Vishoo Verma 2K20/EP/120

Medical Report Generation

Code Flow :

First, get post process data(I have done it)

get 'data/data_entry.json', it is the report sentences.

get 'data/train_split.json' and 'data/test_split.json', it is the ids for train/val/test. get 'data/vocabulary.json', it is the vocabulary extracted from report.

Second, get TFRecord files

get 'data/train.tfrecord' and 'data/test.tfrecord

```
$ python datasets.py
```

Third, go train

you can train directly

```
$ python train.py
```

you can see the train process

```
$ cd ./data
$ tensorboard --logdir='summary'
```

Data Set Flow :

It contains two separate folders named “train” and “validation.” Each of the sub folders contain two types of images, "Frontal" and "Lateral". Every image has a uid corresponding to it. The correspondence is established using the csv sheet named "projections.csv". We have reports corresponding to each "uid". The actual reports for every "uid" is available in "reports.csv". The data is arranged in the following pattern, as shown in Figure 1.

The dataset contains 1916 frontal and lateral images along with corresponding reports as training data and 639 frontal and lateral images along with corresponding reports as validation data.

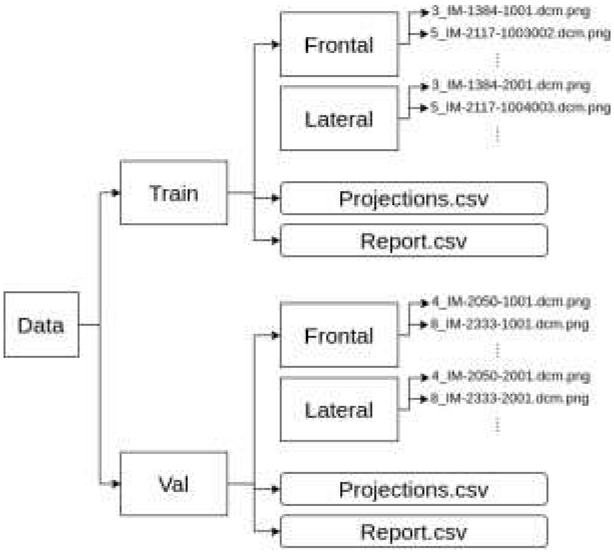


Figure 1: Dataset arrangement

Implementation:

```
import tensorflow as tf
from nets import inception

class Model(object):
    def __init__(self, config, is_training=True, batch_size=26): self.config =
        config
        self.is_training = is_training
        self.batch_size = batch_size
        self.images_frontal = tf.placeholder(dtype=tf.float32, shape=[self.batch_size,
config.image_size, config.image_size, 3])
        self.images_lateral = tf.placeholder(dtype=tf.float32, shape=[self.batch_size,
config.image_size, config.image_size, 3])
        self.sentences = tf.placeholder(dtype=tf.int32, shape=[self.batch_size,
config.max_sentence_num * config.max_sentence_length])
        self.masks = tf.placeholder(dtype=tf.float32, shape=[self.batch_size,
config.max_sentence_num * config.max_sentence_length])

        self.build_cnn()
        self.build_rnn()
        self.build_metrics()
        if is_training:
            self.build_optimizer()
            self.build_summary()

    def build_cnn(self):
        net_f, _ = inception.inception_v3(self.images_frontal, trainable=True,
is_training=True, add_summaries=False, scope='FrontalInceptionV3')
        net_l, _ = inception.inception_v3(self.images_lateral, trainable=True,
is_training=True, add_summaries=False, scope='LateralInceptionV3')

        self.visual_feats = tf.concat([net_f, net_l], axis=1) # [batch_size, print('cnn
4096]
        built.')

    def build_rnn(self):
        with tf.variable_scope("word_embedding"):
            word_embedding_matrix = tf.get_variable(
                                        name='weights',
                                        shape=[self.config.vocabulary_size,
 self.config.word_embedding_size],
                                        trainable=True)

        # 1. build hierarchical rnn SentRNN =
                   tf.nn.rnn_cell.LSTMCell(
            name = 'sent_rnn',
            num_units=self.config.rnn_units)
        if self.is_training:
            SentRNN = tf.nn.rnn_cell.DropoutWrapper( SentRNN,
                input_keep_prob = 1.0 - self.config.rnn_dropout_rate, output_keep_prob
                = 1.0 - self.config.rnn_dropout_rate, state_keep_prob = 1.0 -
                self.config.rnn_dropout_rate)
        WordRNN = tf.nn.rnn_cell.LSTMCell(
            name='word_rnn',
            num_units=self.config.rnn_units)
        if self.is_training:
            WordRNN = tf.nn.rnn_cell.DropoutWrapper( WordRNN,
                input_keep_prob=1.0 - self.config.rnn_dropout_rate,
                output_keep_prob=1.0 - self.config.rnn_dropout_rate,
                state_keep_prob=1.0 - self.config.rnn_dropout_rate)


        # 2. init Sent RNN
        with tf.variable_scope('sent_rnn_initialize'): context
            = tf.layers.dropout(self.visual_feats,
rate=self.config.dropout_rate, training=self.is_training, name='drop_v') init_c
            = tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_c')
            init_h = tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_h')

            SentRNN_last_state = init_c, init_h

        # 3. generate sentence one by one
        predicts = []
        cross_entropies = []
        corrects = []
        for sent_id in range(self.config.max_sentence_num): #
            3.1 sent rnn
            with tf.variable_scope('sent_rnn'):
                SentRNN_output, SentRNN_state = SentRNN(self.visual_feats,
SentRNN_last_state)
                SentRNN_last_state = SentRNN_state

            # 3.2 init Word RNN
            with tf.variable_scope('word_rnn_initialize'):
                context = tf.layers.dropout(SentRNN_output,
rate=self.config.dropout_rate, training=self.is_training, name='drop_s') init_c =
                tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_c')
                init_h = tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_h')

                WordRNN_last_state = init_c, init_h
                WordRNN_last_word = tf.zeros([self.batch_size], tf.int32)

            # 3.3 generate word one by one
            for word_id in range(self.config.max_sentence_length):
                with tf.variable_scope("word_embedding"):
                    word_embedding =
tf.nn.embedding_lookup(word_embedding_matrix, WordRNN_last_word)

                with tf.variable_scope('word_rnn'):
                    WordRNN_output, WordRNN_state = WordRNN(word_embedding,
WordRNN_last_state)
                    WordRNN_last_state = WordRNN_state

                with tf.variable_scope('decode'):
                    WordRNN_output = tf.layers.dropout(WordRNN_output,
rate=self.config.dropout_rate, training=self.is_training, name='drop_d')
                    logits = tf.layers.dense(WordRNN_output,
units=self.config.vocabulary_size, use_bias=True, name='fc_d')
                    predict = tf.argmax(logits, 1)
                    predicts.append(predict)

                tf.get_variable_scope().reuse_variables() if
                self.is_training:
                    WordRNN_last_word = self.sentences[:,
sent_id*self.config.max_sentence_length + word_id]
                else:
                    WordRNN_last_word = predict

                # compute cross entropy loss
                cross_entropy =
tf.nn.sparse_softmax_cross_entropy_with_logits(labels=self.sentences[:,
sent_id*self.config.max_sentence_length + word_id], logits=logits)
                masked_cross_entropy = cross_entropy * self.masks[:,
sent_id*self.config.max_sentence_length + word_id]
                cross_entropies.append(masked_cross_entropy)

                # compute acc
                ground_truth = tf.cast(self.sentences[:,
sent_id*self.config.max_sentence_length + word_id], tf.int64)
                correct = tf.where( tf.equal(predict,
                    ground_truth),
                    tf.cast(self.masks[:, sent_id*self.config.max_sentence_length
+ word_id], tf.float32),
                    tf.cast(tf.zeros_like(predict), tf.float32)
                )
                corrects.append(correct)


        self.predicts = predicts self.cross_entropies
        = cross_entropies self.corrects = corrects

        print('rnn built.')

    def build_metrics(self):
        corrects = tf.stack(self.corrects, axis=1)
        self.accuracy = tf.reduce_sum(corrects) / tf.reduce_sum(self.masks)

        cross_entropies = tf.stack(self.cross_entropies, axis=1)
        self.cross_entropy_loss = tf.reduce_sum(cross_entropies) /
tf.reduce_sum(self.masks)

        self.reg_loss = tf.losses.get_regularization_loss()
        self.loss = self.cross_entropy_loss + self.reg_loss

        print('metrics built.')

    def build_optimizer(self):
        self.global_step = tf.Variable(0, name='global_step', trainable=False)
        learning_rate = tf.constant(self.config.learning_rate) def
        _learning_rate_decay_fn(learning_rate, global_step):
            return tf.train.exponential_decay(
                learning_rate=learning_rate,
                global_step=global_step,
                decay_steps=self.config.decay_iters,
                decay_rate=self.config.decay_rate,
                staircase=True
            )

        learning_rate_decay_fn = _learning_rate_decay_fn
        with tf.variable_scope('optimizer', reuse=tf.AUTO_REUSE):
            optimizer = tf.train.AdamOptimizer(
                learning_rate=learning_rate,
                beta1=0.9,
                beta2=0.999,
                epsilon=1e-8
            )

            self.step_op = tf.contrib.layers.optimize_loss(
                loss=self.loss, global_step=self.global_step,
                learning_rate=learning_rate,
                optimizer=optimizer,
                clip_gradients=5.0,
                learning_rate_decay_fn=learning_rate_decay_fn, #
                variables=other_var_list
            )
        print('optimizer built.')

    def build_summary(self):
        with tf.name_scope("metrics"):
            tf.summary.scalar('cross entropy loss', self.cross_entropy_loss)
            tf.summary.scalar('reg loss', self.reg_loss) tf.summary.scalar('acc',
            self.accuracy)
        self.summary = tf.summary.merge_all() print('summary
        built.')
```

