Solving Sequence Problems with LSTM in Keras: Part 2

By Usman Malik (https://twitter.com/usman_malikk) • September 19, 2019 •
2 Comments (/solving-sequence-problems-with-lstm-in-keras-part-2/#disqus_thread)

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 (/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 prediction 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 contain 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 than 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. The 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 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 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 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 (/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 encoder-decoder model can be used to predict multi-step outputs. The encoder-decoder model is used in a variety of natural language processing (/what-is-natural-language-processing/) applications such as neural machine translation and chatbot development.

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

python (/tag/python/), keras (/tag/keras/), machine learning (/tag/machine-learning/)

ith%20LSTM%20in%20Keras%3A%20Part%202&url=https://stackabuse.com/solving-

hp?u=https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras-part-2/)

:ackabuse.com/solving-sequence-problems-with-lstm-in-keras-part-2/)

ii=true%26url=https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras-

(/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