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

"""
Document Summarization Modules and Models
"""

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.python.ops import seq2seq
from tensorflow.python.ops import math_ops

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

```
### Various types of extractor

def sentence_extractor_nonseqrnn_noatt(sents_ext, encoder_state):
    """Implements Sentence Extractor: No attention and non-sequential RNN
    Args:
    sents_ext: Embedding of sentences to label for extraction
    encoder_state: encoder_state
    Returns:
    extractor output and logits
    """
    # Define Variables
    weight = variable_on_cpu('weight', [FLAGS.size, FLAGS.target_label_size], tf.random_normal_initializer())
    bias = variable_on_cpu('bias', [FLAGS.target_label_size], tf.random_normal_initializer())

    # Get RNN output
    rnn_extractor_output, _ = simple_rnn(sents_ext, initial_state=encoder_state)

    with variable_scope.variable_scope("Reshape-Out"):
        rnn_extractor_output =  reshape_list2tensor(rnn_extractor_output, FLAGS.max_doc_length, FLAGS.size)

        # Get Final logits without softmax
        extractor_output_forlogits = tf.reshape(rnn_extractor_output, [-1, FLAGS.size])
        logits = tf.matmul(extractor_output_forlogits, weight) + bias
        # logits: [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
        logits = tf.reshape(logits, [-1, FLAGS.max_doc_length, FLAGS.target_label_size])
    return rnn_extractor_output, logits
```

```
def sentence_extractor_nonseqrnn_titimgatt(sents_ext, encoder_state, titleimages):
    """Implements Sentence Extractor: Non-sequential RNN with attention over title-images
    Args:
    sents_ext: Embedding of sentences to label for extraction
    encoder_state: encoder_state
    titleimages: Embeddings of title and images in the document
    Returns:
    extractor output and logits
    """

    # Define Variables
    weight = variable_on_cpu('weight', [FLAGS.size, FLAGS.target_label_size], tf.random_normal_initializer())
    bias = variable_on_cpu('bias', [FLAGS.target_label_size], tf.random_normal_initializer())

    # Get RNN output
    rnn_extractor_output, _ = simple_attentional_rnn(sents_ext, titleimages, initial_state=encoder_state)

    with variable_scope.variable_scope("Reshape-Out"):
      rnn_extractor_output = reshape_list2tensor(rnn_extractor_output, FLAGS.max_doc_length, FLAGS.size)

      # Get Final logits without softmax
      extractor_output_forlogits = tf.reshape(rnn_extractor_output, [-1, FLAGS.size])
      logits = tf.matmul(extractor_output_forlogits, weight) + bias
      # logits: [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
      logits = tf.reshape(logits, [-1, FLAGS.max_doc_length, FLAGS.target_label_size])
    return rnn_extractor_output, logits
```

```
def sentence_extractor_seqrnn_docatt(sents_ext, encoder_outputs, encoder_state, sents_labels):
    """Implements Sentence Extractor: Sequential RNN with attention over sentences during encoding
    Args:
    sents_ext: Embedding of sentences to label for extraction
    encoder_outputs, encoder_state
    sents_labels: Gold sent labels for training
    Returns:
    extractor output and logits
    """
    # Define MLP Variables
    weights = {
      'h1': variable_on_cpu('weight_1', [2*FLAGS.size, FLAGS.size], tf.random_normal_initializer()),
      'h2': variable_on_cpu('weight_2', [FLAGS.size, FLAGS.size], tf.random_normal_initializer()),
      'out': variable_on_cpu('weight_out', [FLAGS.size, FLAGS.target_label_size], tf.random_normal_initializer())
      }
    biases = {
      'b1': variable_on_cpu('bias_1', [FLAGS.size], tf.random_normal_initializer()),
      'b2': variable_on_cpu('bias_2', [FLAGS.size], tf.random_normal_initializer()),
      'out': variable_on_cpu('bias_out', [FLAGS.target_label_size], tf.random_normal_initializer())
      }

    # Shift sents_ext for RNN
    with variable_scope.variable_scope("Shift-SentExt"):
        # Create embeddings for special symbol (lets assume all 0) and put in the front by shifting by one
        special_tensor = tf.zeros_like(sents_ext[0]) #  tf.ones_like(sents_ext[0])
        sents_ext_shifted = [special_tensor] + sents_ext[:-1]

    # Reshape sents_labels for RNN (Only used for cross entropy training)
    with variable_scope.variable_scope("Reshape-Label"):
        # only used for training
        sents_labels = reshape_tensor2list(sents_labels, FLAGS.max_doc_length, FLAGS.target_label_size)

    # Define Sequential Decoder
    extractor_outputs, logits = jporg_attentional_seqrnn_decoder(sents_ext_shifted, encoder_outputs, encoder_state, sents_labels, weights, biases)
```

```
# Final logits without softmax
with variable_scope.variable_scope("Reshape-Out"):
    logits = reshape_list2tensor(logits, FLAGS.max_doc_length, FLAGS.target_label_size)
    extractor_outputs = reshape_list2tensor(extractor_outputs, FLAGS.max_doc_length, 2*FLAGS.size)

return extractor_outputs, logits
```

```
def policy_network(vocab_embed_variable, document_placeholder, label_placeholder):
    """Build the policy core network.
    Args:
    vocab_embed_variable: [vocab_size, FLAGS.wordembed_size], embeddings without PAD and UNK
    document_placeholder: [None,(FLAGS.max_doc_length + FLAGS.max_title_length + FLAGS.max_image_length), FLAGS.max_sent_length]
    label_placeholder: Gold label [None, FLAGS.max_doc_length, FLAGS.target_label_size], only used during cross entropy training of JP's model.
