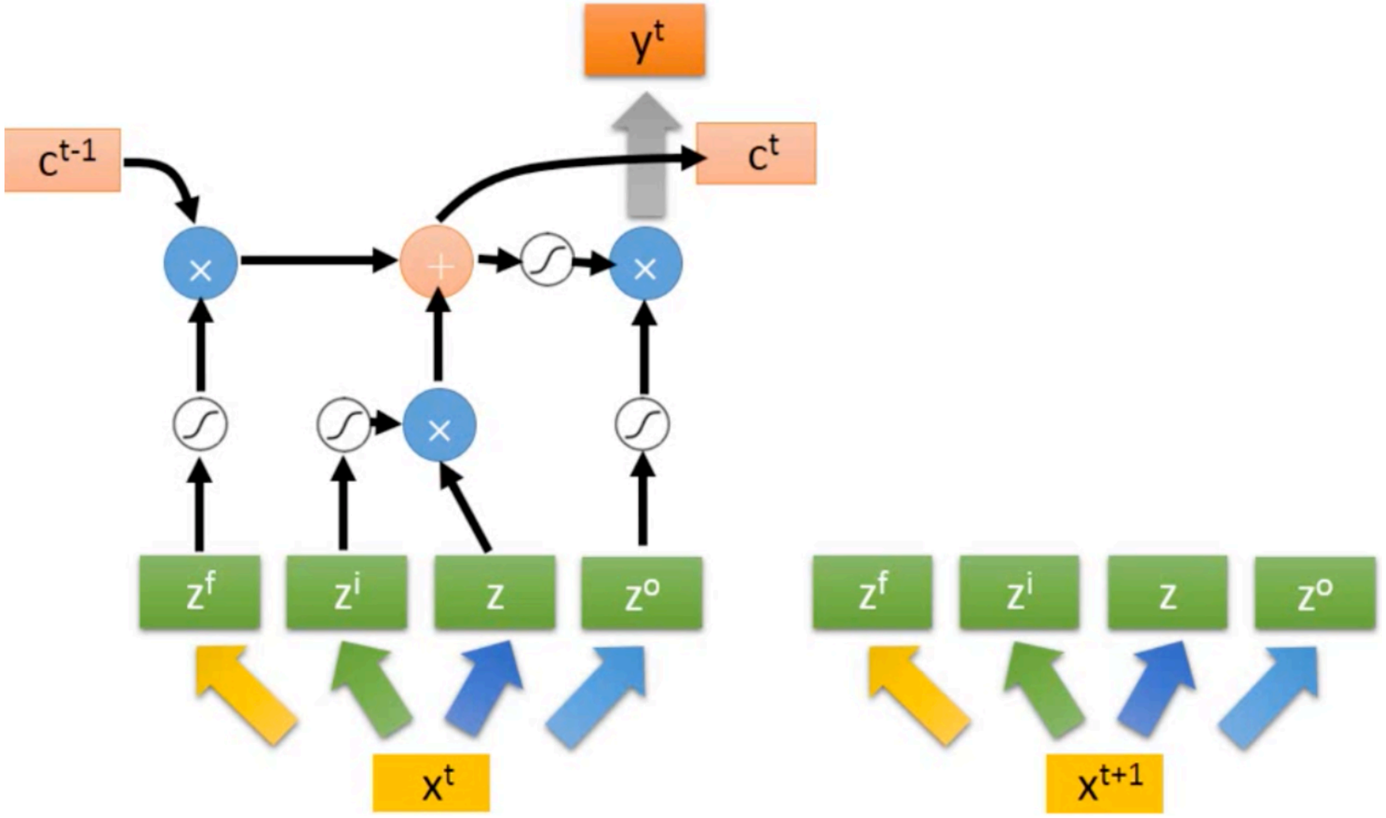
LSTM as an RNN layer



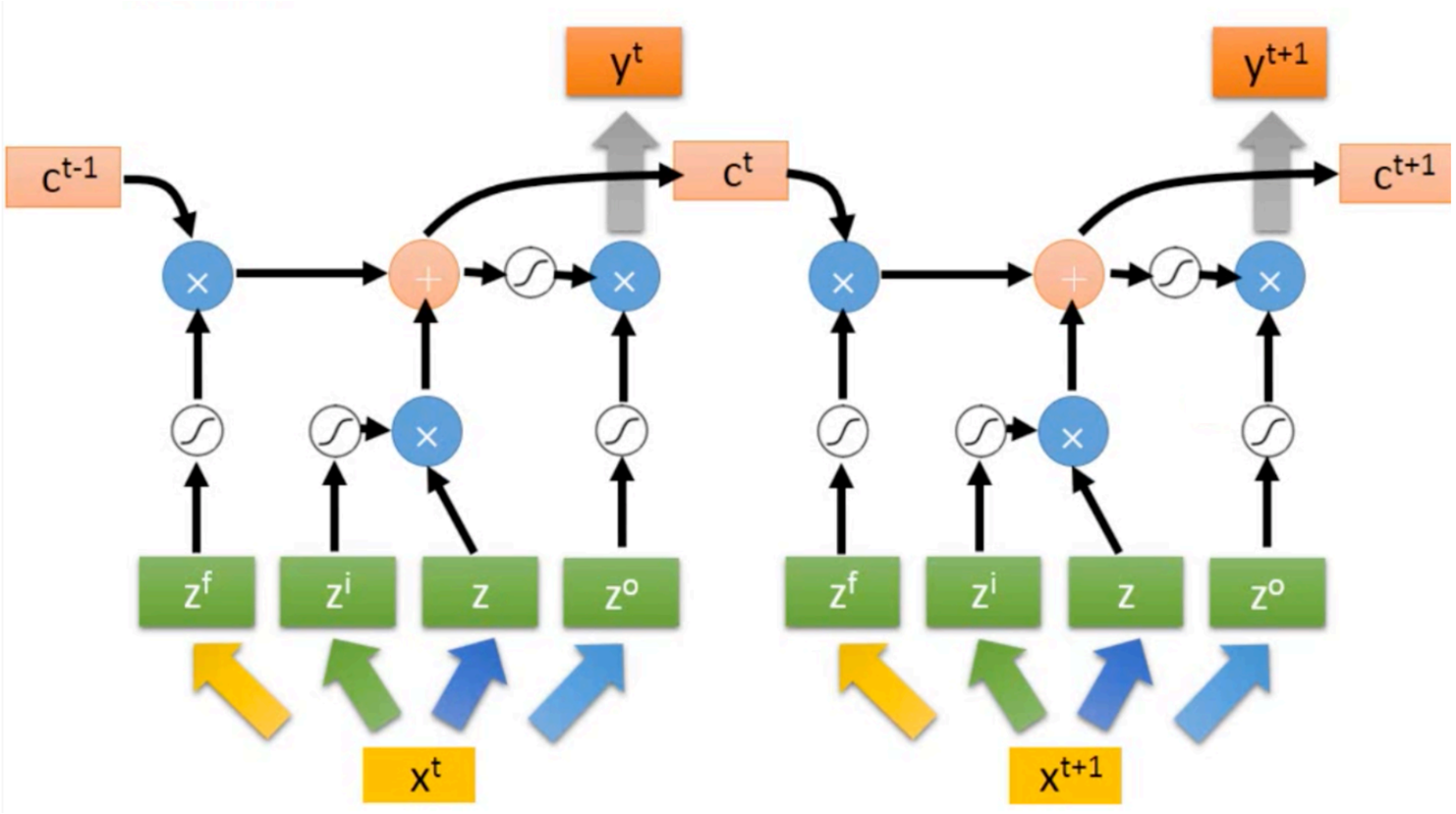
LSTM as an RNN layer



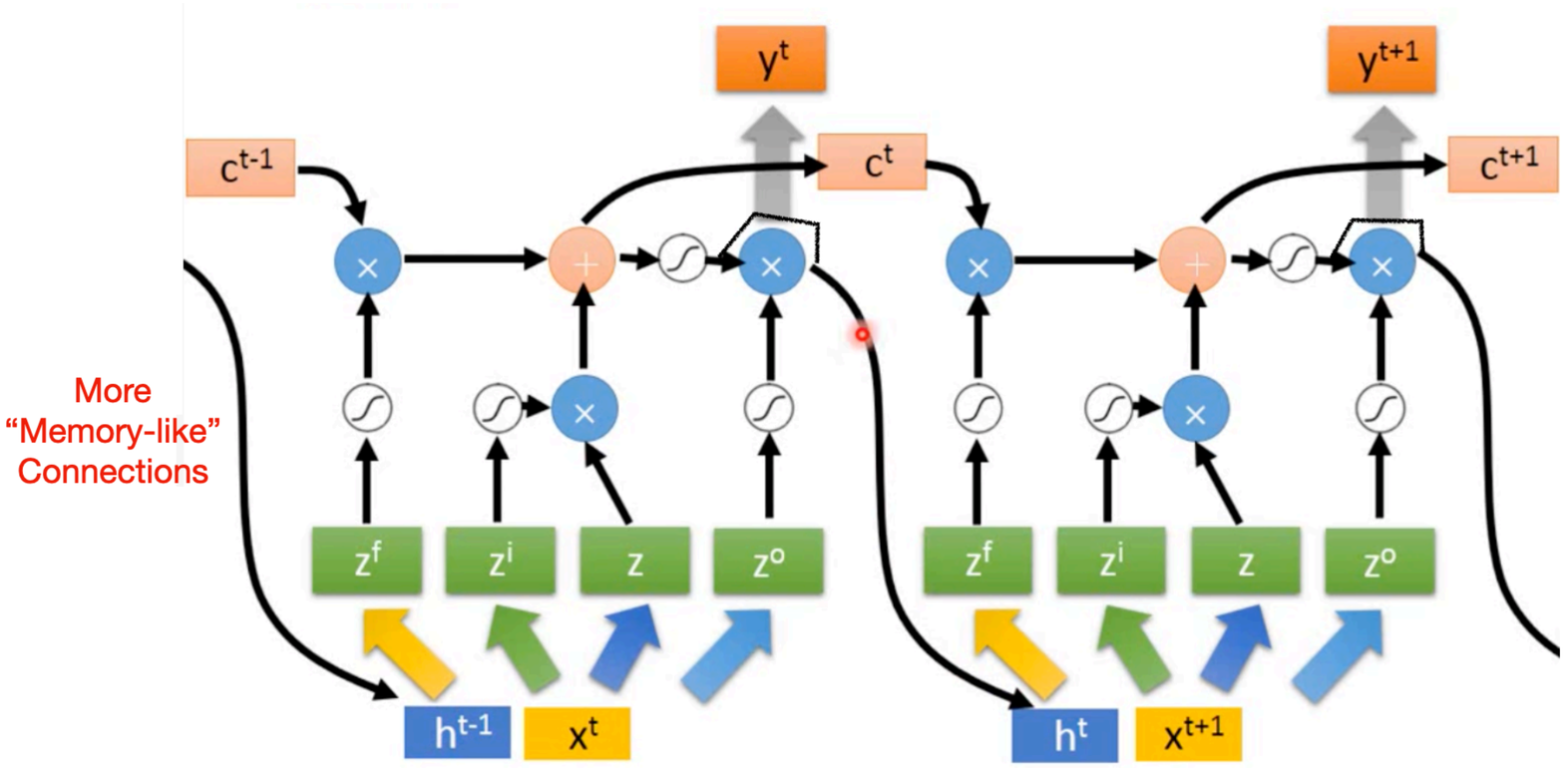
LSTM as an RNN layer



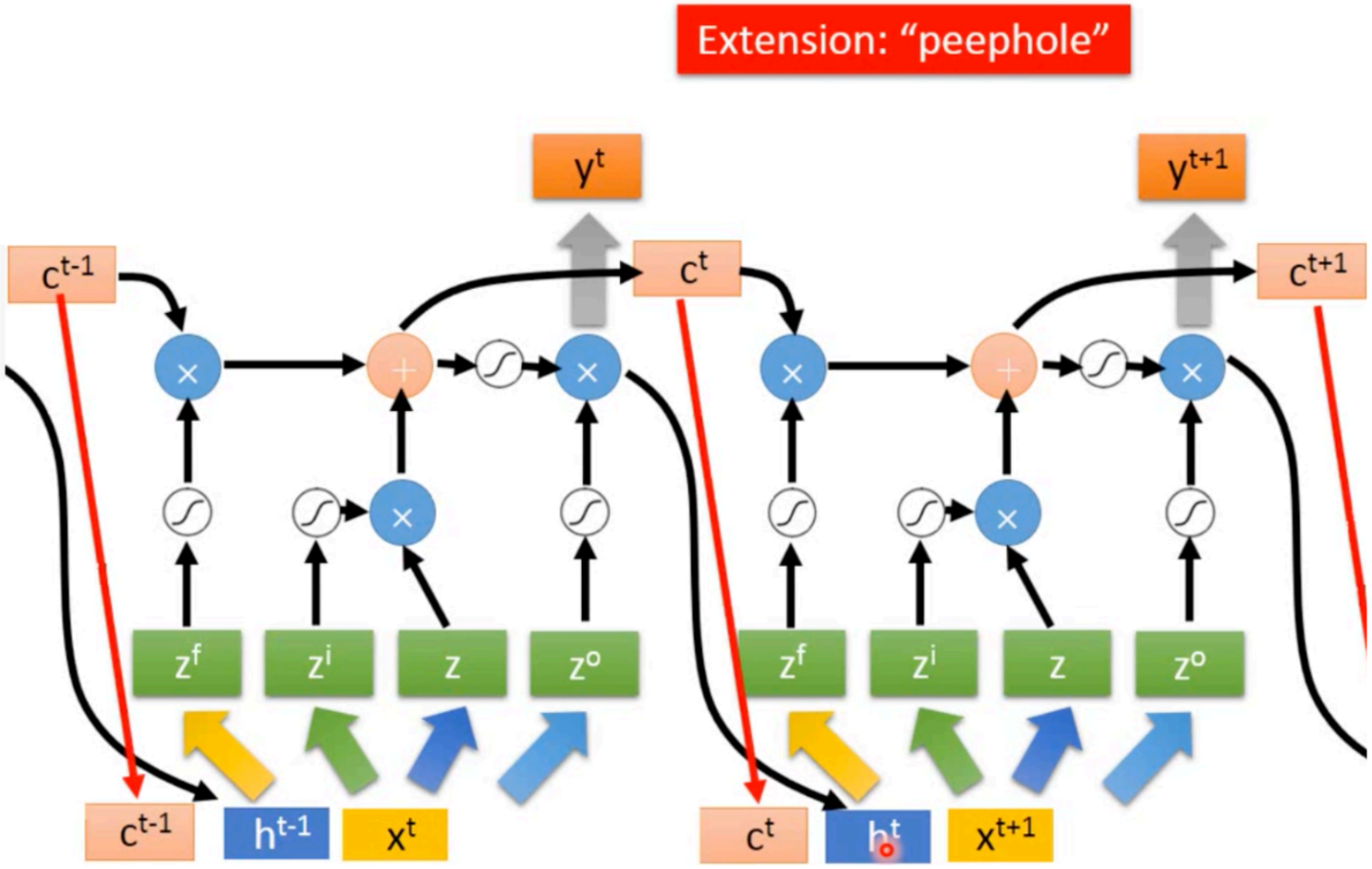
LSTM as an RNN layer



LSTM as an RNN layer

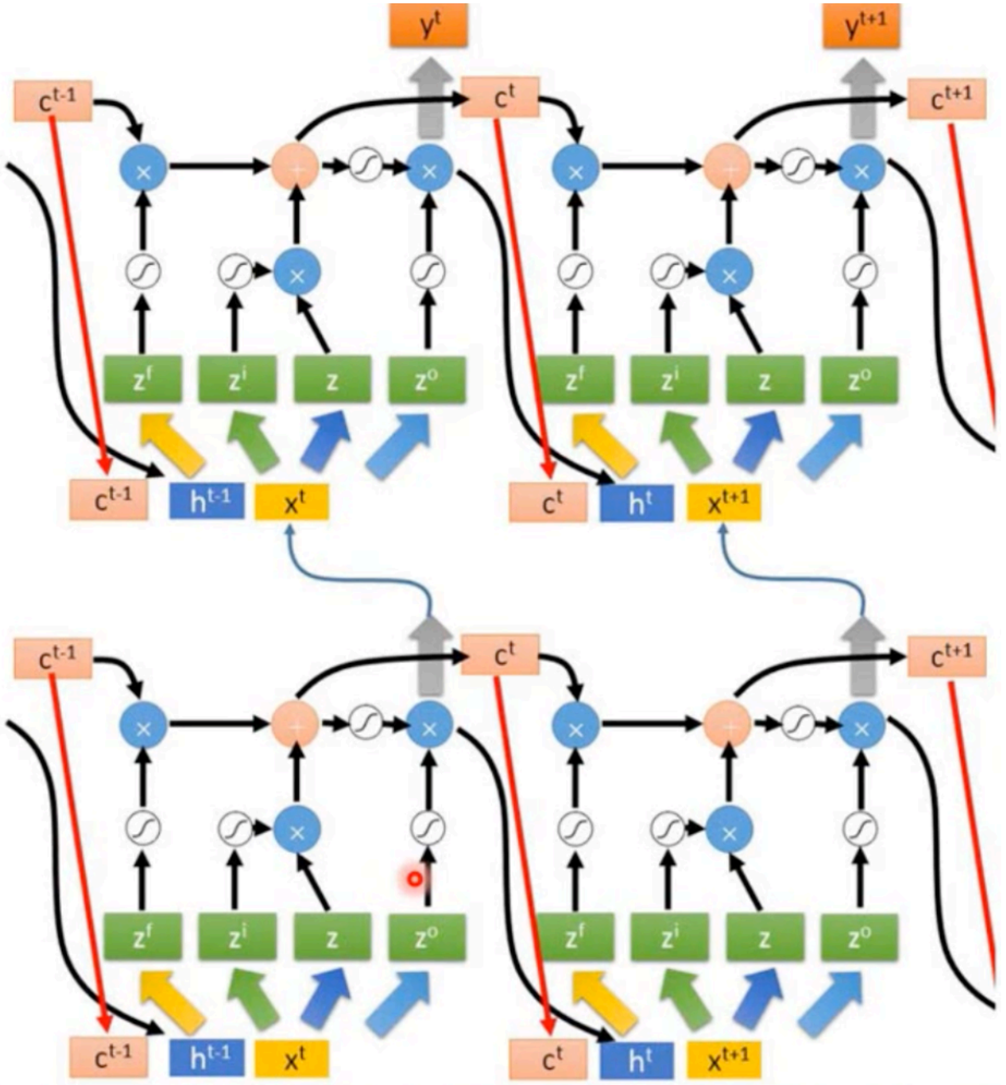


LSTM as an RNN layer



Stack multiple LSTM layers

Multiple-layer LSTM



https://img.komicolle.org/2015-09-20/src/14426967627131.g

Quiz questions:

1. What is a recurrent neural network?

2. What is the use of the “gates” in LSTM?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Different types of tasks for Timeseries

By far, the most common timeseries-related task is *forecasting*: predicting what will happen next in a series.

other things you can do with timeseries:

- *Classification*—Assign one or more categorical labels to a timeseries. For instance, given the timeseries of the activity of a visitor on a website, classify whether the visitor is a bot or a human.
- *Event detection*—Identify the occurrence of a specific expected event within a continuous data stream. A particularly useful application is "hotword detection," where a model monitors an audio stream and detects utterances like "Ok Google" or "Hey Alexa."
- *Anomaly detection*—Detect anything unusual happening within a continuous datastream. Unusual activity on your corporate network? Might be an attacker. Unusual readings on a manufacturing line? Time for a human to go take a look. Anomaly detection is typically done via unsupervised learning, because you often don't know what kind of anomaly you're looking for, so you can't train on specific anomaly examples.

