Machine Learning
A Quantitative Approach

Henry H. Liu

PerfMath

ISBN-13: 978-1985136625
ISBN-10: 1985136627

10 9 8 7 6 5 4 3 2 1
03092018

Appendix D RNN/LSTM Example Implementations with Keras/TensorFlow

This appendix demonstrates an example RNN implementation with Keras/TensorFlow. Keras is a Python-based frontend capable of running on top of a more powerful deep learning engine like TensorFlow. In this section, we demonstrate how to install Keras to work with TensorFlow, and then present an example of using the Keras LSTM stateful and stateless models to predict time-series sequences.

D.1 INSTALLING KERAS/TENSORFLOW

Keras installation is documented in detail at https://keras.io/#installation. I referred to the instructions given at https://www.tensorflow.org/install/install_mac and installed TensorFlow on my MacBook Pro with macOS Sierra version 10.12.6 by choosing the "native" pip option. It was as easy as executing the following command:

```
$sudo easy_install --upgrade pip
$sudo easy_install --upgrade six
$pip3 install tensorflowpip3 install tensorflow
```

Then, I executed the following commands to install Keras from the GitHub source:

```
$cd /Users/henryliu/Documents/ml_dev
$git clone https://github.com/keras-team/keras.git
$cd keras
$sudo python3 setup.py install
```

I got the following error during the above installation:

```
fatal error: 'yaml.h' file not found
```

However, the issue was resolved after I executed the following command:

```
henryliu:keras henryliu$ sudo pip3 install --upgrade keras
```

Then, I clicked Eclipse → Preferences → PyDev → Interpreters → Python Interpreter as shown in Figure D.1, to make sure Keras 2.1.5 was available for use.

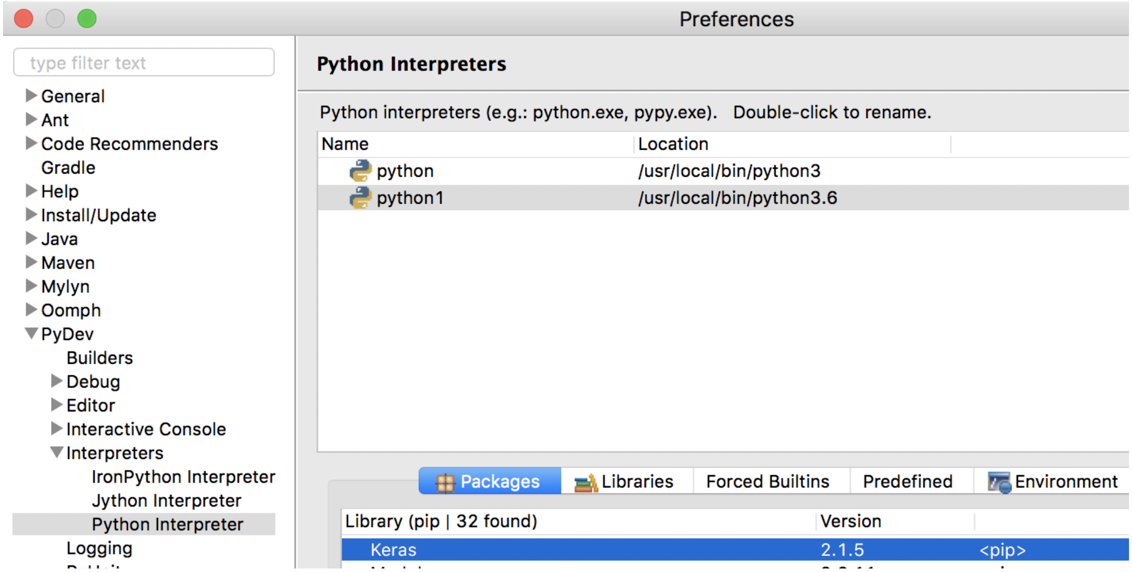

Figure D.1 Keras 2.1.5 made available on Eclipse/PyDev.

Now open the *lstm_time_series,py* file in the directory of *ch11* of the *ml_quantitative* project from this book's download and verify that it runs.

D.2 THE KERAS LSTM TIME SERIES EXAMPLE

In the *ch11* directory of the project, you should find two scripts: *lstm_stateful.py*, which is the original example from Keras's *examples* directory, and *lstm_time_series.py*, which is the same example I adapted to help make it easier to understand this example. I suggest that you run and examine the original script first and note down the questions you have, which might be explained next with the adapted script.

This is a very interesting LSTM example. However, it may not be obvious how it works at the first glance. Before presenting the adapted script, let's review a few basic generic Python examples first.

D.2.1 PANADS DATAFRAME

The LSTM example to be presented uses a Pandas DataFrame object to hold the time series data. Here is how we can generate an arbitrary Pandas DataFrame object:

```
$python3
>>> import pandas as pd
>>> df = pd.DataFrame(np.random.randn(8,4))
```

```
>>> df
          0         1         2         3
0  0.058779  1.070018  0.817022  0.910342
1 -2.871662 -0.346887 -0.313775 -0.082175
2  0.496465  0.835879  1.435297  0.472730
3 -0.602593 -1.569787  0.166828 -1.476208
4  0.205998  0.682439 -0.289690  0.361073
5  0.808703 -1.249511  0.289556  1.530001
6  1.689375 -1.168729 -0.563632 -0.053785
7  0.667124  1.104330 -0.508920 -0.655200
```

Now we have a Pandas DataFrame object with 4 columns and 8 rows. You can query the shape of this Pandas DataFrame object by issuing the following command:

```
>>> df.shape
(8, 4)
```

That is, we can take this as an 8x4 matrix. We can query it further by column as follows:

```
>>> df[0]
0    0.058779
1   -2.871662
2    0.496465
3   -0.602593
4    0.205998
5    0.808703
6    1.689375
7    0.667124
```

This gives the entire column. If we want to limit the rows for a particular column, we can do:

```
>>> df[1][2:]
2    0.835879
3   -1.569787
4    0.682439
5   -1.249511
6   -1.168729
7    1.104330
Name: 1, dtype: float64
```

That is, we got rows 2-7 of column 1, with rows 0-1 skipped, since both columns and rows are zero-index based.

