David Currie Follow
I'm working on Machine Learning projects (focusing on NLP), and actively looking for a job relating to data …
May 9 · 10 min read

Text Summarization with Amazon Reviews

With many products comes many reviews for training.

In this article, we will be using fine food reviews from Amazon to build a model that can summarize text. Specifically, we will be using the description of a review as our input data, and the title of a review as our target data. To download the dataset, and learn more about it, you can find it on Kaggle. If you decide to build a model like mine, you will see that it is able to generate some very good summaries:

Description(1): The coffee tasted great and was at such a good price! I highly recommend this to everyone!

Summary(1): great coffee

Description(2): This is the worst cheese that I have ever bought! I will never buy it again and I hope you won't either!

Summary(2): omg gross gross

The code is written in Python and TensorFlow 1.1 will be our deep learning library. If you haven't used TensorFlow 1.1 yet, you will see that building a seq2seq model is quite a bit different than with previous versions. To help generate some great summaries, we will be using a bi-directional RNN in our encoding layer, and attention in our decoding layer. The model that we will build is similar to Xin Pan's and Peter Liu's model from "Sequence-to-Sequence with Attention Model for Text Summarization" (GitHub). Also, quite a bit of credit should be given to Jaemin Cho tutorial (GitHub). This is my first project with TensorFlow 1.1 and his tutorial really helped me to get my code in working order.

Note: Like my other articles, I will only be showing the main parts of my work, and I will remove most of the comments to help keep this short. You can see the whole project on my Github.

. . .

Preparing the Data

To start things off, I'm going to explain how I cleaned the text:

- Convert to lowercase.
- Replace contractions with their longer forms.
- Remove any unwanted characters (this step needs to be done after replacing the contractions because apostrophes will be removed. Notice the backward slash before the hyphen. Without this slash, all characters that are 'between' the characters before and after the hyphen would be removed. This can create some unwanted effects. To give you an example, typing "a-d" would remove a,b,c,d.).
- Stop words will only be removed from the descriptions. They are not very relevant in training the model, so by removing them we are able to train the model faster because there is less data. They will remain in the summaries because they are rather short and I would prefer for them to sound more like natural phrases.

```
def clean_text(text, remove_stopwords = True):

    # Convert words to lower case
    text = text.lower()

    # Replace contractions with their longer forms
    if True:
        text = text.split()
        new_text = []
        for word in text:
            if word in contractions:
                new_text.append(contractions[word])
            else:
                new_text.append(word)
        text = " ".join(new_text)

    # Format words and remove unwanted characters
    text = re.sub(r'https?:\/\/.*[\r\n]*', '', text, 
                  flags=re.MULTILINE)
    text = re.sub(r'\<a href', ' ', text)
    text = re.sub(r'&amp;', '', text) 
    text = re.sub(r'[_"\-;%()|+&=*%.,!?:#$@\[\]/]', ' ', 
text)
    text = re.sub(r'<br />', ' ', text)
    text = re.sub(r'\'', ' ', text)

    # Optionally, remove stop words
    if remove_stopwords:
        text = text.split()
        stops = set(stopwords.words("english"))
        text = [w for w in text if not w in stops]
        text = " ".join(text)


return text
```

We are going to use pre-trained word vectors to help improve the performance of our model. In the past, I have used GloVe for this, but I found another set of word embeddings, called ConceptNet Numberbatch (CN). Based on the work of its creators, it seems to outperform GloVe, which makes sense because CN is an ensemble of embeddings that includes GloVe.

```
embeddings_index = {}
with open('/Users/Dave/Desktop/Programming/numberbatch-en-
17.02.txt', encoding='utf-8') as f:
    for line in f:
        values = line.split(' ')
        word = values[0]
        embedding = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = embedding
```

We are going to limit our vocabulary to words that are either in CN or occur more than 20 times in our dataset. This will allow us to have very good embeddings for every word because the model can better understand how words are related when they see them more times.