Example task for Regression: Temperature Forecasting

Throughout this chapter, all of our code examples will target a single problem: predicting the temperature 24 hours in the future, given a timeseries of hourly measurements of quantities such as atmospheric pressure and humidity, recorded over the recent past by a set of sensors on the roof of a building. As you will see, it's a fairly challenging problem!

Temperature Forecasting: Get the dataset

We'll work with a weather timeseries dataset recorded at the weather station at the Max Planck Institute for Biogeochemistry in Jena, Germany.[1] In this dataset, 14 different quantities (such as temperature, pressure, humidity, wind direction, and so on) were recorded every 10 minutes over several years. The original data goes back to 2003, but the subset of the data we'll download is limited to 2009–2016.

Let's start by downloading and uncompressing the data:

```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
!unzip jena_climate_2009_2016.csv.zip
```

Temperature Forecasting: Get the dataset

Now let's look at the data.

Listing 10.1 Inspecting the data of the Jena weather dataset

```
import os
fname = os.path.join("jena_climate_2009_2016.csv")

with open(fname) as f:
    data = f.read()

lines = data.split("\n")
header = lines[0].split(",")
lines = lines[1:]
print(header)
print(len(lines))
```

Temperature Forecasting: Get the dataset

This outputs a count of 420,551 lines of data (each line is a timestep: a record of a date and 14 weather-related values), as well as the following header:

```
["Date Time",
 "p (mbar)",
 "T (degC)",
 "Tpot (K)",
 "Tdew (degC)",
 "rh (%)",
 "VPmax (mbar)",
 "VPact (mbar)",
 "VPdef (mbar)",
 "sh (g/kg)",
 "H2OC (mmol/mol)",
 "rho (g/m**3)",
 "wv (m/s)",
 "max. wv (m/s)",
 "wd (deg)"]
```

Temperature Forecasting: Get the dataset

Now, convert all 420,551 lines of data into NumPy arrays: one array for the temperature (in degrees Celsius), and another one for the rest of the data—the features we will use to predict future temperatures. Note that we discard the "Date Time" column.

Listing 10.2 Parsing the data

```
import numpy as np
temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i, :] = values[:]
```

We store column 1 in the "temperature" array.

We store all columns (including the temperature) in the "raw_data" array.

Temperature Forecasting: Get the dataset

Figure 10.1 shows the plot of temperature (in degrees Celsius) over time. On this plot, you can clearly see the yearly periodicity of temperature—the data spans 8 years.

Listing 10.3 Plotting the temperature timeseries

```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

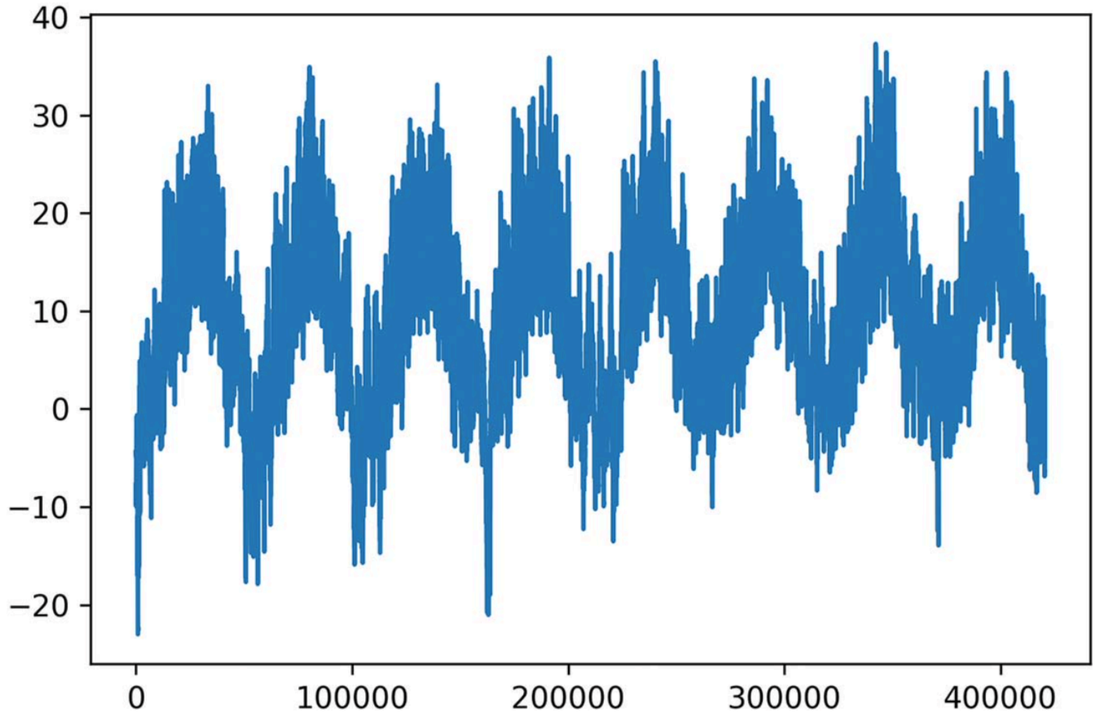


Figure 10.1 Temperature over the full temporal range of the dataset (°C)

Temperature Forecasting: Get the dataset

Figure 10.2 shows a more narrow plot of the first 10 days of temperature data. Because the data is recorded every 10 minutes, you get 24 × 6 = 144 data points per day.

Listing 10.4 Plotting the first 10 days of the temperature timeseries

