drawing

50.040 Natural Language Processing, Summer 2020

Due 19 June 2020, 5pm

Mini Project

Write your student ID and name

STUDENT ID: 1003056

Name: Ivan Christian

Students with whom you have discussed (if any): Ng Jen Yang, Tee Zhi Yao, Eda Tan

Introduction

Language models are very useful for a wide range of applications, e.g., speech recognition and machine translation. Consider a sentence consisting of words $x_1, x_2, …, x_m$, where m is the length of the sentence, the goal of language modeling is to model the probability of the sentence, where $m \geq 1$, $x_i \in V $ and V is the vocabulary of the corpus: $$p(x_1, x_2, …, x_m)$$ In this project, we are going to explore both statistical language model and neural language model on the [Wikitext-2](#) datasets. Download wikitext-2 word-level data and put it under the `data` folder.

Statistical Language Model

A simple way is to view words as independent random variables (i.e., zero-th order Markovian assumption). The joint probability can be written as: $$p(x_1, x_2, …, x_m)=\prod_{i=1}^m p(x_i)$$ However, this model ignores the word order information, to account for which, under the first-order Markovian assumption, the joint probability can be written as: $$p(x_0, x_1, x_2, …, x_m)= \prod_{i=1}^{m}p(x_i \mid x_{i-1})$$ Under the second-order Markovian assumption, the joint probability can be written as: $$p(x_{-1}, x_0, x_1, x_2, …, x_m)= \prod_{i=1}^{m}p(x_i \mid x_{i-2}, x_{i-1})$$ Similar to what we did in HMM, we will assume that $x_{-1}=START, x_0=START, x_m = STOP$ in this definition, where $START, STOP$ are special symbols referring to the start and the end of a sentence.

Parameter estimation

Let's use $count(u)$ to denote the number of times the unigram u appears in the corpus, use $count(v, u)$ to denote the number of times the bigram v, u appears in the corpus, and $count(w, v, u)$ the times the trigram w, v, u appears in the corpus, $u \in V \cup STOP$ and $w, v \in V \cup START$.

And the parameters of the unigram, bigram and trigram models can be obtained using maximum likelihood estimation (MLE).

- In the unigram model, the parameters can be estimated as: $$p(u) = \frac {count(u)}{c}$$, where c is the total number of words in the corpus.
- In the bigram model, the parameters can be estimated as: $$p(u \mid v) = \frac{count(v, u)}{count(v)}$$
- In the trigram model, the parameters can be estimated as: $$p(u \mid w, v) = \frac{count(w, v, u)}{count(w, v)}$$

In [1]:

```
%%javascript
MathJax.Hub.Config({
  TeX: { equationNumbers: { autoNumber: "AMS" } }
});
```

Smoothing the parameters

Note, it is likely that many parameters of bigram and trigram models will be 0 because the relevant bigrams and trigrams involved do not appear in the corpus. If you don't have a way to handle these 0 probabilities, all the sentences that include such bigrams or trigrams will have probabilities of 0.

We'll use a Add-k Smoothing method to fix this problem, the smoothed parameter can be estimated as: \begin{equation} p_{add-k}

We'll use a Add-k Smoothing method to fix this problem, the smoothed parameter can be estimated as: \begin{equation} p_{add-k}(u)= \frac{count(u)+k}{c+k|V^*|} \end{equation} \begin{equation} p_{add-k}(u \mid v)= \frac{count(v, u)+k}{count(v)+k|V^*|} \end{equation} \begin{equation} p_{add-k}(u \mid w, v)= \frac{count(w, v, u)+k}{count(w, v)+k|V^*|} \end{equation}

where $k \in (0, 1)$ is the parameter of this approach, and $|V^*|$ is the size of the vocabulary V^*,here $V^*= V \cup STOP$. One way to choose the value of k is by optimizing the perplexity of the development set, namely to choose the value that minimizes the perplexity.

Perplexity

Given a test set $D^{\prime}$ consisting of sentences $X^{(1)}, X^{(2)}, …, X^{(|D^{\prime}|)}$, each sentence $X^{(j)}$ consists of words $x_1^{(j)}, x_2^{(j)},…,x_{n_j}^{(j)}$, we can measure the probability of each sentence s_i, and the quality of the language model would be the probability it assigns to the entire set of test sentences, namely: \begin{equation} \prod_j^{D^{\prime}}p(X^{(j)}) \end{equation} Let's define average log2 probability as: \begin{equation} l=\frac{1}{c^{\prime}}\sum_{j=1}^{|D^{\prime}|}log_2p(X^{(j)}) \end{equation} $c^{\prime}$ is the total number of words in the test set, $D^{\prime}$ is the number of sentences. And the perplexity is defined as: \begin{equation} perplexity=2^{-l} \end{equation}

The lower the perplexity, the better the language model.

