LING 530F DL-NLP Project: Automatic Text Summarization

Zicong Fan 11205168 zfan@alumni.ubc.ca

Si Yi (Cathy) Meng 32939118 mengxixi@cs.ubc.ca

Zixuan Yin 11687143 krystal_yzx@naver.com

In [2]:
```
import os
import json
import time
import math
import random
import shutil
import datetime
import logging
import pickle
import collections

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
from torch.nn.utils.rnn import pad_packed_sequence, pack_padded_sequence
from torch.optim.lr_scheduler import StepLR
from allennlp.modules.elmo import Elmo, batch_to_ids
from pyrouge import Rouge155
```

In [3]:
```
# logging configurations
LOG_FORMAT = "%(asctime)s %(message)s"
logging.basicConfig(level=logging.INFO, format=LOG_FORMAT, datefmt="%H:%M:%S")

# seeding for reproducibility
random.seed(1)
np.random.seed(2)
torch.manual_seed(3)
torch.cuda.manual_seed(4)

# define directory structure needed for data processing
TMP_DIR = os.path.join("..", "data", "tmp")
TRAIN_DIR = os.path.join("..", "data", "gigaword","train_sample")
DEV_DIR = os.path.join("..", "data", "gigaword","valid")
TEST_DIR = os.path.join("..", "data", "gigaword","test")
MODEL_DIR = os.path.join("..", 'models')
CHECKPOINT_FNAME = "gigaword-1127-2ep.ckpt"
GOLD_DIR = os.path.join(TMP_DIR, "gold")
SYSTEM_DIR = os.path.join(TMP_DIR, "system")
TRUE_HEADLINE_FNAME = 'gold.A.0.txt'
PRED_HEADLINE_FNAME = 'system.0.txt'

for d in [TRAIN_DIR, DEV_DIR, TEST_DIR, TMP_DIR, GOLD_DIR, SYSTEM_DIR, MODEL_DIR]:
    if not os.path.exists(d):
        os.makedirs(d)
```

Extract text body and headline from the Annotated English Gigaword dataset

- This was a script ran separately (modified based on the provided preprocessing script)
- Here we use the `CommunicationReader` in the `concrete` package to read the xml files
    - After extracting the paired headline and body, we tokenize them using `nltk`
    - We lowercased all tokens
    - Removed punctuations
    - Removed pairs where headline had less than 3 tokens

In [ ]:
```
from concrete.util import CommunicationReader
from concrete.util import lun, get_tokens
import json
import os
import glob
import nltk
from nltk.tokenize import word_tokenize
import string
import regex as re
import threading
import queue
import sys
import time

from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()


def readData(data_path):
    '''
    data_path -- path to the file that contains the preprossed data
    '''
    '''return a list of object {'Headline': string, 'Text': string}'''
    data = []
    with open(data_path) as f:
        for line in f:
            obj = json.loads(line)
            data.append(obj)
    return data


def worker(in_queue, out_queue):
    while not stopping.is_set():
```

```
        try:
            tar_file = in_queue.get(True, timeout=1)
            print("Processing %s" % tar_file)
            t = time.time()
            res = preprocess(tar_file, OUTPUT_PATH)
            print("Elapsed Time %.2f"%(time.time() - t))
            out_queue.put(res)

        except:
            continue

        in_queue.task_done()


def preprocess(tar_path, output_path):
    '''
    tar_path  -- tar file to process
    output_path -- directory of the output file
                   each line of the output file has the format {'Headline': string, 'Text': string}
    '''

    fname = "%s.txt" % tar_path.split('/')[-1].split('.')[0]
    out_fname = os.path.join(output_path, fname)

    mem = {}

    with open(out_fname, 'w') as f:
        for (comm, filename) in CommunicationReader(tar_path):
            text = comm.text
            headline_start = text.find("<HEADLINE>")
            headline_end = text.find('</HEADLINE>',headline_start)
            par1_start = text.find("<P>",headline_end)
            par1_end = text.find("</P>",par1_start)
            headline = text[headline_start + len('<HEADLINE>'):headline_end].strip()
            par1 = text[par1_start + len("<P>"):par1_end].strip()
            if headline in mem.keys():
                continue
            else:
                mem[headline] = par1

            # print(headline)
            # print(par1)
```

```
            #process healline
            if comm.id.startswith("XIN"):
                #for xinhua headline, remove anything before : or anything after :
                #Example sentences that need to be modified:
                #Roundup: Gulf Arab markets end on a mixed note
                #Israelis more distrustful of gov't institutions: survey
                a = headline.find(":")
                if a != -1:
                    b = headline.rfind(":")
                    if a == b:
                        if a < len(headline) / 2:
                            headline = headline[a + 1:]
                        else:
                            headline = headline[:b]
                    else:
                        headline = headline[a + 1:b]
            headline_token = word_tokenize(headline)
            #remove punctuations
            headline_token = [t.strip(string.punctuation).lower() for t in headline_token]

            #ignore if headline is too short
            if len(headline_token) < 3:
                continue

            #process the first paragraph
            par1_token = word_tokenize(par1)
            #remove punctuations
            par1_token = [t.strip(string.punctuation).lower() for t in par1_token]

            headline = " ".join([t for t in headline_token])
            par1 = " ".join([t for t in par1_token])
            obj = {'Headline': headline, "Text": par1}
            json_str = json.dumps(obj)
            f.write(json_str + '\n')
    print("completed file %s" % fname)
    return fname

with open('todolist1.txt') as f:
    content = f.readlines()
SOURCES = [x.strip() for x in content]
print(SOURCES)
```

```python
tars = []
for s in SOURCES:
    tars.extend(glob.glob(os.path.join("/media/sda1/gigaword/data/gigaword", s)))


OUTPUT_PATH = os.path.join(".", 'gigaword')
if not os.path.exists(OUTPUT_PATH):
    os.makedirs(OUTPUT_PATH)



stopping = threading.Event()

work = queue.Queue()
results = queue.Queue()
total = len(tars)

# start for workers
for i in range(4):
    t = threading.Thread(target=worker, args=(work, results))
    t.daemon = True
    t.start()

# produce data
for i in range(total):
    work.put(tars[i])

print("waiting for workers to finish")
work.join()
stopping.set()

# get the results
for i in range(total):
    print(results.get())

sys.exit()
```

