drawing

50.040 Natural Language Processing, Summer 2020

Due 19 June 2020, 5pm

Mini Project

Write your student ID and name

STUDENT ID: 1003014

Name: Antonio Miguel Canlas Quizon

Students with whom you have discussed (if any):

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 [ ]:

```
%%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 google.colab import drive
drive.mount('/content/gdrive')
```

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

In [3]:

```
% pwd
% ls
% cd /content/gdrive/'My Drive'/'NLP'/'Mini Project'
% ls
```

```
gdrive/  sample_data/
/content/gdrive/My Drive/NLP/Mini Project
data/            greedy.png  __MACOSX/          sutd.png
Description.pdf  LM.png      mini_project.ipynb
```

In [4]:

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

In [5]:

```
with open('data/wikitext-2-v1/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 [6]:

```
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 [7]:

```
# en_y_to_x = dict()
# for i in numpy_en[:]:
#     en_y_to_x[i[0]+i[1]] = en_y_to_x.get(i[0]+i[1], 0) + 1

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
    ngram_dict = dict()
    for sentence in range(len(sents)):
      # print(sents[sentence])
      # temp_tuple = tuple()
      ngrams = list(zip(*[sents[sentence][i:] for i in range(n)]))
      # print(ngrams)
      for ng in range(len(ngrams)):
        ngram_dict[ngrams[ng]] = ngram_dict.get(ngrams[ng], 0) + 1


    ngram_set = ngram_dict.keys()
      # for i in range(len(sents[sentence]-n+1)):
      #   # print(sents[sentence][i])
      #   ngram_dict[sents[sentence][i]] = ngram_dict.get(sents[sentence][i], 0) + 1






    ### END OF YOUR CODE
    return ngram_set, ngram_dict
```

In [8]:

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

```
28910
```

In [9]:

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

```
577343
```

In [10]:

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

```
1344047
```

In [11]:

```
# List 10 most frequent unigrams, bigrams and trigrams as well as their counts.
# print(unigram_dict)
print('----------------- UNIGRAM--------------')
unigram_counter = Counter(unigram_dict)
print(unigram_counter.most_common(10))
print('----------------- BIGRAM--------------')
bigram_counter = Counter(bigram_dict)
print(bigram_counter.most_common(10))
print('----------------- TRIGRAM--------------')
trigram_counter = Counter(trigram_dict)
print(trigram_counter.most_common(10))
```

```
----------------- UNIGRAM--------------
[(('the',), 130519), ((',',), 99763), (('.',), 73388), (('of',), 56743), (('<unk>',), 53951), (('a
nd',), 49940), (('in',), 44876), (('to',), 39462), (('a',), 36140), (('"',), 28285)]
----------------- BIGRAM--------------
[(('of', 'the'), 17242), (('in', 'the'), 11778), ((',', 'and'), 11643), (('.', 'the'), 11274),
((',', 'the'), 8024), (('<unk>', ','), 7698), (('to', 'the'), 6009), (('on', 'the'), 4495),
(('the', '<unk>'), 4389), (('and', 'the'), 4331)]
----------------- TRIGRAM--------------
[((',', 'and', 'the'), 1393), ((',', '<unk>', ','), 950), (('<unk>', ',', '<unk>'), 901), (('one',
'of', 'the'), 866), (('<unk>', ',', 'and'), 819), (('.', 'however', ','), 775), (('<unk>',
'<unk>', ','), 745), (('.', 'in', 'the'), 726), (('.', 'it', 'was'), 698), (('the', 'united',
'states'), 666)]
```

In [12]:

```
# print(unigram_dict[('<START>',)])
```

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 [13]:

```
################################################
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 [14]:

```
import copy
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 = copy.deepcopy(sents)
    if n==1:
      return sents
    for sentence in range(len(sents)):
      for i in range(n-1):
        padded_sents[sentence].insert(i, START)
      padded_sents[sentence].append(STOP)
    # print(padded_sents[0])

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

In [15]:

```
uni_sents = pad_sents(train_sents, 1)
bi_sents = pad_sents(train_sents, 2)
tri_sents = pad_sents(train_sents, 3)
```

In [16]:

```
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 [17]:

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

Out[17]:

```
(28910, 580825, 1363266)
```

In [18]:

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

```
2007146
```

In [18]:

