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

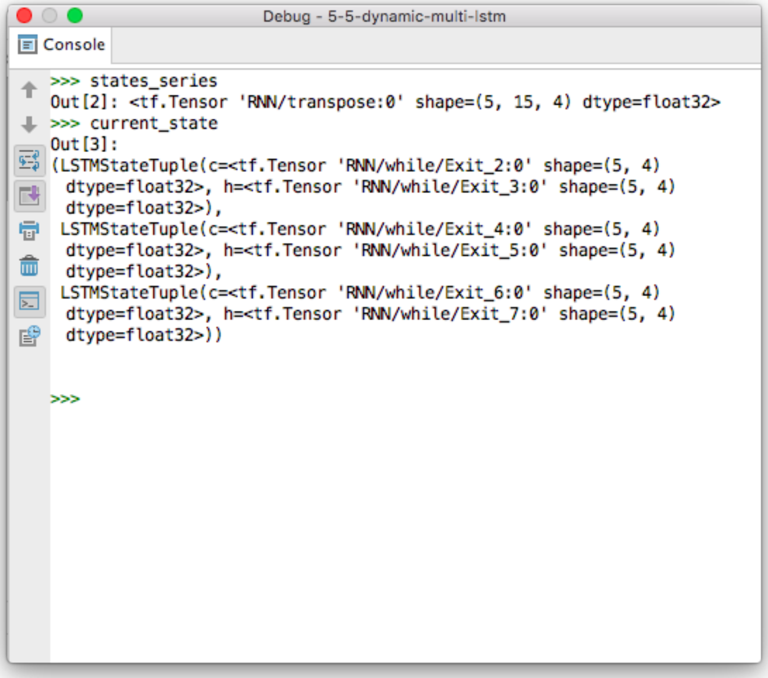
Using the DynamicRNN API in TensorFlow (5/7)

In the previous guide we built a multi-layered LSTM RNN. In this post we will speed it up by not splitting up our inputs and labels into a list, as done on line 41–42 in our code. You may remove these rows where `inputs_series` and `labels_series` are declared. Next change the `tf.nn.rnn` call on line 47 to the following:

```
states_series, current_state = tf.nn.dynamic_rnn(cell,
states_series = tf.reshape(states_series, [-1, state_si

```

The `dynamic_rnn` function takes the batch inputs of shape `[batch_size, truncated_backprop_length, input_size]`, thus the addition of a single dimension on the end. Output will be the last state of every layer in the network as an LSTMStateTuple stored in `current_state` as well as a tensor `states_series` with the shape `[batch_size, truncated_backprop_length, state_size]` containing the hidden state of the last layer across all time-steps.



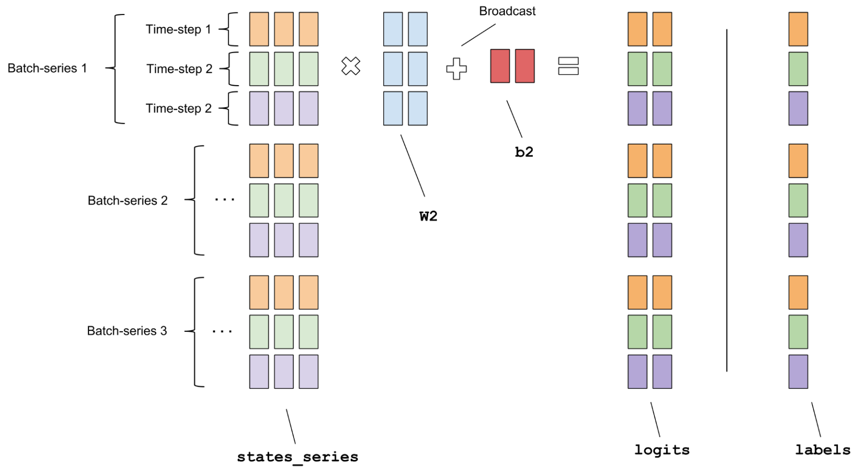
The states are not in lists anymore.

The tensor `states_series` is reshaped on the second row in the code sample above to shape `[batch_size*truncated_backprop_length, state_size]` , we will see the reason for this shortly. You may read more about `dynamic_rnn` in the documentation.

Now input this two lines below the reshaping of the `states_series` .

```
logits = tf.matmul(states_series, W2) + b2 #Broadcasted
labels = tf.reshape(batchY_placeholder, [-1])
```

Notice that we are now only working with tensors, Python lists were a thing of the past. The calculation of the `logits` and the `labels` are visualized below, notice the `state_series` variable that was reshaped earlier. In TensorFlow reshaping is done in C-like index order. It means that we read from the source tensor and “write” to the destination tensor with the last axis index changing fastest, and the first axis index changing slowest. The result of the reshaping will be as visualized in the figure below, where similar colors denote the same time-step, and the vertical grouped spacing of elements denote different batches.



Visualization of the calculations, similar color denote same time-step, vertical spacing denote new batch.

Let’s go trough all the tensors in the figure above, first let’s start with the sizes. We have that `batch_size=3` , `state_size=3` , `num_classes=2` and `truncated_backprop_length=3` . The tensor `states_series` have shape `[batch_size*truncated_backprop_length, state_size]` , `labels` have shape `[batch_size*truncated_backprop_length]` , `logits` have shape `[batch_size*truncated_backprop_length, num_classes]` , `W2` have shape `[state_size, num_classes]` and `b2` have shape `[1, num_classes]` . It can be a bit tricky to keep track of all the tensors, but drawing and visualizing with colors definitely helps.

Next calculate the predictions for the visualization:

```
logits_series = tf.unpack(tf.reshape(logits, [batch_siz
predictions_series = [tf.nn.softmax(logit) for logit in

```

Here we actually split the tensors into lists again. This is perhaps not the best way to do it, but it’s quick and dirty, and the plot function is already expecting a list.

The `sparse_softmax_cross_entropy_with_logits` can take the shape of our tensors! Modify the `losses` calculation to this.

```
losses = tf.nn.sparse_softmax_cross_entropy_with_logits

```

As we can read in the API the `logits` must have the shape `[batch_size, num_classes]` and `labels` must have the shape `[batch_size]`. But now we are treating all time-steps as elements in our batch, so it will work out as we want.

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.

# 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.dynamic_rnn(cell
states_series = tf.reshape(states_series, [-1, state_

logits = tf.matmul(states_series, W2) + b2 #Broadcast
labels = tf.reshape(batchY_placeholder, [-1])

logits_series = tf.unpack(tf.reshape(logits, [batch_s
predictions_series = [tf.nn.softmax(logit) for logit


losses = tf.nn.sparse_softmax_cross_entropy_with_logi
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 part we will regularize the network to use dropout, making it less prone to overfitting.