Seq2Seq docs About

Building seq2seq package in Tensorflow

Apr 17, 2017 • Vasanth Kalingeri

Package outline

We require 3 blocks for the process to work:

1. Encoder
2. Context
3. Decoder

The encoder will encode the source sentence, the hidden states from the encoder are fed into the context block, the context block will hold these hidden states and create the necessary preprocessing for attention. The decoder uses the context object and using attention creates necessary summaries that are helpful for decoding. The most obvious way to design these blocks is using classes.

Tensorflow works by compiling the computation graphs, so the graphs have to be defined first. This definition of all the nodes of the graph thus has to be done in the constructor. The functions of a class will use the session object to execute the nodes created in the constructor. The structure of the code should look like below:

```
class Encoder(object):

    def __init__(self, state_size, num_layers, cell_type='LSTM', embed=Tru
        # Initialize the weights and the cell of the encoder RNN
        # Create nodes that perform the decoding of the RNN
        # Use dynamic_rnn here to make it easier

    def encode(self, session, initial_state, input_list, input_list_length
        # Runs the RNN and returns a list of hidden state vectors based or

class Context(object):

    def __init__(self, hidden_states):
```

```
        # Create a matrix of hidden states obtained
        # Create nodes for last context
        # Create nodes that might be useful for implementing attention

class Decoder(object):

    def __init__(self, context, state_size, num_layers, cell_type='LSTM', 
        # Initialize the weights and the cell of the decoder RNN
        # Store the context object that is created

        # Create nodes for training
        # Use the context for each step where you learn the weights in the
        # The structure of attention should go here because here is where 
        # Should return the outputs and the loss and perform one step of t
        # Get the hidden states, use it to calculate the outputs done by l
        # Implement sampled softmax on this output to find the loss and re

        # Create nodes for greedy_decode

        # Create nodes for beam_decode
        # Generates beam_size proposals and feeds to next timestep
        # Goes till one the outputs generated is an EOS token
```

These forms the blocks for the neural network, a sequence to sequence model is one in which the blocks are connected. So a class for seq2seq model has to be designed as follows:

```
class Seq2Seq(object):

    def __init__(self, params1, params2, params3):
        self.encoder = Encoder(params1)
        self.context = Context(self.encoder.hidden_states, params2)
        self.decoder = Decoder(self.context, params3)
```

This establishes connections between the various abstractions and thus we get a simple seq2seq model. Now we need to fill in the TF code. We store all the components of the seq2seq model in nn.py and the class Seq2Seq is present in **init**.py.

Notes in writing the tensorflow code.

There has to be a cell node that is to be defined, should be named the LSTM cell node.

```
def encode(self, X, X_lengths):
        with tf.name_scope('Encoder') as main_scope:
            self.results = tf.nn.dynamic_rnn(
                                    cell=self.cell,
                                    dtype=tf.float64,
                                    sequence_length=X_lengths,
                                    inputs=X)
            return self.results

with tf.Session() as sess:
        with tf.name_scope('Encoder') as main_scope:
            enc = Encoder(source_vocab_size, state_size, num_layers, cell_
            print sess.run(enc.encode, feed_dict={'X':X, 'X_lengths':X_ler
```

This throws an error because X, X_lengths are not tf.placeholders, so if you want to pass something using the feed_dict they have to be placeholders, you can't just pass them to a function that way. A really interesting way of intializing the states of the LSTM

```
c_state = tf.placeholder(...)
h_state = tf.placeholder(...)
initial_state = tf.nn.rnn_cell.LSTMStateTuple(c_state, h_state)

sess.run(..., feed_dict={c_state: ..., h_state: ...})
```

So there seems to be this idea of a graph, you create a graph and add nodes to it. Then you create a session and then execute that graph that was created. In our case, nodes would have to be added by encoder object, context object and decoder object. We can only do this if we create a graph object outside and pass this to each of instances of the blocks. For this reason, the init code of all three classes will contain a pass of the graph object

```
graph = tf.graph()

# pass this graph to encoder, context and decoder where each of them add r
```

The idea that you are approaching this problem with is that you will create all the nodes of the graph in the encoder, add nodes to the created graph in context, add more nodes in the decoder. In the end you will have three classes whose functions are actually functions and whose graphs you create normally.