Downsampling the training set

- The entire training set would yield a vocabulary that's too big for our memory even after removing low frequency tokens
- Therefore we downsample the training set by randomly dropping data pairs with probability 0.4

In [ ]:
```
# downsample the training set by dropping pairs with probability 0.4
random.seed(0)   # here we used a different seed since it was ran as a separate script
output = os.path.join(".","data","gigaword","train_sample.txt")
with open(output,'w+') as fo:
    for fname in os.listdir(TRAIN_DIR):
        fpath = os.path.join(TRAIN_DIR, fname)
        with open(fpath) as fin:
            for line in fin:
                tmp = random.random()
                if tmp < 0.4:
                    continue
                fo.write(line)
```

In [4]:
```
PAD_token = 0  # padding
SOS_token = 1  # start of sentence
EOS_token = 2  # end of sentence
UNKNOWN_TOKEN = 'unk'

MAX_OUTPUT_LENGTH = 35    # max length of summary generated
MAX_HEADLINE_LENGTH = 30  # max length of headline (target) from the data
MAX_TEXT_LENGTH = 50      # max length of text body from the data
MIN_TEXT_LENGTH = 5       # min length of text body for it to be a valid data point
MIN_FREQUENCY   = 4       # token with frequency <= MIN_FREQUENCY will be converted to 'unk'
MIN_KNOWN_COUNT = 3       # headline (target) must have at least MIN_KNOWN_COUNT number of known toke
ns

EMBEDDING_DIM = 256
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Preprocess tokenized data

- First, we build a frequency dict on the downsampled training set (referred to as the training set hereafter), including all words from text body and headline
- Then we further process the training data
  - **Truncate text body to MAX_TEXT_LENGTH**
  - **Removed pairs where headline is too long (our aim is to generate concise 1-liner summaries)**
  - **Removed pairs where body is too short (barely anything to summarize from)**
  - **Removed pairs where headline does not have enough known (frequent) words**
  - **Replaced all low frequency words with the 'unk' token**
- We sorted all the tokens based on their frequency (from high to low)
  - This is needed for Adaptive Softmax, explained in the paper
- Finally, we build the word2index and the reverse mapping based on the sorted frequencies, giving each token an index based on how often they appear
  - PAD, SOS and EOS appear in every sentence, so it makes sense to put them at the first 3 indices
- We also pickle the data objects (train/dev/test data, word2idx and its reverse map) to allow us directly load them from disk without repetitively processing them to save time

In [5]:
```
pkl_names = ['train_data', 'dev_data', 'test_data', 'word2index', 'index2word']
pickles = []
```

In [6]:
```
vocab_freq_dict = {}
WORD_2_INDEX = {"PAD": 0, "SOS": 1, "EOS": 2}
INDEX_2_WORD = {0: "PAD", 1: "SOS", 2: "EOS"}

def update_freq_dict(freq_dict, tokens):
    for t in tokens:
        if t not in freq_dict:
            freq_dict[t] = 0
        freq_dict[t] += 1