    Returns:
    Outputs of sentence extractor and logits without softmax
    """

    with tf.variable_scope('PolicyNetwork') as scope:

        ### Full Word embedding Lookup Variable
        # PADDING embedding non-trainable
        pad_embed_variable = variable_on_cpu("pad_embed", [1, FLAGS.wordembed_size], tf.constant_initializer(0), trainable=False)
        # UNK embedding trainable
        unk_embed_variable = variable_on_cpu("unk_embed", [1, FLAGS.wordembed_size], tf.constant_initializer(0), trainable=True)
        # Get fullvocab_embed_variable
        fullvocab_embed_variable = tf.concat(axis = 0, values = [pad_embed_variable, unk_embed_variable, vocab_embed_variable])
        # print(fullvocab_embed_variable)

        ### Lookup layer
        with tf.variable_scope('Lookup') as scope:
            document_placeholder_flat = tf.reshape(document_placeholder, [-1])
            document_word_embedding = tf.nn.embedding_lookup(fullvocab_embed_variable, document_placeholder_flat, name="Lookup")
            document_word_embedding = tf.reshape(document_word_embedding, [-1, (FLAGS.max_doc_length + FLAGS.max_title_length + FLAGS.max_image_length),
                                                                    FLAGS.max_sent_length, FLAGS.wordembed_size])
            # print(document_word_embedding)

        ### Convolution Layer
        with tf.variable_scope('ConvLayer') as scope:
            document_word_embedding = tf.reshape(document_word_embedding, [-1, FLAGS.max_sent_length, FLAGS.wordembed_size])
            document_sent_embedding = conv1d_layer_sentence_representation(document_word_embedding) # [None, sentembed_size]
            document_sent_embedding = tf.reshape(document_sent_embedding, [-1, (FLAGS.max_doc_length + FLAGS.max_title_length + FLAGS.max_image_length),
                                                                    FLAGS.sentembed_size])
            # print(document_sent_embedding)
        ### Reshape Tensor to List [-1, (max_doc_length+max_title_length+max_image_length), sentembed_size] -> List of [-1, sentembed_size]
        with variable_scope.variable_scope("ReshapeDoc_TensorToList"):
            document_sent_embedding = reshape_tensor2list(document_sent_embedding, (FLAGS.max_doc_length + FLAGS.max_title_length + FLAGS.max_image_length), FLAGS.sentembed_size)
            # print(document_sent_embedding)

        # document_sents_enc
        document_sents_enc = document_sent_embedding[:FLAGS.max_doc_length]
        if FLAGS.doc_encoder_reverse:
            document_sents_enc = document_sents_enc[::-1]

        # document_sents_ext
        document_sents_ext = document_sent_embedding[:FLAGS.max_doc_length]

        # document_sents_titimg
        document_sents_titimg = document_sent_embedding[FLAGS.max_doc_length:]

        ### Document Encoder
        with tf.variable_scope('DocEnc') as scope:
            encoder_outputs, encoder_state = simple_rnn(document_sents_enc)
        ### Sentence Label Extractor
        with tf.variable_scope('SentExt') as scope:
            if (FLAGS.attend_encoder) and (len(document_sents_titimg) != 0):
                # Multiple decoder
                print("Multiple decoder is not implement yet.")
                exit(0)
                # # Decoder to attend captions
                # attendtitimg_extractor_output, _ = simple_attentional_rnn(document_sents_ext, document_sents_titimg, initial_state=encoder_state)
                # # Attend previous decoder
                # logits = sentence_extractor_seqrnn_docatt(document_sents_ext, attendtitimg_extractor_output, encoder_state, label_placeholder)

            elif (not FLAGS.attend_encoder) and (len(document_sents_titimg) != 0):
                # Attend only titimages during decoding
                extractor_output, logits = sentence_extractor_nonseqrnn_titimgatt(document_sents_ext, encoder_state, document_sents_titimg)

            elif (FLAGS.attend_encoder) and (len(document_sents_titimg) == 0):
                # JP model: attend encoder
                extractor_outputs, logits = sentence_extractor_seqrnn_docatt(document_sents_ext, encoder_outputs, encoder_state, label_placeholder)

            else:
                # Attend nothing
                extractor_output, logits = sentence_extractor_nonseqrnn_noatt(document_sents_ext, encoder_state)

    # print(extractor_output)
    # print(logits)
    return extractor_output, logits
```

```
def baseline_future_reward_estimator(extractor_output):
    """Implements linear regression to estimate future rewards
    Args:
    extractor_output: [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.size or 2*FLAGS.size]
    Output:
    rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
    """

    with tf.variable_scope('FutureRewardEstimator') as scope:

        last_size = extractor_output.get_shape()[2].value

        # Define Variables
        weight = variable_on_cpu('weight', [last_size, 1], tf.random_normal_initializer())
        bias = variable_on_cpu('bias', [1], tf.random_normal_initializer())

        extractor_output_forreward = tf.reshape(extractor_output, [-1, last_size])
        future_rewards = tf.matmul(extractor_output_forreward, weight) + bias
        # future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length, 1]
        future_rewards = tf.reshape(future_rewards, [-1, FLAGS.max_doc_length, 1])
        future_rewards = tf.squeeze(future_rewards)
    return future_rewards
```

```
def baseline_single_future_reward_estimator(extractor_output):
    """Implements linear regression to estimate future rewards for whole document
    Args:
    extractor_output: [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.size or 2*FLAGS.size]
    Output:
    rewards: [FLAGS.batch_size]
    """

    with tf.variable_scope('FutureRewardEstimator') as scope:

        last_size = extractor_output.get_shape()[2].value

        # Define Variables
        weight = variable_on_cpu('weight', [FLAGS.max_doc_length*last_size, 1], tf.random_normal_initializer())
        bias = variable_on_cpu('bias', [1], tf.random_normal_initializer())

        extractor_output_forreward = tf.reshape(extractor_output, [-1, FLAGS.max_doc_length*last_size]) # [FLAGS.batch_size,
FLAGS.max_doc_length*(FLAGS.size or 2*FLAGS.size)]