cnn_vis_sem_rnn_model.py

```
import tensorflow as tf
from nets import inception

class Model(object):
    def __init__(self, config, is_training=True, batch_size=26): self.config =
        config
        self.is_training = is_training
        self.batch_size = batch_size
        self.images_frontal = tf.placeholder(dtype=tf.float32, shape=[self.batch_size,
config.image_size, config.image_size, 3])
        self.images_lateral = tf.placeholder(dtype=tf.float32, shape=[self.batch_size,
config.image_size, config.image_size, 3])
        self.sentences = tf.placeholder(dtype=tf.int32, shape=[self.batch_size,
config.max_sentence_num * config.max_sentence_length])
        self.masks = tf.placeholder(dtype=tf.float32, shape=[self.batch_size,
config.max_sentence_num * config.max_sentence_length])

        self.build_cnn()
        self.build_rnn()
        self.build_metrics()
        if is_training:
            self.build_optimizer()
            self.build_summary()

    def build_cnn(self):
        net_f, _ = inception.inception_v3(self.images_frontal, trainable=True,
is_training=True, add_summaries=False, scope='FrontalInceptionV3')
        net_l, _ = inception.inception_v3(self.images_lateral, trainable=True,
is_training=True, add_summaries=False, scope='LateralInceptionV3')

        self.visual_feats = tf.concat([net_f, net_l], axis=1) # [batch_size, print('cnn
4096]
        built.')

    def build_rnn(self):
        with tf.variable_scope("word_embedding"):
            word_embedding_matrix = tf.get_variable(
                                    name='weights',
                                    shape=[self.config.vocabulary_size,
self.config.word_embedding_size],
                                    trainable=True)

        # 1. build rnn
        WordRNN = tf.nn.rnn_cell.LSTMCell(
            name='word_rnn',
            num_units=self.config.rnn_units)
        if self.is_training:
            WordRNN = tf.nn.rnn_cell.DropoutWrapper( WordRNN,
                input_keep_prob=1.0 - self.config.rnn_dropout_rate,
                output_keep_prob=1.0 - self.config.rnn_dropout_rate,
                state_keep_prob=1.0 - self.config.rnn_dropout_rate)

        predicts = []
        cross_entropies = []
        corrects = []  global
        last_sentence
        # 2. generate first sentence
        for sent_id in range(1):
            # 2.1 init Word RNN
            with tf.variable_scope('word_rnn_initialize_0'): context
                = self.visual_feats
                init_c = tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_c')
                init_h = tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_h')

                WordRNN_last_state = init_c, init_h
                WordRNN_last_word = tf.zeros([self.batch_size], tf.int32)

            # 2.2 generate word one by one
            last_sentence = []
            for word_id in range(self.config.max_sentence_length):
                with tf.variable_scope('word_embedding'):
                    word_embedding = tf.nn.embedding_lookup(word_embedding_matrix,
WordRNN_last_word)

                with tf.variable_scope('word_rnn'):
                    WordRNN_output, WordRNN_state = WordRNN(word_embedding,
WordRNN_last_state)
                    WordRNN_last_state = WordRNN_state

                with tf.variable_scope('decode'):
                    WordRNN_output = tf.layers.dropout(WordRNN_output,
rate=self.config.dropout_rate, training=self.is_training, name='drop_d')
                    logits = tf.layers.dense(WordRNN_output,
units=self.config.vocabulary_size, activation=None, use_bias=True, name='fc_d')
                    predict = tf.argmax(logits, 1)
                    predicts.append(predict)
                    last_sentence.append(predict)

                tf.get_variable_scope().reuse_variables() if
                self.is_training:
                    WordRNN_last_word = self.sentences[:,
sent_id*self.config.max_sentence_length + word_id]
                else:
                    WordRNN_last_word = predict

                # compute cross entropy loss
                cross_entropy =
tf.nn.sparse_softmax_cross_entropy_with_logits(labels=self.sentences[:,
sent_id*self.config.max_sentence_length + word_id], logits=logits)
                masked_cross_entropy = cross_entropy * self.masks[:,
sent_id*self.config.max_sentence_length + word_id]
                cross_entropies.append(masked_cross_entropy)

                # compute acc
                ground_truth = tf.cast(self.sentences[:,
sent_id*self.config.max_sentence_length + word_id], tf.int64)
                correct = tf.where( tf.equal(predict,
                    ground_truth),
                    tf.cast(self.masks[:, sent_id*self.config.max_sentence_length
+ word_id], tf.float32),
                    tf.cast(tf.zeros_like(predict), tf.float32)
                )
                corrects.append(correct)

        # 3. generate next ot last sentence
        for sent_id in range(1, self.config.max_sentence_num): #
            3.1 get sentence feature
            with tf.variable_scope('word_embedding'): if
                self.is_training:
                    word_embeddings = tf.nn.embedding_lookup(word_embedding_matrix,
self.sentences[:, (sent_id- 1)*self.config.max_sentence_length :
sent_id*self.config.max_sentence_length])
                else:
                    batch_sentences = tf.stack(last_sentence, axis=0)         #
last_sentence shape = [max_sentence_length,batch_size]
```

```
                    batch_sentences_tran = tf.transpose(batch_sentences)
                    word_embeddings =
tf.nn.embedding_lookup(word_embedding_matrix, batch_sentences_tran)

            self.semantic_features = self.sentence_encode(word_embeddings) # 3.2

            init Word RNN
            with tf.variable_scope('word_rnn_initialize_%s' % sent_id,
reuse=tf.AUTO_REUSE):
                # vis_context = tf.layers.dense(self.visual_feats, units=1024,
activation=tf.tanh, use_bias=True, name='fc_v')