In [2]:

```
from collections import Counter, namedtuple
import itertools
import numpy as np
```

In [3]:

```
with open('data/wikitext-2/wiki.train.tokens', 'r', encoding='utf8') as f:
    text = f.readlines()
    train_sents = [line.lower().strip('\n').split() for line in text]
    train_sents = [s for s in train_sents if len(s)>0 and s[0] != '=']
```

In [4]:

```
print(train_sents[1])
```

```
['the', 'game', 'began', 'development', 'in', '2010', ',', 'carrying', 'over', 'a', 'large', 'port
ion', 'of', 'the', 'work', 'done', 'on', 'valkyria', 'chronicles', 'ii', '.', 'while', 'it',
'retained', 'the', 'standard', 'features', 'of', 'the', 'series', ',', 'it', 'also', 'underwent',
'multiple', 'adjustments', ',', 'such', 'as', 'making', 'the', 'game', 'more', '<unk>', 'for', 'se
ries', 'newcomers', '.', 'character', 'designer', '<unk>', 'honjou', 'and', 'composer', 'hitoshi',
'sakimoto', 'both', 'returned', 'from', 'previous', 'entries', ',', 'along', 'with', 'valkyria', '
chronicles', 'ii', 'director', 'takeshi', 'ozawa', '.', 'a', 'large', 'team', 'of', 'writers', 'ha
ndled', 'the', 'script', '.', 'the', 'game', "'s", 'opening', 'theme', 'was', 'sung', 'by', 'may',
"'n", '.']
```

Question 1 [code][written]

1. Implement the function **"compute_ngram"** that computes n-grams in the corpus. (Do not take the START and STOP symbols into consideration for now.) For n=1,2,3, the number of unique n-grams should be **28910/577343/1344047**, respectively.
2. List 10 most frequent unigrams, bigrams and trigrams as well as their counts.(Hint: use the built-in function .most_common in Counter class)

In [5]:

```
def compute_ngram(sents, n):
    '''
    Compute n-grams that appear in "sents".
    param:
        sents: list[list[str]] --- list of list of word strings
        n: int --- "n" gram
    return:
        ngram_set: set{str} --- a set of n-grams (no duplicate elements)
        ngram_dict: dict{ngram: counts} --- a dictionary that maps each ngram to its number
occurence in "sents";
        This dict contains the parameters of our ngram model. E.g. if n=2, ngram_dict=
{('a','b'):10, ('b','c'):13}
```

```
        You may need to use "Counter", "tuple" function here.
    '''
    ngram_set = None
    ngram_dict = None
    ### YOUR CODE HERE
    '''
    sents contain list of sentences
    '''

    ngram_dict = dict()
    for sentence in sents:
        for index in range(len(sentence) - n + 1):
            n_gram_word = tuple(sentence[index:index+n])
            ngram_dict.setdefault(n_gram_word, 0)
            ngram_dict[n_gram_word] += 1
    ngram_set = set(ngram_dict)
    ### END OF YOUR CODE
    return ngram_set, ngram_dict
```

In [6]:

```
### ~28xxx
unigram_set, unigram_dict = compute_ngram(train_sents, 1)
print(len(unigram_set))
```

```
28910
```

In [7]:

```
### ~57xxxx
bigram_set, bigram_dict = compute_ngram(train_sents, 2)
print(len(bigram_set))
```

```
577343
```

In [8]:

```
### ~134xxxx
trigram_set, trigram_dict = compute_ngram(train_sents, 3)
print(len(trigram_set))
```

```
1344047
```

In [9]:

```
# List 10 most frequent unigrams, bigrams and trigrams as well as their counts.
def get_top10(ngram_dict, n ):
    '''
    Function to get the top n n-gram slices in the dictionary
    Args:
    - ngram_dict : dict(dictionary containing the n-gram slices)
    - n : int (number of elements you want to see)
    Returns:
    - sorted_dict : dict (dictionary containing the top n elements)
    '''
    sorted_dict = {k: v for k, v in sorted(ngram_dict.items(), key = lambda x : x[1], reverse=True)[
:n]}
    return sorted_dict
```

In [10]:

```
get_top10(unigram_dict, 10)
```

Out[10]:

```
{('the',): 130519,
 (',',): 99763,
 ('.',): 73388,
 ('of',): 56743,
 ('<unk>',): 53951,
```

```
 ('and',): 49940,
 ('in',): 44876,
 ('to',): 39462,
 ('a',): 36140,
 ('"',): 28285}
```

Question 2 [code][written]

In this part, we take the START and STOP symbols into consideration. So we need to pad the **train_sents** as described in "Statistical Language Model" before we apply "compute_ngram" function. For example, given a sentence "I like NLP", in a bigram model, we need to pad it as "START I like NLP STOP", in a trigram model, we need to pad it as "START START I like NLP STOP".

1. Implement the `pad_sents function`.
2. Pad `train_sents`.
3. Apply `compute_ngram` function to these padded sents.
4. Implement `ngram_prob` function. Compute the probability for each n-gram in the variable **ngrams** according to Eq.(1)(2)(3) in **"smoothing the parameters"** .List down the n-grams that have 0 probability.