```

```

In [19]:

```
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\
    if len(ngram) == 1:
      if (ngram[0],) not in unigram_dict:
        return 0.0
      prob = unigram_dic[(ngram[0],)]/num_words
    elif len(ngram) == 2:
      if (ngram[0],) not in unigram_dict:
        return 0.0
      if (ngram[0],ngram[1]) not in bigram_dict:
        return 0.0
      prob = bigram_dict[(ngram[0],ngram[1])]/unigram_dic[(ngram[0],)]
    elif len(ngram) == 3:
      if (ngram[0],ngram[1]) not in bigram_dict:
        return 0.0
      if (ngram[0],ngram[1],ngram[2]) not in trigram_dict:
        return 0.0
      prob = trigram_dict[(ngram[0],ngram[1],ngram[2])]/bigram_dict[(ngram[0],ngram[1])]





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

In [20]:

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

Out[20]:

```
9.960235674499498e-05
```

In [21]:

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

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

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 [22]:

```
with open('data/wikitext-2-v1/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 [23]:

```
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\、
    numerator = 0
    denominator = 0
    # print(ngram)
    # try:
    #   if len(ngram) == 1:
    #     if (ngram[0],) not in unigram_dict:
    #       numerator = 0
    #       denominator = num_words
    #     else:
    #       numerator = unigram_dict[(ngram[0],)]
    #       denominator = num_words
    #   elif len(ngram) == 2:
    #     if (ngram[0],) not in unigram_dict and (ngram[0],ngram[1]) not in bigram_dict:
    #       numerator = 0
    #       denominator = 0
    #     elif (ngram[0],ngram[1]) not in bigram_dict:
    #       numerator = 0
    #     elif (ngram[0],) not in unigram_dict:
    #       denominator = 0
    #     else:
    #       numerator = bigram_dict[(ngram[0],ngram[1])]
    #       denominator = unigram_dic[(ngram[0],)]
    #   elif len(ngram) == 3:
    #     if (ngram[0],ngram[1]) not in bigram_dict and (ngram[0],ngram[1],ngram[2]) not in
trigram_dict:
    #       numerator = 0
    #       denominator = 0
    #     elif (ngram[0],ngram[1]) not in bigram_dict:
    #       denominator = 0
    #     elif (ngram[0],ngram[1],ngram[2]) not in trigram_dict:
    #       numerator = 0
    #     else:
    #       numerator = trigram_dict[(ngram[0],ngram[1],ngram[2])]
    #       denominator = bigram_dict[(ngram[0],ngram[1])]
    #   numerator += k
    #   denominator += V*k
    #   s_prob = numerator/denominator
    # except Exception as e:
    #   print(e)

    try:
      if len(ngram) == 1:
        numerator = unigram_dict.get((ngram[0],),0)
        denominator = num_words
      elif len(ngram) == 2:
        numerator = bigram_dict.get((ngram[0],ngram[1]),0)
        denominator = unigram_dict.get((ngram[0],),0)
      elif len(ngram) == 3:
        numerator = trigram_dict.get((ngram[0],ngram[1],ngram[2]),0)
        denominator = bigram_dict.get((ngram[0],ngram[1]),0)
      numerator += k
      denominator += V*k
```

```
      s_prob = numerator/denominator
    except Exception as e:
      print(e)

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

In [24]:

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

Out[24]:

```
9.311982452086402e-05
```

In [25]:

```
for ng in ngrams:
  print(smooth_ngram_prob(ng,0.5, num_words,unigram_dict, bigram_dict, trigram_dict))
```

```
9.311982452086402e-05
0.00274418131923976
0.002482635498898l563
0.0672640l689559053
3.169672572823227e-05
0.02127371731998512
0.0005184033177812338
0.00437373006853325
3.4584125886218224e-05
3.456738912509938e-05
```

In [26]:

```
import math
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
    # print(len(valid_sents))
    total = 0
    test_words = 0
    for sentence in range(len(valid_sents)):
      test_words += len(valid_sents[sentence])
      ngrams = list(zip(*[valid_sents[sentence][i:] for i in range(n)]))
      # print(ngrams)
      for ng in ngrams:
        # print(ng)
        # print(smooth_ngram_prob(ng, k, num_words, unigram_dic, bigram_dic, trigram_dic))
        total += math.log(smooth_ngram_prob(ng, k, num_words, unigram_dic, bigram_dic, trigram_dic)
,2.0)
    l = 1/test_words
    ppl = 2**(l*total*-1)
    # print(valid_sents[0])
    ### END OF YOUR CODE
    return ppl
```

