```
import itertools
import random

import numpy as np
import tensorflow as tf
from tensorflow.python.ops.rnn_cell import BasicLSTMCell

from basic.read_data import DataSet
from my.tensorflow import get_initializer
from my.tensorflow.nn import softsel, get_logits, highway_network, multi_conv1d
from my.tensorflow.rnn import bidirectional_dynamic_rnn
from my.tensorflow.rnn_cell import SwitchableDropoutWrapper, AttentionCell


def get_multi_gpu_models(config):
    models = []
    for gpu_idx in range(config.num_gpus):
        with tf.name_scope("model_{}".format(gpu_idx)) as scope, tf.device(
                "/{}:{}".format(config.device_type, gpu_idx)):
            model = Model(config, scope, rep=gpu_idx == 0)
            tf.get_variable_scope().reuse_variables()
            models.append(model)
    return models


class Model(object):
    def __init__(self, config, scope, rep=True):
        """

        :param config:
        :param scope: model_0/
        :param rep: gpu_idx == 0, what does this mean
        """
        self.scope = scope
        self.config = config
        self.global_step = tf.get_variable('global_step', shape=[], dtype='int32',
                                           initializer=tf.constant_initializer(0),

        # Define forward inputs here
        M = config.max_num_sents  # = 1, here, only one sentence
        JX = config.max_sent_size  # the length of a sentence is less than 400 word
        JQ = config.max_ques_size  # the length of a question is less than 30 words
        VW = config.word_vocab_size  # the vocabulary is 3064, why so small
        VC = config.char_vocab_size  # the kinds of char is 330, why it has an embe
        W = config.max_word_size  # the length of each word is less than 16 charact

        self.x = tf.placeholder('int32', [config.batch_size, None, None], name='x')
        # x shape[2] is the max sentence length, shape[1] is the max number of sent
        self.cx = tf.placeholder('int32', [config.batch_size, None, None, W], name=
        self.x_mask = tf.placeholder('bool', [config.batch_size, None, None], name=
        self.q = tf.placeholder('int32', [config.batch_size, None], name='q')  # qu
        # shape[1] is the max questions size, i guess it's a vector of int who is t
        self.cq = tf.placeholder('int32', [config.batch_size, None, W], name='cq')
        self.q_mask = tf.placeholder('bool', [config.batch_size, None], name='q_mas
        self.y = tf.placeholder('bool', [config.batch_size, None, None], name='y')
        self.y2 = tf.placeholder('bool', [config.batch_size, None, None], name='y2'
        self.is_train = tf.placeholder('bool', [], name='is_train')
        self.new_emb_mat = tf.placeholder('float', [None, config.word_emb_size], na

        # Define misc
        self.tensor_dict = {}

        # Forward outputs / loss inputs
        self.logits = None
        self.yp = None
        self.var_list = None

        # Loss outputs
        self.loss = None

        self._build_forward()
        self._build_loss()
        self.var_ema = None
        if rep:
            self._build_var_ema()

        if config.mode == 'train':
            self._build_ema()

        self.summary = tf.summary.merge_all()
        self.summary = tf.summary.merge(tf.get_collection("summaries", scope=self.s

    def _build_forward(self):
        config = self.config

        N = config.batch_size
        M = config.max_num_sents
        JX = config.max_sent_size
        JQ = config.max_ques_size
        VW = config.word_vocab_size
        VC = config.char_vocab_size
        W = config.max_word_size
        d = config.hidden_size

        JX = tf.shape(self.x)[2]  # JX max sentence size, length,
        JQ = tf.shape(self.q)[1]  # JQ max questions size, length, is the
        M = tf.shape(self.x)[1]  # m is the max number of sentences
        dc, dw, dco = config.char_emb_size, config.word_emb_size, config.char_out_s
        # dc = 8, each char will be map to 8-number vector,  "char-level word embed
        with tf.variable_scope("emb"):
            if config.use_char_emb:
                with tf.variable_scope("emb_var"), tf.device("/cpu:0"):
                    char_emb_mat = tf.get_variable("char_emb_mat",
                                                   shape=[VC, dc],
                                                   dtype='float')
                    # 330,8 a matrix for each char to its 8-number vector

                with tf.variable_scope("char"):
                    Acx = tf.nn.embedding_lookup(char_emb_mat,
                                                 self.cx)
                    # [N, M, JX, W, dc] 60,None,None,16,8, batch-size,
                    # N is the number of batch_size
                    # M the max number of sentences
                    # JX is the max sentence length
                    # W is  the max length of a word
                    # dc is the vector for each char
                    # map each char to a vector

                    Acq = tf.nn.embedding_lookup(char_emb_mat,
                                                 self.cq)  # [N, JQ, W, dc]