def build_freq_dict(data_dir):
    freq_dict = dict()
    for fname in os.listdir(data_dir):
        logging.info("Working on file: " + fname)
        fpath = os.path.join(data_dir, fname)
        with open(fpath) as f:
            for line in f:
                obj = json.loads(line)
                headline = [t for t in obj['Headline'].split()]
                text = [t for t in obj['Text'].split()]
                update_freq_dict(freq_dict, headline)
                update_freq_dict(freq_dict, text)
    return freq_dict

def remove_low_freq_words(freq_dict, tokens):
    filtered_tokens = []
    known_count = 0
    for t in tokens:
        if freq_dict[t] > MIN_FREQUENCY:
            filtered_tokens.append(t)
            known_count += 1
        else:
            filtered_tokens.append(UNKNOWN_TOKEN)
    return filtered_tokens, known_count


def update_word_index(word2index, index2word, tokens):
    for t in tokens:
        if t not in word2index:
            next_index = len(word2index)
            word2index[t] = next_index
            index2word[next_index] = t
```

```
def read_data(data_dir):
    ignore_count = [0,0,0]
    data = []
    unk_count = 0
    for fname in os.listdir(data_dir):

        fpath = os.path.join(data_dir, fname)
        with open(fpath) as f:
            for line in f:
                obj = json.loads(line)
                headline = [t for t in obj['Headline'].split()]
                text = [t for t in obj['Text'].split()][:MAX_TEXT_LENGTH]
                if data_dir == TRAIN_DIR:
                    if len(headline) > MAX_HEADLINE_LENGTH:
                        ignore_count[1] += 1
                        continue
                    if len(text) < MIN_TEXT_LENGTH:
                        ignore_count[2] +=1
                        continue
                    headline, known_count = remove_low_freq_words(freq_dict, headline)
                    if known_count < MIN_KNOWN_COUNT:
                        ignore_count[0] += 1
                        continue

                    text, _ = remove_low_freq_words(freq_dict, text)
                    for token in (headline + text):
                        if token == 'unk':
                            unk_count += 1
                        elif token not in vocab_freq_dict.keys():
                            vocab_freq_dict[token] = freq_dict[token]

                data.append((headline, text))

    # Now ready to build word indexes
    if data_dir == TRAIN_DIR:
        vocab_freq_dict['unk'] = unk_count
        sorted_words = sorted(vocab_freq_dict, key=vocab_freq_dict.get, reverse=True)
        update_word_index(WORD_2_INDEX, INDEX_2_WORD, sorted_words)

    return data, ignore_count

logging.info("Building frequency dict on TRAIN data...")
```

```
freq_dict = build_freq_dict(TRAIN_DIR)

logging.info("Number of unique tokens: %d", len(freq_dict))

logging.info("Load TRAIN data and remove low frequency tokens...")
train_data, ignore_count = read_data(TRAIN_DIR)
assert len(WORD_2_INDEX) == len(INDEX_2_WORD)
VOCAB_SIZE = len(WORD_2_INDEX)

logging.info("Removed %d pairs due to not enough known words in headline", ignore_count[0])
logging.info("Removed %d pairs due to headline length greater than MAX_HEADLINE_LENGTH", ignore_count
[1])
logging.info("Removed %d pairs due to text length less than MIN_TEXT_LENGTH", ignore_count[2])
logging.info("Number of unique tokens after replacing low frequency ones: %d", VOCAB_SIZE)

logging.info("Load DEV data...")
dev_data, _ = read_data(DEV_DIR)

logging.info("Load TEST data and take a random subset of 2000 valid pairs...")
test_data, _ = read_data(TEST_DIR)
test_data = [data for data in test_data if len(data[1])>0]
random.shuffle(test_data)
test_data = test_data[:2000]

# persist data objects
for i, item in enumerate([train_data, dev_data, test_data, WORD_2_INDEX, INDEX_2_WORD]):
    with open(os.path.join(TMP_DIR, pkl_names[i]+".pkl"), 'wb') as handle:
        pickle.dump(item, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
00:53:38 Building frequency dict on TRAIN data...
00:53:38 Working on file: train_sample.txt
00:54:33 Number of unique tokens: 1016085
00:54:33 Load TRAIN data and remove low frequency tokens...
00:56:01 Removed 10969 pairs due to not enough known words in headline
00:56:01 Removed 73576 pairs due to headline length greater than MAX_HEADLINE_LENGTH
00:56:01 Removed 66603 pairs due to text length less than MIN_TEXT_LENGTH
00:56:01 Number of unique tokens after replacing low frequency ones: 214322
00:56:01 Load DEV data...
00:56:03 Load TEST data and take a random subset of 2000 valid pairs...
```

Load pickles without re-loading from scratch

In [7]:
```
for i, name in enumerate(pkl_names):
    with open(os.path.join(TMP_DIR, name+'.pkl'), 'rb') as handle:
        pickles.append(pickle.load(handle))
train_data = pickles[0]
dev_data = pickles[1]
test_data = pickles[2]
WORD_2_INDEX = pickles[3]
INDEX_2_WORD = pickles[4]

assert len(WORD_2_INDEX) == len(INDEX_2_WORD)
VOCAB_SIZE = len(WORD_2_INDEX)
print("Number of training examples: ", len(train_data))
print("Number of dev examples: ", len(dev_data))
print("Number of test examples: ", len(test_data))
print("Vocabulary size: ", VOCAB_SIZE)
```