```
plt.plot(range(1440), temperature[:1440])
```

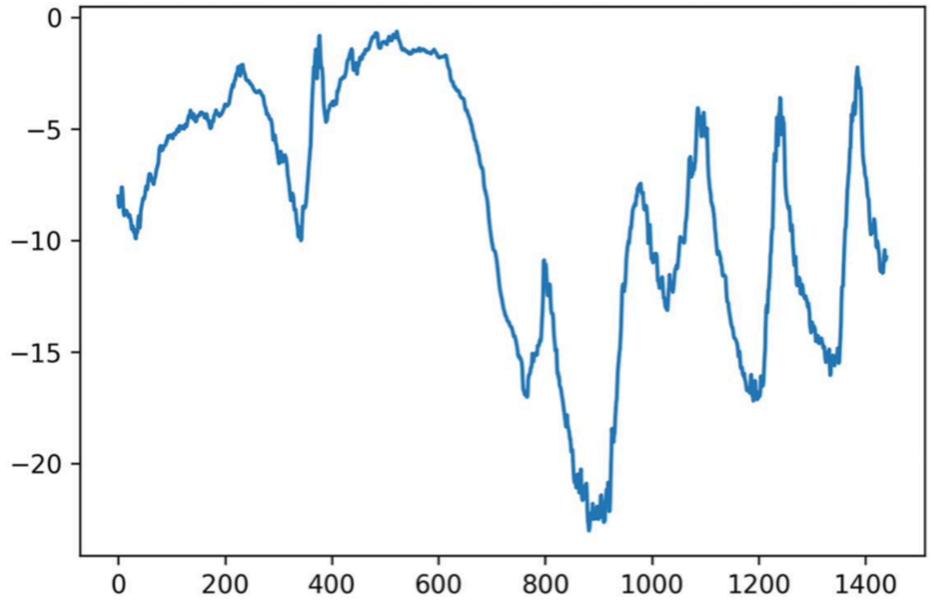
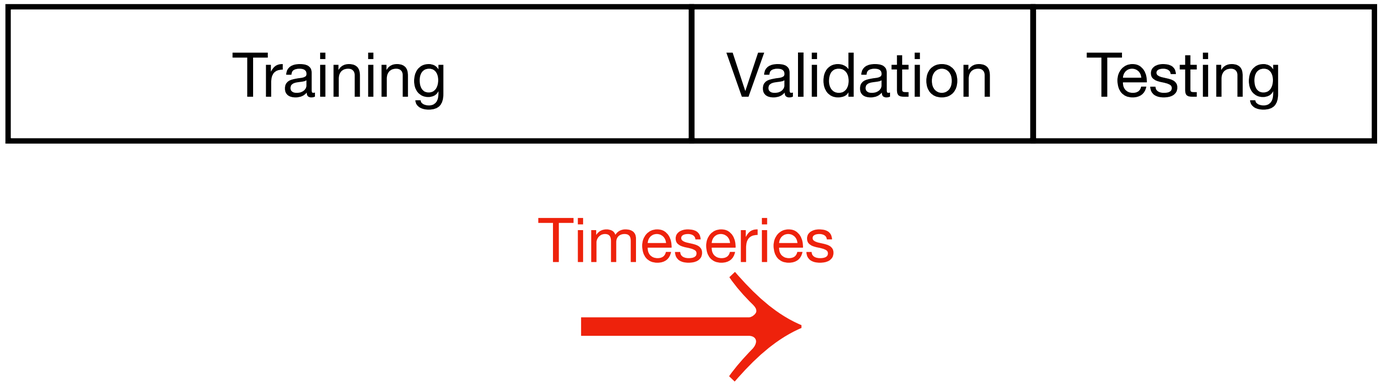


Figure 10.2 Temperature over the first 10 days of the dataset (°C)

On this plot, you can see daily periodicity, especially for the last 4 days.

Temperature Forecasting: Prepare data

In all our experiments, we'll use the first 50% of the data for training, the following 25% for validation, and the last 25% for testing. When working with timeseries data, it's important to use validation and test data that is more recent than the training data, because you're trying to predict the future given the past, not the reverse, and your validation/test splits should reflect that. Some problems happen to be con-



Temperature Forecasting: Prepare data

Listing 10.5 Computing the number of samples we'll use for each data split

```
>>> num_train_samples = int(0.5 * len(raw_data))
>>> num_val_samples = int(0.25 * len(raw_data))
>>> num_test_samples = len(raw_data) - num_train_samples - num_val_samples
```

```
>>> print("num_train_samples:", num_train_samples)
>>> print("num_val_samples:", num_val_samples)
>>> print("num_test_samples:", num_test_samples)
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

Temperature Forecasting: Prepare data

The exact formulation of the problem will be as follows: given data covering the previous five days and sampled once per hour, can we predict the temperature in 24 hours?

First, let's preprocess the data to a format a neural network can ingest. This is easy: the data is already numerical, so you don't need to do any vectorization. But each timeseries in the data is on a different scale (for example, atmospheric pressure, measured in mbar, is around 1,000, while H2OC, measured in millimoles per mole, is around 3). We'll normalize each timeseries independently so that they all take small values on a similar scale. We're going to use the first 210,225 timesteps as training data, so we'll compute the mean and standard deviation only on this fraction of the data.

Listing 10.6 Normalizing the data

```
mean = raw_data[:num_train_samples].mean(axis=0)
raw_data -= mean
std = raw_data[:num_train_samples].std(axis=0)
raw_data /= std
```

Temperature Forecasting: Prepare data

Next, let's create a `Dataset` object that yields batches of data from the past five days along with a target temperature 24 hours in the future. Because the samples in the dataset are highly redundant (sample *N* and sample *N* + `1` will have most of their timesteps in common), it would be wasteful to explicitly allocate memory for every sample. Instead, we'll generate the samples on the fly while only keeping in memory the original `raw_data` and `temperature` arrays, and nothing more.

We could easily write a Python generator to do this, but there's a built-in dataset utility in Keras that does just that (`timeseries_dataset_from_array()`), so we can save ourselves some work by using it. You can generally use it for any kind of timeseries forecasting task.

Temperature Forecasting: Prepare data

Understanding `timeseries_dataset_from_array()`

To understand what `timeseries_dataset_from_array()` does, let's look at a simple example. The general idea is that you provide an array of timeseries data (the `data` argument), and `timeseries_dataset_from_array()` gives you windows extracted from the original timeseries (we'll call them "sequences").

For example, if you use `data = [0 1 2 3 4 5 6]` and `sequence_length=3`, then `timeseries_dataset_from_array()` will generate the following samples: `[0 1 2]`, `[1 2 3]`, `[2 3 4]`, `[3 4 5]`, `[4 5 6]`.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

0 1 2 3 4 5 6 7 8 9

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

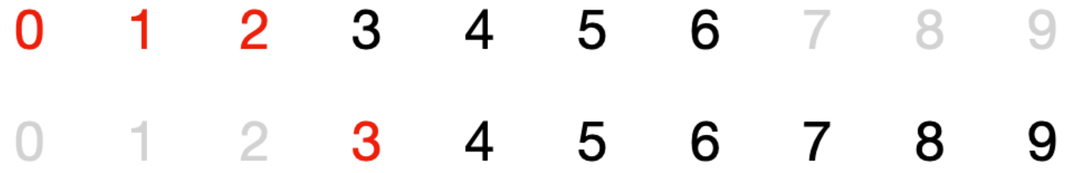
```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.



Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

You can also pass a `targets` argument (an array) to `timeseries_dataset_from_array()`. The first entry of the `targets` array should match the desired target for the first sequence that will be generated from the `data` array. So if you're doing timeseries forecasting, `targets` should be the same array as `data`, offset by some amount.

For instance, with `data = [0 1 2 3 4 5 6 ...]` and `sequence_length=3`, you could create a dataset to predict the next step in the series by passing `targets = [3 4 5 6 ...]`. Let's try it:

```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

Generate an array of sorted integers from 0 to 9.

The sequences we generate will be sampled from [0 1 2 3 4 5 6].

The target for the sequence that starts at data[N] will be data[N + 3].

The sequences will be 3 steps long.