                    # JQ the max length of question
                    # W the max length of words
                    # mao each char in questiosn to vectors

                    Acx = tf.reshape(Acx, [-1, JX, W, dc])
                    Acq = tf.reshape(Acq, [-1, JQ, W, dc])
                    # max questions size, length, max_word_size(16), char_emb_size(

                    filter_sizes = list(map(int, config.out_channel_dims.split(',')
                    heights = list(map(int, config.filter_heights.split(',')))
                    # so here, there are 100 filters and the size of each filter is
                    # different heights and there are different number of these fil

                    assert sum(filter_sizes) == dco, (filter_sizes, dco)
                    with tf.variable_scope("conv"):
                        xx = multi_conv1d(Acx,
                                          filter_sizes,
                                          heights, "VALID",
                                          self.is_train,
                                          config.keep_prob,
                                          scope="xx")
                        if config.share_cnn_weights:
                            tf.get_variable_scope().reuse_variables()
                            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID",
                                              scope="xx")
                        else:
                            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID",
                                              scope="qq")
                        xx = tf.reshape(xx, [-1, M, JX, dco])
                        qq = tf.reshape(qq, [-1, JQ, dco])  # here, xx and qq are t

            if config.use_word_emb:
                with tf.variable_scope("emb_var"), tf.device("/cpu:0"):
                    if config.mode == 'train':
                        word_emb_mat = tf.get_variable("word_emb_mat", dtype='float
                                                       initializer=get_initializer(
                    else:
                        word_emb_mat = tf.get_variable("word_emb_mat", shape=[VW, d
                    if config.use_glove_for_unk:  # create a new word embedding or
                        word_emb_mat = tf.concat([word_emb_mat, self.new_emb_mat],

                with tf.name_scope("word"):
                    Ax = tf.nn.embedding_lookup(word_emb_mat, self.x)  # [N, M, JX,
                    Aq = tf.nn.embedding_lookup(word_emb_mat, self.q)  # [N, JQ, d]
                    self.tensor_dict['x'] = Ax
                    self.tensor_dict['q'] = Aq
                if config.use_char_emb:
                    xx = tf.concat([xx, Ax], 3)  # [N, M, JX, di]
                    qq = tf.concat([qq, Aq], 2)  # [N, JQ, di]
                else:
                    xx = Ax
                    qq = Aq  # here we used cnn and word embedding represented each
        # so for, xx, (batch_size, sentence#, word#, embedding), qq (batch_size, wo
        # highway network
        if config.highway:
            with tf.variable_scope("highway"):
                xx = highway_network(xx, config.highway_num_layers, True, wd=config
                tf.get_variable_scope().reuse_variables()
                qq = highway_network(qq, config.highway_num_layers, True, wd=config

        self.tensor_dict['xx'] = xx
        self.tensor_dict['qq'] = qq
        # same shape with line 173
        cell = BasicLSTMCell(d, state_is_tuple=True)  # d = 100, hidden state numbe
        d_cell = SwitchableDropoutWrapper(cell, self.is_train, input_keep_prob=conf
        x_len = tf.reduce_sum(tf.cast(self.x_mask, 'int32'), 2)  # [N, M], [60,?]
        q_len = tf.reduce_sum(tf.cast(self.q_mask, 'int32'), 1)  # [N] [60]
        # masks are true and false, here, he sums up those truths,
        with tf.variable_scope("prepro"):
            (fw_u, bw_u), ((_, fw_u_f), (_, bw_u_f)) = bidirectional_dynamic_rnn(d_
                                                                                 dt
                                                                                 sc
            u = tf.concat([fw_u, bw_u], 2)  # (60, ?, 200) |  200 becahse combined
            if config.share_lstm_weights:
                tf.get_variable_scope().reuse_variables()
                (fw_h, bw_h), _ = bidirectional_dynamic_rnn(cell, cell, xx, x_len,
                                                            scope='u1')  # [N, M, J
                h = tf.concat([fw_h, bw_h], 3)  # [N, M, JX, 2d]
            else:
                (fw_h, bw_h), _ = bidirectional_dynamic_rnn(cell, cell, xx, x_len,
                                                            scope='h1')  # [N, M, J
                h = tf.concat([fw_h, bw_h], 3)  # [N, M, JX, 2d]
            self.tensor_dict['u'] = u  # [60, ?, 200] for question
            self.tensor_dict['h'] = h  # [60, ?, ?, 200] for article

        with tf.variable_scope("main"):
            if config.dynamic_att:  # todo what is this dynamic attention.