When building your `word_embedding_matrix` , it is very important that you set its "dtype" of np.zeros to float32. The default value is float64, but this won't work with TensorFlow because it expects values to be limited to _32.

```
embedding_dim = 300
nb_words = len(vocab_to_int)


word_embedding_matrix = np.zeros((nb_words, embedding_dim),
                                 dtype=np.float32)
for word, i in vocab_to_int.items():
    if word in embeddings_index:
        word_embedding_matrix[i] = embeddings_index[word]
    else:
        # If word not in CN, create a random embedding for
it
        new_embedding = np.array(np.random.uniform(-1.0,
1.0, embedding_dim))
        embeddings_index[word] = new_embedding
        word_embedding_matrix[i] = new_embedding
```

To help train the model faster, we are going to sort the reviews by the length of the descriptions from shortest to longest. This will help each batch to have descriptions of similar lengths, which will result in less padding, thus less computing. We could do a secondary sort by the length of the summaries, but this would result in quite a bit of looping to organize the data. Plus, I doubt that it would reduce much extra padding since the summaries are rather short.

Some reviews will not be included because of the number of UNK tokens that are in the description or summary. If there is more than 1 UNK in the description or any UNKs in the summary, the review will not be used. This is done to ensure that we are building the model with meaningful data. Fewer than 0.7% of words are UNKs, so not many reviews will be removed.

```
for length in range(min(lengths_texts.counts),
max_text_length):
   for count, words in enumerate(int_summaries):
      if (len(int_summaries[count]) >= min_length and
          len(int_summaries[count]) <= max_summary_length
and
          len(int_texts[count]) >= min_length and
          unk_counter(int_summaries[count]) <=
unk_summary_limit and
          unk_counter(int_texts[count]) <= unk_text_limit
and
          length == len(int_texts[count])
         ):
          sorted_summaries.append(int_summaries[count])
          sorted_texts.append(int_texts[count])
```

. . .

Building the Model

There are quite a few placeholders that we need to make for this model. Most of them are self explanatory, but the just to be clear on a few, `summary_length` and `text_length` are the lengths of each sentence within a batch, and `max_summary_length` is the maximum length of a summary within a batch.

```
def model_inputs():

    input_data = tf.placeholder(tf.int32,[None, None],
name='input')
    targets = tf.placeholder(tf.int32, [None, None],
name='targets')
    lr = tf.placeholder(tf.float32, name='learning_rate')
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
    summary_length = tf.placeholder(tf.int32, (None,),
                                    name='summary_length')
    max_summary_length = tf.reduce_max(summary_length,
                                       name='max_dec_len')
    text_length = tf.placeholder(tf.int32, (None,),
                                 name='text_length')

return input_data, targets, lr, keep_prob, summary_length,
       max_summary_length, text_length
```

To build our encoding layer, we are going to use a bidirectional RNN with LSTMs. If you have worked with earlier versions of TensorFlow, you will notice that this layout is different than how two layers would normally be created. Typically, one would just wrap two LSTMs in tf.contrib.rnn.MultiRNNCell, but this method is no longer viable. Here is a method that I found, which works just as well, despite being more code.

Some things to note:

- You'll need to use `tf.variable_scope` so that your variables are reused with each layer. If you don't really know what I am talking about, then check out TensorFlow's word2vec tutorial.
- Since we are using a bi-direction RNN, the outputs need to be concatenated
- The initializers are the same as those in Pan's and Liu's model.

```
def encoding_layer(rnn_size, sequence_length, num_layers,
                   rnn_inputs, keep_prob):

    for layer in range(num_layers):
        with tf.variable_scope('encoder_{}'.format(layer)):
            cell_fw = tf.contrib.rnn.LSTMCell(rnn_size,

initializer=tf.random_uniform_initializer(-0.1,

0.1,

seed=2))
            cell_fw = tf.contrib.rnn.DropoutWrapper(cell_fw,
                                    input_keep_prob =
keep_prob)



            cell_bw = tf.contrib.rnn.LSTMCell(rnn_size,

initializer=tf.random_uniform_initializer(-0.1,

0.1,

seed=2))
            cell_bw = tf.contrib.rnn.DropoutWrapper(cell_bw,
                                    input_keep_prob =
keep_prob)



            enc_output, enc_state =
tf.nn.bidirectional_dynamic_rnn(
                                     cell_fw,
                                     cell_bw,
                                     rnn_inputs,
                                     sequence_length,
                                     dtype=tf.float32)

    enc_output = tf.concat(enc_output,2)

    return enc_output, enc_state
```

