Recurrent Neural Networks (RNN) Time Series Forecasting

Keras & TensorFlow

Hichem Felouat
hichemfel@gmail.com
https://www.linkedin.com/in/hichemfelouat/

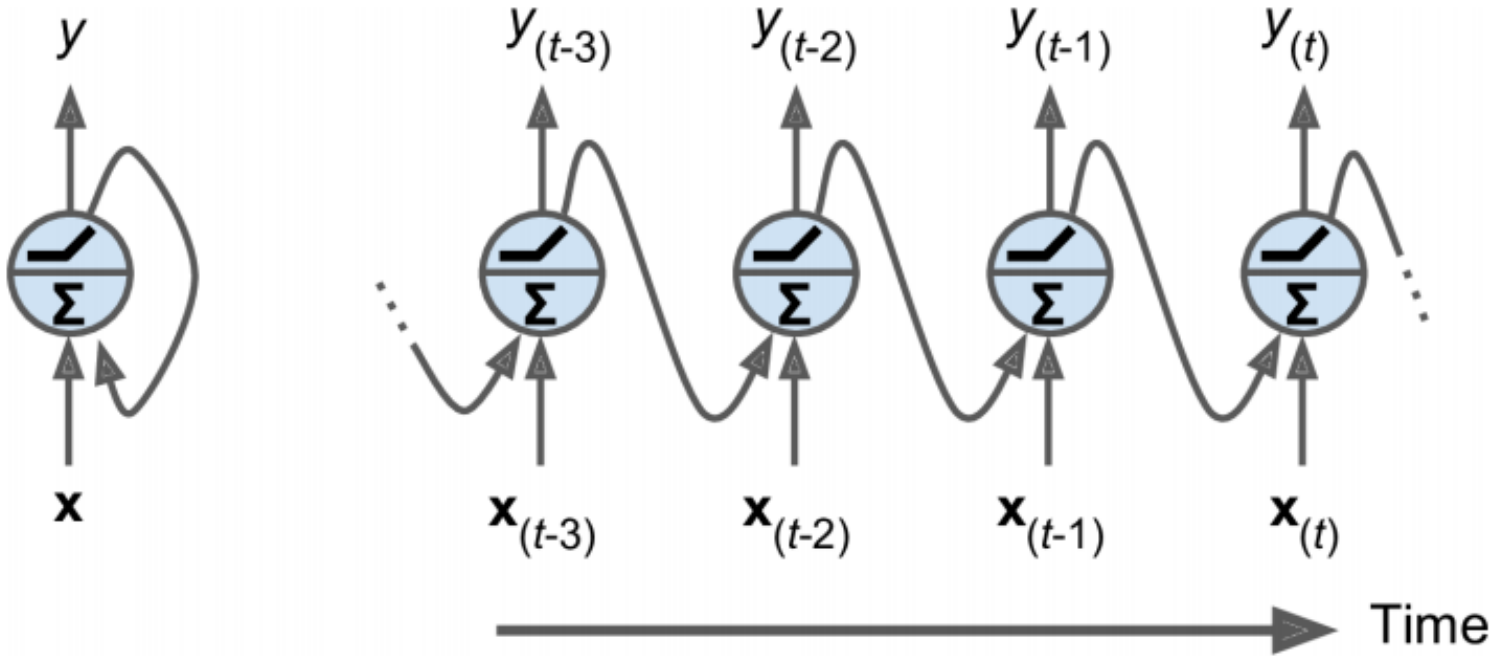
Recurrent Neural Network(RNN)

- RNN are a type of Neural Network where the output from **the previous step are fed as input to the current step**.
- RNN can **predict the future**.
- They can analyze **time-series data** such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents.
- They can work on **sequences of arbitrary lengths**, rather than on fixed-sized inputs like all the nets we have considered so far.
- Useful for **natural language processing** applications such as automatic translation or speech-to-text.

Recurrent Neural Network(RNN)

