CS 395
Homework 4 – Univariate LSTMs
30 Points Total
Due in Canvas by 11:59 PM on Sunday, February 17, 2019

Today we will be digging a little deeper into neural networking models and, specifically, Long Short-Term Memory networks, or LSTMs.

A really good article about LSTMs can be found here: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Specifically, LSTMs are improvements on RNNs (Recurrent Neural Networks). They include a "forget" gate which is quite helpful. LSTMs can be applied to time series forecasting.

There are many types of LSTM models that can be used for each specific type of time series forecasting problem.

In this lab assignment, you will discover how to develop a suite of LSTM models for a range of standard time series forecasting problems.

We will provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem.

After completing this lab, you will learn how to develop LSTM models for univariate time series forecasting.

Univariate LSTM Models

LSTMs can be used to model univariate time series forecasting problems.

These are problems comprised of a single series of observations and a model is required to learn from the series of past observations to predict the next value in the sequence.

We will demonstrate several variations of the LSTM model for univariate time series forecasting.

This section is divided into six parts; they are:

- Data Preparation
- Vanilla LSTM
- Stacked LSTM
- Bidirectional LSTM
- CNN LSTM
- ConvLSTM

Each of these models are demonstrated for one-step univariate time series forecasting but can easily be adapted and used as the input part of a model for other types of time series forecasting problems.

Data Preparation

Before a univariate series can be modeled, it must be prepared.

The LSTM model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn.

Consider a given univariate sequence:

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

We can divide the sequence into multiple input/output patterns called samples, where 3 time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,			y
10, 20, 30		40
20, 30, 40		50
30, 40, 50		60
...
```

The split_sequence() function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

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

We can demonstrate this function on our small contrived dataset above.

The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
	X, y = list(), list()
	for i in range(len(sequence)):
		# find the end of this pattern
		end_ix = i + n_steps
		# check if we are beyond the sequence
		if end_ix > len(sequence)-1:
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequence[i:end_ix], sequence[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 = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
      print(X[i], y[i])
```

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

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

Now that we know how to prepare a univariate series for modeling, let's look at developing LSTM models that can learn the mapping of inputs to outputs, starting with a Vanilla LSTM.

Vanilla LSTM

A Vanilla LSTM is an LSTM model that has a single hidden layer of LSTM units, and an output layer used to make a prediction. We can define a Vanilla LSTM for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps,
n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

The key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features.

We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the split_sequence() function. The shape of the input for each sample is specified in the input_shape argument on the definition of first hidden layer.

We almost always have multiple samples; therefore, the model will expect the input component of training data to have the dimensions or shape:

```
[samples, timesteps, features]
```

Our split_sequence() function in the previous section outputs the X with the shape [samples, timesteps], so we easily reshape it to have an additional dimension for the one feature.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
```

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

In this case, we define a model with 50 LSTM units in the hidden layer and an output layer that predicts a single numerical value.

The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or 'mse' loss function.

Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=200, verbose=0)
```

After the model is fit, we can use it to make a prediction.

We can predict the next value in the sequence by providing the input:

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

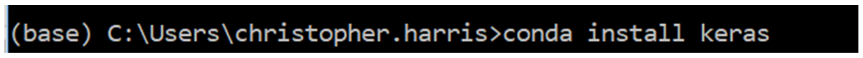
And get a prediction like:

```
[100]
```

The model expects the input shape to be three-dimensional with `[samples, timesteps, features]`, therefore, we must reshape the single input sample before making the prediction.

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

We can tie all of this together and demonstrate how to develop a Vanilla LSTM for univariate time series forecasting and make a single prediction. Before this, ensure you install keras (see below)



