


Solving Sequence Problems with LSTM in Keras

By  Usman Malik (https://twitter.com/usman_malikk) • September 13, 2019 •
3 Comments (/solving-sequence-problems-with-lstm-in-keras/#disqus_thread)



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 problems 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 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 (</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 (</using-machine-learning-to-predict-the-weather-part-1/>), etc., based on historic data. Since, text is also a sequence of words, the knowledge gained in this article can also be used to solve natural language processing (</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 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 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 $30 \times 15 = 450$. Let's see what value do 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 differ 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 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 problem 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 3-dimensional shape. Our input has 25 samples, where each sample consist 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 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 $55 \times 80 = 4400$. 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 prediction 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 single LSTM layer. You can play with different combination 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 single time-step of one or multiple features. Data with 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)
```

Subscribe to our Newsletter

Get occasional tutorials, guides, and reviews in your inbox. No spam ever.

Unsubscribe at any time.

The above script converts the list `X` into 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')
```

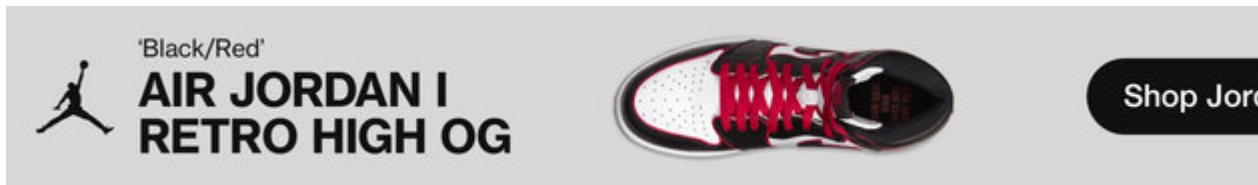
The following script trains the model and makes predictions on the test sequence which is 50, 51, and 52.

```
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 have 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 prediction 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 value 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]]
```

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 (</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 encoder-decoder mechanism that is most commonly used to create chatbots. Till then, happy coding :)

📁 [python \(/tag/python/\)](/tag/python/), [keras \(/tag/keras/\)](/tag/keras/), [machine learning \(/tag/machine-learning/\)](/tag/machine-learning/)

ire?

%20Problems%20with%20LSTM%20in%20Keras&url=https://stackabuse.com/solving-tm-in-keras/)

.com/sharer/sharer.php?u=https://stackabuse.com/solving-sequence-problems-with-

n/share?url=https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras/)

com/shareArticle?mini=true%26url=https://stackabuse.com/solving-sequence-s/%26source=https://stackabuse.com)



(</author/usman/>)

About Usman Malik (</author/usman/>)

🏠 Paris (France) 🐦 Twitter (https://twitter.com/usman_malikk)

Programmer | Blogger | Data Science Enthusiast | PhD To Be | Arsenal FC for Life

Subscribe to our Newsletter

Get occasional tutorials, guides, and reviews in your inbox. No spam ever. Unsubscribe at any time.