```
# Author: Shashi Narayan
# Date: September 2016
# Project: Document Summarization
# H2020 Summa Project
#####################################

"""
Document Summarization Model Utilities
"""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np
import tensorflow as tf
from tensorflow.python.ops import variable_scope
import tensorflow.contrib.seq2seq as seq2seq
#from tensorflow.contrib import rnn
# from tensorflow.python.ops import seq2seq

# from tf.nn import variable_scope
from my_flags import FLAGS
```

```
### Get Variable

def variable_on_cpu(name, shape, initializer, trainable=True):
  """Helper to create a Variable stored on CPU memory.
  Args:
    name: name of the variable
    shape: list of ints
    initializer: initializer for Variable
    trainable: is trainable
  Returns:
    Variable Tensor
  """
  with tf.device('/cpu:0'):
    dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
    var = tf.get_variable(name, shape, initializer=initializer, dtype=dtype, trainable=trainable)
  return var

def get_vocab_embed_variable(vocab_size):
  '''Returns vocab_embed_variable without any local initialization
  '''
  vocab_embed_variable = ""
  if FLAGS.trainable_wordembed:
    vocab_embed_variable = variable_on_cpu("vocab_embed", [vocab_size, FLAGS.wordembed_size], tf.constant_initializer(0), trainable=True)
  else:
    vocab_embed_variable = variable_on_cpu("vocab_embed", [vocab_size, FLAGS.wordembed_size], tf.constant_initializer(0), trainable=False)

  return vocab_embed_variable

def get_lstm_cell():
  """Define LSTM Cell
  """
  single_cell = tf.nn.rnn_cell.BasicLSTMCell(FLAGS.size) if (FLAGS.lstm_cell == "lstm") else tf.nn.rnn_cell.GRUCell(FLAGS.size)
  cell = single_cell
  if FLAGS.num_layers > 1:
    cell = tf.nn.rnn_cell.MultiRNNCell([single_cell] * FLAGS.num_layers)
  return cell
```

```
### Reshaping

def reshape_tensor2list(tensor, n_steps, n_input):
  """Reshape tensor [?, n_steps, n_input] to lists of n_steps items with [?, n_input]
  """
  # Prepare data shape to match `rnn` function requirements
  # Current data input shape (batch_size, n_steps, n_input)
  # Required shape: 'n_steps' tensors list of shape (batch_size, n_input)
  #
  # Permuting batch_size and n_steps
  tensor = tf.transpose(tensor, perm=[1, 0, 2], name='transpose')
  # Reshaping to (n_steps*batch_size, n_input)
  tensor = tf.reshape(tensor, [-1, n_input], name='reshape')
  # Split to get a list of 'n_steps' tensors of shape (batch_size, n_input)
  tensor = tf.split(tensor, n_steps, 0, name='split')
  return tensor

def reshape_list2tensor(listoftensors, n_steps, n_input):
  """Reshape lists of n_steps items with [?, n_input] to tensor [?, n_steps, n_input]
  """
  # Reverse of _reshape_tensor2list
  tensor = tf.concat(axis = 0, values = listoftensors, name="concat") # [n_steps * ?, n_input]
  tensor = tf.reshape(tensor, [n_steps, -1, n_input], name='reshape') # [n_steps, ?, n_input]
  tensor = tf.transpose(tensor, perm=[1, 0, 2], name='transpose') # [?, n_steps, n_input]
  return tensor
```

```
### Convolution, LSTM, RNNs

def multilayer_perceptron(final_output, weights, biases):
  """MLP over output with attention over enc outputs
  Args:
     final_output: [batch_size x 2*size]
  Returns:
     logit:  [batch_size x target_label_size]
  """

  # Layer 1
  layer_1 = tf.add(tf.matmul(final_output, weights["h1"]), biases["b1"])
  layer_1 = tf.nn.relu(layer_1)

  # Layer 2
  layer_2 = tf.add(tf.matmul(layer_1, weights["h2"]), biases["b2"])
  layer_2 = tf.nn.relu(layer_2)

  # output layer
  layer_out = tf.add(tf.matmul(layer_2, weights["out"]), biases["out"])

  return layer_out
```

```
def conv1d_layer_sentence_representation(sent_wordembeddings):
  """Apply mulitple conv1d filters to extract sentence respresentations
  Args:
  sent_wordembeddings: [None, max_sent_length, wordembed_size]
  Returns:
  sent_representations: [None, sentembed_size]
  """

  representation_from_filters = []

  output_channel = 0
  if FLAGS.handle_filter_output == "sum":
    output_channel = FLAGS.sentembed_size
  else: # concat
    output_channel = FLAGS.sentembed_size / FLAGS.max_filter_length
    if (output_channel *  FLAGS.max_filter_length != FLAGS.sentembed_size):
      print("Error: Make sure (output_channel *  FLAGS.max_filter_length) is equal to FLAGS.sentembed_size.")