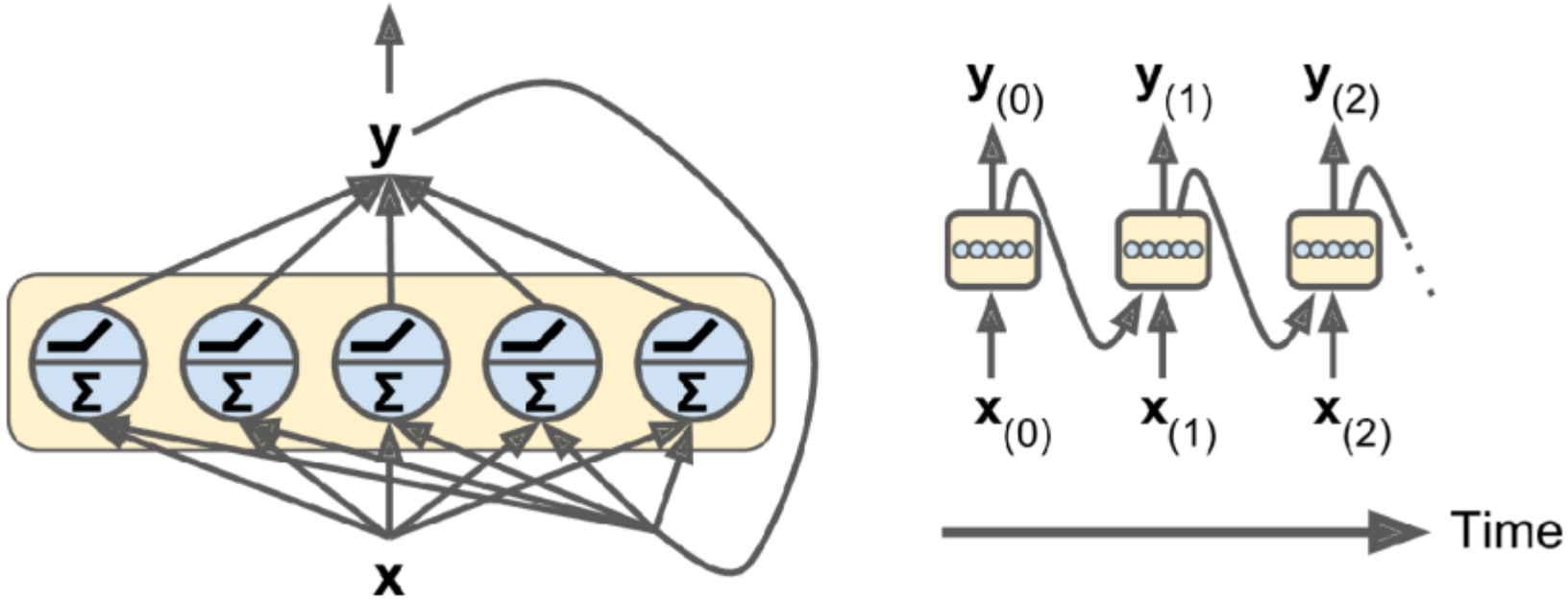
- **The simplest** possible RNN composed of **one neuron** receiving inputs, producing an output, and sending that output back to itself (figure -left).

