Demonstration of Memory with a Long Short-Term Memory Network in Python

by **Jason Brownlee** on May 12, 2017 in **Long Short-Term Memory Networks**

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning over long sequences.

This differentiates them from regular multilayer neural networks that do not have memory and can only learn a mapping between input and output patterns.

It is important to understand the capabilities of complex neural networks like LSTMs on small contrived problems as this understanding will help you scale the network up to large and even very large problems.

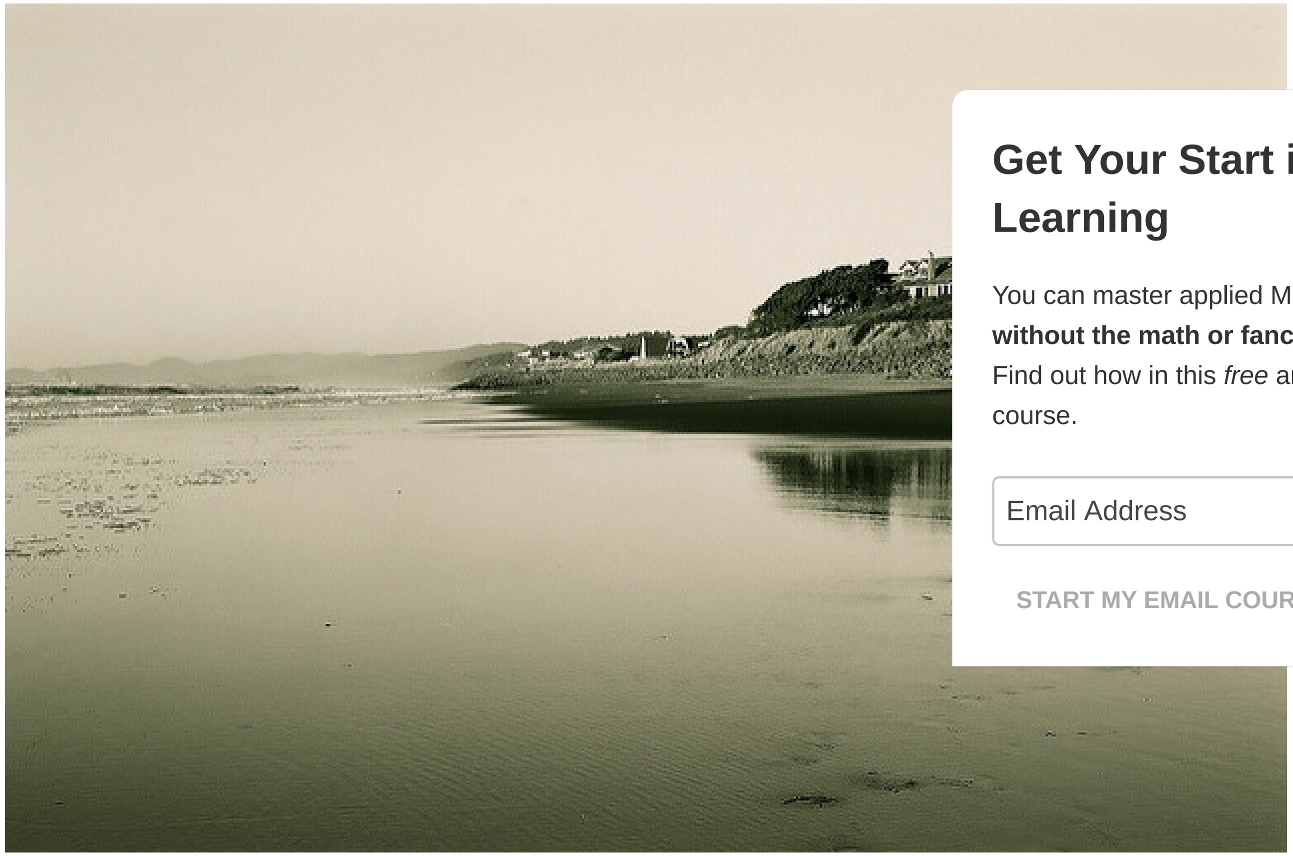
[redacted]

In this tutorial, you will discover the capability of LSTMs to remember and recall.