We can even shift a column down, e.g.:

```
>>> x=df[0].shift(1)
>>> x
0         NaN
1    0.058779
2   -2.871662
3    0.496465
4   -0.602593
```

```
5    0.205998
6    0.808703
7    1.689375
```

The LSTM example to be examined uses the above Pandas DataFrame functions.

Next, we review the *numpy's repeat* function.

D.2.2 NUMPY'S REPEAT FUNCTION

Here is an example of the use of NumPy's *repeat* function:

```
>>> import numpy as np
>>> x=np.array([[1,2],[3,4]])
>>> x
array([[1, 2],
      [3, 4]])
>>> np.repeat(x, 2)
array([1, 1, 2, 2, 3, 3, 4, 4])
```

In this case, `x` is a 2x2 matrix. The above `repeat` function repeats each element by n times, here n = 2, and then flattens the matrix row by row into one row.

If we specify `axis = 0`, we would get:

```
>>> np.repeat(x, 2, axis=0)
array([[1, 2],
      [1, 2],
      [3, 4],
      [3, 4]])
```

That is, each row is repeated twice and we end up with four rows with the first two rows and the last two rows identical, respectively.

If we specify `axis = 1`, we would get:

```
>>> np.repeat(x, 2, axis=1)
array([[1, 1, 2, 2],
      [3, 3, 4, 4]])
```

That is, each element is repeated twice without being flattened, and we end up with two rows and four columns.

Now, we are ready to present the adapted LSTM example that models a time series sequence of limited length.

D.3 AN LSTM EXAMPLE THAT MODELS A TIME SERIES SEQUENCE OF LIMITED LENGTH

Listing D.1 shows the adapted Keras LSTM example that models a time series sequence of limited length. First, run this example in your env and make sure you get the similar results as shown in Listing D.2 and Figures D.2 and 3. If you ran the original Keras *lstm_staeful.py* script, you would notice that the following changes had been made with the adapted example:

- We changed the `model.fit` parameter `verbose` from 1 to 0 so that the output would not show the lengthy training steps within each epoch.
- All charts have `x` and `y` labels, and the title includes the `tsteps` and `lahead` parameter values used.
- Figure D.2 shows the training data and expected output as the moving average of the subsequences with their rolling window length defined by the parameter `tsteps`. For example, with `tsteps` = 2 and the first two data points of –0.084532 and 0.021696, the first output would be computed as (–0.084532 + 0.021696) / 2 = –0.031418. In general, the moving average with a given window length of `tsteps` is defined as

$$y_n = \frac{1}{tsteps} \sum_{i=n}^{n-tsteps+1} x_i$$

- Fig. D.3 shows the predicted versus the expected with the stateful and stateless LSTM models, respectively. The original example shows the differences between the predicted and expected, which is less obvious visually in terms of how well the two agree with each other. We also added the RMSE value to the title to help evaluate the accuracy of the model. As is seen, the RMSE values are 0.014 and 0.031 for the stateful and stateless models, respectively, indicating that the stateful model predicted better than the stateless model. You can visually verify this by examining the two subplots in Figure D.3.

Next, we examine how the adapted script works.

Before we start, I suggest that you take a quick look at the script shown in Listing D.1 or on your PyDev/Eclipse IDE to get an idea of how it works in general. Then, I'll help you understand some details, such as:

- Lines 5-6 import Keras `Sequential`, `Dense` and `LSTM` models. The `Sequential` and `LSTM` models are obvious, but what about the `Dense` model? A Dense model in Keras is just a fully-connected layer. As we know, all ANN models have at least a hidden layer and an output layer with a specified number of units or neurons. Therefore, a layer is one of the most basic elements in composing an ANN model.
- Lines 7-8 import the `sqrt` and `mean_squared_error` functions for computing RMSE.
- Lines 9-12 define either the model parameters or job running parameters. Out of those parameters, `tsteps` and `lahead` are not so obvious in terms of what they are. In fact, `tsteps` specifies the length of the sub-sequence for calculating the moving average as the output corresponding to that subsequence, while `lahead` specifies the length of the input subsequence for training the LSTM model. We will see more how these two parameters affect the outcome of an LSTM model later.
- Lines 22-25 define a random number generation function for a given amplitude and the length of the array. This is the input sequence that this example uses. You can replace it with an input sequence of your own for a real application.
- Line 26 defines the number of data points to drop, based on the values of `tsteps` and `lahead`.
- Line 28 uses the function `rolling` to compute expected output as the target values for the supervised LSTM training.
- Lines 40-49 output the input and output data characteristics.