Problem you are facing? You are creating everything in the graph in the init, but when you add an embedding it is failing because you are not adding the embedding correctly.

Solution: Create a list of them, create a tensor out of that list using tf.stack and then transpose the columns to get the correct embedding matrix.

So now done with the encoder class that is defined as follows:

```
class Encoder(object):

    def __init__(self, graph, source_vocab_size, state_size, num_layers, m
                 cell_type='LSTM', embed_size=None, train_embed=True):
```

Take as input the graph object and make that the default graph to which all the nodes are added.

```
        self.graph = graph
        with self.graph.as_default():

            self.cell = _create_cell(state_size, num_layers, cell_type)
```

Where _create_cell function looks like below:

```
# Function returns the correct cell based on cell type
def _create_cell(state_size, num_layers, cell_type):
    def single_cell(state_size, cell_type):
        if cell_type == 'LSTM':
            cell = tf.contrib.rnn.BasicLSTMCell(state_size, state_is_tuple
        elif cell_type == 'GRU':
            cell = tf.contrib.rnn.GRUCell(state_size, state_is_tuple=True)
        return cell
    # Increases the number of LSTMs accordingly
    if num_layers > 1:
        cell = tf.contrib.rnn.MultiRNNCell([_single_cell(state_size, cell_
    else:
        cell = single_cell(state_size, cell_type)
    return cell
```

Now with the cell defined, we optionally create the embedding matrix, the embeddings are initialized from a file like word2vec.

```
            self.max_length = max_length
            # Initialize the embedding
            self.embedding = None
            self.inp_dims = source_vocab_size
```

```
        if embed_size:
            self.inp_dims = embed_size
            if train_embed is False:
                embeddings_matrix = np.array(pickle.load(open('../../r
            else:
                embeddings_matrix = np.random.rand(source_vocab_size, 

            self.embedding = variable_scope.get_variable("embedding", 
                                    [source_vocab_size, embed_size
                                    initializer=tf.constant_initia
                                    trainable=train_embed)




        # Create placeholders for encoder_inputs and lengths
        self.encoder_inputs = tf.placeholder(tf.int32, [None, max_leng
        self.encoder_lengths = tf.placeholder(tf.float64, [None], name
```

We feed the encoder_inputs which are just index numbers as input, this has to be converted to the embedding matrix using the embedding lookup function, this is done here below. It is ensured that with or without embeddings, the size of the input matrix is always NxTxD

```
        if self.embedding:
            #Create embedding lookup function for the entire batch
            self.embed_inputs = []
            for t in xrange(max_length):
                encoder_inp = self.encoder_inputs[:, t]
                self.embed_inputs.append(tf.cast(tf.nn.embedding_looku
            # Transpose the time axis so we have shape NxTxD tensor
            self.embed_inputs = tf.transpose(tf.stack(self.embed_input
        else:
            self.embed_inputs = tf.cast(self.encoder_inputs, tf.float6
            # Need to reshape to work with dynamic_rnn input
            self.embed_inputs = tf.reshape(self.embed_inputs, [-1, max

        self.enc_states, _ = tf.nn.dynamic_rnn(
                            cell=self.cell,
                            dtype=tf.float64,
                            sequence_length=self.encoder_lengths,
                            inputs=self.embed_inputs)
```

The graph definition ends here. We now define functions that work on the sessions of the graph.

```
    def encode(self, session, enc_inputs, enc_lengths):

        input_feed = {self.encoder_inputs: enc_inputs, self.encoder_length
        results = session.run(self.enc_states, feed_dict=input_feed)

        return results
```

Context class

The main function of the context class is to hold the entire hstates and work on that in various ways. So for now let the init function just store the hstates tensor that is created.

