Erik Hallström Follow
Studied Engineering Physics and in Machine Learning at Royal Institute of Technology in Stockholm. Also…
Nov 18, 2016 · 2 min read

Using the LSTM API in TensorFlow (3/7)

In the previous post we modified our to code to use the TensorFlow native RNN API. Now we will go about to build a modification of a RNN that called a "Recurrent Neural Network with Long short-term memory" or RNN-LSTM. This architecture was pioneered by Jürgen Schmidhuber among others. One problem with the RNN when using long time-dependencies (`truncated_backprop_length` is large) is the "vanishing gradient problem". One way to counter this is using a state that is "protected" and "selective". The RNN-LSTM remembers, forgets and chooses what to pass on and output depending on the current state and input.

Since this primarily is a practical tutorial I won't go into more detail about the theory, I recommend reading this article again, continue with the "Modern RNN architectures". After you have done that read and look at the figures on this page. Notice that the last mentioned resource are using vector concatenation in their calculations.

In the previous article we didn't have to allocate the internal weight matrix and bias, that was done by TensorFlow automatically "under the hood". A LSTM RNN has many more "moving parts", but by using the native API it will also be very simple.

Different state

A LSTM have a "cell state" and a "hidden state", to account for this you need to remove `_current_state` on line 79 in the previous script and replace it with this:

```
_current_cell_state = np.zeros((batch_size, state_size)
_current_hidden_state = np.zeros((batch_size, state_siz

```

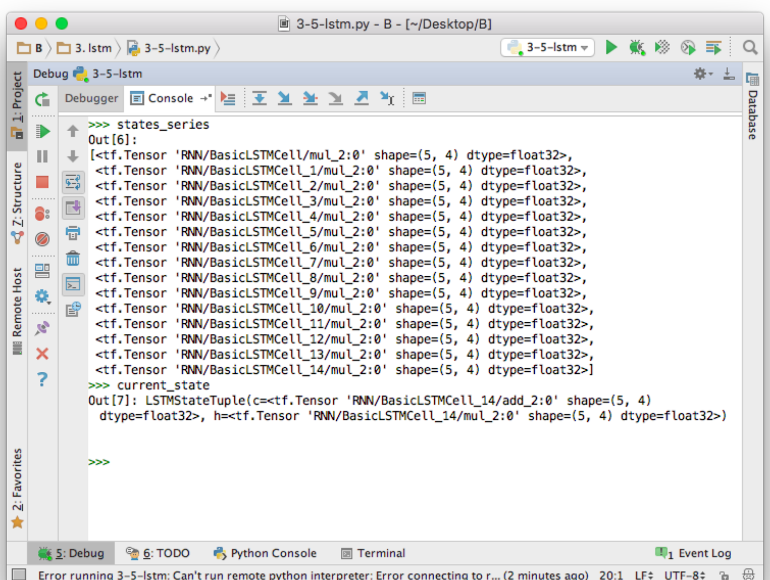
TensorFlow uses a data structure called `LSTMStateTuple` internally for its LSTM:s, where the first element in the tuple is the cell state, and the second is the hidden state. So you need to change line 28 where the `init_state` is placeholders are declared to these lines:

```
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,
```

Changing the forward pass is now straight forward, you just change the function call to create a LSTM and supply the initial state-tuple on line 38–39.

```
cell = tf.nn.rnn_cell.BasicLSTMCell(state_size, state_i
states_series, current_state = tf.nn.rnn(cell, inputs_s
```

The `states_series` will be a list of *hidden states* as tensors, and `current_state` will be a LSTMStateTuple which shows both the *hidden*- and the *cell state* on the last time-step as shown below:



Outputs of the previous states and the last LSTMStateTuple

So the `current_state` returns the cell- and hidden state in a tuple. They should be separated after calculation and supplied to the placeholders in the run-function on line 90.

```
_total_loss, _train_step, _current_state, _predictions
    [total_loss, train_step, current_state, prediction
    feed_dict={
        batchX_placeholder: batchX,
        batchY_placeholder: batchY,
        cell_state: _current_cell_state,
        hidden_state: _current_hidden_state

    })
```

Whole program

This is the full code for creating a RNN with Long short-term memory.

```
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

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_

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

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.BasicLSTMCell(state_size, state
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

    plt.draw()
    plt.pause(0.0001)
```

Next step

In the next article we will create a multi-layered or “deep” recurrent neural network, also with long short-term memory.