```
Number of training examples:  3768318
Number of dev examples:  346462
Number of test examples:  2000
Vocabulary size:  214322
```

Closer look at our data

In [10]:
```
headline_lens = []
text_lens = []
for headline, text in train_data:
    headline_lens.append(len(headline))
    text_lens.append(len(text))
avg_headline = np.mean(np.asarray(headline_lens))
avg_text = np.mean(np.asarray(text_lens))
med_headline = np.median(np.asarray(headline_lens))
med_text = np.median(np.asarray(text_lens))
print("Average headline length: ", avg_headline)
print("Median headline length: ", med_headline)
print("Average text length: ", avg_text)
print("Median text length: ", med_text)
c = collections.Counter(vocab_freq_dict)
print("Most common 20 words: ")
print('\n'.join(map(str, c.most_common(20))))
```

```
Average headline length:  8.774473120368292
Median headline length:  8.0
Average text length:  29.69044358782884
Median text length:  29.0
Most common 20 words:
('the', 9853746)
('p', 5444407)
('to', 5383613)
('of', 5143127)
('in', 4837902)
('a', 4805019)
('and', 3446732)
('s', 2464411)
('on', 2431951)
('for', 2134049)
('said', 1423173)
('that', 1395990)
('with', 1348119)
('by', 1205789)
('at', 1136133)
('is', 1061512)
('as', 1029065)
('from', 905984)
('was', 870863)
('it', 853869)
```

- Example data pairs from train data

In [21]:
```
sample = random.sample(train_data, 3)
for headline, text in sample:
    print("Headline: ", " ".join(headline))
    print("Text: ", " ".join(text))
    print("\n")
```

```
Headline:  powell rules out dominant role for pakistan in postwar afghanistan eds updates with powel
l statement on taliban
Text:  secretary of state colin powell is ruling out a dominant role for pakistan or any other natio
n in afghanistan s postwar government


Headline:  urgent guy philippe i will not disarm
Text:  haitian rebel leader guy philippe said tuesday that he will not disarm despite international
pressure


Headline:  u.s firm unveils portable satellite phone system for southeast asia
Text:  a u.s company launched a satellite phone service in asia tuesday that uses a laptop unk instr
ument to make calls send faxes and transmit data
```

Load ELMo Embeddings

- We use the ELMo model with dimension 256 to generate pre_trained embeddings for our vocabulary
- Since ELMo is context-based, meaning it may give a different embedding for a token that appears in a different sentence, we need to perform the following
  - Pass in the entire training set (where the vocabulary is taken from)
  - For each pair, we concatenate the text body and the headline as if it was all in one sentence, and pass that into ELMo (in a batch)
  - For each embedding we get back, we check if we already have an embedding for this token, if we do, we'll take the average of the embeddings for this same token (over all counts of this token)
- Since this process takes hours, we ran it once and pickle the result

In [ ]:
```
options_file = "https://s3-us-west-2.amazonaws.com/allennlp/models/elmo/2x1024_128_2048cnn_1xhighway/
elmo_2x1024_128_2048cnn_1xhighway_options.json"
weight_file = "https://s3-us-west-2.amazonaws.com/allennlp/models/elmo/2x1024_128_2048cnn_1xhighway/e
lmo_2x1024_128_2048cnn_1xhighway_weights.hdf5"

class ELMoEmbedding():
    def __init__(self, corpus, options, weights, dim, batch_size=32):
        self.elmo = Elmo(options, weights, 1, dropout=0).to(device)
        self.dim = dim
        self.corpus = corpus
        self.word_embedding_dict = {}

        # Start loading embeddings
        random.shuffle(corpus)
        end_index = len(corpus) - len(corpus) % batch_size
        input_seqs = []
        target_seqs = []

        # Choose random pairs
        for i in range(0, end_index, batch_size):
            pairs = corpus[i:i+batch_size]
            sentences = [pair[0] + pair[1] for pair in pairs]
            character_ids = batch_to_ids(sentences).to(device)
            embeddings = self.elmo(character_ids)["elmo_representations"][0].cpu().data.numpy()

            for i, sent in enumerate(sentences):
                for j, token in enumerate(sent):
                    token_count = freq_dict[token]
                    token_emb = embeddings[i,j,:]
                    if token not in self.word_embedding_dict.keys():
                        self.word_embedding_dict[token] = token_emb/token_count

                    else:
                        token_emb = np.sum([token_emb/token_count, self.word_embedding_dict[token]],
axis=0)
                        self.word_embedding_dict[token] = token_emb

    def get_word_vector(self, word):
        if word not in self.word_embedding_dict.keys():
            embedding = np.random.uniform(low=-1, high=1, size=self.dim).astype(np.float32)
            self.word_embedding_dict[word] = embedding
            return embedding
```