- Lines 50-59 plot the input versus expected output, as shown in Figure D.2. Note that you need to click on the red-cross icon at the upper left corner in order to continue the script execution.
- Lines 60-68 create the model, with a parameter named `stateful` passed in. This is how an LSTM model is created in Keras. It starts with a Sequential mode, then adds an LSTM block with 20 units, an input shape defined by (`lahead`, 1) or (`input seq length`, `output seq length`), a batch size, and a `stateful` parameter. The difference between a stateful and a stateless LSTM model is about whether the state is maintained between batches. These two models are trained differently as we will see later.
- Lines 72-94 define a function about how to split data. A ratio of 0.8 is hard-coded, which means that 80% will be used for training and 20% for testing.
- Lines 101-104 define how the stateful LSTM model is trained, while line 105 calls the `predict` function to predict on the test time series sequence.
- Line 110 defines how the stateless LSTM model is trained, while line 111 calls the `predict` function to predict on the test time series sequence.

You may have noticed that the stateful and stateless models are trained differently. The stateful model is trained epoch-by-epoch, with the state reset by calling the `reset_states` function from one epoch to the next. However, the stateless model does not have such a constraint – it has the `epochs` parameter passed to the `fit` function all in one, as shown in line 110. The details of how the Keras LSTM model works internally are beyond the scope of this text, and you can pursue it further by consulting the Keras documentation or examining the Keras source code.

Listing D.1 lstm_time_series.py (with comments removed to save space)

```
1   from __future__ import print_function
2   import numpy as np
3   import matplotlib.pyplot as plt
4   import pandas as pd
5   from keras.models import Sequential
6   from keras.layers import Dense, LSTM
7   from math import sqrt
8   from sklearn.metrics import mean_squared_error

9   input_len = 1000
10  tsteps = 2 #rolling window length
11  lahead = 1
12  batch_size = 1
13  epochs = 10

14  print("*" * 33)
15  if lahead >= tsteps:
16      print("STATELESS LSTM WILL ALSO CONVERGE")
17  else:
18      print("STATELESS LSTM WILL NOT CONVERGE")
19  print("*" * 33)

20  np.random.seed(1986)

21  print('Generating Data...')
```

```
def gen_uniform_amp(amp=1, xn=10000):
    data_input = np.random.uniform(-1 * amp, +1 * amp, xn)
    data_input = pd.DataFrame(data_input)
    return data_input

to_drop = max(tsteps - 1, lahead - 1)
data_input = gen_uniform_amp(amp=0.1, xn=input_len + to_drop)

# set the target to be a N-point average of the input
expected_output = data_input.rolling(window=tsteps, center=False).mean()

if lahead > 1:
    print("data_input_values:\n", data_input.values)
    data_input = np.repeat(data_input.values, repeats=lahead, axis=1)
    print("data_input after repeat\n", data_input)
    data_input = pd.DataFrame(data_input)
    for i, c in enumerate(data_input.columns):
        data_input[c] = data_input[c].shift(i)

# drop the nan
expected_output = expected_output[to_drop:]
data_input = data_input[to_drop:]

print('Input shape:', data_input.shape)
print('Output shape:', expected_output.shape)
print('Input head: ')
print(data_input.head())
print('Output head: ')
print(expected_output.head())
print('Input tail: ')
print(data_input.tail())
print('Output tail: ')
print(expected_output.tail())

print('Plotting input and expected output')
n = 50
#n = input_len
plt.plot(data_input[0][:n], '-')
plt.plot(expected_output[0][:n], '-')
plt.xlabel('x')
plt.ylabel ('y')
plt.legend(['Input', 'Expected output'])
plt.title('Input vs Expected (tsteps = %i)' %(tsteps))
plt.show()

def create_model(stateful):
    model = Sequential()
    model.add(LSTM(20,
              input_shape=(lahead, 1),
              batch_size=batch_size,
              stateful=stateful))
```

```
    model.add(Dense(1))
    model.compile(loss='mse', optimizer='adam')
    return model

print('Creating Stateful Model...')
model_stateful = create_model(stateful=True)

# split train/test data
def split_data(x, y, ratio=0.8):
    to_train = int(input_len * ratio)
    # tweak to match with batch_size
    to_train -= to_train % batch_size
    print("to_train = ", to_train)

    x_train = x[:to_train]
    y_train = y[:to_train]
    x_test = x[to_train:]
    y_test = y[to_train:]

    # tweak to match with batch_size
    to_drop = x.shape[0] % batch_size
    if to_drop > 0:
        x_test = x_test[:-1 * to_drop]
        y_test = y_test[:-1 * to_drop]

    # some reshaping
    reshape_3 = lambda x: x.values.reshape((x.shape[0], x.shape[1], 1))
    x_train = reshape_3(x_train)
    x_test = reshape_3(x_test)

    reshape_2 = lambda x: x.values.reshape((x.shape[0], 1))
    y_train = reshape_2(y_train)
    y_test = reshape_2(y_test)

    return (x_train, y_train), (x_test, y_test)

(x_train, y_train), (x_test, y_test) = split_data(data_input, expected_output)
print('x_train.shape: ', x_train.shape)
print('y_train.shape: ', y_train.shape)
print('x_test.shape: ', x_test.shape)
print('y_test.shape: ', y_test.shape)

print('Training')
for i in range(epochs):
    print('Epoch', i + 1, '/', epochs)
    model_stateful.fit(x_train,
                       y_train,
                       batch_size=batch_size,
                       epochs=1,
                       verbose=0,
                       validation_data=(x_test, y_test),
                       shuffle=False)
```

```
    model_stateful.reset_states()

predicted_stateful = model_stateful.predict(x_test, batch_size=batch_size)

rmse = sqrt(mean_squared_error(y_test.flatten()[tsteps - 1:],
predicted_stateful.flatten()[tsteps - 1:]))
print('Stateful LSTM RMSE: %.3f' % rmse)
print('Creating Stateless Model...')
model_stateless = create_model(stateful=False)

model_stateless.fit(x_train,
                    y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=0,
                    validation_data=(x_test, y_test),
                    shuffle=False)

predicted_stateless = model_stateless.predict(x_test,
batch_size=batch_size)
rmse = sqrt(mean_squared_error(y_test.flatten()[tsteps - 1:],
predicted_stateless.flatten()[tsteps - 1:]))
print('Stateless LSTM RMSE: %.3f' % rmse)
```

