DeepSpeech

February 20, 2019

1 Introduction

In this notebook we will reproduce the results of Deep Speech: Scaling up end-to-end speech recognition. The core of the system is a bidirectional recurrent neural network (BRNN) trained to ingest speech spectrograms and generate English text transcriptions.

Let a single utterance x and label y be sampled from a training set $S = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ...\}$. Each utterance, $x^{(i)}$ is a time-series of length $T^{(i)}$ where every time-slice is a vector of audio features, $x_t^{(i)}$ where $t = 1, \ldots, T^{(i)}$. We use spectrograms as our features; so $x_{t,p}^{(i)}$ denotes the power of the p-th frequency bin in the audio frame at time t. The goal of our BRNN is to convert an input sequence x into a sequence of character probabilities for the transcription y, with $\hat{y}_t = \mathbb{P}(c_t \mid x)$, where $c_t \in \{a, b, c, ..., z, space, apostrophe, blank\}$. (The significance of *blank* will be explained below.)

Our BRNN model is composed of 5 layers of hidden units. For an input x, the hidden units at layer l are denoted $h^{(l)}$ with the convention that $h^{(0)}$ is the input. The first three layers are not recurrent. For the first layer, at each time t, the output depends on the spectrogram frame x_t along with a context of C frames on each side. (We typically use $C \in \{5, 7, 9\}$ for our experiments.) The remaining non-recurrent layers operate on independent data for each time step. Thus, for each time t, the first 3 layers are computed by:

$$h_t^{(l)} = g(W^{(l)} h_t^{(l-1)} + b^{(l)})$$

where $g(z) = \min\{\max\{0, z\}, 20\}$ is the clipped rectified-linear (ReLu) activation function and $W^{(l)}$, $b^{(l)}$ are the weight matrix and bias parameters for layer l. The fourth layer is a bidirectional recurrent layer[1]. This layer includes two sets of hidden units: a set with forward recurrence, $h^{(f)}$, and a set with backward recurrence $h^{(b)}$:

$$h_t^{(f)} = g(W^{(4)} h_t^{(3)} + W_r^{(f)} h_{t-1}^{(f)} + b^{(4)})$$
$$h_t^{(b)} = g(W^{(4)} h_t^{(3)} + W_r^{(b)} h_{t+1}^{(b)} + b^{(4)})$$

Note that $h^{(f)}$ must be computed sequentially from $t = 1$ to $t = T^{(i)}$ for the i-th utterance, while the units $h^{(b)}$ must be computed sequentially in reverse from $t = T^{(i)}$ to $t = 1$.

The fifth (non-recurrent) layer takes both the forward and backward units as inputs

$$h^{(5)} = g(W^{(5)} h^{(4)} + b^{(5)})$$

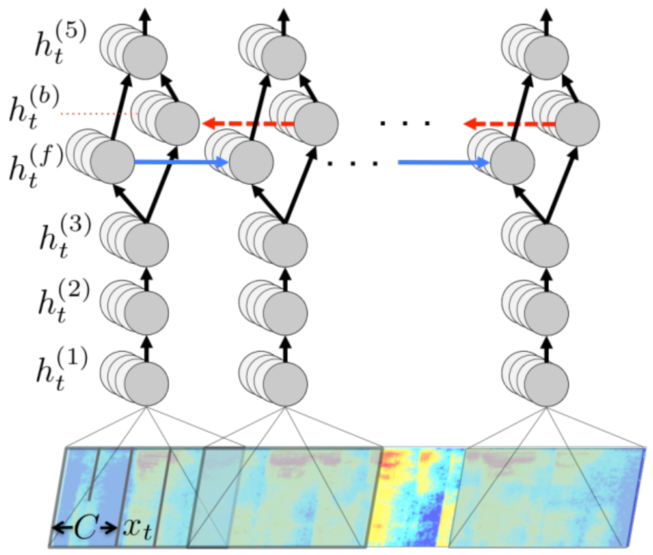
where $h^{(4)} = h^{(f)} + h^{(b)}$. The output layer is a standard softmax function that yields the predicted character probabilities for each time slice t and character k in the alphabet:

$$h_{t,k}^{(6)} = \hat{y}_{t,k} \equiv \mathbb{P}(c_t = k \mid x) = \frac{\exp\left((W^{(6)}h_t^{(5)})_k + b_k^{(6)}\right)}{\sum_j \exp\left((W^{(6)}h_t^{(5)})_j + b_j^{(6)}\right)}$$

Here $b_k^{(6)}$ denotes the k-th bias and $(W^{(6)}h_t^{(5)})_k$ the k-th element of the matrix product.

