How to Handle Missing Timesteps in Sequence Prediction Problems with Python

by **Jason Brownlee** on June 21, 2017 in **Long Short-Term Memory Networks**

It is common to have missing observations from sequence data.

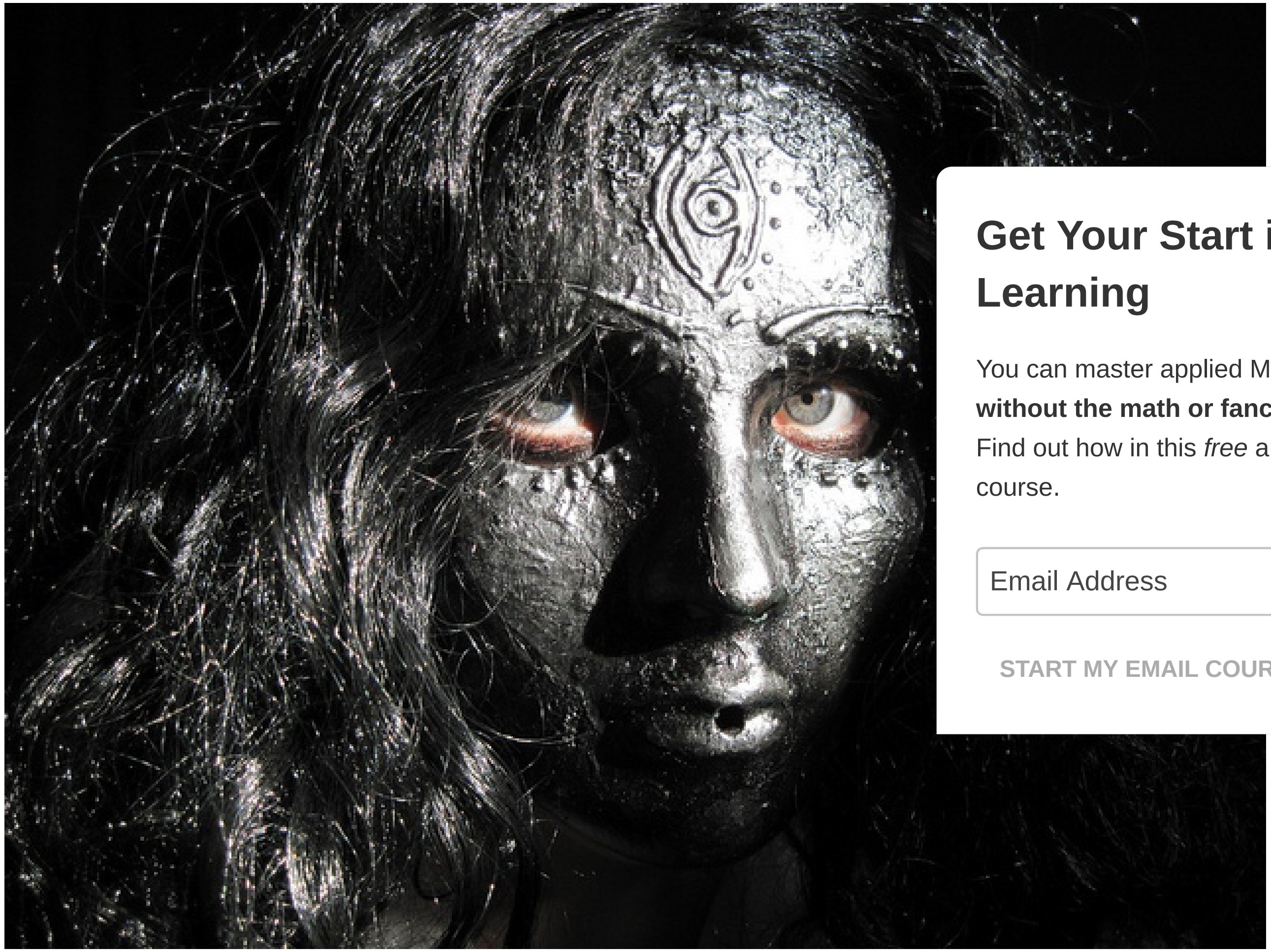
Data may be corrupt or unavailable, but it is also possible that your data has variable length sequences by definition. Those sequences with fewer timesteps may be considered to have missing values.