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

1. 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 Vanilla LSTM (you may consider changing the last print statement). Show all outputs (1 point)
2. Change the activation function from 'relu' to 2 other ones we have learned about. Make sure the comments indicate that you are using the Vanilla LSTM and the activation function you used you may consider changing the last print statement). Run each at least 3 times. Show all outputs (2 points)

```
# univariate 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):
   X, y = list(), list()
   for i in range(len(sequence)):
         # find the end of this pattern
         end_ix = i + n_steps
         # check if we are beyond the sequence
         if end_ix > len(sequence)-1:
               break
         # gather input and output parts of the pattern
         seq_x, seq_y = sequence[i:end_ix], sequence[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 = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# 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(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, 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.

Stacked LSTM

Multiple hidden LSTM layers can be stacked one on top of another in what is referred to as a Stacked LSTM model. An LSTM layer requires a three-dimensional input and LSTMs by default will produce a two-dimensional output as an interpretation from the end of the sequence.

We can address this by having the LSTM output a value for each time step in the input data by setting the *return_sequences=True* argument on the layer. This allows us to have 3D output from hidden LSTM layer as input to the next. We can therefore define a Stacked LSTM as follows.

3. 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 Stacked LSTM (you may consider changing the last print statement). Show all outputs (1 point)
4. Change the activation function from ‘relu’ to 2 other ones we have learned about. Make sure the comments indicate that you are using the Stacked LSTM and the activation function you used you may consider changing the last print statement). Run each at least 3 times. Show all outputs (2 points)

```
# univariate 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
def split_sequence(sequence, n_steps):
   X, y = list(), list()
   for i in range(len(sequence)):
         # find the end of this pattern
         end_ix = i + n_steps
         # check if we are beyond the sequence
         if end_ix > len(sequence)-1:
               break
         # gather input and output parts of the pattern
         seq_x, seq_y = sequence[i:end_ix], sequence[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 = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# 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(50, activation='relu', return_sequences=True,
input_shape=(n_steps, n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

5. What are the primary differences between the Stacked LSTM and the Vanilla LSTM? Describe and explain what is occurring differently. When would you use the Stacked LSTM instead of the Vanilla LSTM? When would you use the Vanilla LSTM instead of the Stacked LSTM? (3 points).

Bidirectional LSTM
On some sequence prediction problems, it can be beneficial to allow the LSTM model to learn the input sequence both forward and backwards and concatenate both interpretations. This is called a Bidirectional LSTM.

We can implement a Bidirectional LSTM for univariate time series forecasting by wrapping the first hidden layer in a wrapper layer called **Bidirectional**. The complete example of the Bidirectional LSTM for univariate time series forecasting is listed below.

6. 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 Bidirectional LSTM (you may consider changing the last print statement) Show all outputs (1 point)
7. Change the activation function from 'relu' to 2 other ones we have learned about. Make sure the comments indicate that you are using the Bidirectional LSTM and the activation function you used you may consider changing the last print statement). Run each at least 3 times. Show all outputs (2 points)

```
# univariate bidirectional 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 Bidirectional

# split a univariate sequence
def split_sequence(sequence, n_steps):
   X, y = list(), list()
   for i in range(len(sequence)):
         # find the end of this pattern
         end_ix = i + n_steps
         # check if we are beyond the sequence
         if end_ix > len(sequence)-1:
               break
         # gather input and output parts of the pattern
         seq_x, seq_y = sequence[i:end_ix], sequence[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 = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# 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(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps,
n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

8. What are the primary differences between the Bidirectional LSTM and the Vanilla LSTM? Describe and explain what is occurring differently. When would you use the Bidirectional LSTM instead of the Vanilla LSTM? When would you use the Vanilla LSTM instead of the Bidirectional LSTM? (3 points).

CNN LSTM

A convolutional neural network, or CNN for short, is a type of neural network developed for working with two-dimensional image data. We saw this last week with the Fashion MNIST dataset.

The CNN can be very effective at automatically extracting and learning features from one-dimensional sequence data such as univariate time series data.

A CNN model can be used in a hybrid model with an LSTM backend where the CNN is used to interpret subsequences of input that together are provided as a sequence to an LSTM model to interpret. This hybrid model is called a CNN-LSTM.

The first step is to split the input sequences into subsequences that can be processed by the CNN model. For example, we can first split our univariate time series data into input/output samples with four steps as input and one as output. Each sample can then be split into two sub-samples, each with 2 time steps. The CNN can interpret each subsequence of 2 time steps and provide a time series of interpretations of the subsequences to the LSTM model to process as input.

We can parameterize this and define the number of subsequences as n_seq and the number of time steps per subsequence as n_steps. The input data can then be reshaped to have the required structure:

```
[samples, subsequences, timesteps, features]
```

We want to reuse the same CNN model when reading in each sub-sequence of data separately.

This can be achieved by wrapping the entire CNN model in a time-distributed wrapper that will apply the entire model once per input, in this case, once per input subsequence.

The CNN model first has a convolutional layer for reading across the subsequence that requires several filters and a kernel size to be specified. The number of filters is the number of reads or interpretations of the input sequence. The kernel size is the number of time steps included of each 'read' operation of the input sequence.

The convolution layer is followed by a max pooling layer that distills the filter maps down to 1/4 of their size that includes the most salient features. These structures are then flattened down to a single one-dimensional vector to be used as a single input time step to the LSTM layer.

