drawing

50.040 Natural Language Processing, Summer 2020

Due 19 June 2020, 5pm
Mini Project

Write your student ID and name

STUDNET ID: 1002934

Name: Chloe Zheng

Students with whom you have discussed (if any):

Lionell, Yi Xuan, Jia Yee, Yong Quan, Caleb, Bryan

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 $x_1, x_2, \ldots, x_m$, 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: $p(x_1, x_2, \ldots, x_m)$ 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: $p(x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i)$ 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: $p(x_0, x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-1})$ Under the second-order Markovian assumption, the joint probability can be written as: $p(x_{-1}, x_0, x_1, x_2, \ldots, x_m) = \prod_{i=1}^{m} p(x_i \mid x_{i-2}, x_{i-1})$ Similar to what we did in HMM, we will assume that $x_{-1} = \mathrm{START}$, $x_0 = \mathrm{START}$, $x_m = \mathrm{STOP}$ 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, $u \in V \cup \mathrm{STOP}$ and $w, v \in V \cup \mathrm{START}$.

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

In [275]:
```
%%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: $p_{add-k}(u) = \frac{\mathrm{count}(u) + k}{c + k|V^*|}$ $p_{add-k}(u \mid v) = \frac{\mathrm{count}(v, u) + k}{\mathrm{count}(v) + k|V^*|}$

$p_{add-k}(u \mid w, v) = \frac{\mathrm{count}(w, v, u) + k}{\mathrm{count}(w, v) + k|V^*|}$

where $k \in (0, 1)$ is the parameter of this approach, and $|V^*|$ is the size of the vocabulary V^*,here $V^* = V \cup \mathrm{STOP}$. 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' consisting of sentences $X^{(1)}, X^{(2)}, \ldots, X^{(|D'|)}$, each sentence $X^{(j)}$ consists of words $x_{(j)1}, x_{(j)2}, \ldots, x_{(j)n_j}$, 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: $D' \prod j p(X^{(j)})$ Let's define average log2 probability as: $l = 1c' |D'| \sum j=1 \log_2 p(X^{(j)})$ c' is the total number of words in the test set, D' is the number of sentences. And the perplexity is defined as: perplexity = 2^{-l}

The lower the perplexity, the better the language model.

In [276]:
```
from collections import Counter, namedtuple
import itertools
import numpy as np
import itertools
import math
```

In [277]:
```
#file = open('/content/drive/My Drive/Colab Notebooks/data/wikitext-2/wiki.train.tokens')

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 [278]:
```
print(train_sents[1])
```

```
['the', 'game', 'began', 'development', 'in', '2010', ',', 'carrying', 'over', 'a', 'large', 'portion', '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', 'series', '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', 'handled', '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 [279]:
```
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 ng
ram 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 = {}

    grams = []

    # iterate over outer loop
    for sentence in sents:
        for word in range(len(sentence) - n + 1):
            for gram in range(n):
                grams.append(sentence[word + gram])

            tuple_hold = tuple(grams)
            ngram_dict.setdefault(tuple_hold, 0)
            # empty list in ea iteration
            grams = []
            ngram_dict[tuple_hold] +=1

    ngram_set = ngram_dict.keys()

    return ngram_set, ngram_dict
```

In [280]:

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

28910

In [281]:

```
### ~57xxxx 577343
bigram_set, bigram_dict = compute_ngram(train_sents, 2)
print(len(bigram_set))
```

577343

In [282]:

```
### ~134xxxx 1344047
trigram_set, trigram_dict = compute_ngram(train_sents, 3)
print(len(trigram_set))
```

1344047

In [283]:

```
# List 10 most frequent unigrams, bigrams and trigrams as well as their counts.

counts_uni = Counter(unigram_dict).most_common(10)

counts_bi = Counter(bigram_dict).most_common(10)

counts_tri = Counter(trigram_dict).most_common(10)

print("uni", counts_uni)
print("\n\nbi", counts_bi)
print("\n\ntri", counts_tri)
```

uni [(('the',), 130519), ((',',), 99763), (('.',), 73388), (('of',), 56743), (('<unk>',), 53951), (('and',), 49940), (('in',), 44876), (('to',), 39462), (('a',), 36140), (('"',), 28285)]

bi [(('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)]

tri [((',', '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)]

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 [284]:

```
##################################################


ngrams = list()
with open('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 [285]:

```
def pad_sent(sent, n):
    if n > 1:
        padded = [START for _ in range(n-1)]
        padded.extend(sent)
    else:
        padded = sent
    return padded
```

In [286]:

```
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 = [pad_sent(sent, n) for sent in sents]
    print(padded_sents[0])

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

In [287]:

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

```
['senjō', 'no', 'valkyria', '3', ':', '<unk>', 'chronicles', '(', 'japanese', ':', '戦場のヴァルキュリア3', ',', 'lit', '.', 'valkyria', 'of', 'the', 'battlefield', '3', ')', ',', 'commonly', 'referred', 'to', 'as', 'valkyria', 'chronicles', 'iii', 'outside', 'japan', ',', 'is', 'a', 'tactical', 'role', '@-@', 'playing', 'video', 'game', 'developed', 'by', 'sega', 'and', 'media.vision', 'for', 'the', 'playstation', 'portable', '.', 'released', 'in', 'january', '2011', 'in', 'japan', ',', 'it', 'is', 'the', 'third', 'game', 'in', 'the', 'valkyria', 'series', '.', '<unk>', 'the', 'same', 'fusion', 'of', 'tactical', 'and', 'real', '@-@', 'time', 'gameplay', 'as', 'its', 'predecessors', ',', 'the', 'story', 'runs', 'parallel', 'to', 'the', 'first', 'game', 'and', 'follows', 'the', '"', 'nameless', '"', ',', 'a', 'penal', 'military', 'unit', 'serving', 'the', 'nation', 'of', 'gallia', 'during', 'the', 'second', 'europan', 'war', 'who', 'perform', 'secret', 'black', 'operations', 'and', 'are', 'pitted', 'against', 'the', 'imperial', 'unit', '"', '<unk>', 'raven', '"', '.']
['<START>', 'senjō', 'no', 'valkyria', '3', ':', '<unk>', 'chronicles', '(', 'japanese', ':', '戦場のヴァルキュリア3', ',', 'lit', '.', 'valkyria', 'of', 'the', 'battlefield', '3', ')', ',', 'commonly', 'referred', 'to', 'as', 'valkyria', 'chronicles', 'iii', 'outside', 'japan', ',', 'is', 'a', 'tactical', 'role', '@-@', 'playing', 'video', 'game', 'developed', 'by', 'sega', 'and', 'media.vision', 'for', 'the', 'playstation', 'portable', '.', 'released', 'in', 'january', '2011', 'in', 'japan', ',', 'it', 'is', 'the', 'third', 'game', 'in', 'the', 'valkyria', 'series', '.', '<unk>', 'the', 'same', 'fusion', 'of', 'tactical', 'and', 'real', '@-@', 'time', 'gameplay', 'as', 'its', 'predecessors', ',', 'the', 'story', 'runs', 'parallel', 'to', 'the', 'first', 'game', 'and', 'follows', 'the', '"', 'nameless', '"', ',', 'a', 'penal', 'military', 'unit', 'serving', 'the', 'nation', 'of', 'gallia', 'during', 'the', 'second', 'europan', 'war', 'who', 'perform', 'secret', 'black', 'operations', 'and', 'are', 'pitted', 'against', 'the', 'imperial', 'unit', '"', '<unk>', 'raven', '"', '.']
['<START>', '<START>', 'senjō', 'no', 'valkyria', '3', ':', '<unk>', 'chronicles', '(', 'japanese', ':', '戦場のヴァルキュリア3', ',', 'lit', '.', 'valkyria', 'of', 'the', 'battlefield', '3', ')', ',', 'commonly', 'referred', 'to', 'as', 'valkyria', 'chronicles', 'iii', 'outside', 'japan', ',', 'is', 'a', 'tactical', 'role', '@-@', 'playing', 'video', 'game', 'developed', 'by', 'sega', 'and', 'media.vision', 'for', 'the', 'playstation', 'portable', '.', 'released', 'in', 'january', '2011', 'in', 'japan', ',', 'it', 'is', 'the', 'third', 'game', 'in', 'the', 'valkyria', 'series', '.', '<unk>', 'the', 'same', 'fusion', 'of', 'tactical', 'and', 'real', '@-@', 'time', 'gameplay', 'as', 'its', 'predecessors', ',', 'the', 'story', 'runs', 'parallel', 'to', 'the', 'first', 'game', 'and', 'follows', 'the', '"', 'nameless', '"', ',', 'a', 'penal', 'military', 'unit', 'serving', 'the', 'nation', 'of', 'gallia', 'during', 'the', 'second', 'europan', 'war', 'who', 'perform', 'secret', 'black', 'operations', 'and', 'are', 'pitted', 'against', 'the', 'imperial', 'unit', '"', '<unk>', 'raven', '"', '.']
```

In [288]:

```
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 [289]:

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

Out[289]:

```
(28910, 580115, 1356254)
```

In [290]:

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

2007146

In [291]:
```
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 occurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-
gram to its number of occurence 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

    ngram_tuple = tuple(ngram)

    if len(ngram) == 1:
        #prob = count(u) / c
        # to find number of occurrence the unigram list appears from the unigram_dict
        prob = unigram_dic.get(ngram_tuple, 0) / num_words

    elif len(ngram) == 2:
        # count(v,u)/ count(v)
        prob = bigram_dic.get(ngram_tuple, 0) / unigram_dic.get(tuple([ngr
am_tuple[0]]), 0)

    elif len(ngram) ==3:
        #count(w,v,u)/count(w,v)
        prob = trigram_dic.get(ngram_tuple, 0) / bigram_dic.get(tuple(ngra
m_tuple[0:2]), 0)

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

In [292]:
```
### ~9.96e-05
print(tuple(ngrams))
ngram_tuple = tuple(ngrams)
print('weird', ngram_tuple[0])

ngram_prob(ngrams[0], num_words,unigram_dict, bigram_dict, trigram_dict)
```

(['the', 'computer'], ['go', 'to'], ['have', 'had'], ['and', 'the'], ['can', 'sea'], ['a', 'number', 'of'], ['with', 'respect', 'to'], ['i
n', 'terms', 'of'], ['not', 'good', 'bad'], ['first', 'start', 'with'])
weird ['the', 'computer']

Out[292]: 9.960235674499498e-05

In [293]:
```
### List down the n-grams that have 0 probability.

# iterate through ngrams
for given in ngrams:
```

```
#    print('give', given)
    get_ngram_prob = ngram_prob(given, num_words,unigram_dict, bigram_dict
, trigram_dict)
    # get the ith ngram and put into get_ngram_prob
    if get_ngram_prob == 0:
        print('ngrams with 0 probability:', given)
```

ngrams with 0 probability: ['can', 'sea']
ngrams with 0 probability: ['not', 'good', 'bad']
ngrams with 0 probability: ['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 ∊ [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 [294]:
```
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)