In [27]:

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

Out[27]:

```
840.7347306217125
```

In [28]:

```
n = [1,2,3]
k = [0.1, 0.3, 0.5, 0.7, 0.9]
### YOUR CODE HEREb
best = 10000000000000
for i in range(len(n)):
  for j in range(len(k)):
    print('------------------- N = {} K = {} --------------------'.format(n[i],k[j]))
    if n[i]==1:
      res = perplexity(n[i], k[j], num_words, uni_valid_sents, unigram_dict, bigram_dict, 
trigram_dict)
      best = min(best,res)
      print(res)
    elif n[i]==2:
      res = perplexity(n[i], k[j], num_words, bi_valid_sents, unigram_dict, bigram_dict, 
trigram_dict)
      best = min(best,res)
      print(res)
    elif n[i]==3:
      res = perplexity(n[i], k[j], num_words, tri_valid_sents, unigram_dict, bigram_dict, 
trigram_dict)
      best = min(best,res)
      print(res)




### END OF YOUR CODE
```

```
------------------- N = 1 K = 0.1 --------------------
840.7347306217125
------------------- N = 1 K = 0.3 --------------------
841.1427277044075
------------------- N = 1 K = 0.5 --------------------
841.5959678936316
------------------- N = 1 K = 0.7 --------------------
842.0904494786319
------------------- N = 1 K = 0.9 --------------------
842.6227084935349
------------------- N = 2 K = 0.1 --------------------
739.5817358293406
------------------- N = 2 K = 0.3 --------------------
1061.3982617375789
------------------- N = 2 K = 0.5 --------------------
1289.1491260338778
------------------- N = 2 K = 0.7 --------------------
1477.190939955735
------------------- N = 2 K = 0.9 --------------------
1641.5907324574955
------------------- N = 3 K = 0.1 --------------------
4773.649128295989
------------------- N = 3 K = 0.3 --------------------
6676.617325676175
------------------- N = 3 K = 0.5 --------------------
7831.228457980847
------------------- N = 3 K = 0.7 --------------------
8684.056079338212
------------------- N = 3 K = 0.9 --------------------
9364.604903261927
```

In [28]:

Results

Therefore, the best combination would be **N = 2, K = 0.1** with 739.58 perplexity

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 [29]:

```
with open('data/wikitext-2-v1/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 [30]:

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

Out[30]:

```
689.3929590954306
```

Neural Language Model (RNN)

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 [31]:

```
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 /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

Out[31]:

```
<torch._C.Generator at 0x7fe96c053c30>
```

In [32]:

```
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 [33]:

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

```
Vocabulary size: 28908
```

In [34]:

```
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 [35]:

```
#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 [36]:

```
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(num_embeddings=self.vocab_size,embedding_dim=self.emb_size)
        self.rnn = nn.LSTM(input_size=self.emb_size,hidden_size=self.hidden_size,num_layers=self.nu
m_layer)
        self.linear = nn.Linear(in_features=self.hidden_size,out_features=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
        batch_sents = self.embedding(batch_sents)
        batch_sents,hidden = self.rnn(batch_sents,hidden)
        batch_sents = self.linear(batch_sents)
        normalized_score = F.log_softmax(batch_sents,dim=2)


        ### END OF YOUR CODE
        return normalized_score, hidden
```

In [37]:

```
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)

            optimizer.zero_grad()
            prediction,_ = model(text)
            loss = criterion(prediction.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 [38]:

```
import math
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 = math.exp(test_loss)
        ### END OF YOUR CODE
        return ppl
```

In [39]:

```
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')
optimizer = optim.Adam(LM.parameters(), lr=1e-3, betas=(0.7, 0.99))
```

In [40]:

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

```
Epoch: 1, Training Loss: 6.0691, Validation Loss: 5.1777
Epoch: 2, Training Loss: 5.4015, Validation Loss: 4.9643
Epoch: 3, Training Loss: 5.1293, Validation Loss: 4.8661
Epoch: 4, Training Loss: 4.9561, Validation Loss: 4.8139
Epoch: 5, Training Loss: 4.8310, Validation Loss: 4.7835
Epoch: 6, Training Loss: 4.7311, Validation Loss: 4.7638
Epoch: 7, Training Loss: 4.6476, Validation Loss: 4.7500
Epoch: 8, Training Loss: 4.5765, Validation Loss: 4.7421
Epoch: 9, Training Loss: 4.5150, Validation Loss: 4.7385
Epoch: 10, Training Loss: 4.4606, Validation Loss: 4.7391
```

In [41]:

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

Out[41]:

```
99.14565658706458
```

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 [42]:

```
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

    end = TEXT.vocab.stoi["<eos>"]
    # print(end)
    hidden = None
    for _ in range(max_len):
      norm,hidden = model(torch.tensor([ID]).unsqueeze(1).to(device),hidden)
      ID =  torch.argmax(norm)
      nextword = TEXT.vocab.itos[ID.item()]
      strings.append(nextword)
      # print(strings)
      if(ID.item()== end):
        break

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

In [90]:

```
# 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


    k_beam = [(0, [0]*(max_len+1))]

    # l : point on target sentence to predict
```

```
    for l in range(max_len):
        all_k_beams = []
        for prob, sent_predict in k_beam:
            # predicted = model.predict([np.array([src_input]), np.array([sent_predict])])[0]
            # # top k!
            # possible_k = predicted[l].argsort()[-beam_size:][::-1]
            norm,hidden = model(torch.tensor([ID]).unsqueeze(1).to(device),hidden)
            possible_k = torch.topk(norm,beam_size)
            # print(possible_k)
            ID = possible_k.indices[0][0].tolist()
            print(ID)

            # add to all possible candidates for k-beams
            # all_k_beams += [
            #     (
            #         sum(np.log(possible_k.values[0][0].tolist()[i][sent_predict[i+1]]) for i in r
ange(l)) + np.log(possible_k.values[0][0].tolist()[i][next_wid]),
            #         list(sent_predict[:l+1])+[next_wid]+[0]*(max_len-l-1)
            #     )
            #     for next_wid in possible_k.indices[0][0].tolist()
            # ]
            for next_wid in possible_k.indices[0][0].tolist():
              for i in range(l):
                all_k_beams += [
                  (
                      sum(np.log(possible_k.values[0][0].tolist()[i][sent_predict[i+1]]) ) + np.log
possible_k.values[0][0].tolist()[i][next_wid]),
                      list(sent_predict[:l+1])+[next_wid]+[0]*(max_len-l-1)
                  )]
                print(all_k_beams)




        # top k
            k_beam = sorted(all_k_beams)[-beam_size:]

    return k_beam

    ### You may find TEXT.vocab.itos useful.
    ### YOUR CODE HERE
    # ret = []
    # end = TEXT.vocab.stoi["<eos>"]
    # sequences = [[list(), 0.0]]
    # # print(end)
    # hidden = None
    # norm,hidden = model(torch.tensor([ID]).unsqueeze(1).to(device),hidden)
    # ID = torch.topk(norm,beam_size)
    # for _ in range(1,max_len):
    #   all_candidates = list()
    #   for i in range(1,max_len):
    #     for j in range(beam_size):
    #       norm,hidden = model(torch.tensor([ID.indices[0][0][j]]).unsqueeze(1).to(device),hidden,
    #       ID = torch.topk(norm,beam_size)
    #       candidate = [seq + ID.indices[0][0][j], ID.scores[0][0][j] - log(row[j])]
    #       all_candidates.append(candidate)
      # ordered = torch.sort(norm,descending= True)
      # print(ordered[:beam_size])


      # ID =  torch.argmax(norm)
      # nextword = TEXT.vocab.itos[ID.item()]
      # strings.append(nextword)
      # all_candidates = list()
      # for i in range(len(sequences)):
      #   seq, score = sequences[i]
      #   for j in range(len(row)):
      #     candidate = [seq + [j], score - log(row[j])]
      #     all_candidates.append(candidate)
      # if(ID.item()== end):
      #   break
      # ordered = sorted(all_candidates, key=lambda tup:tup[1])
  # select k best
      # sequences = ordered[:beam_size]

    return
```

Result

I tried to implement the beam search, but was not able to produce the desired output. I tried by finding the topk and adding the log likelihood of all the words.

In [91]:

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

Out[91]:

```
['he', 'was', 'a', 'member', 'of', 'the', '<', 'unk', '>', '.', '<eos>']
```

In [92]:

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

```
[19, 38, 30]
```

Out[92]:

```
[]
```

char-level LM

Question 7 [code]

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

In [52]:

```
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 = []
    [char_list.append(x) for x in string]
    ### END OF YOUR CODE
    return char_list
```

In [53]:

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

Out[53]:

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

In [54]:

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

In [55]:

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

```
Vocabulary size: 247
```

In [56]:

```
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 [57]:

```
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 = None
        self.rnn = None
        self.linear = None

        self.embedding = nn.Embedding(num_embeddings=self.vocab_size,embedding_dim=self.emb_size)
        self.rnn = nn.LSTM(input_size=self.emb_size,hidden_size=self.hidden_size,num_layers=self.nu
m_layer)
        self.linear = nn.Linear(in_features=self.hidden_size,out_features=self.vocab_size)

    def forward(self, batch_sents, hidden):
        ### YOUR CODE HERE
        batch_sents = self.embedding(batch_sents)
        batch_sents,hidden = self.rnn(batch_sents,hidden)
        batch_sents = self.linear(batch_sents)
        normalized_score = F.log_softmax(batch_sents,dim=2)


        ### END OF YOUR CODE
        return normalized_score, hidden
```

In [58]:

```
def char_train(model, train_iter, valid_iter, criterion, optimizer, vocab_size, 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)

            optimizer.zero_grad()
            prediction,_ = model(text,None)
            loss = criterion(prediction.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,None)
            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 [59]:

```
def char_test(model, vocab_size, test_iter, criterion):
    '''
    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,None)
            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 = math.exp(test_loss)
        ### END OF YOUR CODE
        return ppl
```

In [60]:

```
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 [61]:

```
char_train(CLM, ctrain_iter, cvalid_iter, char_criterion, char_optimizer, char_vocab_size, num_epoc
```

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

```
Epoch: 1, Training Loss: 1.8418, Validation Loss: 1.5487
Epoch: 2, Training Loss: 1.5469, Validation Loss: 1.4422
Epoch: 3, Training Loss: 1.4730, Validation Loss: 1.3966
Epoch: 4, Training Loss: 1.4354, Validation Loss: 1.3714
Epoch: 5, Training Loss: 1.4115, Validation Loss: 1.3548
Epoch: 6, Training Loss: 1.3946, Validation Loss: 1.3425
Epoch: 7, Training Loss: 1.3818, Validation Loss: 1.3323
Epoch: 8, Training Loss: 1.3713, Validation Loss: 1.3239
Epoch: 9, Training Loss: 1.3628, Validation Loss: 1.3170
Epoch: 10, Training Loss: 1.3559, Validation Loss: 1.3114
```

In [62]:

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

Out[62]:

```
3.685492545227015
```

In [67]:

```
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
    end = CHAR_TEXT.vocab.stoi["<eos>"]
    # print(end)
    hidden = None
    for _ in range(1,max_len):
      norm,hidden = model(torch.tensor([ID]).unsqueeze(1).to(device),hidden)
      ID =  torch.argmax(norm)
      nextword = CHAR_TEXT.vocab.itos[ID.item()]
      strings.append(nextword)
      # print(strings)
      if(ID.item()== end):
        break
    ### END OF YOUR CODE
    return strings
```

In [68]:

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

Out[68]:

```
['h',
 'e',
 ' ',
 's',
 't',
 'a',
 't',
 'e',
 ' ',
 ',',
 ' ',
 'a',
 'n',
 'd',
 ' ',
```

```
 't',
 'h',
 'e',
 ' ',
 's',
 'e',
 'c',
 'o',
 'n',
 'd',
 ' ',
 't',
 'h',
 'e',
 ' ',
 's',
 't',
 'o',
 'r',
 'y',
 ' ',
 ',',
 ' ',
 'a',
 'n',
 'd',
 ' ',
 't',
 'h',
 'e',
 ' ',
 's',
 't',
 'o',
 'r',
 'y',
 ' ',
 ',',
 ' ',
 'a',
 'n',
 'd',
 ' ',
 't',
 'h',
 'e',
 ' ',
 's',
 't']
```

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.