In [11]:

```
##################################################
ngrams = list()
with open(r'data/ngram.txt','r') as f:
    for line in f:
        ngrams.append(line.strip('\n').split())
print(ngrams)
##################################################
```

```
[['the', 'computer'], ['go', 'to'], ['have', 'had'], ['and', 'the'], ['can', 'sea'], ['a',
'number', 'of'], ['with', 'respect', 'to'], ['in', 'terms', 'of'], ['not', 'good', 'bad'], ['first
', 'start', 'with']]
```

In [12]:

```
START = '<START>'
STOP = '<STOP>'
####################################
def pad_sents(sents, n):
    '''
    Pad the sents according to n.
    params:
        sents: list[list[str]] --- list of sentences.
        n: int --- specify the padding type, 1-gram, 2-gram, or 3-gram.
    return:
        padded_sents: list[list[str]] --- list of padded sentences.
    '''
    padded_sents = None
    ### YOUR CODE HERE
    padded_sents = []

    start = '<START>'
    stop = '<STOP>'

    start_ls, stop_ls = [], []

    for i in range(n):
        start_ls.append(start)
        stop_ls.append(stop)

    for sent in sents:
        padded_sent = start_ls + sent + stop_ls
        padded_sents.append(padded_sent)

    ### END OF YOUR CODE
    return padded_sents
```

In [13]:

```
uni_sents = pad_sents(train_sents, 1)
bi_sents = pad_sents(train_sents, 2)
```

```
tri_sents = pad_sents(train_sents, 3)
```

In [14]:

```
unigram_set, unigram_dict = compute_ngram(uni_sents, 1)
bigram_set, bigram_dict = compute_ngram(bi_sents, 2)
trigram_set, trigram_dict = compute_ngram(tri_sents, 3)
```

In [15]:

```
### (28xxx, 58xxxx, 136xxxx)
len(unigram_set),len(bigram_set),len(trigram_set)
```

Out[15]:

```
(28912, 580827, 1363978)
```

In [16]:

```
### ~ 200xxxx; total number of words in wikitext-2.train
num_words = sum([v for _,v in unigram_dict.items()])
print(num_words)
```

```
2042258
```

In [17]:

```
def ngram_prob(ngram, num_words, unigram_dic, bigram_dic, trigram_dic):
    '''
    params:
        ngram: list[str] --- a list that represents n-gram
        num_words: int --- total number of words
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram to its number of o
ccurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram to its number of oc
curence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram to its number
occurence in "sents";
    return:
        prob: float --- probability of the "ngram"
    '''
    prob = None
    ### YOUR CODE HERE
    '''
    unigram : count(u)

    '''
    uni_prob = 0
    bi_prob = 0
    tri_prob = 0

    try:

        if len(ngram) == 1:
            for word in ngram:
#                 if (word,) not in unigram_dic:
#                     numerator = unigram_dic[('<unk>',)]
#                 else:
                numerator = unigram_dic[(word,)]
                unigram_prob = numerator / num_words
                uni_prob *= unigram_prob
            prob = uni_prob
        elif len(ngram) == 2:
            bigram_denominator = unigram_dic[(ngram[0],)]
            bigram_numerator = bigram_dic[tuple(ngram)]
            prob = bigram_numerator/bigram_denominator
        elif len(ngram) == 3:
            trigram_denominator = bigram_dic[(ngram[0],ngram[1])]
            trigram_numerator = trigram_dic[tuple(ngram)]
            prob = trigram_numerator/trigram_denominator
    except:
        prob = 0
```

```
    ### END OF YOUR CODE
    return prob
```

In [18]:

```
### ~9.96e-05
ngram_prob(ngrams[0], num_words,unigram_dict, bigram_dict, trigram_dict)
```

Out[18]:

```
9.960235674499498e-05
```

In [19]:

```
### List down the n-grams that have 0 probability.
for ngram in ngrams:
    p = ngram_prob(ngram, num_words,unigram_dict, bigram_dict, trigram_dict)
    if p == 0:
        print(ngram, p)
```

```
['can', 'sea'] 0
['not', 'good', 'bad'] 0
['first', 'start', 'with'] 0
```

Question 3 [code][written]

1. Implement `smooth_ngram_prob` function to estimate ngram probability with `add-k` smoothing technique. Compute the smoothed probabilities of each n-gram in the variable **"ngrams"** according to Eq.(1)(2)(3) in **"smoothing the parameters"** section.
2. Implement `perplexity` function to compute the perplexity of the corpus "**valid_sents**" according to the Equations (4),(5),(6) in **perplexity** section. The computation of $p(X^{(j)})$ depends on the n-gram model you choose. If you choose 2-gram model, then you need to calculate $p(X^{(j)})$ based on Eq.(2) in **smoothing the parameter** section. Hint: convert probability to log probability.
3. Try out different $k\in [0.1, 0.3, 0.5, 0.7, 0.9]$ and different n-gram model ($n=1,2,3$). Find the n-gram model and k that gives the best perplexity on "**valid_sents**" (smaller is better).