Temperature Forecasting: Prepare data

```
for inputs, targets in dummy_dataset:
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

The sequences will be batched in batches of size 2.

This bit of code prints the following results:

```
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
```

Temperature Forecasting: Prepare data

We'll use `timeseries_dataset_from_array()` to instantiate three datasets: one for training, one for validation, and one for testing.

We'll use the following parameter values:

- `sampling_rate = 6`—Observations will be sampled at one data point per hour: we will only keep one data point out of 6.
- `sequence_length = 120`—Observations will go back 5 days (120 hours).
- `delay = sampling_rate * (sequence_length + 24 - 1)`—The target for a sequence will be the temperature 24 hours after the end of the sequence.

When making the training dataset, we'll pass `start_index = 0` and `end_index = num_train_samples` to only use the first 50% of the data. For the validation dataset, we'll pass `start_index = num_train_samples` and `end_index = num_train_samples + num_val_samples` to use the next 25% of the data. Finally, for the test dataset, we'll pass `start_index = num_train_samples + num_val_samples` to use the remaining samples.

Temperature Forecasting: Prepare data

Listing 10.7 Instantiating datasets for training, validation, and testing

```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256

train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=0,
    end_index=num_train_samples)
```

Temperature Forecasting: Prepare data

```
val_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples,
    end_index=num_train_samples + num_val_samples)

test_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples + num_val_samples)
```

Temperature Forecasting: Prepare data

Each dataset yields a tuple `(samples, targets)`, where `samples` is a batch of 256 samples, each containing 120 consecutive hours of input data, and `targets` is the corresponding array of 256 target temperatures. Note that the samples are randomly shuffled, so two consecutive sequences in a batch (like `samples[0]` and `samples[1]`) aren't necessarily temporally close.

Listing 10.8 Inspecting the output of one of our datasets

```
>>> for samples, targets in train_dataset:
>>>     print("samples shape:", samples.shape)
>>>     print("targets shape:", targets.shape)
>>>     break
samples shape: (256, 120, 14)
targets shape: (256,)
```

Quiz question:

1. How to prepare a dataset for sequential data?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Try 1: A common-sense, non-machine learning baseline

Idea: use current temperature as the prediction for temperature in 24 hours.

Note: Such common-sense baselines can be useful when you're approaching a new problem for which there is no known solution (yet). A classic example is that of unbalanced classification tasks, where some classes are much more common than others. If your dataset contains 90% instances of class A and 10% instances of class B, then a common-sense approach to the classification task is to always predict "A" when presented with a new sample. Such a classifier is 90% accurate overall, and any learning-based approach should therefore beat this 90% score in order to demonstrate usefulness. Sometimes, such elementary baselines can prove surprisingly hard to beat.

Try 1: A common-sense, non-machine learning baseline

Let's evaluate this approach, using the mean absolute error (MAE) metric, defined as follows:

```
np.mean(np.abs(preds - targets))
```

Here's the evaluation loop.

Baseline performance:

MAE = 2.44 degrees Celsius for validation data
MAE = 2.62 degrees Celsius for test data

Listing 10.9 Computing the common-sense baseline MAE

```
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples, targets in dataset:
        preds = samples[:, -1, 1] * std[1] + mean[1]
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

print(f"Validation MAE: {evaluate_naive_method(val_dataset):.2f}")
print(f"Test MAE: {evaluate_naive_method(test_dataset):.2f}")
```

The temperature feature is at column 1, so samples[:, -1, 1] is the last temperature measurement in the input sequence. Recall that we normalized our features, so to retrieve a temperature in degrees Celsius, we need to un-normalize it by multiplying it by the standard deviation and adding back the mean.

Quiz question:

1. Why does the non-machine learning method here make sense?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Try 2: Fully connected neural network

Listing 10.10 Training and evaluating a densely connected model

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.

Try 2: Fully connected neural network

Listing 10.10 Training and evaluating a densely connected model

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

120×14 inputs

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.

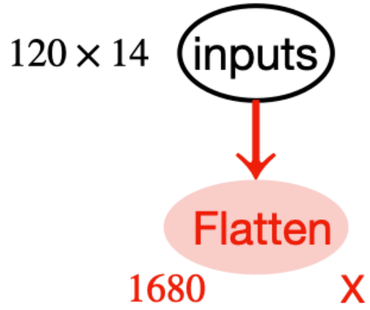
Try 2: Fully connected neural network

Listing 10.10 Training and evaluating a densely connected model

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.



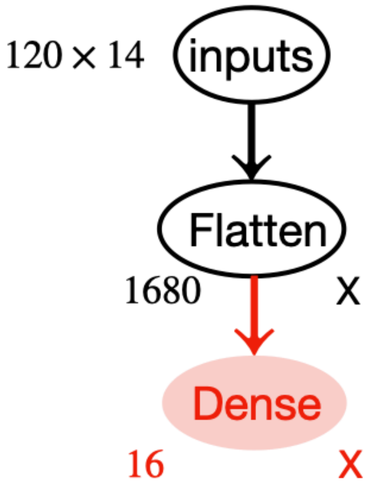
Try 2: Fully connected neural network

Listing 10.10 Training and evaluating a densely connected model

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.



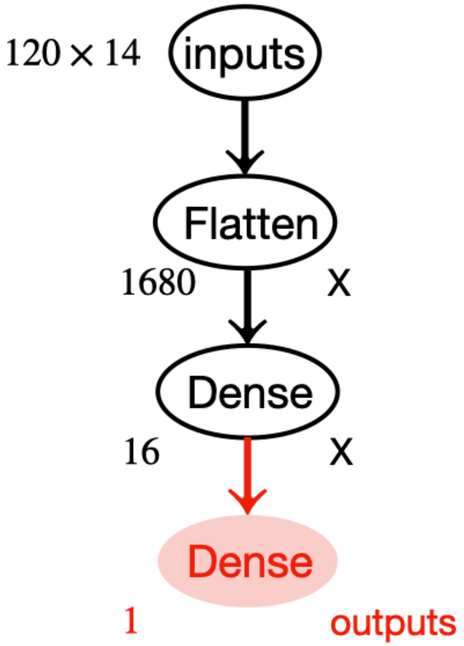
Try 2: Fully connected neural network

Listing 10.10 Training and evaluating a densely connected model

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.



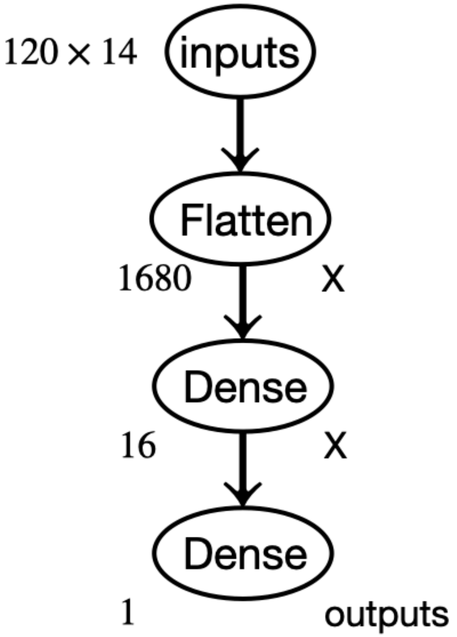
Try 2: Fully connected neural network

Listing 10.10 Training and evaluating a densely connected model

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

We use a callback to save the best-performing model.

Reload the best model and evaluate it on the test data.



Try 2: Fully connected neural network

Let's display the loss curves for validation and training (see figure 10.3).

Listing 10.11 Plotting results