Listing D.2 Console output of running the lstm_time_series.py script

```
Using TensorFlow backend.
*********************************
STATELESS LSTM WILL NOT CONVERGE
*********************************
Generating Data...
Input shape: (1000, 1)
Output shape: (1000, 1)
Input head:
       0
1 -0.084532
2  0.021696
3  0.079500
4  0.008981
5  0.040544
Output head:
       0
1 -0.035379
2 -0.031418
3  0.050598
4  0.044240
5  0.024763
Input tail:
         0
996   0.010251
```

```
997  -0.027833
998   0.003984
999   0.028471
1000 -0.057877
Output tail:
            0
996   0.025187
997  -0.008791
998  -0.011925
999   0.016227
1000 -0.014703
Plotting input and expected output
Creating Stateful Model...
to_train =  800
x_train.shape:  (800, 1, 1)
y_train.shape:  (800, 1)
x_test.shape:  (200, 1, 1)
y_test.shape:  (200, 1)
Training
Epoch 1 / 10
2018-03-24 00:09:13.361252: I tensorflow/core/platform/cpu_feature_guard.cc:140] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Epoch 2 / 10
Epoch 3 / 10
Epoch 4 / 10
Epoch 5 / 10
Epoch 6 / 10
Epoch 7 / 10
Epoch 8 / 10
Epoch 9 / 10
Epoch 10 / 10
Predicting
Stateful LSTM RMSE: 0.014
Creating Stateless Model...
Training
Predicting
Stateless LSTM RMSE: 0.031
Plotting Results
```

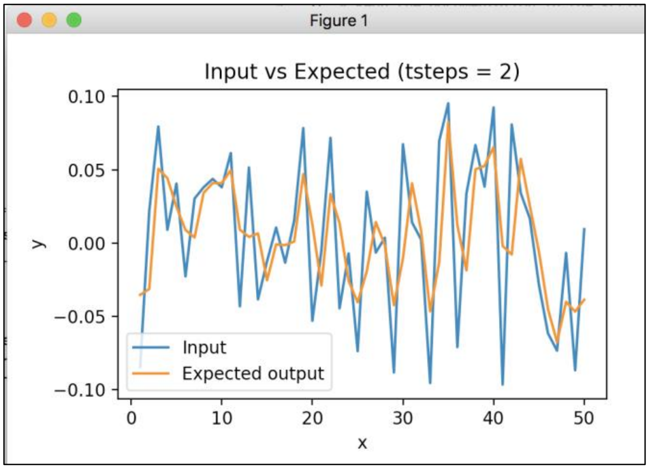

Figure D.2 Time series sequence input versus expected with a rolling window length of `tsteps = 2`.