In [20]:

```
with open('data/wikitext-2/wiki.valid.tokens', 'r', encoding='utf8') as f:
    text = f.readlines()
    valid_sents = [line.lower().strip('\n').split() for line in text]
    valid_sents = [s for s in valid_sents if len(s)>0 and s[0] != '=']

uni_valid_sents = pad_sents(valid_sents, 1)
bi_valid_sents = pad_sents(valid_sents, 2)
tri_valid_sents = pad_sents(valid_sents, 3)
```

In [21]:

```
def smooth_ngram_prob(ngram, k, num_words, unigram_dic, bigram_dic, trigram_dic):
    '''
    params:
        ngram: list[str] --- a list that represents n-gram
        k: float
        num_words: int --- total number of words
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram to its number of o
ccurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram to its number of oc
curence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram to its number
occurence in "sents";
    return:
        s_prob: float --- probability of the "ngram"
    '''
    s_prob = 0
    V = len(unigram_dic) + 1
```

```
    ### YOUR CODE HERE


    if len(ngram) == 1:
        for word in ngram:
#             if (word,) not in unigram_dic:
#                 numerator = unigram_dic[('<unk>',)]
#             else:
            numerator = unigram_dic[(word,)]
            s_prob = (numerator + k) / (num_words + V)
    elif len(ngram) == 2:

        bigram_denominator = unigram_dic[(ngram[0],)]
        try:
            bigram_numerator = bigram_dic[tuple(ngram)]
        except:
            bigram_numerator = 0
        s_prob = (bigram_numerator + k) / (bigram_denominator + k*V)
    elif len(ngram) == 3:

        try:
            trigram_denominator = bigram_dic[(ngram[0],ngram[1])]
            trigram_numerator = trigram_dic[tuple(ngram)]
        except:
            trigram_denominator = 0
            trigram_numerator = 0
        s_prob = (trigram_numerator + k) / (trigram_denominator + k * V)

    ### END OF YOUR CODE
    return s_prob
```

In [22]:

```
### ~ 9.31e-05
smooth_ngram_prob(ngrams[0], 0.5, num_words, unigram_dict, bigram_dict, trigram_dict)
```

Out[22]:

```
9.311918220664871e-05
```

In [23]:

```
def perplexity(n, k, num_words, valid_sents, unigram_dic, bigram_dic, trigram_dic):
    '''
    compute the perplexity of valid_sents
    params:
        n: int --- n-gram model you choose.
        k: float --- smoothing parameter.
        num_words: int --- total number of words in the traning set.
        valid_sents: list[list[str]] --- list of sentences.
        unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram to its number of o
ccurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram to its number of oc
curence in "sents";
        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram to its number
occurence in "sents";
    return:
        ppl: float --- perplexity of valid_sents
    '''
    ppl = None
    ### YOUR CODE HERE
    c = 0
    l = 0
    for sentence in valid_sents:
        c += len(sentence)
        s_p = 0
        for i in range(len(sentence)):
            s_p += np.log2(smooth_ngram_prob(sentence[i:i+n], k, num_words, unigram_dic, bigram_dic
, trigram_dic))
        l+= s_p
    l /= c
    ppl = 2 **(-l)

    ### END OF YOUR CODE
    return ppl
```

In [24]:

```
### ~ 840
perplexity(1, 0.1, num_words, uni_valid_sents, unigram_dict, bigram_dict, trigram_dict)
```

Out[24]:

```
836.9592060281042
```

In [25]:

```
n = [1,2,3]
k = [0.1, 0.3, 0.5, 0.7, 0.9]
### YOUR CODE HERE

for i in n:
    for j in k:
        print(f'n = {i}, k = {j}, ppl = {perplexity(i, j, num_words, uni_valid_sents, unigram_dict,
bigram_dict, trigram_dict)}')

### END OF YOUR CODE
```

```
n = 1, k = 0.1, ppl = 836.9592060281042
n = 1, k = 0.3, ppl = 834.9977532200639
n = 1, k = 0.5, ppl = 833.0914520584979
n = 1, k = 0.7, ppl = 831.2359545332025
n = 1, k = 0.9, ppl = 829.4274815419767
n = 2, k = 0.1, ppl = 783.8781349685943
n = 2, k = 0.3, ppl = 1116.7472231327645
n = 2, k = 0.5, ppl = 1352.731252461768
n = 2, k = 0.7, ppl = 1547.7695868017806
n = 2, k = 0.9, ppl = 1718.4083087461815
n = 3, k = 0.1, ppl = 5128.537607667474
n = 3, k = 0.3, ppl = 7272.002814119486
n = 3, k = 0.5, ppl = 8557.354797026534
n = 3, k = 0.7, ppl = 9503.041224722776
n = 3, k = 0.9, ppl = 10256.287856035962
```

Question 4 [code]

Evaluate the perplexity of the test data **test_sents** based on the best n-gram model and k you have found on the validation data (Q 3.3).