```
        else:
            return self.word_embedding_dict[word]

logging.info("Start loading training data embeddings with ELMo")
elmo_embedding = ELMoEmbedding(train_data, options_file, weight_file, dim=EMBEDDING_DIM)

logging.info("Start gathering pretrained embeddings")

pretrained_embeddings = []
for i in range(VOCAB_SIZE):
    pretrained_embeddings.append(elmo_embedding.get_word_vector(INDEX_2_WORD[i]))

with open(os.path.join(TMP,  "elmo_pretrained.pkl"), 'wb') as handle:
    pickle.dump(pretrained_embeddings, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Load pickled embeddings without generating from scratch

In [6]:

```
with open(os.path.join(TMP_DIR,  "elmo_pretrained.pkl"), 'rb') as handle:
    pretrained_embeddings = pickle.load(handle)
```

Some helper functions for training

- When we retrieve the token indices, we append the `EOS` to let the model learn to predict the next word as `EOS` when it should stop
- We also pad a sequence with `PAD` when it doesn't meet max_length

In [7]:
```
# Return a list of indexes, one for each word in the sentence, plus EOS
def indexes_from_sentence(tokens,isHeadline):
    default_idx = WORD_2_INDEX[UNKNOWN_TOKEN]
    idxs = [WORD_2_INDEX.get(word, default_idx) for word in tokens]
    if isHeadline:
        idxs = idxs + [EOS_token]
    return idxs

# Pad a sentence with the PAD token
def pad_seq(seq, max_length):
    seq += [PAD_token for i in range(max_length - len(seq))]
    return seq
```

Adaptive Softmax

- explained in the paper

In [8]:
```
def masked_adasoft(logits, target, lengths, adasoft):
    loss = 0
    for i in range(logits.size(0)):
        mask = (np.array(lengths) > i).astype(int)

        mask = torch.LongTensor(np.nonzero(mask)[0]).to(device)
        logits_i = logits[i].index_select(0, mask)
        logits_i = logits_i.to(device)

        targets_i = target[i].index_select(0, mask).to(device)

        asm_output = adasoft(logits_i, targets_i)
        loss += asm_output.loss*len(targets_i)

    loss /= sum(lengths)

    return loss
```

Model Architecture

- seq2seq (GRU encoder, GRU decoder, Luong Attention)
- more explanations in the paper

In [9]:
```
def param_init(params):
    for name, param in params:
        if 'bias' in name:
             nn.init.constant_(param, 0.0)
        elif 'weight' in name:
            nn.init.xavier_normal_(param)

class EncoderRNN(nn.Module):
    """
    Scalars:
    input_size: vocabulary size
    hidden_size: the hidden dimension
    n_layers: number of hidden layers in GRU

    """
    def __init__(self, input_size, hidden_size, embed_size,pretrained_embeddings, n_layers, dropout):
        super(EncoderRNN, self).__init__()

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.dropout = dropout
        self.embed_size = embed_size

        self.embedding = nn.Embedding(input_size, embed_size).from_pretrained(torch.FloatTensor(pretr
ained_embeddings), freeze=True)

        self.gru = nn.GRU(embed_size, hidden_size, n_layers, dropout=self.dropout, bidirectional=True
)
        param_init(self.gru.named_parameters())

    def forward(self, input_seqs, input_lengths, hidden=None):
        embedded = self.embedding(input_seqs)
        packed = torch.nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)

        outputs, hidden = self.gru(packed, hidden)

        # unpack (back to padded)
        outputs, output_lengths = torch.nn.utils.rnn.pad_packed_sequence(outputs)
        return outputs, hidden
```

```
class Attn(nn.Module):
    def __init__(self, hidden_size):
        super(Attn, self).__init__()
        self.hidden_size = hidden_size

    def forward(self, hidden, encoder_outputs):
        attn_energies = torch.bmm(hidden.transpose(0,1), encoder_outputs.permute(1,2,0)).squeeze(1)
        return F.softmax(attn_energies, dim=1).unsqueeze(1)


class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, embed_size, pretrained_embeddings, n_layers=1, dropo
ut=0.1):
        super(DecoderRNN, self).__init__()

        # Keep for reference
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout
        self.embed_size = embed_size

        # Define layers
        self.embedding = nn.Embedding(output_size, hidden_size).from_pretrained(torch.FloatTensor(pre
trained_embeddings), freeze=True)
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(embed_size, hidden_size, n_layers, dropout=dropout)
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, FC_DIM)

        # Use Attention
        self.attn = Attn(hidden_size)
        param_init(self.gru.named_parameters())
        param_init(self.concat.named_parameters())
        param_init(self.out.named_parameters())

    def forward(self, input_seq, last_hidden, encoder_outputs):
        # Note: we run this one step at a time

        # Get the embedding of the current input word (last output word)
        batch_size = input_seq.size(0)
        embedded = self.embedding(input_seq)
        embedded = self.embedding_dropout(embedded)
```

```
        embedded = embedded.view(1, batch_size, self.embed_size) # S=1 x B x N

        # Get current hidden state from input word and last hidden state
        rnn_output, hidden = self.gru(embedded, last_hidden)

        # Calculate attention from current RNN state and all encoder outputs;
        # apply to encoder outputs to get weighted average
        attn_weights = self.attn(rnn_output, encoder_outputs)
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1)) # B x S=1 x N

        # Attentional vector using the RNN hidden state and context vector
        # concatenated together (Luong eq. 5)
        rnn_output = rnn_output.squeeze(0) # S=1 x B x N -> B x N
        context = context.squeeze(1)       # B x S=1 x N -> B x N
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))

        # Finally predict next token (Luong eq. 6, without softmax)
        output = self.out(concat_output)

        # Return final output, hidden state, and attention weights (for visualization)
        return output, hidden, attn_weights
```