        future_rewards = tf.matmul(extractor_output_forreward, weight) + bias # [FLAGS.batch_size, 1]
        # future_rewards: [FLAGS.batch_size, 1]
        future_rewards = tf.squeeze(future_rewards) # [FLAGS.batch_size]
    return future_rewards
```

```
### Loss Functions

def mean_square_loss_doclevel(future_rewards, actual_reward):
    """Implements mean_square_loss for futute reward prediction
    args:
    future_rewards: [FLAGS.batch_size]
    actual_reward: [FLAGS.batch_size]
    Output
    Float Value
    """
    with tf.variable_scope('MeanSquareLoss') as scope:
        sq_loss = tf.square(future_rewards - actual_reward) # [FLAGS.batch_size]

        mean_sq_loss = tf.reduce_mean(sq_loss)

        tf.add_to_collection('mean_square_loss', mean_sq_loss)

    return mean_sq_loss
```

```python
def mean_square_loss(future_rewards, actual_reward, weights):
    """Implements mean_square_loss for futute reward prediction
    args:
    future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
    actual_reward: [FLAGS.batch_size]
    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
    Output
    Float Value
    """
    with tf.variable_scope('MeanSquareLoss') as scope:
        actual_reward = tf.expand_dims(actual_reward, 1) # [FLAGS.batch_size, 1]
        sq_loss = tf.square(future_rewards - actual_reward) # [FLAGS.batch_size, FLAGS.max_doc_length]

        mean_sq_loss = 0
        if FLAGS.weighted_loss:
            sq_loss = tf.mul(sq_loss, weights)
            sq_loss_sum = tf.reduce_sum(sq_loss)
            valid_sentences = tf.reduce_sum(weights)
            mean_sq_loss = sq_loss_sum / valid_sentences
        else:
            mean_sq_loss = tf.reduce_mean(sq_loss)

        tf.add_to_collection('mean_square_loss', mean_sq_loss)

    return mean_sq_loss
```

```python
def cross_entropy_loss(logits, labels, weights):
    """Estimate cost of predictions
    Add summary for "cost" and "cost/avg".
    Args:
    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    labels: Sentence extraction gold levels [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
    Returns:
    Cross-entropy Cost
    """
    with tf.variable_scope('CrossEntropyLoss') as scope:
        # Reshape logits and labels to match the requirement of softmax_cross_entropy_with_logits
        logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
        labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, labels) # [FLAGS.batch_size*FLAGS.max_doc_length]
        cross_entropy = tf.reshape(cross_entropy, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
        if FLAGS.weighted_loss:
            cross_entropy = tf.mul(cross_entropy, weights)

        # Cross entroy / document
        cross_entropy = tf.reduce_sum(cross_entropy, reduction_indices=1) # [FLAGS.batch_size]
        cross_entropy_mean = tf.reduce_mean(cross_entropy, name='crossentropy')

        # ## Cross entroy / sentence
        # cross_entropy_sum = tf.reduce_sum(cross_entropy)
        # valid_sentences = tf.reduce_sum(weights)
        # cross_entropy_mean = cross_entropy_sum / valid_sentences

        # cross_entropy = -tf.reduce_sum(labels * tf.log(logits), reduction_indices=1)
        # cross_entropy_mean = tf.reduce_mean(cross_entropy, name='crossentropy')

        tf.add_to_collection('cross_entropy_loss', cross_entropy_mean)
        # # # The total loss is defined as the cross entropy loss plus all of
        # # # the weight decay terms (L2 loss).
        # # return tf.add_n(tf.get_collection('losses'), name='total_loss')
    return cross_entropy_mean
```

```
def predict_labels(logits):
    """ Predict self labels
    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    Return [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    """
    with tf.variable_scope('PredictLabels') as scope:
        # Reshape logits for argmax and argmin
        logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
        # Get labels predicted using these logits
        logits_argmax = tf.argmax(logits, 1) # [FLAGS.batch_size*FLAGS.max_doc_length]
        logits_argmax = tf.reshape(logits_argmax, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
        logits_argmax = tf.expand_dims(logits_argmax, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]

        logits_argmin = tf.argmin(logits, 1) # [FLAGS.batch_size*FLAGS.max_doc_length]
        logits_argmin = tf.reshape(logits_argmin, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
        logits_argmin = tf.expand_dims(logits_argmin, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]

        # Convert argmin and argmax to labels, works only if FLAGS.target_label_size = 2
        labels = tf.concat(2, [logits_argmin, logits_argmax]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
        dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
        labels = tf.cast(labels, dtype)

        return labels
```

```
def estimate_ltheta_ot(logits, labels, future_rewards, actual_rewards, weights):
    """
    Args:
    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    labels: Label placeholdr for self prediction [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
    actual_reward: [FLAGS.batch_size]
    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
    Returns:
    [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    """
    with tf.variable_scope('LTheta_Ot') as scope:
        # Get Reward Weights: External reward - Predicted reward
        actual_rewards = tf.tile(actual_rewards, [FLAGS.max_doc_length]) # [FLAGS.batch_size * FLAGS.max_doc_length] , [a,b] * 3 = [a, b, a, b, a, b]
        actual_rewards = tf.reshape(actual_rewards, [FLAGS.max_doc_length, -1]) # [FLAGS.max_doc_length, FLAGS.batch_size], # [[a,b], [a,b], [a,b]]
        actual_rewards = tf.transpose(actual_rewards) # [FLAGS.batch_size, FLAGS.max_doc_length] # [[a,a,a], [b,b,b]]

        diff_act_pred = actual_rewards - future_rewards # [FLAGS.batch_size, FLAGS.max_doc_length]
        diff_act_pred = tf.expand_dims(diff_act_pred, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
        # Convert (FLAGS.target_label_size = 2)
        diff_act_pred = tf.concat(2, [diff_act_pred, diff_act_pred]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]

