Erik Hallström Follow
Studied Engineering Physics and in Machine Learning at Royal Institute of Technology in Stockholm. Also…
Nov 10, 2016 · 7 min read

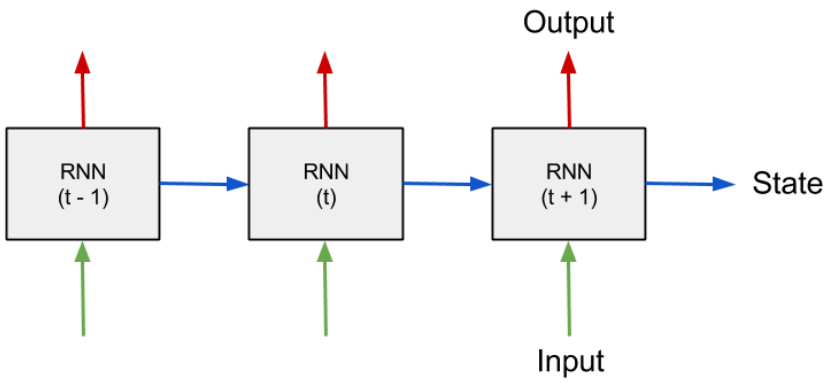
How to build a Recurrent Neural Network in TensorFlow (1/7)

In this tutorial I'll explain how to build a simple working Recurrent Neural Network in TensorFlow. This is the first in a series of seven parts where various aspects and techniques of building Recurrent Neural Networks in TensorFlow are covered. A short introduction to TensorFlow is available here. For now, let's get started with the RNN!

What is a RNN?

It is short for "Recurrent Neural Network", and is basically a neural network that can be used when your data is treated as a sequence, where the particular order of the data-points matter. More importantly, this sequence can be of *arbitrary length*.

The most straight-forward example is perhaps a time-series of numbers, where the task is to predict the next value given previous values. The input to the RNN at every time-step is the *current value* as well as a *state vector* which represent what the network has "seen" at time-steps before. This state-vector is the encoded memory of the RNN, initially set to zero.



Schematic of a RNN processing sequential data over time.

The best and most comprehensive article explaining RNN:s I've found so far is this article by researchers at UCSD, highly recommended. For now you only need to understand the basics, read it until the "Modern RNN architectures"-section. That will be covered later.

Although this article contains some explanations, it is mostly focused on the practical part, how to build it. You are encouraged to look up more theory on the Internet, there are plenty of good explanations.

Setup

We will build a simple Echo-RNN that remembers the input data and then echoes it after a few time-steps. First let's set some constants we'll need, what they mean will become clear in a moment.

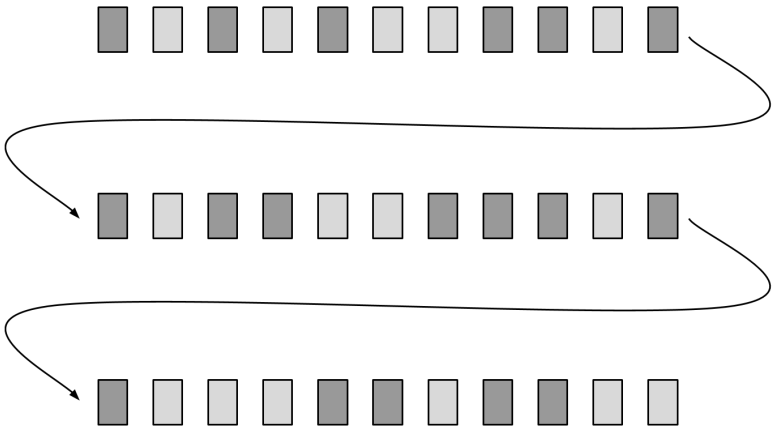
```
from __future__ import print_function, division
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

num_epochs = 100
total_series_length = 50000
truncated_backprop_length = 15
state_size = 4
num_classes = 2
```

Generate data

Now generate the training data, the input is basically a random binary vector. The output will be the "echo" of the input, shifted `echo_step` steps to the right.

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

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

Notice the reshaping of the data into a matrix with `batch_size` rows. Neural networks are trained by approximating the gradient of loss function with respect to the neuron-weights, by looking at only a small subset of the data, also known as a *mini-batch*. The theoretical reason for doing this is further elaborated in this question. The reshaping takes the whole dataset and puts it into a matrix, that later will be sliced up into these mini-batches.



Schematic of the reshaped data-matrix, arrow curves shows adjacent time-steps that ended up on different rows. Light-gray rectangle represent a “zero” and dark-gray a “one”.

