50.040 Natural Language Processing, Summer 2022

Homework 3

Due 29 July 2022, 5pm

Write your student ID and name

ID: 1004365

Name: Lee Jet Xuen

Students whom you have discussed with (if any):
Jerome Heng 1004115
Brandon Chong 1004104
Tay Sze Chang 1004301

Requirements:

- **Use Python to complete this homework.**
- **Follow the honor code strictly.**
- ***Use torch >= 1.5.1***
- ***Use torchtext >= 0.6.0***

In [1]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

from torch.utils.data import Dataset, DataLoader
from torchtext import data
from collections import namedtuple
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

from nltk.translate.bleu_score import corpus_bleu
```

In [2]:

```
from google.colab import drive
drive.mount('/content/drive')
# directory = r"drive/MyDrive/50.040 NLP/homework3/"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount
("/content/drive", force_remount=True).
```

In [3]:

```
%cd drive/MyDrive/50.040 NLP/homework3/
%pwd
```

```
/content/drive/MyDrive/50.040 NLP/homework3
```

Out[3]:

```
'/content/drive/MyDrive/50.040 NLP/homework3'
```

Part 2: Neural Machine Translation [25 points]

Dataset

In [4]:

```
STOP_TOKEN = '</s>'
START_TOKEN = '<s>'
UNK_TOKEN = '<unk>'
PAD_TOKEN = '<pad>'

class TranslationDataset(Dataset):
    def __init__(self, sent_pairs, src_word2idx, tgt_word2idx, tokenizer, max_len):
        self.pairs = sent_pairs
        self.src_w2i = src_word2idx
        self.tgt_w2i = tgt_word2idx
        self.tokenizer = tokenizer
        self.max_len = max_len
    def __len__(self):
        return len(self.pairs)

    def __getitem__(self, idx):
        src_ids = []
        tgt_ids = []
        src = self.pairs[idx].src
        tgt = self.pairs[idx].tgt

        src_words = self.tokenizer(src)
        tgt_words = self.tokenizer(tgt)
        for i in src_words:
            try:
                idx = self.src_w2i[i]
            except KeyError:
                idx = self.src_w2i[UNK_TOKEN]
            src_ids.append(idx)
        for j in tgt_words:
            try:
                idx = self.tgt_w2i[j]
            except KeyError:
                idx = self.tgt_w2i[UNK_TOKEN]
            tgt_ids.append(idx)

        src_length = len(src_ids)
        tgt_length = len(tgt_ids)
        if src_length < self.max_len:
            src_ids = src_ids + [self.src_w2i[STOP_TOKEN]] + [self.src_w2i[PAD_TOKEN]] *
(self.max_len - src_length - 1)
            assert len(src_ids) == self.max_len
            src_length += 1
        else:
            src_ids = src_ids[:self.max_len-1] + [self.src_w2i[STOP_TOKEN]]
            src_length = self.max_len

        if tgt_length < self.max_len-1:
            tgt_ids = [self.tgt_w2i[START_TOKEN]] + tgt_ids + [self.tgt_w2i[STOP_TOKEN]]
+\
            [self.tgt_w2i[PAD_TOKEN]] * (self.max_len - tgt_length - 2)
            assert len(tgt_ids) == self.max_len
            tgt_length += 2
        else:
            tgt_ids = [self.tgt_w2i[START_TOKEN]] + tgt_ids[:self.max_len-2] + [self.tgt
_w2i[STOP_TOKEN]]
            tgt_length = self.max_len

        src_mask = np.zeros(self.max_len)
        tgt_mask = np.zeros(self.max_len)
        src_mask[:src_length] = 1
        tgt_mask[:tgt_length] = 1

        return torch.LongTensor(src_ids), torch.LongTensor(tgt_ids), torch.LongTensor([s
rc_length]), \
        torch.LongTensor([tgt_length]),  torch.BoolTensor(src_mask), torch.BoolTensor(tg
t_mask), tgt
```

utils

Question 6

Before we build our model, we need to preprocess our data. Implement `read_corpus` function.

In [5]:

```
Pair = namedtuple('Pair', ['src','tgt'])


def read_corpus(data_path):
    '''
    param:
    data_path: str --- path to the data file

    return:
    src: list[str] --- contains the source language sentences; each sentence is a string;
    tgt: list[str] --- contains the target language sentences; each sentence is a string;
    src_vocab: set(str) --- contains all the source language words appearing in the data
file; each word is a string;
    src_tgt: set(str) --- --- contains all the target language words appearing in the dat
a file; each word is a string;

    '''
    with open(data_path, 'r', encoding='utf-8') as d:
        data = d.readlines()
        src, tgt = [], []
        src_vocab, tgt_vocab = set(), set()