Batching helper

In [10]:
```
def random_batch(batch_size, data):
    random.shuffle(data)
    end_index = len(data) - len(data) % batch_size
    input_seqs = []
    target_seqs = []

    # Choose random pairs
    for i in range(0, end_index, batch_size):
        pairs = data[i:i+batch_size]
        input_seqs = [indexes_from_sentence( pair[1], isHeadline=False) for pair in pairs]
        target_seqs = [indexes_from_sentence(pair[0], isHeadline=True) for pair in pairs]
        seq_pairs = sorted(zip(input_seqs, target_seqs), key=lambda p: len(p[0]), reverse=True)
        input_seqs, target_seqs = zip(*seq_pairs)

        input_lengths = [len(s) for s in input_seqs]
        input_padded = [pad_seq(s, max(input_lengths)) for s in input_seqs]

        target_lengths = [len(s) for s in target_seqs]
        target_padded = [pad_seq(s, max(target_lengths)) for s in target_seqs]

        input_var = Variable(torch.LongTensor(input_padded)).transpose(0, 1)
        target_var = Variable(torch.LongTensor(target_padded)).transpose(0, 1)

        input_var = input_var.to(device)
        target_var = target_var.to(device)
        yield input_var, input_lengths, target_var, target_lengths
```

Training subroutine for each batch

- Here we run each batch data through the encoder
- Encoder outputs (combined with previous step's decoder output) are ran through the decoder one step at a time until max_target_length is reached as teacher forcing
- Loss is computed for all decoder outputs against the target sequence
- Backpropagate, clip the gradient's norm to prevent gradient explosion
- Finally, weights are updated

In [11]:
```
def train_batch(input_batches, input_lengths, target_batches, target_lengths, batch_size, encoder, de
coder, encoder_optimizer, decoder_optimizer, clip):

    # Zero gradients of both optimizers
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()
    loss = 0 # Added onto for each word

    input_batches = input_batches.to(device)

    # Run words through encoder
    encoder_outputs, encoder_hidden = encoder(input_batches, input_lengths, None)

    # Prepare input and output variables
    decoder_input = Variable(torch.LongTensor([SOS_token] * batch_size)).to(device)
    decoder_hidden = torch.cat((encoder_hidden[0], encoder_hidden[1]),1)
    for i in range(1, encoder.n_layers):
        decoder_hidden = torch.stack((decoder_hidden,torch.cat((encoder_hidden[i*2],encoder_hidden[i*
2+1]),1)))
    decoder_hidden = decoder_hidden.to(device)

    max_target_length = max(target_lengths)
    all_decoder_outputs = Variable(torch.zeros(max_target_length, batch_size, FC_DIM)).to(device)

    # Run through decoder one time step at a time
    for t in range(max_target_length):
        decoder_output, decoder_hidden, decoder_attn = decoder(
            decoder_input, decoder_hidden, encoder_outputs
        )

        all_decoder_outputs[t] = decoder_output
        decoder_input = target_batches[t] # Next input is current target
    # Loss calculation and backpropagation
    loss = masked_adasoft(all_decoder_outputs, target_batches, target_lengths, crit)
    loss.backward()

    # Clip gradient norms
    ec = torch.nn.utils.clip_grad_norm_(encoder.parameters(), clip)
    dc = torch.nn.utils.clip_grad_norm_(decoder.parameters(), clip)

    # Update parameters with optimizers
    encoder_optimizer.step()
```

```
    decoder_optimizer.step()

    return loss.item(), ec, dc
```

