# **Solving Sequence Problems with LSTM in Keras**

In this article, you will learn how to perform time series forecasting that is used to solve sequence problems.

Time series forecasting refers to the type of problems where we have to predict an outcome based on time dependent inputs. A typical example of time series data is stock market data where stock prices change with time. Similarly, the hourly temperature of a particular place also changes and can also be considered as time series data. Time series data is basically a sequence of data, hence time series problems are often referred to as sequence problems.

[Recurrent Neural Networks](https://en.wikipedia.org/wiki/Recurrent_neural_network) (RNN) have been proven to efficiently solve sequence problems. Particularly, [Long Short Term Memory Network](https://en.wikipedia.org/wiki/Long_short-term_memory) (LSTM), which is a variation of RNN, is currently being used in a variety of domains to solve sequence problems.

## **Types of Sequence Problems**

Sequence problems can be broadly categorized into the following categories:

1. **One-to-One:** Where there is one input and one output. Typical example of a one-to-one sequence problem is the case where you have an image and you want to predict a single label for the image.
2. **Many-to-One:** In many-to-one sequence problems, we have a sequence of data as input and we have to predict a single output. Text classification is a prime example of many-to-one sequence problems where we have an input sequence of words and we want to predict a single output tag.
3. **One-to-Many:** In one-to-many sequence problems, we have a single input and a sequence of outputs. A typical example is an image and its corresponding description.
4. **Many-to-Many**: Many-to-many sequence problems involve a sequence input and a sequence output. For instance, stock prices of 7 days as input and stock prices of next 7 days as outputs. Chatbots are also an example of many-to-many sequence problems where a text sequence is an input and another text sequence is the output.

This article is part 1 of the series. In this article, we will see how LSTM and its different variants can be used to solve one-to-one and many-to-one sequence problems. In the [next part of this series](https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras-part-2/), we will see how to solve one-to-many and many-to-many sequence problems. We will be working with Python's Keras library.

After reading this article, you will be able solve problems like stock price prediction, [weather prediction](https://stackabuse.com/using-machine-learning-to-predict-the-weather-part-1/), etc., based on historical data. Since, text is also a sequence of words, the knowledge gained in this article can also be used to solve [natural language processing](https://stackabuse.com/what-is-natural-language-processing/) tasks such as text classification, language generation, etc.

## **One-to-One Sequence Problems**

As I said earlier, in one-to-one sequence problems, there is a single input and a single output. In this section we will see two types of sequence problems. First we will see how to solve one-to-one sequence problems with a single feature and then we will see how to solve one-to-one sequence problems with multiple features.

### **One-to-One Sequence Problems with a Single Feature**

In this section, we will see how to solve a one-to-one sequence problem where each time-step has a single feature.

Let's first import the required libraries that we are going to use in this article:

from numpy import array

from keras.preprocessing.text import one\_hot

from keras.preprocessing.sequence import pad\_sequences

from keras.models import Sequential

from keras.layers.core import Activation, Dropout, Dense

from keras.layers import Flatten, LSTM

from keras.layers import GlobalMaxPooling1D

from keras.models import Model

from keras.layers.embeddings import Embedding

from sklearn.model\_selection import train\_test\_split

from keras.preprocessing.text import Tokenizer

from keras.layers import Input

from keras.layers.merge import Concatenate

from keras.layers import Bidirectional

import pandas as pd

import numpy as np

import re

import matplotlib.pyplot as plt

#### **Creating the Dataset**

In this next step, we will prepare the dataset that we are going to use for this section.

X = list()

Y = list()

X = [x+1 for x in range(20)]

Y = [y \* 15 for y in X]

print(X)

print(Y)

In the script above, we create 20 inputs and 20 outputs. Each input consists of one time-step, which in turn contains a single feature. Each output value is *15 times the corresponding input value*. If you run the above script, you should see the input and output values as shown below:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

[15, 30, 45, 60, 75, 90, 105, 120, 135, 150, 165, 180, 195, 210, 225, 240, 255, 270, 285, 300]

The input to the LSTM layer should be in 3D shape i.e. (samples, time-steps, features). The samples are the number of samples in the input data. We have 20 samples in the input. The time-steps is the number of time-steps per sample. We have 1 time-step. Finally, features correspond to the number of features per time-step. We have one feature per time-step.

We can reshape our data via the following command:

X = array(X).reshape(20, 1, 1)

#### **Solution via Simple LSTM**

Now we can create our simple LSTM model with one LSTM layer.

model = Sequential()

model.add(LSTM(50, activation='relu', input\_shape=(1, 1)))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

print(model.summary())

In the script above, we create an LSTM model with one LSTM layer of 50 neurons and relu activation functions. You can see the input shape is (1,1) since our data has one time-step with one feature. Executing the above script prints the following summary:

Layer (type) Output Shape Param #

=================================================================

lstm\_16 (LSTM) (None, 50) 10400

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_15 (Dense) (None, 1) 51

=================================================================

Total params: 10,451

Trainable params: 10,451

Non-trainable params: 0

Let's now train our model:

model.fit(X, Y, epochs=2000, validation\_split=0.2, batch\_size=5)

We train our model for 2000 epochs with a batch size of 5. You can choose any number. Once the model is trained, we can make predictions on a new instance.