Once we have computed a prediction for $\mathbb{P}(c_t = k \mid x)$, we compute the CTC loss[2] $\mathcal{L}(\hat{\dagger}, \dagger)$ to measure the error in prediction. During training, we can evaluate the gradient $\nabla\mathcal{L}(\hat{\dagger}, \dagger)$ with respect to the network outputs given the ground-truth character sequence y. From this point, computing the gradient with respect to all of the model parameters may be done via back-propagation through the rest of the network. We use the Adam method for training[3].

The complete BRNN model is illustrated in the figure below.



DeepSpeech BRNN

2 Data Import

The import routines for the TED-LIUM have yet to be written.

```
In [1]: #from ted_lium import input_data
        #ted_lium = input_data.read_data_sets("./TEDLIUM_release2")
```

3 Preliminaries

3.1 Imports

Here we first import all of the packages we require to implement the DeepSpeech BRNN.

```
In [2]: import tensorflow as tf
        from tensorflow.python.framework.constant_op import constant
        import numpy as np
```

3.2 Global Constants

Next we introduce several constants used in the algorithm below. In particular, we define * learning_rate - The learning rate we will employ in Adam optimizer[3] * training_iters- The number of iterations we will train for * batch_size- The number of elements in a batch * display_step- The number of iterations we cycle through before displaying progress

```
In [3]: learning_rate = 0.001   # TODO: Determine a reasonable value for this
        training_iters = 100000 # TODO: Determine a reasonable value for this
        batch_size = 128        # TODO: Determine a reasonable value for this
        display_step = 10       # TODO: Determine a reasonable value for this
```

Note that we use the Adam optimizer[3] instead of Nesterov's Accelerated Gradient [4] used in the original DeepSpeech paper, as, at the time of writing, TensorFlow does not have an implementation of Nesterov's Accelerated Gradient [4].

As we will also employ dropout on the feedforward layers of the network, we need to define a parameter dropout_rate that keeps track of the dropout rate for these layers

```
In [4]: dropout_rate = 0.05
```

One more constant required of the non-recurrant layers is the clipping value of the ReLU. We capture that in the value of the variable relu_clip

```
In [5]: relu_clip = 20 # TODO: Validate this is a reasonable value
```

3.3 Geometric Constants

Now we will introduce several constants related to the geometry of the network.

The network views each speech sample as a sequence of time-slices $x_t^{(i)}$ of length $T^{(i)}$. As the speech samples vary in length, we know that $T^{(i)}$ need not equal $T^{(j)}$ for $i \neq j$. However, BRNN in TensorFlow are unable to deal with sequences with differing lengths. Thus, we must pad speech sample sequences with trailing zeros such that they are all of the same length. This common padded length is captured in the variable n_steps

```
In [6]: n_steps = 500 # TODO: Determine this programatically from the longest speech sample
```

Each of the n_steps vectors is the Fourier transform of a time-slice of the speech sample. The number of "bins" of this Fourier transform is dependent upon the sample rate of the data set. Generically, if the sample rate is 8kHz we use 80bins. If the sample rate is 16kHz we use 160bins... We capture the dimension of these vectors, equivalently the number of bins in the Fourier transform, in the variable n_input

```
In [7]: n_input = 160 # TODO: Determine this programatically from the sample rate
```

As previously mentioned, the BRNN is not simply fed the Fourier transform of a given time-slice. It is fed, in addition, a context of $C \in \{5, 7, 9\}$ frames on either side of the frame in question. The number of frames in this context is captured in the variable n_context

```
In [8]: n_context = 5 # TODO: Determine the optimal value using a validation data set
```

Next we will introduce constants that specify the geometry of some of the non-recurrent layers of the network. We do this by simply specifying the number of units in each of the layers

```
In [9]: n_hidden_1 = n_input + 2*n_input*n_context # Note: This value was not specified in the o
        n_hidden_2 = n_input + 2*n_input*n_context # Note: This value was not specified in the o
        n_hidden_5 = n_input + 2*n_input*n_context # Note: This value was not specified in the o
```

where `n_hidden_1` is the number of units in the first layer, `n_hidden_2` the number of units in the second, and `n_hidden_5` the number in the fifth. We haven't forgotten about the third or sixth layer. We will define their unit count below.

