Chapter 10

How to Develop Bidirectional LSTMs

10.0.1 Lesson Goal

The goal of this lesson is to learn how to develop Bidirectional LSTM models. After completing this lesson, you will know:

- The Bidirectional LSTM architecture and how to implement it in Keras.
- The cumulative sum prediction problem.
- How to develop a Bidirectional LSTM for the cumulative sum prediction problem.

10.0.2 Lesson Overview

This lesson is divided into 7 parts; they are:

1. The Bidirectional LSTM.
2. Cumulative Sum Prediction Problem.
3. Define and Compile the Model.
4. Fit the Model.
5. Evaluate the Model.
6. Make Predictions With the Model.
7. Complete Example.

Let's get started.

10.1 The Bidirectional LSTM

10.1.1 Architecture

We have seen the benefit of reversing the order of input sequences for LSTMs discussed in the introduction of Encoder-Decoder LSTMs.

> We were surprised by the extent of the improvement obtained by reversing the words in the source sentences.

— *Sequence to Sequence Learning with Neural Networks*, 2014.

Bidirectional LSTMs focus on the problem of getting the most out of the input sequence by stepping through input time steps in both the forward and backward directions. In practice, this architecture involves duplicating the first recurrent layer in the network so that there are now two layers side-by-side, then providing the input sequence as-is as input to the first layer and providing a reversed copy of the input sequence to the second. This approach was developed some time ago as a general approach for improving the performance of Recurrent Neural Networks (RNNs).

> To overcome the limitations of a regular RNN ... we propose a bidirectional recurrent neural network (BRNN) that can be trained using all available input information in the past and future of a specific time frame. ... The idea is to split the state neurons of a regular RNN in a part that is responsible for the positive time direction (forward states) and a part for the negative time direction (backward states)

— *Bidirectional Recurrent Neural Networks*, 1997.

This approach has been used to great effect with LSTM Recurrent Neural Networks. Providing the entire sequence both forwards and backwards is based on the assumption that the whole sequence is available. This is generally a requirement in practice when using vectorized inputs. Nevertheless, it may raise a philosophical concern where ideally time steps are provided in order and just-in-time. The use of providing an input sequence bi-directionally was justified in the domain of speech recognition because there is evidence that in humans, the context of the whole utterance is used to interpret what is being said rather than a linear interpretation.

> ... relying on knowledge of the future seems at first sight to violate causality. How can we base our understanding of what we've heard on something that hasn't been said yet? However, human listeners do exactly that. Sounds, words, and even whole sentences that at first mean nothing are found to make sense in the light of future context. What we must remember is the distinction between tasks that are truly online - requiring an output after every input - and those where outputs are only needed at the end of some input segment.

— *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures*, 2005.

Although Bidirectional LSTMs were developed for speech recognition, the use of Bidirectional input sequences is now a staple of sequence prediction with LSTMs as an approach for lifting model performance.

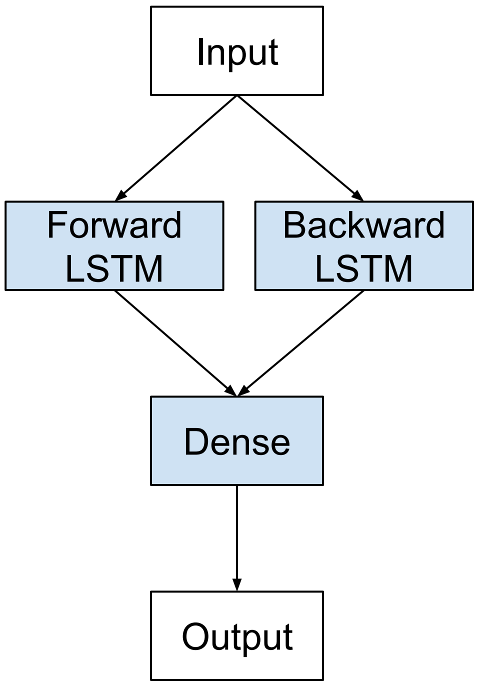

Figure 10.1: Bidirectional LSTM Architecture.

10.1.2 Implementation

The `LSTM` layer in Keras allow you to specify the directionality of the input sequence. This can be done by setting the `go_backwards` argument to `True` (defaults to `False`).

```
model = Sequential()
model.add(LSTM(..., input_shape=(...), go_backwards=True))
...
```

Listing 10.1: Example of a Vanilla LSTM model with backward input sequences.