- We can represent this tiny network against the time axis, as shown in (figure - right). This is called **unrolling the network through time**.



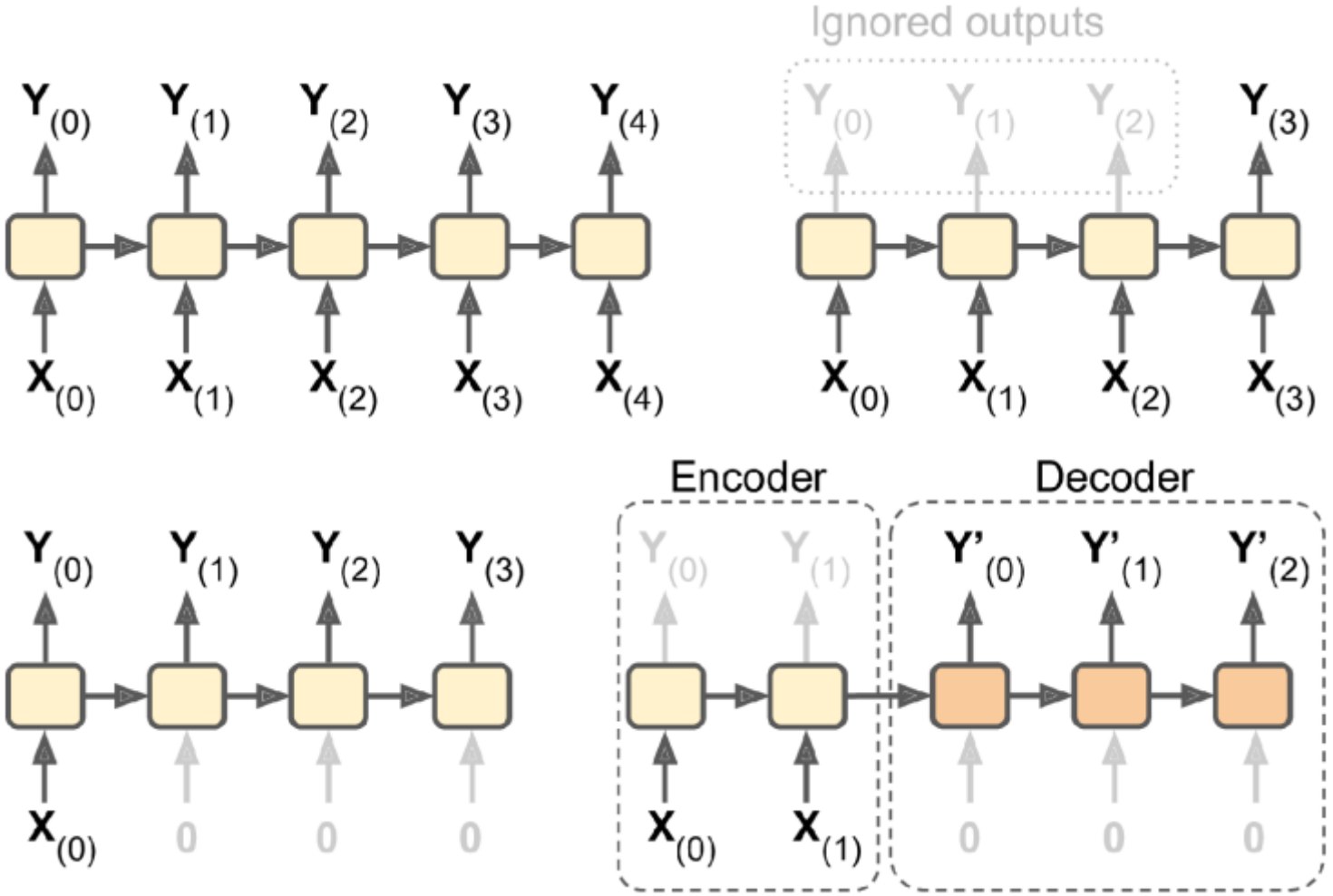
Recurrent Neural Network(RNN)