A LSTM BRNN consists of a pair of LSTM RNN's. One LSTM RNN that works "forward in time"

and a second LSTM RNN that works "backwards in time"

The dimension of the cell state, the upper line connecting subsequent LSTM units, is independent of the input dimension and the same for both the forward and backward LSTM RNN.

Hence, we are free to choose the dimension of this cell state independent of the input dimension. We capture the cell state dimension in the variable `n_cell_dim`.

```
In [10]: n_cell_dim = n_input + 2*n_input*n_context # TODO: Is this a reasonable value
```

The number of units in the third layer, which feeds in to the LSTM, is determined by `n_cell_dim` as follows

```
In [11]: n_hidden_3 = 2 * n_cell_dim
```

Next, we introduce an additional variable `n_character` which holds the number of characters in the target language plus one, for the *blamk*. For English it is the cardinality of the set $\{a, b, c, ..., z, space, apostrophe, blank\}$ we referred to earlier.

```
In [12]: n_character = 29 # TODO: Determine if this should be extended with other punctuation
```

The number of units in the sixth layer is determined by `n_character` as follows

```
In [13]: n_hidden_6 = n_character
```

4 Graph Creation

Next we concern ourselves with graph creation.

First we create several place holders in our graph. The first two `x` and `y` are placeholders for our training data pairs.

```
In [14]: x = tf.placeholder("float", [None, n_steps, n_input + 2*n_input*n_context])
         y = tf.placeholder("string", [None, 1])
```

As `y` represents the text transcript of each element in a batch, it is of type "string" and has shape `[None, 1]` where the `None` dimension corresponds to the number of elements in the batch.

The placeholder `x` is a place holder for the the speech spectrograms along with their prefix and postfix contexts for each element in a batch. As it represents a spectrogram, its type is "float". The `None` dimension of its shape

```
[None, n_steps, n_input + 2*n_input*n_context]
```

has the same meaning as the None dimension in the shape of y. The n_steps dimension of its shape indicates the number of time-slices in the sequence. Finally, the n_input + 2*n_input*n_context dimension of its shape indicates the number of bins in Fourier transform n_input along with the number of bins in the prefix-context n_input*n_context and postfix-contex n_input*n_context.

The next placeholders we introduce istate_fw and istate_bw correspond to the initial states and cells of the forward and backward LSTM networks. As both of these are floats of dimension n_cell_dim, we define istate_fw and istate_bw as follows

```
In [15]: istate_fw = (tf.placeholder("float", [None, n_cell_dim]), tf.placeholder("float", [None
         istate_bw = (tf.placeholder("float", [None, n_cell_dim]), tf.placeholder("float", [None
```

As we will be employing dropout on the feedforward layers of the network we will also introduce a placeholder keep_prob which is a placeholder for the dropout rate for the feedforward layers

```
In [16]: keep_prob = tf.placeholder(tf.float32)
```

We will define the learned variables through two dictionaries. The first dictionary weights holds the learned weight variables. The second biases holds the learned bias variables.

The weights dictionary has the keys 'h1', 'h2', 'h3', 'h5', and 'h6' each keyed against the values of the corresponding weight matrix. In particular, the first key 'h1' is keyed against a value which is the learned weight matrix that converts an input vector of dimension n_input + 2*n_input*n_context to a vector of dimension n_hidden_1. Similarly, the second key 'h2' is keyed against a value which is the weight matrix converting an input vector of dimension n_hidden_1 to one of dimension n_hidden_2. The keys 'h3', 'h5', and 'h6' are similar. Likewise, the biases dictionary has biases for the various layers.