        # Reshape logits and labels to match the requirement of softmax_cross_entropy_with_logits
        logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
        logits = tf.nn.softmax(logits)
        logits = tf.reshape(logits, [-1, FLAGS.max_doc_length, FLAGS.target_label_size]) # [FLAGS.batch_size, FLAGS.max_doc_length,
FLAGS.target_label_size]

        # Get the difference
        diff_logits_indicator = logits - labels # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]



        # Multiply with reward
        d_ltheta_ot = tf.mul(diff_act_pred, diff_logits_indicator) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]

        # Multiply with weight
        weights = tf.expand_dims(weights, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
        weights = tf.concat(2, [weights, weights]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
        d_ltheta_ot = tf.mul(d_ltheta_ot, weights) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]

        return d_ltheta_ot

# def estimate_ltheta_ot_mixer(logits, labels_gold, labels_pred, future_rewards, actual_rewards, weights, annealing_step):
#     """
#     Args:
#     logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     labels_gold: Label placeholdr for gold labels [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     labels_pred: Label placeholdr for self prediction [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
#     actual_reward: [FLAGS.batch_size]
#     weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
#     annealing_step: [1], single value but in tensor form
#     Returns:
#     [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     """
#     with tf.variable_scope('LTheta_Ot_Mixer') as scope:
#         print(annealing_step)

#         policygradloss_length = tf.reduce_sum(annealing_step) * FLAGS.annealing_step_delta
#         crossentryloss_length = FLAGS.max_doc_length - policygradloss_length
#        # Reshape logits and partition
#        logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
#        logits = tf.nn.softmax(logits)
#        logits = tf.reshape(logits, [-1, FLAGS.max_doc_length, FLAGS.target_label_size]) # [FLAGS.batch_size, FLAGS.max_doc_length,
FLAGS.target_label_size]
#        logits_list = reshape_tensor2list(logits, FLAGS.max_doc_length, FLAGS.target_label_size)
#        logits_ce_gold_list = logits_list[0:crossentryloss_length]
#        logits_ce_gold = reshape_list2tensor(logits_ce_gold_list, crossentryloss_length, FLAGS.target_label_size) # [FLAGS.batch_size,
crossentryloss_length, FLAGS.target_label_size]
#        logits_reward_list = logits_list[crossentryloss_length:]
#        logits_reward = reshape_list2tensor(logits_reward_list, policygradloss_length, FLAGS.target_label_size) # [FLAGS.batch_size,
policygradloss_length, FLAGS.target_label_size]

#        # Crossentropy loss with gold labels: partition gold_labels
#        labels_gold_list = reshape_tensor2list(labels_gold, FLAGS.max_doc_length, FLAGS.target_label_size)
#        labels_gold_used_list = labels_gold_list[0:crossentryloss_length]
#        labels_gold_used = reshape_list2tensor(labels_gold_used_list, crossentryloss_length, FLAGS.target_label_size) # [FLAGS.batch_size,
crossentryloss_length, FLAGS.target_label_size]

#        # d_ltheta_ot : cross entropy
#        diff_logits_goldlabels = logits_ce_gold - labels_gold_used # [FLAGS.batch_size, crossentryloss_length, FLAGS.target_label_size]

#        # Policy gradient for rest

#        # Get Reward Weights: External reward - Predicted reward
#        actual_rewards = tf.tile(actual_rewards, [FLAGS.max_doc_length]) # [FLAGS.batch_size * FLAGS.max_doc_length] , [a,b] * 3 = [a, b, a, b, a, b]
#        actual_rewards = tf.reshape(actual_rewards, [FLAGS.max_doc_length, -1]) # [FLAGS.max_doc_length, FLAGS.batch_size], # [[a,b], [a,b], [a,b]]
#        actual_rewards = tf.transpose(actual_rewards) # [FLAGS.batch_size, FLAGS.max_doc_length] # [[a,a,a], [b,b,b]]
#        diff_act_pred = actual_rewards - future_rewards # [FLAGS.batch_size, FLAGS.max_doc_length]
#        diff_act_pred = tf.expand_dims(diff_act_pred, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
#        # Convert (FLAGS.target_label_size = 2)
#        diff_act_pred = tf.concat(2, [diff_act_pred, diff_act_pred]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#        # Get used reward diff
#        diff_act_pred_list = reshape_tensor2list(diff_act_pred, FLAGS.max_doc_length, FLAGS.target_label_size)
#        diff_reward_act_pred_used_list = diff_act_pred_list[crossentryloss_length:]
#        diff_reward_act_pred_used = reshape_list2tensor(diff_reward_act_pred_used_list, policygradloss_length, FLAGS.target_label_size) # [FLAGS.batch_size, policygradloss_length, FLAGS.target_label_size]

#        # Partition predicted labels
#        labels_pred_list = reshape_tensor2list(labels_pred, FLAGS.max_doc_length, FLAGS.target_label_size)
#        labels_pred_used_list = labels_pred_list[crossentryloss_length:]
#        labels_pred_used = reshape_list2tensor(labels_pred_used_list, policygradloss_length, FLAGS.target_label_size) # [FLAGS.batch_size, policygradloss_length, FLAGS.target_label_size]

#        # d_ltheta_ot : reward weighted
#        diff_logits_predlabels = logits_reward - labels_pred_used # [FLAGS.batch_size, policygradloss_length, FLAGS.target_label_size]
#        # Multiply with reward
#        reward_weighted_diff_logits_predlabels = tf.mul(diff_reward_act_pred_used, diff_logits_predlabels) # [FLAGS.batch_size, policygradloss_length, FLAGS.target_label_size]

#        # Concat both part
#        d_ltheta_ot_mixer = tf.concat(1, [diff_logits_goldlabels, reward_weighted_diff_logits_predlabels]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]

#        # Multiply with weight
#        weights = tf.expand_dims(weights, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
#        weights = tf.concat(2, [weights, weights]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#        d_ltheta_ot_mixer = tf.mul(d_ltheta_ot_mixer, weights) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]

#        return d_ltheta_ot_mixer
```

```
def reward_weighted_cross_entropy_loss_multisample(logits, labels, actual_rewards, weights):
    """Estimate cost of predictions
    Add summary for "cost" and "cost/avg".