- You can easily create a **layer** of recurrent neurons. At each time step **t**, every neuron receives both the input vector **x(t)** and the **output vector** from the previous time step **y(t–1)**.



A recurrent neuron (left) unrolled through time (right)

Recurrent Neural Network(RNN)



Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks.

Recurrent Neural Network(RNN)

1. **sequence-to-sequence** network is useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future (i.e., from N – 1 days ago to tomorrow).

2. **sequence-to-vector** network. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from –1 [hate] to +1 [love]).

3. **vector-to-sequence** network. For example, the input could be an image (or the output of a CNN), and the output could be a caption for that image.

4. **Encoder–Decoder**. For example, this could be used for translating a sentence from one language to another.

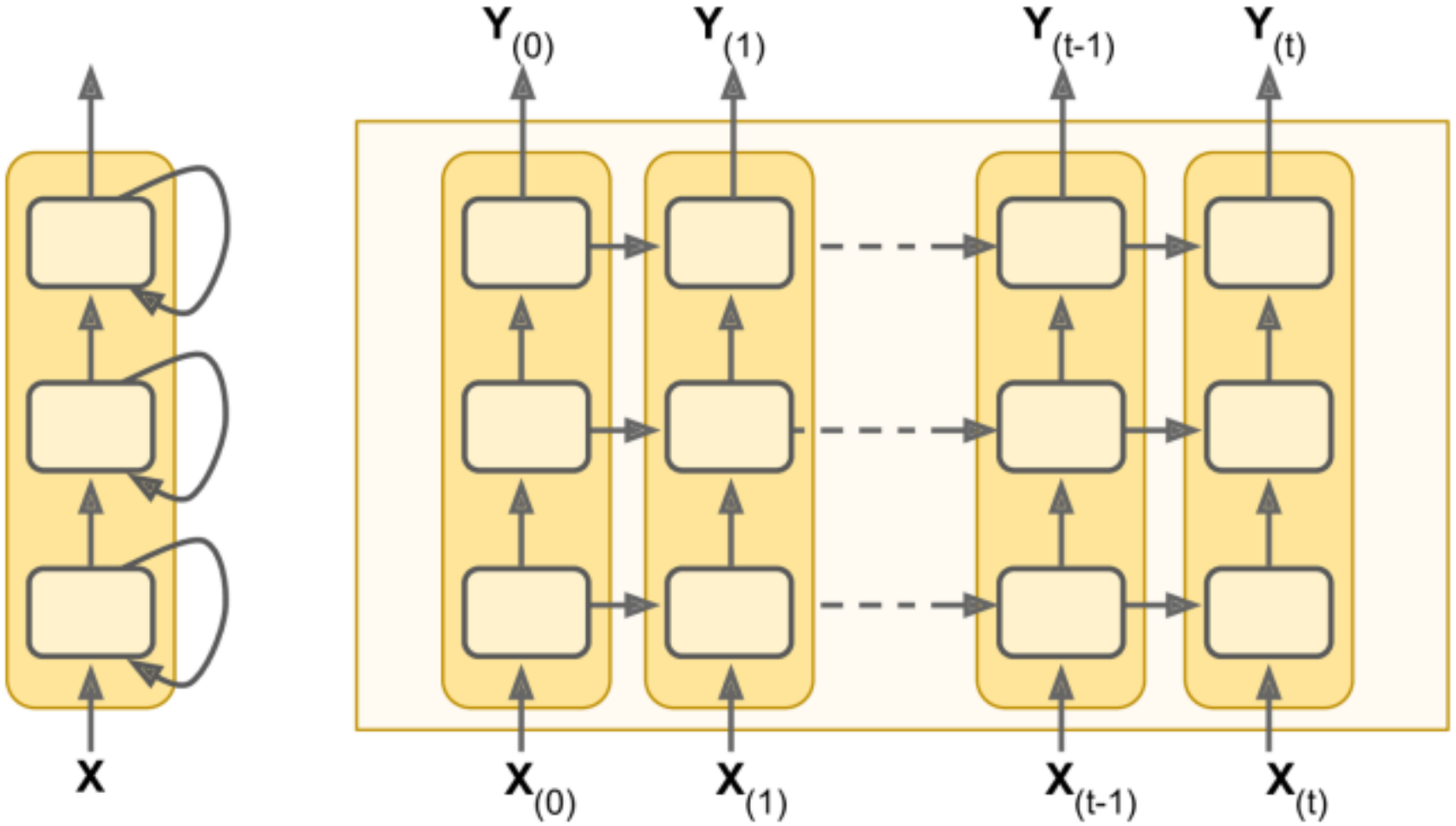
Training RNNs

- To train an RNN, the trick is to **unroll** it through time and then simply use regular backpropagation. This strategy is called **backpropagation through time (BPTT)**.

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

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

Deep RNNs



Deep RNN (left) unrolled through time (right)

Deep RNNs

```
model = keras.models.Sequential([
keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
keras.layers.SimpleRNN(20, return_sequences=True),
keras.layers.SimpleRNN(1) ])
```

- **Make sure to set return_sequences=True for all recurrent layers except the last one, if you only care about the last output.**

- **It might be preferable to replace the output layer with a Dense layer.**

```
model = keras.models.Sequential([
keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
keras.layers.SimpleRNN(20),
keras.layers.Dense(1) ])
```

Deep RNNs

- To turn the model into a **sequence-to-sequence** model, we must set **return_sequences=True** in all recurrent layers (**even the last one**), and we must apply the output Dense layer at every time step.

- Keras offers a **TimeDistributed layer** for this very purpose: it wraps any layer (e.g., a Dense layer) and applies it at every time step of its input sequence.

```
model = keras.models.Sequential([
keras.layers.SimpleRNN(20, return_sequences=True,
                                    input_shape=[None, 1]),
keras.layers.SimpleRNN(20, return_sequences=True),
keras.layers.TimeDistributed(keras.layers.Dense(10))  ])
```

Long Short-Term Memory (LSTM)