        # 'data' is a list of strings; each element of this list represents a sentence wh
ich ended with "\n" .
        # Source language sentence (French) and target language sentence (English) are sp
lit by "\t"
        # Don't forget to remove the special "\n" symbol of each sentence string

        ### YOUR CODE HERE

        for i in data:
          s, t = i.split("\t")
          src.append(s.strip())
          src_vocab = src_vocab.union(s.strip().split())

          tgt.append(t.strip())
          tgt_vocab = tgt_vocab.union(t.strip().split())

        ### END OF YOUR CODE
        assert len(src) == len(tgt)
        return src, tgt, src_vocab, tgt_vocab

def lang_pairs(src, tgt):
    pairs = []
    for s,t in zip(src, tgt):
        pairs.append(Pair(src=s, tgt=t))
    return pairs

def build_w2i(vocab):

    w2i = {}
    for i, w in enumerate(vocab):
        w2i[w] = i
    w2i[START_TOKEN] = len(w2i)
    w2i[STOP_TOKEN] = len(w2i)
    w2i[UNK_TOKEN] = len(w2i)
    w2i[PAD_TOKEN] = len(w2i)

    return w2i
```

```
def build_i2w(w2i):
    i2w = {}
    for k,v in w2i.items():
        i2w[v] = k
    return i2w
```

In [6]:

```
train_src, train_tgt, src_vocab, tgt_vocab = read_corpus(r'data/part2/train')
dev_src, dev_tgt, _, _ = read_corpus(r'data/part2/dev')
test_src, test_tgt, _, _ = read_corpus(r'data/part2/test')

train_sent_pairs = lang_pairs(train_src,train_tgt)
dev_sent_pairs = lang_pairs(dev_src,dev_tgt)
test_sent_pairs = lang_pairs(test_src,test_tgt)

fr_w2i = build_w2i(src_vocab)
en_w2i = build_w2i(tgt_vocab)
en_i2w = build_i2w(en_w2i)
```

In [7]:

```
print(len(src_vocab), len(tgt_vocab), len(train_src), len(train_tgt))
print(len(fr_w2i), len(en_w2i), len(en_i2w))
```

```
40992 25370 140000 140000
40996 25374 25374
```

In [8]:

```
train_src[0], train_tgt[0]
```

Out[8]:

```
("Elle ne voulait pas qu'il joue au poker.",
 "She didn't want him to play poker.")
```

Model

Question 7

Implement part of the `__init__` function in `Encoder` class and `Decoder` class.

Question 8

Implement the `__init__` and `forward` function `Attention` class. This function will generate attention distribution α_t.

Question 9

Implement the `forward` function in `Encoder` class . This function converts source sentences into word embedding tensors X, generates $h_1^{enc}, h_2^{enc}, \ldots, h_m^{enc}$ and computes initial hidden state h_0^{dec}, and initial cell state c_0^{dec}.

Question 10

Implement the `forward` function in `Decoder` class. This function constructs $\bar{y}_t$ and runs the `decode_one_step` function over every time step of the input sentence.

Question 11

Implement `decode_one_step` function in `Decoder` class. This function applies the decoder's LSTM Cell for a

single time step, computing the encoding of the target word h_t^{dec}, the attention distribution α_t, attention output a_t and the combined output o_t. Hint: You should be using the implemented "Attention" class for the computation of the attention distribution

In [9]:

```
BeamNode = namedtuple('BeamNode',['prev_node', 'prev_hidden','prev_o_t', 'wordID', 'score', 'length'])
Translation = namedtuple('Translation',['sent', 'score'])

device = 'cuda:0' if torch.cuda.is_available else 'cpu'

class Encoder(nn.Module):
    def __init__(self, encoder_config):
        super(Encoder, self).__init__()
        self.hidden_size = encoder_config['hidden_size']
        self.num_layers = encoder_config['num_layers']
        self.bidir = encoder_config['bidirectional']
        self.vocab_size = encoder_config['vocab_size']
        self.emb_size = encoder_config['emb_size']
        self.src_emb_matrix = encoder_config['src_embedding']

        self.scr_embedding = None
        self.W_h = None
        self.W_c = None

        ### TODO Initialize variables:
        #               self.scr_embedding: Embedding layer for source language
        #               self.W_h: Linear layer without bias (W_h describled in the PDF)
        #               self.W_c: Linear layer without bias (W_c described in the PDF)
        #
        #   You need to use nn.Embedding function and two variables we have initialized for you.
        #   You need to use nn.Linear function and one variable we have initialized for you.
        #   For the use of nn.Embedding function, please refer to https://pytorch.org/docs/stable/nn.html#torch.nn.embedding
        #   For the use of nn.Linear function, please refer to https://pytorch.org/docs/stable/nn.html#torch.nn.Linear
        #   In nn.Linear function, the matrix multiplication is a transposed version of the Eq.(1) in description PDF.