```
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1,
activation='relu'), input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
```

Next, we can define the LSTM part of the model that interprets the CNN model's read of the input sequence and makes a prediction.

```
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
```

We can tie all of this together; the complete example of a CNN LSTM model for univariate time series forecasting is listed below.

9. 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 CNN LSTM (you may consider changing the last print statement) Show all outputs (1 point)
10. Change the activation function from 'relu' to 2 other ones we have learned about. Make sure the comments indicate that you are using the CNN LSTM and the activation function you used you may consider changing the last print statement). Run each at least 3 times. Show all outputs (2 points)

```
# univariate cnn 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 Flatten
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
      X, y = list(), list()
      for i in range(len(sequence)):
            # find the end of this pattern
            end_ix = i + n_steps
            # check if we are beyond the sequence
            if end_ix > len(sequence)-1:
                  break
            # gather input and output parts of the pattern
            seq_x, seq_y = sequence[i:end_ix], sequence[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 = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences,
timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
# define model
model = Sequential()
```

```
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1,
activation='relu'), input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
x_input = x_input.reshape((1, n_seq, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

11. What are the primary differences between the CNN LSTM and the Vanilla LSTM? Describe and explain what is occurring differently. When would you use the CNN LSTM instead of the Vanilla LSTM? When would you use the Vanilla LSTM instead of the CNN LSTM? (3 points).

ConvLSTM

A type of LSTM related to the CNN-LSTM is the ConvLSTM, where the convolutional reading of input is built directly into each LSTM unit. The ConvLSTM was developed for reading two-dimensional spatial-temporal data, but can be adapted for use with univariate time series forecasting.

The layer expects input as a sequence of two-dimensional images, therefore the shape of input data must be:

```
[samples, timesteps, rows, columns, features]
```

For our purposes, we can split each sample into subsequences where timesteps will become the number of subsequences, or n_seq, and columns will be the number of time steps for each subsequence, or n_steps. The number of rows is fixed at 1 as we are working with one-dimensional data.

We can now reshape the prepared samples into the required structure.

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows,
columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
```

We can define the ConvLSTM as a single layer in terms of the number of filters and a two-dimensional kernel size in terms of (rows, columns). As we are working with a one-dimensional series, the number of rows is always fixed to 1 in the kernel. The output of the model must then be flattened before it can be interpreted, and a prediction made.

```
model.add(ConvLSTM2D(filters=64, kernel_size=(1,2), activation='relu',
input_shape=(n_seq, 1, n_steps, n_features)))
model.add(Flatten())
```

The full code is below.

12. 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 Conv-LSTM (you may consider changing the last print statement). Show all outputs (1 point)
13. Change the activation function from 'relu' to 2 other ones we have learned about. Make sure the comments indicate that you are using the Conv-LSTM and the activation function you used you may consider changing the last print statement). Run each at least 3 times. Show all outputs (2 points)

```
# univariate convlstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import ConvLSTM2D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
	X, y = list(), list()
	for i in range(len(sequence)):
		# find the end of this pattern
		end_ix = i + n_steps
		# check if we are beyond the sequence
		if end_ix > len(sequence)-1:
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequence[i:end_ix], sequence[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 = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows,
columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
# define model
model = Sequential()
model.add(ConvLSTM2D(filters=64, kernel_size=(1,2), activation='relu',
input_shape=(n_seq, 1, n_steps, n_features)))
model.add(Flatten())
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
x_input = x_input.reshape((1, n_seq, 1, n_steps, n_features))
```

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

14. What are the primary differences between the Conv-LSTM and the CNN LSTM? Describe and explain what is occurring differently. When would you use the Conv-LSTM instead of the CNN LSTM? When would you use the CNN LSTM instead of the Conv-LSTM? (3 points).
15. 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. (3 points)

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