CS 395
Homework 6 – Multi-Step LSTMs
35 Points Total
Due in Canvas by 11:59 PM on Sunday, March 3, 2019

Today we continue looking at Long Short-Term Memory networks, or LSTMs. As a reminder, a really good article about LSTMs can be found here: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

In our 2 last labs (HW 4 and HW 5), we looked at univariate and multivariate LSTMs. In this lab, you will discover how to develop a suite of LSTM models for a multi-step problems.

Multi-step LSTM Models

A time series forecasting problem that requires a prediction of multiple time steps into the future can be referred to as *multi-step time series forecasting*.

Specifically, these are problems where the forecast horizon or interval is more than one timestep.

There are two main types of LSTM models that can be used for multi-step forecasting; they are:

- Vector Output Model
- Encoder-Decoder Model

Before we look at these models, let's first look at the preparation of data for multi-step forecasting.

Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components.

Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps.

For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

We could use the last three timesteps as input and forecast the next two timesteps. The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Output:

```
[40, 50]
```

The split_sequence() function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
      X, y = list(), list()
      for i in range(len(sequence)):
            # find the end of this pattern
            end_ix = i + n_steps_in
            out_end_ix = end_ix + n_steps_out
            # check if we are beyond the sequence
            if out_end_ix > len(sequence):
                  break
            # gather input and output parts of the pattern
            seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
            X.append(seq_x)
            y.append(seq_y)
      return array(X), array(y)
```

We can demonstrate this function on the small contrived dataset.

The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequence)):
  # find the end of this pattern
  end_ix = i + n_steps_in
  out_end_ix = end_ix + n_steps_out
  # check if we are beyond the sequence
  if out_end_ix > len(sequence):
   break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
  X.append(seq_x)
  y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

1. Run the above code and show your results (1 point)

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

Now that we know how to prepare data for multi-step forecasting, let's look at some LSTM models that can learn this mapping.

Vector Output Model

Like other types of neural network models, the LSTM can output a vector directly that can be interpreted as a multi-step forecast.

This approach was seen in the previous section were one timestep of each output time series was forecasted as a vector.

As with the LSTMs for univariate data in a prior section, the prepared samples must first be reshaped. The LSTM expects data to have a three-dimensional structure of `[samples, timesteps, features]`, and in this case, we only have one feature so the reshape is straightforward.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

With the number of input and output steps specified in the n_steps_in and n_steps_out variables, we can define a multi-step time-series forecasting model.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input:

```
[70, 80, 90]
```

And get a prediction like:

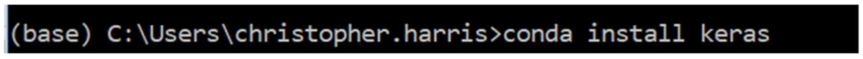
```
[100, 110]
```

As expected by the model, the shape of the single sample of input data when making the prediction must be [1, 3, 1] for the 1 sample, 3 timesteps of the input, and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
```

Tying all of this together, the Stacked LSTM for multi-step forecasting with a univariate time series is listed below.

Make sure you have keras installed if you hadn't done it before in the previous 2 labs:



```
(base) C:\Users\christopher.harris>conda install keras
```

2. Cut and paste the following code and run at least 3 times. Make sure you provide some comment in the output that states that it is Vector Output LSTM (you may consider changing the last print statement). Show your input (once) and all outputs (1 point)
3. Change the activation function from 'relu' to 2 other activation functions we have learned about. Make sure the comments indicate that you are using the Vector Output LSTM and the activation function you used you may consider changing the last print statement). Run each function at least 3 times. Show code and all outputs (4 points: 2 points each)

```
# univariate multi-step vector-output stacked lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequence)):
  # find the end of this pattern
  end_ix = i + n_steps_in
  out_end_ix = end_ix + n_steps_out
  # check if we are beyond the sequence
  if out_end_ix > len(sequence):
   break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
  X.append(seq_x)
  y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True,
input_shape=(n_steps_in, n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=50, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
```

```
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Running the example prepares the data, fits the model, and makes a prediction. Your results will vary given the stochastic nature of the algorithm; that is why we try running the example a few times. We can see that the model predicts the next value in the sequence.

Encoder-Decoder Model

A model specifically developed for forecasting variable length output sequences is called the Encoder-Decoder LSTM.

The model was designed for prediction problems where there are both input and output sequences, so-called sequence-to-sequence, or seq2seq problems, such as translating text from one language to another. This model can be used for multi-step time series forecasting.

As its name suggests, the model is comprised of two sub-models: the encoder and the decoder.

The encoder is a model responsible for reading and interpreting the input sequence. The output of the encoder is a fixed length vector that represents the model's interpretation of the sequence. The encoder is traditionally a Vanilla LSTM model, although other encoder models can be used such as Stacked, Bidirectional, and CNN models.