        ### YOUR CODE HERE (3 lines)

        self.src_embedding = nn.Embedding(self.vocab_size, self.emb_size)
        self.W_h = nn.Linear(self.hidden_size * 2, self.hidden_size, bias=False)
        self.W_c = nn.Linear(self.hidden_size * 2, self.hidden_size, bias=False)

        ### END OF YOUR CODE

        if self.src_emb_matrix is not None:
            self.src_embedding.weight.data.copy_(torch.FloatTensor(self.src_emb_matrix))
            self.src_embedding.weight.requires_grad = True

        self.rnn = nn.LSTM(input_size = self.emb_size,
                           hidden_size = self.hidden_size,
                           num_layers = self.num_layers,
                           bidirectional = self.bidir,
                           batch_first  = True)

    def forward(self, src_ids, src_length):
        '''
        params:
            src_ids: torch.LongTensor of shape (batch_size, max_len)
            src_length: torch.LongTensor of shape (batch_size,) contains the actual length of each sentence in the batch
        return:
            encoder_outputs: torch.FloatTensor of shape(batch_size, max_len_in_batch, 2*hidden_size); the hidden states produced by Bi-LSTM
            decoder_init: tuple(last_hidden, last_cell); last_hidden: torch.FloatTensorof shape (batch_size, 2*hidden_size);
```

```
                                                            last_cell: torch.FloatTensor of shape(batch_size, 2*hidden_size);
                                                            they are h_0^{dec},c_0^{dec} in our description PDF
        '''

        encoder_outputs, decoder_init = None, None
        src_length = torch.as_tensor(src_length, dtype=torch.int64, device='cpu').squeeze(1)

        ### TODO:
        ###     1. feed the "src_ids" into the src embedding layer to get a tensor X of shape (batch_size, max_len, emb_size)
        ###     2. apply "pack_padded_sequence" function to X to get a new tensor X_packed
        ###         (tip: set batch_first=True, enforced_sorted=False in the pack_padded_sequence function)
        ###     3. use Bi-LSTM (rnn) to encode  "X_packed" to get "encoder_outputs", "last_hidden", "last_cell"
        ###     4. apply "pad_packed_sequence" to encoder_outputs (remember to set batch_first=True);
        ###     5. note that last_hidden/last_cell is of shape (2, batch_size, hidden_size);
        ###         we want a shape of (batch_size, 2*hidden_size)
        ###     6. apply linear transformation W_h, W_c to last_hidden/last cell to get the initial decoder hidden state
        ###         (batch_size, hidden_size) and initial decoder cell state (batch_size, hidden_size).
        ### You may use these functions in your implemetation:
        ###     pack_padded_sequence: https://pytorch.org/docs/stable/nn.html#torch.nn.utils.rnn.pack_padded_sequence
        ###     pad_packed_sequence: https://pytorch.org/docs/stable/nn.html#torch.nn.utils.rnn.pad_packed_sequence
        ###     torch.cat: https://pytorch.org/docs/stable/torch.html#torch.cat

        ### YOUR CODE HERE (~ 9 lines)

        tensor_x = self.src_embedding(src_ids)
        tensor_x_packed = pack_padded_sequence(tensor_x, src_length, batch_first=True, enforce_sorted=False)
        encoder_out, l = self.rnn(tensor_x_packed)
        h, c = l
        encoder_outputs, _ = pad_packed_sequence(encoder_out, batch_first=True)

        h = torch.cat((h[0,:,:], h[1,:,:]), dim=1)
        c = torch.cat((c[0,:,:], c[1,:,:]), dim=1)

        h = self.W_h(h)
        c = self.W_c(c)

        decoder_init = (h, c)

        ### END OF YOUR CODE
        return encoder_outputs, decoder_init

class Decoder(nn.Module):
    def __init__(self, decoder_config):
        super(Decoder,self).__init__()
        self.hidden_size = decoder_config['hidden_size']
        self.vocab_size = decoder_config['vocab_size']
        self.emb_size = decoder_config['emb_size']
        self.tgt_emb_matrix = decoder_config['tgt_embedding']

        self.rnn = None
        self.W_u = None
        self.tgt_embedding = None
        self.attention = Attention(self.hidden_size)

        ### TODO Initialize variables:
        #               self.tgt_embedding: nn.Embedding layer for source language; You need to use nn.Embedding function
        #                                   and 2 variables we have initialized for you.
        #                self.rnn: nn.LSTMCell ; You need to use nn.LSTMCell function and 2 variables we have initialized for you.
        #                self.W_u: nn.Linear layer without bias (W_u described in the PDF)