```
import matplotlib.pyplot as plt
loss = history.history["mae"]
val_loss = history.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "bo", label="Training MAE")
plt.plot(epochs, val_loss, "b", label="Validation MAE")
plt.title("Training and validation MAE")
plt.legend()
plt.show()
```

Worse than Baseline performance:

MAE = 2.44 degrees Celsius for validation data
MAE = 2.62 degrees Celsius for test data

Fully connected network:

MAE = 2.52 degrees Celsius for validation data

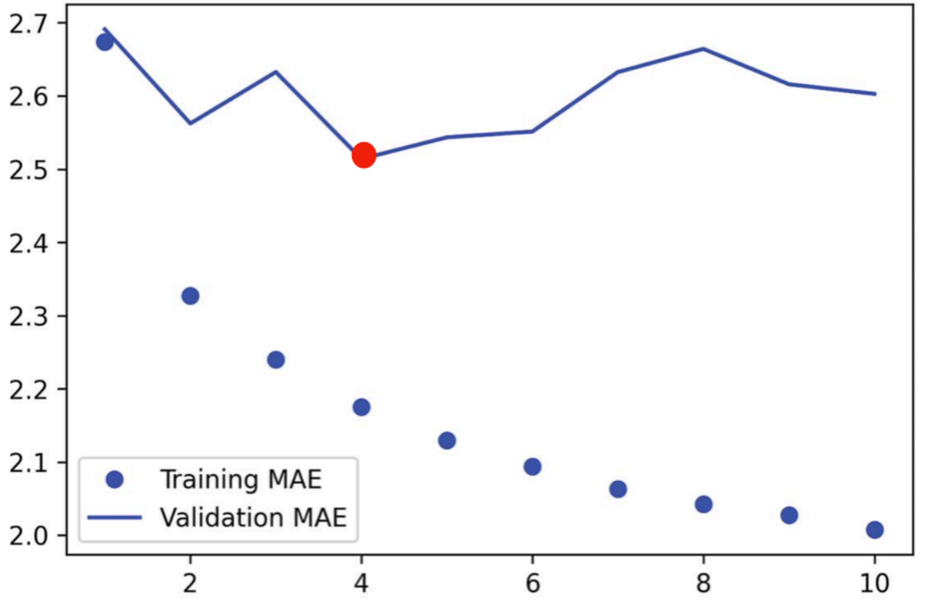


Figure 10.3 Training and validation MAE on the Jena temperature-forecasting task with a simple, densely connected network

Quiz question:

1. Why is not the fully connected network here performing as well as the simple baseline method?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

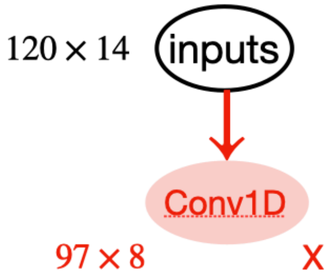
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

120×14 inputs

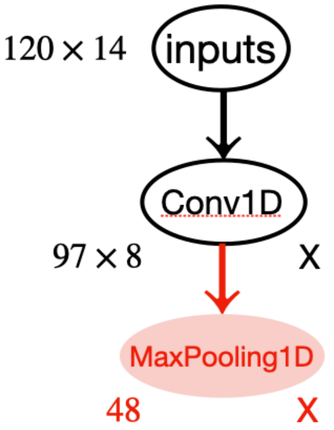
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



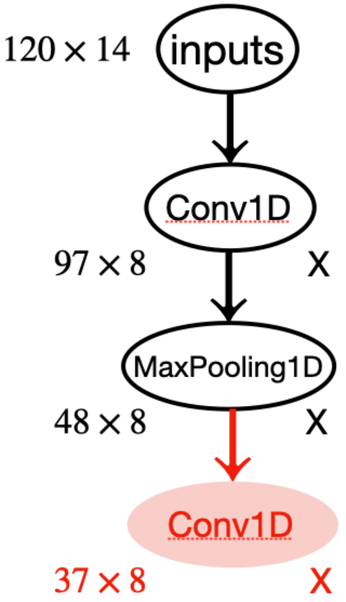
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



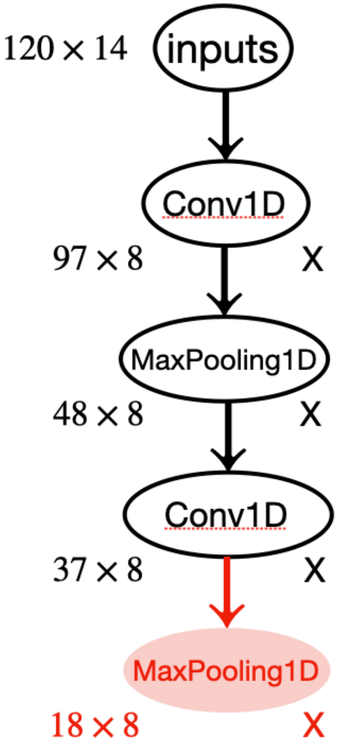
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



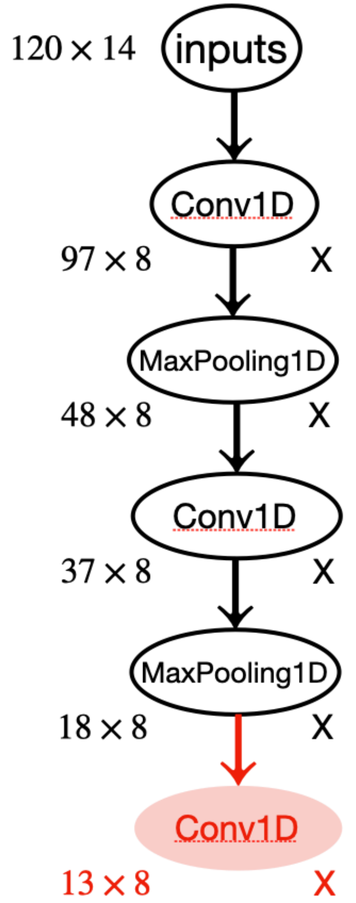
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



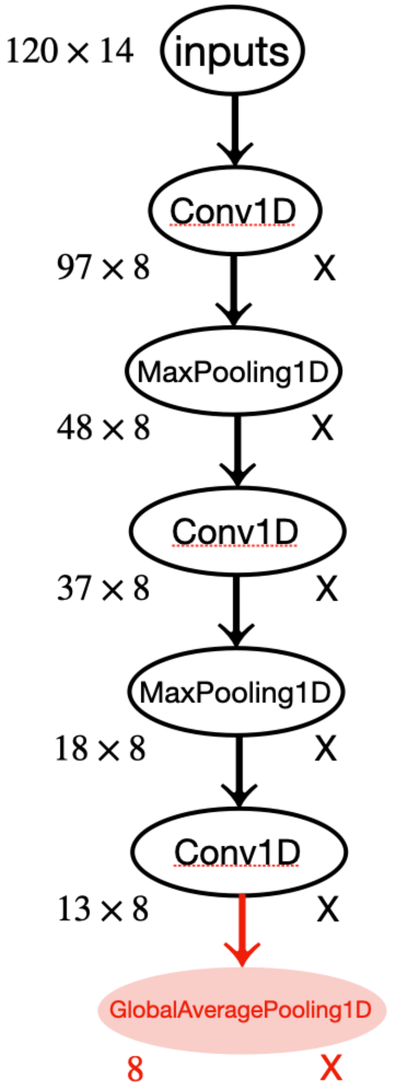
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



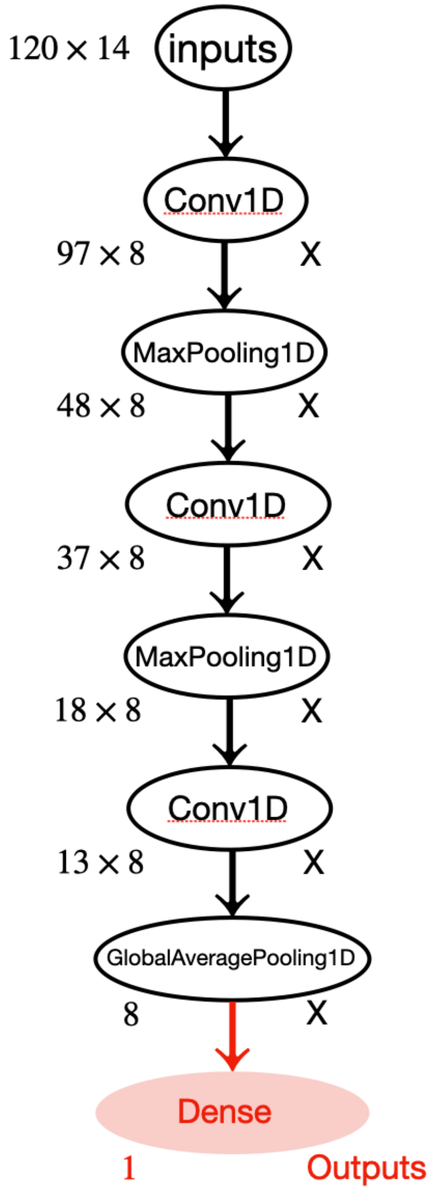
Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



Try 3: 1-d convolutional neural network

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



Try 3: 1-d convolutional neural network

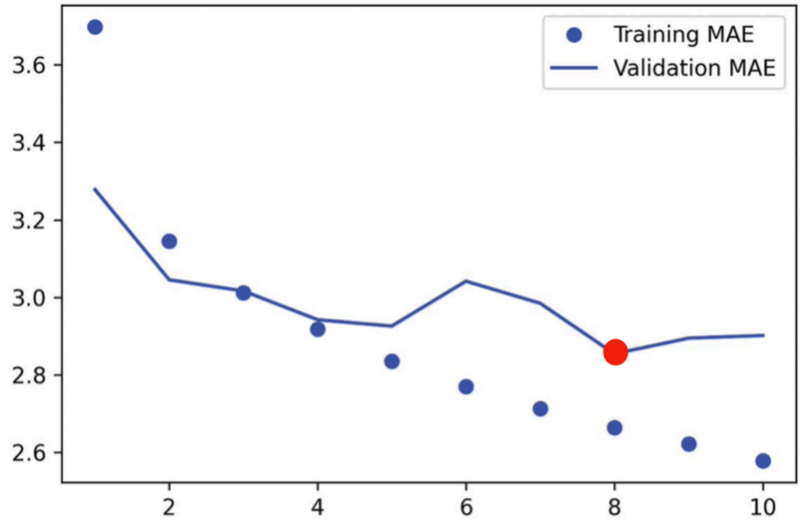


Figure 10.4 Training and validation MAE on the Jena temperature-forecasting task with a 1D convnet

1-d CNN:

MAE = 2.85 degrees Celsius for validation data

Worse than Baseline performance:

MAE = 2.44 degrees Celsius for validation data
MAE = 2.62 degrees Celsius for test data

Worse than Fully connected network:

MAE = 2.52 degrees Celsius for validation data

Quiz questions:

1. How does it make sense to use a 1-d CNN to process the time-series data here?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Try 4: Recurrent Neural Network (RNN)

Sometimes we need the model to read the time-series data sequentially, and remember what it saw before.

There's a family of neural network architectures designed specifically for this use case: recurrent neural networks. Among them, the Long Short Term Memory (LSTM) layer has long been very popular. We'll see in a minute how these models work, but let's start by giving the LSTM layer a try.