print(bi_valid_sents[1])
```

['homarus', 'gammarus', ',', 'known', 'as', 'the', 'european', 'lobster', 'or', 'common', 'lobster', ',', 'is', 'a', 'species', 'of', '<unk>', 'lobster', 'from', 'the', 'eastern', 'atlantic', 'ocean', ',', 'mediterranean', 'sea', 'and', 'parts', 'of', 'the', 'black', 'sea', '.', 'it', 'is', 'closely', 'related', 'to', 'the', 'american', 'lobster', ',', 'h.', 'americanus', '.', 'it', 'may', 'grow', 'to', 'a', 'length', 'of', '60', 'cm', '(', '24', 'in', ')', 'and', 'a', 'mass', 'of', '6', 'kilograms', '(', '13', 'lb', ')', ',', 'and', 'bears', 'a', 'conspicuous', 'pair', 'of', 'claws', '.', 'in', 'life', ',', 'the', 'lobsters', 'are', 'blue', ',', 'only', 'becoming', '"', 'lobster', 'red', '"', 'on', 'cooking', '.', 'mating', 'occurs', 'in', 'the', 'summer', ',', 'producing', 'eggs', 'which', 'are', 'carried', 'by', 'the', 'females', 'for', 'up', 'to', 'a', 'year', 'before', 'hatching', 'into', '<unk>', 'larvae', '.', 'homarus', 'gammarus', 'is', 'a', 'highly', 'esteemed', 'food', ',', 'and', 'is', 'widely', 'caught', 'using', 'lobster', 'pots', ',', 'mostly', 'around', 'the', 'british', 'isles', '.']
['<START>', 'homarus', 'gammarus', ',', 'known', 'as', 'the', 'european', 'lobster', 'or', 'common', 'lobster', ',', 'is', 'a', 'species', 'of', '<unk>', 'lobster', 'from', 'the', 'eastern', 'atlantic', 'ocean', ',', 'mediterranean', 'sea', 'and', 'parts', 'of', 'the', 'black', 'sea', '.', 'it', 'is', 'closely', 'related', 'to', 'the', 'american', 'lobster', ',', 'h.', 'americanus', '.', 'it', 'may', 'grow', 'to', 'a', 'length', 'of', '60', 'cm', '(', '24', 'in', ')', 'and', 'a', 'mass', 'of', '6', 'kilograms', '(', '13', 'lb', ')', ',', 'and', 'bears', 'a', 'conspicuous', 'pair', 'of', 'claws', '.', 'in', 'life', ',', 'the', 'lobsters', 'are', 'blue', ',', 'only', 'becoming', '"', 'lobster', 'red', '"', 'on', 'cooking', '.', 'mating', 'occurs', 'in', 'the', 'summer', ',', 'producing', 'eggs', 'which', 'are', 'carried', 'by', 'the', 'females', 'for', 'up', 'to', 'a', 'year', 'before', 'hatching', 'into', '<unk>', 'larvae', '.', 'homarus', 'gammarus', 'is', 'a', 'highly', 'esteemed', 'food', ',', 'and', 'is', 'widely', 'caught', 'using', 'lobster', 'pots', ',', 'mostly', 'around', 'the', 'british', 'isles', '.']

['<START>', '<START>', 'homarus', 'gammarus', ',', 'known', 'as', 'the', 'european', 'lobster', 'or', 'common', 'lobster', ',', 'is', 'a', 'species', 'of', '<unk>', 'lobster', 'from', 'the', 'eastern', 'atlantic', 'ocean', ',', 'mediterranean', 'sea', 'and', 'parts', 'of', 'the', 'black', 'sea', '.', 'it', 'is', 'closely', 'related', 'to', 'the', 'american', 'lobster', ',', 'h.', 'americanus', '.', 'it', 'may', 'grow', 'to', 'a', 'length', 'of', '60', 'cm', '(', '24', 'in', ')', 'and', 'a', 'mass', 'of', '6', 'kilograms', '(', '13', 'lb', ')', ',', 'and', 'bears', 'a', 'conspicuous', 'pair', 'of', 'claws', '.', 'in', 'life', ',', 'the', 'lobsters', 'are', 'blue', ',', 'only', 'becoming', '"', 'lobster', 'red', '"', 'on', 'cooking', '.', 'mating', 'occurs', 'in', 'the', 'summer', ',', 'producing', 'eggs', 'which', 'are', 'carried', 'by', 'the', 'females', 'for', 'up', 'to', 'a', 'year', 'before', 'hatching', 'into', '<unk>', 'larvae', '.', 'homarus', 'gammarus', 'is', 'a', 'highly', 'esteemed', 'food', ',', 'and', 'is', 'widely', 'caught', 'using', 'lobster', 'pots', ',', 'mostly', 'around', 'the', 'british', 'isles', '.']
['<START>', 'homarus', 'gammarus', 'is', 'a', 'large', '<unk>', ',', 'with', 'a', 'body', 'length', 'up', 'to', '60', 'centimetres', '(', '24', 'in', ')', 'and', 'weighing', 'up', 'to', '5', '–', '6', 'kilograms', '(', '11', '–', '13', 'lb', ')', ',', 'although', 'the', 'lobsters', 'caught', 'in', 'lobster', 'pots', 'are', 'usually', '23', '–', '38', 'cm', '(', '9', '–', '15', 'in', ')', 'long', 'and', 'weigh', '0', '@.@', '7', '–', '2', '@.@', '2', 'kg', '(', '1', '@.@', '5', '–', '4', '@.@', '9', 'lb', ')', '.', 'like', 'other', 'crustaceans', ',', 'lobsters', 'have', 'a', 'hard', '<unk>', 'which', 'they', 'must', 'shed', 'in', 'order', 'to', 'grow', ',', 'in', 'a', 'process', 'called', '<unk>', '(', '<unk>', ')', '.', 'this', 'may', 'occur', 'several', 'times', 'a', 'year', 'for', 'young', 'lobsters', ',', 'but', 'decreases', 'to', 'once', 'every', '1', '–', '2', 'years', 'for', 'larger', 'animals', '.']

In [295]:
```
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 occurences in "sents";
        bigram_dicV = len(unigram_dic) + 1 : dict{ngram: counts} --- a dictionary that maps each 2-gram to its number of occurence 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\