        # For the use of nn.Embedding function, please refer to https://pytorch.org/docs/stable/nn.html#
        # For the use of nn.Linear function, please refer to https://pytorch.org/docs/stable/nn.html#torch.nn.Linear
        # In nn.Linear function, the matrix multiplication is a transposed version of the Eq.(1) in description PDF.
        # For the use of nn.LSTMCell function, please refer to https://pytorch.org/docs/stable/nn.html#lstmcell
        # Think about the shape of \bar{y}_t in the description PDF when initializing self.rnn with nn.LSTMCell

        ### YOUR CODE HERE (4 lines)

        self.tgt_embedding = nn.Embedding(self.vocab_size, self.emb_size)
        self.rnn = nn.LSTMCell(self.emb_size + self.hidden_size, self.hidden_size)
        self.W_u = nn.Linear(3 * self.hidden_size, self.hidden_size, bias=False)

        ### END OF YOUR CODE

        if self.tgt_emb_matrix is not None:
          self.tgt_embedding.weight.data.copy_(torch.Tensor(self.tgt_emb_matrix))
          self.tgt_embedding.weight.requires_grad = True

    def forward(self, tgt_ids, tgt_lengths, encoder_outputs, encoder_output_masks, decoder_init):
        '''
        params:
            tgt_ids: torch.LongTensor of shape (batch_size, max_len); each element is a number specifying the position of
                    a word in a embedding matrix
            tgt_lengths: torch.LongTensor of shape (batch_size,) contains the actual length of each sentence in the batch
            encoder_outputs: torch.FloatTensosr of shape ( batch_size, max_len_in_batch, 2*hidden_size);
                                "max_len_in_batch" is the max length in a batch. It is less than "max_len".
            encoder_output_masks: torch.BoolTensor of shape (batch_size, max_len), specifying which positions are pad tokens.
            decoder_init: tuple(h_0, c_0); the output "decoder_init" of the encoder;
                            h_0 of shape (batch_size, hidden_size), c_0 of shape (batch_size, hidden_size)
        return:
            combined_outputs: torch.FloatTensor of shape (max_len_batch, batch_size, hidden_size)
        '''

        decoder_state = decoder_init
        max_len_batch = torch.max(tgt_lengths) -1               # don't consider the end token
        batch_size = encoder_outputs.size()[0]
        o_prev = torch.zeros(batch_size, self.hidden_size, device='cuda:0' if torch.cuda.is_available() else 'cpu')

        combined_outputs = []

        ### TODO:
        ###     1. feed the "tgt_ids" into the embedding layer to get a tensor "Y" of shape (batch_size, max_len, emb_size)
        ###     2. construct a for loop with range 0:max_len_batch
        ###         within the for loop:
        ###                             1). slice Y by indexing; you should have y_t of shape (batch_size, emb_size)
        ###                             2). concatenate y_t with o_prev , yielding ybar_t as described in the PDF
        ###                             3). feed ybar_t and "decoder state", "encoder_outputs", "encoder_output_masks" into function "decode_one_step()"
        ###                                 and it will output new "decoder_state" (a tuple)
```

```
, new "o_t"
        ###                         4). append new "o_t" to "combined_outputs"
        ###                         5). update "o_prev" with new "o_t"
        ###     3. use "torch.stack" function to process combined_outputs (a list of tens
ors; each tensor of shape (batch_size, hidden_size)) to
        ###          a single tensor of shape (max_len_batch, batch_size, hidden_size)
        ###
        ### You may use these functions in your implementation:
        ###     torch.cat: https://pytorch.org/docs/stable/torch.html#torch.cat
        ###     torch.stack: https://pytorch.org/docs/stable/torch.html#torch.stack
        ### YOUR CODE HERE (~ 8 lines)

        Y = self.tgt_embedding(tgt_ids)

        for i in range(0, max_len_batch):
          ybar_t = torch.cat((Y[:,i,:], o_prev), dim=1)
          decoder_state, o_t = self.decode_one_step(ybar_t,
                                                    decoder_state,
                                                    encoder_outputs,
                                                    encoder_output_masks)
          combined_outputs.append(o_t)
          o_prev = o_t

        combined_outputs = torch.stack(combined_outputs,
                                       dim=0)