Main train loop

- For each epoch, we go through the dataset once and train in batches
- We log the running loss every 25 batches
- We evaluate on a random pair every 100 batches
    - Run the text through the model, print the generated headline/summary, compare it with ground truth
- Every 1000 batches we update a checkpoint

In [12]:
```
def train(pairs, encoder, decoder, encoder_optimizer, decoder_optimizer, n_epochs, batch_size, clip):
    logging.info("Start training")

    for epoch in range(n_epochs):
        logging.info("Starting epoch: %d", epoch)
        running_loss = 0

        # Get training data for this epoch
        for batch_ind, batch_data in enumerate(random_batch(batch_size, pairs)):

            encoder_scheduler.step()
            decoder_scheduler.step()

            input_seqs, input_lengths, target_seqs, target_lengths = batch_data
            # Run the train subroutine
            loss, ec, dc = train_batch(
                input_seqs, input_lengths, target_seqs, target_lengths, batch_size,
                encoder, decoder,
                encoder_optimizer, decoder_optimizer, clip
            )
            # Keep track of loss
            running_loss += loss

            if batch_ind % 25 == 0:
                avg_running_loss = running_loss / 25
                running_loss = 0
                logging.info("Iteration: %d running loss: %f", batch_ind, avg_running_loss)

            if batch_ind % 100 == 0:
                logging.info("Iteration: %d, evaluating", batch_ind)
                evaluate_randomly(encoder, decoder, pairs)

            if batch_ind % 1000 == 0:
                logging.info("Iteration: %d model saved",batch_ind)
                save_checkpoint(encoder, decoder, encoder_optimizer, decoder_optimizer, name=CHECKPOI
NT_FNAME)
```

In [13]:
```
def save_checkpoint(encoder, decoder, encoder_optimizer, decoder_optimizer, name=CHECKPOINT_FNAME):
    path = os.path.join(MODEL_DIR, name)
    torch.save({'encoder_model_state_dict': encoder.state_dict(),
                'decoder_model_state_dict': decoder.state_dict(),
                'encoder_optimizer_state_dict':encoder_optimizer.state_dict(),
                'decoder_optimizer_state_dict':decoder_optimizer.state_dict(),
                'timestamp': str(datetime.datetime.now()),
                }, path)

def load_checkpoint(encoder, decoder, encoder_optimizer, decoder_optimizer, name=CHECKPOINT_FNAME):
    path = os.path.join(MODEL_DIR, name)
    if os.path.isfile(path):
        logging.info("Loading checkpoint")
        checkpoint = torch.load(path)
        encoder.load_state_dict(checkpoint['encoder_model_state_dict'])
        decoder.load_state_dict(checkpoint['decoder_model_state_dict'])
        encoder_optimizer.load_state_dict(checkpoint['encoder_optimizer_state_dict'])
        decoder_optimizer.load_state_dict(checkpoint['decoder_optimizer_state_dict'])
```

Evaluation

In [14]:
```
def evaluate(input_seq, encoder, decoder, max_length=MAX_OUTPUT_LENGTH):
    with torch.no_grad():
        input_seqs = [indexes_from_sentence( input_seq, isHeadline = False)]
        input_lengths = [len(input_seq) for input_seq in input_seqs]
        input_batches = Variable(torch.LongTensor(input_seqs)).transpose(0, 1).to(device)

        # Set to eval mode to disable dropout
        encoder.train(False)
        decoder.train(False)

        # Run through encoder
        encoder_outputs, encoder_hidden = encoder(input_batches, input_lengths, None)

        # Create starting vectors for decoder
        decoder_input = Variable(torch.LongTensor([SOS_token])).to(device) # SOS
        decoder_hidden = torch.cat((encoder_hidden[0], encoder_hidden[1]),1)
        for i in range(1, encoder.n_layers):
            decoder_hidden = torch.stack((decoder_hidden,torch.cat((encoder_hidden[i*2],encoder_hidde
n[i*2+1]),1)))
        decoder_hidden = decoder_hidden.to(device)

        # Store output words and attention states
        decoded_words = []
        decoder_attentions = torch.zeros(max_length + 1, max_length + 1).to(device)

        # Run through decoder
        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )

            # Choose top word from output
            ni = crit.predict(decoder_output)
            if ni == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(INDEX_2_WORD[int(ni)])

            # Next input is chosen word
            decoder_input = Variable(torch.LongTensor([ni]))
            decoder_input = decoder_input.to(device)
```

```
        # Set back to training mode
        encoder.train(True)
        decoder.train(True)

        return decoded_words
```