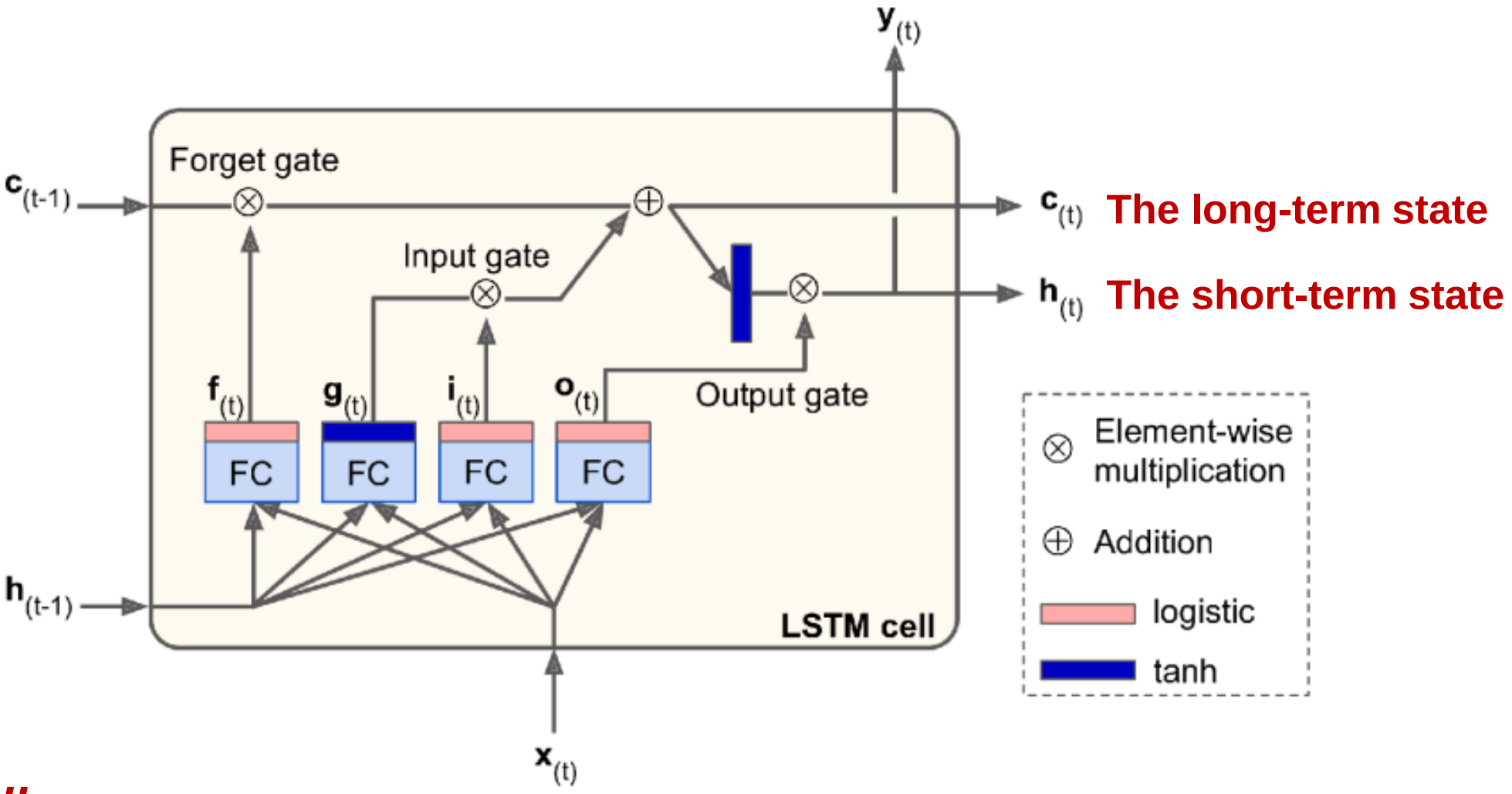
- When the data traversing an RNN, **some information is lost at each time step**. After a while, the RNN's state contains virtually no trace of the first inputs.

- To tackle this problem, various types of **cells with long-term memory** have been introduced like **LSTM cells**.

- The key idea of LSTM is that **the network can learn** what to store in the **long-term** state, what to **throw away**, and what to **read from** it.

- LSTM can be used very much like a basic cell, except **it will perform much better**, training will **converge faster**, and it will **detect long-term** dependencies in the data.

Long Short-Term Memory (LSTM)



LSTM cell

Long Short-Term Memory (LSTM)

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True,
                                    input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),

  keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

The LSTM layer uses an optimized implementation when running on a GPU.