In [26]:

```
with open('data/wikitext-2/wiki.test.tokens', 'r', encoding='utf8') as f:
    text = f.readlines()
    test_sents = [line.lower().strip('\n').split() for line in text]
    test_sents = [s for s in test_sents if len(s)>0 and s[0] != '=']

uni_test_sents = pad_sents(test_sents, 1)
bi_test_sents = pad_sents(test_sents, 2)
tri_test_sents = pad_sents(test_sents, 3)
```

In [27]:

```
### YOUR CODE HERE
perplexity(2, 0.1, num_words,bi_test_sents, unigram_dict, bigram_dict, trigram_dict)
### END OF YOUR CODE
```

Out[27]:

```
653.2327035169877
```

Neural Language Model (RNN)

drawing

drawing

We will create a LSTM language model as shown in figure and train it on the Wikitext-2 dataset. The data generators (train_iter, valid_iter, test_iter) have been provided. The word embeddings together with the parameters in the LSTM model will be learned from scratch.

Pytorch and torchtext are required in this part. Do not make any changes to the provided code unless you are requested to do so.

Question 5 [code]

- Implement the `__init__` function in `LangModel` class.
- Implement the `forward` function in `LangModel` class.
- Complete the training code in `train` function. Then complete the testing code in `test` function and compute the perplexity of the test data `test_iter`. The test perplexity should be below 150.

In [28]:

```
import torchtext
import torch
import torch.nn.functional as F
from torchtext.datasets import WikiText2
from torch import nn, optim
from torchtext import data
from nltk import word_tokenize
import nltk
nltk.download('punkt')
torch.manual_seed(222)
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\chris\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[28]:

```
<torch._C.Generator at 0x17cd71e9c30>
```

In [29]:

```
def tokenizer(text):
    '''Tokenize a string to words'''
    return word_tokenize(text)

START = '<START>'
STOP = '<STOP>'
#Load and split data into three parts
TEXT = data.Field(lower=True, tokenize=tokenizer, init_token=START, eos_token=STOP)
train, valid, test = WikiText2.splits(TEXT)
```

In [30]:

```
#Build a vocabulary from the train dataset
TEXT.build_vocab(train)
print('Vocabulary size:', len(TEXT.vocab))
```

```
Vocabulary size: 28905
```

In [31]:

```
BATCH_SIZE = 64
# the length of a piece of text feeding to the RNN layer
BPTT_LEN = 32
# train, validation, test data
train_iter, valid_iter, test_iter = data.BPTTIterator.splits((train, valid, test),
                                                                batch_size=BATCH_SIZE,
                                                                bptt_len=BPTT_LEN,
                                                                repeat=False)
```

In [32]:

```
#Generate a batch of train data
batch = next(iter(train_iter))
text, target = batch.text, batch.target
# print(batch.dataset[0].text[:32])
# print(text[0:3],target[:3])
print('Size of text tensor',text.size())
print('Size of target tensor',target.size())
```

```
Size of text tensor torch.Size([32, 64])
Size of target tensor torch.Size([32, 64])
```

In [33]:

```
class LangModel(nn.Module):
    def __init__(self, lang_config):
        super(LangModel, self).__init__()
        self.vocab_size = lang_config['vocab_size']
        self.emb_size = lang_config['emb_size']
        self.hidden_size = lang_config['hidden_size']
        self.num_layer = lang_config['num_layer']

        self.embedding = None
        self.rnn = None
        self.linear = None

        ### TODO:
        ###    1. Initialize 'self.embedding' with nn.Embedding function and 2 variables we have
initialized for you
        ###    2. Initialize 'self.rnn' with nn.LSTM function and 3 variables we have initialized
for you
        ###    3. Initialize 'self.linear' with nn.Linear function and 2 variables we have
initialized for you
        ### Reference:
        ###        https://pytorch.org/docs/stable/nn.html

        ### YOUR CODE HERE (3 lines)



        self.embedding = nn.Embedding(self.vocab_size, self.emb_size)
        self.rnn = nn.LSTM(self.emb_size, self.hidden_size, self.num_layer)
        self.linear = nn.Linear(self.hidden_size, self.vocab_size)
        ### END OF YOUR CODE

    def forward(self, batch_sents, hidden=None):
        '''
        params:
            batch_sents: torch.LongTensor of shape (sequence_len, batch_size)
        return:
            normalized_score: torch.FloatTensor of shape (sequence_len, batch_size, vocab_size)
        '''
        normalized_score = None
        hidden = hidden
        ### TODO:
        ###      1. Feed the batch_sents to self.embedding
        ###      2. Feed the embeddings to self.rnn. Remember to pass "hidden" into self.rnn, even
if it is None. But we will
        ###         use "hidden" when implementing greedy search.
        ###      3. Apply linear transformation to the output of self.rnn
        ###      4. Apply 'F.log_softmax' to the output of linear transformation
        ###
        ### YOUR CODE HERE
        embedded = self.embedding(batch_sents)
        out, hidden = self.rnn(embedded, (hidden))
        out = self.linear(out)
        normalized_score = F.log_softmax(out,dim=2)
        ### END OF YOUR CODE
        return normalized_score, hidden
```

In [34]:

```
def train(model, train_iter, valid_iter, vocab_size, criterion, optimizer, num_epochs):
    for n in range(num_epochs):
        train_loss = 0
```