                p0 = h
                u = tf.reshape(tf.tile(tf.expand_dims(u, 1), [1, M, 1, 1]), [N * M,
                q_mask = tf.reshape(tf.tile(tf.expand_dims(self.q_mask, 1), [1, M,
                first_cell = AttentionCell(cell, u, mask=q_mask, mapper='sim',
                                           input_keep_prob=self.config.input_keep_p
            else:
                p0 = attention_layer(config, self.is_train, h, u, h_mask=self.x_mas
                                     tensor_dict=self.tensor_dict)
                cell2 = BasicLSTMCell(d, state_is_tuple=True)  # d = 100, hidden st
                first_cell = SwitchableDropoutWrapper(cell2, self.is_train, input_k

            (fw_g0, bw_g0), _ = bidirectional_dynamic_rnn(first_cell, first_cell, i
                                                          dtype='float',
                                                          scope='g0')  # [N, M, JX,
            g0 = tf.concat([fw_g0, bw_g0], 3)
            cell3 = BasicLSTMCell(d, state_is_tuple=True)  # d = 100, hidden state
            first_cell3 = SwitchableDropoutWrapper(cell3, self.is_train, input_keep

            (fw_g1, bw_g1), _ = bidirectional_dynamic_rnn(first_cell3, first_cell3,
                                                          scope='g1')  # [N, M, JX,
            g1 = tf.concat([fw_g1, bw_g1], 3)

            logits = get_logits([g1, p0], d, True, wd=config.wd, input_keep_prob=co
                                mask=self.x_mask, is_train=self.is_train, func=conf
            a1i = softsel(tf.reshape(g1, [N, M * JX, 2 * d]), tf.reshape(logits, [N
            a1i = tf.tile(tf.expand_dims(tf.expand_dims(a1i, 1), 1), [1, M, JX, 1])
            cell4 = BasicLSTMCell(d, state_is_tuple=True)  # d = 100, hidden state
            first_cell4 = SwitchableDropoutWrapper(cell4, self.is_train, input_keep

            (fw_g2, bw_g2), _ = bidirectional_dynamic_rnn(first_cell4, first_cell4,
                                                          tf.concat([p0, g1, a1i, g
                                                          x_len, dtype='float', sco
            g2 = tf.concat([fw_g2, bw_g2], 3)
            logits2 = get_logits([g2, p0], d, True, wd=config.wd, input_keep_prob=c
                                 mask=self.x_mask,
                                 is_train=self.is_train, func=config.answer_func, s

            flat_logits = tf.reshape(logits, [-1, M * JX])
            flat_yp = tf.nn.softmax(flat_logits)  # [-1, M*JX]
            yp = tf.reshape(flat_yp, [-1, M, JX])
            flat_logits2 = tf.reshape(logits2, [-1, M * JX])
            flat_yp2 = tf.nn.softmax(flat_logits2)
            yp2 = tf.reshape(flat_yp2, [-1, M, JX])

            self.tensor_dict['g1'] = g1
            self.tensor_dict['g2'] = g2

            self.logits = flat_logits
            self.logits2 = flat_logits2
            self.yp = yp
            self.yp2 = yp2

    def _build_loss(self):
        config = self.config
        JX = tf.shape(self.x)[2]
        M = tf.shape(self.x)[1]
        JQ = tf.shape(self.q)[1]
        loss_mask = tf.reduce_max(tf.cast(self.q_mask, 'float'), 1)
        losses = tf.nn.softmax_cross_entropy_with_logits_v2(
            logits=self.logits, labels=tf.cast(tf.reshape(self.y, [-1, M * JX]), 'f
        ce_loss = tf.reduce_mean(loss_mask * losses)
        tf.add_to_collection('losses', ce_loss)
        ce_loss2 = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
            logits=self.logits2, labels=tf.cast(tf.reshape(self.y2, [-1, M * JX]),
        tf.add_to_collection("losses", ce_loss2)

        self.loss = tf.add_n(tf.get_collection('losses', scope=self.scope), name='l
        tf.summary.scalar(self.loss.op.name, self.loss)
        tf.add_to_collection('ema/scalar', self.loss)
    def _build_ema(self):
        self.ema = tf.train.ExponentialMovingAverage(self.config.decay)
        ema = self.ema
        tensors = tf.get_collection("ema/scalar", scope=self.scope) + tf.get_collec
        ema_op = ema.apply(tensors)