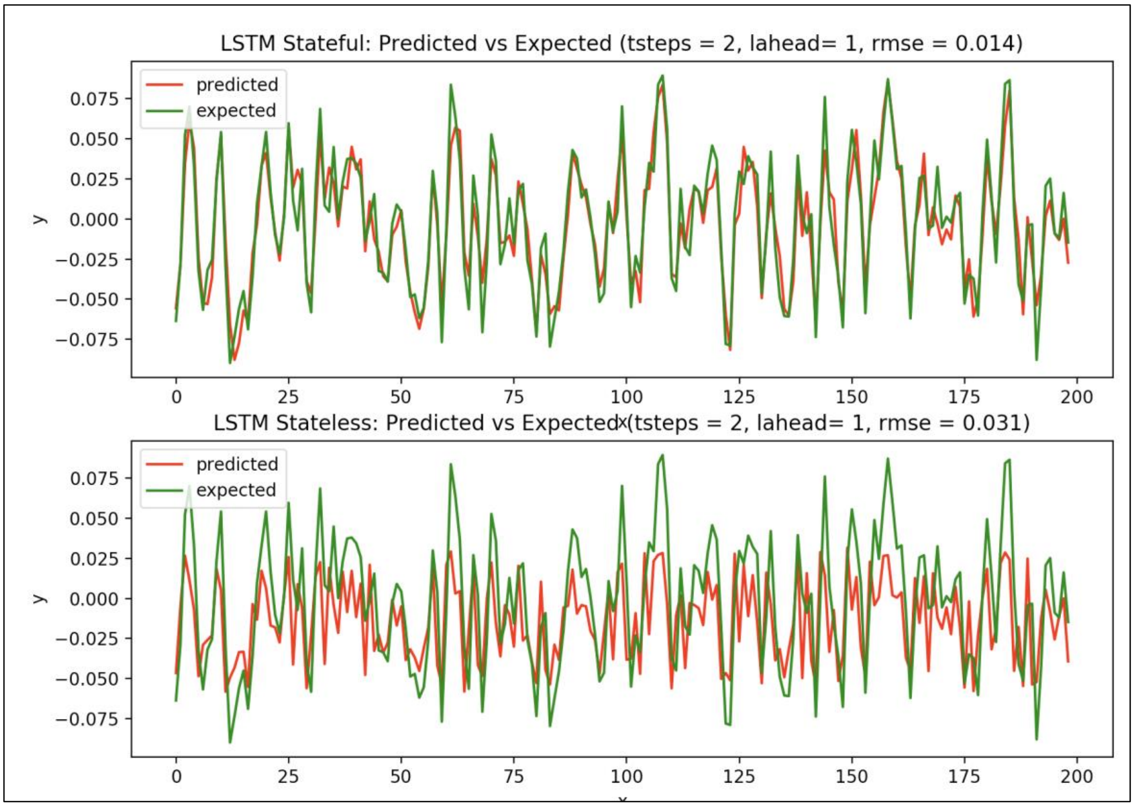

Figure D.3 Time series sequence modeled with stateful and stateless LSTM models, respectively.

D.3 MORE EXPERIMENTS WITH THE KERAS LSTM TIME SERIES EXAMPLE

Although this LSTM example is simple, it gives us sufficient opportunities to experiment. If you examine the comments from the source code of this example, you will find the bulk of it is about the parameter `lahead`. You might think that we should always use the same value for the `lahead` and `tsteps` parameters, in which case, both the stateful and stateless models converge, but according to the comments in the original source code of the example, one can also specify `lahead < tsteps`, in which case, the input subsequence length is smaller than the moving averaging rolling window view length and only the stateful model converges. Perhaps this latter case gives us an option to truncate a long input subsequence in order to speed up training stateful models.

In this section, we try a few more experiments to see how the parameters of `tsteps` and `lahead` work out with each other, and also how the stateful and stateless models compare with each other. First, we try an example with `tsteps = 2` and `lahead = 3`, and then an example with `tsteps = 3` and `lahead = 2`. The results are presented in the next two subsections.

D.3.1 STATEFUL VS STATELESS LSTM MODELS WITH TSTEPS = 2 AND LAHEAD = 3

First, Listing D.3 shows the input and output data characteristics. Note the following:

- The `values` attribute of the `data_input` *DataFarme* object gives a column vector, which is turned into a 3-column matrix after its `repeat` function is called.
- The next segment of the output shows how those 3 columns are shifted after the `shift` function is called on each column vector. Note the index '`c`' in `data_input[c]` that identifies the column vector with the given index `c`. The first column is not shifted as shift by '0' is no shift.
- The input head and tail outputs show how the input subsequences are prepared for training. For example, under "`input head`:", we see how the first 3 elements of column 0, [0.021696, 0.079500, 0.008981], have been turned into a row vector or subsequence with the index value of 4. This is used as input for every step that an LSTM model is trained.
- The output is just a single column vector with an output subsequence length of 1.

Figs. D.4 and 5 show the input versus expected output chart and the stateful versus stateless model predictions, respectively. In this case, the RMSE values are 0.004 and 0.001 for the stateful model and stateless model, respectively, with the stateless model performed better than the stateful model. However, both the stateful model and stateless model performed significantly better than the previous case with `tsteps` = 2 and `lahead` = 1, as shown in Fig. D.3.

Listing D.3 Input and output data characteristics with tsteps = 2 and lahead = 3

```
Using TensorFlow backend.
Generating Data...
data_input_values:
 [[ 0.01377506]
 [-0.08453234]
 [ 0.02169589]
 ...
```

```
 [ 0.02847093]
 [-0.05787658]
 [-0.09730742]]
data_input after repeat
 [[ 0.01377506  0.01377506  0.01377506]
 [-0.08453234 -0.08453234 -0.08453234]
 [ 0.02169589  0.02169589  0.02169589]
 ...
 [ 0.02847093  0.02847093  0.02847093]
 [-0.05787658 -0.05787658 -0.05787658]
 [-0.09730742 -0.09730742 -0.09730742]]
i =  0  c =  0 data_input[c] 0    0.013775
1   -0.084532
2    0.021696
3    0.079500
4    0.008981
Name: 0, dtype: float64
i =  0  c =  0 data_input[c] after 0    0.013775
1   -0.084532
2    0.021696
3    0.079500
4    0.008981
Name: 0, dtype: float64
i =  1  c =  1 data_input[c] 0    0.013775
1   -0.084532
2    0.021696
3    0.079500
4    0.008981
Name: 1, dtype: float64
i =  1  c =  1 data_input[c] after 0         NaN
1    0.013775
2   -0.084532
3    0.021696
4    0.079500
Name: 1, dtype: float64
i =  2  c =  2 data_input[c] 0    0.013775
1   -0.084532
2    0.021696
3    0.079500
4    0.008981
Name: 2, dtype: float64
i =  2  c =  2 data_input[c] after 0         NaN
1         NaN
2    0.013775
3   -0.084532
4    0.021696
Name: 2, dtype: float64
Input shape: (1000, 3)
Output shape: (1000, 1)
```

```
Input head:
          0         1         2
2  0.021696 -0.084532  0.013775
3  0.079500  0.021696 -0.084532
4  0.008981  0.079500  0.021696
5  0.040544  0.008981  0.079500
6 -0.022773  0.040544  0.008981
Input tail:
             0         1         2
997  -0.027833  0.010251  0.040122
998   0.003984 -0.027833  0.010251
999   0.028471  0.003984 -0.027833
1000 -0.057877  0.028471  0.003984
1001 -0.097307 -0.057877  0.028471
Output head:
          0
2 -0.031418
3  0.050598
4  0.044240
5  0.024763
6  0.008886
Output tail:
             0
997  -0.008791
998  -0.011925
999   0.016227
1000 -0.014703
1001 -0.077592
```