```
        target_num = 0
        model.train()
        for batch in train_iter:

            text, targets = batch.text.to(device), batch.target.to(device)
            loss = None

            ### we don't consider "hidden" here. So according to the default setting, "hidden"
will be None
            ### YOU CODE HERE (~5 lines)
            model.zero_grad()
            output, _ = model(text)

            loss = criterion(output.view(-1, vocab_size), targets.view(-1))
            loss.backward()
            optimizer.step()

            ### END OF YOUR CODE
            ##########################################
            train_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)

        train_loss /= target_num

        # monitor the loss of all the predictions
        val_loss = 0
        target_num = 0
        model.eval()
        for batch in valid_iter:
            text, targets = batch.text.to(device), batch.target.to(device)

            prediction,_ = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))

            val_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)
        val_loss /= target_num

        print('Epoch: {}, Training Loss: {:.4f}, Validation Loss: {:.4f}'.format(n+1, train_loss, v
al_loss))
```

In [35]:

```
def test(model, vocab_size, criterion, test_iter):
    '''
    params:
        model: LSTM model
        test_iter: test data
    return:
        ppl: perplexity
    '''
    ppl = None
    test_loss = 0
    target_num = 0
    with torch.no_grad():
        for batch in test_iter:
            text, targets = batch.text.to(device), batch.target.to(device)

            prediction,_ = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))

            test_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)

        test_loss /= target_num

        ### Compute perplexity according to "test_loss"
        ### Hint: Consider how the loss is computed.
        ### YOUR CODE HERE(1 line)
        ppl = 2**(test_loss) # NLLoss
        ### END OF YOUR CODE
        return ppl
```

In [36]:

```
num_epochs=10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
vocab_size = len(TEXT.vocab)

config = {'vocab_size':vocab_size,
          'emb_size':128,
          'hidden_size':128,
          'num_layer':1}

LM = LangModel(config)
LM = LM.to(device)

criterion = nn.NLLLoss(reduction='mean').to(device)
optimizer = optim.Adam(LM.parameters(), lr=1e-3, betas=(0.7, 0.99))
```

In [37]:

```
train(LM, train_iter, valid_iter,vocab_size, criterion, optimizer, num_epochs)
```

```
Epoch: 1, Training Loss: 6.0684, Validation Loss: 5.1855
Epoch: 2, Training Loss: 5.4048, Validation Loss: 4.9640
Epoch: 3, Training Loss: 5.1295, Validation Loss: 4.8605
Epoch: 4, Training Loss: 4.9541, Validation Loss: 4.8123
Epoch: 5, Training Loss: 4.8266, Validation Loss: 4.7835
Epoch: 6, Training Loss: 4.7266, Validation Loss: 4.7667
Epoch: 7, Training Loss: 4.6443, Validation Loss: 4.7576
Epoch: 8, Training Loss: 4.5740, Validation Loss: 4.7538
Epoch: 9, Training Loss: 4.5130, Validation Loss: 4.7533
Epoch: 10, Training Loss: 4.4593, Validation Loss: 4.7558
```

In [38]:

```
# < 150
test(LM, vocab_size, criterion, test_iter)
```

Out[38]:

```
24.313683028512312
```

Question 6 [code]

When we use trained language model to generate a sentence given a start token, we can choose either `greedy search` or `beam search`.

drawing

As shown above, `greedy search` algorithm will pick the token which has the highest probability and feed it to the language model as input in the next time step. The model will generate `max_len` number of tokens at most.

- Implement `word_greedy_search`
- **[optional]** Implement `word_beam_search`

In [39]:

```
def word_greedy_search(model, start_token, max_len):
    '''
    param:
        model: nn.Module --- language model
        start_token: str --- e.g. 'he'
        max_len: int --- max number of tokens generated
    return:
        strings: list[str] --- list of tokens, e.g., ['he', 'was', 'a', 'member', 'of',...]
    '''
    model.eval()
    ID = TEXT.vocab.stoi[start_token]
    strings = [start_token]
    hidden = None

    ### You may find TEXT.vocab.itos useful.
    ### YOUR CODE HERE
```

```python
    for i in range(max_len):
        out , _ = model(torch.LongTensor([[ID]]).to(device)) # out would be a id
        ID = torch.argmax(out, dim =-1)
        strings.append(TEXT.vocab.itos[ID])

    ### END OF YOUR CODE
    return strings
```

In [40]:

```python
word_greedy_search(LM, 'he', 64)
```

Out[40]:

```
['he',
 'was',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
 'and',
 'the',
 '<',
 'unk',
 '>',
 ',',
```

```
 'and',
 'the',
 '<',
 'unk']
```

In [41]:

```python
# BeamNode = namedtuple('BeamNode', ['prev_node', 'prev_hidden', 'wordID', 'score', 'length'])
# LMNode = namedtuple('LMNode', ['sent', 'score'])

def word_beam_search(model, start_token, max_len, beam_size):
    model.eval()
    ID = TEXT.vocab.stoi[start_token]
    strings = [start_token]
    hidden = None

    from collections import defaultdict
    d = defaultdict(list)
    scores= defaultdict(float)

    for i in range(1,beam_size+1):
        d[i]= [TEXT.vocab.stoi[start_token]]
        scores[i] = 0
    out , _ = model(torch.LongTensor([[ID]]).to(device)) # out would be a id

    top_val, topID = torch.topk(out, dim =-1,k=beam_size,sorted=True)
    top_val = top_val.squeeze(1).reshape(-1)
    topID = topID.squeeze(1).reshape(-1)

    for i in d:
        d[i].append(int(topID[i-1]))
        scores[i] += float(top_val[i-1])


    for i in range(1,max_len):
        temp_list = []
        for j in d:
            out, _ =  model(torch.LongTensor([[d[j][i]]]).to(device))
            top_val, topID = torch.topk(out, dim =-1,k=beam_size,sorted=True)
            top_val = top_val.squeeze(1).reshape(-1)
            topID = topID.squeeze(1).reshape(-1)
            for val, ids in list(zip(top_val, topID)):
                temp_list.append((d[j],scores[j]+ float(val),[int(ids)]))
        temp_list.sort(key=lambda x:x[1], reverse = True )
        temp_list = temp_list[:3]


        for k in d:
            temp_val = temp_list[k-1][0] + temp_list[k-1][2]
            d[k] = temp_val
            scores[k] = temp_list[k-1][1]
#         print(f'Current index: {i}, current top-k : {d}, current top k score : {scores}')

    strings = [TEXT.vocab.itos[code] for code in d[1]]
    return strings
```

In [42]:

```python
word_beam_search(LM, 'he', 64, 3)
```

Out[42]:

```
['he',
 'was',
 'the',
 '<',
 'unk',
 '>',
 '.',
 '<eos>',
 '=',
 '=',
 '=',
 '=',
 '=',
```

```
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=',
 '=']
```

char-level LM

Question 7 [code]

- Implement `char_tokenizer`
- Implement `CharLangModel`, `char_train`, `char_test`
- Implement `char_greedy_search`

In [43]:

```python
def char_tokenizer(string):
    '''
    param:
        string: str --- e.g. "I love this assignment"
    return:
        char_list: list[str] --- e.g. ['I', 'l', 'o', 'v', 'e', ' ', 't', 'h', 'i', 's', ...]
    '''
    char_list = None
```

```
    ### YOUR CODE HERE
    char_list = []
    for char in string:
        char_list.append(char)

    ### END OF YOUR CODE
    return char_list
```

In [44]:

```
test_str = 'test test test'
char_tokenizer(test_str)
```

Out[44]:

```
['t', 'e', 's', 't', ' ', 't', 'e', 's', 't', ' ', 't', 'e', 's', 't']
```

In [45]:

```
CHAR_TEXT = data.Field(lower=True, tokenize=char_tokenizer ,init_token='<START>',
eos_token='<STOP>')
ctrain, cvalid, ctest = WikiText2.splits(CHAR_TEXT)
```

In [46]:

```
CHAR_TEXT.build_vocab(ctrain)
print('Vocabulary size:', len(CHAR_TEXT.vocab))
```

```
Vocabulary size: 247
```

In [47]:

```
BATCH_SIZE = 32
# the length of a piece of text feeding to the RNN layer
BPTT_LEN = 128
# train, validation, test data
ctrain_iter, cvalid_iter, ctest_iter = data.BPTTIterator.splits((ctrain, cvalid, ctest),
                                                                batch_size=BATCH_SIZE,
                                                                bptt_len=BPTT_LEN,
                                                                repeat=False)
```

In [48]:

```
class CharLangModel(nn.Module):
    def __init__(self, lang_config):
        ### YOUR CODE HERE
        super(CharLangModel,self).__init__()
        self.vocab_size = lang_config['vocab_size']
        self.emb_size = lang_config['emb_size']
        self.hidden_size = lang_config['hidden_size']
        self.num_layer = lang_config['num_layer']


        self.embedding = nn.Embedding( num_embeddings = self.vocab_size, embedding_dim = self.emb_s
ize)

        self.rnn = nn.LSTM(self.emb_size, self.hidden_size, self.num_layer)

        self.linear = nn.Linear(in_features=self.hidden_size, out_features=self.vocab_size)



    def forward(self, batch_sents, hidden=None):
        ### YOUR CODE HERE
        hidden = hidden
        embedded = self.embedding(batch_sents)
        out, hidden = self.rnn(embedded, (hidden))
        out = self.linear(out)
        normalized_score = F.log_softmax(out,dim=2)

        return normalized_score
```

In [49]:

```
def char_train(model, train_iter, valid_iter, criterion, optimizer, vocab_size, num_epochs):
    ### YOUR CODE HERE
     for n in range(num_epochs):
        train_loss = 0
        target_num = 0
        hidden = None
        model.train()
        for batch in train_iter:

            text, targets = batch.text.to(device), batch.target.to(device)
            loss = None