Bidirectional LSTMs are a small step on top of this capability. Specifically, Bidirectional LSTMs are supported in Keras via the `Bidirectional` layer wrapper that essentially merges the output from two parallel LSTMs, one with input processed forward and one with output processed backwards. This wrapper takes a recurrent layer (e.g. the first hidden `LSTM` layer) as an argument.

```
model = Sequential()
model.add(Bidirectional(LSTM(...), input_shape=(...)))
...
```

Listing 10.2: Example of a `Bidirectional` wrapped `LSTM` layer.

The `Bidirectional` wrapper layer also allows you to specify the merge mode; that is how the forward and backward outputs should be combined before being passed on to the next layer. The options are:

- `'sum'`: The outputs are added together.
- `'mul'`: The outputs are multiplied together.

- `'concat'`: The outputs are concatenated together (the default), providing double the number of outputs to the next layer.
- `'ave'`: The average of the outputs is taken.

The default mode is to concatenate, and this is the method often used in studies of bidirectional LSTMs. In general, it might be a good idea to test each of the merge modes on your problem to see if you can improve upon the concatenate default option.

10.2 Cumulative Sum Prediction Problem

We will define a simple sequence classification problem to explore bidirectional LSTMs called the cumulative sum prediction problem. This section is divided into the following parts:

1. Cumulative Sum.
2. Sequence Generation.
3. Generate Multiple Sequences.

10.2.1 Cumulative Sum

The problem is defined as a sequence of random values between 0 and 1. This sequence is taken as input for the problem with each number provided once per time step. A binary label (0 or 1) is associated with each input. The output values are all 0. Once the cumulative sum of the input values in the sequence exceeds a threshold, then the output value flips from 0 to 1.

A threshold of one quarter ($\frac{1}{4}$) the sequence length is used. For example, below is a sequence of 10 input time steps (`X`):

```
0.63144003 0.29414551 0.91587952 0.95189228 0.32195638 0.60742236 0.83895793 0.18023048
    0.84762691 0.29165514
```

Listing 10.3: Example input sequence of random real values.

The corresponding classification output (`y`) would be:

```
0 0 0 1 1 1 1 1 1 1
```

Listing 10.4: Example output sequence of cumulative sum values.

We will frame the problem to make the best use of the Bidirectional LSTM architecture. The output sequence will be produced after the entire input sequence has been fed into the model. Technically, this means this is a sequence-to-sequence prediction problem that requires a many-to-many prediction model. It is also the case that the input and output sequences have the same number of time steps (length).

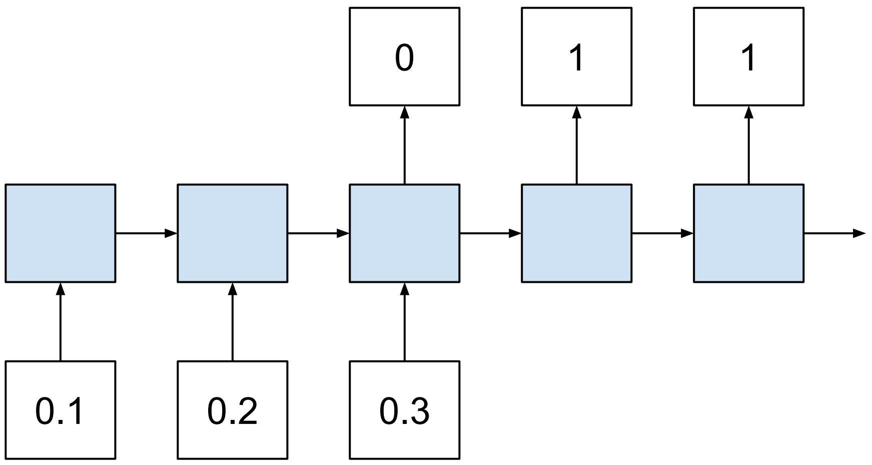

Figure 10.2: Cumulative sum prediction problem framed with a many-to-many prediction model.

10.2.2 Sequence Generation

We can implement this in Python. The first step is to generate a sequence of random values. We can use the `random()` function from the random module.

```python
# create a sequence of random numbers in [0,1]
X = array([random() for _ in range(10)])
```

Listing 10.5: Example creating an input sequence of random real values.

We can define the threshold as one-quarter the length of the input sequence.

```python
# calculate cut-off value to change class values
limit = 10/4.0
```

Listing 10.6: Example of calculating the cumulative sum threshold.