    Args:
    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    labels: Label placeholdr for multiple sampled prediction [FLAGS.batch_size, 1, FLAGS.max_doc_length, FLAGS.target_label_size]
    actual_rewards: [FLAGS.batch_size, 1]
    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
    Returns:
    Cross-entropy Cost
    """

    with tf.variable_scope('RWCELossMultiSample') as scope:
        # Expand logits and weights for roll outs
        logits_temp = tf.expand_dims(logits, 1) # [FLAGS.batch_size, 1, FLAGS.max_doc_length, FLAGS.target_label_size]
        weights_temp = tf.expand_dims(weights, 1) # [FLAGS.batch_size, 1, FLAGS.max_doc_length]
        logits_expanded = logits_temp
        weights_expanded = weights_temp
        # for ridx in range(1,FLAGS.num_sample_rollout):
        #     logits_expanded = tf.concat(1, [logits_expanded, logits_temp]) # [FLAGS.batch_size, n++, FLAGS.max_doc_length, FLAGS.target_label_size]
        #     weights_expanded = tf.concat(1, [weights_expanded, weights_temp]) # [FLAGS.batch_size, n++, FLAGS.max_doc_length]

        # Reshape logits and labels to match the requirement of softmax_cross_entropy_with_logits
        logits_expanded = tf.reshape(logits_expanded, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*1*FLAGS.max_doc_length, FLAGS.target_label_size]
        labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*1*FLAGS.max_doc_length, FLAGS.target_label_size]

        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits = logits_expanded, labels = labels) # [FLAGS.batch_size*1*FLAGS.max_doc_length]
        cross_entropy = tf.reshape(cross_entropy, [-1, 1, FLAGS.max_doc_length])  # [FLAGS.batch_size, 1, FLAGS.max_doc_length]
        if FLAGS.weighted_loss:
            cross_entropy = tf.multiply(cross_entropy, weights_expanded) # [FLAGS.batch_size, 1, FLAGS.max_doc_length]

        # Reshape actual rewards
        actual_rewards = tf.reshape(actual_rewards, [-1]) # [FLAGS.batch_size*1]
        # [[a, b], [c, d], [e, f]] 3x2 => [a, b, c, d, e, f] [6]
        actual_rewards = tf.tile(actual_rewards, [FLAGS.max_doc_length]) # [FLAGS.batch_size * 1 * FLAGS.max_doc_length]
        # [a, b, c, d, e, f] * 2 = [a, b, c, d, e, f, a, b, c, d, e, f] [12]
        actual_rewards = tf.reshape(actual_rewards, [FLAGS.max_doc_length, -1]) # [FLAGS.max_doc_length, FLAGS.batch_size*1]
```

```
# [[a, b, c, d, e, f], [a, b, c, d, e, f]] [2, 6]
actual_rewards = tf.transpose(actual_rewards) # [FLAGS.batch_size*1, FLAGS.max_doc_length]
# [[a,a], [b,b], [c,c], [d,d], [e,e], [f,f]] [6 x 2]
actual_rewards = tf.reshape(actual_rewards, [-1, 1, FLAGS.max_doc_length]) # [FLAGS.batch_size, 1, FLAGS.max_doc_length],
# [[[a,a], [b,b]], [[c,c], [d,d]], [[e,e], [f,f]]] [3 x 2 x 2]

# Multiply with reward
reward_weighted_cross_entropy = tf.multiply(cross_entropy, actual_rewards) # [FLAGS.batch_size, 1, FLAGS.max_doc_length]

# Cross entroy / sample / document
reward_weighted_cross_entropy = tf.reduce_sum(reward_weighted_cross_entropy, reduction_indices=2) # [FLAGS.batch_size, 1]
reward_weighted_cross_entropy_mean = tf.reduce_mean(reward_weighted_cross_entropy, name='rewardweightedcemultisample')

tf.add_to_collection('reward_cross_entropy_loss_multisample', reward_weighted_cross_entropy_mean)

return reward_weighted_cross_entropy_mean
```

```python
def reward_weighted_cross_entropy_loss(logits, labels, actual_rewards, weights):
    """Estimate cost of predictions
    Add summary for "cost" and "cost/avg".
    Args:
    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    labels: Label placeholdr for self prediction [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    actual_reward: [FLAGS.batch_size]
    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
    Returns:
    Cross-entropy Cost
    """

    with tf.variable_scope('RewardWeightedCrossEntropyLoss') as scope:

        # Reshape logits and labels to match the requirement of softmax_cross_entropy_with_logits
        logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
        labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, labels) # [FLAGS.batch_size*FLAGS.max_doc_length]
        cross_entropy = tf.reshape(cross_entropy, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
        if FLAGS.weighted_loss:
            cross_entropy = tf.mul(cross_entropy, weights) # [FLAGS.batch_size, FLAGS.max_doc_length]

        # Reshape actual rewards
        actual_rewards = tf.tile(actual_rewards, [FLAGS.max_doc_length]) # [FLAGS.batch_size * FLAGS.max_doc_length] , [a,b] * 3 = [a, b, a, b, a, b]
        actual_rewards = tf.reshape(actual_rewards, [FLAGS.max_doc_length, -1]) # [FLAGS.max_doc_length, FLAGS.batch_size], # [[a,b], [a,b], [a,b]]
        actual_rewards = tf.transpose(actual_rewards) # [FLAGS.batch_size, FLAGS.max_doc_length] # [[a,a,a], [b,b,b]]

        # Multiply with reward
        reward_weighted_cross_entropy = tf.mul(cross_entropy, actual_rewards) # [FLAGS.batch_size, FLAGS.max_doc_length]

        # Cross entroy / document
        reward_weighted_cross_entropy = tf.reduce_sum(reward_weighted_cross_entropy, reduction_indices=1) # [FLAGS.batch_size]
        reward_weighted_cross_entropy_mean = tf.reduce_mean(reward_weighted_cross_entropy, name='rewardweightedcrossentropy')

        tf.add_to_collection('reward_cross_entropy_loss', reward_weighted_cross_entropy_mean)

        return reward_weighted_cross_entropy_mean
```

