Animation (Predicting Animation Sequences with LSTM)

Author: **Benjamin Kenwright**

In this tutorial, you'll learn how to perform time series forecasting that is used to solve animation sequence problems (e.g., in character animation systems).

Time series forecasting refers to the type of problems where you have to predict an outcome based on time dependent inputs. A typical example of time series data is recorded animation data where movements/signals change with time (considered a 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 (RNN) have been proven to efficiently solve sequence problems. Particularly, Long Short Term Memory Network (LSTM), which is a variation of RNN, is currently being used in a variety of domains to solve sequence problems.

- Ref: https://stackabuse.com/solving-sequence-problems-with-lstm-in-keras

The structure of this short tutorial is:

1. Basic input/output relationship (signals/data)
2. Simple test signals (1 to 1)
3. More complex test examples
4. Compared and evaluation

> Note the examples use simple test signals, such as, sin waves, so that you can experiment with input/output data relationships - without the need of loading in data and converting it to the correct format/length. Once you're comfortable with the underlaying principles, it's easy to modify the examples to fit your specific problem (e.g., loading in animation data for training).

Types of Animation Sequence Problems

Animation sequence problems can be broadly categorized into the following categories:

- 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 a single input (joint signal) and you want to predict a the type of motion for associated with it (i.e., a label).
- Many-to-One: In many-to-one sequence problems, you have a sequence of data as the input and you have to predict a single output. Motion classification is a prime example of many-to-one sequence problems where you have an input sequence of signals from multiple joints and you want to predict a single output tag (walk, run, climb motion).
- One-to-Many: In one-to-many sequence problems, you have single input and a sequence of outputs. A typical example is an animation signal and its corresponding possiblities.
- Many-to-Many: Many-to-many sequence problems involve a sequence input and a sequence output. For instance, animation for 20 seconds as input and animation of next 20 seconds as outputs.

> Predictive animation systmes are an example of many-to-many sequence problems where one movement sequence is the input and another movement sequence is the output.

This tutorial will show how LSTM and its different variants can be used to solve different animation sequence problems (from one-to-one to many-to-many sequence problems).

The examples/tests will use Python's Keras library.

After completing this tutorial, you should be able to create animation solutions to prediction motions - based on historic data. Since, the majority of the sequences will be on signals, the knowledge gained in this tutorial can also be used to solve other related processing tasks such as signal classification (music), signal generation and so on.

One-to-One Animation/Signal Sequence Problems

For one-to-one sequence problems, there is a **single input and a single output**. You'll learn of two types of sequence problems. First you'll see how to solve one-to-one sequence problems with a **single feature** and then you'll see how to solve one-to-one sequence problems with **multiple features**.

One-to-One Sequence with a Single Feature

You'll solve a one-to-one sequence problem where each time-step has a single feature.

> Note, at the top of the file, you'll import the required libraries that you'll use for the examples (i.e., Keras library and associatated helpers, like numpy)

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
import tensorflow as tf
import numpy as np
import math

# input a sin wave (input: t, output:signal aplitude)
# 1 input and 1 output (test training/signal)
X = []
Y = []
for i in range(0,100):
  X.append( i/100 )
  Y.append( math.sin( 2.0*math.pi*(i/100) ) )

# 100 inputs and 100 outputs
# input_shape => (samples, timestep, features )
X = np.array(X).reshape(100, 1, 1) # have to use numpy arrays!!! (lists can't be used)
Y = np.array(Y)

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())
```

```
model.fit(X, Y, epochs=20, batch_size=5, verbose=0)

test_input = np.array([0.5])
test_input = test_input.reshape((1, 1, 1))
test_output = model.predict(test_input, verbose=0)
print('input of 0.5, output is:', test_output)

## todo, plot input vs output, instead of 1 input, 2 inputs
# dt,0 -> sinwave, dt,1-> square save, ....
```

```
Model: "sequential_16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_28 (LSTM)               (None, 50)                10400
_________________________________________________________________
dense_24 (Dense)             (None, 1)                 51
=================================================================
Total params: 10,451
Trainable params: 10,451
Non-trainable params: 0
_________________________________________________________________
None
input of 0.5, output is: [[0.06043848]]
```

How it works?

1. Create the Dataset (input 0->1) (output sin wave amplitude 0->2pi)
2. In the script above, you create 100 inputs and 100 outputs. Each input consists of one time-step, which in turn contains a single feature. Each output value is the 'sin' value of the corresponding input value (i.e., multiplied by 2pi).

