**How to Use the TimeDistributed Layer for Long Short-Term Memory Networks in Python**

by [**Jason Brownlee**](https://machinelearningmastery.com/author/jasonb/) on May 17, 2017 in [**Long Short-Term Memory Networks**](https://machinelearningmastery.com/category/lstm/)

Long Short-Term Networks or LSTMs are a popular and powerful type of Recurrent Neural Network, or RNN.

They can be quite difficult to configure and apply to arbitrary sequence prediction problems, even with well defined and “easy to use” interfaces like those provided in the Keras deep learning library in Python.

One reason for this difficulty in Keras is the use of the TimeDistributed wrapper layer and the need for some LSTM layers to return sequences rather than single values.

In this tutorial, you will discover different ways to configure LSTM networks for sequence prediction, the role that the TimeDistributed layer plays, and exactly how to use it.

After completing this tutorial, you will know:

* How to design a one-to-one LSTM for sequence prediction.
* How to design a many-to-one LSTM for sequence prediction without the TimeDistributed Layer.
* How to design a many-to-many LSTM for sequence prediction with the TimeDistributed Layer.

Let’s get started.



How to Use the TimeDistributed Layer for Long Short-Term Memory Networks in Python  
Photo by [jans canon](https://www.flickr.com/photos/43158397@N02/5774000092/), some rights reserved.

**Tutorial Overview**

This tutorial is divided into 5 parts; they are:

1. TimeDistributed Layer
2. Sequence Learning Problem
3. One-to-One LSTM for Sequence Prediction
4. Many-to-One LSTM for Sequence Prediction (without TimeDistributed)
5. Many-to-Many LSTM for Sequence Prediction (with TimeDistributed)

**Environment**

This tutorial assumes a Python 2 or Python 3 development environment with SciPy, NumPy, and Pandas installed.

The tutorial also assumes scikit-learn and Keras v2.0+ are installed with either the Theano or TensorFlow backend.

For help setting up your Python environment, see the post:

* [How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda](http://machinelearningmastery.com/setup-python-environment-machine-learning-deep-learning-anaconda/)

**Need help with LSTMs for Sequence Prediction?**

Take my free 7-day email course and discover 6 different LSTM architectures (with sample code).

Click to sign-up and also get a free PDF Ebook version of the course.

[**Start Your FREE Mini-Course Now!**](https://machinelearningmastery.lpages.co/leadbox/1403a9373f72a2%3A164f8be4f346dc/5754903989321728/)

**TimeDistributed Layer**

LSTMs are powerful, but hard to use and hard to configure, especially for beginners.

An added complication is the [TimeDistributed](https://keras.io/layers/wrappers/#timedistributed) Layer (and the former *TimeDistributedDense* layer) that is cryptically described as a layer wrapper:

*This wrapper allows us to apply a layer to every temporal slice of an input.*

How and when are you supposed to use this wrapper with LSTMs?

The confusion is compounded when you search through discussions about the wrapper layer on the Keras GitHub issues and StackOverflow.

For example, in the issue “[When and How to use TimeDistributedDense](https://github.com/fchollet/keras/issues/1029),” fchollet (Keras’ author) explains:

*TimeDistributedDense applies a same Dense (fully-connected) operation to every timestep of a 3D tensor.*

This makes perfect sense if you already understand what the TimeDistributed layer is for and when to use it, but is no help at all to a beginner.

This tutorial aims to clear up confusion around using the TimeDistributed wrapper with LSTMs with worked examples that you can inspect, run, and play with to help your concrete understanding.

**Sequence Learning Problem**

We will use a simple sequence learning problem to demonstrate the TimeDistributed layer.

In this problem, the sequence [0.0, 0.2, 0.4, 0.6, 0.8] will be given as input one item at a time and must be in turn returned as output, one item at a time.

Think of it as learning a simple echo program. We give 0.0 as input, we expect to see 0.0 as output, repeated for each item in the sequence.

We can generate this sequence directly as follows:



|  |  |
| --- | --- |
| 1  2  3  4 | from numpy import array  length = 5  seq = array([i/float(length) for i in range(length)])  print(seq) |

Running this example prints the generated sequence:



|  |  |
| --- | --- |
| 1 | [ 0.   0.2  0.4  0.6  0.8] |

The example is configurable and you can play with longer/shorter sequences yourself later if you like. Let me know about your results in the comments.