Let's say we want to predict the output for an input of 30. The actual output should be 30 x 15 = 450. Let's see what value we get. First, we need to convert our test data to the right shape i.e. 3D shape, as expected by LSTM. The following script predicts the output for the number 30:

test\_input = array([30])

test\_input = test\_input.reshape((1, 1, 1))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

I got an output value of 437.86 which is slightly less than 450.

**Note:** It is important to mention that the outputs that you obtain by running the scripts will be different from mine. This is because the LSTM neural network initializes weights with random values and your values. But overall, the results should not differ much.

#### **Solution via Stacked LSTM**

Let's now create a stacked LSTM and see if we can get better results. The dataset will remain the same, the model will be changed. Look at the following script:

model = Sequential()

model.add(LSTM(50, activation='relu', return\_sequences=True, input\_shape=(1, 1)))

model.add(LSTM(50, activation='relu'))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

print(model.summary())

In the above model, we have two LSTM layers. Notice, the first LSTM layer has parameter return\_sequences, which is set to True. When the return sequence is set to True, the output of the hidden state of each neuron is used as an input to the next LSTM layer. The summary of the above model is as follows:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

lstm\_33 (LSTM) (None, 1, 50) 10400

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_34 (LSTM) (None, 50) 20200

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_24 (Dense) (None, 1) 51

=================================================================

Total params: 30,651

Trainable params: 30,651

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Next, we need to train our model as shown in the following script:

history = model.fit(X, Y, epochs=2000, validation\_split=0.2, verbose=1, batch\_size=5)

Once the model is trained, we will again make predictions on the test data point i.e. 30.

test\_input = array([30])

test\_input = test\_input.reshape((1, 1, 1))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

I got an output of 459.85 which is better than 437, the number that we achieved via a single LSTM layer.

### **One-to-One Sequence Problems with Multiple Features**

In the last section, each input sample had one time-step, where each time-step had one feature. In this section we will see how to solve one-to-one sequence problems where input time-steps have multiple features.

#### **Creating the Dataset**

Let's first create our dataset. Look at the following script:

nums = 25

X1 = list()

X2 = list()

X = list()

Y = list()

X1 = [(x+1)\*2 for x in range(25)]

X2 = [(x+1)\*3 for x in range(25)]

Y = [x1\*x2 for x1,x2 in zip(X1,X2)]

print(X1)

print(X2)

print(Y)

In the script above, we create three lists: X1, X2, and Y. Each list has 25 elements, which means that the total sample size is 25. Finally, Y contains the output. X1, X2, and Y lists have been printed below:

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]

[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75]

[6, 24, 54, 96, 150, 216, 294, 384, 486, 600, 726, 864, 1014, 1176, 1350, 1536, 1734, 1944, 2166, 2400, 2646, 2904, 3174, 3456, 3750]

Each element in the output list is basically the product of the corresponding elements in the X1 and X2 lists. For instance, the second element in the output list is 24, which is the product of the second element in list X1 i.e. 4, and the second element in the list X2 i.e. 6.

The input will consist of the combination of X1 and X2 lists, where each list will be represented as a column. The following script creates the final input:

X = np.column\_stack((X1, X2))

print(X)

Here is the output:

[[ 2 3]

 [ 4 6]

 [ 6 9]

 [ 8 12]

 [10 15]

 [12 18]

 [14 21]

 [16 24]

 [18 27]

 [20 30]

 [22 33]

 [24 36]

 [26 39]

 [28 42]

 [30 45]

 [32 48]

 [34 51]

 [36 54]

 [38 57]

 [40 60]

 [42 63]

 [44 66]

 [46 69]

 [48 72]

 [50 75]]

Here the X variable contains our final feature set. You can see it contains two columns i.e. two features per input. As we discussed earlier, we need to convert the input into a 3-dimensional shape. Our input has 25 samples, where each sample consists of 1 time-step and each time-step consists of 2 features. The following script reshapes the input.

X = array(X).reshape(25, 1, 2)

#### **Solution via Simple LSTM**

We are now ready to train our LSTM models. Let's first develop a single LSTM layer model as we did in the previous section:

model = Sequential()

model.add(LSTM(80, activation='relu', input\_shape=(1, 2)))

model.add(Dense(10, activation='relu'))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

print(model.summary())

Here our LSTM layer contains 80 neurons. We have two dense layers where the first layer contains 10 neurons and the second dense layer, which also acts as the output layer, contains 1 neuron. The summary of the model is as follows:

Layer (type) Output Shape Param #

=================================================================

lstm\_38 (LSTM) (None, 80) 26560

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_29 (Dense) (None, 10) 810

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_30 (Dense) (None, 1) 11

=================================================================

Total params: 27,381

Trainable params: 27,381

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

None

The following script trains the model:

model.fit(X, Y, epochs=2000, validation\_split=0.2, batch\_size=5)

Let's test our trained model on a new data point. Our data point will have two features i.e. (55,80) the actual output should be 55 x 80 = 4400. Let's see what our algorithm predicts. Execute the following script:

test\_input = array([55,80])

test\_input = test\_input.reshape((1, 1, 2))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

I got 3263.44 in the output, which is far from the actual output.