        ### END OF YOUR CODE
        return combined_outputs

    def decode_one_step(self, ybar_t, decoder_state, encoder_outputs, encoder_output_mask
s):
        '''
        param:
            ybar_t: torch.FloatTensor of shape (batch_size, emb_size + hidden_size)
            decoder_state: tuple(h_t, c_t); h_t of shape (batch_size, hidden_size); c_t o
f shape (batch_size, hidden_size);
            encoder_hiddens: torch.FloatTensosr of shape ( batch_size, max_len_in_batch,
2*hidden_size); "max_len_in_batch" is the max length in a batch. It is less than "max_len
".
            encoder_hidden_masks: torch.BoolTensor of shape (batch_size, max_len), speci
fying which positions are pad tokens.
        return:
            decoder_state: tuple(h_t, c_t); both h_t and c_t have a shape (batch_size, hi
dden_size)
            o_t: torch.FloatTensor of shape (batch_size, hidden_size)
        '''
        ### TODO:
        ###     1. Apply the decoder (self.rnn) to "ybar_t", "decoder_state", yielding a
new "decoder_state"
        ###     2. split the decoder state into two parts, "h" and "c"; h has a shape (ba
tch_size, hidden_size); c has a shape (batch_size, hidden_size)
        ###     3. apply the "Attention" module to "h", "encoder_outputs", "encoder_outpu
t_masks", yielding attention weights (alpha_t in the PDF) of shape (batch_size, max_len_i
n_batch)
        ###     4. apply torch.bmm function to alpha_t and "encoder_hiddens", yielding th
e "a_t" in PDF.
        ###        You also need to use "unsqueeze" and "squeeze" function here. Be sure
to specify the "dim" parameter in these two functions.
        ###        "a_t" has a shape (batch_size, 2*hidden_size)
        ###     5. concatenate "a_t" and "h", yielding "u_t" in the PDF; "u_t" has a shap
e (batch_size, 3*hidden_size)
        ###     6. apply linear transformation W_u and "torch.tanh" function to "u_t", yi
elding "o_t" of shape (batch_size, hidden_size)

        ### You may use these functions in your implementation:
        ###     torch.cat: https://pytorch.org/docs/stable/torch.html#torch.cat
        ###     torch.bmm: https://pytorch.org/docs/stable/torch.html#torch.bmm
        ###     torch.tanh: https://pytorch.org/docs/stable/torch.html#torch.tanh
        ###     torch.squeeze: https://pytorch.org/docs/stable/torch.html#torch.squeeze
        ###     torch.unsqueeze: https://pytorch.org/docs/stable/torch.html#torch.unsquee
ze
```

```
        ### YOUR CODE HERE (~6 lines)

        decoder_state = self.rnn(ybar_t, decoder_state)
        h, c = decoder_state
        attention_w = self.attention(h, encoder_outputs, encoder_output_masks)
        attention_w = torch.bmm(attention_w.unsqueeze(dim=1), encoder_outputs).squeeze(1
)

        u_t = torch.cat((h, attention_w), dim = 1)
        o_t = torch.tanh(self.W_u(u_t))

        ## END OF YOUR CODE

        return decoder_state, o_t

class Attention(nn.Module):
    def __init__(self, hidden_size):
        super(Attention,self).__init__()

        ### TODO Initialize variables:
        #               self.W_outputs: nn.Linear layer without bias (W_outputs describl
ed in the PDF);
        #               self.W_combined: nn.Linear layer without bias (W_combined descri
bled in the PDF)
        #               self.W_alignment: nn.Parameter layer (W_alignment describled in
the PDF)

        # For the use of nn.Linear function, please refer to https://pytorch.org/docs/sta
ble/nn.html#torch.nn.Linear
        #   In nn.Linear function, the matrix multiplication is a transposed version of t
he Eq.(1) in description PDF.
        # For the use of nn.Parameter function, please refer to https://pytorch.org/docs/
stable/generated/torch.nn.parameter.Parameter.html
        #   This is used as "weights" matrix for the alignment scores

        ### YOUR CODE HERE (4~6 lines)

        self.W_outputs = nn.Linear(2 * hidden_size, hidden_size, bias=False)
        self.W_combined = nn.Linear(hidden_size, hidden_size, bias=False)
        self.W_alignment = nn.Parameter(torch.FloatTensor(hidden_size, 1))

        ### END OF YOUR CODE

    def forward(self, decoder_hiddens, encoder_outputs, encoder_output_masks):
        '''
        compute the attention weights \alpha_t in the PDF
        param:
            h: torch.FloatTensor of shape (batch_size, hidden_size)
            encoder_outputs: torch.FloatTensosr of shape ( batch_size, max_len_in_batch,
2*hidden_size); "max_len_in_batch" is the max length in a batch. It is less than "max_len
".
            encoder_output_masks: torch.BoolTensor of shape (batch_size, max_len), speci
fying which positions are pad tokens. False -- pad token; True -- not pad token
        return:
            attn_weights: torch.FloatTensor of shape (batch_size, max_len_in_batch)
        '''


        decoder_hiddens = decoder_hiddens.unsqueeze(1)                  ### (batch_size,
hidden_size, 1)
        max_len_in_batch = encoder_outputs.size()[1]