```
X = np.array(X).reshape(100, 1, 1)
```

3. The input to LSTM layer should be a 3D shape i.e. **(samples, time-steps, features)**. The samples are the number of samples in the input data. You have 100 samples in the input. The time-steps is the number of time-steps per sample. You have 1 time-step. Finally, features correspond to the number of features per time-step. You have one feature per time-step.
4. In the script above, you 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 your data has one time-step with one feature.
5. You train your model for 20 epochs with a batch size of 5. You can choose any number. Once the model is trained, you can make predictions on a new instance.
6. Let's say you want to predict the output for an input of 0.5. The actual output should be `sin(0.5*2*pi) = 0`. To test this, you first need to convert your input test data to the right shape i.e. 3D shape, as expected by LSTM.

```
test_input = array([0.5])
test_input = test_input.reshape((1, 1, 1))
test_output = model.predict(test_input, verbose=0)
print(test_output)
```

Note: It is important to mention that the outputs that you obtain by running the scripts may not always be the same. This is because the LSTM neural network initializes weights with random values and your values may change. But overall, the results should not vary too much.

Stacked LSTM

You'll now create a stacked LSTM and see if you can get better results. The dataset will remain the same (sin wave), however, the model will be changed. Look at the following script:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
import tensorflow as tf
import numpy as np
import math

# input a sin wave (input: t, output:signal aplitude)
# 1 input and 1 output (test training/signal)
X = []
Y = []
for i in range(0,100):
  X.append( i/100 )
  Y.append( math.sin( 2.0*math.pi*(i/100) ) )

# 100 inputs and 100 outputs
# input_shape => (samples, timestep, features )
X = np.array(X).reshape(100, 1, 1) # have to use numpy arrays!!! (lists can't be used)
Y = np.array(Y)

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())

model.fit(X, Y, epochs=20, batch_size=5, verbose=0)

test_input = np.array([0.5])
test_input = test_input.reshape((1, 1, 1))
test_output = model.predict(test_input, verbose=0)
print('input of 0.5, output is:', test_output)
```

```
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_29 (LSTM)               (None, 1, 50)             10400
_________________________________________________________________
lstm_30 (LSTM)               (None, 50)                20200
_________________________________________________________________
dense_25 (Dense)             (None, 1)                 51
=================================================================
Total params: 30,651
```

```
Trainable params: 30,651
Non-trainable params: 0
_________________________________________________________________
None
input of 0.5, output is: [[0.09945047]]
```

In the above model, you have **two LSTM layers**. Notice, the first LSTM layer now 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.

Once the model is trained, you again make predictions on the test data point i.e. 0.5.

One-to-One Sequence with Multiple Features

In the previous approach, each input sample had one time-step, where each time-step had one feature.

Now, you'll solve one-to-one sequence problems where input time-steps have multiple features.

> An example of where you might use multiple features in an animation system - you may have multiple joint signals all changing based on the current time-steop, however, they all influence the final output, i.e., single 'position' (that is certain combinations of joint signals create specific poses)

For this example, you'll create 2 inputs, but the output will depend on them (i.e., you'll sum the inputs together to get the output). This means the inputs are 'connected'(or coupled). This is typical in a character animation structure, where changes to certain joint angles in the hierarchy will influence the final overall pose (position of end-effectors).