#### **Solution via Stacked LSTM**

Let's now create a more complex LSTM with multiple LSTM and dense layers and see if we can improve our answer:

model = Sequential()

model.add(LSTM(200, activation='relu', return\_sequences=True, input\_shape=(1, 2)))

model.add(LSTM(100, activation='relu', return\_sequences=True))

model.add(LSTM(50, activation='relu', return\_sequences=True))

model.add(LSTM(25, activation='relu'))

model.add(Dense(20, activation='relu'))

model.add(Dense(10, activation='relu'))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

print(model.summary())

The model summary is as follows:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

lstm\_53 (LSTM) (None, 1, 200) 162400

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_54 (LSTM) (None, 1, 100) 120400

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_55 (LSTM) (None, 1, 50) 30200

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_56 (LSTM) (None, 25) 7600

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_43 (Dense) (None, 20) 520

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_44 (Dense) (None, 10) 210

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_45 (Dense) (None, 1) 11

=================================================================

Total params: 321,341

Trainable params: 321,341

Non-trainable params: 0

The next step is to train our model and test it on the test data point i.e. (55,80).

To improve the accuracy, we will reduce the batch size, and since our model is more complex now we can also reduce the number of epochs. The following script trains the LSTM model and makes predictions on the test datapoint.

history = model.fit(X, Y, epochs=1000, validation\_split=0.1, verbose=1, batch\_size=3)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

In the output, I got a value of 3705.33 which is still less than 4400, but is much better than the previously obtained value of 3263.44 using a single LSTM layer. You can play with different combinations of LSTM layers, dense layers, batch size and the number of epochs to see if you get better results.

## **Many-to-One Sequence Problems**

In the previous sections we saw how to solve one-to-one sequence problems with LSTM. In a one-to-one sequence problem, each sample consists of a single time-step of one or multiple features. Data with a single time-step cannot be considered sequence data in a real sense. Densely connected neural networks have been proven to perform better with single time-step data.

Real sequence data consists of multiple time-steps, such as stock market prices of past 7 days, a sentence containing multiple words, and so on.

In this section, we will see how to solve many-to-one sequence problems. In many-to-one sequence problems, each input sample has more than one time-step, however the output consists of a single element. Each time-step in the input can have one or more features. We will start with many-to-one sequence problems having one feature, and then we will see how to solve many-to-one problems where input time-steps have multiple features.

### **Many-to-One Sequence Problems with a Single Feature**

Let's first create the dataset. Our dataset will consist of 15 samples. Each sample will have 3 time-steps where each time-step will consist of a single feature i.e. a number. The output for each sample will be the sum of the numbers in each of the three time-steps. For instance, if our sample contains a sequence 4,5,6 the output will be 4 + 5 + 6 = 10.

#### **Creating the Dataset**

Let's first create a list of integers from 1 to 45. Since we want 15 samples in our dataset, we will reshape the list of integers containing the first 45 integers.

X = np.array([x+1 for x in range(45)])

print(X)

In the output, you should see the first 45 integers:

[ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45]

We can reshape it into number of samples, time-steps and features using the following function:

X = X.reshape(15,3,1)

print(X)

The above script converts the list X into a 3-dimensional shape with 15 samples, 3 time-steps, and 1 feature. The script above also prints the reshaped data.

[[[ 1]

 [ 2]

 [ 3]]

 [[ 4]

 [ 5]

 [ 6]]

 [[ 7]

 [ 8]

 [ 9]]

 [[10]

 [11]

 [12]]

 [[13]

 [14]

 [15]]

 [[16]

 [17]

 [18]]

 [[19]

 [20]

 [21]]

 [[22]

 [23]

 [24]]

 [[25]

 [26]

 [27]]

 [[28]

 [29]

 [30]]

 [[31]

 [32]

 [33]]

 [[34]

 [35]

 [36]]

 [[37]

 [38]

 [39]]

 [[40]

 [41]

 [42]]

 [[43]

 [44]

 [45]]]

We have converted our input data into the right format, let's now create our output vector. As I said earlier, each element in the output will be equal to the sum of the values in the time-steps in the corresponding input sample. The following script creates the output vector:

Y = list()

for x in X:

 Y.append(x.sum())

Y = np.array(Y)

print(Y)

The output array Y looks like this:

[ 6 15 24 33 42 51 60 69 78 87 96 105 114 123 132]

#### **Solution via Simple LSTM**

Let's now create our model with one LSTM layer.

model = Sequential()

model.add(LSTM(50, activation='relu', input\_shape=(3, 1)))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

The following script trains our model:

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

Once the model is trained, we can use it to make predictions on the test data points. Let's predict the output for the number sequence 50,51,52. The actual output should be 50 + 51 + 52 = 153. The following script converts our test points into a 3-dimensional shape and then predicts the output:

test\_input = array([50,51,52])

test\_input = test\_input.reshape((1, 3, 1))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

I got 145.96 in the output, which is around 7 points less than the actual output value of 153.