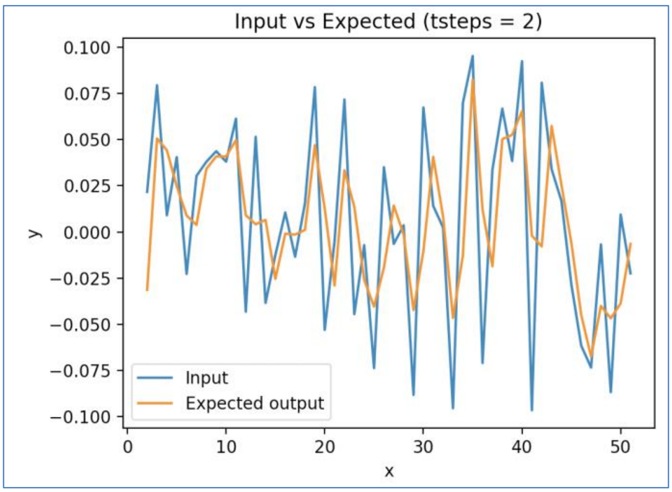

Figure D.4 Moving averaging of a random sequence with `tsteps` = 2.

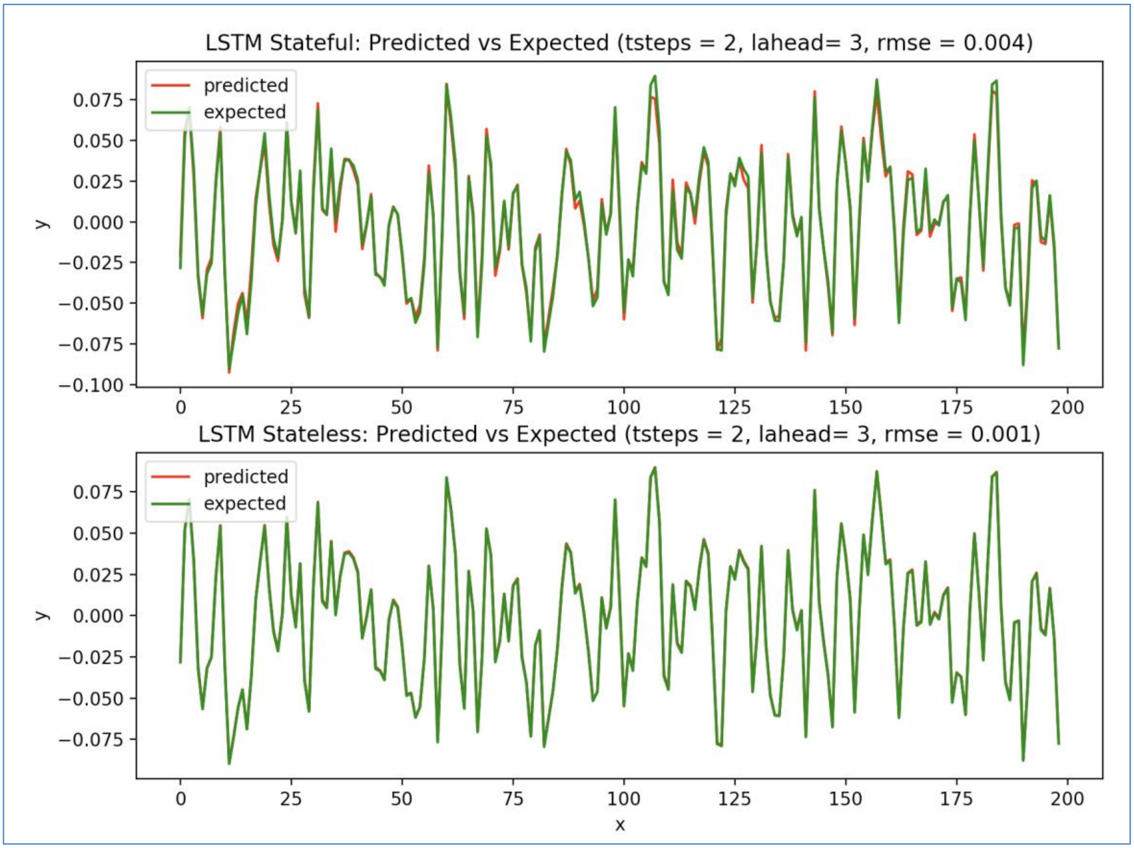

Figure D.5 Stateful versus stateless LSTM models for a random sequence with `tsteps` = 2 and `lahead` = 3.

D.3.2 STATEFUL VS STATELESS LSTM MODELS WITH TSTEPS = 3 AND LAHEAD = 2

This is a case that `lahead` < `tsteps`, with input versus the expected chart shown in Fig. D.6. When the script was run, a console output line was displayed, saying "STATELESS LSTM WILL NOT CONVERGE". This is confirmed by Figure D.7, which shows a good agreement between the expected and predicted with an RMSE value of 0.005 for the stateful model, but a not so good agreement between the expected and predicted with an RMSE value of 0.022 for the stateless model.

Next, we tried a run with `lahead` = `tsteps = 2`, as presented in the next section.

