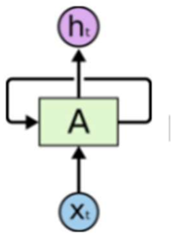
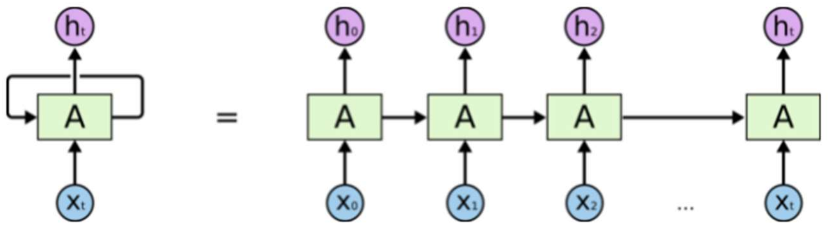
# Long Short-Term Memory (LSTM) Neural Networks

## 1. Introduction to LSTM neural networks

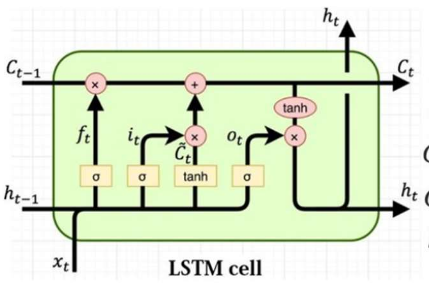
Dense neural networks do not have memory. For some problems, it’s not a major issue, but for others, especially when the order of the input data matters, it is crucial to use networks with kind of memory. For example, in NLP (Natural Language Processing), the order of the words is important for the meaning, and a neural network should have a memory mechanism to keep track of this order. Recurrent neural networks such as the one below can mimic memory effects. To make it simple, let *A* denotes any operator and *t* might be thought of as the time.



To see why such network has memory, let’s discretize the time into steps {0, 1, 2,…} and see how it works. What’s coming out of this network at the first time step is $h_0 = AX_0$ then at the second time step, the new entry of the network will be $h_0 + X_1$. Therefore, the output of the network at the second time step will be $h_1 = A(h_0 + X_1) = A(AX_0 + X_1) = A^2X_0 + AX_1$. In other word, $h_1$ has kept track of the inputs $X_0$ and $X_1$. Similarly for the next time steps we would have $h_2 = A^3X_0 + A^2X_1 + AX_3$, and so on. The problem with such networks is that they are not designed to be coded by modern neural network libraries such as Keras. Indeed, those libraries assume a networks to be directed acyclic graphs (i.e. networks without closed loops). Fortunately, such a loop can be unrolled as follows to give the same result:



A LSTM network is basically such a structure, but with the operator *A* being a tad more complicated than just a linear operator. Here it is:



$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right)$$
$$h_t = \tanh(C_t) * o_t$$

At this stage, it’s not necessary to understand all the internal details of a LSTM cell. What’s important to understand is summarized below:

- the number of cells in the unrolled loop depends on the length of the input data {$X_0$, $X_1$, $X_2$…}. Therefore, it does need to be specified in Keras.
- each cell shares the same parameters {$U^i$, $U^f$, $U^0$, $U^g$, $W^i$, $W^f$, $W^o$, $W^g$}. Those are matrices that play the same role as the weight matrices for dense layers. The size of these matrices must be specified in Keras (just like the number of neurons in a dense layer).
- for what's coming out of a LSTM layer, Keras gives you two choices: if you want it to return the output of the last cell $h_t$, you should set *return_sequences=False*. This is default parameter in Keras. If you want Keras to return the whole sequence {$h_0$, $h_1$,..,$h_t$}, you should set *return_sequences=True*.

In this tutorial, you will discover how to develop a suite of LSTM models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem. The models are demonstrated on small time series problems intended to make it simple and easy to understand.