        for var in tf.get_collection("ema/scalar", scope=self.scope):
            ema_var = ema.average(var)
            tf.summary.scalar(ema_var.op.name, ema_var)
        for var in tf.get_collection("ema/vector", scope=self.scope):
            ema_var = ema.average(var)
            tf.summary.histogram(ema_var.op.name, ema_var)

        with tf.control_dependencies([ema_op]):
            self.loss = tf.identity(self.loss)

    def _build_var_ema(self):
        self.var_ema = tf.train.ExponentialMovingAverage(self.config.var_decay)
        ema = self.var_ema
        ema_op = ema.apply(tf.trainable_variables())
        with tf.control_dependencies([ema_op]):
            self.loss = tf.identity(self.loss)

    def get_loss(self):
        return self.loss

    def get_global_step(self):
        return self.global_step

    def get_var_list(self):
        return self.var_list

    def get_feed_dict(self, batch, is_train, supervised=True):
        assert isinstance(batch, DataSet)
        config = self.config
        N, M, JX, JQ, VW, VC, d, W = \
            config.batch_size, config.max_num_sents, config.max_sent_size, \
            config.max_ques_size, config.word_vocab_size, config.char_vocab_size, c
        feed_dict = {}

        if config.len_opt:
            """
            Note that this optimization results in variable GPU RAM usage (i.e. can
            First test without len_opt and make sure no OOM, and use len_opt
            """
            if sum(len(sent) for para in batch.data['x'] for sent in para) == 0:
                new_JX = 1
            else:
                new_JX = max(len(sent) for para in batch.data['x'] for sent in para
            JX = min(JX, new_JX)

            if sum(len(ques) for ques in batch.data['q']) == 0:
                new_JQ = 1
            else:
                new_JQ = max(len(ques) for ques in batch.data['q'])
            JQ = min(JQ, new_JQ)
        if config.cpu_opt:
            if sum(len(para) for para in batch.data['x']) == 0:
                new_M = 1
            else:
                new_M = max(len(para) for para in batch.data['x'])
            M = min(M, new_M)

        x = np.zeros([N, M, JX], dtype='int32')
        cx = np.zeros([N, M, JX, W], dtype='int32')
        x_mask = np.zeros([N, M, JX], dtype='bool')
        q = np.zeros([N, JQ], dtype='int32')
        cq = np.zeros([N, JQ, W], dtype='int32')
        q_mask = np.zeros([N, JQ], dtype='bool')

        feed_dict[self.x] = x
        feed_dict[self.x_mask] = x_mask
        feed_dict[self.cx] = cx
        feed_dict[self.q] = q
        feed_dict[self.cq] = cq
        feed_dict[self.q_mask] = q_mask
        feed_dict[self.is_train] = is_train
        if config.use_glove_for_unk:
            feed_dict[self.new_emb_mat] = batch.shared['new_emb_mat']

        X = batch.data['x']
        CX = batch.data['cx']

        if supervised:
            y = np.zeros([N, M, JX], dtype='bool')
            y2 = np.zeros([N, M, JX], dtype='bool')
            feed_dict[self.y] = y
            feed_dict[self.y2] = y2

            for i, (xi, cxi, yi) in enumerate(zip(X, CX, batch.data['y'])):
                start_idx, stop_idx = random.choice(yi)
                j, k = start_idx
                j2, k2 = stop_idx
                if config.single:
                    X[i] = [xi[j]]
                    CX[i] = [cxi[j]]
                    j, j2 = 0, 0
                if config.squash:
                    offset = sum(map(len, xi[:j]))
                    j, k = 0, k + offset
                    offset = sum(map(len, xi[:j2]))
                    j2, k2 = 0, k2 + offset
                y[i, j, k] = True
                y2[i, j2, k2 - 1] = True

        def _get_word(word):
            d = batch.shared['word2idx']
            for each in (word, word.lower(), word.capitalize(), word.upper()):
                if each in d:
                    return d[each]
            if config.use_glove_for_unk:
                d2 = batch.shared['new_word2idx']
                for each in (word, word.lower(), word.capitalize(), word.upper()):