                context = tf.concat([self.visual_feats, self.semantic_features],
axis=1)
                context = tf.layers.dropout(context,
rate=self.config.dropout_rate, training=self.is_training, name='drop_s') init_c =
                tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_c')
                init_h = tf.layers.dense(context, units=self.config.rnn_units,
activation=tf.tanh, use_bias=True, name='fc_h')

                WordRNN_last_state = init_c, init_h
                WordRNN_last_word = tf.zeros([self.batch_size], tf.int32)

            # 3.3 generate word one by one
            last_sentence = []
            for word_id in range(self.config.max_sentence_length):
                with tf.variable_scope("word_embedding"):
                    word_embedding =
tf.nn.embedding_lookup(word_embedding_matrix, WordRNN_last_word)

                with tf.variable_scope('word_rnn'):
                    WordRNN_output, WordRNN_state = WordRNN(word_embedding,
WordRNN_last_state)
                    WordRNN_last_state = WordRNN_state

                with tf.variable_scope('decode'):
                    WordRNN_output = tf.layers.dropout(WordRNN_output,
rate=self.config.dropout_rate, training=self.is_training, name='drop_d')
                    logits = tf.layers.dense(WordRNN_output,
units=self.config.vocabulary_size, activation=None, use_bias=True, name='fc_d')
                    predict = tf.argmax(logits, 1)
                    predicts.append(predict)
                    last_sentence.append(predict)

                tf.get_variable_scope().reuse_variables()
                if self.is_training:
                    WordRNN_last_word = self.sentences[:, sent_id *
self.config.max_sentence_length + word_id]
                else:
                    WordRNN_last_word = predict

                # compute cross entropy loss
                cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=self.sentences[:, sent_id *
self.config.max_sentence_length + word_id],
                    logits=logits)
                masked_cross_entropy = cross_entropy * self.masks[:,
                                                       sent_id *
self.config.max_sentence_length + word_id]
                cross_entropies.append(masked_cross_entropy)

                # compute acc
                ground_truth = tf.cast(self.sentences[:, sent_id *
self.config.max_sentence_length + word_id],
                                       tf.int64)
                correct = tf.where( tf.equal(predict,
                    ground_truth),
                    tf.cast(self.masks[:, sent_id *
self.config.max_sentence_length + word_id], tf.float32),
                    tf.cast(tf.zeros_like(predict), tf.float32)
                )
                corrects.append(correct)

        self.predicts = predicts self.cross_entropies
        = cross_entropies self.corrects = corrects

        print('rnn built.')

    def build_metrics(self):
        corrects = tf.stack(self.corrects, axis=1)
        self.accuracy = tf.reduce_sum(corrects) / tf.reduce_sum(self.masks)

        cross_entropies = tf.stack(self.cross_entropies, axis=1) self.cross_entropy_loss
        = tf.reduce_sum(cross_entropies) /
tf.reduce_sum(self.masks)

        self.reg_loss = tf.losses.get_regularization_loss()
        self.loss = self.cross_entropy_loss + self.reg_loss
        print('metrics built.')

    def build_optimizer(self):
        self.global_step = tf.Variable(0, name='global_step', trainable=False)
        learning_rate = tf.constant(self.config.learning_rate)
        def _learning_rate_decay_fn(learning_rate, global_step):
            return tf.train.exponential_decay(
                learning_rate=learning_rate,
                global_step=global_step,
                decay_steps=self.config.decay_iters,
                decay_rate=self.config.decay_rate,
                staircase=True
            )

        learning_rate_decay_fn = _learning_rate_decay_fn
        with tf.variable_scope('optimizer', reuse=tf.AUTO_REUSE):
            optimizer = tf.train.AdamOptimizer(
                learning_rate=learning_rate,
                beta1=0.9,
                beta2=0.999,
                epsilon=1e-8
            )

            self.step_op = tf.contrib.layers.optimize_loss(
                loss=self.loss, global_step=self.global_step,
                learning_rate=learning_rate,
                optimizer=optimizer,
                clip_gradients=5.0,
                learning_rate_decay_fn=learning_rate_decay_fn, #
                variables=other_var_list
            )
        print('optimizer built.')

    def build_summary(self):
        with tf.name_scope("metrics"):
            tf.summary.scalar('cross entropy loss', self.cross_entropy_loss)
            tf.summary.scalar('reg loss', self.reg_loss) tf.summary.scalar('acc',
            self.accuracy)

        self.summary = tf.summary.merge_all() print('summary
        built.')

    def sentence_encode(self, word_embeddings):
        with tf.variable_scope('sentence_encode', reuse=tf.AUTO_REUSE):
            net = tf.layers.conv1d(word_embeddings, filters=1024, kernel_size=3,
strides=1)
            sent_feature1 = tf.layers.max_pooling1d(net,
pool_size=self.config.max_sentence_length - 2, strides=100)
            net  =  tf.layers.conv1d(net,  filters=1024,  kernel_size=3,  strides=1)
            sent_feature2 = tf.layers.max_pooling1d(net,
pool_size=self.config.max_sentence_length - 2 - 4, strides=100)
            net  =  tf.layers.conv1d(net,  filters=1024,  kernel_size=3,  strides=1)
            sent_feature3 = tf.layers.max_pooling1d(net,
pool_size=self.config.max_sentence_length - 2 - 6, strides=100)
        sent_feature1  =  tf.reshape(sent_feature1,  shape=[self.batch_size,  1024])
        sent_feature2  =  tf.reshape(sent_feature2,  shape=[self.batch_size,  1024])
        sent_feature3  =  tf.reshape(sent_feature3,  shape=[self.batch_size,  1024])
        semantic_features = tf.concat([sent_feature1, sent_feature2,
sent_feature3], axis=1)
        return semantic_features
```