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
import tensorflow as tf
import numpy as np
import math

# input a sin wave (input: t, output:signal aplitude)
# 1 input and 1 output (test training/signal)
X1 = []
X2 = []
Y  = []
for i in range(0,100):
  X1.append( i )    # 0-99
  X2.append( i*2 )  #
  Y.append( (i) + (i*2) ) # input is the sum of x1 + x2

X = np.column_stack((X1, X2))
# input_shape => (samples, timestep, features )
X = np.array(X).reshape(100, 1, 2)

# 100 inputs and 100 outputs
Y = np.array(Y)

model = Sequential()
model.add(LSTM(90, 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())

model.fit(X, Y, epochs=20, batch_size=5, verbose=0)

test_input = np.array([5,12])
test_input = test_input.reshape((1, 1, 2))
test_output = model.predict(test_input, verbose=0)
print('input of 5 and 12, output is:', test_output)
```

```
Model: "sequential_32"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_54 (LSTM)               (None, 90)                33480
_________________________________________________________________
dense_52 (Dense)             (None, 10)                910
_________________________________________________________________
dense_53 (Dense)             (None, 1)                 11
=================================================================
Total params: 34,401
Trainable params: 34,401
Non-trainable params: 0
_________________________________________________________________
None
input of 5 and 12, output is: [[12.801463]]
```

In the script above, you'll create three lists: X1, X2, and Y. Each list has 100 elements, which means that that the total sample size is 100. Finally, Y contains the output.

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

```
X = np.column_stack((X1, X2))
```

The X variable contains your final feature set. It contains two columns i.e. two features per input. As before, you need to convert the input into 3-dimensional shape. Your input has 100 samples, where each sample consist of 1 time-step and each time-step consists of 2 features. The following script reshapes the input.

```
X = np.array(X).reshape(100, 1, 2)
```

The example above uses a single LSTM layer model (start simple). Here your LSTM layer contains 90 neurons. You 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.

> Note, when you try the above example, the result will be far from the actual (desired) output, however, you'll fix this next by adding more layers.

Stacked LSTM

You'll now build upon the previous example, but create a more complex LSTM with multiple LSTM and

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
import tensorflow as tf
import numpy as np
import math

# input a sin wave (input: t, output:signal aplitude)
# 1 input and 1 output (test training/signal)
X1 = []
X2 = []
Y  = []
for i in range(0,100):
  X1.append( i )
  X2.append( i*2 )
  Y.append( (i) + (i*2) )

X = np.column_stack((X1, X2))
# input_shape => (samples, timestep, features )
X = np.array(X).reshape(100, 1, 2)

# 100 inputs and 100 outputs
Y = np.array(Y)

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())

model.fit(X, Y, epochs=20, batch_size=2, verbose=0)

test_input = np.array([5,12])
test_input = test_input.reshape((1, 1, 2))
test_output = model.predict(test_input, verbose=0)
print('input of 5 and 12, output is:', test_output)
```

```
Model: "sequential_33"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
lstm_55 (LSTM)              (None, 1, 200)            162400
_________________________________________________________________
lstm_56 (LSTM)              (None, 1, 100)            120400
_________________________________________________________________
lstm_57 (LSTM)              (None, 1, 50)             30200
_________________________________________________________________
lstm_58 (LSTM)              (None, 25)                7600
_________________________________________________________________
dense_54 (Dense)            (None, 20)                520
_________________________________________________________________
dense_55 (Dense)            (None, 10)                210
_________________________________________________________________
dense_56 (Dense)            (None, 1)                 11
```

```
=================================================================
Total params: 321,341
Trainable params: 321,341
Non-trainable params: 0
_________________________________________________________________
None
input of 5 and 12, output is: [[17.35761]]
```

To improve the accuracy, you also reduce the batch size, and since your model is more complex now you could also reduce the number of epochs. The above script trains the LSTM model and makes prediction on the test datapoint.

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 Animation Sequence Problems

Up to now, you've seen 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. However, 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 a 10 second animation sequence (many components).

You'll now 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. You'll start with many-to-one sequence problems having one feature, and then you'll see how to solve many-to-one problems where input time-steps have multiple features.