      exit(0)

  for filterwidth in range(1,FLAGS.max_filter_length+1):
    # print(filterwidth)

    with tf.variable_scope("Conv1D_%d"%filterwidth) as scope:

      # Convolution
      conv_filter = variable_on_cpu("conv_filter_%d" % filterwidth, [filterwidth, FLAGS.wordembed_size, output_channel],
tf.truncated_normal_initializer())
      # print(conv_filter.name, conv_filter.get_shape())
      conv = tf.nn.conv1d(sent_wordembeddings, conv_filter, 1, padding='VALID') # [None, out_width=(max_sent_length-(filterwidth-1)), output_channel]
      conv_biases = variable_on_cpu("conv_biases_%d" % filterwidth, [output_channel], tf.constant_initializer(0.0))
      pre_activation = tf.nn.bias_add(conv, conv_biases)
      conv = tf.nn.relu(pre_activation) #  [None, out_width, output_channel]
      # print(conv.name, conv.get_shape())



      # Max pool: Reshape conv to use max_pool
```

```
      conv_reshaped = tf.expand_dims(conv, 1) # [None, out_height:1, out_width, output_channel]
      # print(conv_reshaped.name, conv_reshaped.get_shape())
      out_height = conv_reshaped.get_shape()[1].value
      out_width = conv_reshaped.get_shape()[2].value
      # print(out_height,out_width)
      maxpool = tf.nn.max_pool(conv_reshaped, [1,out_height,out_width,1], [1,1,1,1], padding='VALID') # [None, 1, 1, output_channel]
      # print(maxpool.name, maxpool.get_shape())

      # Local Response Normalization
      maxpool_norm = tf.nn.lrn(maxpool, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75) # Settings from cifar10
      # print(maxpool_norm.name, maxpool_norm.get_shape())

      # Get back to original dimension
      maxpool_sqz = tf.squeeze(maxpool_norm, [1,2]) # [None, output_channel]
      # print(maxpool_sqz.name, maxpool_sqz.get_shape())

    representation_from_filters.append(maxpool_sqz)
    # print(representation_from_filters)

  final_representation = []
  with tf.variable_scope("FinalOut") as scope:
    if FLAGS.handle_filter_output == "sum":
      final_representation = tf.add_n(representation_from_filters)
    else:
      final_representation = tf.concat(axis = 1, values = representation_from_filters)

  return final_representation




def simple_rnn(rnn_input, initial_state=None):
```

```
  """Implements Simple RNN
  Args:
  rnn_input: List of tensors of sizes [-1, sentembed_size]
  Returns:
  encoder_outputs, encoder_state
  """
  # Setup cell
  cell_enc = get_lstm_cell()

  # Setup RNNs
  dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
  rnn_outputs, rnn_state = tf.contrib.rnn.static_rnn(cell_enc, rnn_input, dtype=dtype, initial_state=initial_state)
  # print(rnn_outputs)
  # print(rnn_state)

  return rnn_outputs, rnn_state
```

```
def simple_attentional_rnn(rnn_input, attention_state_list, initial_state=None):
```

```
  """Implements Simple RNN
  Args:
  rnn_input: List of tensors of sizes [-1, sentembed_size]
  attention_state_list: List of tensors of sizes [-1, sentembed_size]