#### **Solution via Stacked LSTM**

Let's now create a complex LSTM model with multiple layers and see if we can get better results. Execute the following script to create and train a complex model with multiple LSTM and dense layers:

model = Sequential()

model.add(LSTM(200, activation='relu', return\_sequences=True, input\_shape=(3, 1)))

model.add(LSTM(100, activation='relu', return\_sequences=True))

model.add(LSTM(50, activation='relu', return\_sequences=True))

model.add(LSTM(25, activation='relu'))

model.add(Dense(20, activation='relu'))

model.add(Dense(10, activation='relu'))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

Let's now test our model on the test sequence i.e. 50, 51, 52:

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The answer I got here is 155.37, which is better than the 145.96 result that we got earlier. In this case, we have a difference of only 2 points from 153, which is the actual answer.

#### **Solution via Bidirectional LSTM**

[Bidirectional LSTM](https://en.wikipedia.org/wiki/Bidirectional_recurrent_neural_networks) is a type of LSTM which learns from the input sequence from both forward and backward directions. The final sequence interpretation is the concatenation of both forward and backward learning passes. Let's see if we can get better results with bidirectional LSTMs.

The following script creates a bidirectional LSTM model with one bidirectional layer and one dense layer which acts as the output of the model.

from keras.layers import Bidirectional

model = Sequential()

model.add(Bidirectional(LSTM(50, activation='relu'), input\_shape=(3, 1)))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

The following script trains the model and makes predictions on the test sequence which is 50, 51, and 52.

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The result I got is 152.26 which is just a fraction short of the actual result. Therefore, we can conclude that for our dataset, bidirectional LSTM with single layer outperforms both the single layer and stacked unidirectional LSTMs.

### **Many-to-one Sequence Problems with Multiple Features**

In a many-to-one sequence problem we have an input where each time-steps consists of multiple features. The output can be a single value or multiple values, one per feature in the input time step. We will cover both the cases in this section.

#### **Creating the Dataset**

Our dataset will contain 15 samples. Each sample will consist of 3 time-steps. Each time-steps will have two features.

Let's create two lists. One will contain multiples of 3 until 135 i.e. 45 elements in total. The second list will contain multiples of 5, from 1 to 225. The second list will also contain 45 elements in total. The following script creates these two lists:

X1 = np.array([x+3 for x in range(0, 135, 3)])

print(X1)

X2 = np.array([x+5 for x in range(0, 225, 5)])

print(X2)

You can see the contents of the list in the following output:

[ 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54

 57 60 63 66 69 72 75 78 81 84 87 90 93 96 99 102 105 108

 111 114 117 120 123 126 129 132 135]

[ 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90

 95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175 180

 185 190 195 200 205 210 215 220 225]

Each of the above list represents one feature in the time sample. The aggregated dataset can be created by joining the two lists as shown below:

X = np.column\_stack((X1, X2))

print(X)

The output shows the aggregated dataset:

[ 6 10]

 [ 9 15]

 [ 12 20]

 [ 15 25]

 [ 18 30]

 [ 21 35]

 [ 24 40]

 [ 27 45]

 [ 30 50]

 [ 33 55]

 [ 36 60]

 [ 39 65]

 [ 42 70]

 [ 45 75]

 [ 48 80]

 [ 51 85]

 [ 54 90]

 [ 57 95]

 [ 60 100]

 [ 63 105]

 [ 66 110]

 [ 69 115]

 [ 72 120]

 [ 75 125]

 [ 78 130]

 [ 81 135]

 [ 84 140]

 [ 87 145]

 [ 90 150]

 [ 93 155]

 [ 96 160]

 [ 99 165]

 [102 170]

 [105 175]

 [108 180]

 [111 185]

 [114 190]

 [117 195]

 [120 200]

 [123 205]

 [126 210]

 [129 215]

 [132 220]

 [135 225]]

We need to reshape our data into three dimensions so that it can be used by LSTM. We have 45 rows in total and two columns in our dataset. We will reshape our dataset into 15 samples, 3 time-steps, and two features.

X = array(X).reshape(15, 3, 2)

print(X)

You can see the 15 samples in the following output:

[[[ 3 5]

 [ 6 10]

 [ 9 15]]

 [[ 12 20]

 [ 15 25]

 [ 18 30]]

 [[ 21 35]

 [ 24 40]

 [ 27 45]]

 [[ 30 50]

 [ 33 55]

 [ 36 60]]

 [[ 39 65]

 [ 42 70]

 [ 45 75]]

 [[ 48 80]

 [ 51 85]

 [ 54 90]]

 [[ 57 95]

 [ 60 100]

 [ 63 105]]

 [[ 66 110]

 [ 69 115]

 [ 72 120]]

 [[ 75 125]

 [ 78 130]

 [ 81 135]]

 [[ 84 140]

 [ 87 145]

 [ 90 150]]

 [[ 93 155]

 [ 96 160]

 [ 99 165]]

 [[102 170]

 [105 175]

 [108 180]]

 [[111 185]

 [114 190]

 [117 195]]

 [[120 200]

 [123 205]

 [126 210]]

 [[129 215]

 [132 220]

 [135 225]]]

The output will also have 15 values corresponding to 15 input samples. Each value in the output will be the sum of the two feature values in the third time-step of each input sample. For instance, the third time-step of the first sample has features 9 and 15, hence the output will be 24. Similarly, the two feature values in the third time-step of the 2nd sample are 18 and 30; the corresponding output will be 48, and so on.