Try 4: Recurrent Neural Network (RNN)

Listing 10.12 A simple LSTM-based model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

Try 4: Recurrent Neural Network (RNN)

Listing 10.12 A simple LSTM-based model

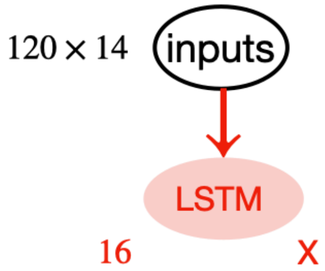
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

120×14 inputs

Try 4: Recurrent Neural Network (RNN)

Listing 10.12 A simple LSTM-based model

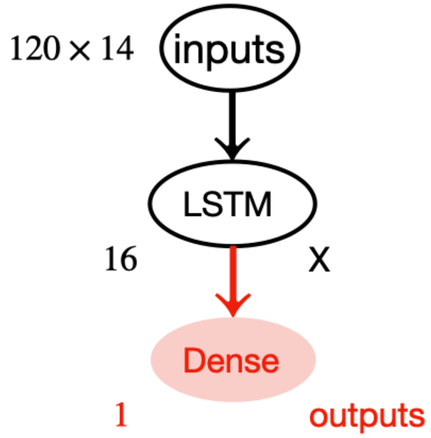
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



Try 4: Recurrent Neural Network (RNN)

Listing 10.12 A simple LSTM-based model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



Try 4: Recurrent Neural Network (RNN)

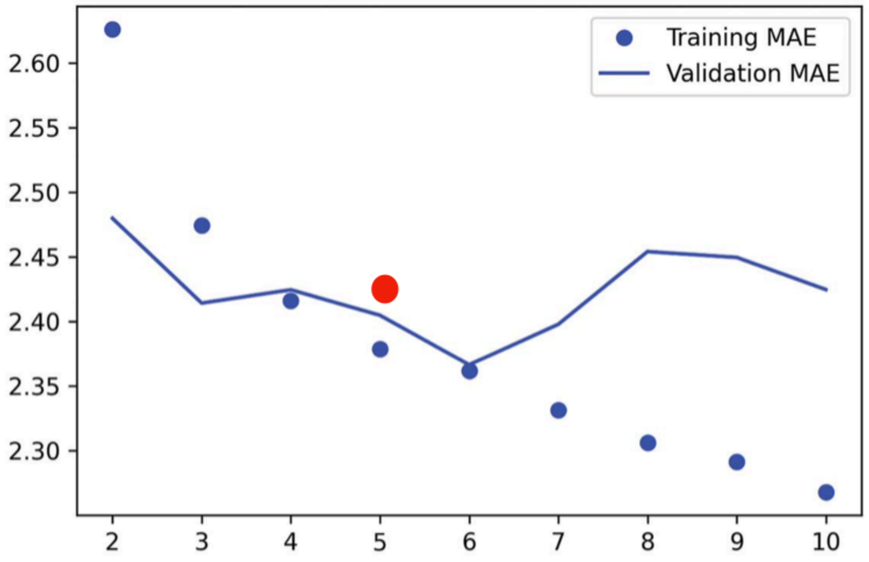


Figure 10.5 Training and validation MAE on the Jena temperature-forecasting task with an LSTM-based model (note that we omit epoch 1 on this graph, because the high training MAE (7.75) at epoch 1 would distort the scale)

LSTM:

MAE = 2.36 degrees Celsius for validation data
MAE = 2.55 degrees Celsius for test data

Better than Baseline performance:
MAE = 2.44 degrees Celsius for validation data
MAE = 2.62 degrees Celsius for test data

Better than Fully connected network:
MAE = 2.52 degrees Celsius for validation data

Better than 1-d CNN:
MAE = 2.85 degrees Celsius for validation data

Quiz questions:

1. Why does RNN work well here?

Roadmap of this lecture:

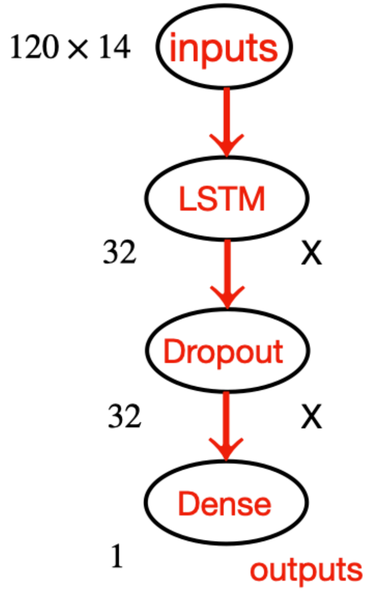
1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

Try 5. LSTM with recurrent dropout

Listing 10.22 Training and evaluating a dropout-regularized LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=50,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

To regularize the Dense layer, we also add a Dropout layer after the LSTM.



Try 5. LSTM with recurrent dropout

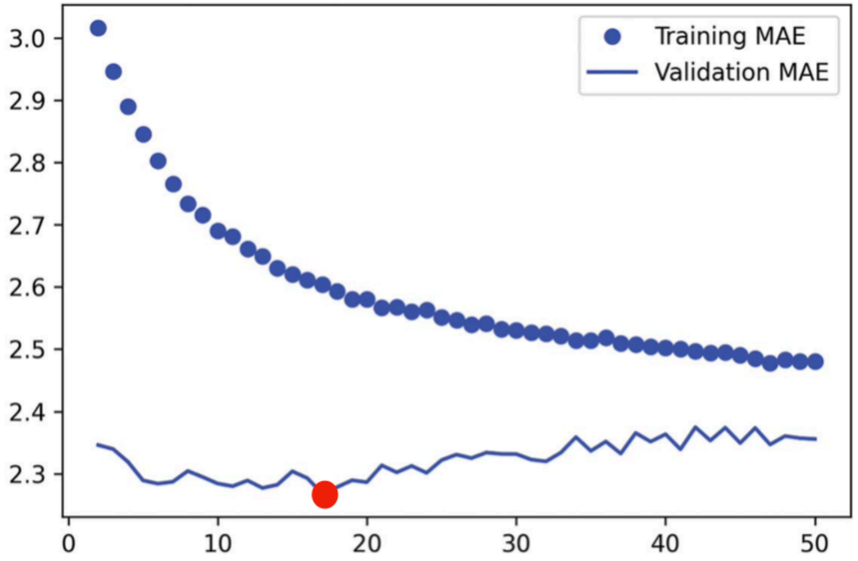


Figure 10.11 Training and validation loss on the Jena temperature-forecasting task with a dropout-regularized LSTM

LSTM with recurrent dropout:

MAE = 2.27 degrees Celsius for validation data
MAE = 2.45 degrees Celsius for test data

Better than Baseline performance:
MAE = 2.44 degrees Celsius for validation data
MAE = 2.62 degrees Celsius for test data

Better than Fully connected network:
MAE = 2.52 degrees Celsius for validation data

Better than 1-d CNN:
MAE = 2.85 degrees Celsius for validation data

Better than LSTM:
MAE = 2.36 degrees Celsius for validation data
MAE = 2.55 degrees Celsius for test data

Quiz question:

1. Why does LSTM work well here?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