Concretely these dictionaries are given by

```
In [17]: # Store layers weight & bias
         # TODO: Is random_normal the best distribution to draw from?
         weights = {
             'h1': tf.Variable(tf.random_normal([n_input + 2*n_input*n_context, n_hidden_1])),
             'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
             'h3': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_3])),
             'h5': tf.Variable(tf.random_normal([(2 * n_cell_dim), n_hidden_5])),
             'h6': tf.Variable(tf.random_normal([n_hidden_5, n_hidden_6]))
         }
         biases = {
             'b1': tf.Variable(tf.random_normal([n_hidden_1])),
             'b2': tf.Variable(tf.random_normal([n_hidden_2])),
             'b3': tf.Variable(tf.random_normal([n_hidden_3])),
             'b5': tf.Variable(tf.random_normal([n_hidden_5])),
             'b6': tf.Variable(tf.random_normal([n_hidden_6]))
         }
```

Next we introduce a utility function BiRNN that can take our placeholders x, istate_fw, and istate_bw along with the dictionaries weights and biases and add all the apropos operators to our default graph.

```
In [18]: def BiRNN(_X, _istate_fw, _istate_bw, _weights, _biases):
             # Input shape: [batch_size, n_steps, n_input + 2*n_input*n_context]
             _X = tf.transpose(_X, [1, 0, 2])  # Permute n_steps and batch_size
             # Reshape to prepare input for first layer
             _X = tf.reshape(_X, [-1, n_input + 2*n_input*n_context]) # (n_steps*batch_size, n_i

             #Hidden layer with clipped RELU activation and dropout
             layer_1 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(_X, _weights['h1']), _biases['b1']
             layer_1 = tf.nn.dropout(layer_1, keep_prob)
             #Hidden layer with clipped RELU activation and dropout
             layer_2 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_1, _weights['h2']), _biases[
             layer_2 = tf.nn.dropout(layer_2, keep_prob)
             #Hidden layer with clipped RELU activation and dropout
             layer_3 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_2, _weights['h3']), _biases[
             layer_3 = tf.nn.dropout(layer_3, keep_prob)

             # Define lstm cells with tensorflow
             # Forward direction cell
             lstm_fw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)
             # Backward direction cell
             lstm_bw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)

             # Split data because rnn cell needs a list of inputs for the BRNN inner loop
             layer_3 = tf.split(0, n_steps, layer_3)

             # Get lstm cell output
             outputs, output_state_fw, output_state_bw = tf.nn.bidirectional_rnn(cell_fw=lstm_fw
                                                                                 cell_bw=lstm_bw
                                                                                 inputs=layer_3,
                                                                                 initial_state_f
                                                                                 initial_state_b

             # Reshape outputs from a list of n_steps tensors each of shape [batch_size, 2*n_cel
             # to a single tensor of shape [n_steps*batch_size, 2*n_cell_dim]
             outputs = tf.pack(outputs[0])
             outputs = tf.reshape(outputs, [-1, 2*n_cell_dim])

             #Hidden layer with clipped RELU activation and dropout
             layer_5 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(outputs, _weights['h5']), _biases[
             layer_5 = tf.nn.dropout(layer_5, keep_prob)
             #Hidden layer with softmax function
             layer_6 = tf.nn.softmax(tf.add(tf.matmul(layer_5, _weights['h6']), _biases['b6']))

             # Reshape layer_6 from a tensor of shape [n_steps*batch_size, n_hidden_6]
             # to a tensor of shape [batch_size, n_steps, n_hidden_6]
             layer_6 = tf.reshape(layer_6, [n_steps, batch_size, n_hidden_6])
             layer_6 = tf.transpose(layer_6, [1, 0, 2])  # Permute n_steps and batch_size
```

```
            # Return layer_6
            return layer_6
```

The first few lines of the function BiRNN

```
def BiRNN(_X, _istate_fw, _istate_bw, _weights, _biases):
    # Input shape: [batch_size, n_steps, n_input + 2*n_input*n_context]
    _X = tf.transpose(_X, [1, 0, 2])  # Permute n_steps and batch_size
    # Reshape to prepare input for first layer
    _X = tf.reshape(_X, [-1, n_input + 2*n_input*n_context])
    ...
```

reshape _X which has shape [batch_size, n_steps, n_input + 2*n_input*n_context] initially, to a tensor with shape [n_steps*batch_size, n_input + 2*n_input*n_context]. This is done to prepare the batch for input into the first layer which expects a tensor of rank 2.