```
# def reward_weighted_cross_entropy_loss(logits, labels, future_rewards, actual_rewards, weights):
#     """Estimate cost of predictions
#     Add summary for "cost" and "cost/avg".
#     Args:
#     logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     labels: Label placeholdr for self prediction [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
#     actual_reward: [FLAGS.batch_size]
#     weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
#     Returns:
#     Cross-entropy Cost
#     """

#     with tf.variable_scope('RewardWeightedCrossEntropyLoss') as scope:
#         # Get Reward Weights: External reward - Predicted reward
#         actual_rewards = tf.tile(actual_rewards, [FLAGS.max_doc_length]) # [FLAGS.batch_size * FLAGS.max_doc_length] , [a,b] * 3 = [a, b, a, b, a, b]
#         actual_rewards = tf.reshape(actual_rewards, [FLAGS.max_doc_length, -1]) # [FLAGS.max_doc_length, FLAGS.batch_size], # [[a,b], [a,b], [a,b]]
#         actual_rewards = tf.transpose(actual_rewards) # [FLAGS.batch_size, FLAGS.max_doc_length] # [[a,a,a], [b,b,b]]

#         # Error: actual_rewards = tf.reshape(tf.tile(actual_rewards, [FLAGS.max_doc_length]),[-1, FLAGS.max_doc_length]) # [FLAGS.batch_size,
FLAGS.max_doc_length]

#         diff_act_pred = future_rewards - actual_rewards # actual_rewards - future_rewards # [FLAGS.batch_size, FLAGS.max_doc_length]

#         # Reshape logits and labels to match the requirement of softmax_cross_entropy_with_logits
#         logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
#         labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
#         cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, labels) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         cross_entropy = tf.reshape(cross_entropy, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         if FLAGS.weighted_loss:
#             cross_entropy = tf.mul(cross_entropy, weights) # [FLAGS.batch_size, FLAGS.max_doc_length]

#         # Multiply with reward
#         reward_weighted_cross_entropy = tf.mul(cross_entropy, diff_act_pred) # [FLAGS.batch_size, FLAGS.max_doc_length]

#         # Cross entroy / document
#        reward_weighted_cross_entropy = tf.reduce_sum(reward_weighted_cross_entropy, reduction_indices=1) # [FLAGS.batch_size]
#        reward_weighted_cross_entropy_mean = tf.reduce_mean(reward_weighted_cross_entropy, name='rewardweightedcrossentropy')

#        tf.add_to_collection('reward_cross_entropy_loss', reward_weighted_cross_entropy_mean)

#        return reward_weighted_cross_entropy_mean

# def temp_reward_weighted_cross_entropy_loss(logits, labels, future_rewards, actual_rewards, weights):
#    """Estimate cost of predictions
#    Add summary for "cost" and "cost/avg".
#    Args:
#    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#    labels: Label placeholdr for self prediction [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#    future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
#    actual_reward: [FLAGS.batch_size]
#    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
#    Returns:
#    Cross-entropy Cost
#    """

#    with tf.variable_scope('TempRewardWeightedCrossEntropyLoss') as scope:
#        # Get Reward Weights: External reward - Predicted reward
#        actual_rewards = tf.tile(actual_rewards, [FLAGS.max_doc_length]) # [FLAGS.batch_size * FLAGS.max_doc_length] , [a,b] * 3 = [a, b, a, b, a, b]
#        actual_rewards = tf.reshape(actual_rewards, [FLAGS.max_doc_length, -1]) # [FLAGS.max_doc_length, FLAGS.batch_size], # [[a,b], [a,b], [a,b]]
#        actual_rewards = tf.transpose(actual_rewards) # [FLAGS.batch_size, FLAGS.max_doc_length] # [[a,a,a], [b,b,b]]

#        diff_act_pred = future_rewards - actual_rewards # actual_rewards - future_rewards # [FLAGS.batch_size, FLAGS.max_doc_length]

#        # Reshape logits and labels to match the requirement of softmax_cross_entropy_with_logits
#        logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
#        labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
#        cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, labels) # [FLAGS.batch_size*FLAGS.max_doc_length]
#        cross_entropy = tf.reshape(cross_entropy, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#        if FLAGS.weighted_loss:
#            cross_entropy = tf.mul(cross_entropy, weights) # [FLAGS.batch_size, FLAGS.max_doc_length]

#        # Multiply with reward
#         reward_weighted_cross_entropy = tf.mul(cross_entropy, diff_act_pred) # [FLAGS.batch_size, FLAGS.max_doc_length]

#         # Cross entroy / document
#         reward_weighted_cross_entropy = tf.reduce_sum(reward_weighted_cross_entropy, reduction_indices=1) # [FLAGS.batch_size]
#         reward_weighted_cross_entropy_mean = tf.reduce_mean(reward_weighted_cross_entropy, name='rewardweightedcrossentropy')

#         optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

#         # Compute gradients of policy network
#         policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="PolicyNetwork")