config.py

```
class Config(object): def
    init__(self):
        self.imgs_dir_path = './data/NLMCXR_png_pairs'
        self.data_entry_path = './data/data_entry.json'
        self.train_list_path = './data/train_split.json'
        self.test_list_path = './data/test_split.json'
        self.vocabulary_path = './data/vocabulary.json'
        self.train_tfrecord_path = './data/tfrecords/train.tfrecord'
        self.test_tfrecord_path = './data/tfrecords/test.tfrecord'
        self.pretrain_cnn_model_frontal =
 './data/pretrain_model/frontal_inception_v3.ckpt'
         self.pretrain_cnn_model_lateral =
 './data/pretrain_model/lateral_inception_v3.ckpt'
        self.summary_path = './data/summary/'  self.model_path
        = './data/model/my-test' self.result_res_path =
        './data/result/res.json'  self.result_gts_path =
        './data/result/gts.json'

        self.batch_size = 26
        self.vocabulary_size = 2068
        self.rnn_units = 512
        self.word_embedding_size = 512
        self.image_size = 299
        self.max_sentence_num = 8
        self.max_sentence_length = 50
        self.epoch_num = 50
        self.train_num = 2761
        self.test_num = 350

        self.learning_rate = 1e-4
        self.dropout_rate = 0.5
        self.rnn_dropout_rate = 0.3
        self.decay_iters = 5 * self.train_num / self.batch_size self.decay_rate
        = 0.9
```