  Returns:
  outputs, state
  """

  # Reshape attention_state_list to tensor
  attention_states = reshape_list2tensor(attention_state_list, len(attention_state_list), FLAGS.sentembed_size)

  # Setup cell
  cell = get_lstm_cell()

  # Setup attentional RNNs
  dtype = tf.float16 if FLAGS.use_fp16 else tf.float32

  # if initial_state == None:
  #   batch_size = tf.shape(rnn_input[0])[0]
  #   initial_state = cell.zero_state(batch_size, dtype)

  rnn_outputs, rnn_state = seq2seq.attention_decoder(rnn_input, initial_state, attention_states, cell,
                                                     output_size=None, num_heads=1, loop_function=None, dtype=dtype,
                                                     scope=None, initial_state_attention=False)
  # print(rnn_outputs)
  # print(rnn_state)
  return rnn_outputs, rnn_state



### Special decoders
```

```
def jporg_attentional_seqrnn_decoder(sents_ext, encoder_outputs, encoder_state, sents_labels, weights, biases):
  """
  Implements JP's special decoder: attention over encoder
  """

  # Setup cell
  cell_ext = get_lstm_cell()

  # Define Sequential Decoder
  with variable_scope.variable_scope("JP_Decoder"):
    state = encoder_state
    extractor_logits = []
    extractor_outputs = []
    prev = None
    for i, inp in enumerate(sents_ext):
      if prev is not None:
        with variable_scope.variable_scope("loop_function"):
          inp = _loop_function(inp, extractor_logits[-1], sents_labels[i-1])
      if i > 0:
        variable_scope.get_variable_scope().reuse_variables()
      # Create Cell
      output, state = cell_ext(inp, state)
      prev = output

      # Convert output to logit
      with variable_scope.variable_scope("mlp"):
        combined_output = [] # batch_size, 2*size
        if FLAGS.doc_encoder_reverse:
          combined_output = tf.concat(axis = 1, values = [output, encoder_outputs[(FLAGS.max_doc_length - 1) - i]])
        else:
          combined_output = tf.concat(axis = 1, values = [output, encoder_outputs[i]])

        logit = multilayer_perceptron(combined_output, weights, biases)

      extractor_logits.append(logit)
      extractor_outputs.append(combined_output)
```

```
  return extractor_outputs, extractor_logits

### Private Functions

def _loop_function(current_inp, ext_logits, gold_logits):
  """ Update current input wrt previous logits
  Args:
  current_inp: [batch_size x sentence_embedding_size]
  ext_logits: [batch_size x target_label_size] [1, 0]
  gold_logits: [batch_size x target_label_size]
  Returns:
  updated_inp: [batch_size x sentence_embedding_size]
  """

  prev_logits = gold_logits
  if not FLAGS.authorise_gold_label:
    prev_logits = ext_logits
    prev_logits = tf.nn.softmax(prev_logits) # [batch_size x target_label_size]

  prev_logits = tf.split(1, FLAGS.target_label_size, prev_logits) # [[batch_size], [batch_size], ...]