## 2. Univariate LSTM model

Before a univariate series can be modelled, the data must be prepared. The LSTM model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn. Consider a given simple univariate sequence: $[10, 20, 30, 40, 50, 60, 70, 80, 90]$. We can divide the sequence into multiple input/output patterns called samples. If three time steps are used as input and one time step is used as output for the prediction, the dataset would look like as follows:

```
X,            y
10, 20, 30,   40
20, 30, 40,   50
30, 40, 50,   60
...
```

The *split_sequence()* function below implements this behaviour and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```python
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
    X, y = list(), list()
    for i in range(len(sequence)):
        # find the end of this pattern
        end_ix = i + n_steps
        # check if we are beyond the sequence
        if end_ix > len(sequence)-1:
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

We can now test this function with the simple sequence given above and see how it works:

```python
# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
    print(X[i], y[i])
```

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Now that we know how to prepare a univariate series for modelling, let's look at developing LSTM models. A key point to understand is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features. We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the *split_sequence()* function.
The shape of the input for each sample is specified in the input shape argument on the definition of first hidden layer. We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: *[samples, timesteps, features]*. *Our split_sequence()* function in the previous section outputs the X with the shape *[samples, timesteps]*, so it is necessary to reshape it to have an additional dimension for the one feature.

```python
# reshape from [samples, timesteps] into [samples, timesteps, features]
print('Shape of X before reshaping:', X.shape)
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
print('Shape of X after reshaping:', X.shape)
```

```
Shape of X before reshaping: (6, 3)
Shape of X after reshaping: (6, 3, 1)
```

We define a LSTM model with cells having 50 units in the hidden layer (this is the equivalent of the number of neurons in a dense layer) and an output layer that predicts a single numerical value. This is why we add a dense layer, which takes as input a vector of size 50 and returns a scalar. The model is fit using the Adam optimizer and the mean squared error ‘mse’ loss function. Once the model is defined, we can fit it on the training dataset using 200 epochs.

```python
# univariate lstm example
from keras import Input
from keras.layers import LSTM
from keras.layers import Dense
from keras.models import Model

# define model
x0 = Input(shape = (n_steps, n_features))
x1 = LSTM(50, activation='relu')(x0)
output = Dense(1)(x1)
model = Model(inputs = x0 , outputs = output)
model.compile(optimizer='adam', loss='mse')

# fit model
model.fit(X, y, epochs=200, verbose=1)
```

```
Epoch 1/200
1/1 [==============================] - 1s 924ms/step - loss: 3549.7039
Epoch 2/200
1/1 [==============================] - 0s 13ms/step - loss: 3487.3362
Epoch 3/200
```

Then, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: [70, 80, 90]. And expecting the model to predict something like: [100]. The model expects the input shape to be three-dimensional with *[samples, timesteps, features]*, therefore, we must not forget to reshape the single input sample before making the prediction. We can see that the model predicts the next value in the sequence with good accuracy.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))

y_pred = model.predict(x_input, verbose=0)
print(y_pred)
```

```
[[102.569626]]
```

Just like dense layers, multiple hidden LSTM layers can be stacked one on top of another in what is referred to as a Stacked LSTM model. An LSTM layer requires a three-dimensional input and LSTMs by default will produce a two-dimensional as shown below for the previous model.

```
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_1 (InputLayer)        [(None, 3, 1)]            0

_________________________________________________________________
lstm (LSTM)                 (None, 50)                10400
_________________________________________________________________
dense (Dense)               (None, 1)                 51
=================================================================
Total params: 10,451
Trainable params: 10,451
Non-trainable params: 0
_________________________________________________________________
```

We can address this by having the LSTM output a value for each time step in the input data by setting the *return_sequences*=*True* argument on the layer. This allows us to have 3D output from hidden LSTM layer as input to the next. We can therefore define a Stacked LSTM as follows.

```
# define model with stacked layers
x0 = Input(shape = (n_steps, n_features))
x1 = LSTM(50, activation='relu', return_sequences=True)(x0)
x2 = LSTM(50, activation='relu')(x1)
output = Dense(1)(x2)
model = Model(inputs = x0 , outputs = output)
model.compile(optimizer='adam', loss='mse')
```

Using *model.summary()*, we can see that the dimensions of what's coming out of the first LSTM layer is now suitable for the input of the second LSTM layer.