datasets.py

```
import tensorflow as tf
import json, nltk, os
import numpy as np

import config
from utils import image_utils

def get_train_batch(tfrecord_path, config, batch_size=26):
    tfrecord_path_list = [tfrecord_path]

    # 1. get filename_queue
    filename_queue = tf.train.string_input_producer(tfrecord_path_list,
shuffle=False)

    # 2. get image pixels, sentence, mask, image_id
    reader = tf.TFRecordReader()
    _, serialized_example = reader.read(filename_queue) features =
    tf.parse_single_example(
        serialized_example,
        features={
            'image_frontal_pixels': tf.FixedLenFeature([config.image_size *
config.image_size * 3], tf.float32),
            'image_lateral_pixels': tf.FixedLenFeature([config.image_size *
config.image_size * 3], tf.float32),
            'sentence':
tf.FixedLenFeature([config.max_sentence_num*config.max_sentence_length], tf.int64),
            'mask':
tf.FixedLenFeature([config.max_sentence_num*config.max_sentence_length], tf.int64),
            'image_id': tf.FixedLenFeature([1], tf.int64),
        }
    )
    image_frontal = tf.reshape(features['image_frontal_pixels'], [config.image_size,
config.image_size, 3])
    image_lateral = tf.reshape(features['image_lateral_pixels'], [config.image_size,
config.image_size, 3])
    sentence = features['sentence']
    mask = features['mask'] image_id
    = features['image_id']

    # 3. get tf.tfrecord.batch
    image_frontal_batch, image_lateral_batch, sentece_batch, mask_batch,  image_id_batch
= tf.train.shuffle_batch(
        [image_frontal, image_lateral, sentence, mask, image_id], batch_size=batch_size,
        capacity=3 * batch_size,
        min_after_dequeue=2 * batch_size
    )

    return image_frontal_batch, image_lateral_batch, sentece_batch, mask_batch,
image_id_batch

def get_train_tfrecord(imgs_path, data_entry_path, split_list_path, vocabulary_path,
tfrecord_path, config):
    with open(vocabulary_path, 'r') as f:
        vocabulary = json.load(f)
    word2id = {}
    for i in range(vocabulary._len_()): word2id[vocabulary[i]]
        = i

    filenames = os.listdir(imgs_path) with
    open(data_entry_path, 'r') as f:
        data_dict = json.load(f)
    with open(split_list_path, 'r') as f:
        split_id_list = json.load(f)

    writer = tf.python_io.TFRecordWriter(tfrecord_path)

    for id in split_id_list:
        two_name = []
        for filename in filenames:
            if id == filename.split('_')[0]:
                two_name.append(filename)

        frontal_image_name, lateral_image_name = two_name[0], two_name[1] if
        two_name[0] > two_name[1]:
            frontal_image_name, lateral_image_name = two_name[1], two_name[0]

        image_frontal = image_utils.getImages(os.path.join(imgs_path,
frontal_image_name), config.image_size)
        image_frontal = image_frontal.reshape([config.image_size*config.image_size*3])
        image_lateral = image_utils.getImages(os.path.join(imgs_path,
lateral_image_name), config.image_size)
        image_lateral = image_lateral.reshape([config.image_size*config.image_size*3])

        sent_list = data_dict[id]
        if sent_list.__len__() > config.max_sentence_num:
            sent_list = sent_list[:config.max_sentence_num]

        word_list = []
        for sent in sent_list:
            words = nltk.word_tokenize(sent)
            if words.__len__() >= config.max_sentence_length: fori
                in range(config.max_sentence_length - 1):
                    word_list.append(words[i])
                word_list.append('</S>')
            else:
                for i  in range(words._len_()):
                    word_list.append(words[i])
                word_list.append('</S>')
                for _ in range(config.max_sentence_length - words.__len__() - 1):
                    word_list.append('<EOS>')
        for _ in range(config.max_sentence_num - sent_list.__len__()):
            word_list.append('</S>')
            for _ in range(config.max_sentence_length-1):
                word_list.append('<EOS>')
        # print(word_list.__len__())

        sentence = np.zeros(shape=[config.max_sentence_num * config.max_sentence_length],
dtype=np.int64)
        mask = np.ones(shape=[config.max_sentence_num * config.max_sentence_length],
dtype=np.int64)
        for i in range(config.max_sentence_num*config.max_sentence_length): sentence[i]
            = word2id[word_list[i]]
            if word_list[i] == '<EOS>':
                mask[i] = 0

        image_id = int(id[3:]) example
        = tf.train.Example(
            features=tf.train.Features(
                feature={
                    'image_frontal_pixels':
tf.train.Feature(float_list=tf.train.FloatList(value=image_frontal)),
                    'image_lateral_pixels':
tf.train.Feature(float_list=tf.train.FloatList(value=image_lateral)),
                    'sentence':
tf.train.Feature(int64_list=tf.train.Int64List(value=sentence)),
                    'mask':
tf.train.Feature(int64_list=tf.train.Int64List(value=mask)),
                    'image_id':
tf.train.Feature(int64_list=tf.train.Int64List(value=[image_id]))
                }
            )
        )
        serialized = example.SerializeToString()
        writer.write(serialized)

    print('%s write to tfrecord success!' % tfrecord_path) #

config = config.Config()
# #1. get train.tfrecord
# get_train_tfrecord(config.imgs_dir_path, config.data_entry_path,
config.train_list_path, config.vocabulary_path, config.train_tfrecord_path, config)
#
# #2. get test.tfrecord
# get_train_tfrecord(config.imgs_dir_path, config.data_entry_path,
config.test_list_path, config.vocabulary_path, config.test_tfrecord_path,  config)
```

demo.py

```
import tensorflow as tf
import numpy as np
import json

from config import Config from
utils import image_utils
from cnn_hier_rnn_model import Model  #
from cnn_sem_rnn_model import Model

def get_test_data(img_frontal_path, img_lateral_path, config): image_frontal
    = np.zeros([1, config.image_size, config.image_size, 3])
    image_frontal[0] = image_utils.getImages(img_frontal_path, config.image_size)
    image_lateral = np.zeros([1, config.image_size, config.image_size, 3])
    image_lateral[0] = image_utils.getImages(img_lateral_path, config.image_size)

    sentence = np.zeros([1, config.max_sentence_num *
config.max_sentence_length])
    mask = np.zeros([1, config.max_sentence_num * config.max_sentence_length]) return

    image_frontal, image_lateral, sentence, mask

def get_sentences(predicts_list, config):
    with open(config.vocabulary_path, 'r') as f:
        vocabulary_list = json.load(f)
    word2id = {}
    for i in range(vocabulary_list.__len__()):
        word2id[vocabulary_list[i]] = i
    id2word = {v: k for k, v in word2id.items()}

    sentence_list = []
    for i in range(config.max_sentence_num): sentence
        = []
        for j in range(config.max_sentence_length):
            id = int(predicts_list[0][i*config.max_sentence_length + j][0]) if
            id2word[id] == '</S>':
                break
            else:
                sentence.append(id2word[id])
        sentence = ' '.join(sentence)
        sentence_list.append(sentence)
    return sentence_list

FLAGS = tf.app.flags.FLAGS
tf.flags.DEFINE_string('img_frontal_path', './data/experiments/CXR1900_IM-0584-
1001.png', 'The frontal image path')
tf.flags.DEFINE_string('img_lateral_path', './data/experiments/CXR1900_IM-0584- 2001.png',
'The lateral image path')
tf.flags.DEFINE_string('model_path', './data/model/my-test-1000', 'The test model
path')


img_frontal_path    =    FLAGS.img_frontal_path
img_lateral_path    =    FLAGS.img_lateral_path
model_path = FLAGS.model_path

config = Config()
mt = Model(is_training=False, batch_size=1)
```

```
img_frontal, img_lateral, sentence, mask = get_test_data(img_frontal_path,
img_lateral_path, config)

saver = tf.Saver()
with tf.Session() as sess: sess.run(tf.global_variables_initializer())
    saver.restore(sess, model_path)
    feed_dict = {
        mt.images_frontal: img_frontal,
        mt.images_lateral: img_lateral,
        mt.sentences: sentence, mt.masks:
        mask
    }
    predicts_list = sess.run([mt.predicts], feed_dict=feed_dict) sentence_list =

get_sentences(predicts_list, config)

print('The generate report:')
for sentence in sentence_list:
    print('\t %s' % sentence)
```