The following script creates and displays the output vector:

[ 24 48 72 96 120 144 168 192 216 240 264 288 312 336 360]

Let's now solve this many-to-one sequence problem via simple, stacked, and bidirectional LSTMs.

#### **Solution via Simple LSTM**

model = Sequential()

model.add(LSTM(50, activation='relu', input\_shape=(3, 2)))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

The model is trained. We will create a test data point and then will use our model to make predictions on the test point.

test\_input = array([[8, 51],

 [11,56],

 [14,61]])

test\_input = test\_input.reshape((1, 3, 2))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The sum of two features of the third time-step of the input is 14 + 61 = 75. Our model with one LSTM layer predicted 73.41, which is pretty close.

#### **Solution via Stacked LSTM**

The following script trains a stacked LSTM and makes predictions on test point:

model = Sequential()

model.add(LSTM(200, activation='relu', return\_sequences=True, input\_shape=(3, 2)))

model.add(LSTM(100, activation='relu', return\_sequences=True))

model.add(LSTM(50, activation='relu', return\_sequences=True))

model.add(LSTM(25, activation='relu'))

model.add(Dense(20, activation='relu'))

model.add(Dense(10, activation='relu'))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output I received is 71.56, which is worse than the simple LSTM. Seems like our stacked LSTM is overfitting.

#### **Solution via Bidirectional LSTM**

Here is the training script for simple bidirectional LSTM along with code that is used to make predictions on the test data point:

from keras.layers import Bidirectional

model = Sequential()

model.add(Bidirectional(LSTM(50, activation='relu'), input\_shape=(3, 2)))

model.add(Dense(1))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output is 76.82 which is pretty close to 75. Again, bidirectional LSTM seems to be outperforming the rest of the algorithms.

Till now we have predicted single values based on multiple features values from different time-steps. There is another case of many-to-one sequences where you want to predict one value for each feature in the time-step. For instance, the dataset we used in this section has three time-steps and each time-step has two features. We may want to predict individual values for each feature series. The following example makes it clear, suppose we have the following input:

[[[ 3 5]

 [ 6 10]

 [ 9 15]]

In the output, we want one time-step with two features as shown below:

[12, 20]

You can see the first value in the output is a continuation of the first series and the second value is the continuation of the second series. We can solve such problems by simply changing the number of neurons in the output dense layer to the number of features values that we want in the output. However, first we need to update our output vector Y. The input vector will remain the same:

Y = list()

for x in X:

 new\_item = list()

 new\_item.append(x[2][0]+3)

 new\_item.append(x[2][1]+5)

 Y.append(new\_item)

Y = np.array(Y)

print(Y)

The above script creates an updated output vector and prints it on the console, the output looks like this:

[[ 12 20]

 [ 21 35]

 [ 30 50]

 [ 39 65]

 [ 48 80]

 [ 57 95]

 [ 66 110]

 [ 75 125]

 [ 84 140]

 [ 93 155]

 [102 170]

 [111 185]

 [120 200]

 [129 215]

 [138 230]]

ADVERTISEMENT

Let's now train our simple, stacked and bidirectional LSTM networks on our dataset. The following script trains a simple LSTM:

model = Sequential()

model.add(LSTM(50, activation='relu', input\_shape=(3, 2)))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

The next step is to test our model on the test data point. The following script creates a test data point:

test\_input = array([[20,34],

 [23,39],

 [26,44]])

test\_input = test\_input.reshape((1, 3, 2))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The actual output is [29, 45]. Our model predicts [29.089157, 48.469097], which is pretty close.

Let's now train a stacked LSTM and predict the output for the test data point:

model = Sequential()

model.add(LSTM(100, activation='relu', return\_sequences=True, input\_shape=(3, 2)))

model.add(LSTM(50, activation='relu', return\_sequences=True))

model.add(LSTM(25, activation='relu'))

model.add(Dense(10, activation='relu'))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=500, validation\_split=0.2, verbose=1)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output is [29.170143, 48.688267], which is again very close to actual output.

Finally, we can train our bidirectional LSTM and make prediction on the test point:

from keras.layers import Bidirectional

model = Sequential()

model.add(Bidirectional(LSTM(50, activation='relu'), input\_shape=(3, 2)))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output is [29.2071, 48.737988].

You can see once again that bidirectional LSTM makes the most accurate prediction.

## **Conclusion**

Simple neural networks are not suitable for solving sequence problems since in sequence problems, in addition to current input, we need to keep track of the previous inputs as well. Neural Networks with some sort of memory are more suited to solving sequence problems. LSTM is one such network.

In this article, we saw how different variants of the LSTM algorithm can be used to solve one-to-one and many-to-one sequence problems. This is the first part of the article. In the [second part](https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras-part-2/), we will see how to solve one-to-many and many-to-many sequence problems. We will also study the encoder-decoder mechanism that is most commonly used to create chatbots. Till then, happy coding :)

This is the second and final part of the two-part series of articles on solving sequence problems with LSTMs. In the [part 1 of the series](https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras/), I explained how to solve one-to-one and many-to-one sequence problems using LSTM. In this part, you will see how to solve one-to-many and many-to-many sequence problems via LSTM in Keras.