**One-to-One LSTM for Sequence Prediction**

Before we dive in, it is important to show that this sequence learning problem can be learned piecewise.

That is, we can reframe the problem into a dataset of input-output pairs for each item in the sequence. Given 0, the network should output 0, given 0.2, the network must output 0.2, and so on.

This is the simplest formulation of the problem and requires the sequence to be split into input-output pairs and for the sequence to be predicted one step at a time and gathered outside of the network.

The input-output pairs are as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | X, y  0.0, 0.0  0.2, 0.2  0.4, 0.4  0.6, 0.6  0.8, 0.8 |

The input for LSTMs must be three dimensional. We can reshape the 2D sequence into a 3D sequence with 5 samples, 1 time step, and 1 feature. We will define the output as 5 samples with 1 feature.



|  |  |
| --- | --- |
| 1  2 | X = seq.reshape(5, 1, 1)  y = seq.reshape(5, 1) |

We will define the network model as having 1 input with 1 time step. The first hidden layer will be an LSTM with 5 units. The output layer with be a fully-connected layer with 1 output.

The model will be fit with efficient ADAM optimization algorithm and the mean squared error loss function.

The batch size was set to the number of samples in the epoch to avoid having to make the LSTM stateful and manage state resets manually, although this could just as easily be done in order to update weights after each sample is shown to the network.

The complete code listing is provided below:



|  |  |
| --- | --- |
| 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 | from numpy import array  from keras.models import Sequential  from keras.layers import Dense  from keras.layers import LSTM  # prepare sequence  length = 5  seq = array([i/float(length) for i in range(length)])  X = seq.reshape(len(seq), 1, 1)  y = seq.reshape(len(seq), 1)  # define LSTM configuration  n\_neurons = length  n\_batch = length  n\_epoch = 1000  # create LSTM  model = Sequential()  model.add(LSTM(n\_neurons, input\_shape=(1, 1)))  model.add(Dense(1))  model.compile(loss='mean\_squared\_error', optimizer='adam')  print(model.summary())  # train LSTM  model.fit(X, y, epochs=n\_epoch, batch\_size=n\_batch, verbose=2)  # evaluate  result = model.predict(X, batch\_size=n\_batch, verbose=0)  for value in result:  print('%.1f' % value) |

Running the example first prints the structure of the configured network.

We can see that the LSTM layer has 140 parameters. This is calculated based on the number of inputs (1) and the number of outputs (5 for the 5 units in the hidden layer), as follows:



|  |  |
| --- | --- |
| 1  2  3  4 | n = 4 \* ((inputs + 1) \* outputs + outputs^2)  n = 4 \* ((1 + 1) \* 5 + 5^2)  n = 4 \* 35  n = 140 |

We can also see that the fully connected layer only has 6 parameters for the number of inputs (5 for the 5 inputs from the previous layer), number of outputs (1 for the 1 neuron in the layer), and the bias.



|  |  |
| --- | --- |
| 1  2  3 | n = inputs \* outputs + outputs  n = 5 \* 1 + 1  n = 6 |



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type)                 Output Shape              Param #  =================================================================  lstm\_1 (LSTM)                (None, 1, 5)              140  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_1 (Dense)              (None, 1, 1)              6  =================================================================  Total params: 146.0  Trainable params: 146  Non-trainable params: 0.0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

The network correctly learns the prediction problem.



|  |  |
| --- | --- |
| 1  2  3  4  5 | 0.0  0.2  0.4  0.6  0.8 |

**Many-to-One LSTM for Sequence Prediction (without  TimeDistributed)**

In this section, we develop an LSTM to output the sequence all at once, although without the TimeDistributed wrapper layer.

The input for LSTMs must be three dimensional. We can reshape the 2D sequence into a 3D sequence with 1 sample, 5 time steps, and 1 feature. We will define the output as 1 sample with 5 features.



|  |  |
| --- | --- |
| 1  2 | X = seq.reshape(1, 5, 1)  y = seq.reshape(1, 5) |

Immediately, you can see that the problem definition must be slightly adjusted to support a network for sequence prediction without a TimeDistributed wrapper. Specifically, output one vector rather build out an output sequence one step at a time. The difference may sound subtle, but it is important to understanding the role of the TimeDistributed wrapper.