After completing this tutorial, you will know:

- How to define a small sequence prediction problem that only an RNN like LSTMs can solve using memory.
- How to transform the problem representation so that it is suitable for learning by LSTMs.
- How to design an LSTM to solve the problem correctly.

Let's get started.



A Demonstration of Memory in a Long Short-Term Memory Network
Photo by crazlei, some rights reserved.

Environment

This tutorial assumes you have a working Python 2 or 3 environment with SciPy, Keras 2.0 or higher with a TensorFlow or Theano 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

Sequence Problem Description

The problem is to predict values of a sequence one at a time.

Given one value in the sequence, the model must predict the next value in the sequence. For examp st
predict the value "1".

There are two different sequences that the model must learn and correctly predict.

A wrinkle is that there is conflicting information between the two sequences and that the model must |.
the sequence it is currently predicting) in order to correctly predict each full sequence.

This wrinkle is important to prevent the model from memorizing each single-step input-output pair of e
model may be inclined to do.

The two sequences to be learned are as follows:

- 3, 0, 1, 2, 3
- 4, 0, 1, 2, 4

We can see that the first value of the sequence is repeated as the last value of the sequence. This is the indicator that provides context to the model as to which sequence it is working on.

The conflict is the transition from the second to last items in each sequence. In sequence one, a "2" is given as an input and a "3" must be predicted, whereas in sequence two, a "2" is given as input and a "4" must be predicted.

This is a problem that a multilayer Perceptron and other non-recurrent neural networks cannot learn. [redacted]

This is a simplified version of "*Experiment 2*" used to demonstrate LSTM long-term memory capabilities in Hochreiter and Schmidhuber's 1997 paper Long Short Term Memory (PDF).

Problem Representation

This section is divided into 3 parts; they are:

1. One Hot Encoding
2. Input-Output Pairs
3. Reshape Data

One Hot Encoding

We will use a one hot encoding to represent the learning problem for the LSTM.

That is, each input and output value will be represented as a binary vector with 5 elements, because the alphabet of the problem is 5 unique values.

For example, the 5 values of [0, 1, 2, 3, 4] are represented as the following 5 binary vectors:

```
0: [1, 0, 0, 0, 0]
```

```
1: [0, 1, 0, 0, 0]
2: [0, 0, 1, 0, 0]
3: [0, 0, 0, 1, 0]
4: [0, 0, 0, 0, 1]
```

[redacted]

We can do this with a simple function that will take a sequence and return a list of binary vectors for each value in the sequence. The function *encode()* below implements this behavior.

```
# binary encode an input pattern, return a list of binary vectors
def encode(pattern, n_unique):
    encoded = list()
    for value in pattern:
        row = [0.0 for x in range(n_unique)]
        row[value] = 1.0
        encoded.append(row)
    return encoded
```

We can test it on the first sequence and print the resulting list of binary vectors. The complete examp

```
# binary encode an input pattern, return a list of binary vectors
def encode(pattern, n_unique):
    encoded = list()
    for value in pattern:
        row = [0.0 for x in range(n_unique)]
        row[value] = 1.0
        encoded.append(row)
    return encoded

seq1 = [3, 0, 1, 2, 3]
encoded = encode(seq1, 5)
for vector in encoded:
    print(vector)
```

Running the example prints each binary vector. Note that we use the floating point values 0.0 and 1. for the model.

```
[0.0, 0.0, 0.0, 1.0, 0.0]
[1.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 1.0, 0.0]
```

Input-Output Pairs

[redacted]

The next step is to split a sequence of encoded values into input-output pairs.

This is a supervised learning representation of the problem such that machine learning problems can learn how to map an input pattern (*X*) to an output pattern (*y*).

For example, the first sequence has the following input-output pairs to be learned:

```
X,  y
3,  0
0,  1
1,  2
2,  3
```

Instead of the raw numbers, we must create these mapping pairs from the one hot encoded binary vectors.