In this tutorial, you will discover how you can handle data with missing values for sequence prediction problems in Python with the Keras deep learning library.

After completing this tutorial, you will know:

- How to remove rows that contain a missing timestep.
- How to mark missing timesteps and force the network to learn their meaning.
- How to mask missing timesteps and exclude them from calculations in the model.

Let's get started.



How to Handle Missing Timesteps in Sequence Prediction Problems with Python
Photo by Sybil Liberty, some rights reserved.

Overview

This section is divided into 3 parts; they are:

1. Echo Sequence Prediction Problem
2. Handling Missing Sequence Data
3. Learning With Missing Sequence Values

Environment

This tutorial assumes you have a Python SciPy environment installed. You can use either Python 2 or 3 with this example.

This tutorial assumes you have Keras (v2.0.4+) installed with either the TensorFlow (v1.1.0+) or The

This tutorial also assumes you have scikit-learn, Pandas, NumPy, and Matplotlib installed.

If you need help setting up your Python environment, see this post:

- How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda

Echo Sequence Prediction Problem

The echo problem is a contrived sequence prediction problem where the objective is to remember a called a lag observation.

For example, the simplest case is to predict the observation from the previous timestep that is, echo

```
Time 1: Input 45
Time 2: Input 23, Output 45
Time 3: Input 73, Output 23
...
```

The question is, what do we do about timestep 1?

We can implement the echo sequence prediction problem in Python.

This involves two steps: the generation of random sequences and the transformation of random seq[redacted]

Generate Random Sequence

We can generate sequences of random values between 0 and 1 using the random() function in the random module.

We can put this in a function called generate_sequence() that will generate a sequence of random floating point values for the desired number of timesteps.

This function is listed below.

```
# generate a sequence of random values
def generate_sequence(n_timesteps):
    return [random() for _ in range(n_timesteps)]
```

Frame as Supervised Learning

Sequences must be framed as a supervised learning problem when using neural networks.

That means the sequence needs to be divided into input and output pairs.

The problem can be framed as making a prediction based on a function of the current and previous t

[redacted]

Or more formally:

```
y(t) = f(X(t), X(t-1))
```

Where y(t) is the desired output for the current timestep, f() is the function we are seeking to approximate with our neural network, and X(t) and X(t-1) are the observations for the current and previous timesteps.

The output could be equal to the previous observation, for example, y(t) = X(t-1), but it could as easily be y(t) = X(t). The model that we train on this problem does not know the true formulation and must learn this relationship.

This mimics real sequence prediction problems where we specify the model as a function of some fixed set of sequenced timesteps, but we don't know the actual functional relationship from past observations to the desired output value.

We can implement this framing of an echo problem as a supervised learning problem in python.

The Pandas shift() function can be used to create a shifted version of the sequence that can be usec p. This can be concatenated with the raw sequence to provide the X(t-1) and X(t) input values.

```
df = DataFrame(sequence)
df = concat([df.shift(1), df], axis=1)
```

We can then take the values from the Pandas DataFrame as the input sequence (X) and use the firs

```
# specify input and output data
X, y = values, values[:, 0]
```

Putting this all together, we can define a function that takes the number of timesteps as an argument d generate_data().

```
# generate data for the lstm
def generate_data(n_timesteps):
    # generate sequence
    sequence = generate_sequence(n_timesteps)
    sequence = array(sequence)
    # create lag
    df = DataFrame(sequence)
    df = concat([df.shift(1), df], axis=1)
    values = df.values
    # specify input and output data
    X, y = values, values[:, 0]
    return X, y
```

[redacted]

Sequence Problem Demonstration

We can tie the generate_sequence() and generate_data() code together into a worked example.