metrics.py

```
from pycocoevalcap.bleu.bleu import Bleu from
pycocoevalcap.cider.cider import Cider
from pycocoevalcap.meteor.meteor import Meteor from
pycocoevalcap.rouge.rouge import Rouge import json

def coco_caption_metrics_hier(predicts_list, sentences_list, image_id_list, config,
batch_size=26, is_training=True):
    with open(config.vocabulary_path, 'r') as file:
        vocabulary_list = json.load(file)
    word2id = {}
    for i in range(vocabulary_list.__len__()):
        word2id[vocabulary_list[i]] = i
    id2word = {v: k for k, v in word2id.items()}

    gts = {}
    res = {}
    for i in range(0, predicts_list.__len__()): for
        j in range(0, batch_size):
            sent_pre, sent_gt = [], []
            for k in range(config.max_sentence_num * config.max_sentence_length):
                id_input = int(predicts_list[i][k][j])
                sent_pre.append(id2word[id_input])

                id_gt = sentences_list[i][j][k]
                if (not id2word[id_gt].__eq__('</S>')) and (not
id2word[id_gt].__eq__('<EOS>')):
                    sent_gt.append(id2word[id_gt])

            # sent_pre2 = sent_pre
            sent_pre2 = []
            for n in range(config.max_sentence_num):
                for m in range(config.max_sentence_length):
                    word = sent_pre[n*config.max_sentence_length + m] if
                    word != '</S>':
                        sent_pre2.append(word)
                    else:
                        break

            str_pre, str_gt = ' '.join(sent_pre2), ' '.join(sent_gt) image_id
            = image_id_list[i][j][0]
            gts[image_id] = [str_gt]
            res[image_id] = [str_pre]

    if not is_training:
        with open(config.result_gts_path, 'w') as file:
            json.dump(gts, file)
        with open(config.result_res_path, 'w') as file:
            json.dump(res, file)

    bleu_scorer = Bleu(n=4)
    bleu, _ = bleu_scorer.compute_score(gts=gts, res=res)

    rouge_scorer = Rouge()
    rouge, _ = rouge_scorer.compute_score(gts=gts, res=res)

    cider_scorer = Cider()
    cider, _ = cider_scorer.compute_score(gts=gts, res=res)

    meteor_scorer = Meteor()
    meteor, _ = meteor_scorer.compute_score(gts=gts, res=res)

    for i in range(4):
        bleu[i] = round(bleu[i], 4)

    return bleu, round(meteor, 4), round(rouge, 4), round(cider, 4)
```