```
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_2 (InputLayer)        [(None, 3, 1)]            0

_________________________________________________________________
lstm_1 (LSTM)               (None, 3, 50)             10400
_________________________________________________________________
lstm_2 (LSTM)               (None, 50)                20200
_________________________________________________________________
dense_1 (Dense)             (None, 1)                 51
=================================================================
Total params: 30,651
Trainable params: 30,651
Non-trainable params: 0
_________________________________________________________________
```

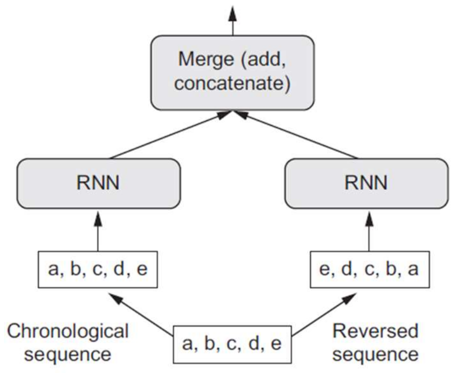
As before, you can check the performance of this model.

```
# fit model
model.fit(X, y, epochs=200, verbose=0)

# make prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
y_pred = model.predict(x_input, verbose=0)
print(y_pred)
```

```
[[101.82018]]
```

On some sequence prediction problems, it can be beneficial to allow the LSTM model to learn the input sequence both forward and backwards (for example, with input [10, 20, 30] and [30, 20, 10]). This is done in parallel, and results of each layer are then concatenated. This is called a Bidirectional LSTM (see below).



We can implement a Bidirectional LSTM without having to reverse the data, Keras takes care of this for you. In terms of input/output dimensions, a bidirectional layer has the same requirements as a LSTM layer. It can be coded as shown below.

```
from keras.layers import Bidirectional

# define model
x0 = Input(shape = (n_steps, n_features))
x1 = Bidirectional(LSTM(50, activation='relu'))(x0)
output = Dense(1)(x1)
model = Model(inputs = x0 , outputs = output)
model.compile(optimizer='adam', loss='mse')
```

Since a bidirectional LSTM layer has two layers working in parallel, its number of parameters is twice the number of parameters of a simple LSTM layer, as can be seen using the *model.summary()* instruction.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 3, 1)]            0
_________________________________________________________________
bidirectional (Bidirectional (None, 100)               20800
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 20,901
Trainable params: 20,901
Non-trainable params: 0
_________________________________________________________________
```

## 3. Multivariate LSTM model

Multivariate time series data means data where there is more than one observation (or feature) for each time step. There are two main models with multivariate time series data:

1. Multiple Input Series models.
2. Multiple Output Series models.

### 3.1 Multiple input series model

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has an observation at the same time steps. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
import numpy as np

# define input/output sequence
in_seq1 = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = np.array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = np.array([25, 45, 65, 85, 105, 125, 145, 165, 185])
```

We can reshape these three arrays of data as a single dataset where each row is a time step, and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# stack columns
dataset = np.vstack((in_seq1, in_seq2, out_seq)).T
print(dataset)
```

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

As with the univariate time series, we must structure these data into samples with input and output elements. If we chose three input time steps, then the first sample would look as follows: and the output would be 65.

```
10, 15
20, 25
30, 35
```

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case, 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named *split_sequences()* that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
    X, y = list(), list()
    for i in range(len(sequences)):
        # find the end of this pattern
        end_ix = i + n_steps
        # check if we are beyond the dataset
        if end_ix > len(sequences):
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

We can now process the dataset (assuming three time steps for each input) and see the result.

```
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
    print(X[i], y[i])
```

```
(7, 3, 2) (7,)
[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

As required by a LSTM network, we see that the processed data has shape *[samples, timesteps, features]*. Then, the model can be built and evaluated just like in the previous section.