Many-to-One Sequence Problems with a Single Feature

First you'll create the dataset. Your dataset will consist of **20 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 your sample contains a sequence 2,4,6 the output will be 2 + 4 + 6 = 12

```
n = 20 * 3 # 60
X = np.array([x+1 for x in range(60)])
print(X)
# reshape it into number of samples, time-steps and features
# converts the list X into 3-dimensional shape with 20 samples, 3 time-steps, and 1 feature.
X = X.reshape(20,3,1)
print(X)

Y = []
for x in X:
  Y.append(x.sum())
```

```
Y = np.array(Y)
print(Y)
```

```
[ 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 46 47 48
 49 50 51 52 53 54 55 56 57 58 59 60]
[[[ 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]]
```

Simple LSTM

Start with an uncomplicated example

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
import tensorflow as tf
import numpy as np
import math

# input/output data 3 sets of 20 samples
X = np.array([x+1 for x in range(60)])
#print(X)
# reshape it into number of samples, time-steps and features
# converts the list X into 3-dimensional shape with 20 samples, 3 time-steps, and 1 feature.
X = X.reshape(20,3,1)
#print(X)

Y = []
for x in X:
  Y.append(x.sum())

Y = np.array(Y)
#print(Y)

model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(3, 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

model.fit(X, Y, epochs=20, batch_size=2, verbose=0)

test_input = np.array([3,4,5])
test_input = test_input.reshape((1, 3, 1))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
     output is: [[6.3132954]]
```

Once the model is trained, you can use it to make predictions on the test data points. For example, predict the output for the number sequence 3,4,5. The actual output should be 3 + 4 + 5 = 12.

Stacked LSTM

Taking the previous example further (i.e., adding extra layers). Easy to create a complex LSTM model with multiple layers and will help you get better results.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
import tensorflow as tf
import numpy as np
import math
```

```
# input/output data 3 sets of 20 samples
X = np.array([x+1 for x in range(60)])
#print(X)
# reshape it into number of samples, time-steps and features
# converts the list X into 3-dimensional shape with 20 samples, 3 time-steps, and 1 feature.
X = X.reshape(20,3,1)
#print(X)

Y = []
for x in X:
  Y.append(x.sum())

Y = np.array(Y)
#print(Y)

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')

model.fit(X, Y, epochs=20, batch_size=2, verbose=0)

test_input = np.array([3,4,5])
test_input = test_input.reshape((1, 3, 1))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
     output is: [[10.3440695]]
```

Bidirectional LSTM

The previous solution was good - but it could be better! **Bidirectional LSTM** 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.

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 tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.keras.layers import Bidirectional ## ** Notice
import tensorflow as tf
import numpy as np
import math

# input/output data 3 sets of 20 samples
X = np.array([x+1 for x in range(60)])
#print(X)
# reshape it into number of samples, time-steps and features
# converts the list X into 3-dimensional shape with 20 samples, 3 time-steps, and 1 feature.
```

```
X = X.reshape(20,3,1)
#print(X)

Y = []
for x in X:
  Y.append(x.sum())

Y = np.array(Y)
#print(Y)

model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(3, 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

#model.fit(X, Y, epochs=20, batch_size=2, verbose=0)
model.fit(X,Y, epochs=200, validation_split=0.2, verbose=0)

test_input = np.array([3,4,5])
test_input = test_input.reshape((1, 3, 1))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
     output is: [[8.757509]]
```

Testing the Bidirectional version, you'll find an increased improvement in prediction - that is, 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 you 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.

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

Start by creating two lists. link text

```
# Dataset
X1 = np.array([x+3 for x in range(0, 3*45, 3)]) # 45 elements in total.
print(X1)

X2 = np.array([x+5 for x in range(0, 5*45, 5)]) # 45 elements in total.
print(X2)

#  aggregated dataset can be created by joining the two lists
X = np.column_stack((X1, X2))
print(X)

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

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

```
print(X)
```

```
[  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]
[[  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]]
[[[  3   5]
  [  6  10]
  [  9  15]]

 [[ 12  20]
  [ 15  25]
  [ 18  30]]
```

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.