We will define the model as having one input with 5 time steps. The first hidden layer will be an LSTM with 5 units. The output layer is afully-connectedd layer with 5 neurons.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # create LSTM  model = Sequential()  model.add(LSTM(5, input\_shape=(5, 1)))  model.add(Dense(length))  model.compile(loss='mean\_squared\_error', optimizer='adam')  print(model.summary()) |

Next, we fit the model for only 500 epochs and a batch size of 1 for the single sample in the training dataset.



|  |  |
| --- | --- |
| 1  2 | # train LSTM  model.fit(X, y, epochs=500, batch\_size=1, verbose=2) |

Putting this all together, the complete code listing is provided below.



|  |  |
| --- | --- |
| 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 | from numpy import array  from keras.models import Sequential  from keras.layers import Dense  from keras.layers import LSTM  # prepare sequence  length = 5  seq = array([i/float(length) for i in range(length)])  X = seq.reshape(1, length, 1)  y = seq.reshape(1, length)  # define LSTM configuration  n\_neurons = length  n\_batch = 1  n\_epoch = 500  # create LSTM  model = Sequential()  model.add(LSTM(n\_neurons, input\_shape=(length, 1)))  model.add(Dense(length))  model.compile(loss='mean\_squared\_error', optimizer='adam')  print(model.summary())  # train LSTM  model.fit(X, y, epochs=n\_epoch, batch\_size=n\_batch, verbose=2)  # evaluate  result = model.predict(X, batch\_size=n\_batch, verbose=0)  for value in result[0,:]:  print('%.1f' % value) |

Running the example first prints a summary of the configured network.

We can see that the LSTM layer has 140 parameters as in the previous section.

The LSTM units have been crippled and will each output a single value, providing a vector of 5 values as inputs to the fully connected layer. The time dimension or sequence information has been thrown away and collapsed into a vector of 5 values.

We can see that the fully connected output layer has 5 inputs and is expected to output 5 values. We can account for the 30 weights to be learned as follows:



|  |  |
| --- | --- |
| 1  2  3 | n = inputs \* outputs + outputs  n = 5 \* 5 + 5  n = 30 |

The summary of the network is reported as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type)                 Output Shape              Param #  =================================================================  lstm\_1 (LSTM)                (None, 5)                 140  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  dense\_1 (Dense)              (None, 5)                 30  =================================================================  Total params: 170.0  Trainable params: 170  Non-trainable params: 0.0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

The model is fit, printing loss information before finalizing and printing the predicted sequence.

The sequence is reproduced correctly, but as a single piece rather than stepwise through the input data. We may have used a Dense layer as the first hidden layer instead of LSTMs as this usage of LSTMs does not take much advantage of their full capability for sequence learning and processing.



|  |  |
| --- | --- |
| 1  2  3  4  5 | 0.0  0.2  0.4  0.6  0.8 |

**Many-to-Many LSTM for Sequence Prediction (with TimeDistributed)**

In this section, we will use the TimeDistributed layer to process the output from the LSTM hidden layer.

There are two key points to remember when using the TimeDistributed wrapper layer:

* **The input must be (at least) 3D**. This often means that you will need to configure your last LSTM layer prior to your TimeDistributed wrapped Dense layer to return sequences (e.g. set the “return\_sequences” argument to “True”).
* **The output will be 3D**. This means that if your TimeDistributed wrapped Dense layer is your output layer and you are predicting a sequence, you will need to resize your y array into a 3D vector.

We can define the shape of the output as having 1 sample, 5 time steps, and 1 feature, just like the input sequence, as follows:



|  |  |
| --- | --- |
| 1 | y = seq.reshape(1, length, 1) |

We can define the LSTM hidden layer to return sequences rather than single values by setting the “*return\_sequences*” argument to true.



|  |  |
| --- | --- |
| 1 | model.add(LSTM(n\_neurons, input\_shape=(length, 1), return\_sequences=True)) |

This has the effect of each LSTM unit returning a sequence of 5 outputs, one for each time step in the input data, instead of single output value as in the previous example.