train.py

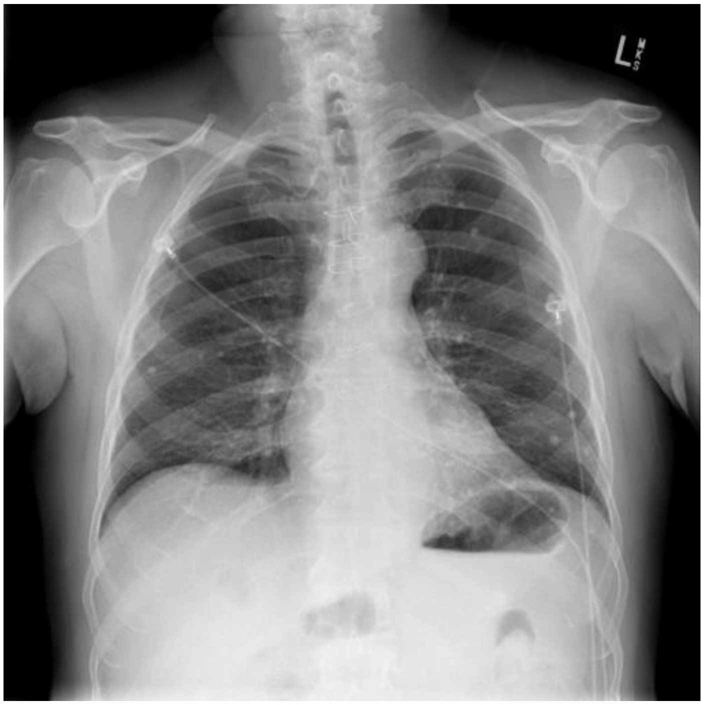
```
import tensorflow as tf
from tensorflow.contrib import slim
import numpy as np

import datasets
import metrics
from config import Config
from cnn_hier_rnn_model import Model
# from cnn_vis_sem_rnn_model import Model

def train():
    c = Config()
    md = Model(is_training=True, config=c, batch_size=c.batch_size) mt =
    Model(is_training=False, config=c, batch_size=3)

    print('Read Data...')
    image_frontal_batch, image_lateral_batch, sentence_batch, mask_batch, image_id_batch
= datasets.get_train_batch(c.train_tfrecord_path, c, md.batch_size)
    image_frontal_batch2, image_lateral_batch2, sentence_batch2, mask_batch2,
image_id_batch2 = datasets.get_train_batch(c.test_tfrecord_path, c, mt.batch_size)

    init_fn_frontal =
slim.assign_from_checkpoint_fn(c.pretrain_cnn_model_frontal,
slim.get_model_variables('FrontalInceptionV3'))
    init_fn_lateral = slim.assign_from_checkpoint_fn(c.pretrain_cnn_model_lateral,
slim.get_model_variables('LateralInceptionV3'))

    saver = tf.train.Saver(max_to_keep=100)
    print('Train Model...')
    with tf.Session() as sess:
        train_writer = tf.summary.FileWriter(c.summary_path, sess.graph)
        sess.run(tf.global_variables_initializer())
        sess.run(tf.local_variables_initializer())  init_fn_frontal(sess)
        init_fn_lateral(sess)

        coord = tf.train.Coordinator()     # queue manage
        threads = tf.train.start_queue_runners(coord=coord)

        iter = 0
        # loss_list, acc_list, predicts_list, sentences_list, image_id_list = [], [],
[], [], []
        for epoch in range(c.epoch_num):
            for _ in range(c.train_num / md.batch_size):
                images_frontal, images_lateral, sentences, masks, image_ids =
sess.run([image_frontal_batch, image_lateral_batch, sentence_batch, mask_batch,
image_id_batch])
                feed_dict = {
                    md.images_frontal: images_frontal,
                    md.images_lateral: images_lateral,
                    md.sentences: sentences,  md.masks:
                    masks
                }
                _, _summary, _global_step, _loss, _acc, _predicts, = sess.run(
                    [md.step_op, md.summary, md.global_step, md.loss,
md.accuracy, md.predicts], feed_dict=feed_dict)
                train_writer.add_summary(_summary, _global_step)

                # loss_list.append(_loss) #
                acc_list.append(_acc)
                # predicts_list.append(_predicts) #
                sentences_list.append(sentences) #
                image_id_list.append(image_ids)

                iter += 1
                if iter % 100 == 0: #
                    train test
                    # bleu, meteor, rouge, cider =
metrics.coco_caption_metrics_hier(predicts_list,
                    #
   sentences_list,
                    #
   image_id_list,
                    #
   config=c,
                    #
   batch_size=md.batch_size,
                    #
   is_training=md.is_training)
                    # print('iter = %s, loss = %.4f, acc = %.4f, bleu = %s, meteor =
%s, rouge = %s, cider = %s' %
                    #       (iter, np.mean(loss_list), np.mean(acc_list), bleu,
meteor, rouge, cider))

                    # test test
                    loss_list, acc_list, predicts_list, sentences_list,
image_id_list = [], [], [], [], []
                    for _ in range(c.test_num / mt.batch_size):
                        images_frontal, images_lateral, sentences, masks,
image_ids = sess.run([image_frontal_batch2, image_lateral_batch2,
sentence_batch2, mask_batch2, image_id_batch2])
                        feed_dict = {
                            mt.images_frontal: images_frontal,
                            mt.images_lateral: images_lateral,
                            mt.sentences: sentences,
                            mt.masks: masks
                        }
                        _loss, _acc, _predicts = sess.run([mt.loss, mt.accuracy,
mt.predicts], feed_dict=feed_dict)
                        loss_list.append(_loss)
                        acc_list.append(_acc)
                        predicts_list.append(_predicts)
                        sentences_list.append(sentences)
                        image_id_list.append(image_ids)

                    bleu, meteor, rouge, cider =
metrics.coco_caption_metrics_hier(predicts_list,

 sentences_list,

 image_id_list,

 config=c,

 batch_size=mt.batch_size,

 is_training=mt.is_training)
                    print('---------iter = %s, loss = %.4f, acc = %.4f, bleu =
%s, meteor = %s, rouge = %s, cider = %s' %
                          (iter, np.mean(loss_list), np.mean(acc_list), bleu,
meteor, rouge, cider))
                    # loss_list, acc_list, predicts_list, sentences_list,
image_id_list = [], [], [], [], []

                    saver.save(sess, c.model_path, global_step=iter)

        coord.request_stop()
        coord.join(threads)
```

```
train()
```

Demo

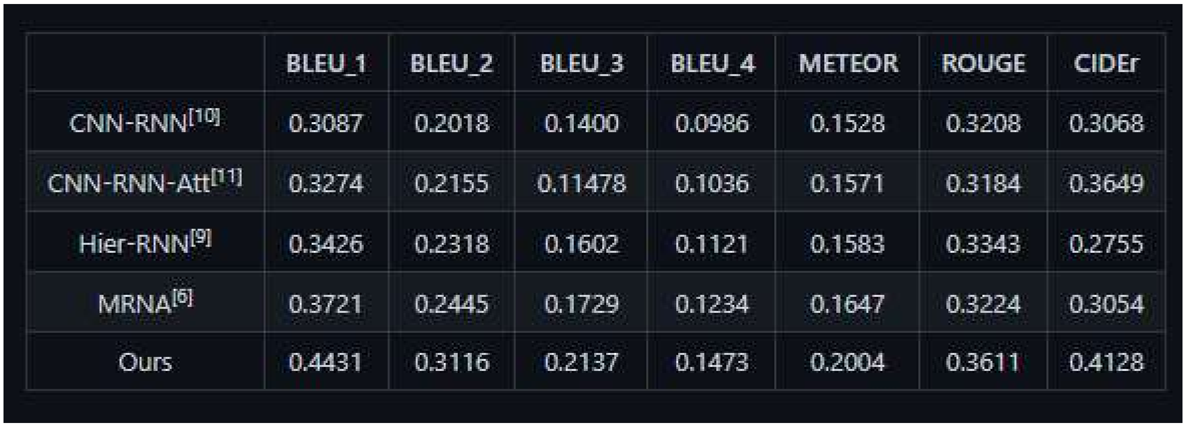
You could use two chest x-ray images to test

```
$ python demo.py --img_frontal_path='./data/experiments/CXR1900_IM-0584-1001.png'
-- img_lateral_path='./data/experiments/CXR1900_IM-0584-2001.png' --
model_path='./data/model/my-test-2500'
```

