./folder/config.py

```
# -*- coding:utf-8 -*-
# @Author: wangli
# @Date: 2019-03-26

import os

class config():
    def __init__(self):
        # data
        self.max_len = 15
        self.train_path = './train_part.txt'
        self.eval_path = None
        self.test_path = None

        # save dict path
        self.save_dict_path = './source/saved_dict'
        self.counter_path = os.path.join(self.save_dict_path,
'couter_dict.pkl')
        self.vocab_path = os.path.join(self.save_dict_path,
'vocab_dict.pkl')

        # net
        self.embedding_size = 512
        self.gru_hidden_size = 128
        self.gru_layers = 1
        self.bidirection = True
```

./folder/main.py

```
# -*- coding:utf-8 -*-
# @Author: wangli
# @Date: 2019-03-26

import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import os
from config import config
from preprocess import Vocab, Preprocess, dataset
from model import Encoder, Attention, Decoder
from torch.utils.data import DataLoader, Dataset

if __name__ == "__main__":
    print('==> Loading config......')
    cfg = config()
    print('==> Preprocessing data......')
    voc = Vocab(cfg)
    voc.gen_counter_dict()
    voc.gen_vocab()
    cfg.vocab_len = voc.vocab_len
    print('The length of vocab is: {}'.format(cfg.vocab_len))

    prep = Preprocess(cfg, voc.vocab)
    pairs = prep.gen_pair_sen()
    print('pairs sentences generated.')
    pairs = prep.tokenize(pairs)
    print('sentences tokenized.')

    traindataset = dataset(pairs, voc.vocab)
    traindataloader = DataLoader(traindataset, batch_size=5,
shuffle=False)
    one_iter = iter(traindataloader).next()

    encoder = Encoder(cfg)
    encoder_outputs, _ = encoder(one_iter['inputs'], one_iter['length'])
```

./folder/model.py

```
# -*- coding:utf-8 -*-
# @Author: wangli
# @Date: 2019-03-26

import torch
import torch.nn as nn
import torch.nn.functional as F

from config import config
# cfg = config()

class Encoder(nn.Module):
    def __init__(self, cfg):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding(cfg.vocab_len, cfg.embedding_size)
        self.gru = nn.GRU(input_size=cfg.embedding_size,
hidden_size=cfg.gru_hidden_size,
                          num_layers=cfg.gru_layers,
bidirectional=cfg.bidirection)

    def forward(self, inputs, in_lengths):
        inputs = self.embedding(inputs)
        outs, hidden = self.gru(inputs.transpose(0,1))
#         packed = torch.nn.utils.rnn.pack_padded_sequence(input=inputs,
lengths=in_lengths, batch_first=True)
#         outs, hidden = self.gru(packed)
#         unpack = torch.nn.utils.rnn.pad_packed_sequence(outs)
#         return unpack, hidden
```

```
        return outs, hidden

class Attention(nn.Module):
    def __init__(self, method):
        self.method = method
        super(Attention, self).__init__()
        if self.method not in ['dot']:
            raise ValueError(self.method, 'is not a appropriate method. ')

    def dot(self, rnn_out, encoder_outputs):
        return torch.sum(rnn_out * encoder_outputs, dim=2)

    def forward(self, rnn_out, encoder_outputs):
        # encoder_outputs shape: [seq_len, batch_size,
num_directions*hidden_size]
        # rnn_out shape: [1, batch_size, num_directions*hidden_size]
        if self.method == 'dot':
            # attention shape: [seq_len, batch_size]
            attention = self.dot(rnn_out, encoder_outputs)
            attention = attention.t()
            attention = F.softmax(attention, dim=1)

        return attention

class Decoder(nn.Module):
    def __init__(self, cfg, dropout):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(cfg.vocab_len, cfg.embedding_size)
        self.gru = nn.GRU(inputs=cfg.embedding_size,
hidden_size=cfg.gru_hidden_size,
                          num_layers=cfg.gru_layers,
bidirectional=cfg.bidirection)
        self.embedding_dropout = nn.Dropout(dropout)
        self.attn = Attention('dot')

    def forward(self, decoder_inputs, last_hidden, encoder_outputs):
        # decoder_inputs shape: [batch_size, 1]
        # encoder_outputs shape: [seq_len, batch_size,
num_directions*hidden_size]
        # embedded shape: [batch_size, 1, embedding_size]
        embedded = self.embedding(decoder_inputs)
        embedded = self.embedding_dropout(embedded)
        # rnn_inputs shape: [1, batch_size, embedding_size]
        # rnn_outs shape: [1, batch_size, num_directions*hidden_size]
        rnn_outs, hidden = self.gru(embedded.transpose(0,1), last_hidden)
        attention = self.attn(rnn_outs, encoder_outputs)
```

