Erik Hallström Follow
Studied Engineering Physics and in Machine Learning at Royal Institute of Technology in Stockholm. Also…
Nov 19, 2016 · 3 min read

Using the Multilayered LSTM API in TensorFlow (4/7)

In the previous article we learned how to use the TensorFlow API to create a Recurrent neural network with Long short-term memory. In this post we will make that architecture deep, introducing a LSTM with multiple layers.

One thing to notice is that for every layer of the network we will need a hidden state and a cell state. Typically the input to the next LSTM-layer will be the previous state for that particular layer as well as the hidden activations of the "lower" or previous layer. There is a good diagram in this article.

We could continue to store the states for each layer in many `LSTMTuples`, but that would require a lot of overhead. You can only input data to the placeholders trough the `feed_dict` as Python lists or Numpy arrays anyways (not as `LSTMTuples`) so we still would have to convert between the datatypes. Why not save the whole state for the network in a big tensor? In order to do this the first thing we want to do is to replace `_current_cell_state` and `_current_hidden_state` on line 81–82 with the more generic:

```
_current_state = np.zeros((num_layers, 2, batch_size, s

```

You also have to declare the new setting `num_layers = 3` in the beginning of the file, but you may choose any number of layers. The "2" refers to the two states, cell- and hidden-state. So for each layer and each sample in a batch, we have both a cell state and a hidden state vector with the size `state_size`.

Now modify lines 93 to 103 (the run function and the separation of the state tuple) back to the original statement, since the state is now stored in a single tensor.

```
_total_loss, _train_step, _current_state, _predictions_
    [total_loss, train_step, current_state, predictions
    feed_dict={
        batchX_placeholder: batchX,
        batchY_placeholder: batchY,
        init_state: _current_state
```

You can change these lines 28 to 30 in the previous post:

```
cell_state = tf.placeholder(tf.float32, [batch_size, st
hidden_state = tf.placeholder(tf.float32, [batch_size,
init_state = tf.nn.rnn_cell.LSTMStateTuple(cell_state,
```

To a single placeholder containing the whole state.

```
init_state = tf.placeholder(tf.float32, [num_layers, 2,

```

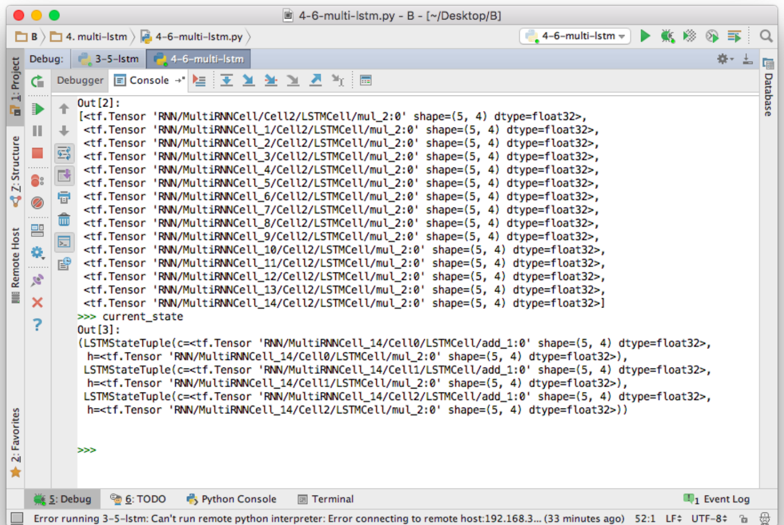
Since the TensorFlow Multilayer-LSTM-API accepts the state as a tuple of LSTMTuples, we need to unpack the state state into this structure. For each layer in the state we then create a `LSTMTuple` stated, and put these in a tuple, as shown below. Add this just after the `init_state` placeholder.

```
state_per_layer_list = tf.unpack(init_state, axis=0)
rnn_tuple_state = tuple(
    [tf.nn.rnn_cell.LSTMStateTuple(state_per_layer_list
     for idx in range(num_layers)]
)
```

The forward pass on lines 40 and 41 should be changed to this:

```
# Forward passes
cell = tf.nn.rnn_cell.LSTMCell(state_size, state_is_tup
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layers,
states_series, current_state = tf.nn.rnn(cell, inputs_s
```

The multi-layered LSTM is created by first making a single `LSMTCell`, and then duplicating this cell in an array, supplying it to the `MultiRNNCell` API call. The forward pass uses the usual `tf.nn.rnn`, let's print the output of this function, the `states_series` and `current_state` variables.



Output of the previous states and the last LSTMStateTuples

Take a look at the tensor names between single quotes, we see that the RNN is unrolled 15 times. In the `states_series` all outputs have the name "Cell2", it means that we get the output of the last LSTM layer's hidden state in the list. Furthermore the LSTMStateTuple in the `current_state` gives the whole state of all layers in the network. "Cell0" refers to the first layer, "Cell1" to the second and "Cell2" to the third and final layer, "h" and "c" refers to hidden- and cell state.

Whole program

This is the whole self-contained script, just copy and run.

```
from __future__ import print_function, division
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

num_epochs = 100
total_series_length = 50000
truncated_backprop_length = 15
state_size = 4
num_classes = 2
echo_step = 3
batch_size = 5
num_batches = total_series_length//batch_size//trunca
num_layers = 3

def generateData():
    x = np.array(np.random.choice(2, total_series_len
    y = np.roll(x, echo_step)
    y[0:echo_step] = 0

    x = x.reshape((batch_size, -1))  # The first inde
    y = y.reshape((batch_size, -1))

    return (x, y)

batchX_placeholder = tf.placeholder(tf.float32, [batc
batchY_placeholder = tf.placeholder(tf.int32, [batch_

init_state = tf.placeholder(tf.float32, [num_layers,

state_per_layer_list = tf.unpack(init_state, axis=0)
rnn_tuple_state = tuple(
    [tf.nn.rnn_cell.LSTMStateTuple(state_per_layer_li
     for idx in range(num_layers)]
)

W2 = tf.Variable(np.random.rand(state_size, num_class
b2 = tf.Variable(np.zeros((1,num_classes)), dtype=tf.

# Unpack columns
inputs_series = tf.split(1, truncated_backprop_length
```

```
labels_series = tf.unpack(batchY_placeholder, axis=1)

# Forward passes
cell = tf.nn.rnn_cell.LSTMCell(state_size, state_is_t
cell = tf.nn.rnn_cell.MultiRNNCell([cell] * num_layer
states_series, current_state = tf.nn.rnn(cell, inputs

logits_series = [tf.matmul(state, W2) + b2 for state
predictions_series = [tf.nn.softmax(logits) for logit

losses = [tf.nn.sparse_softmax_cross_entropy_with_log
total_loss = tf.reduce_mean(losses)

train_step = tf.train.AdagradOptimizer(0.3).minimize(

def plot(loss_list, predictions_series, batchX, batch
    plt.subplot(2, 3, 1)
    plt.cla()
    plt.plot(loss_list)

    for batch_series_idx in range(5):
        one_hot_output_series = np.array(predictions_
        single_output_series = np.array([(1 if out[0]

        plt.subplot(2, 3, batch_series_idx + 2)
        plt.cla()
        plt.axis([0, truncated_backprop_length, 0, 2]
        left_offset = range(truncated_backprop_length
        plt.bar(left_offset, batchX[batch_series_idx,
        plt.bar(left_offset, batchY[batch_series_idx,
        plt.bar(left_offset, single_output_series * 0
```

Next step

In the next article we will speed up the graph creation by not splitting up our inputs and labels into a Python list.