Context_size decides the number of different ways of looking at a given hidden state vector. We define the variables that will be used by the attention model in context, so the idea is whenever learning takes place, it will be like the context class is doing the learning, the decoder class is doing the operation of training and computing the loss

```
class Context(object):

    def __init__(self, graph, hstates, encoder_size, decoder_size, context

        self.graph = graph
        self.context_size = context_size
        with self.graph.as_default():
            self.V = variable_scope.get_variable("context_V",
                                                 [context_size, decoder_siz
                                                 initializer=tf.random_norm
                                                 dtype=tf.float64)

            self.W = variable_scope.get_variable("context_W",
                                                 [context_size, encoder_siz
                                                 initializer=tf.random_norm
                                                 dtype=tf.float64)

            self.V1 = variable_scope.get_variable("context_v1",
                                                  [1, context_size],
                                                  initializer=tf.random_norm
                                                  dtype=tf.float64)
```

```
            self.last_context = hstates[:, -1, :]

            # (N x T x enc_size) x (enc_size x context_size) = (N x T x co
            # To create such a product we resize hstates first do matmul a
            F = tf.reshape(hstates, [-1, encoder_size])
            self.WF = tf.matmul(F, tf.transpose(self.W, [1,0]))
            self.WF = tf.reshape(self.WF, [-1, tf.shape(hstates)[1], conte
            # WF is a (N x T x context_size) tensor that the attention moc
```

As we see above the context class just contains some definitions, there is no real function performed by this class except for graph definition.

Decoder class

Need to perform the same function as encoder so needs to define a class take in the size and stuff and perform decoding given the context.

```
class Decoder(object):
    """
        Function of the decoder is to just do decoding given the context v
        Implement RNN that just performs this decoding, again you have to
    """
    def __init__(self, graph, context, target_vocab_size, state_size,
                 num_layers, max_length, embedding, cell_type='LSTM'):

        self.graph = graph
        self.embedding = embedding
        with self.graph.as_default():
            # Variable definitions
            self.cell = _create_cell(state_size, num_layers, cell_type)
            self.w_out = variable_scope.get_variable("w_out",
                                                    [state_size, target_vocab_
                                                    initializer=tf.random_norm
                                                    dtype=tf.float64)
            self.b_out = variable_scope.get_variable("b_out",
                                                    [target_vocab_size],
                                                    initializer=tf.random_norm
                                                    dtype=tf.float64)
            self.decoder_inputs = tf.placeholder(tf.int32, [None, max_leng
            self.decoder_lengths = tf.placeholder(tf.float64, [None], name
            self.decoder_outputs = tf.placeholder(tf.float32, [None, max_l
```

```
        # Creating embeddings if needed
        if self.embedding:
            #Create embedding lookup function for the entire batch
            self.embed_inputs = []
            for t in xrange(max_length):
                decoder_inp = self.decoder_inputs[:, t]
                self.embed_inputs.append(tf.cast(tf.nn.embedding_looku
            # Transpose the time axis so we have shape NxTxD tensor
            self.embed_inputs = tf.transpose(tf.stack(self.embed_input
        else:
            self.embed_inputs = tf.cast(self.decoder_inputs, tf.float6
            # Need to reshape to work with dynamic_rnn input
            self.embed_inputs = tf.reshape(self.embed_inputs, [-1, max

        # Currently summary vector is being created without attention
        self.summary = context.last_context
        # TODO add ops that would create the summary vector using atte

        # TODO Now embed inputs is of size N x T x D to include the co
        # TODO the summary vector to make the dimensions N x T x (D +


        print self.embed_inputs

        # Running the step of the decoder
        self.dec_states, _ = tf.nn.dynamic_rnn(
                    cell=self.cell,
                    dtype=tf.float64,
                    sequence_length=self.decoder_lengths,
                    inputs=self.embed_inputs)

        # Training the network based on the output of decoder
        self.outputs = []
        losses = []

        for t in xrange(max_length):
            output = tf.matmul(self.dec_states[:, t, :], self.w_out) +
            # Need to convert this to probabilities

            loss = tf.nn.sampled_softmax_loss(tf.cast(tf.transpose(sel
                                            tf.cast(self.b_out, tf.flo
                                            tf.reshape(self.decoder_ou
                                            tf.cast(self.dec_states[:,
                                            num_sampled=1000,
```

```
                                                num_classes=target_vocab_s
                                                num_true=1)

                losses.append(loss)
                self.outputs.append(output)

            self.outputs = tf.stack(self.outputs) # N x T x source_vocab
            losses = tf.stack(losses) # N x T

            # Mask the losses that don't carry meaning and average over th
            mask = tf.sequence_mask(self.decoder_lengths, max_length) # N 
            losses = losses * tf.cast(mask, tf.float32)
            self.total_avg_loss = tf.cast(tf.reduce_sum(losses), tf.float6

            # TODO: Functions that perform beam_decode and greed_decode ha
```