        ### TODO:
        ###     1. apply linear transformation "W_outputs" to "encoder_outputs"; the resu
lt has  a shape (batch_size, max_len_in_batch, hidden_size)
        ###     2. add the "decoder_hiddens" and the "encoder_outputs" together; the resu
lt has  a shape (batch_size, max_len_in_batch + 1, hidden_size)
        ###     3. apply linear transformation "W_combined" to "concatenated outputs"; th
e result has  a shape (batch_size, max_len_in_batch + 1, hidden_size)
        ###     4. apply Tanh function to the linear-transformed "concatenated outputs";
the result has  a shape (batch_size, max_len_in_batch + 1, hidden_size)
        ###     5. apply torch.matmul to the result of step 4 and "W_alignment", yielding
```

```
score e_t of shape (batch_size, max_len_in_batch, 1);
        ###        squeeze e_t in the last dimension
        ###     6. apply torch.Tensor.masked_fill_() function to "e_t"; the parameters of
this function are Bool tensor "encoder_output_masks" and a constant "-float('inf')";
        ###        before "torch.Tensor.masked_fill_()" function, this "encoder_output_m
asks" should be sliced to have a shape (batch_size, max_len_in_batch) (Only the first max
_len_in_batch columns will be kept)
        ###     7. apply "F.softmax()" function to "e_t", yielding "alpha_t" of shape (ba
tch_size, max_len_in_batch)

        ### You may use these functions in your implementation:
        ###     torch.matmul: https://pytorch.org/docs/stable/generated/torch.matmul.html
        ###     torch.squeeze: https://pytorch.org/docs/stable/torch.html#torch.squeeze
        ###     torch.Tensor.masked_fill_: https://pytorch.org/docs/stable/tensors.html#t
orch.Tensor.masked_fill_
        ###     F.softmax: https://pytorch.org/docs/stable/nn.functional.html#torch.nn.fu
nctional.softmax

        ### YOUR CODE HERE (4~6 lines)

        encoder_outputs = self.W_outputs(encoder_outputs)

        _sum = decoder_hiddens + encoder_outputs
        linear_sum = self.W_combined(_sum)
        tanh_sum = torch.tanh(linear_sum)
        e_t = torch.matmul(tanh_sum, self.W_alignment).squeeze(-1)

        encoder_output_masks = encoder_output_masks[:, 0:max_len_in_batch]
        encoder_output_masks = ~encoder_output_masks
        e_t.masked_fill_(encoder_output_masks, -float('inf'))
        attn_weights = F.softmax(e_t, 1)

        ### END OF YOUR CODE

        return attn_weights

class NMT(nn.Module):
    def __init__(self, encoder_config, decoder_config):
        super(NMT, self).__init__()
        self.encoder = Encoder(encoder_config)
        self.decoder = Decoder(decoder_config)
        self.encoder_config = encoder_config
        self.decoder_config = decoder_config
        self.W_v = nn.Linear(decoder_config['hidden_size'], decoder_config['vocab_size']
)

    def forward(self, src_ids, src_lengths, src_masks, tgt_ids, tgt_lengths, tgt_masks):
        # src_ids:(batch_size, max_len)
        # src_lengths: (batch_size)
        # src_mask: (batch_size, max_len)
        # tgt_ids: (batch_size, max_len)
        # tgt_lengths: (batch_size)
        # tgt_masks: (batch_size, max_len)

        encoder_outputs, decoder_init_hidden = self.encoder(src_ids, src_lengths)
        outputs = self.decoder(tgt_ids, tgt_lengths, encoder_outputs, src_masks, decoder
_init_hidden)
        tgt_unnormalized_score = self.W_v(outputs)
        tgt_log_prob = F.log_softmax(tgt_unnormalized_score, dim=-1)

        max_len_batch = torch.max(tgt_lengths)
        tgt_masks = tgt_masks[:, :max_len_batch].permute(1, 0) # (l,b)
        tgt_ids = tgt_ids.permute(1,0)[:max_len_batch, :] #(l,b)
        tgt_words_log_prob = torch.gather(tgt_log_prob, -1, tgt_ids[1:].unsqueeze(-1)).s
queeze(-1) * tgt_masks[1:].float()

        tgt_sents_log_prob = torch.sum(tgt_words_log_prob, dim=0)
        return tgt_sents_log_prob       #(b)


    def beam_search(self, src_ids, src_length, beam_size):
        # src_ids: (batch_size, max_len)
```