The cumulative sum of the input sequence can be calculated using the `cumsum()` NumPy function. This function returns a sequence of cumulative sum values, e.g.:

```
pos1, pos1+pos2, pos1+pos2+pos3, ...
```

Listing 10.7: Example of calculating a cumulative sum output sequence.

We can then calculate the output sequence as to whether each cumulative sum value exceeded the threshold.

```python
# determine the class outcome for each item in cumulative sequence
y = array([0 if x < limit else 1 for x in cumsum(X)])
```

Listing 10.8: Example of implementing the calculating of the cumulative sum threshold.

The function below, named `get_sequence()`, draws all of this together, taking as input the length of the sequence, and returns the `X` and `y` components of a new problem case.

```python
# create a sequence classification instance
def get_sequence(n_timesteps):
  # create a sequence of random numbers in [0,1]
  X = array([random() for _ in range(n_timesteps)])
  # calculate cut-off value to change class values
  limit = n_timesteps/4.0
  # determine the class outcome for each item in cumulative sequence
  y = array([0 if x < limit else 1 for x in cumsum(X)])
  return X, y
```

Listing 10.9: Function to create a random input and output sequence.

We can test this function with a new 10-step sequence as follows:

```
from random import random
from numpy import array
from numpy import cumsum

# create a cumulative sum sequence
def get_sequence(n_timesteps):
  # create a sequence of random numbers in [0,1]
  X = array([random() for _ in range(n_timesteps)])
  # calculate cut-off value to change class values
  limit = n_timesteps/4.0
  # determine the class outcome for each item in cumulative sequence
  y = array([0 if x < limit else 1 for x in cumsum(X)])
  return X, y

X, y = get_sequence(10)
print(X)
print(y)
```

Listing 10.10: Example of generating a random input and output sequence.

Running the example first prints the generated input sequence followed by the matching output sequence.

```
[ 0.22228819 0.26882207 0.069623 0.91477783 0.02095862 0.71322527
0.90159654 0.65000306 0.88845226 0.4037031 ]
[0 0 0 0 0 0 1 1 1 1]
```

Listing 10.11: Example output from generating a random input and output sequence.

10.2.3 Generate Multiple Sequences

We can define a function to create multiple sequences. The function below named `get_sequences()` takes the number of sequences to generate and the number of time steps per sequence as arguments and calls `get_sequence()` to generate the sequences. Once the specified number of sequences have been generated, the list of input and output sequences are reshaped to be three-dimensional and suitable for use with LSTMs.

```
# create multiple samples of cumulative sum sequences
def get_sequences(n_sequences, n_timesteps):
  seqX, seqY = list(), list()
  # create and store sequences
  for _ in range(n_sequences):
    X, y = get_sequence(n_timesteps)
    seqX.append(X)
    seqY.append(y)
  # reshape input and output for lstm
  seqX = array(seqX).reshape(n_sequences, n_timesteps, 1)
  seqY = array(seqY).reshape(n_sequences, n_timesteps, 1)
  return seqX, seqY
```

Listing 10.12: Function to generate sequences and format them for LSTM models.

We are now ready to start developing a Bidirectional LSTM for this problem.

10.3 Define and Compile the Model

First, we can define the complexity of the problem. We will limit the number of input time steps to a modest size; in this case, 10. This means the input shape will be 10 time steps with 1 feature.

```
# define problem
n_timesteps = 10
```

Listing 10.13: Example of configuring the problem.

Next, we need to define the hidden `LSTM` layer wrapped in a `Bidirectional` layer. We will use 50 memory cells in the `LSTM` hidden layer. The `Bidirectional` wrapper will double this, creating a second layer parallel to the first, also with 50 memory cells.

```
model.add(Bidirectional(LSTM(50, return_sequences=True), input_shape=(n_timesteps, 1)))
```

Listing 10.14: Example of adding the `Bidirectional` input layer.

A vector of 50 output values from each of the forward and backward `LSTM` hidden layers will be concatenated (the default merge method of the `Bidirectional` wrapper layer) to create a 100 element vector output. This is provided as input to a `Dense` layer that is wrapped in a `TimeDistributed` layer. This has the effect of reusing the weights of the `Dense` layer in order to create each output time step.

```
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
```

Listing 10.15: Example of adding the `TimeDistributed` output layer.

The `Bidirectional` LSTM layers return sequences to the `TimeDistributed` wrapped `Dense` layer. This has the effect of providing one concatenated 100 element vector to the `Dense` layer as input for each output time step. If a `TimeDistributed` wrapper was not used, a single 100 element vector would be provided to the `Dense` layer from which it would be required to output 10 time steps of classification. This would seem to be a more challenging problem for the model.