Image captioning is a classic example of one-to-many sequence problems where you have a single image as input and you have to predict the image description in the form of a word sequence. Similarly, stock market prediction for the next X days, where input is the stock price of the previous Y days, is a classic example of many-to-many sequence problems.

In this article you will see very basic examples of one-to-many and many-to-many problems. However, the concepts learned in this article will lay the foundation for solving advanced sequence problems, such as stock price prediction and automated image captioning that we will see in the upcoming articles.

## **One-to-Many Sequence Problems**

One-to-many sequence problems are the type of sequence problems where input data has one time-step and the output contains a vector of multiple values or multiple time-steps. In this section, we will see how to solve one-to-many sequence problems where the input has a single feature. We will then move on to see how to work with multiple features input to solve one-to-many sequence problems.

### **One-to-Many Sequence Problems with a Single Feature**

Let's first create a dataset and understand the problem that we are going to solve in this section.

#### **Creating the Dataset**

The following script imports the required libraries:

from numpy import array

from keras.preprocessing.text import one\_hot

from keras.preprocessing.sequence import pad\_sequences

from keras.models import Sequential

from keras.layers.core import Activation, Dropout, Dense

from keras.layers import Flatten, LSTM

from keras.layers import GlobalMaxPooling1D

from keras.models import Model

from keras.layers.embeddings import Embedding

from sklearn.model\_selection import train\_test\_split

from keras.preprocessing.text import Tokenizer

from keras.layers import Input

from keras.layers.merge import Concatenate

from keras.layers import Bidirectional

import pandas as pd

import numpy as np

import re

import matplotlib.pyplot as plt

And the following script creates the dataset:

X = list()

Y = list()

X = [x+3 for x in range(-2, 43, 3)]

for i in X:

 output\_vector = list()

 output\_vector.append(i+1)

 output\_vector.append(i+2)

 Y.append(output\_vector)

print(X)

print(Y)

Here is the output:

[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43]

[[2, 3], [5, 6], [8, 9], [11, 12], [14, 15], [17, 18], [20, 21], [23, 24], [26, 27], [29, 30], [32, 33], [35, 36], [38, 39], [41, 42], [44, 45]]

Our input contains 15 samples with one time-step and one feature value. For each value in the input sample, the corresponding output vector contains the next two integers. For instance, if the input is 4, the output vector will contain values 5 and 6. Hence, the problem is a simple one-to-many sequence problem.

The following script reshapes our data as required by the LSTM:

X = np.array(X).reshape(15, 1, 1)

Y = np.array(Y)

We can now train our models. We will train simple and stacked LSTMs.

#### **Solution via Simple LSTM**

model = Sequential()

model.add(LSTM(50, activation='relu', input\_shape=(1, 1)))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

model.fit(X, Y, epochs=1000, validation\_split=0.2, batch\_size=3)

Once the model is trained we can make predictions on the test data:

test\_input = array([10])

test\_input = test\_input.reshape((1, 1, 1))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The test data contains a value 10. In the output, we should get a vector containing 11 and 12. The output I received is [10.982891 12.109697] which is actually very close to the expected output.

#### **Solution via Stacked LSTM**

The following script trains stacked LSTMs on our data and makes prediction on the test points:

model = Sequential()

model.add(LSTM(50, activation='relu', return\_sequences=True, input\_shape=(1, 1)))

model.add(LSTM(50, activation='relu'))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The answer is [11.00432 11.99205] which is very close to the actual output.

#### **Solution via Bidirectional LSTM**

The following script trains a bidirectional LSTM on our data and then makes a prediction on the test set.

from keras.layers import Bidirectional

model = Sequential()

model.add(Bidirectional(LSTM(50, activation='relu'), input\_shape=(1, 1)))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output I received is [11.035181 12.082813]

### **One-to-Many Sequence Problems with Multiple Features**

In this section we will see one-to-many sequence problems where input samples will have one time-step, but two features. The output will be a vector of two elements.

#### **Creating the Dataset**

As always, the first step is to create the dataset:

nums = 25

X1 = list()

X2 = list()

X = list()

Y = list()

X1 = [(x+1)\*2 for x in range(25)]

X2 = [(x+1)\*3 for x in range(25)]

for x1, x2 in zip(X1, X2):

 output\_vector = list()

 output\_vector.append(x1+1)

 output\_vector.append(x2+1)

 Y.append(output\_vector)

X = np.column\_stack((X1, X2))

print(X)

Our input dataset looks like this:

[[ 2 3]

 [ 4 6]

 [ 6 9]

 [ 8 12]

 [10 15]

 [12 18]

 [14 21]

 [16 24]

 [18 27]

 [20 30]

 [22 33]

 [24 36]

 [26 39]

 [28 42]

 [30 45]

 [32 48]

 [34 51]

 [36 54]

 [38 57]

 [40 60]

 [42 63]

 [44 66]

 [46 69]

 [48 72]

 [50 75]]

You can see each input time-step consists of two features. The output will be a vector which contains the next two elements that correspond to the two features in the time-step of the input sample. For instance, for the input sample [2, 3], the output will be [3, 4], and so on.