So far so good, all the nodes seemed to be correctly defined, now we need to design the seq2seq class that actually implements all these functions.

Seq2Seq class

This class has to be designed such that all of the nodes can be tied up easily together into one coherent unit.

Getting an error that the cell already exists, do you want to reuse. For now just set it to reuse and check for other errors, correct this part of the code later. Solved this error by using different scopes.

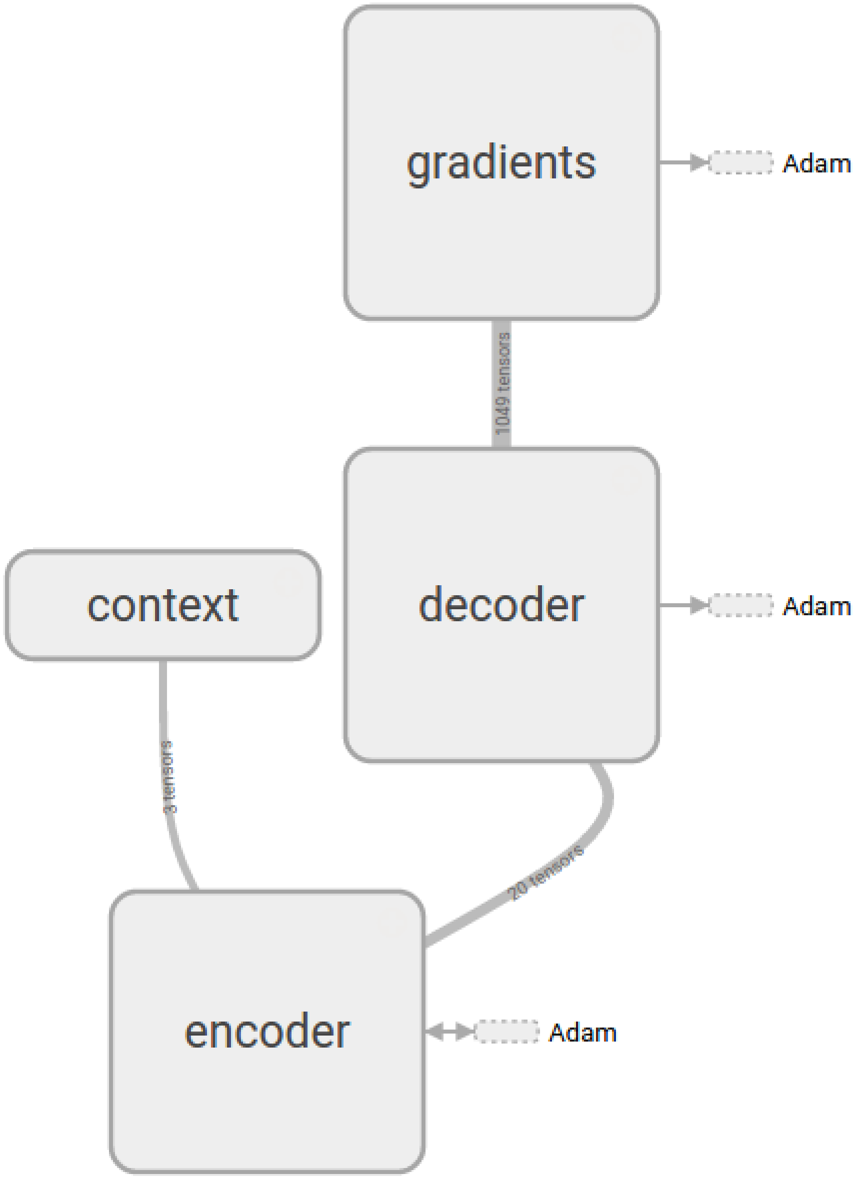
The sampler says that log-uniform-sampling should only be used when the words are given in order where the most frequent words are present first — Have to make sure that this is the case, else training will become slower due to incorrect sampling.

```
class Seq2Seq(object):

    def __init__(self, graph, source_vocab_size, enc_size, enc_layers, enc
                 context_size, target_vocab_size, dec_size, dec_layers, dec
                 cell_type='LSTM', embed_size=None, train_embed=True):

        # Initialize all the passed variables
        self.graph = graph
        self.source_vocab_size = source_vocab_size
        self.enc_size = enc_size
        self.enc_layers = enc_layers
```

```
        self.enc_max_length = enc_max_length
        self.context_size = context_size
        self.target_vocab_size = target_vocab_size
        self.dec_size = dec_size
        self.dec_layers = dec_layers
        self.dec_max_length = dec_max_length
        self.cell_type = cell_type
        self.embed_size = embed_size

        # Create encoder namescope
        with self.graph.as_default():
            with tf.variable_scope('encoder'):
                self.encoder = Encoder(graph, source_vocab_size, enc_size,
                                    cell_type, embed_size, train_embed)

            # Context namescope
            with tf.variable_scope('context'):
                self.context = Context(graph, self.encoder.enc_states, enc

            print self.encoder.embedding
            # Decoder namescope

            with tf.variable_scope('decoder'):
                self.decoder = Decoder(graph, self.context, target_vocab_s
                                    dec_layers, dec_max_length, self.encod

            self.loss = self.decoder.total_avg_loss

            # Either get all trainable variables here and apply the gradie
            opt = tf.train.AdamOptimizer() # Use default hyperparams for r
            train_step = opt.minimize(self.loss)
```