Simple LSTM

Lets solve this many-to-one sequence problem via simple LSTM.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.keras.layers import Bidirectional
import tensorflow as tf
import numpy as np
import math

X1 = np.array([x+3 for x in range(0, 3*45, 3)]) # 45 elements in total.
#print(X1)

X2 = np.array([x+5 for x in range(0, 5*45, 5)]) # 45 elements in total.
#print(X2)

#  aggregated dataset can be created by joining the two lists
X = np.column_stack((X1, X2))
#print(X)

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

X = np.array(X).reshape(15, 3, 2)
#print(X)

Y = []
for x in X:
  Y.append(x.sum())

Y = np.array(Y)
#print(Y)

model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(3, 2)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

model.fit(X, Y, epochs=20, validation_split=0.2, verbose=0)


test_input = np.array([ [3,4],
                        [6,7],
                        [9,10] ] );
test_input = test_input.reshape((1, 3, 2))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
    output is: [[4.145947]]
```

Stacked LSTM

Takes the previous example and train a stacked LSTM and makes predictions on the test point.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.keras.layers import Bidirectional
import tensorflow as tf
import numpy as np
import math

X1 = np.array([x+3 for x in range(0, 3*45, 3)]) # 45 elements in total.
#print(X1)

X2 = np.array([x+5 for x in range(0, 5*45, 5)]) # 45 elements in total.
#print(X2)

#  aggregated dataset can be created by joining the two lists
X = np.column_stack((X1, X2))
#print(X)

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

X = np.array(X).reshape(15, 3, 2)
#print(X)

Y = []
for x in X:
  Y.append(x.sum())

Y = np.array(Y)
#print(Y)

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')

model.fit(X, Y, epochs=20, validation_split=0.2, verbose=0)


test_input = np.array([ [3,4],
                        [6,7],
                        [9,10] ] );
test_input = test_input.reshape((1, 3, 2))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
output is: [[1.0489745]]
```

Bidirectional LSTM

Simple bidirectional LSTM along with code that is used to make predictions on the test point.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.keras.layers import Bidirectional
import tensorflow as tf
import numpy as np
import math

X1 = np.array([x+3 for x in range(0, 3*45, 3)]) # 45 elements in total.
#print(X1)

X2 = np.array([x+5 for x in range(0, 5*45, 5)]) # 45 elements in total.
#print(X2)

#  aggregated dataset can be created by joining the two lists
X = np.column_stack((X1, X2))
#print(X)

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

X = np.array(X).reshape(15, 3, 2)
#print(X)

Y = []
for x in X:
  Y.append(x.sum())

Y = np.array(Y)
#print(Y)

model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(3, 2)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

model.fit(X, Y, epochs=20, validation_split=0.2, verbose=0)


test_input = np.array([ [3,4],
                        [6,7],
                        [9,10] ] );
test_input = test_input.reshape((1, 3, 2))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
output is: [[3.7327428]]
```

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. You can solve such problems by simply changing the number of neurons in the output dense layer to the number of features values that you want in the output. However, first you need to update our output vector Y

```
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)
```

```
     [[ 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]]
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow.keras.layers import Bidirectional
import tensorflow as tf
import numpy as np
import math

X1 = np.array([x+3 for x in range(0, 3*45, 3)]) # 45 elements in total.
#print(X1)

X2 = np.array([x+5 for x in range(0, 5*45, 5)]) # 45 elements in total.
#print(X2)

#  aggregated dataset can be created by joining the two lists
X = np.column_stack((X1, X2))
#print(X)

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

X = np.array(X).reshape(15, 3, 2)
#print(X)

Y = []
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)

# simple version:
#model = Sequential()
#model.add(LSTM(50, activation='relu', input_shape=(3, 2)))
#model.add(Dense(2))
#model.compile(optimizer='adam', loss='mse')

# bidirectional version
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(3, 2)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')

model.fit(X, Y, epochs=20, validation_split=0.2, verbose=0)


test_input = np.array([ [3,4],
                        [6,7],
                        [9,10] ] );
test_input = test_input.reshape((1, 3, 2))
test_output = model.predict(test_input, verbose=0)
print('output is:', test_output)
```

```
     output is: [[2.1390767]]
```

Conclusion

Simple neural networks are not suitable for solving **sequence problems** since in sequence problems, in addition to current input, you 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.

You saw how different variants of the LSTM algorithm can be used to solve one-to-one and many-to-one sequence problems. From simple one-to-one through to one-to-man and finally many-to-many sequence problems.