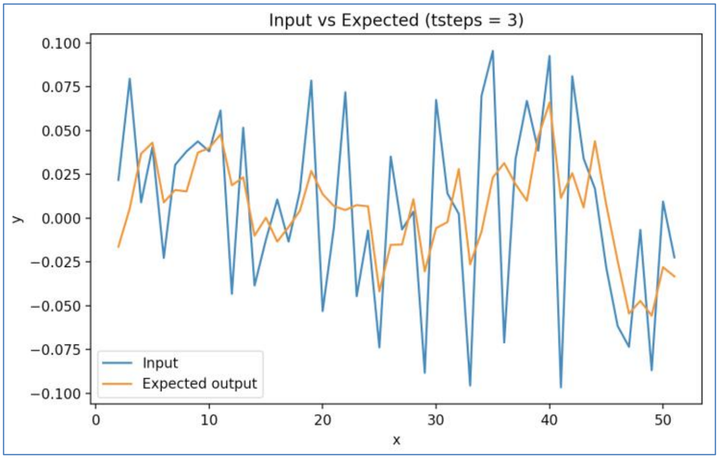

Figure D.6 Moving averaging of a random sequence with `tsteps` = 3.

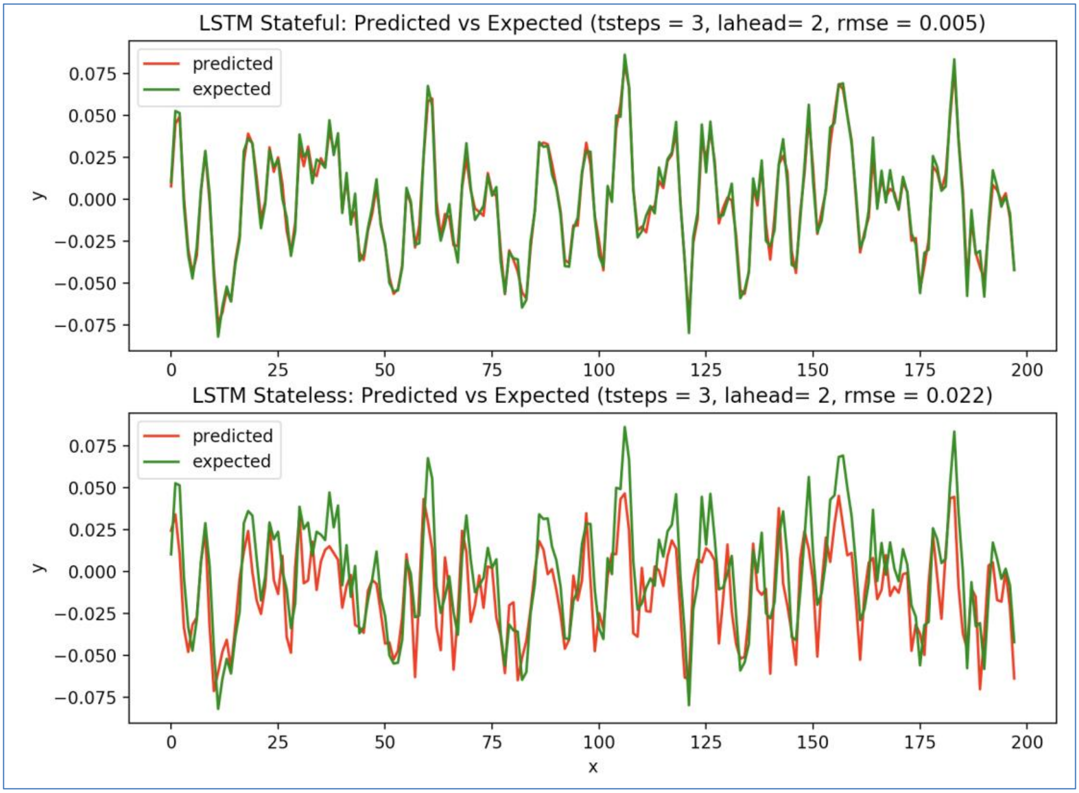

Figure D.7 Stateful versus stateless LSTM models for a random sequence with `tsteps` = 3 and `lahead` = 2.

D.3.3 STATEFUL VS STATELESS LSTM MODELS WITH TSTEPS = 2 AND LAHEAD = 2

You may wonder what would be the case if we had `tsteps` = `lahead` = 2. Fig. D.8 shows the results. In this case, we have a perfect prediction with the stateless model, but the stateful model performed poorly, with an RMSE value of 0.040. To verify that it is repeatable, I ran it the second time, with the similar results shown in Figure D.9. Thus, it seems that with this set of settings for the parameters of `tsteps` and `lahead`, the stateless model does perform significantly better than the stateful model.

This concludes our RNN/LSTM example with Keras/TensorFlow.