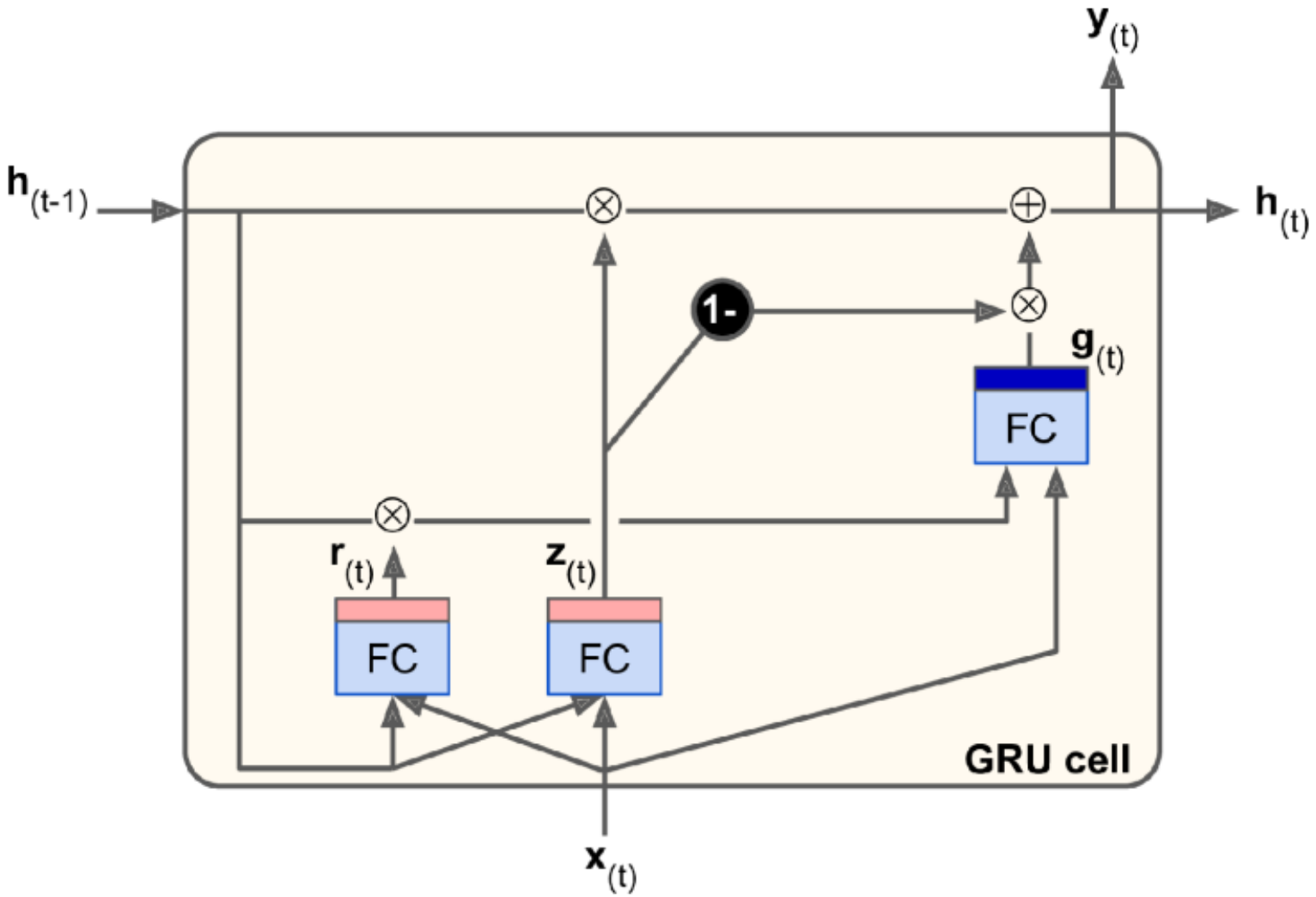
Gated Recurrent Unit (GRU)

- The GRU cell is a **simplified version of the LSTM** cell, and it seems to perform just as well.

- GRU **often improves performance**, but not always, and there is no clear pattern for which tasks are better off with or without them: **you will have to try it on your task and see if it helps**.

- **model.add(keras.layers.GRU(N))**

Gated Recurrent Unit (GRU)



GRU cell

Gated Recurrent Unit (GRU)

- LSTM and GRU cells are one of the main reasons behind the success of RNNs.
- They can tackle **much longer sequences** than simple RNNs.

- They **still have a fairly limited short-term memory**, and they have a **hard time learning long-term patterns** in sequences of 100 time steps or more, such as audio samples, long time series, or long sentences.

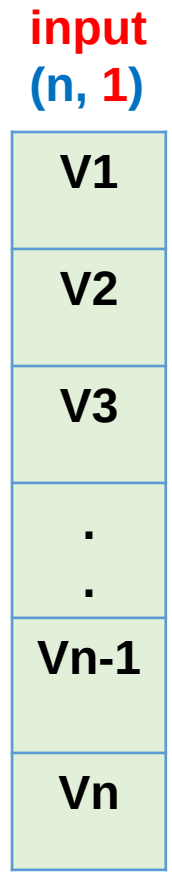
- **One way to solve this is to shorten the input sequences, for example using 1D convolutional layers.**