```python
        # src_lengths: (1, 1)
        # beam_size: int

        STOP_ID = self.decoder_config['en_w2i'][STOP_TOKEN]
        max_decode_length = 30
        encoder_outputs, decoder_init_hidden = self.encoder(src_ids, src_length)
        encoder_output_masks = torch.BoolTensor(np.ones((1,src_length.item()))).to(device)

        START_ID = self.decoder_config['en_w2i']['<s>']
        prev_o_t = torch.zeros(1, self.decoder_config['hidden_size']).to(device)
        input_beam_nodes = [BeamNode(prev_node=None, prev_hidden=decoder_init_hidden, prev_o_t=prev_o_t , wordID=START_ID,
                            score=0, length=1)]

        finished_beam = 0
        end_beam = []
        max_finished_beam = beam_size
        while finished_beam < max_finished_beam and input_beam_nodes[0].length < max_decode_length:
            cur_hidden = []
            cur_o_t = []
            prev_scores = []
            cur_len = input_beam_nodes[0].length

            for n in input_beam_nodes:
                y_t = self.decoder.tgt_embedding(torch.LongTensor([n.wordID]).to(device))
                y_t = torch.cat((y_t, n.prev_o_t), dim=1)

                decoder_hidden, o_t = self.decoder.decode_one_step(y_t, n.prev_hidden, encoder_outputs, encoder_output_masks)
                cur_hidden.append(decoder_hidden)
                cur_o_t.append(o_t)
                prev_scores.append(n.score)

            o_t = torch.stack(cur_o_t, dim=0)
            scores = self.W_v(o_t).squeeze(1)
            prev_scores = torch.Tensor(prev_scores).unsqueeze(-1).expand_as(scores).to(device)

            assert len(scores.size()) == 2
            assert scores.size(0) == len(input_beam_nodes)
            assert scores.size(1) == self.decoder_config['vocab_size']

            log_prob = F.log_softmax(scores, dim=-1)
            cur_score = (log_prob + prev_scores).view(-1)
            topk_score, topk_pos = torch.topk(cur_score, beam_size)

            node_ids = topk_pos // self.decoder_config['vocab_size']
            word_ids = topk_pos % self.decoder_config['vocab_size']

            next_nodes = []
            for score, node_id, word_id in zip(topk_score, node_ids, word_ids):
                score = score.item()
                node_id = node_id.item()
                word_id = word_id.item()

                node = BeamNode(prev_node=input_beam_nodes[node_id], prev_hidden=cur_hidden[node_id],
                                prev_o_t=cur_o_t[node_id] , score=score,
                                wordID=word_id, length=cur_len+1)

                if word_id == STOP_ID:
                    beam_size -= 1
                    end_beam.append(node)
                    finished_beam += 1
                else:
                    next_nodes.append(node)

            input_beam_nodes = next_nodes
            if cur_len + 1 >= max_decode_length:
                end_beam.extend(next_nodes)

        seqs = []
        for n in end_beam:
            seq = []
            score = n.score
            while True:
                prev_node = n.prev_node
                wordID = n.wordID
                try:
                    word = self.decoder_config['en_i2w'][wordID]
                except KeyError:
                    word = UNK_TOKEN
                # print(word)
                seq.append(word)
                if prev_node.wordID == START_ID:
                    break
                n = prev_node
            seqs.append(Translation(sent=seq[-1:0:-1], score=score))

        return seqs
```

metric

In [10]:

```
device = 'cuda:0' if torch.cuda.is_available else 'cpu'

def eval_ppl(model, dev_iter):
    model.eval()

    cum_loss = 0.
    cum_tgt_words = 0.

    with torch.no_grad():
        for batch_data in dev_iter:
            batch_data = tuple(t.to(device) for t in batch_data[:-1])
            b_src_ids, b_tgt_ids, b_src_len, b_tgt_len, b_src_mask, b_tgt_mask = batch_d
ata
            batch_loss = -1 * model(b_src_ids, b_src_len, b_src_mask, b_tgt_ids, b_tgt_l
en, b_tgt_mask).sum()
            cum_loss += batch_loss.item()
            b_num_words = b_tgt_len.sum() - b_tgt_len.size(0)
            cum_tgt_words += b_num_words

        ppl = np.exp(cum_loss/cum_tgt_words.item())

    model.train()
    return ppl

def compute_corpus_bleu_score(references, predictions):
    # references: List[List[str]]
    # prediction: Liset[List[str]]
    return corpus_bleu([[ref] for ref in references], predictions)
```

main

In [11]:

```
def train(train_iter, dev_iter, encoder_config, decoder_config, epoch):

    model = NMT(encoder_config, decoder_config)
    model = model.to(device)
    model.train()