The complete example is listed below.

```
from random import random
from numpy import array
from pandas import concat
from pandas import DataFrame

# generate a sequence of random values
def generate_sequence(n_timesteps):
    return [random() for _ in range(n_timesteps)]

# generate data for the lstm
def generate_data(n_timesteps):
    # generate sequence
    sequence = generate_sequence(n_timesteps)
    sequence = array(sequence)
    # create lag
    df = DataFrame(sequence)
    df = concat([df.shift(1), df], axis=1)
    values = df.values
    # specify input and output data
    X, y = values, values[:, 0]
    return X, y

# generate sequence
n_timesteps = 10
X, y = generate_data(n_timesteps)
# print sequence
for i in range(n_timesteps):
    print(X[i], '=>', y[i])
```

Running this example generates a sequence, converts it to a supervised representation, and prints each X,y pair.

```
[ nan 0.18961404] => nan
[ 0.18961404 0.25956078] => 0.189614044109
[ 0.25956078 0.30322084] => 0.259560776929
[ 0.30322084 0.72581287] => 0.303220844801
[ 0.72581287 0.02916655] => 0.725812865047
[ 0.02916655 0.88711086] => 0.0291665472554
[ 0.88711086 0.34267107] => 0.88711086298
[ 0.34267107 0.3844453 ] => 0.342671068373
[ 0.3844453 0.89759621] => 0.384445299683
[ 0.89759621 0.95278264] => 0.897596208691
```

We can see that we have NaN values on the first row.

This is because we do not have a prior observation for the first value in the sequence. We have to fill that space with something.

But we cannot fit a model with NaN inputs.

Handling Missing Sequence Data

There are two main ways to handle missing sequence data.

They are to remove rows with missing data and to fill the missing timesteps with another value.

For more general methods for handling missing data, see the post:

- How to Handle Missing Data with Python

The best approach for handling missing sequence data will depend on your problem and your chose exploring each method and see what works best.

Remove Missing Sequence Data

In the case where we are echoing the observation in the previous timestep, the first row of data does

That is, in the example above, given the input:

```
[        nan  0.18961404]
```

and the output:

```
nan
```

There is nothing meaningful that can be learned or predicted.

[redacted]

The best case here is to delete this row.

We can do this during the formulation of the sequence as a supervised learning problem by removing all rows that contain a NaN value. Specifically, the dropna() function can be called prior to splitting the data into X and y components.

The complete example is listed below:

```
from random import random
from numpy import array
from pandas import concat
from pandas import DataFrame

# generate a sequence of random values
def generate_sequence(n_timesteps):
    return [random() for _ in range(n_timesteps)]

# generate data for the lstm
def generate_data(n_timesteps):
    # generate sequence
    sequence = generate_sequence(n_timesteps)
    sequence = array(sequence)
    # create lag
    df = DataFrame(sequence)
    df = concat([df.shift(1), df], axis=1)
    # remove rows with missing values
    df.dropna(inplace=True)
    values = df.values
    # specify input and output data
    X, y = values, values[:, 0]
    return X, y

# generate sequence
n_timesteps = 10
X, y = generate_data(n_timesteps)
# print sequence
for i in range(len(X)):
    print(X[i], '=>', y[i])
```

Running the example results in 9 X,y pairs instead of 10, with the first row removed.

```
[ 0.60619475  0.24408238] => 0.606194746194
[ 0.24408238  0.44873712] => 0.244082383195
[ 0.44873712  0.92939547] => 0.448737123424
[ 0.92939547  0.74481645] => 0.929395472523
[ 0.74481645  0.69891311] => 0.744816453809
[ 0.69891311  0.8420314 ] => 0.69891310578
[ 0.8420314   0.58627624] => 0.842031399202
[ 0.58627624  0.48125348] => 0.586276240292
[ 0.48125348  0.75057094] => 0.481253484036
```

[redacted]

Replace Missing Sequence Data

In the case when the echo problem is configured to echo the observation at the current timestep, then the first row will contain meaningful information.