1D Convolutional Layers

```
model = keras.models.Sequential([

    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2,
                        padding="valid", input_shape=[None, 1]),

    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```



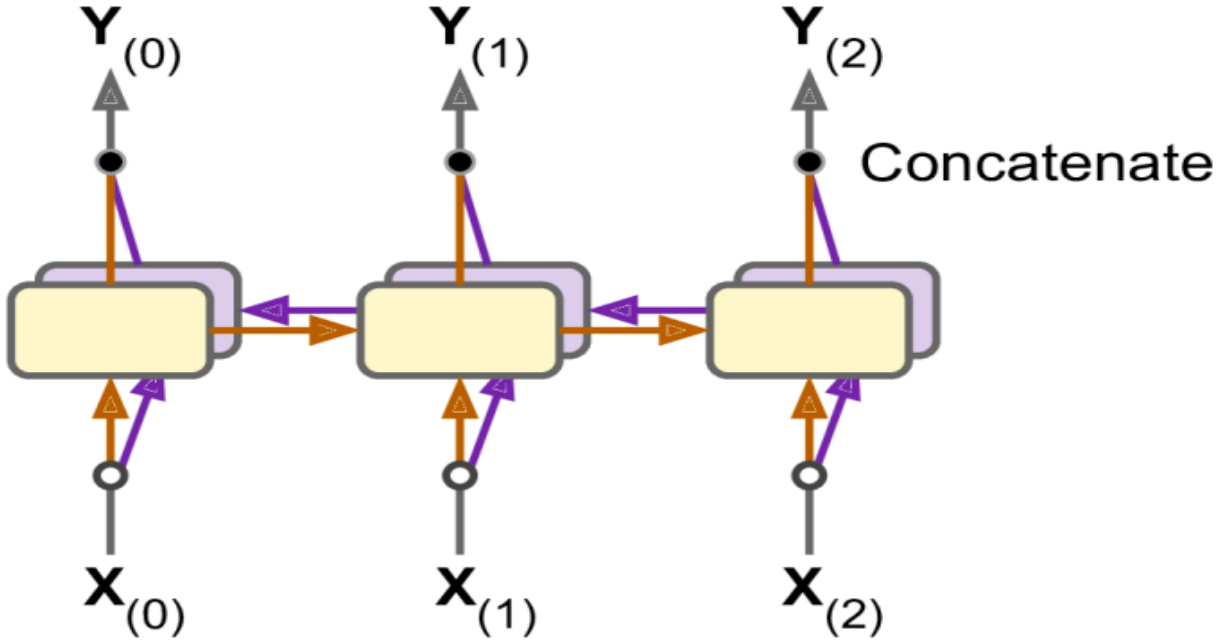
Bidirectional RNNs

For example: in **Neural Machine Translation**, it is often preferable to look ahead at the next words before encoding a given word.

- Consider the phrases "the queen of the United Kingdom", "the queen of hearts", and "the queen bee": to properly encode the word **“queen”**, you need to look ahead.

Bidirectional RNNs

- To implement this, **run two recurrent layers** on the same inputs, one reading the words from **left to right** and the other reading them from **right to left**. Then simply **concatenating** them.



Bidirectional RNNs- keras.layers.Bidirectional

```
model = Sequential()

model.add(Bidirectional(LSTM(10,
          return_sequences=True), input_shape=(5, 10)))
model.add(Bidirectional(LSTM(10)))

model.add(Dense(5))