Putting this together, the model is defined below. The `sigmoid` activation is used in the `Dense` output layer and the binary log loss is optimized as each output time step is a binary classification of whether or not the cumulative sum threshold has been exceeded. The Adam implementation of gradient descent is used to optimize the weights and the classification accuracy is calculated during model training and evaluation.

```
# define LSTM
model = Sequential()
model.add(Bidirectional(LSTM(50, return_sequences=True), input_shape=(n_timesteps, 1)))
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])
print(model.summary())
```

Listing 10.16: Example of defining and compiling the Bidirectional LSTM model.

Running this code prints a summary of the compiled model. We can confirm that the Dense layer has 100 weights (plus the bias), one for each item in the 100 element concatenated vector provided from the Bidirectional wrapped LSTM hidden layer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional_1 (Bidirection (None, 10, 100)           20800
_________________________________________________________________
time_distributed_1 (TimeDist (None, 10, 1)             101
=================================================================
Total params: 20,901
Trainable params: 20,901
Non-trainable params: 0
_________________________________________________________________
```

Listing 10.17: Example output from defining and compiling the Bidirectional LSTM model.

10.4 Fit the Model

We can use the get_sequences() function to generate a large number of random examples on which to fit the model. We can simplify training by using the number of randomly generated sequences as a proxy for epochs. This allows us to generate a large number of examples, in this case 50,000, store them in memory, and fit them in one Keras epoch.

The batch size of 10 is used to balance learning speed and computational efficiency. Both the number of samples and batch size were found with some trial and error. Experiment with different values and see if you can train an accurate model with less computational effort.

```
# train LSTM
X, y = get_sequences(50000, n_timesteps)
model.fit(X, y, epochs=1, batch_size=10)
```

Listing 10.18: Example of fitting a compiled the Bidirectional LSTM model.

Fitting the model does not take long. A progress bar is provided during training and the log loss and model accuracy are updated each batch.

```
50000/50000 [==============================] - 97s - loss: 0.0508 - acc: 0.9817
```

Listing 10.19: Example output from fitting a compiled the Bidirectional LSTM model.

10.5 Evaluate the Model

We can evaluate the model by generating 100 new random sequences and calculating the accuracy of the predictions made by the fit model.

```
# evaluate LSTM
X, y = get_sequences(100, n_timesteps)
```

```
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))
```

Listing 10.20: Example of evaluating a fit the Bidirectional LSTM model.

Running this example prints both the log loss and accuracy. We can see that the model achieves 100% accuracy. The accuracy may vary when you run the example given the stochastic nature of the algorithms. You should see model skill in the high 90s. Try running the example a few times.

```
Loss: 0.016752, Accuracy: 100.000000
```

Listing 10.21: Example output from evaluating a fit the Bidirectional LSTM model.

10.6 Make Predictions with the Model

We can make predictions in a similar way as evaluating the model. In this case, we will generate 10 new random sequences, make predictions for each, and compare the predicted output sequence to the expected output sequence.

```
# make predictions
for _ in range(10):
  X, y = get_sequences(1, n_timesteps)
  yhat = model.predict_classes(X, verbose=0)
  exp, pred = y.reshape(n_timesteps), yhat.reshape(n_timesteps)
  print('y=%s, yhat=%s, correct=%s' % (exp, pred, array_equal(exp,pred)))
```

Listing 10.22: Example of making predictions with a fit the Bidirectional LSTM model.

Running the example prints both the expected (`y`) and predicted (`yhat`) output sequences and whether or not the predicted sequence was correct. We can see that, at least in this case, 2 of the 10 sequences were predicted with an error at one time step.

Your specific results will vary, but you should see similar behavior on average. This is a challenging problem, and even for a model that fits a large number of examples and shows good accuracy, it can still make errors in predicting new sequences.