For example, we can change the definition of y from values[:, 0] to values[:, 0] and re-run the demonstration to produce a sample of this problem, as follows:

```
[        nan  0.50513289] => 0.505132894821
[ 0.50513289  0.22879667] => 0.228796667421
[ 0.22879667  0.66980995] => 0.669809946421
[ 0.66980995  0.10445146] => 0.104451463568
[ 0.10445146  0.70642423] => 0.70642422679
[ 0.70642423  0.10198636] => 0.101986362328
[ 0.10198636  0.49648033] => 0.496480332278
[ 0.49648033  0.06201137] => 0.0620113728356
[ 0.06201137  0.40653087] => 0.406530870804
[ 0.40653087  0.63299264] => 0.632992635565
```

We can see that the first row is given the input:

```
[        nan  0.50513289]
```

and the output:

```
0.505132894821
```

Which could be learned from the input.

The problem is, we still have a NaN value to handle.

Instead of removing the rows with NaN values, we can replace all NaN values with a specific value that does not appear naturally in the input, such as -1. To do this, we can use the fillna() Pandas function.

The complete example is listed below:

[redacted]

```python
from random import random
from numpy import array
from pandas import concat
from pandas import DataFrame

# generate a sequence of random values
def generate_sequence(n_timesteps):
    return [random() for _ in range(n_timesteps)]

# generate data for the lstm
def generate_data(n_timesteps):
    # generate sequence
    sequence = generate_sequence(n_timesteps)
    sequence = array(sequence)
    # create lag
    df = DataFrame(sequence)
    df = concat([df.shift(1), df], axis=1)
    # replace missing values with -1
    df.fillna(-1, inplace=True)
    values = df.values
    # specify input and output data
    X, y = values, values[:, 1]
    return X, y

# generate sequence
n_timesteps = 10
X, y = generate_data(n_timesteps)
# print sequence
for i in range(len(X)):
    print(X[i], '=>', y[i])
```

Running the example, we can see that the NaN value in the first column of the first row was replaced

```
[-1.  0.94641256] => 0.946412559807
[ 0.94641256 0.11958645] => 0.119586451733
[ 0.11958645 0.50597771] => 0.505977714614
[ 0.50597771 0.92496641] => 0.924966407025
[ 0.92496641 0.15011979] => 0.150119790096
[ 0.15011979 0.69387197] => 0.693871974256
[ 0.69387197 0.9194518 ] => 0.919451802966
[ 0.9194518 0.78690337] => 0.786903370269
[ 0.78690337 0.17017999] => 0.170179993691
[ 0.17017999 0.82286572] => 0.822865722747
```

Learning with Missing Sequence Values

There are two main options when learning a sequence prediction problem with marked missing valu[redacted]

The problem can be modeled as-is and we can encourage the model to learn that a specific value means “missing.” Alternately, the special missing values can be masked and explicitly excluded from the prediction calculations.

We will take a look at both cases for the contrived “echo the current observation” problem with two inputs.

Learning Missing Values

We can develop an LSTM for the prediction problem.

The input is defined by 2 timesteps with 1 feature. A small LSTM with 5 memory units in the first hidden layer is defined and a single output layer with a linear activation function.

The network will be fit using the mean squared error loss function and the efficient ADAM optimizatic

```
# define model
model = Sequential()
model.add(LSTM(5, input_shape=(2, 1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

To ensure that the model learns a generalized solution to the problem, that is to always returns the ir w random sequence every epoch. The network will be fit for 500 epochs and updates will be performec (batch_size=1).

```
# fit model
for i in range(500):
    X, y = generate_data(n_timesteps)
    model.fit(X, y, epochs=1, batch_size=1, verbose=2)
```

Once fit, another random sequence will be generated and the predictions from the model will be compared to the expected values. This will provide a concrete idea of the skill of the model.