```
# define model
n_steps = 3
n_features = X.shape[2]
x0 = Input(shape = (n_steps, n_features))
x1 = LSTM(50, activation='relu')(x0)
output = Dense(1)(x1)
model = Model(inputs = x0 , outputs = output)
model.compile(optimizer='adam', loss='mse')

# fit model
model.fit(X, y, epochs=200, verbose=1)
```

```
Epoch 1/200
1/1 [==============================] - 1s 1s/step - loss: 19541.4727
Epoch 2/200
1/1 [==============================] - 0s 3ms/step - loss: 19261.1719
```

Let’s test the model for a sample *[[80, 85], [90, 95], [100, 105]]* from which we should expect a predicted value equal to 205.

```
# demonstrate prediction
x_input = np.array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
y_pred = model.predict(x_input, verbose=0)
print(y_pred)
```

```
[[205.65337]]
```

## 3.2 Multiple output series model

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

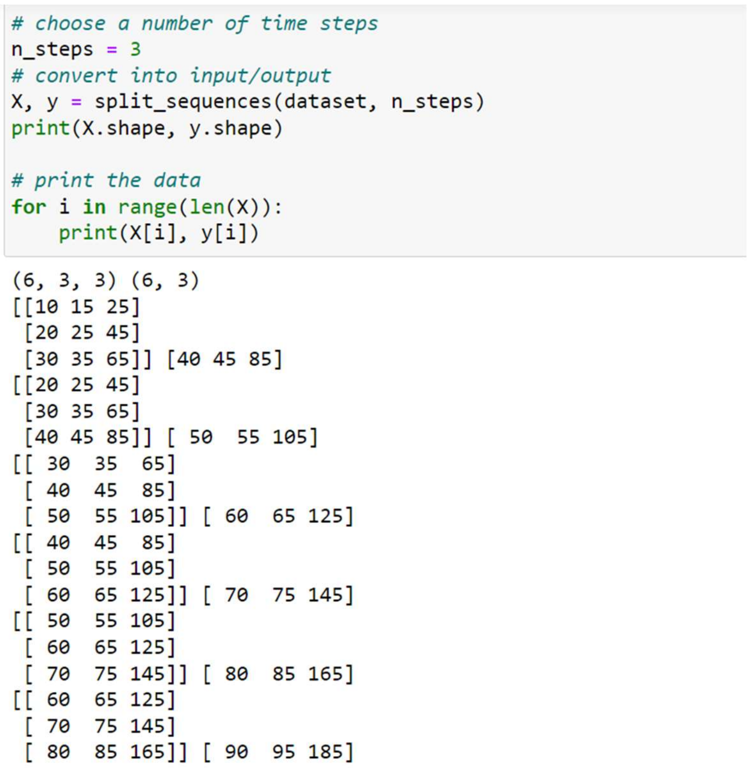
We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples to train a model. For example, for the first sample of this dataset :

```
10, 15, 25
20, 25, 45
30, 35, 65
```

the model to be built should predict the next figure of each column, i.e [ 40, 45, 85 ]. As before, it is necessary to prepare the input data with the correct shapes. The *split sequences()* function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
    X, y = list(), list()
    for i in range(len(sequences)):
        # find the end of this pattern
        end_ix = i + n_steps
        # check if we are beyond the dataset
        if end_ix > len(sequences)-1:
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)
```

We can now process the 3-columns dataset. Running the code below first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). The data is ready to use in an LSTM model that expects three-dimensional input and two-dimensional output shapes for the X and y components of each sample. Then, each of the samples is printed showing the input and output components of each sample.

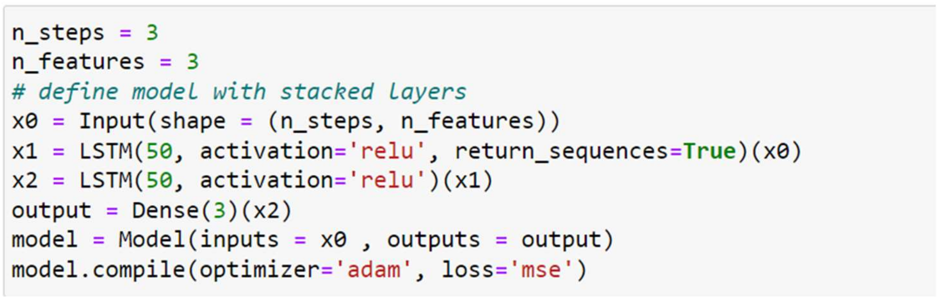


```
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)

# print the data
for i in range(len(X)):
    print(X[i], y[i])
```

```
(6, 3, 3) (6, 3)
[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50  55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60  65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70  75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80  85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90  95 185]
```