#         # print(policy_network_variables)

#         # Compute gradients of policy network
#         grads_and_vars = optimizer.compute_gradients(reward_weighted_cross_entropy_mean, var_list=policy_network_variables)
#         # print(grads_and_vars)

#         return actual_rewards, cross_entropy, diff_act_pred, reward_weighted_cross_entropy, reward_weighted_cross_entropy_mean, grads_and_vars
# def cross_entropy_loss_selfprediction(logits, weights):
#     """Optimizing expected reward: Weighted cross entropy
#     args:
#     logits: Logits without softmax. [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
#     return:
#     [FLAGS.batch_size, FLAGS.max_doc_length]
#     """
#     with tf.variable_scope('SelfPredCrossEntropyLoss') as scope:
#         # Reshape logits for argmax and argmin
#         logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]

#         # Get labels if predicted using these logits
#         logits_argmax = tf.argmax(logits, 1) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         logits_argmax = tf.reshape(logits_argmax, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         logits_argmax = tf.expand_dims(logits_argmax, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
#         logits_argmin = tf.argmin(logits, 1) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         logits_argmin = tf.reshape(logits_argmin, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         logits_argmin = tf.expand_dims(logits_argmin, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
#         # Convert argmin and argmax to labels, works only if FLAGS.target_label_size = 2
#         labels = tf.concat(2, [logits_argmin, logits_argmax]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#         dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
#         labels = tf.cast(labels, dtype)
#         labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]

#         # softmax_cross_entropy_with_logits
#         cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, labels) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         cross_entropy = tf.reshape(cross_entropy, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         if FLAGS.weighted_loss:
#             cross_entropy = tf.mul(cross_entropy, weights)
#     return cross_entropy
# def weighted_cross_entropy_loss(logits, future_rewards, actual_reward, weights):
#     """Optimizing expected reward: Weighted cross entropy
#     args:
#     logits: Logits without softmax. [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     future_rewards: [FLAGS.batch_size, FLAGS.max_doc_length]
#     actual_reward: [FLAGS.batch_size]
#     weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
#     """

#     with tf.variable_scope('WeightedCrossEntropyLoss') as scope:
#         # Get Weights: External reward - Predicted reward
#         actual_reward = tf.reshape(tf.tile(actual_reward, [FLAGS.max_doc_length]),[-1, FLAGS.max_doc_length]) # [FLAGS.batch_size, FLAGS.max_doc_length]
#         diff_act_pred = future_rewards - actual_reward # actual_reward - future_rewards # [FLAGS.batch_size, FLAGS.max_doc_length]

#         # Reshape logits for argmax and argmin
#         logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]

#         # Get labels if predicted using these logits
#         logits_argmax = tf.argmax(logits, 1) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         logits_argmax = tf.reshape(logits_argmax, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         logits_argmax = tf.expand_dims(logits_argmax, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
#         logits_argmin = tf.argmin(logits, 1) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         logits_argmin = tf.reshape(logits_argmin, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         logits_argmin = tf.expand_dims(logits_argmin, 2) # [FLAGS.batch_size, FLAGS.max_doc_length, 1]
#         # Convert argmin and argmax to labels, works only if FLAGS.target_label_size = 2
#         labels = tf.concat(2, [logits_argmin, logits_argmax]) # [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#         dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
#         labels = tf.cast(labels, dtype)
#         labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]

#         # softmax_cross_entropy_with_logits
#         cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits, labels) # [FLAGS.batch_size*FLAGS.max_doc_length]
#         cross_entropy = tf.reshape(cross_entropy, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
#         if FLAGS.weighted_loss:
#             cross_entropy = tf.mul(cross_entropy, weights)

#         # Multiply with reward
#        cross_entropy = tf.mul(cross_entropy, diff_act_pred)

#        # Cross entroy / document
#        cross_entropy = tf.reduce_sum(cross_entropy, reduction_indices=1) # [FLAGS.batch_size]
#        cross_entropy_mean = tf.reduce_mean(cross_entropy, name='crossentropy')