For example, the first input-output pairs for 3->0 would be:

```
X,              y
[0, 0, 0, 1, 0]     [1, 0, 0, 0, 0]
```

Below is a function named *to_xy_pairs()* that will create lists of *X* and *y* patterns given a list of encod

```
# create input/output pairs of encoded vectors, returns X, y
def to_xy_pairs(encoded):
    X,y = list(),list()
    for i in range(1, len(encoded)):
        X.append(encoded[i-1])
        y.append(encoded[i])
    return X, y
```

We can put this together with the one hot encoding function above and print the encoded input and o

```
# binary encode an input pattern, return a list of binary vectors
def encode(pattern, n_unique):
    encoded = list()
    for value in pattern:
        row = [0.0 for x in range(n_unique)]
        row[value] = 1.0
        encoded.append(row)
    return encoded

# create input/output pairs of encoded vectors, returns X, y
```

```
def to_xy_pairs(encoded):
    X,y = list(),list()
    for i in range(1, len(encoded)):
        X.append(encoded[i-1])
        y.append(encoded[i])
    return X, y

seq1 = [3, 0, 1, 2, 3]
encoded = encode(seq1, 5)
X, y = to_xy_pairs(encoded)
for i in range(len(X)):
    print(X[i], y[i])
```

[redacted]

Running the example prints the input and output pairs for each step in the sequence.

```
[0.0, 0.0, 0.0, 1.0, 0.0] [1.0, 0.0, 0.0, 0.0, 0.0]
[1.0, 0.0, 0.0, 0.0, 0.0] [0.0, 1.0, 0.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0, 0.0] [0.0, 0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 1.0, 0.0, 0.0] [0.0, 0.0, 0.0, 1.0, 0.0]
```

Reshape Data

The final step is to reshape the data so that it can be used by the LSTM network directly.

The Keras LSTM expects input patterns (X) as a three-dimensional NumPy array with the dimension

In the case of one sequence of input data, the dimensions will be [4, 1, 5] because we have 4 rows c
each row.

We can create a 2D NumPy array from our list of X patterns, then reshape it into the required 3D for

```
df = DataFrame(X)
values = df.values
array = values.reshape(4, 1, 5)
```

We must also convert the list of output patterns (y) into a 2D NumPy Array.

Below is a function named *to_lstm_dataset()* that takes a sequence as an input and the size of the sequence alphabet and returns an *X* and *y* dataset ready for use with an LSTM. It performs the required conversions of the sequence to a one-hot encoding and to input-output pairs before reshaping the data.

```
# convert sequence to x/y pairs ready for use with an LSTM
def to_lstm_dataset(sequence, n_unique):
    # one hot encode
    encoded = encode(sequence, n_unique)
    # convert to in/out patterns
    X,y = to_xy_pairs(encoded)
    # convert to LSTM friendly format
    dfX, dfy = DataFrame(X), DataFrame(y)
    lstmX = dfX.values
    lstmX = lstmX.reshape(lstmX.shape[0], 1, lstmX.shape[1])
    lstmY = dfy.values
    return lstmX, lstmY
```

[redacted]

This function can be called with each sequence as follows:

```
seq1 = [3, 0, 1, 2, 3]
seq2 = [4, 0, 1, 2, 4]
n_unique = len(set(seq1 + seq2))

seq1X, seq1Y = to_lstm_dataset(seq1, n_unique)
seq2X, seq2Y = to_lstm_dataset(seq2, n_unique)
```

We now have all of the pieces to prepare the data for the LSTM.

Learn Sequences with an LSTM

In this section, we will define the LSTM to learn the input sequences.

This section is divided into 4 sections:

1. LSTM Configuration
2. LSTM Training
3. LSTM Evaluation
4. LSTM Complete Example

LSTM Configuration

We want the LSTM to make one-step predictions, which we have defined in the format and shape of our dataset. We also want the LSTM to be updated with errors after each time step, this means we will need to use a batch-size of one.

Keras LSTMs are not stateful between batches by default. We can make them stateful by setting the [redacted] managing the training epochs manually to ensure that the internal state of the LSTM is reset after each sequence.

We must define the shape of the batch using the *batch_input_shape* argument with 3 dimensions [*batch size, time steps, and features*] which will be 1, 1, and 5 respectively.

The network topology will be configured with one hidden LSTM layer with 20 units and a normal Dense layer with 5 outputs for each of the 5 columns in an output pattern. A sigmoid (logistic) activation function will be used on the output layer because of the binary outputs and the default tanh (hyperbolic tangent) activation function will be used on the LSTM layer.