            ### we don't consider "hidden" here. So according to the default setting, "hidden" 
will be None
            ### YOU CODE HERE (~5 lines)
            model.zero_grad()
            output = model(text)

            loss = criterion(output.view(-1, vocab_size), targets.view(-1))
            loss.backward()
            optimizer.step()

            ### END OF YOUR CODE
            ##########################################
            train_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)

        train_loss /= target_num

        # monitor the loss of all the predictions
        val_loss = 0
        target_num = 0
        model.eval()
        for batch in valid_iter:
            text, targets = batch.text.to(device), batch.target.to(device)

            prediction = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))

            val_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)
        val_loss /= target_num

        print('Epoch: {}, Training Loss: {:.4f}, Validation Loss: {:.4f}'.format(n+1, train_loss, v
al_loss))
```

In [50]:

```
def char_test(model, vocab_size, test_iter, criterion):
    ### YOUR CODE HERE
    '''
    params:
        model: LSTM model
        test_iter: test data
    return:
        ppl: perplexity
    '''
    ppl = None
    test_loss = 0
    target_num = 0
    with torch.no_grad():
        for batch in test_iter:
            text, targets = batch.text.to(device), batch.target.to(device)

            prediction = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))

            test_loss += loss.item() * targets.size(0) * targets.size(1)
            target_num += targets.size(0) * targets.size(1)

        test_loss /= target_num
```

```python
        ### Compute perplexity according to "test_loss"
        ### Hint: Consider how the loss is computed.
        ### YOUR CODE HERE(1 line)
        ppl = 2**(test_loss) # NLLoss
        ### END OF YOUR CODE
        return ppl
```

In [51]:

```python
num_epochs=10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
char_vocab_size = len(CHAR_TEXT.vocab)

config = {'vocab_size':char_vocab_size,
         'emb_size':128,
         'hidden_size':128,
         'num_layer':1}

CLM = CharLangModel(config)
CLM = CLM.to(device)

char_criterion = nn.NLLLoss(reduction='mean')
char_optimizer = optim.Adam(CLM.parameters(), lr=1e-3, betas=(0.7, 0.99))
```

In [52]:

```python
char_train(CLM, ctrain_iter, cvalid_iter, char_criterion, char_optimizer, char_vocab_size, num_epoc
hs)
```

```
Epoch: 1, Training Loss: 1.8430, Validation Loss: 1.5488
Epoch: 2, Training Loss: 1.5472, Validation Loss: 1.4417
Epoch: 3, Training Loss: 1.4721, Validation Loss: 1.3932
Epoch: 4, Training Loss: 1.4325, Validation Loss: 1.3657
Epoch: 5, Training Loss: 1.4077, Validation Loss: 1.3481
Epoch: 6, Training Loss: 1.3905, Validation Loss: 1.3360
Epoch: 7, Training Loss: 1.3778, Validation Loss: 1.3267
Epoch: 8, Training Loss: 1.3678, Validation Loss: 1.3193
Epoch: 9, Training Loss: 1.3596, Validation Loss: 1.3130
Epoch: 10, Training Loss: 1.3526, Validation Loss: 1.3076
```

In [53]:

```python
# <10
char_test(CLM, char_vocab_size, ctest_iter, char_criterion)
```

Out[53]:

```
2.466842651869756
```

In [54]:

```python
def char_greedy_search(model, start_token, max_len):
    '''
    param:
        model: nn.Module --- language model
        start_token: str --- e.g. 'h'
        max_len: int --- max number of tokens generated
    return:
        strings: list[str] --- list of tokens, e.g., ['h', 'e', ' ', 'i', 's',...]
    '''
    model.eval()
    ID = CHAR_TEXT.vocab.stoi[start_token]
    strings = [start_token]
    hidden = None

    ### You may find CHAR_TEXT.vocab.itos useful.
    ### YOUR CODE HERE
    for i in range(max_len):
        out = model(torch.LongTensor([[ID]]).to(device)) # out would be a id
        ID = torch.argmax(out, dim =-1)
        strings.append(CHAR_TEXT.vocab.itos[ID])
```

```
    ### END OF YOUR CODE
    return strings
```

In [55]:

```
char_greedy_search(CLM, 'a', 64)
```

Out[55]:

```
['a',
 'n',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h',
 'e',
 ' ',
 't',
 'h']
```

Requirements:

- This is an individual report.
- Complete the code using Python.
- List students with whom you have discussed if there are any.
- Follow the honor code strictly.

Free GPU Resources

We suggest that you run neural language models on machines with GPU(s). Google provides the free online platform Colaboratory, a research tool for machine learning education and research. It's a Jupyter notebook environment that requires no setup to use as common packages have been pre-installed. Google users can have access to a Tesla T4 GPU (approximately 15G memory). Note that when you connect to a GPU-based VM runtime, you are given a maximum of 12 hours at a time on the VM.

It is convenient to upload local Jupyter Notebook files and data to Colab, please refer to the tutorial.

In addition, Microsoft also provides the online platform Azure Notebooks for research of data science and machine learning, there are free trials for new users with credits.