```
# evaluate model on new data
X, y = generate_data(n_timesteps)
yhat = model.predict(X)
for i in range(len(X)):
    print('Expected', y[i,0], 'Predicted', yhat[i,0])
```

[redacted]

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

```python
from random import random
from numpy import array
from pandas import concat
from pandas import DataFrame
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# generate a sequence of random values
def generate_sequence(n_timesteps):
    return [random() for _ in range(n_timesteps)]

# generate data for the lstm
def generate_data(n_timesteps):
    # generate sequence
    sequence = generate_sequence(n_timesteps)
    sequence = array(sequence)
    # create lag
    df = DataFrame(sequence)
    df = concat([df.shift(1), df], axis=1)
    # replace missing values with -1
    df.fillna(-1, inplace=True)
    values = df.values
    # specify input and output data
    X, y = values, values[:, 1]
    # reshape
    X = X.reshape(len(X), 2, 1)
    y = y.reshape(len(y), 1)
    return X, y

n_timesteps = 10
# define model
model = Sequential()
model.add(LSTM(5, input_shape=(2, 1)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
# fit model
for i in range(500):
    X, y = generate_data(n_timesteps)
    model.fit(X, y, epochs=1, batch_size=1, verbose=2)
# evaluate model on new data
X, y = generate_data(n_timesteps)
yhat = model.predict(X)
for i in range(len(X)):
    print('Expected', y[i,0], 'Predicted', yhat[i,0])
```

[redacted]

Running the example prints the loss each epoch and compares the expected vs. the predicted output at the end of a run for one sequence.

Reviewing the final predictions, we can see that the network learned the problem and predicted "good enough" outputs, even in the presence of missing values.

```
...
Epoch 1/1
0s - loss: 1.5992e-04
Epoch 1/1
0s - loss: 1.3409e-04
Epoch 1/1
0s - loss: 1.1581e-04
Epoch 1/1
0s - loss: 2.6176e-04
Epoch 1/1
0s - loss: 8.8303e-05
Expected 0.390784174343 Predicted 0.394238
Expected 0.688580469278 Predicted 0.690463
Expected 0.347155799665 Predicted 0.329972
Expected 0.345075533266 Predicted 0.333037
Expected 0.456591840482 Predicted 0.450145
Expected 0.842125610156 Predicted 0.839923
Expected 0.354087132135 Predicted 0.342418
Expected 0.601406667694 Predicted 0.60228
Expected 0.368929815424 Predicted 0.351224
Expected 0.716420996314 Predicted 0.719275
```

You could experiment further with this example and mark 50% of the t-1 observations for a given sec he model over time.

Masking Missing Values

The marked missing input values can be masked from all calculations in the network.

We can do this by using a Masking layer as the first layer to the network.

When defining the layer, we can specify which value in the input to mask. If all features for a timestep contain the masked value, then the whole timestep will be excluded from calculations.

This provides a middle ground between excluding the row completely and forcing the network to lear

Because the Masking layer is the first in the network, it must specify the expected shape of the input, [redacted]

```
model.add(Masking(mask_value=-1, input_shape=(2, 1)))
```

We can tie all of this together and re-run the example. The complete code listing is provided below.

```
from random import random
from numpy import array
from pandas import concat
from pandas import DataFrame
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Masking

# generate a sequence of random values
def generate_sequence(n_timesteps):
    return [random() for _ in range(n_timesteps)]