```
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
```

The decoder uses the output of the encoder as an input. First, the fixed-length output of the encoder is repeated, once for each required time step in the output sequence.

```
model.add(RepeatVector(n_steps_out))
```

This sequence is then provided to an LSTM decoder model. The model must output a value for each value in the output time step, which can be interpreted by a single output model.

```
model.add(LSTM(100, activation='relu', return_sequences=True))
```

We can use the same output layer or layers to make each one-step prediction in the output sequence. This can be achieved by wrapping the output part of the model in a TimeDistributed wrapper.

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

The full definition for an Encoder-Decoder model for multi-step time series forecasting is listed below.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
```

As with other LSTM models, the input data must be reshaped into the expected three-dimensional shape of `[samples, timesteps, features]`.

```
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

In the case of the Encoder-Decoder model, the output, or y part, of the training dataset must also have this shape. This is because the model will predict a given number of time steps with a given number of features for each input sample.

```
y = y.reshape((y.shape[0], y.shape[1], n_features))
```

The complete example of an Encoder-Decoder LSTM for multi-step time series forecasting is listed below.

4. Cut and paste the following code and run at least 3 times. Make sure you provide some comment in the output that states that it is Encoder-Decoder Model Vanilla LSTM (you may consider changing the last print statement). Show your input (once) and all outputs (1 point)
5. Change the activation function from 'relu' to 2 other activation functions we have learned about. Make sure the comments indicate that you are using the Encoder-Decoder Model Vanilla LSTM and the activation function you used you may consider changing the last print statement). Run each function at least 3 times. Show code and all outputs (4 points: 2 points each)
6. Change the activation function back to 'relu'. Change this model to two of the other univariate LSTM models (i.e., Stacked, Bidirectional, CNN, or Conv LSTM) we examined in HW 4. Encoder-Decoder Model <x> LSTM, where <x> is the other univariate function model. Run each model at least 3 times. Show code and all outputs (8 points total, 4 points each)

```
# univariate multi-step encoder-decoder lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequence)):
  # find the end of this pattern
  end_ix = i + n_steps_in
  out_end_ix = end_ix + n_steps_out
  # check if we are beyond the sequence
  if out_end_ix > len(sequence):
   break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
  X.append(seq_x)
  y.append(seq_y)
 return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
```

```
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps,
features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
y = y.reshape((y.shape[0], y.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in,
n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=100, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Multivariate Multi-Step LSTM Models

It is possible to mix and match the different types of LSTM models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging.

In this section, we look at short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

- Multiple Input Multi-Step Output.
- Multiple Parallel Input and Multi-Step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

Multiple Input Multi-Step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series.

For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

We may use three prior time steps of each of the two input time series to predict two timesteps of the output time series.

Input:

```
10, 15
20, 25
30, 35
```

Output:

```
65
85
```

The split_sequences() function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
	X, y = list(), list()
	for i in range(len(sequences)):
		# find the end of this pattern
		end_ix = i + n_steps_in
		out_end_ix = end_ix + n_steps_out-1
		# check if we are beyond the dataset
		if out_end_ix > len(sequences):
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-
		1:out_end_ix, -1]
		X.append(seq_x)
		y.append(seq_y)
	return array(X), array(y)
```

We can demonstrate this on our contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
  # find the end of this pattern
  end_ix = i + n_steps_in
  out_end_ix = end_ix + n_steps_out-1
  # check if we are beyond the dataset
  if out_end_ix > len(sequences):
   break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-
1:out_end_ix, -1]
  X.append(seq_x)
  y.append(seq_y)
 return array(X), array(y)
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
 print(X[i], y[i])
```

7. Run the above code and show your results (1 point)

Running the example first prints the shape of the prepared training data.

We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three timesteps, and two variables for the 2 input time series.