To create our training and inference decoding layers, there are some functions that are new for TF 1.1.. To break things down for you:

- `TrainingHelper` reads a sequence of integers from the encoding layer.
- `BasicDecoder` processes the sequence with the decoding cell, and an output layer, which is a fully connected layer. `initial_state` comes from our `DynamicAttentionWrapperState` that you will see soon.
- `dynamic_decode` creates our outputs that will be used for training.

```
def training_decoding_layer(dec_embed_input, summary_length,
                            dec_cell, initial_state,
output_layer,
                            vocab_size, max_summary_length):

    training_helper = tf.contrib.seq2seq.TrainingHelper(
                         inputs=dec_embed_input,
                         sequence_length=summary_length,
                         time_major=False)



    training_decoder = tf.contrib.seq2seq.BasicDecoder(
                          dec_cell,
                          training_helper,
                          initial_state,
                          output_layer)

    training_logits, _ = tf.contrib.seq2seq.dynamic_decode(
                            training_decoder,
                            output_time_major=False,
                            impute_finished=True,

maximum_iterations=max_summary_length)
    return training_logits
```

As you can see, this is very similar to the training layer. The main difference is `GreedyEmbeddingHelper` , which uses the argmax of the output (treated as logits) and passes the result through an embedding layer to get the next input. Although it is asking for `start_tokens` , we only have one, `<GO>` .

```
def inference_decoding_layer(embeddings, start_token,
end_token,
                             dec_cell, initial_state,
output_layer,
                             max_summary_length,
batch_size):


    start_tokens = tf.tile(tf.constant([start_token],
                                       dtype=tf.int32),
                                       [batch_size],
                                       name='start_tokens')

    inference_helper = 
tf.contrib.seq2seq.GreedyEmbeddingHelper(
                           embeddings,
                           start_tokens,
                           end_token)

    inference_decoder = tf.contrib.seq2seq.BasicDecoder(
                            dec_cell,
                            inference_helper,
                            initial_state,
                            output_layer)

    inference_logits, _ = tf.contrib.seq2seq.dynamic_decode(
                              inference_decoder,
                              output_time_major=False,
                              impute_finished=True,

maximum_iterations=max_summary_length)

    return inference_logits
```

Although the decoding layer may look a bit complex, it can be broken down into three parts: decoding cell, attention, and getting our logits.

Decoding Cell:

- Just a two layer LSTM with dropout.

Attention:

- I'm using Bhadanau instead of Luong for my attention style. Using it helps the model to train faster and can produce better results (here's a good paper that compares the two along with many of aspects of a seq2seq model).