Now that the graph construction seems to be compiling correctly, we have to go to the next step of actually training the model and verifying the loss. Before training, we also have to use tensorboard to verify if the graph we constructed is the correct graph.

The visualization on tensorboard looks like below:



On looking into tensorboard, it is seen that the context and decoder have no connectivity, which is true as the decoder is not using the context vector. To make use of the context vector, we have to look into how dynamic_rnn code is working and modify parts of that code to simulate the rnn in steps. The tensorboard has a connection between the encoder and the decoder since we are sharing the word embeddings between the encoder and the decoder.

The decoder cannot be implemented as a dynamic_rnn because the input at each stage changes depending on the previous hidden state, we can do this with a custom dynamic rnn that augments the input at each stage accordingly. By augment it should just calculate the summary vector and concatenate it with the the input to the RNN.

Beautiful, tensorflow defines this function raw_rnn present in tf.nn.raw_rnn that is present just to implement such decoders, need to understand its working. An example implementation is given below:

```
# A simple implementation of `dynamic_rnn` via `raw_rnn` looks like this:

inputs = tf.placeholder(shape=(max_time, batch_size, input_depth),
                      dtype=tf.float32)
sequence_length = tf.placeholder(shape=(batch_size,), dtype=tf.int32)
inputs_ta = tf.TensorArray(dtype=tf.float32, size=max_time)
inputs_ta = inputs_ta.unstack(inputs)
cell = tf.contrib.rnn.LSTMCell(num_units)

def loop_fn(time, cell_output, cell_state, loop_state):
    emit_output = cell_output  # == None for time == 0
    if cell_output is None:  # time == 0
      next_cell_state = cell.zero_state(batch_size, tf.float32)
    else:
      next_cell_state = cell_state
    elements_finished = (time >= sequence_length) # check which all batche
    finished = tf.reduce_all(elements_finished)
    next_input = tf.cond(
                finished, # if all the inputs in the batch are over
                lambda: tf.zeros([batch_size, input_depth], dtype=tf.float
                lambda: inputs_ta.read(time)) # read next timestep
    next_loop_state = None
    return (elements_finished, next_input, next_cell_state,
            emit_output, next_loop_state)

outputs_ta, final_state, _ = raw_rnn(cell, loop_fn)
outputs = outputs_ta.stack()
```