A log (cross entropy) loss function will be optimized when fitting the network because of the binary outputs and the efficient ADAM optimization algorithm will be used with all default parameters.

The Keras code to define the LSTM network for this problem is listed below.

```
model = Sequential()
model.add(LSTM(20, batch_input_shape=(1, 1, 5), stateful=True))
model.add(Dense(5, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
```

LSTM Training

We must fit the model manually one epoch at a time.

Within one epoch we can fit the model on each sequence, being sure to reset state after each seque

The model does not need to be trained for long given the simplicity of the problem; in this case only

Below is an example of how the model can be fit on each sequence across all epochs.

```
# train LSTM
for i in range(250):
    model.fit(seq1X, seq1Y, epochs=1, batch_size=1, verbose=1, shuffle=False)
    model.reset_states()
    model.fit(seq2X, seq2Y, epochs=1, batch_size=1, verbose=0, shuffle=False)
    model.reset_states()
```

I like to see some feedback on the loss function when fitting a network, so verbose output is turned c

[redacted]

LSTM Evaluation

Next, we can evaluate the fit model by predicting each step of the learned sequences.

We can do this by predicting the outputs for each sequence.

The *predict_classes()* function can be used on the LSTM model that will predict the class directly. It does this by performing an *argmax()* on the output binary vector and returning the index of the predicted column with the largest output. The output indices map perfectly onto the integers used in the sequence (by careful design above). An example of making a prediction is listed below:

```
result = model.predict_classes(seq1X, batch_size=1, verbose=0)
```

We can make a prediction, then print the result in the context of the input pattern and the expected o

LSTM Complete Example

We can now tie the whole tutorial together.

The complete code listing is provided below.

First, the data is prepared, then the model is fit and the predictions of both sequences are printed.

```
from pandas import DataFrame
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# binary encode an input pattern, return a list of binary vectors
def encode(pattern, n_unique):
    encoded = list()
    for value in pattern:
        row = [0.0 for x in range(n_unique)]
        row[value] = 1.0
        encoded.append(row)
    return encoded

# create input/output pairs of encoded vectors, returns X, y
def to_xy_pairs(encoded):
    X,y = list(),list()
    for i in range(1, len(encoded)):
        X.append(encoded[i-1])
        y.append(encoded[i])
    return X, y

# convert sequence to x/y pairs ready for use with an LSTM
def to_lstm_dataset(sequence, n_unique):
    # one hot encode
    encoded = encode(sequence, n_unique)
    # convert to in/out patterns
    X,y = to_xy_pairs(encoded)
    # convert to LSTM friendly format
    dfX, dfy = DataFrame(X), DataFrame(y)
    lstmX = dfX.values
    lstmX = lstmX.reshape(lstmX.shape[0], 1, lstmX.shape[1])
    lstmY = dfy.values
    return lstmX, lstmY

# define sequences
seq1 = [3, 0, 1, 2, 3]
seq2 = [4, 0, 1, 2, 4]
# convert sequences into required data format
n_unique = len(set(seq1 + seq2))
seq1X, seq1Y = to_lstm_dataset(seq1, n_unique)
seq2X, seq2Y = to_lstm_dataset(seq2, n_unique)
# define LSTM configuration
n_neurons = 20
n_batch = 1
n_epoch = 250
n_features = n_unique
# create LSTM
model = Sequential()
model.add(LSTM(n_neurons, batch_input_shape=(n_batch, 1, n_features), stateful=True))
model.add(Dense(n_unique, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
# train LSTM
for i in range(n_epoch):
    model.fit(seq1X, seq1Y, epochs=1, batch_size=n_batch, verbose=1, shuffle=False)
    model.reset_states()
    model.fit(seq2X, seq2Y, epochs=1, batch_size=n_batch, verbose=0, shuffle=False)
    model.reset_states()

# test LSTM on sequence 1
print('Sequence 1')
result = model.predict_classes(seq1X, batch_size=n_batch, verbose=0)
model.reset_states()
for i in range(len(result)):
    print('X=%.1f y=%.1f, yhat=%.1f' % (seq1[i], seq1[i+1], result[i]))

# test LSTM on sequence 2
print('Sequence 2')
result = model.predict_classes(seq2X, batch_size=n_batch, verbose=0)
model.reset_states()
for i in range(len(result)):
    print('X=%.1f y=%.1f, yhat=%.1f' % (seq2[i], seq2[i+1], result[i]))
```

[redacted]

Running the example provides feedback regarding the model's loss on the first sequence each epoch.