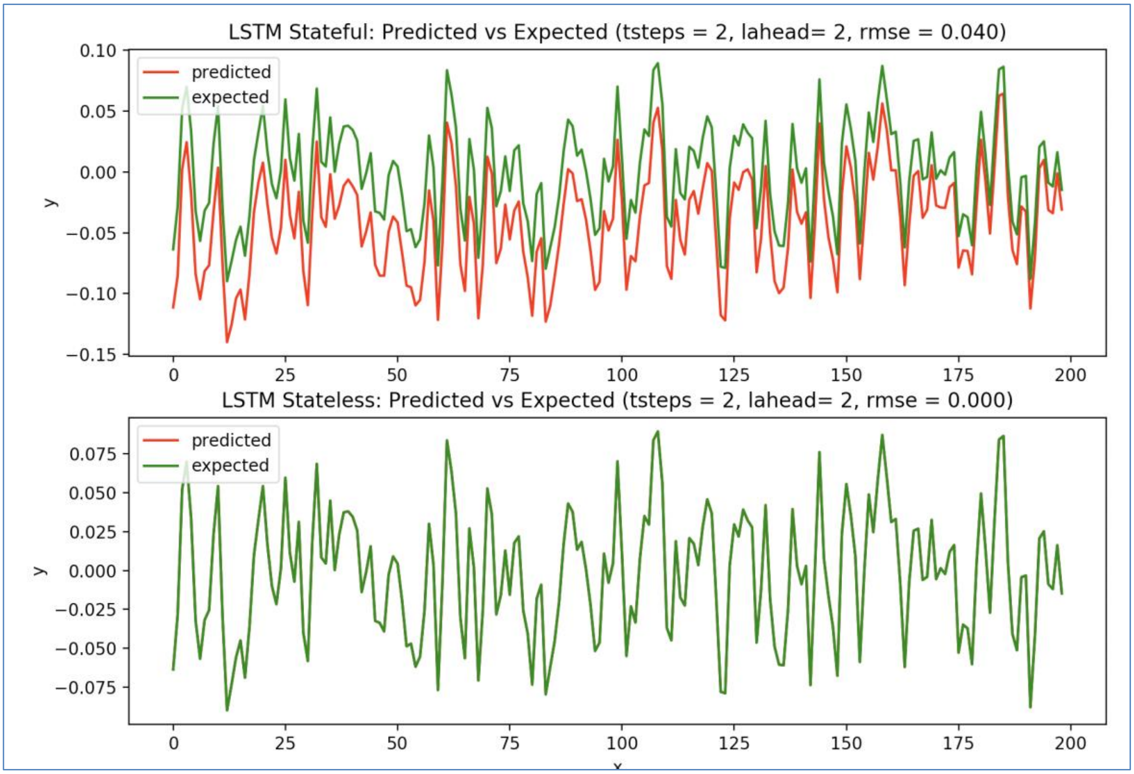

Figure D.8 Stateful versus stateless LSTM models for a random sequence with `tsteps` = 2 and `lahead` = 2.

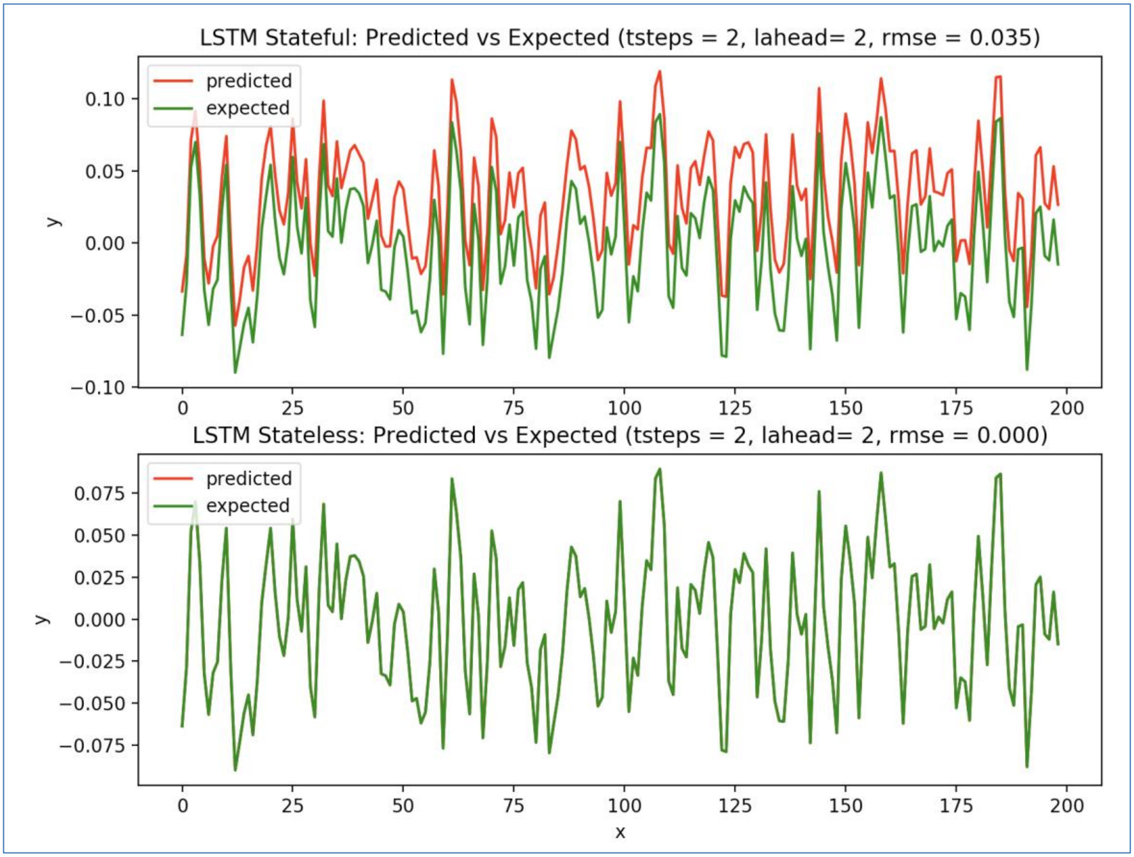

Figure D.9 Stateful versus stateless LSTM models for a random sequence with `tsteps` = 2 and `lahead` = 2 (second run).