We also can use the TimeDistributed on the output layer to wrap a fully connected Dense layer with a single output.



|  |  |
| --- | --- |
| 1 | model.add(TimeDistributed(Dense(1))) |

The single output value in the output layer is key. It highlights that we intend to output one time step from the sequence for each time step in the input. It just so happens that we will process 5 time steps of the input sequence at a time.

The TimeDistributed achieves this trick by applying the same Dense layer (same weights) to the LSTMs outputs for one time step at a time. In this way, the output layer only needs one connection to each LSTM unit (plus one bias).

For this reason, the number of training epochs needs to be increased to account for the smaller network capacity. I doubled it from 500 to 1000 to match the first one-to-one example.

Putting this together, the full code listing is provided below.



|  |  |
| --- | --- |
| 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 | from numpy import array  from keras.models import Sequential  from keras.layers import Dense  from keras.layers import TimeDistributed  from keras.layers import LSTM  # prepare sequence  length = 5  seq = array([i/float(length) for i in range(length)])  X = seq.reshape(1, length, 1)  y = seq.reshape(1, length, 1)  # define LSTM configuration  n\_neurons = length  n\_batch = 1  n\_epoch = 1000  # create LSTM  model = Sequential()  model.add(LSTM(n\_neurons, input\_shape=(length, 1), return\_sequences=True))  model.add(TimeDistributed(Dense(1)))  model.compile(loss='mean\_squared\_error', optimizer='adam')  print(model.summary())  # train LSTM  model.fit(X, y, epochs=n\_epoch, batch\_size=n\_batch, verbose=2)  # evaluate  result = model.predict(X, batch\_size=n\_batch, verbose=0)  for value in result[0,:,0]:  print('%.1f' % value) |

Running the example, we can see the structure of the configured network.

We can see that as in the previous example, we have 140 parameters in the LSTM hidden layer.

The fully connected output layer is a very different story. In fact, it matches the one-to-one example exactly. One neuron that has one weight for each LSTM unit in the previous layer, plus one for the bias input.

This does two important things:

* Allows the problem to be framed and learned as it was defined, that is one input to one output, keeping the internal process for each time step separate.
* Simplifies the network by requiring far fewer weights such that only one time step is processed at a time.

The one simpler fully connected layer is applied to each time step in the sequence provided from the previous layer to build up the output sequence.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type)                 Output Shape              Param #  =================================================================  lstm\_1 (LSTM)                (None, 5, 5)              140  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  time\_distributed\_1 (TimeDist (None, 5, 1)              6  =================================================================  Total params: 146.0  Trainable params: 146  Non-trainable params: 0.0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Again, the network learns the sequence.



|  |  |
| --- | --- |
| 1  2  3  4  5 | 0.0  0.2  0.4  0.6  0.8 |

We can think of the framing of the problem with time steps and a TimeDistributed layer as a more compact way of implementing the one-to-one network in the first example. It may even be more efficient (space or time wise) at a larger scale.

**Further Reading**

Below are some resources and discussions on the TimeDistributed layer you may like to dive in into.

* [TimeDistributed Layer](https://keras.io/layers/wrappers/#timedistributed) in the Keras API
* [TimeDistributed](https://github.com/fchollet/keras/blob/master/keras/layers/wrappers.py#L56) code on GitHub
* [The difference between ‘Dense’ and ‘TimeDistributedDense’ of ‘Keras’](http://datascience.stackexchange.com/questions/10836/the-difference-between-dense-and-timedistributeddense-of-keras) on StackExchange
* [When and How to use TimeDistributedDense](https://github.com/fchollet/keras/issues/1029) on GitHub

**Summary**

In this tutorial, you discovered how to develop LSTM networks for sequence prediction and the role of the TimeDistributed layer.

Specifically, you learned:

* How to design a one-to-one LSTM for sequence prediction.
* How to design a many-to-one LSTM for sequence prediction without the TimeDistributed Layer.
* How to design a many-to-many LSTM for sequence prediction with the TimeDistributed Layer.

Do you have any questions?  
Ask your questions in the comments below and I will do my best to answer them.