At the end of the run, each sequence is printed in the context of the predictions.

```
...
4/4 [==============================] - 0s - loss: 0.0930
Epoch 1/1
4/4 [==============================] - 0s - loss: 0.0927
Epoch 1/1
4/4 [==============================] - 0s - loss: 0.0925
Sequence 1
X=3.0 y=0.0, yhat=0.0
X=0.0 y=1.0, yhat=1.0
X=1.0 y=2.0, yhat=2.0
X=2.0 y=3.0, yhat=3.0
Sequence 2
X=4.0 y=0.0, yhat=0.0
X=0.0 y=1.0, yhat=1.0
X=1.0 y=2.0, yhat=2.0
X=2.0 y=4.0, yhat=4.0
```

The results show two important things:

- That the LSTM correctly learned each sequence one step at a time.
- That the LSTM used the context of each sequence to correctly resolve the conflicting input pairs.

In essence, the LSTM was able to remember the input pattern at the beginning of the sequence 3 time steps ago to correctly predict the last value in the sequence.

This memory and ability of LSTMs to relate observations distant in time is the key capability that makes LSTMs so powerful and why they are so widely used.

Although the example is trivial, LSTMs are able to demonstrate this same capability across 100s, an

Extensions

This section lists ideas for extensions to the examples in this tutorial.

- **Tuning**. The configurations for the LSTM (epochs, units, etc.) were chosen after some trial and error. It is possible that a much simpler configuration can achieve the same result on this problem. Some search of parameters is required.
- **Arbitrary Alphabets**. The alphabet of 5 integers was chosen arbitrarily. This could be changed to other symbols and larger alphabets.
- **Long Sequences**. The sequences used in this example were very short. The LSTM is able to demonstrate the same capability on much longer sequences of 100s and 1000s of time steps.
- **Random Sequences**. The sequences used in this tutorial were linearly increasing. New sequences of random values can be created, allowing the LSTM to devise a generalized solution rather than one specialized to the two sequences used in this tutorial.
- **Batch Learning**. Updates were made to the LSTM after each time step. Explore using batch up
- **Shuffle Epoch**. The sequences were shown in the same order each epoch during training and ne sequences so that sequence 1 and 2 are fit within an epoch, which might improve the generaliz the same alphabet.

Did you explore any of these extensions?
Share your results in the comments below. I'd love to see what you came up with.

Further Reading

I strongly recommend reading the original 1997 LSTM paper by Hochreiter and Schmidhuber; it is ve

- Long Short Term Memory, 1997 [PDF]

Summary

In this tutorial, you discovered a key capability of LSTMs in their ability to remember over multiple time steps.

Specifically, you learned:

- How to define a small sequence prediction problem that only an RNN like LSTMs can solve using memory.
- How to transform the problem representation so that it is suitable for learning by LSTMs.

[redacted]

- How to design an LSTM to solve the problem correctly.

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

About Jason Brownlee

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

View all posts by Jason Brownlee →

< Multi-step Time Series Forecasting with Long Short-Term Memory Networks in Python

How to use Different Batch Sizes when Training and Predicting with LSTMs >

16 Responses to *Demonstration of Memory with a Long Short-Term Memor*

Tim May 13, 2017 at 4:00 pm #

Thank you for the tutorial! What's the point in making the LSTM stateful if you reset it after ever

Jason Brownlee May 14, 2017 at 7:26 am #

Great question,

I made the LSTM stateful so that I can control exactly when that reset occurs. You are correct in that that resets occurred at appropriate times.

Get Your Start in Machine Learning

You can master applied Machine Learning **without the math or fancy degree**.
Find out how in this *free* and *practical* email course.