The output portion of the samples is two-dimensional for the six samples and the two timesteps for each sample to be predicted. We can now develop an LSTM model for multi-step predictions.

A vector output or an encoder-decoder model could be used. In this case, we will demonstrate a vector output with a Stacked LSTM.

The complete example is listed below.

8. Cut and paste the following code and run at least 3 times. Make sure you provide some comment in the output that states that it is Vector Output Stacked LSTM (you may consider changing the last print statement). Show your input (once) and all outputs (1 point)
9. Change the activation function from 'relu' to 2 other activation functions we have learned about. Make sure the comments indicate that you are using the Vector Output Stacked LSTM and the activation function you used you may consider changing the last print statement). Run each function at least 3 times. Show code and all outputs (4 points: 2 points each)

```
# multivariate multi-step stacked lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
 X, y = list(), list()
 for i in range(len(sequences)):
  # find the end of this pattern
  end_ix = i + n_steps_in
  out_end_ix = end_ix + n_steps_out-1
  # check if we are beyond the dataset
  if out_end_ix > len(sequences):
   break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
  X.append(seq_x)
  y.append(seq_y)
 return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True,
input_shape=(n_steps_in, n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Running the example fits the model and predicts the next two timesteps of the output sequence beyond the dataset.

We would expect the next two steps to be: `[185, 205]`

It is a challenging framing of the problem with very little data, and the arbitrarily configured version of the model gets close.

Multiple Parallel Input and Multi-Step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series.

For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

We may use the last three timesteps from each of the three time series as input to the model and predict the next time steps of each of the three time series as output.

The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Output:

```
40, 45, 85
50, 55, 105
```

The split_sequences() function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
	X, y = list(), list()
	for i in range(len(sequences)):
		# find the end of this pattern
		end_ix = i + n_steps_in
		out_end_ix = end_ix + n_steps_out
		# check if we are beyond the dataset
		if out_end_ix > len(sequences):
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequences[i:end_ix, :],
		sequences[end_ix:out_end_ix, :]
		X.append(seq_x)
		y.append(seq_y)
```

```
        return array(X), array(y)
```

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
      X, y = list(), list()
      for i in range(len(sequences)):
            # find the end of this pattern
            end_ix = i + n_steps_in
            out_end_ix = end_ix + n_steps_out
            # check if we are beyond the dataset
            if out_end_ix > len(sequences):
                  break
            # gather input and output parts of the pattern
            seq_x, seq_y = sequences[i:end_ix, :],
            sequences[end_ix:out_end_ix, :]
            X.append(seq_x)
            y.append(seq_y)
      return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
      print(X[i], y[i])
```

10. Run the above code and show your results (1 point)

Running the example first prints the shape of the prepared training dataset.

We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively.

The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

We can use either the Vector Output or Encoder-Decoder LSTM to model this problem. In this case, we will use the Encoder-Decoder model.

The complete example is listed below.

11. Cut and paste the following code and run at least 3 times. Make sure you provide some comment in the output that states that it is Multi-Step Encoder-Decoder LSTM (you may consider changing the last print statement). Show your input (once) and all outputs (1 point)
12. Change the activation function from 'relu' to 2 other activation functions we have learned about. Make sure the comments indicate that you are using the Multi-Step Encoder-Decoder LSTM and the activation function you used you may consider changing the last print statement). Run each function at least 3 times. Show code and all outputs (4 points: 2 points each)

```
# multivariate multi-step encoder-decoder lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
      X, y = list(), list()
      for i in range(len(sequences)):
            # find the end of this pattern
            end_ix = i + n_steps_in
            out_end_ix = end_ix + n_steps_out
            # check if we are beyond the dataset
            if out_end_ix > len(sequences):
                  break
            # gather input and output parts of the pattern
            seq_x, seq_y = sequences[i:end_ix, :],
sequences[end_ix:out_end_ix, :]
            X.append(seq_x)
            y.append(seq_y)
      return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
```

```
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(200, activation='relu', input_shape=(n_steps_in,
n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(200, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(n_features)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=300, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset.

We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

13. Provide a single table that illustrates the different LSTM models, the activation functions used, and the average of the 3 results (predictions) you received to 3 decimal places. What are your general findings? Follow the template below. (4 points)

LSTM Model	**Activation Function Used**	**Avg. of 3 Results**	**General Findings**