  prev_weight_one = prev_logits[0]

  updated_inp = tf.mul(current_inp, prev_weight_one)
  # print(updated_inp)

  return updated_inp



### SoftMax and Predictions
```

```
def convert_logits_to_softmax(batch_logits, session=None):
  """ Convert logits to probabilities
  batch_logits: [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  """
  # Convert logits [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size] to probabilities
  batch_logits = tf.reshape(batch_logits, [-1, FLAGS.target_label_size])
  batch_softmax_logits = tf.nn.softmax(batch_logits)
  batch_softmax_logits = tf.reshape(batch_softmax_logits, [-1, FLAGS.max_doc_length, FLAGS.target_label_size])
  # Convert back to numpy array
  batch_softmax_logits = batch_softmax_logits.eval(session=session)
  return batch_softmax_logits

def predict_topranked(batch_softmax_logits, batch_weights, batch_filenames):
  """ Predict top ranked outputs: cnn:3, dm:4
  batch_softmax_logits: Numpy Array [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  batch_weights: Numpy Array [batch_size, FLAGS.max_doc_length]
  batch_filenames: String [batch_size]
  Return:
  batch_predicted_labels: [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  """

  def process_to_chop_pad(orgids, requiredsize):
    if (len(orgids) >= requiredsize):
      return orgids[:requiredsize]
    else:
      padids = [0] * (requiredsize - len(orgids))
      return (orgids + padids)

  batch_size = batch_softmax_logits.shape[0]




  # Numpy dtype
```

```
dtype = np.float16 if FLAGS.use_fp16 else np.float32

batch_predicted_labels = np.empty((batch_size, FLAGS.max_doc_length, FLAGS.target_label_size), dtype=dtype)

for batch_idx in range(batch_size):
  softmax_logits = batch_softmax_logits[batch_idx]
  weights = process_to_chop_pad(batch_weights[batch_idx], FLAGS.max_doc_length)
  filename = batch_filenames[batch_idx]

  # Find top scoring sentence to consider for summary, if score is same, select sentences with low indices
  oneprob_sentidx = {}
  for sentidx in range(FLAGS.max_doc_length):
    prob = softmax_logits[sentidx][0] # probability of predicting one
    weight = weights[sentidx]
    if weight == 1:
      if prob not in oneprob_sentidx:
        oneprob_sentidx[prob] = [sentidx]
      else:
        oneprob_sentidx[prob].append(sentidx)
    else:
      break
  oneprob_keys = oneprob_sentidx.keys()
  oneprob_keys.sort(reverse=True)

  # Rank sentences with scores: if same score lower ones ranked first
  sentindices = []
  for oneprob in oneprob_keys:
    sent_withsamescore = oneprob_sentidx[oneprob]
    sent_withsamescore.sort()
    sentindices += sent_withsamescore




  # Select Top Sentences : CNN-3 and DM-4
```

```
    final_sentences = []
    if filename.startswith("cnn-"):
      final_sentences = sentindices[:3]
    elif filename.startswith("dailymail-"):
      final_sentences = sentindices[:4]
    else:
      print(filename)
      print("Filename does not have cnn or dailymail in it.")
      exit(0)

    # Final Labels
    labels_vecs = [[1, 0] if (sentidx in final_sentences) else [0, 1] for sentidx in range(FLAGS.max_doc_length)]
    batch_predicted_labels[batch_idx] = np.array(labels_vecs[:], dtype=dtype)

  return batch_predicted_labels

def predict_toprankedthree(batch_softmax_logits, batch_weights):
  """ Convert logits to probabilities
  batch_softmax_logits: Numpy Array [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  batch_weights: Numpy Array [batch_size, FLAGS.max_doc_length], it may not be [batch_size, FLAGS.max_doc_length] called for validation and test sets,
not padded.