# generate data for the lstm
def generate_data(n_timesteps):
    # generate sequence
    sequence = generate_sequence(n_timesteps)
    sequence = array(sequence)
    # create lag
    df = DataFrame(sequence)
    df = concat([df.shift(1), df], axis=1)
    # replace missing values with -1
    df.fillna(-1, inplace=True)
    values = df.values
    # specify input and output data
    X, y = values, values[:, 1]
    # reshape
    X = X.reshape(len(X), 2, 1)
    y = y.reshape(len(y), 1)
    return X, y

n_timesteps = 10
# define model
model = Sequential()
model.add(Masking(mask_value=-1, input_shape=(2, 1)))
model.add(LSTM(5))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
# fit model
for i in range(500):
    X, y = generate_data(n_timesteps)
    model.fit(X, y, epochs=1, batch_size=1, verbose=2)
# evaluate model on new data
X, y = generate_data(n_timesteps)
yhat = model.predict(X)
for i in range(len(X)):
    print('Expected', y[i,0], 'Predicted', yhat[i,0])
```

Again, the loss is printed each epoch and the predictions are compared to expected values for a final sequence.

Again, the predictions appear good enough to a few decimal places.

```
...
Epoch 1/1
0s - loss: 1.0252e-04
Epoch 1/1
0s - loss: 6.5545e-05
Epoch 1/1
0s - loss: 3.0831e-05
Epoch 1/1
0s - loss: 1.8548e-04
Epoch 1/1
0s - loss: 7.4286e-05
Expected 0.550889403319 Predicted 0.538004
Expected 0.24252028132 Predicted 0.243288
Expected 0.718869927574 Predicted 0.724669
Expected 0.355185878917 Predicted 0.347479
Expected 0.240554707978 Predicted 0.242719
Expected 0.769765554707 Predicted 0.776608
Expected 0.660782450416 Predicted 0.656321
Expected 0.692962017672 Predicted 0.694851
Expected 0.0485233839401 Predicted 0.0722362
Expected 0.35192019185 Predicted 0.339201
```

Which Method to Choose?

These one-off experiments are not sufficient to evaluate what would work best on the simple echo sequence prediction problem.

They do provide templates that you can use on your own problems.

I would encourage you to explore the 3 different ways of handling missing values in your sequence prediction problems. They were:

- Removing rows with missing values.

[redacted]

- Mark and learn missing values.
- Mask and learn without missing values.

Try each approach on your sequence prediction problem and double down on what appears to work best.

Summary

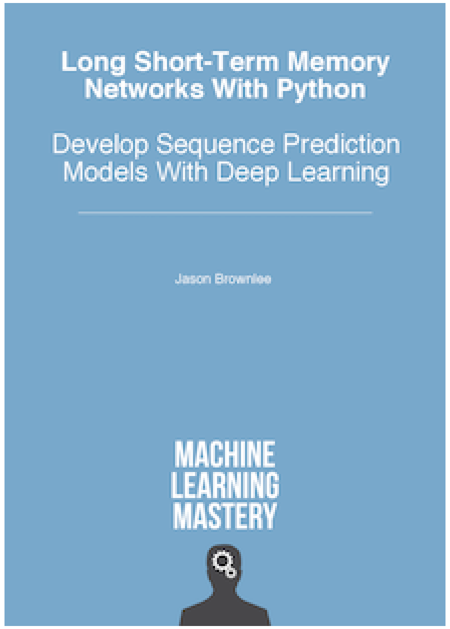
It is common to have missing values in sequence prediction problems if your sequences have variable lengths.

In this tutorial, you discovered how to handle missing data in sequence prediction problems in Python with Keras.

Specifically, you learned:

- How to remove rows that contain a missing value.
- How to mark missing values and force the model to learn their meaning.
- How to mask missing values to exclude them from calculations in the model.

Do you have any questions about handling missing sequence data?
Ask your questions in the comments and I will do my best to answer.



About Jason Brownlee

Dr. Jason Brownlee is a husband, proud father, academic researcher, author, professional devel