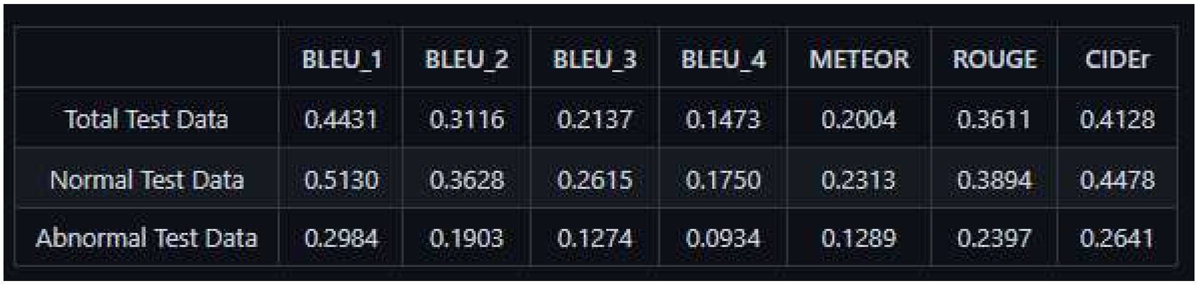


```
$ The generate report:
    no acute cardiopulmonary
    abnormality the lungs are clear
    there is no focal
    consolidation there is no
    focal consolidation
    there is no pneumothorax or pneumothorax
```

Metrics Result :



	BLEU_1	BLEU_2	BLEU_3	BLEU_4	METEOR	ROUGE	CIDEr
CNN-RNN[10]	0.3087	0.2018	0.1400	0.0986	0.1528	0.3208	0.3068
CNN-RNN-Att[11]	0.3274	0.2155	0.11478	0.1036	0.1571	0.3184	0.3649
Hier-RNN[9]	0.3426	0.2318	0.1602	0.1121	0.1583	0.3343	0.2755
MRNA[6]	0.3721	0.2445	0.1729	0.1234	0.1647	0.3224	0.3054
Ours	0.4431	0.3116	0.2137	0.1473	0.2004	0.3611	0.4128



	BLEU_1	BLEU_2	BLEU_3	BLEU_4	METEOR	ROUGE	CIDEr
Total Test Data	0.4431	0.3116	0.2137	0.1473	0.2004	0.3611	0.4128
Normal Test Data	0.5130	0.3628	0.2615	0.1750	0.2313	0.3894	0.4478
Abnormal Test Data	0.2984	0.1903	0.1274	0.0934	0.1289	0.2397	0.2641

Code Frame Work :

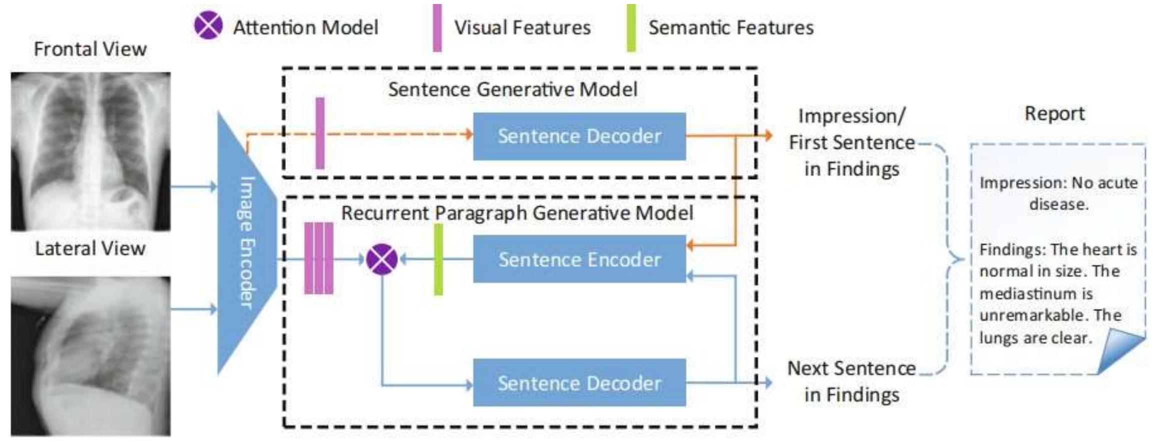


Fig. 2. The architecture of the proposed multimodal recurrent generation model with attention for radiology reports. Best viewed in color.

Summary of Process

- To start, since it is simple to connect this task with the Image2Text Task, I use the Image Captions techniques to address the issues with this task, similar to CNN+RNN techniques.

- Second, while this work involves numerous sentences, I discovered that the Image Captions approach can only handle one sentence. I therefore employ techniques for creating image paragraph descriptions, such as CNN+Hierarchical RNN.

- After that, I discovered that the reports for this task had descriptions of the Impression and Findings, so I used the QA + Hierarchical RNN approach to address the issues with this assignment.

- Finally, I discovered that linguistic information is more significant than image information due to the small scale dataset.

References :

[1] TieNet Text-Image Embedding Network for Common Thorax Disease Classification and Reporting in Chest X-rays, Xiaosong Wang et at, CVPR 2018, NIH

[2] On the Automatic Generation of Medical Imaging Reports, Baoyu Jing et al, ACL 2018, CMU

[3] Multimodal Recurrent Model with Attention for Automated Radiology Report Generation, Yuan Xue, MICCAI 2018, PSU

[4] Hybrid Retrieval-Generation Reinforced Agent for Medical Image Report Generation, Christy Y. Li et al, NIPS 2018, CMU

[5] Knowledge-Driven Encode, Retrieve, Paraphrase for Medical Image Report Generation, Christy Y. Li et al, AAAI 2019, DU

[6]A Hierarchical Approach for Generating Descriptive Image Paragraphs, Jonathan Krause et al, CVPR 2017, Stanford

[7] Show and Tell: A Neural Image Caption Generator, Oriol Vinyals et al, CVPR 2015, Google

[8] Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, Kelvin Xu et at, ICML 2015