  Return:
  batch_predicted_labels: [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  """

  def process_to_chop_pad(orgids, requiredsize):
    if (len(orgids) >= requiredsize):
      return orgids[:requiredsize]
    else:
      padids = [0] * (requiredsize - len(orgids))
      return (orgids + padids)

  batch_size = batch_softmax_logits.shape[0]



  # Numpy dtype
```

```
dtype = np.float16 if FLAGS.use_fp16 else np.float32

batch_predicted_labels = np.empty((batch_size, FLAGS.max_doc_length, FLAGS.target_label_size), dtype=dtype)

for batch_idx in range(batch_size):
  softmax_logits = batch_softmax_logits[batch_idx]
  weights = process_to_chop_pad(batch_weights[batch_idx], FLAGS.max_doc_length)

  # Find top three scoring sentence to consider for summary, if score is same, select sentences with low indices
  oneprob_sentidx = {}
  for sentidx in range(FLAGS.max_doc_length):
    prob = softmax_logits[sentidx][0] # probability of predicting one
    weight = weights[sentidx]
    if weight == 1:
      if prob not in oneprob_sentidx:
        oneprob_sentidx[prob] = [sentidx]
      else:
        oneprob_sentidx[prob].append(sentidx)
    else:
      break
  oneprob_keys = oneprob_sentidx.keys()
  oneprob_keys.sort(reverse=True)
  # Rank sentences with scores: if same score lower ones ranked first
  sentindices = []
  for oneprob in oneprob_keys:
    sent_withsamescore = oneprob_sentidx[oneprob]
    sent_withsamescore.sort()
    sentindices += sent_withsamescore

  # Select Top 3
  final_sentences = sentindices[:3]




  # Final Labels
  labels_vecs = [[1, 0] if (sentidx in final_sentences) else [0, 1] for sentidx in range(FLAGS.max_doc_length)]
```

```
    batch_predicted_labels[batch_idx] = np.array(labels_vecs[:], dtype=dtype)

  return batch_predicted_labels

def sample_three_forsummary(batch_softmax_logits):
  """ Sample three ones to select in the summary
  batch_softmax_logits: Numpy Array [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  Return:
  batch_predicted_labels: [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  """

  batch_size = batch_softmax_logits.shape[0]

  # Numpy dtype
  dtype = np.float16 if FLAGS.use_fp16 else np.float32

  batch_sampled_labels = np.empty((batch_size, FLAGS.max_doc_length, FLAGS.target_label_size), dtype=dtype)

  for batch_idx in range(batch_size):
    softmax_logits = batch_softmax_logits[batch_idx] # [FLAGS.max_doc_length, FLAGS.target_label_size]

    # Collect probabilities for predicting one for a sentence
    sentence_ids = range(FLAGS.max_doc_length)
    sentence_oneprobs = [softmax_logits[sentidx][0] for sentidx in sentence_ids]
    normalized_sentence_oneprobs =  [item/sum(sentence_oneprobs) for item in sentence_oneprobs]

    # Sample three sentences to select for summary from this distribution
    final_sentences = np.random.choice(sentence_ids, p=normalized_sentence_oneprobs, size=3, replace=False)

    # Final Labels
    labels_vecs = [[1, 0] if (sentidx in final_sentences) else [0, 1] for sentidx in range(FLAGS.max_doc_length)]
    batch_sampled_labels[batch_idx] = np.array(labels_vecs[:], dtype=dtype)

  return batch_sampled_labels

def smaple_with_numpy_random_choice(sentence_ids, normalized_sentence_oneprobs, no_ones_tosample):
  sampled_final_sentences = np.random.choice(sentence_ids, p=normalized_sentence_oneprobs, size=no_ones_tosample, replace=False)
```

```
  sampled_final_sentences.sort()
  return sampled_final_sentences


def multisample_three_forsummary(batch_softmax_logits, batch_gold_label, batch_weight):
  """ Sample three ones to select in the summary: Mix from gold and sampled, one sample always include gold
  batch_softmax_logits: Numpy Array [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  batch_gold_label: Numpy Array [batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
  batch_weight: Numpy Array [batch_size, FLAGS.max_doc_length]
  Return:
  # batch_gold_sampled_label_multisample: [batch_size, FLAGS.num_sample_rollout, FLAGS.max_doc_length, FLAGS.target_label_size]
  batch_gold_sampled_labelstr_multisample: [batch_size, FLAGS.num_sample_rollout]
  """

  # Start Sampling for each document
  batch_size = batch_softmax_logits.shape[0]
  dtype = np.float16 if FLAGS.use_fp16 else np.float32
  # batch_gold_sampled_label_multisample = np.empty((batch_size, FLAGS.num_sample_rollout, FLAGS.max_doc_length, FLAGS.target_label_size), dtype=dtype)
  batch_gold_sampled_labelstr_multisample = np.empty((batch_size, FLAGS.num_sample_rollout), dtype="S20")

  # Number of ones to sample (this is required is a document has less than three sentences)
  batch_gold_label_one, _ = np.split(batch_gold_label, [1], axis=2) # [batch_size, max_doc_length, 1]
  # print(batch_gold_label_one[0])
  batch_gold_label_tosample_one = np.squeeze(batch_gold_label_one, axis=2) # [batch_size, max_doc_length]
  # print(batch_gold_label_tosample_one[0])
  batch_gold_label_tosample_count = np.sum(batch_gold_label_tosample_one , axis=1) # [batch_size]
  # print(batch_gold_label_tosample_count[0])




  # Probabilities used to Sample
  batch_oneprob_usetosample, _ = np.split(batch_softmax_logits, [1], axis=2) # [batch_size, max_doc_length, 1]
```

```
  # print(batch_oneprob_usetosample[0])
  batch_oneprob_usetosample = np.squeeze(batch_oneprob_usetosample, axis=2) # [batch_size, max_doc_length]
  # print(batch_oneprob_usetosample[0])

  for batch_idx in range(batch_size):

    # Always keep the oracle sample
    # batch_gold_sampled_label_multisample[batch_idx][0] = np.array(batch_gold_label[batch_idx][:], dtype=dtype)
    one_labels = []
    for sentidx in range(FLAGS.max_doc_length):
      if int(batch_gold_label[batch_idx][sentidx][0]) == 1:
        one_labels.append(str(sentidx))
    batch_gold_sampled_labelstr_multisample[batch_idx][0] = "-".join(one_labels)

    # Rest: Sample (FLAGS.num_sample_rollout - 1) times
    sentence_oneprobs = batch_oneprob_usetosample[batch_idx] # [max_doc_length]
    # print(sentence_oneprobs)
    sentence_ids = range(len(sentence_oneprobs)) # [max_doc_length]

    #  Make sure that it is not sampled from out of the document,
    #  consider weight while normalising, if w = 0, p = exact 0,
    #  if w is all zero, no_ones_tosample will be 0 and we wont be
    #  here in the first place, and at the same time sum = 1

    weight_samplepart = batch_weight[batch_idx] # [max_doc_length]
    # print(weight_samplepart)

    # Smooth normalization, weight considered. Nonzero will alwyas be >= no_ones_tosample, because for them w = 1
    # Get sum: this will never be zero as w is never all 0
    l1_norm_sum = sum(np.multiply(sentence_oneprobs, weight_samplepart)) + (0.000000000001*sum(weight_samplepart))
    normalized_sentence_oneprobs =  [((item_prob+0.000000000001)*item_weight)/l1_norm_sum for item_prob, item_weight in zip(sentence_oneprobs,
weight_samplepart)]
    # print(normalized_sentence_oneprobs)

    # Number of ones to sample
    no_ones_tosample = int(batch_gold_label_tosample_count[batch_idx])
    # print(no_ones_tosample)
```

```
    # Start sampling (FLAGS.num_sample_rollout - 1) times
    for rollout_idx in range(1, FLAGS.num_sample_rollout):
      sampled_final_sentences_sorted = smaple_with_numpy_random_choice(sentence_ids, normalized_sentence_oneprobs, no_ones_tosample)

      # # Final Labels # This step here, will couse the following loop for the same sample, does not ignore duplicates or take adv of pool
      # sampled_labels_vecs = [[1, 0] if (sentidx in sampled_final_sentences) else [0, 1] for sentidx in sentence_ids]  # [max_doc_length,
target_label_size]
      # # Store
      # batch_gold_sampled_label_multisample[batch_idx][rollout_idx] = np.array(sampled_labels_vecs[:], dtype=dtype)
      batch_gold_sampled_labelstr_multisample[batch_idx][rollout_idx] = "-".join([str(sentidx) for sentidx in sampled_final_sentences_sorted])

  return batch_gold_sampled_labelstr_multisample
  # return batch_gold_sampled_label_multisample, batch_gold_sampled_labelstr_multisample
```