We are now ready to fit an LSTM model on this data. For this example, we will use a Stacked LSTM where the number of time steps and parallel series (features) are specified for the input layer via the input shape argument. Each LSTM layer has 100 units (this is the equivalent of the number of neurons in a dense layer). Since the model should return a vector of dimension 3, the last dense layer must have 3 neurons. Of course, one should not forget to set *return_sequences*=*True* to make sure that what's coming out of the first LSTM layer has the correct shape for the second one.



```
n_steps = 3
n_features = 3
# define model with stacked layers
x0 = Input(shape = (n_steps, n_features))
x1 = LSTM(50, activation='relu', return_sequences=True)(x0)
x2 = LSTM(50, activation='relu')(x1)
output = Dense(3)(x2)
model = Model(inputs = x0 , outputs = output)
model.compile(optimizer='adam', loss='mse')
```

Then, we can fit and evaluate the model to see how it performs. For that, we give the following input:

```
70, 75, 145
80, 85, 165
90, 95, 185
```

from which we should expect the output [100, 105, 205]. As usual, it is necessary to reshape this sample in the right format.

```
# fit model
model.fit(X, y, epochs=400, verbose=0)

# demonstrate prediction
x_input = np.array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
y_pred = model.predict(x_input, verbose=0)
print(y_pred)
```

```
[[100.7706  105.80254 206.39705]]
```

## 4. Multi-step LSTM Models

A time series forecasting problem that requires a prediction of multiple time steps into the future can be referred to as multi-step time series forecasting. Specifically, these are problems where the forecast horizon or interval is more than one time step. As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps.
For example, given the univariate time series: [10, 20, 30, 40, 50, 60, 70, 80, 90], we could use the last three time steps as input and forecast the next two time steps. The first sample would be [10, 20, 30] for the input, and [40, 50] for the output. The *split_sequence()* function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
    X, y = list(), list()
    for i in range(len(sequence)):
        # find the end of this pattern
        end_ix = i + n_steps_in
        out_end_ix = end_ix + n_steps_out
        # check if we are beyond the sequence
        if out_end_ix > len(sequence):
            break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return np.array(X), np.array(y)
```

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
    print(X[i], y[i])
```

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Now that we know how to prepare data for multi-step forecasting, let's look at some LSTM models that can learn this mapping. Like other types of neural network models, the LSTM can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. As with the LSTMs for univariate data in a prior section, the prepared samples must first be reshaped. The LSTM expects data to have a three-dimensional structure of *[samples, timesteps, features]*, and in this case, we only have one feature so the reshape is straightforward.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
print(X.shape)
```

```
(5, 3, 1)
```

The model can be built as before:

```
n_steps = 3
n_features = 1
# define model with stacked layers
x0 = Input(shape = (n_steps, n_features))
x1 = LSTM(50, activation='relu', return_sequences=True)(x0)
x2 = LSTM(50, activation='relu')(x1)
output = Dense(2)(x2)
model = Model(inputs = x0 , outputs = output)
model.compile(optimizer='adam', loss='mse')
```

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input [70, 80, 90] and we would expect the predicted output to be [100, 110]. Let’s check this:

```
# fit model
model.fit(X, y, epochs=500, verbose=0)

# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
y_pred = model.predict(x_input, verbose=0)
print(y_pred)
```

```
[[101.67877  112.611946]]
```

## 5. Further work

You can test bidirectionnal LSTM networks for the last examples and see if it improves results. LSTM is not the only recurrent layer available in Keras. Another one commonly used is called GRU for Gated Recurrent Unit. It work exactly in the same way as a LSTM layer. All you need to do is changing LSTM in the codes by GRU and see if it works better.