Email Address

START MY EMAIL COURSE

Birkey June 3, 2017 at 2:55 pm #

REPLY

Hi Jason,

If we want num_lag = 2, that is:
x1,x2 -> y

Get Your Start in Machine Learning

————-
(3, 0) -> 1
(0, 1) -> 2
(1, 2) -> 3

the to_xy_pairs function needs to be updated to:

```
python
def to_xy_pairs(encoded):
X, y = list(), list()
for i in range(2, len(encoded)):
X.append(encoded[i - 2]) # <-- num_lag = 2
X.append(encoded[i - 1])
y.append(encoded[i])
return X, y
```

correspondingly, to_lstm_dataset() updated as:
`python
lstmX = lstmX.reshape(int(lstmX.shape[0] / 2), 2, lstmX.shape[1]) # 1.0 | 1.0
(0.0, 1.0) -> 2.0 | 2.0
(1.0, 2.0) -> 3.0 | 3.0

In preprocessing, can we just use function series_to_supervised() (post on May 8, 2017) directly?

I guess not.
For it generates data like:
– var1(t-2) var1(t-1) var1(t)
you have to frame them as a sequence:
– X: [var1(t-2), var1(t-1), …]
– y: [var1(t), …]

Correct?
so the function series_to_superviesed() 1) for demonstration, 2) produce intermediate results ?

Jason Brownlee June 4, 2017 at 7:48 am # REPLY

Perhaps, you could try it.

In this case, we are providing the sequence one-time step at a time, rather than BPTT over the whole sequence. A truly amazing demonstration of the memory capability of LSTMs.

Birkey June 5, 2017 at 7:38 pm # REPLY

Agreed: for one-time step, no BPTT, but for two-time step, LSTM does learn through BPTT.

Also, for the network topology:
Layer (type) Output Shape Param # Connected to
==
ts_input (InputLayer) (None, 2, 18) 0

when lag = 2, output shape of the input layer is 3-D tensor (time step is 2), just like learning a lar

Birkey June 3, 2017 at 3:01 pm #

(continued 🙂

to_lstm_dataset()

now proceeds:

X = [[0.0, 0.0, 0.0, 1.0, 0.0], [1.0, 0.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0]]
y = [[0.0, 1.0, 0.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0, 0.0]]
lstmX.shape = (3, 2, 5)
lstmY.shape = (3, 5)

– X: 3 batches, __each batch contains 2 time steps__, each time step input a vector of 5 dimensions
– y: 3 results, correspondingly.

So I guess:
1) for uni-variable series-to-suerpvised, to_xy_pairs() is enough.
2) for multi-variable series-to-suerpvised, series_to_supervised() + to_xy_pairs() is enough.

Danil June 5, 2017 at 4:46 pm # REPLY

Great article, thank you.

Is it beneficial to select batch_size > 1 for stateful LSTM ?
If yes , how to reshape any DataFrame (dfX) to [batch_size, 1, num_features] input suitable for stateful L

batch_size = 128
tsteps = 1

# X_train.shape = (107904, 10)
# Y_train.shape = (107904, 2)
# 107904 % 128 == 0
# X_train and Y_train are both DataFrames

model = Sequential()
model.add(LSTM(50,
input_shape=(tsteps, Y_train.shape[1]),
batch_size=batch_size,
return_sequences=True,
stateful=True))
model.add(LSTM(50,
return_sequences=False,
stateful=True))
model.add(Dense(X_train.shape[1]))
model.compile(loss='mse', optimizer='rmsprop')

I'm getting error:

ValueError: Error when checking input: expected lstm_1_input to have 3 dimensions, but got array with [redacted]

Jason Brownlee June 6, 2017 at 9:23 am # REPLY

No reshaping is needed for a batch size of 1, it is more of a matter of how to run your epochs.

Birkey June 6, 2017 at 12:26 pm # REPLY

Hi, Danil,

you need to reshape your input x to 3-D tensor as [num_samples, num_time_steps, feature_dimens

Danilo July 4, 2017 at 4:36 pm #

```
# test LSTM on sequence 1
print('Sequence 1')
result = model.predict_classes(seq1X, batch_size=n_batch, verbose=0)
model.reset_states()
for i in range(len(result)):
print('X=%.5f y=%.5f, yhat=%.5f' % (seq1[i], seq1[i+1], result[i]))
```

""is there any way to ask this network to predict the next output after result[i]?"" This only predicts the data that it already knows from training set.

thank you.