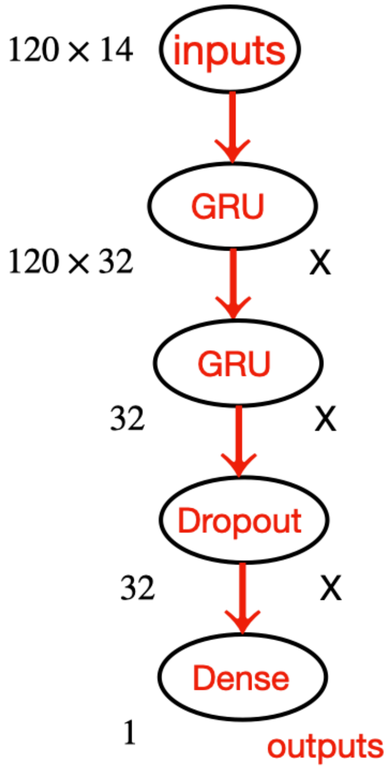
Try 6. Stacking RNN layers

In the following example, we'll try a stack of two dropout-regularized recurrent layers. For a change, we'll use Gated Recurrent Unit (GRU) layers instead of LSTM. GRU is very similar to LSTM—you can think of it as a slightly simpler, streamlined version of the LSTM architecture. It was introduced in 2014 by Cho et al. when recurrent networks were just starting to gain interest anew in the then-tiny research community.[6]

Try 6. Stacking RNN layers

Listing 10.23 Training and evaluating a dropout-regularized, stacked GRU model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_stacked_gru_dropout.keras",
                                    save_best_only=True)
]


model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=50,
                    validation_data=val_dataset,
                    callbacks=callbacks)
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```



Try 6. Stacking RNN layers

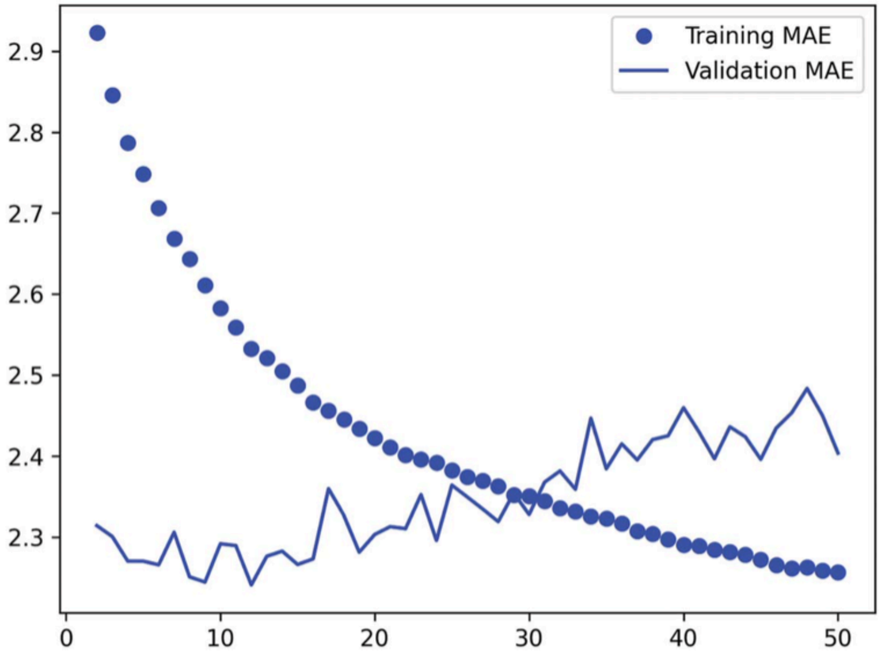


Figure 10.12 Training and validation loss on the Jena temperature-forecasting task with a stacked GRU network

Stacking RNN layers:

MAE = 2.39 degrees Celsius for test data

Better than LSTM with recurrent dropout:
MAE = 2.27 degrees Celsius for validation data
MAE = 2.45 degrees Celsius for test data

Better than Baseline performance:
MAE = 2.44 degrees Celsius for validation data
MAE = 2.62 degrees Celsius for test data

Better than Fully connected network:
MAE = 2.52 degrees Celsius for validation data

Better than 1-d CNN:
MAE = 2.85 degrees Celsius for validation data

Better than LSTM:
MAE = 2.36 degrees Celsius for validation data
MAE = 2.55 degrees Celsius for test data

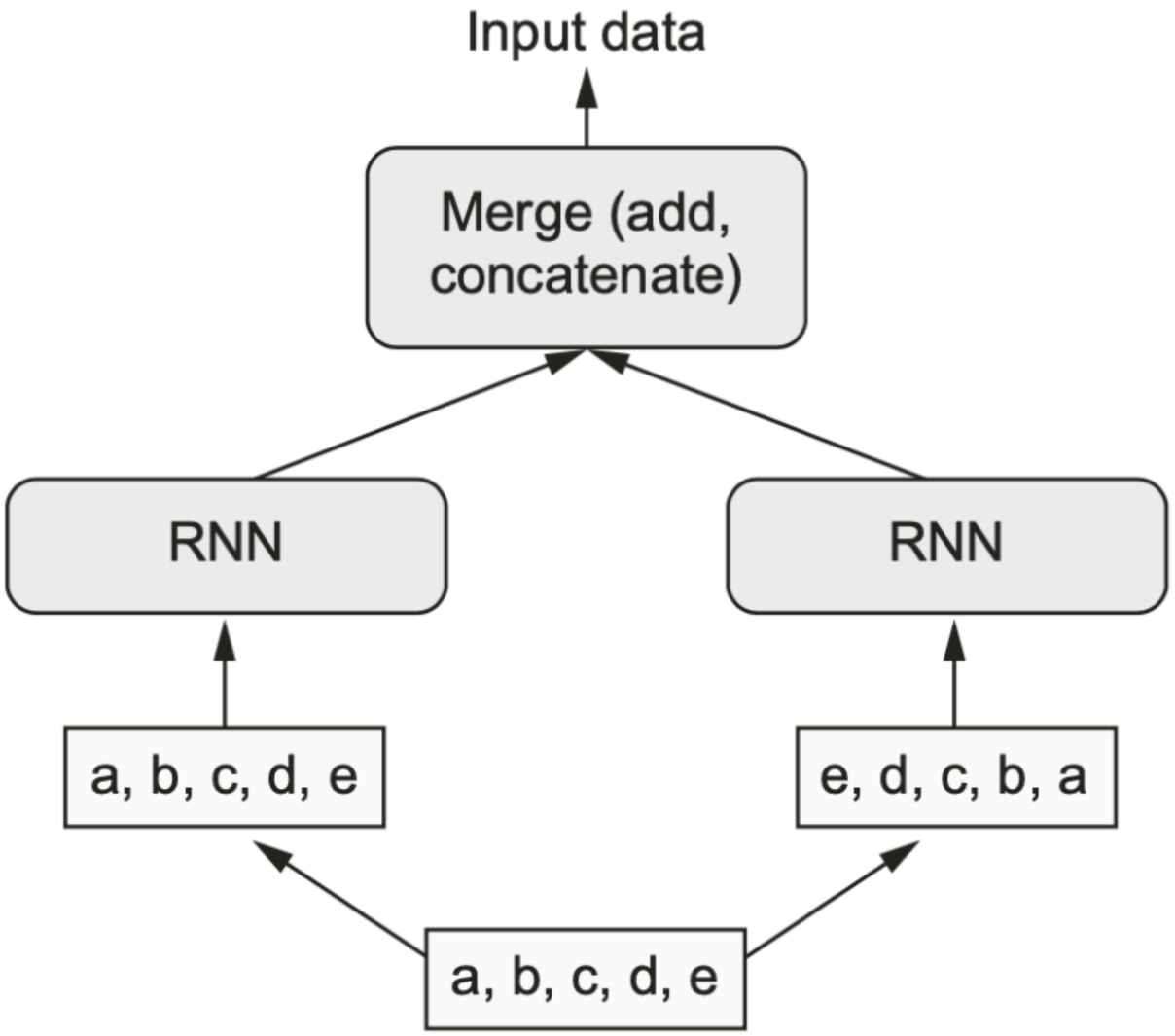
Quiz question:

1. Why does it help to stack multiple layers of RNN?

Roadmap of this lecture:

1. Recurrent neural network (RNN)
2. Use RNN for temperature forecasting
    - 2.1 Try 1: A non-machine learning baseline
    - 2.2 Try 2: Use a fully connected neural network
    - 2.3 Try 3: Use a 1-d CNN
    - 2.4 Try 4: Use an RNN
    - 2.5 Try 5: Use LSTM with recurrent dropout
    - 2.6 Try 6: Stack RNN layers
    - 2.7 Try 7: Use bidirectional RNN

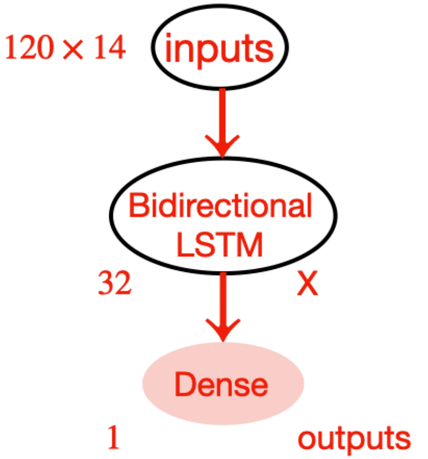
Try 7. Bidirectional RNN



Try 7. Bidirectional RNN

Listing 10.24 Training and evaluating a bidirectional LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset)
```



Performance for temperature prediction:
No better than a simple LSTM.

Quiz question:

1. What are applications where bidirectional RNN would be helpful?