                    if each in d2:
                        return d2[each] + len(d)
            return 1

        def _get_char(char):
            d = batch.shared['char2idx']
            if char in d:
                return d[char]
            return 1

        for i, xi in enumerate(X):
            if self.config.squash:
                xi = [list(itertools.chain(*xi))]
            for j, xij in enumerate(xi):
                if j == config.max_num_sents:
                    break
                for k, xijk in enumerate(xij):
                    if k == config.max_sent_size:
                        break
                    each = _get_word(xijk)
                    assert isinstance(each, int), each
                    x[i, j, k] = each
                    x_mask[i, j, k] = True

        for i, cxi in enumerate(CX):
            if self.config.squash:
                cxi = [list(itertools.chain(*cxi))]
            for j, cxij in enumerate(cxi):
                if j == config.max_num_sents:
                    break
                for k, cxijk in enumerate(cxij):
                    if k == config.max_sent_size:
                        break
                    for l, cxijkl in enumerate(cxijk):
                        if l == config.max_word_size:
                            break
                        cx[i, j, k, l] = _get_char(cxijkl)

        for i, qi in enumerate(batch.data['q']):
            for j, qij in enumerate(qi):
                q[i, j] = _get_word(qij)
                q_mask[i, j] = True

        for i, cqi in enumerate(batch.data['cq']):
            for j, cqij in enumerate(cqi):
                for k, cqijk in enumerate(cqij):
                    cq[i, j, k] = _get_char(cqijk)
                    if k + 1 == config.max_word_size:
                        break

        return feed_dict


def bi_attention(config, is_train, h, u, h_mask=None, u_mask=None, scope=None, tens
    with tf.variable_scope(scope or "bi_attention"):
        JX = tf.shape(h)[2]
        M = tf.shape(h)[1]
        JQ = tf.shape(u)[1]
        h_aug = tf.tile(tf.expand_dims(h, 3), [1, 1, 1, JQ, 1]) # tf expand dims 3
        u_aug = tf.tile(tf.expand_dims(tf.expand_dims(u, 1), 1), [1, M, JX, 1, 1])
        if h_mask is None:
            hu_mask = None
        else:
            h_mask_aug = tf.tile(tf.expand_dims(h_mask, 3), [1, 1, 1, JQ])
            u_mask_aug = tf.tile(tf.expand_dims(tf.expand_dims(u_mask, 1), 1), [1,
            hu_mask = h_mask_aug & u_mask_aug
        # equation 1.
        u_logits = get_logits([h_aug, u_aug], None, True, wd=config.wd, mask=hu_mas
                              is_train=is_train, func=config.logit_func, scope='u_l
        u_a = softsel(u_aug, u_logits)  # [N, M, JX, d]
        h_a = softsel(h, tf.reduce_max(u_logits, 3))  # [N, M, d]
        h_a = tf.tile(tf.expand_dims(h_a, 2), [1, 1, JX, 1])

        if tensor_dict is not None:
            a_u = tf.nn.softmax(u_logits)  # [N, M, JX, JQ]
            a_h = tf.nn.softmax(tf.reduce_max(u_logits, 3))
            tensor_dict['a_u'] = a_u
            tensor_dict['a_h'] = a_h
            variables = tf.get_collection(tf.GraphKeys.VARIABLES, scope=tf.get_vari
            for var in variables:
                tensor_dict[var.name] = var

        return u_a, h_a


def attention_layer(config, is_train, h, u, h_mask=None, u_mask=None, scope=None, t
    with tf.variable_scope(scope or "attention_layer"):
        JX = tf.shape(h)[2]  # the length of sentence
        M = tf.shape(h)[1]  # the max number of sentences
        JQ = tf.shape(u)[1]  # the length of question
        if config.q2c_att or config.c2q_att:
            u_a, h_a = bi_attention(config, is_train, h, u, h_mask=h_mask, u_mask=u
        if not config.c2q_att:
            u_a = tf.tile(tf.expand_dims(tf.expand_dims(tf.reduce_mean(u, 1), 1), 1
        if config.q2c_att:
            p0 = tf.concat([h, u_a, h * u_a, h * h_a], 3)
        else:
            p0 = tf.concat([h, u_a, h * u_a], 3)
        return p0
```