Building the computational graph

TensorFlow works by first building up a computational graph, that specifies what operations will be done. The input and output of this graph is typically multidimensional arrays, also known as tensors. The graph, or parts of it can then be executed iteratively in a session, this can either be done on the CPU, GPU or even a resource on a remote server.

Variables and placeholders

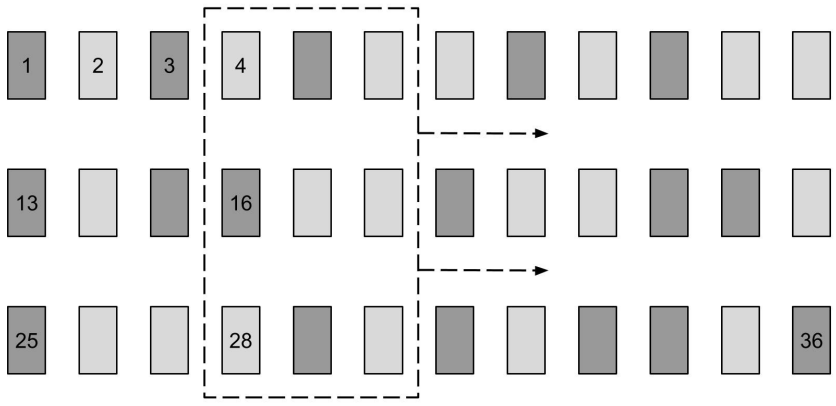
The two basic TensorFlow data-structures that will be used in this example are *placeholders* and *variables*. On each run the batch data is fed to the placeholders, which are “starting nodes” of the computational graph. Also the RNN-state is supplied in a placeholder, which is saved from the output of the previous run.

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

init_state = tf.placeholder(tf.float32, [batch_size, st
```

The weights and biases of the network are declared as TensorFlow *variables*, which makes them persistent across runs and enables them to be updated incrementally for each batch.

```
W = tf.Variable(np.random.rand(state_size+1, state_size
b = tf.Variable(np.zeros((1,state_size)), dtype=tf.floa

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

The figure below shows the input data-matrix, and the current batch `batchX_placeholder` is in the dashed rectangle. As we will see later, this "batch window" is slided `truncated_backprop_length` steps to the right at each run, hence the arrow. In our example below `batch_size = 3`, `truncated_backprop_length = 3`, and `total_series_length = 36`. Note that these numbers are just for visualization purposes, the values are different in the code. The series order index is shown as numbers in a few of the data-points.



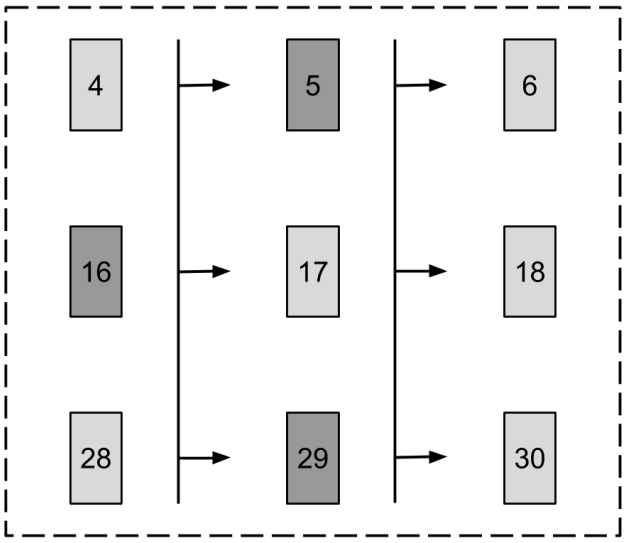
Schematic of the training data, the current batch is sliced out in the dashed rectangle. The time-step index of the datapoint is displayed.

Unpacking

Now it's time to build the part of the graph that resembles the actual RNN computation, first we want to split the batch data into adjacent time-steps.

```
# Unpack columns
inputs_series = tf.unpack(batchX_placeholder, axis=1)
labels_series = tf.unpack(batchY_placeholder, axis=1)
```

As you can see in the picture below that is done by unpacking the columns (`axis = 1`) of the batch into a Python list. The RNN will simultaneously be training on different parts in the time-series; steps 4 to 6, 16 to 18 and 28 to 30 in the current batch-example. The reason for using the variable names `"plural"_"series"` is to emphasize that the variable is a list that represent a time-series with multiple entries at each step.



Schematic of the current batch split into columns, the order index is shown on each data-point and arrows show adjacent time-steps.