- `DynamicAttentionWrapper` applies the attention mechanism to our decoding cell.
- `DynamicAttentionWrapperState` creates the initial state that we use for our training and inference layers. Since we are using a bidirectional RNN in our decoding layer, we can only using either the forward or backward state. I chose forward since this is the state that Pan and Liu chose.

```
def decoding_layer(dec_embed_input, embeddings, enc_output,
                   enc_state, vocab_size, text_length,
                   summary_length, max_summary_length,
rnn_size,
                   vocab_to_int, keep_prob, batch_size,
num_layers):

    for layer in range(num_layers):
        with tf.variable_scope('decoder_{}'.format(layer)):
            lstm = tf.contrib.rnn.LSTMCell(rnn_size,

initializer=tf.random_uniform_initializer(-0.1,

0.1,

seed=2))
            dec_cell = tf.contrib.rnn.DropoutWrapper(
                           lstm,
                           input_keep_prob = keep_prob)

    output_layer = Dense(vocab_size,
           kernel_initializer =
tf.truncated_normal_initializer(
                                    mean=0.0,
                                    stddev=0.1))

    attn_mech = tf.contrib.seq2seq.BahdanauAttention(
                      rnn_size,
                      enc_output,
                      text_length,
                      normalize=False,
                      name='BahdanauAttention')



    dec_cell =
tf.contrib.seq2seq.DynamicAttentionWrapper(dec_cell,

attn_mech,

rnn_size)
```

```
    initial_state =
tf.contrib.seq2seq.DynamicAttentionWrapperState(
                        enc_state[0],
                        _zero_state_tensors(rnn_size,
                                            batch_size,
                                            tf.float32))

    with tf.variable_scope("decode"):
        training_logits = training_decoding_layer(
                              dec_embed_input,
                              summary_length,
                              dec_cell,
                              initial_state,
                              output_layer,
                              vocab_size,
                              max_summary_length)

    with tf.variable_scope("decode", reuse=True):
        inference_logits = inference_decoding_layer(
                              embeddings,
                              vocab_to_int['<GO>'],
                              vocab_to_int['<EOS>'],
                              dec_cell,
                              initial_state,
                              output_layer,
                              max_summary_length,
                              batch_size)


    return training_logits, inference_logits
```

We are going to use our `word_embedding_matrix` that we created earlier as our embeddings. Both the encoding and decoding sequences will use these embeddings.

The rest of this function is mainly collecting the outputs of the previous functions so that they are ready to be used to train the model and generate new summaries.

```
def seq2seq_model(input_data, target_data, keep_prob,
text_length,
                  summary_length, max_summary_length,
                  vocab_size, rnn_size, num_layers,
vocab_to_int,
                  batch_size):

    embeddings = word_embedding_matrix

    enc_embed_input = tf.nn.embedding_lookup(embeddings,
input_data)
    enc_output, enc_state = encoding_layer(rnn_size,
                                             text_length,
                                             num_layers,
                                             enc_embed_input,
                                             keep_prob)

    dec_input = process_encoding_input(target_data,
                                         vocab_to_int,
                                         batch_size)
    dec_embed_input = tf.nn.embedding_lookup(embeddings,
dec_input)

    training_logits, inference_logits  = decoding_layer(
         dec_embed_input,
         embeddings,
         enc_output,
         enc_state,
         vocab_size,
         text_length,
         summary_length,
         max_summary_length,
         rnn_size,
         vocab_to_int,
         keep_prob,
         batch_size,
         num_layers)

    return training_logits, inference_logits
```

Creating the batches is pretty typical. The only thing I want to point out here is creating the `pad_summaries_lengths` and `pad_texts_lengths`. These contain the lengths of the summaries/texts that are within a batch and will be used as the values for the inputs: `summary_length` and `text_length`. I understand that it might seem like a weird method to find these input values, but it was the best/only solution that I could find.

```
def get_batches(summaries, texts, batch_size):


    for batch_i in range(0, len(texts)//batch_size):
        start_i = batch_i * batch_size
        summaries_batch = summaries[start_i:start_i +
batch_size]
        texts_batch = texts[start_i:start_i + batch_size]
        pad_summaries_batch = np.array(pad_sentence_batch(
                                 summaries_batch))
        pad_texts_batch =
np.array(pad_sentence_batch(texts_batch))

        pad_summaries_lengths = []
        for summary in pad_summaries_batch:
            pad_summaries_lengths.append(len(summary))

        pad_texts_lengths = []
        for text in pad_texts_batch:
            pad_texts_lengths.append(len(text))

        yield (pad_summaries_batch,
               pad_texts_batch,
               pad_summaries_lengths,
               pad_texts_lengths)
```