Let's reshape our data:

X = np.array(X).reshape(25, 1, 2)

Y = np.array(Y)

#### **Solution via Simple LSTM**

model = Sequential()

model.add(LSTM(50, activation='relu', input\_shape=(1, 2)))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

model.fit(X, Y, epochs=1000, validation\_split=0.2, batch\_size=3)

Let's now create our test point and see how well our algorithm performs:

test\_input = array([40, 60])

test\_input = test\_input.reshape((1, 1, 2))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The input is [40, 60], the output should be [41, 61]. The output predicted by our simple LSTM is [40.946873 60.941723] which is very close to the expected output.

#### **Solution via Stacked LSTM**

model = Sequential()

model.add(LSTM(50, activation='relu', return\_sequences=True, input\_shape=(1, 2)))

model.add(LSTM(50, activation='relu'))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

test\_input = array([40, 60])

test\_input = test\_input.reshape((1, 1, 2))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output in this case is: [40.978477 60.994644]

#### **Solution via Bidirectional LSTM**

from keras.layers import Bidirectional

model = Sequential()

model.add(Bidirectional(LSTM(50, activation='relu'), input\_shape=(1, 2)))

model.add(Dense(2))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The output obtained is: [41.0975 61.159065]

## **Many-to-Many Sequence Problems**

In one-to-many and many-to-one sequence problems, we saw that the output vector can contain multiple values. Depending upon the problem, an output vector containing multiple values can be considered as having single (since the output contains one time-step data in strict terms) or multiple (since one vector contains multiple values) outputs.

However, in some sequence problems, we want multiple outputs divided over time-steps. In other words, for each time-step in the input, we want a corresponding time-step in the output. Such models can be used to solve many-to-many sequence problems with variable lengths.

### **Encoder-Decoder Model**

To solve such sequence problems, the encoder-decoder model has been designed. The encoder-decoder model is basically a fancy name for neural network architecture with two LSTM layers.

The first layer works as an encoder layer and encodes the input sequence. The decoder is also an LSTM layer, which accepts three inputs: the encoded sequence from the encoder LSTM, the previous hidden state, and the current input. During training the actual output at each time-step is used to train the encoder-decoder model. While making predictions, the encoder output, the current hidden state, and the previous output is used as input to make predictions at each time-step. These concepts will become more understandable when you will see them in action in an upcoming section.

### **Many-to-Many Sequence Problems with Single Feature**

In this section we will solve many-to-many sequence problems via the encoder-decoder model, where each time-step in the input sample will contain one feature.

Let's first create our dataset.

#### **Creating the Dataset**

X = list()

Y = list()

X = [x for x in range(5, 301, 5)]

Y = [y for y in range(20, 316, 5)]

X = np.array(X).reshape(20, 3, 1)

Y = np.array(Y).reshape(20, 3, 1)

The input X contains 20 samples where each sample contains 3 time-steps with one feature. One input sample looks like this:

[[[ 5]

 [ 10]

 [ 15]]

You can see that the input sample contains 3 values that are basically 3 consecutive multiples of 5. The corresponding output sequence for the above input sample is as follows:

[[[ 20]

 [ 25]

 [ 30]]

The output contains the next three consecutive multiples of 5. You can see the output in this case is different from what we have seen in the previous sections. For the encoder-decoder model, the output should also be converted into a 3D format containing the number of samples, time-steps, and features. This is because the decoder generates an output per time-step.

We have created our dataset; the next step is to train our models. We will train stacked LSTM and bidirectional LSTM models in the following sections.

#### **Solution via Stacked LSTM**

The following script creates the encoder-decoder model using stacked LSTMs:

from keras.layers import RepeatVector

from keras.layers import TimeDistributed

model = Sequential()

*# encoder layer*

model.add(LSTM(100, activation='relu', input\_shape=(3, 1)))

*# repeat vector*

model.add(RepeatVector(3))

*# decoder layer*

model.add(LSTM(100, activation='relu', return\_sequences=True))

model.add(TimeDistributed(Dense(1)))

model.compile(optimizer='adam', loss='mse')

print(model.summary())

In the above script, the first LSTM layer is the encoder layer.

Next, we have added the repeat vector to our model. The repeat vector takes the output from the encoder and feeds it repeatedly as input at each time-step to the decoder. For instance, in the output we have three time-steps. To predict each output time-step, the decoder will use the value from the repeat vector, the hidden state from the previous output and the current input.

Next we have a decoder layer. Since the output is in the form of a time-step, which is a 3D format, the return\_sequences for the decoder model has been set True. The TimeDistributed layer is used to individually predict the output for each time-step.

The model summary for the encoder-decoder model created in the script above is as follows:

Layer (type) Output Shape Param #

=================================================================

lstm\_40 (LSTM) (None, 100) 40800

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

repeat\_vector\_7 (RepeatVecto (None, 3, 100) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_41 (LSTM) (None, 3, 100) 80400

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

time\_distributed\_7 (TimeDist (None, 3, 1) 101

=================================================================

Total params: 121,301

Trainable params: 121,301

Non-trainable params: 0

You can see that the repeat vector only repeats the encoder output and has no parameters to train.

The following script trains the above encoder-decoder model.