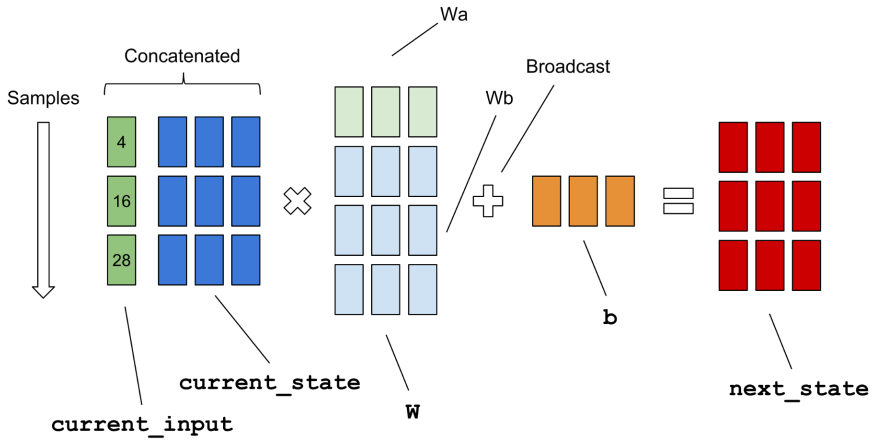
The fact that the training is done on three places simultaneously in our time-series, requires us to save three instances of states when propagating forward. That has already been accounted for, as you see that the `init_state` placeholder has `batch_size` rows.

Forward pass

Next let's build the part of the graph that does the actual RNN computation.

```
1  # Forward pass
2  current_state = init_state
3  states_series = []
4  for current_input in inputs_series:
5      current_input = tf.reshape(current_input, [batch_s
6      input_and_state_concatenated = tf.concat(1, [curre
7
8      next_state = tf.tanh(tf.matmul(input_and_state_con
```

Notice the concatenation on line 6, what we actually want to do is calculate the sum of two affine transforms `current_input * Wa + current_state * Wb` in the figure below. By concatenating those two tensors you will only use one matrix multiplication. The addition of the bias `b` is broadcasted on all samples in the batch.



Schematic of the computations of the matrices on line 8 in the code example above, the non-linear transform arctan is omitted.

You may wonder the variable name `truncated_backprop_length` is supposed to mean. When a RNN is trained, it is actually treated as a deep neural network with reoccurring weights in every layer. These layers will not be unrolled to the beginning of time, that would be too computationally expensive, and are therefore truncated at a limited

number of time-steps. In our sample schematics above, the error is backpropagated three steps in our batch.

Calculating loss

This is the final part of the graph, a fully connected softmax layer from the state to the output that will make the classes *one-hot encoded*, and then calculating the loss of the batch.

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

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

```

The last line is adding the training functionality, TensorFlow will perform back-propagation for us automatically—the computation graph is executed once for each mini-batch and the network-weights are updated incrementally.

Notice the API call to `sparse_softmax_cross_entropy_with_logits`, it automatically calculates the softmax internally and then computes the cross-entropy. In our example the classes are mutually exclusive (they are either zero or one), which is the reason for using the "Sparse-softmax", you can read more about it in the API. The usage is to have `logits` is of shape `[batch_size, num_classes]` and `labels` of shape `[batch_size]`.

Visualizing the training

There is a visualization function so we can se what's going on in the network as we train. It will plot the loss over the time, show training input, training output and the current predictions by the network on different sample series in a training batch.

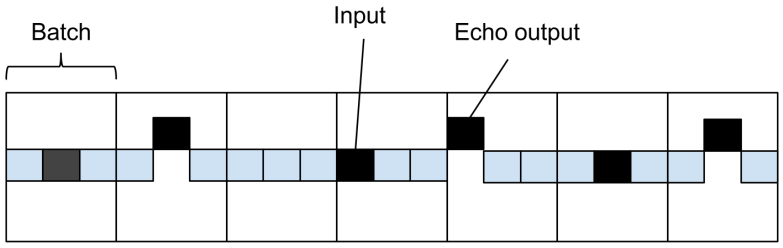
```
def plot(loss_list, predictions_series, batchX, batchY
    plt.subplot(2, 3, 1)
    plt.cla()
    plt.plot(loss_list)

    for batch_series_idx in range(5):
        one_hot_output_series = np.array(predictions_s
        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)
```

Running a training session

It's time to wrap up and train the network, in TensorFlow the graph is executed in a session. New data is generated on each epoch (not the usual way to do it, but it works in this case since everything is predictable).