These are the hyperparameters that I used to train this model. I don't think that there is anything too exciting about them, they're pretty standard. The 100 epochs may have caught you eye. I'm using this large value so that my model becomes fully trained, and only stops training as a result of early stopping (when the loss stops decreasing).

```
epochs = 100
batch_size = 64
rnn_size = 256
num_layers = 2
learning_rate = 0.01
keep_probability = 0.75
```

I'm going to skip over building the graph and how to train the model. There are a few neat things, such as how I incorporated learning rate decay and early stopping, but I don't think it's worth taking up the space here (check out my GitHub if your interested).

. . .

Generating Your Own Summaries

You will be able to either create your own descriptions or use one from the dataset as your input data. It's a bit more fun to use your own descriptions because you can be creative to see what kind summary the model creates. The function below will prepare your description for the model by using the `clean_text` function that I described earlier.

```
def text_to_seq(text):

    text = clean_text(text)
    return [vocab_to_int.get(word, vocab_to_int['<UNK>'])
for word in text.split()]
```

We need to load quite a few tensors to generate new summaries. Nonetheless, this isn't too difficult. When running the session, `input_data` and `text_length` need to be multiplied by `batch_size` to match the input parameters of the model. You can set `summary_length` to whatever value you like, but I decided to make it random to keep things exciting. The main thing to remember is to keep it within the range of the summary lengths that the model was trained with, i.e. 2–13.

```
input_sentence = "This is the worst cheese that I have ever
bought! I will never buy it again and I hope you won't
either!"
text = text_to_seq(input_sentence)

#random = np.random.randint(0,len(clean_texts))
#input_sentence = clean_texts[random]
#text = text_to_seq(clean_texts[random])

checkpoint = "./best_model.ckpt"


loaded_graph = tf.Graph()
with tf.Session(graph=loaded_graph) as sess:

    loader = tf.train.import_meta_graph(checkpoint +
'.meta')
    loader.restore(sess, checkpoint)

    input_data = loaded_graph.get_tensor_by_name('input:0')
    logits =
loaded_graph.get_tensor_by_name('predictions:0')
    text_length =
loaded_graph.get_tensor_by_name('text_length:0')
    summary_length =
         loaded_graph.get_tensor_by_name('summary_length:0')
    keep_prob =
loaded_graph.get_tensor_by_name('keep_prob:0')

    answer_logits = sess.run(logits, {
         input_data: [text]*batch_size,
         summary_length: [np.random.randint(4,8)],
         text_length: [len(text)]*batch_size,
         keep_prob: 1.0})[0]

# Remove the padding from the tweet
pad = vocab_to_int["<PAD>"]

print('\nOriginal Text:', input_sentence)


print('Text')
print('Word Ids:    {}'.format([i for i in text if i !=
pad]))
print('Input Words: {}'.format([int_to_vocab[i] for
                                 i in text if i != pad]))

print('\nSummary')
print('Word Ids: {}'.format([i for i in answer_logits if i
!= pad]))
print('Response Words: {}'.format([int_to_vocab[i] for i in
                                  answer_logits if i !=
pad]))
```

One last thing to note, I trained my model with only a subset of 50,000 reviews. Since I was using my MacBook Pro, it would have taken me days to train this model if I used all of the data. Whenever I can, I use a GPU on FloydHub to train my models. If you take the time, you can upload the data and ConceptNet Numberbatch to FloydHub, then train the model with all of the reviews. Let me know how to goes!

. . .

That’s all for this project! I hope that you found it interesting and learned a thing or two. Although this model performs rather well, and with just a subset of the data, it would be neat to try to expand the architecture to improve the quality of the generated summaries. This paper uses both RNNs and CNNs to predict sentiment, perhaps it could also work here. If anyone tries to do this, it would be great if you posted a link in the comments section below because I’d be excited to see it.

Thanks for reading, and if you have any questions or comments, please let me know! Cheers!