#        tf.add_to_collection('reward_cross_entropy_loss', cross_entropy_mean)
#        # # # The total loss is defined as the cross entropy loss plus all of
#        # # # the weight decay terms (L2 loss).
#        # # return tf.add_n(tf.get_collection('losses'), name='total_loss')
#    return cross_entropy_mean









































### Training functions
```

```python
def train_cross_entropy_loss(cross_entropy_loss):
    """ Training with Gold Label: Pretraining network to start with a better policy
    Args: cross_entropy_loss
    """
    with tf.variable_scope('TrainCrossEntropyLoss') as scope:

        optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

        # Compute gradients of policy network
        policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="PolicyNetwork")
        # print(policy_network_variables)
        grads_and_vars = optimizer.compute_gradients(cross_entropy_loss, var_list=policy_network_variables)
        # print(grads_and_vars)

        # Apply Gradients
        return optimizer.apply_gradients(grads_and_vars)

def train_meansq_loss(futreward_meansq_loss):
    """ Training with Gold Label: Pretraining network to start with a better policy
    Args: futreward_meansq_loss
    """
    with tf.variable_scope('TrainMeanSqLoss') as scope:
        optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

        # Compute gradients of Future reward estimator
        futreward_estimator_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="FutureRewardEstimator")
        # print(futreward_estimator_variables)
        grads_and_vars = optimizer.compute_gradients(futreward_meansq_loss, var_list=futreward_estimator_variables)
        # print(grads_and_vars)

        # Apply Gradients
        return optimizer.apply_gradients(grads_and_vars)
```

```python
def train_neg_expectedreward(reward_weighted_cross_entropy_loss_multisample):
    """Training with Policy Gradient: Optimizing expected reward
    args:
    reward_weighted_cross_entropy_loss_multisample
    """
    with tf.variable_scope('TrainExpReward') as scope:

        optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

        # Compute gradients of policy network
        policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="PolicyNetwork")
        # print(policy_network_variables)

        # Compute gradients of policy network
        grads_and_vars = optimizer.compute_gradients(reward_weighted_cross_entropy_loss_multisample, var_list=policy_network_variables)
        # print(grads_and_vars)

        # Clip gradient: Pascanu et al. 2013, Exploding gradient problem
        grads_and_vars_capped_norm = [(tf.clip_by_norm(grad, 5.0), var) for grad, var in grads_and_vars]

        # Apply Gradients
        # return optimizer.apply_gradients(grads_and_vars)
        return optimizer.apply_gradients(grads_and_vars_capped_norm)
```

```
# def train_neg_expectedreward(reward_weighted_cross_entropy_loss):
#     """Training with Policy Gradient: Optimizing expected reward
#     args:
#     reward_weighted_cross_entropy_loss
#     """
#     with tf.variable_scope('TrainExpReward') as scope:

#         optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

#         # Compute gradients of policy network
#         policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="PolicyNetwork")
#         # print(policy_network_variables)

#         # Compute gradients of policy network
#         grads_and_vars = optimizer.compute_gradients(reward_weighted_cross_entropy_loss, var_list=policy_network_variables)
#         # print(grads_and_vars)

#         # Clip gradient: Pascanu et al. 2013, Exploding gradient problem
#         grads_and_vars_capped_norm = [(tf.clip_by_norm(grad, 5.0), var) for grad, var in grads_and_vars]

#         # Apply Gradients
#         # return optimizer.apply_gradients(grads_and_vars)
#         return optimizer.apply_gradients(grads_and_vars_capped_norm)
# def train_neg_expectedreward(logits, d_ltheta_ot):
#     """Training with Policy Gradient: Optimizing expected reward
#     args:
#     logits: Logits without softmax. [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     d_ltheta_ot: Placeholder [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
#     """
#     with tf.variable_scope('TrainExpReward') as scope:

#         optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

#         # Modify logits with d_ltheta_ot
#         logits = tf.mul(logits, d_ltheta_ot)

#         # Compute gradients of policy network
#         policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="PolicyNetwork")
#         # print(policy_network_variables)

#         # Compute gradients of policy network
#         grads_and_vars = optimizer.compute_gradients(logits, var_list=policy_network_variables)
#         # print(grads_and_vars)

#         # Clip gradient: Pascanu et al. 2013, Exploding gradient problem
#         grads_and_vars_capped_norm = [(tf.clip_by_norm(grad, 5.0), var) for grad, var in grads_and_vars]

#         # Apply Gradients
#         # return optimizer.apply_gradients(grads_and_vars)
#         return optimizer.apply_gradients(grads_and_vars_capped_norm)

# def temp_train_neg_expectedreward(logits, d_ltheta_ot):
#     with tf.variable_scope('TempTrainExpReward') as scope:

#         optimizer = tf.train.AdamOptimizer(learning_rate=FLAGS.learning_rate, name='adam')

#         # Modify logits with d_ltheta_ot
#         logits = tf.mul(logits, d_ltheta_ot)

#         # Compute gradients of policy network
#        policy_network_variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, scope="PolicyNetwork")
#        # print(policy_network_variables)

#        # Compute gradients of policy network
#        grads_and_vars = optimizer.compute_gradients(logits, var_list=policy_network_variables)

#        grads_and_vars_capped_norm = [(tf.clip_by_norm(grad, 5.0), var) for grad, var in grads_and_vars]

#        grads_and_vars_capped_val = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in grads_and_vars]



#        # tf.clip_by_norm(t, clip_norm, axes=None, name=None)
#        # https://www.tensorflow.org/versions/r0.11/api_docs/python/train/gradient_clipping



#        return grads_and_vars, grads_and_vars_capped_norm, grads_and_vars_capped_val
```

```
### Accuracy Calculations

def accuracy(logits, labels, weights):
  """Estimate accuracy of predictions
  Args:
    logits: Logits from inference(). [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    labels: Sentence extraction gold levels [FLAGS.batch_size, FLAGS.max_doc_length, FLAGS.target_label_size]
    weights: Weights to avoid padded part [FLAGS.batch_size, FLAGS.max_doc_length]
  Returns:
    Accuracy: Estimates average of accuracy for each sentence
  """
  with tf.variable_scope('Accuracy') as scope:
    logits = tf.reshape(logits, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
    labels = tf.reshape(labels, [-1, FLAGS.target_label_size]) # [FLAGS.batch_size*FLAGS.max_doc_length, FLAGS.target_label_size]
    correct_pred = tf.equal(tf.argmax(logits,1), tf.argmax(labels,1)) # [FLAGS.batch_size*FLAGS.max_doc_length]
    correct_pred =  tf.reshape(correct_pred, [-1, FLAGS.max_doc_length])  # [FLAGS.batch_size, FLAGS.max_doc_length]
    correct_pred = tf.cast(correct_pred, tf.float32)
    # Get Accuracy
    accuracy = tf.reduce_mean(correct_pred, name='accuracy')
    if FLAGS.weighted_loss:
      correct_pred = tf.multiply(correct_pred, weights)
      correct_pred = tf.reduce_sum(correct_pred, reduction_indices=1) # [FLAGS.batch_size]
      doc_lengths = tf.reduce_sum(weights, reduction_indices=1) # [FLAGS.batch_size]
      correct_pred_avg = tf.div(correct_pred, doc_lengths)
      accuracy = tf.reduce_mean(correct_pred_avg, name='accuracy')
  return accuracy

# Improve it to show exact accuracy (top three ranked ones), not all.
```