```
1  with tf.Session() as sess:
2      sess.run(tf.initialize_all_variables())
3      plt.ion()
4      plt.figure()
5      plt.show()
6      loss_list = []
7
8      for epoch_idx in range(num_epochs):
9          x,y = generateData()
10         _current_state = np.zeros((batch_size, state_s
11
12         print("New data, epoch", epoch_idx)
13
14         for batch_idx in range(num_batches):
15             start_idx = batch_idx * truncated_backprop
16             end_idx = start_idx + truncated_backprop_l
17
18             batchX = x[:,start_idx:end_idx]
19             batchY = y[:,start_idx:end_idx]
20
21             _total_loss, _train_step, _current_state,
22                 [total_loss, train_step, current_state
23                 feed_dict={
```

You can see that we are moving `truncated_backprop_length` steps forward on each iteration (line 15–19), but it is possible have different strides. This subject is further elaborated in this article. The downside with doing this is that `truncated_backprop_length` need to be significantly larger than the time dependencies (three steps in our case) in order to encapsulate the relevant training data. Otherwise there might a lot of “misses”, as you can see on the figure below.

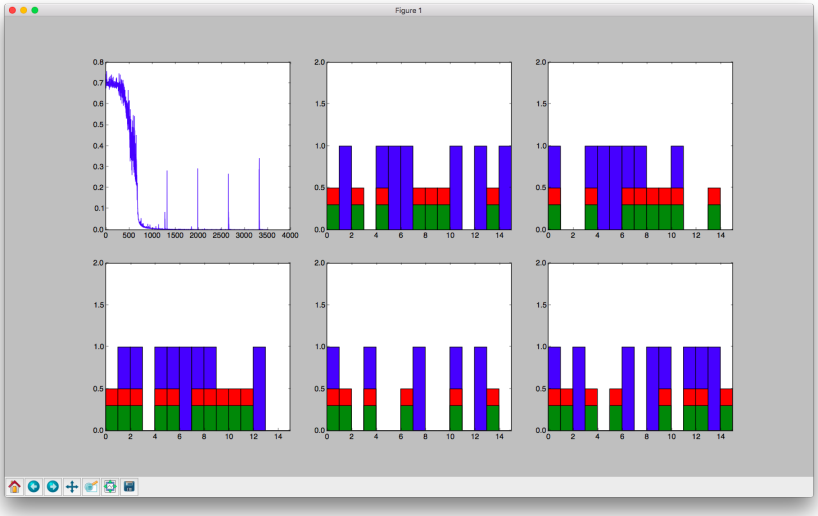


Time series of squares, the elevated black square symbolizes an echo-output, which is activated three steps from the echo input (black square). The sliding batch window is also striding three steps at each run, which in our sample case means that no batch will encapsulate the dependency, so it can not train.

Also realize that this is just simple example to explain how a RNN works, this functionality could easily be programmed in just a few lines of code. The network will be able to exactly learn the echo behavior so there is no need for testing data.

The program will update the plot as training progresses, shown in the picture below. Blue bars denote a training input signal (binary one), red bars show echos in the training output and green bars are the echos the net is generating. The different bar plots show different sample series in the current batch.

Our algorithm will fairly quickly learn the task. The graph in the top-left corner shows the output of the loss function, but why are there spikes in the curve? Think of it for a moment, answer is below.



Visualization of the loss, input and output training data (blue, red) as well as the prediction (green).

The reason for the spikes is that we are starting on a new epoch, and generating new data. Since the matrix is reshaped, the first element on each row is adjacent to the last element in the previous row. The first few elements on all rows (except the first) have dependencies that will not be included in the state, so the net will always perform badly on the first batch.

Whole program

This is the whole runnable program, just copy-paste and run. After each part in the article series the whole runnable program will be presented. If a line is referenced by number, these are the line numbers that we mean.

```
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_

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

W = tf.Variable(np.random.rand(state_size+1, state_si
b = tf.Variable(np.zeros((1,state_size)), dtype=tf.fl

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

# Forward pass
current_state = init_state
```

```
42
43
44
45
46
47
48      s
49      cur
50
51  logits_seri
52  predictions_s
53
54  losses = [tf.nn.s
55  total_loss = tf.red
56
57  train_step = tf.train.A
58
59  def plot(loss_list, predict
60      plt.subplot(2, 3, 1)
61      plt.cla()
62      plt.plot(loss_list)
63
64      for batch_series_idx in range(5):
65          one_hot_output_series = np.arra
66          single_output_series = np.array([
67
68          plt.subplot(2, 3, batch_series_idx + 2
69          plt.cla()
70          plt.axis([0, truncated_backprop_length, 0,
71          left_offset = range(truncated_backprop_leng
```

Next step

In the next post in this series we will be simplify the computational graph creation by using the native TensorFlow RNN API.