to helping developers get started and get good at applied machine learning. Learn more.

View all posts by Jason Brownlee →

< Data Preparation for Variable Length Input Sequences

A Gentle Introduction to Backpropagation Through Time >

10 Responses to *How to Handle Missing Timesteps in Sequence Prediction Problems with Python*

Nader June 21, 2017 at 9:57 am # REPLY

Fantastic !

Jason Brownlee June 22, 2017 at 6:03 am # REPLY

I'm glad it helps Nader.

James Mashiyane June 23, 2017 at 7:21 am #

I really like your books, they have really helped me, I'm using 4 of them Time Series Forecastin Learning from scratch. Especially the Machine Learning from scratch has helped a lot with my python sk Tensor Flow and Keras will be coming soon. Thanks a lot.

Jason Brownlee June 23, 2017 at 7:41 am #

Thanks for your support James.

Adam July 28, 2017 at 1:33 pm # REPLY

If I want to normalize input data, I should replace Missing data first or normalizing input data?

Jason Brownlee July 29, 2017 at 8:02 am #

REPLY

Yes, I would impute before scaling.

Jeff Lim August 29, 2017 at 12:36 pm #

REPLY

On Replace Missing Sequence Data, we should change the definition of y from values[:, 0] to values[:, 1], right?

Jason Brownlee August 29, 2017 at 5:13 pm #

Yes, from the post:

In the case when the echo problem is configured to echo the observation at the current timeste

baojia li September 1, 2017 at 8:37 pm #

if we change generate_sequence:
def generate_sequence(n_timesteps):
return random.randint(34,156,n_timesteps)

The results and the real value will be a lot of error
why?

Jason Brownlee September 2, 2017 at 6:08 am #

REPLY

Because neural networks cannot predict a pseudo random series.

[redacted]

Leave a Reply

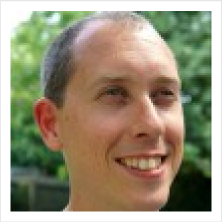
Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

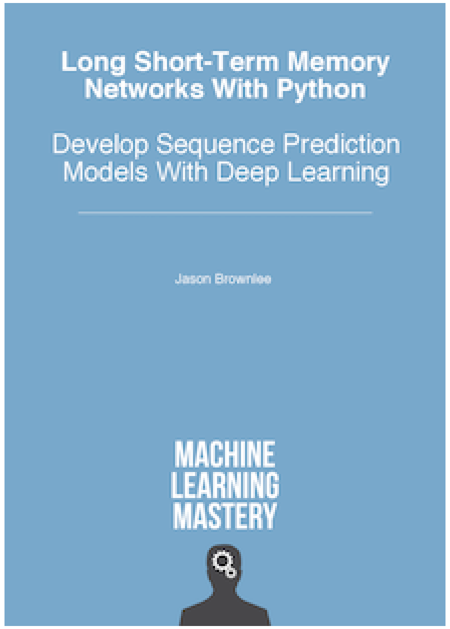
Welcome to Machine Learning Mastery



Hi, I'm Dr. Jason Brownlee.
My goal is to make practitioners like YOU awesome at applied machine learning.

Read More

Deep Learning for Sequence Prediction



Long Short-Term Memory Networks With Python

Develop Sequence Prediction Models With Deep Learning

Jason Brownlee

MACHINE LEARNING MASTERY

POPULAR

Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras
JULY 21, 2016

Your First Machine Learning Project in Python Step-By-Step
JUNE 10, 2016

Develop Your First Neural Network in Python With Keras Step-By-Step
MAY 24, 2016

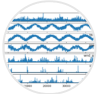
Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras
JULY 26, 2016

How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda
MARCH 13, 2017

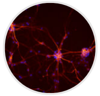
Time Series Forecasting with the Long Short-Term Memory Network in Python
APRIL 7, 2017

Multi-Class Classification Tutorial with the Keras Deep Learning Library
JUNE 2, 2016

Regression Tutorial with the Keras Deep Learning Library in Python
JUNE 9, 2016



Multivariate Time Series Forecasting with LSTMs in Keras
AUGUST 14, 2017



How to Implement the Backpropagation Algorithm From Scratch In Python
NOVEMBER 7, 2016

Privacy | Contact | About