Jason Brownlee July 6, 2017 at 10:12 am # REPLY

Yes, fit the model on all the data, then call model.predict()

[redacted]

Dima October 11, 2017 at 4:18 am # REPLY

Thank you for this course ! Why doesn't it converge if n_batch = 4 or 2 ? Does it make any difference in this example ?

Jason Brownlee October 11, 2017 at 7:57 am # REPLY

In this case, we are constrained to fixed batch sizes because the model is stateful.

Dima October 11, 2017 at 10:26 am #

Hi Jason, thanks for your respond
If we set n_batch = 4, it will not converge and result in either 0123 for both sequences or 0124.
It behaves as if it doesn't keep state …
I see the only diiference that if batch size =1 it makes weight update 4 times passing through the
batch =4 it makes weight update just once , then reseting state and going to the second sequen

I aslo tried to concatenate both sequences so I could run 1 sequnce of 8 pairs and still I get the
batch size ==1 (i.e. 01230124), otherwise if I set batch size = 4 or 8 it results in either 01230123

What am I missing here ?

I also tried other examples in your course , one of them "Understanding Stateful LSTM Recurren
learned the alphabet with success when increasing the batch size from 1 to the size of the training dataset batch_size=len(dataX) =26.

Dima October 11, 2017 at 10:52 pm # REPLY

It would be really helpful to see this example in stateless mode .
What would the unput shape look like ?

[redacted]

Dima October 19, 2017 at 1:05 am #

REPLY

Here is a very simple example of LSTM in stateless mode and we train it on a very simple sequence [0–>1] and [0–>2] . Any idea why it won't converge in stateless mode?

We have a batch of size 2 with 2 samples and it supposed to keep the state inside the batch.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
import numpy
# define sequences
seq = [0, 1, 0, 2]
# convert sequences into required data format
seqX=numpy.array([[( 1. , 0. , 0.)], [( 1. , 0. , 0.)]])
seqY=numpy.array([( 0. , 1. , 0.) , ( 0. , 0. , 1.)])

# define LSTM configuration
n_unique = len(set(seq))
n_neurons = 20
n_batch = 2
n_epoch = 300
n_features = n_unique
# create LSTM
model = Sequential()
model.add(LSTM(n_neurons, input_shape=( 1, n_features) ))
model.add(Dense(n_unique, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='Adam')
# train LSTM
model.fit(seqX, seqY, epochs=n_epoch, batch_size=n_batch, verbose=2, shuffle=False)

print('Sequence')
result = model.predict_classes(seqX, batch_size=n_batch, verbose=0)
```

[redacted]

for i in range(2):
print('X=%.1f y=%.1f, yhat=%.1f' % (0, i+1, result[i]))

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

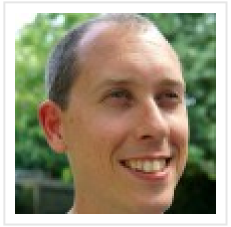
SUBMIT COMMENT

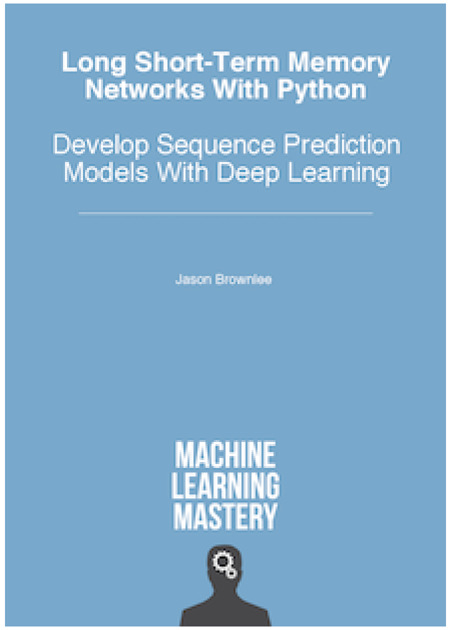
Welcome to Machine Learning Mastery

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

Read More

[redacted]




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

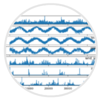
Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras
JULY 26, 2016

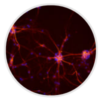
How to Setup a Python Environment for Machine Learning and Deep Learning with Anaconda
MARCH 13, 2017

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