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

Let's create a test-point and see if our encoder-decoder model is able to predict the multi-step output. Execute the following script:

test\_input = array([300, 305, 310])

test\_input = test\_input.reshape((1, 3, 1))

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

Our input sequence contains three time-step values 300, 305 and 310. The output should be the next three multiples of 5 i.e. 315, 320 and 325. I received the following output:

[[[316.02878]

 [322.27145]

 [328.5536 ]]]

You can see that the output is in 3D format.

#### **Solution via Bidirectional LSTM**

Let's now create encoder-decoder model with bidirectional LSTMs and see if we can get better results:

from keras.layers import RepeatVector

from keras.layers import TimeDistributed

model = Sequential()

model.add(Bidirectional(LSTM(100, activation='relu', input\_shape=(3, 1))))

model.add(RepeatVector(3))

model.add(Bidirectional(LSTM(100, activation='relu', return\_sequences=True)))

model.add(TimeDistributed(Dense(1)))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

The above script trains the encoder-decoder model via bidirectional LSTM. Let's now make predictions on the test point i.e. [300, 305, 310].

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

Here is the output:

[[[315.7526 ]

 [321.47153]

 [327.94025]]]

The output I got via bidirectional LSTMs is better than what I got via the simple stacked LSTM-based encoder-decoder model.

### **Many-to-Many Sequence Problems with Multiple Features**

As you might have guessed it by now, in many-to-many sequence problems, each time-step in the input sample contains multiple features.

#### **Creating the Dataset**

Let's create a simple dataset for our problem:

X = list()

Y = list()

X1 = [x1 for x1 in range(5, 301, 5)]

X2 = [x2 for x2 in range(20, 316, 5)]

Y = [y for y in range(35, 331, 5)]

X = np.column\_stack((X1, X2))

In the script above we create two lists X1 and X2. The list X1 contains all the multiples of 5 from 5 to 300 (inclusive) and the list X2 contains all the multiples of 5 from 20 to 315 (inclusive). Finally, the list Y, which happens to be the output contains all the multiples of 5 between 35 and 330 (inclusive). The final input list X is a column-wise merger of X1 and X2.

As always, we need to reshape our input X and output Y before they can be used to train LSTM.

X = np.array(X).reshape(20, 3, 2)

Y = np.array(Y).reshape(20, 3, 1)

You can see the input X has been reshaped into 20 samples of three time-steps with 2 features where the output has been reshaped into similar dimensions but with 1 feature.

The first sample from the input looks like this:

[[ 5 20]

[ 10 25]

[ 15 30]]

The input contains 6 consecutive multiples of integer 5, three each in the two columns. Here is the corresponding output for the above input sample:

[[ 35]

[ 40]

[ 45]]

As you can see, the output contains the next three consecutive multiples of 5.

Let's now train our encoder-decoder model to learn the above sequence. We will first train a simple stacked LSTM-based encoder-decoder.

#### **Solution via Stacked LSTM**

The following script trains the stacked LSTM model. You can see that the input shape is now (3, 2) corresponding to three time-steps and two features in the input.

from keras.layers import RepeatVector

from keras.layers import TimeDistributed

model = Sequential()

model.add(LSTM(100, activation='relu', input\_shape=(3, 2)))

model.add(RepeatVector(3))

model.add(LSTM(100, activation='relu', return\_sequences=True))

model.add(TimeDistributed(Dense(1)))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

Let's now create a test point that will be used for making a prediction.

X1 = [300, 305, 310]

X2 = [315, 320, 325]

test\_input = np.column\_stack((X1, X2))

test\_input = test\_input.reshape((1, 3, 2))

print(test\_input)

The test point looks like this:

[[[300 315]

 [305 320]

 [310 325]]]

The actual output of the above test point is [330, 335, 340]. Let's see what are model predicts:

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

The predicted output is:

[[[324.5786 ]

 [328.89658]

 [335.67603]]]

The output is far from being correct.

#### **Solution via Bidirectional LSTM**

Let's now train the encoder-decoder model based on bidirectional LSTMs and see if we can get improved results. The following script trains the model.

from keras.layers import RepeatVector

from keras.layers import TimeDistributed

model = Sequential()

model.add(Bidirectional(LSTM(100, activation='relu', input\_shape=(3, 2))))

model.add(RepeatVector(3))

model.add(Bidirectional(LSTM(100, activation='relu', return\_sequences=True)))

model.add(TimeDistributed(Dense(1)))

model.compile(optimizer='adam', loss='mse')

history = model.fit(X, Y, epochs=1000, validation\_split=0.2, verbose=1, batch\_size=3)

The following script makes predictions on the test set:

test\_output = model.predict(test\_input, verbose=0)

print(test\_output)

Here is the output:

[[[330.49133]

 [335.35327]

 [339.64398]]]

The output achieved is pretty close to the actual output i.e. [330, 335, 340]. Hence our bidirectional LSTM outperformed the simple LSTM.

## **Conclusion**

This is the second part of my article on "Solving Sequence Problems with LSTM in Keras" ([part 1 here](https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras/)). In this article you saw how to solve one-to-many and many-to-many sequence problems in LSTM. You also saw how the encoder-decoder model can be used to predict multi-step outputs. The encoder-decoder model is used in a variety of [natural language processing](https://stackabuse.com/what-is-natural-language-processing/) applications such as neural machine translation and chat-bot development.

In the upcoming article, we will see the application of the encoder-decoder model in NLP.