But the catch with this implementation is that the batch_size has to be predefined, this is bad, since we would like to keep a varying batch size. Solution: Create a placeholder for batch_size and pass it == feels hacky (ok for now). True this was very hacky, can't pass batch_size and place it that way since it would be a tensor then. We need batch_size to be an integer. How to do this? Solved: batch size is not required since the output from attention would be NxH and can be concatenated directly

```
batch_size = tf.shape(self.embed_inputs)[0]
....

summary = word_attention(next_cell_state)
...
tf.concat([inputs_ta.read(time), summary], 1) # creates the correct input
```

The problem I am facing right now is that I need a function that implements attention but it will create that op everytime it runs that part of the code? Let it create, would it really create ? test it by doing this, create two tensors and do a = b+c twice and check if two nodes for a are created by displaying the entire graph. Answer: Doesn't create duplicates, so attention can be implemented this way.

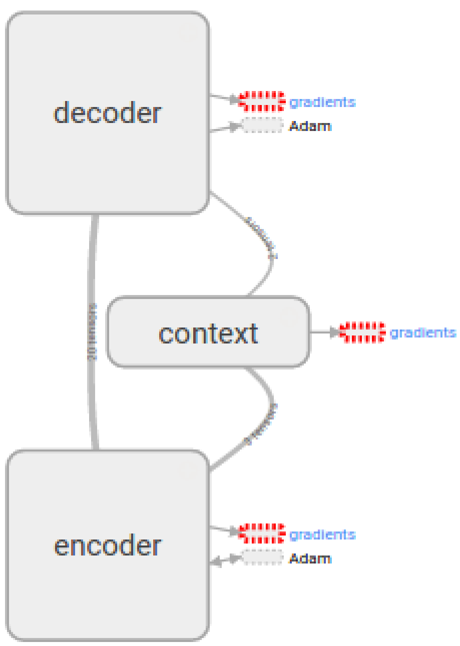
Although previous tf.concat should work in theory, not working, resulting in (?, ?) arrays… why ? Seems to work when I implement in terminal, unable to see where wrong? Soln: The main error was in tf.cond returning multi shape tensors so the entire code flow was getting messed up. Fixed.

The replacement for dynamic_rnn is as below:

```
            batch_size = tf.shape(self.embed_inputs)[0]

            def attention(prev_state, inp):
                # Creates the summary from the context
                # Returns the input concatenated with the summary
                summary = context.last_context
                # TODO implement attention based summary
                return tf.concat([summary, inp], 1)

            def last_context(prev_state, inp):
                # Creates the summary from the context
                # Returns the input concatenated with the summary
                summary = context.last_context
                return tf.concat([summary, inp], 1)


            def loop_fn(time, cell_output, cell_state, loop_state):
                emit_output = cell_output  # == None for time == 0
                if cell_output is None:  # time == 0
                    next_cell_state = self.cell.zero_state(batch_size, tf.
                else:
                    next_cell_state = cell_state
```

```
                elements_finished = (time >= self.decoder_lengths) # check
                finished = tf.reduce_all(elements_finished)
                # This condition ensures that based on the input_length de
                next_input = tf.cond(
                                finished, # if all the inputs in the batch
                                lambda: tf.zeros([batch_size, embed_size +
                                lambda: last_context(next_cell_state, self
                next_loop_state = None
                return (elements_finished, next_input, next_cell_state,
                        emit_output, next_loop_state)

            outputs_ta, final_state, _ = tf.nn.raw_rnn(self.cell, loop_fn)
            self.dec_states = outputs_ta.stack()
```

Visualizing the model on tensorboard we get:



Now the next step would be to implement greedy_decode and beam_decode.

Seq2Seq docs

Seq2Seq docs
vasanth.kalingeri@gmail.com

jekyll
jekyllrb

Write an awesome description for your new site here. You can edit this line in _config.yml. It will appear in your document head meta (for Google search results) and in your feed.xml site description.