model.compile(loss="categorical_crossentropy",
                                    optimizer="rmsprop")
```

Forecasting a Time Series

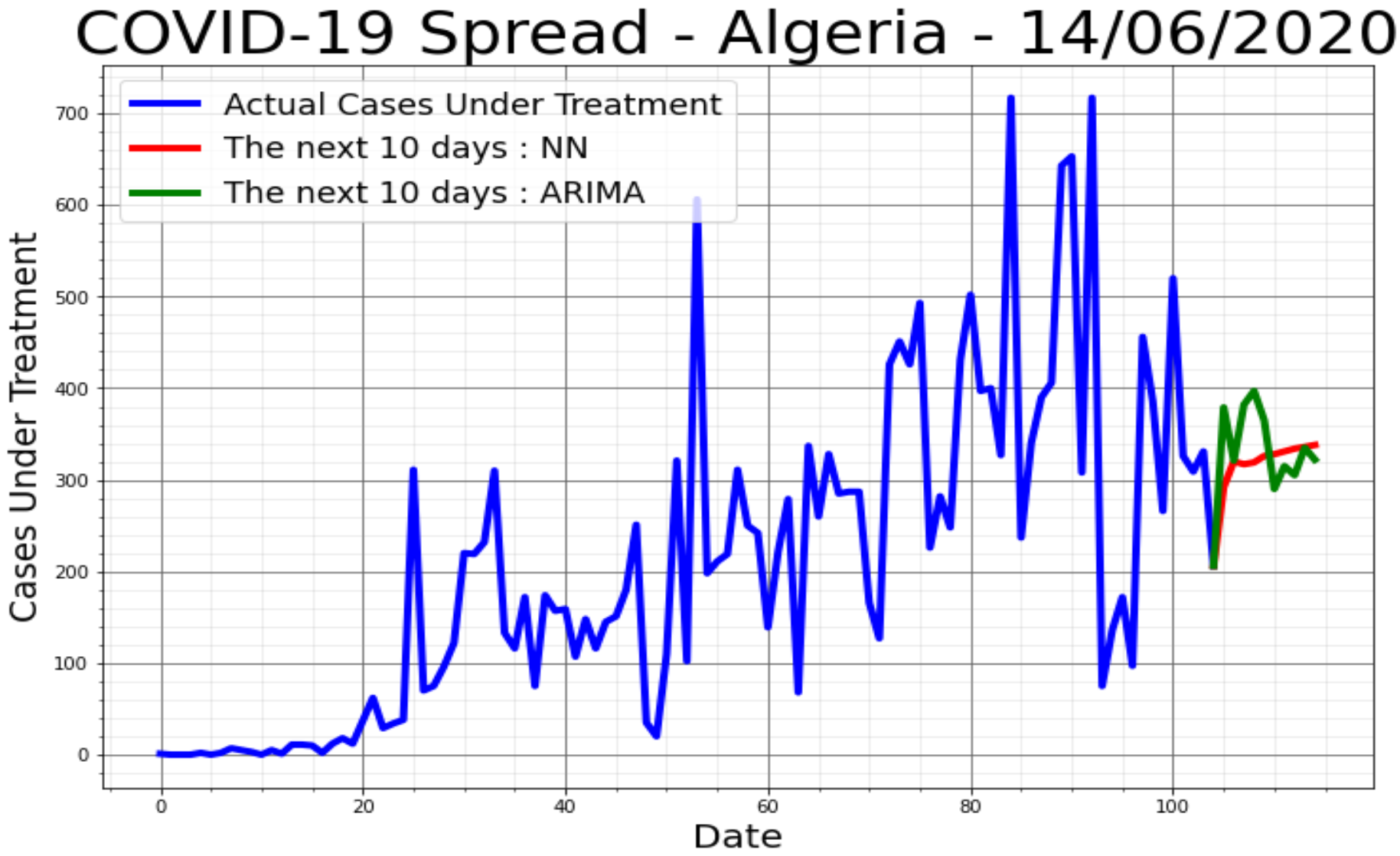
Forecasting a Time Series

- A time series **is a series of data points indexed in time order**. Most commonly, a time series is a sequence taken at successive **equally spaced points in time**. Thus it is a sequence of discrete-time data.

- **single value** per time step = **univariate time series**.
- **multiple values** per time step = **multivariate time series**.

- A typical task is to **predict future values**, which is called **forecasting**. Another common task is to **fill in the blanks**: to predict missing values from the past. This is called **imputation**.

Forecasting a Time Series



Trend and Seasonality

- Some of the models require you to first remove the **trend and seasonality.**

- **For example**, if you are studying the number of active users on your website, and it is growing by 10% every month, you would have to remove this trend from the time series. Similarly, if you are trying to predict the amount of sunscreen lotion sold every month, you will probably observe strong seasonality: since it sells well every summer, a similar pattern will be repeated every year. You would have to remove this seasonality from the time series, for example by differencing technique.

- Once the model is trained and starts making predictions, you would have to add the trend and the seasonal pattern back to get the final predictions.

Transform Time Series to Supervised Learning

Supervised

X	y
5	1
4	0
5	1
3	1
6	0
3	1

$$Y = \boldsymbol{F}(x)$$

Goal: approximate the real underlying mapping so well that when you have new input data (**X**), you can predict the output variables (**y**) for that data

Time series

Time	Measure
1	1
2	0
3	1
4	0
5	0
6	?

Sequence of actions

Time series to Supervised

Time	Lag_1 or X	y
1	?	1
2	1	0
3	0	1
4	1	0
5	0	0
6	0	?

- **Using historical data to predict future action**
- **same time sequence**

Trend and Seasonality

```
from pandas import DataFrame
from pandas import concat
# ********************************************************************************
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        n_vars = 1 if type(data) is list else data.shape[1]
        df = DataFrame(data)
        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(n_in, 0, -1):
                cols.append(df.shift(i))
                names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
        # forecast sequence (t, t+1, ... t+n)
        for i in range(0, n_out):
                cols.append(df.shift(-i))
                if i == 0:
                        names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
                else:
                        names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
        # put it all together
        agg = concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values
        if dropnan:
                agg.dropna(inplace=True)
        return agg
```

Trend and Seasonality

```
# One-Step Univariate Forecasting
print("One-Step Univariate Forecasting : \n")
values = [x for x in range(10)]
print("values : \n",values)
i_in_1    = 2  # past observations
n_out_1 = 2 # future observations
data = series_to_supervised(values, i_in_1, n_out_1)
print(data)
# ****************************************************************************
# Multivariate Forecasting
print("Multivariate Forecasting : \n")
raw = DataFrame()
raw["ob1"] = [x for x in range(10)]
raw["ob2"] = [x for x in range(50, 60)]
values = raw.values
print("values : \n",values)
i_in_2     = 2 # past observations
n_out_2 = 1 # future observations
data = series_to_supervised(values, i_in_2, n_out_2)
print(data)
```

Thank you for your attention

Hichem Felouat ...