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    best_eval_ppl = float('inf')

    for it in range(epoch):
        total_train_loss = 0.
        total_train_words = 0

        for batch_data in train_iter:
            batch_data = tuple(t.to(device) for t in batch_data[:-1])
            b_src_ids, b_tgt_ids, b_src_len, b_tgt_len, b_src_mask, b_tgt_mask = batch_d
ata

            optimizer.zero_grad()

            batch_loss = -1 * model(b_src_ids, b_src_len, b_src_mask, b_tgt_ids, b_tgt_l
en, b_tgt_mask).sum()
            loss = batch_loss / batch_size

            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 5)
            optimizer.step()

            total_train_loss += batch_loss.item()
            total_train_words += b_tgt_len.sum() - b_tgt_len.size(0)

        print('train_loss:{}, train_ppl:{} '.format(total_train_loss/batch_size, np.exp(
total_train_loss/total_train_words.item())))

        e_ppl = eval_ppl(model, dev_iter)
        if e_ppl < best_eval_ppl:
            print('better model found!')
            print('eval_ppl:', e_ppl)
            torch.save(model.state_dict(), 'best_model.pt')
            best_eval_ppl = e_ppl

def test(model, test_iter):
    # support only batch_size = 1
    model.eval()
    corpus_reference = []
    corpus_prediction = []
    with torch.no_grad():
        for batch_data in test_iter:
            # Run Inference
            raw_sent = batch_data[-1]
            batch_data = tuple(t.to(device) for t in batch_data[:-1])
            b_src_ids, b_tgt_ids, b_src_len, b_tgt_len, b_src_mask, b_tgt_mask = batch_d
ata

            seqs = model.beam_search(b_src_ids, b_src_len, 2)
            sorted_seqs = sorted(seqs, key=lambda x: x.score, reverse=True)
            corpus_prediction.append(sorted_seqs[0].sent)

            # Get Raw Sentence
            ref = list(tokenizer(x) for x in raw_sent)
            corpus_reference += ref

        bleu = compute_corpus_bleu_score(corpus_reference, corpus_prediction)
        print('BLEU score on Test set:{}'.format(bleu))

#--------------------------------------------------------------------------------------
-----------
```

In [12]:

```
fr_emb = None
en_emb = None

encoder_config = {'hidden_size': 256,
                  'num_layers': 1,
                  'bidirectional':True,
                  'vocab_size':  len(fr_w2i),
                  'emb_size':300,
                  'src_embedding': fr_emb}

decoder_config = {'hidden_size': 256,
                  'vocab_size': len(en_w2i),
                  'emb_size': 300,
                  'tgt_embedding': en_emb,
                  'en_w2i':en_w2i,
                  'en_i2w':en_i2w}
max_len = 30
batch_size = 32
tokenizer = lambda x: x.split()

train_dataset = TranslationDataset(train_sent_pairs, fr_w2i, en_w2i, tokenizer, max_len)
dev_dataset = TranslationDataset(dev_sent_pairs, fr_w2i, en_w2i, tokenizer, max_len)
test_dataset = TranslationDataset(test_sent_pairs, fr_w2i, en_w2i, tokenizer, max_len)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
dev_loader = DataLoader(dev_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)

device = 'cuda:0' if torch.cuda.is_available else 'cpu'
epoch = 8

train(train_loader, dev_loader, encoder_config, decoder_config, epoch)
model = NMT(encoder_config, decoder_config)
model.load_state_dict(torch.load(r'best_model.pt'))
model = model.to(device)
test(model, test_loader)
```

```
train_loss:107934.74969863892, train_ppl:31.51580308173513
better model found!
eval_ppl: 12.920242120630345
train_loss:65049.0252494812, train_ppl:8.000515187891743
better model found!
eval_ppl: 8.400872037758555
train_loss:47960.85104227066, train_ppl:4.633100381775522
better model found!
eval_ppl: 7.226042409882179
train_loss:37530.700488090515, train_ppl:3.3194271462652907
better model found!
eval_ppl: 6.799640881773428
train_loss:30458.435219049454, train_ppl:2.6477335572254534
better model found!
eval_ppl: 6.710091781475826
train_loss:25579.061647892, train_ppl:2.265326266740426
train_loss:22321.147998571396, train_ppl:2.041263289248497
train_loss:20076.605898857117, train_ppl:1.899925564018366
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:382: UserWarning: __floordiv
__ is deprecated, and its behavior will change in a future version of pytorch. It current
ly rounds toward 0 (like the 'trunc' function NOT 'floor'). This results in incorrect rou
nding for negative values. To keep the current behavior, use torch.div(a, b, rounding_mod
e='trunc'), or for actual floor division, use torch.div(a, b, rounding_mode='floor').
```

```
BLEU score on Test set:0.334477168766918
```