The next few lines of BiRNN

```
    #Hidden layer with clipped RELU activation and dropout
    layer_1 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(_X, _weights['h1']), _biases['b1'])), relu_
    layer_1 = tf.nn.dropout(layer_1, keep_prob)
    ...
```

pass _X through the first layer of the non-recurrent neural network, then apply dropout to the result.

The next few lines do the same thing, but for the second and third layers

```
    #Hidden layer with clipped RELU activation and dropout
    layer_2 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_1, _weights['h2']), _biases['b2'])),
    layer_2 = tf.nn.dropout(layer_2, keep_prob)
    #Hidden layer with clipped RELU activation and dropout
    layer_3 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_2, _weights['h3']), _biases['b3'])),
    layer_3 = tf.nn.dropout(layer_3, keep_prob)
```

Next we create the forward and backward LSTM units

```
    # Define lstm cells with tensorflow
    # Forward direction cell
    lstm_fw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)
    # Backward direction cell
    lstm_bw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)
```

both of which have inputs of length n_cell_dim and bias 1.0 for the forget gate of the LSTM.

The next line of the funtion BiRNN does a bit more data preparation.

```
    # Split data because rnn cell needs a list of inputs for the RNN inner loop
    layer_3 = tf.split(0, n_steps, layer_3)
```

It splits layer_3 in to n_steps tensors along dimension 0 as the LSTM BRNN expects its input to be of shape n_steps *[batch_size, 2*n_cell_dim].

The next line of BiRNN

```
# Get lstm cell output
outputs, output_state_fw, output_state_bw  = tf.nn.bidirectional_rnn(cell_fw=lstm_fw_cell,
                                                                     cell_bw=lstm_bw_cell,
                                                                     inputs=layer_3,
                                                                     initial_state_fw=_istat
                                                                     initial_state_bw=_istat
```

feeds `layer_3` to the LSTM BRNN cell and obtains the LSTM BRNN output.

The next lines convert `outputs` from a list of rank two tensors into a rank two tensor in preparation for passing it to the next neural network layer

```
# Reshape outputs from a list of n_steps tensors each of shape [batch_size, 2*n_cell_dim]
# to a single tensor of shape [n_steps*batch_size, 2*n_cell_dim]
outputs = tf.pack(outputs)
outputs = tf.reshape(outputs, [-1, 2*n_cell_dim])
```

The next couple of lines feed `outputs` to the fifth hidden layer

```
#Hidden layer with clipped RELU activation and dropout
layer_5 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(outputs, _weights['h5']), _biases['b5'])),
layer_5 = tf.nn.dropout(layer_5, keep_prob)
```

The next line of `BiRNN`

```
#Hidden layer with softmax function
layer_6 = tf.nn.softmax(tf.add(tf.matmul(layer_5, _weights['h6']), _biases['b6']))
```

Applies the weight matrix `_weights['h6']` and bias `_biases['h6']`to the output of `layer_5` creating `n_classes` dimensional vectors, then performs softmax on them.

The next lines of `BiRNN`

```
# Reshape layer_6 from a tensor of shape [n_steps*batch_size, n_hidden_6]
# to a tensor of shape [batch_size, n_steps, n_hidden_6]
layer_6 = tf.reshape(layer_6, [n_steps, batch_size, n_hidden_6])
layer_6 = tf.transpose(layer_6, [1, 0, 2])  # Permute n_steps and batch_size
```

reshapes `layer_6` to the slightly more useful shape `[batch_size, n_steps, n_hidden_6]`.

The final line of `BiRNN` returns `layer_6`

```
# Return layer_6
return layer_6
```

Next we actually call `BiRNN` with the apropos data

```
In [19]: layer_6 = BiRNN(x, istate_fw, istate_bw, weights, biases)
```

5 Loss Function

In accord with Deep Speech: Scaling up end-to-end speech recognition, the loss function used by our network should be the CTC loss function[2]. Unfortunately, as of this writing, the CTC loss function[2] is not implemented within TensorFlow[5]. Thus we will have to implement it ourselves. The next few sections are dedicated to this implementation.