./folder/preprocess.py

```
# -*- coding:utf-8 -*-
# @Author: wangli
# @Date: 2019-03-26
```

```python
import json
import numpy as np
import pickle
import os
import torch
from tqdm import tqdm
from collections import Counter
from torch.utils.data import Dataset, DataLoader
from config import config


class Vocab():
    """
    词表类，包括了生成统计词数量，生成词索引表方法
    """

    def __init__(self, cfg):
        self.counter = Counter()
        self.vocab = dict()
        self.vocab_len = 0

    def gen_counter_dict(self):
        """
        统计词频，并保存为pickle文件
        """

        if os.path.exists(cfg.counter_path):
            self.counter = pickle.load(open(cfg.counter_path, 'rb'))
            print('counter dict loaded.')
        else:
            # mkdir -p 如果上级目录不存在，会首先建立上级目录
            if not os.path.exists(cfg.save_dict_path):
                os.system('mkdir -p ' +  cfg.save_dict_path)
            f = open(cfg.train_path, 'r', encoding='utf-8')
            for line in tqdm(f):
                conversation = ' ' .join(json.loads(line)['conversation'])
                self.counter.update(conversation.split())
            f.close()

            pickle.dump(self.counter, open(cfg.counter_path, 'wb'))
            print('counter dict saved.')

    def gen_vocab(self):
        """
        生成词索引表，并保存为pickle文件
        """

        if os.path.exists(cfg.vocab_path):
            self.vocab = pickle.load(open(cfg.vocab_path, 'rb'))
            self.vocab_len = len(self.vocab)
            print('vocab dict loaded.')
        else:
            # mkdir -p 如果上级目录不存在，会首先建立上级目录
            if not os.path.exists(cfg.save_dict_path):
                os.system('mkdir -p ' +  cfg.save_dict_path)
            PAD = 0 # padding token
            SOS = 1 # start of sentence
            EOS = 2 # end of sentence
            UNK = 3 # out of vocabulary
            # 按照词频排序
            sort_tmp = sorted(self.counter.items(), key=lambda x: x[1],
reverse=True)
            # 词表
            vocab_list = list(map(lambda x: x[0], sort_tmp))
            self.vocab_len = len(vocab_list) + 4
            self.vocab.update({'PAD':0, 'SOS': 1, 'EOS': 2, 'UNK': 3})
            self.vocab.update(dict(zip(vocab_list,
range(4,self.vocab_len))))

            # 存成二进制的pickle文件
            pickle.dump(self.vocab, open(cfg.vocab_path, 'wb'))
            print('vocab dict saved.')

class Preprocess():
    def __init__(self, cfg, vocab_dict):
        self.vocab_dict = vocab_dict

    def gen_pair_sen(self):
        """从一轮对话中生成对话长度减1的pair对。即上一句为输入，下一句为目标输出"""

        pairs = []
        f = open(cfg.train_path, 'r', encoding='utf-8')
        for line in tqdm(f):
            conversation = json.loads(line)['conversation']
            for i in range(len(conversation)-1):
                pairs.append([conversation[i].split(),
conversation[i+1].split()])
        return pairs

    def cut_pad(self, line):
        """句子长度大于max_len的截掉，小于max_len的补'PAD'"""

        line[0] = line[0] + (cfg.max_len - len(line[0]))*['PAD'] if
len(line[0]) < cfg.max_len \
                                                                else
line[0][:cfg.max_len]
        line[1] = line[1] + (cfg.max_len - len(line[1]))*['PAD'] if
len(line[1]) < cfg.max_len \
                                                                else
line[1][:cfg.max_len]
        return line

    def tokenize(self, pairs):
        return list(map(self.cut_pad, pairs))


class dataset(Dataset):
    def __init__(self, pairs, vocab_dict):
        self.pairs = pairs
        self.vocab_dict = vocab_dict
        # super.__init__()

    def __getitem__(self, index):
        inputs, targets = pairs[index]
        inputs = [self.vocab_dict[char] if self.vocab_dict.get(char) or
self.vocab_dict.get(char)==0
                                else self.vocab_dict['UNK'] for char in inputs]
        targets = [self.vocab_dict[char] if self.vocab_dict.get(char) or
self.vocab_dict.get(char)==0
                                else self.vocab_dict['UNK'] for char in targets]
        i_length = len(inputs) - inputs.count(0)
        t_length = len(targets) - targets.count(0)
        mask = [1]*t_length + [0]*targets.count(0)
        outs = {
            'inputs': inputs,
            'length': i_length,
            'targets': targets,
            'mask': mask
        }
        return {key: torch.tensor(value) for key, value in outs.items()}

    def __len__(self):
        return len(self.pairs)

```

./Read.py

```
#!/usr/bin/env python

import os

pre = './'

lists = []

def gci(filepath):
#遍历filepath下所有文件，包括子目录
  files = os.listdir(filepath)
  for fi in files:
    fi_d = os.path.join(filepath,fi)
    if os.path.isdir(fi_d):
      gci(fi_d)
    elif fi_d.endswith('.py') and fi != '__init__.py':
      lists.append(fi_d)
      print(fi_d)

gci(pre)

with open('code.md', 'w') as file:
```

````
    for location in lists:
      content = location + '\n```python\n' +\
         open(location).read() + '\n```\n'
      file.write(content)
````