In [15]:
```
def evaluate_randomly(encoder, decoder, pairs):
    article = random.choice(pairs)
    headline = article[0]
    text = article[1]
    print('>', ' '.join(text))
    print('=', ' '.join(headline))

    output_words = evaluate(text, encoder, decoder)
    output_sentence = ' '.join(output_words)

    print('<', output_sentence)
```

Testing with Rouge

In [16]:
```
r = Rouge155()
r.system_dir = SYSTEM_DIR
r.model_dir = GOLD_DIR
r.system_filename_pattern = 'system.(\d+).txt'
r.model_filename_pattern = 'gold.[A-Z].#ID#.txt'
```

In [17]:
```
def write_headlines_to_file(template, directory, headlines):
    logging.info("Writing %d headlines to file", len(headlines))
    for i, line in enumerate(headlines):
        fpath = os.path.join(directory, template % i)
        with open(fpath, 'w+') as f:
            f.write(' '.join(line)+'\n')

def test_rouge(data, encoder, decoder):
    # some clean up
    shutil.rmtree(GOLD_DIR)
    os.mkdir(GOLD_DIR)
    shutil.rmtree(SYSTEM_DIR)
    os.mkdir(SYSTEM_DIR)

    filtered_data = []
    for headline, text in data:
        if len(headline) > MAX_HEADLINE_LENGTH:
            continue
        else:
            filtered_data.append((headline, text))

    logging.info("Start testing")

    original_len = len(filtered_data)
    filtered_data = [d for d in filtered_data if len(d[1])>0]
    logging.info("%d text have length equal 0", original_len - len(filtered_data))

    texts = [text for (_, text) in filtered_data]
    true_headlines = [headline for (headline,_) in filtered_data]
    write_headlines_to_file("gold.A.%d.txt", GOLD_DIR, true_headlines)

    pred_headlines = [evaluate(text, encoder, decoder) for text in texts]
    assert len(true_headlines) == len(pred_headlines)
    write_headlines_to_file("system.%d.txt", SYSTEM_DIR, pred_headlines)
    output = r.convert_and_evaluate()
    print(output)
```

Hyperparameters

- Choices explained in the paper

In [18]:
```
# Model architecture related
HIDDEN_SIZE = 200
N_LAYERS = 2
DROPOUT_PROB = 0.25
DECODER_LEARNING_RATIO = 5.0

# Training and optimization related
N_EPOCHS = 2
BATCH_SIZE = 32
GRAD_CLIP = 50.0
LR = 1e-3
WEIGHT_DECAY = 1e-4

# Adasoft related
CUTOFFS = [1000, 20000]
FC_DIM = 1024
```

Kick off training

In [19]:
```
# Init models
encoder = EncoderRNN(VOCAB_SIZE, HIDDEN_SIZE, EMBEDDING_DIM, pretrained_embeddings, N_LAYERS, dropout
=DROPOUT_PROB).to(device)
decoder = DecoderRNN(2*HIDDEN_SIZE, VOCAB_SIZE, EMBEDDING_DIM, pretrained_embeddings, N_LAYERS, dropo
ut=DROPOUT_PROB).to(device)

# Init optimizers
encoder_optimizer = torch.optim.Adam(encoder.parameters(), lr=LR, weight_decay=WEIGHT_DECAY)
decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=LR*DECODER_LEARNING_RATIO, weight_decay
=WEIGHT_DECAY)
encoder_scheduler = StepLR(encoder_optimizer, step_size=60000, gamma=0.25)
decoder_scheduler = StepLR(decoder_optimizer, step_size=60000, gamma=0.25)

# Load from checkpoint if has one
load_checkpoint(encoder, decoder, encoder_optimizer, decoder_optimizer, CHECKPOINT_FNAME)

# Init adasoft
crit = nn.AdaptiveLogSoftmaxWithLoss(FC_DIM, VOCAB_SIZE, CUTOFFS).to(device)

#train(train_data, encoder, decoder, encoder_optimizer, decoder_optimizer, N_EPOCHS, BATCH_SIZE, GRAD
_CLIP)
```

```
23:30:53 Loading checkpoint
```

Evaluate with Rouge metric on dev data

In [20]:
```
test_rouge(dev_data, encoder, decoder)
```

Evaluate with Rouge metric on test data

In [ ]:
```
test_rouge(test_data, encoder, decoder)
```

Code References

- ELMo embeddings: https://github.com/allenai/allennlp/blob/master/tutorials/how_to/elmo.md (https://github.com/allenai/allennlp/blob/master/tutorials/how_to/elmo.md)
- Seq2Seq tutorial: https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation-batched.ipynb (https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation-batched.ipynb)