```
y=[0 0 0 0 0 0 1 1 1 1], yhat=[0 0 0 0 0 0 1 1 1 1], correct=True
y=[0 0 0 0 1 1 1 1 1 1], yhat=[0 0 0 0 1 1 1 1 1 1], correct=True
y=[0 0 0 1 1 1 1 1 1 1], yhat=[0 0 0 1 1 1 1 1 1 1], correct=True
y=[0 0 0 0 0 0 0 1 1 1], yhat=[0 0 0 0 0 0 0 0 1 1], correct=False
y=[0 0 0 0 0 1 1 1 1 1], yhat=[0 0 0 0 0 1 1 1 1 1], correct=True
y=[0 0 0 1 1 1 1 1 1 1], yhat=[0 0 0 1 1 1 1 1 1 1], correct=True
y=[0 0 0 0 0 1 1 1 1 1], yhat=[0 0 0 0 0 0 1 1 1 1], correct=False
y=[0 0 0 0 1 1 1 1 1 1], yhat=[0 0 0 0 1 1 1 1 1 1], correct=True
y=[0 0 0 0 0 0 0 0 1 1], yhat=[0 0 0 0 0 0 0 0 1 1], correct=True
y=[0 0 0 1 1 1 1 1 1 1], yhat=[0 0 0 1 1 1 1 1 1 1], correct=True
```

Listing 10.23: Example output from making predictions with a fit the Bidirectional LSTM model.

10.7 Complete Example

The full example is listed below for completeness and your reference.

```
from random import random
from numpy import array
from numpy import cumsum
from numpy import array_equal
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import TimeDistributed
from keras.layers import Bidirectional

# create a cumulative sum sequence
def get_sequence(n_timesteps):
  # create a sequence of random numbers in [0,1]
  X = array([random() for _ in range(n_timesteps)])
  # calculate cut-off value to change class values
  limit = n_timesteps/4.0
  # determine the class outcome for each item in cumulative sequence
  y = array([0 if x < limit else 1 for x in cumsum(X)])
  return X, y

# create multiple samples of cumulative sum sequences
def get_sequences(n_sequences, n_timesteps):
  seqX, seqY = list(), list()
  # create and store sequences
  for _ in range(n_sequences):
    X, y = get_sequence(n_timesteps)
    seqX.append(X)
    seqY.append(y)
  # reshape input and output for lstm
  seqX = array(seqX).reshape(n_sequences, n_timesteps, 1)
  seqY = array(seqY).reshape(n_sequences, n_timesteps, 1)
  return seqX, seqY

# define problem
n_timesteps = 10

# define LSTM
model = Sequential()
model.add(Bidirectional(LSTM(50, return_sequences=True), input_shape=(n_timesteps, 1)))
model.add(TimeDistributed(Dense(1, activation='sigmoid')))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['acc'])
print(model.summary())

# train LSTM
X, y = get_sequences(50000, n_timesteps)
model.fit(X, y, epochs=1, batch_size=10)

# evaluate LSTM
X, y = get_sequences(100, n_timesteps)
loss, acc = model.evaluate(X, y, verbose=0)
print('Loss: %f, Accuracy: %f' % (loss, acc*100))

# make predictions
for _ in range(10):
  X, y = get_sequences(1, n_timesteps)
```

```
yhat = model.predict_classes(X, verbose=0)
exp, pred = y.reshape(n_timesteps), yhat.reshape(n_timesteps)
print('y=%s, yhat=%s, correct=%s' % (exp, pred, array_equal(exp,pred)))
```

Listing 10.24: Complete example of the Bidirectional LSTM on the Cumulative Sum problem.

10.8 Further Reading

This section provides some resources for further reading.

10.8.1 Research Papers

- *Bidirectional Recurrent Neural Networks*, 1997.
- *Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures*, 2005.
- *Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition*, 2005.
- *Speech Recognition with Deep Recurrent Neural Networks*, 2013.
  https://arxiv.org/abs/1303.5778

10.8.2 APIs

- `random()` Python API.
  https://docs.python.org/3/library/random.html
- `cumsum()` NumPy API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.cumsum.html
- `Bidirectional` Keras API.
  https://keras.io/layers/wrappers/#bidirectional

10.9 Extensions

Do you want to dive deeper into Bidirectional LSTMs? This section lists some challenging extensions to this lesson.

- List 5 examples of sequence prediction problems that may benefit from a Bidirectional LSTM.
- Tune the number of memory cells, training examples, and batch size to develop a smaller or faster trained model with 100% accuracy.
- Design and execute an experiment to compare model size to problem complexity (e.g. sequence length).
- Design and execute an experiment to compare forward, backward, and bidirectional LSTM input direction.

- Design and execute an experiment to compare the combination methods for the Bidirectional LSTM wrapper layer.

Post your extensions online and share the link with me; I'd love to see what you come up with!

10.10 Summary

In this lesson, you discovered how to develop a Bidirectional LSTM model. Specifically, you learned:

- The Bidirectional LSTM architecture and how to implement it in Keras.
- The cumulative sum prediction problem.
- How to develop a Bidirectional LSTM for the cumulative sum prediction problem.

In the next lesson, you will discover how to develop and evaluate the Generative LSTM model.