    ngram_tuple = tuple(ngram)

    if len(ngram) == 1:
        #prob = count(u) / c
        # to find number of occurrence the unigram list appears from the unigram_dict
        s_prob = (unigram_dic.get(ngram_tuple, 0) + k) / (num_words + k*V)

    elif len(ngram) == 2:
        # count(v,u)/ count(v)
        s_prob = (bigram_dic.get(ngram_tuple, 0) + k) / (unigram_dic.get(tuple([ngram_tuple[0]]), 0) + k*V)

    elif len(ngram) ==3:
        #count(w,v,u)/count(w,v)
        s_prob = (trigram_dic.get(ngram_tuple, 0)+k) / (bigram_dic.get(tuple(ngram_tuple[0:2]), 0) + k*V)

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

In [296]:
```
### ~ 9.31e-05
smooth_ngram_prob(ngrams[0], 0.5, num_words, unigram_dict, bigram_dict, tr
igram_dict)
```

Out[296]: 9.311982452086402e-05

In [297]:
```
def perplexity(n, k, num_words, valid_sents, unigram_dic, bigram_dic, trig
ram_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 occurences in "sents";
        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-
gram to its number of occurence 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
    prob_sentence = 0;

    # get sentence
    for sents in valid_sents:
        sum_prob = 0
        for i in range(len(sents)-n+1):

            # get ngrams
            ngram = sents[i:i+n]
            # get prob of ngram from list of words
            prob = smooth_ngram_prob(ngram, k, num_words, unigram_dic, big
ram_dic, trigram_dic)
            log_prob = math.log(prob,2)
            sum_prob += log_prob

        # summation of log2 p(x^(j) over test set D'
        prob_sentence += sum_prob

    # get len of each sent
    ea_sent = [len(l) for l in valid_sents]

    # sum the len of each sent
    len_ts = sum(ea_sent)
    l = (1/len_ts)*prob_sentence
    ppl = 2**(-l)

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

In [298]:
```
### ~ 840
```

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

Out[298]: 840.7347306258201

In [299]:
```
n = [1,2,3]
k = [0.1, 0.3, 0.5, 0.7, 0.9]

# for each n, find the best k (smallest perp)
### YOUR CODE HERE
d = {}
for ngram in n:
    # iterate over the k values
    for kth in k:
        ppl = perplexity(ngram, kth, num_words, uni_valid_sents, unigram_dict, bigram_dict, trigram_dict)
        d[kth] = ppl
#       print('dis', ngram, kth, ppl)
    min_val = min(d.items(), key=lambda x: x[1])
    print('for n =',ngram,'the best k that gives the smallest perp: ', min_val)

### END OF YOUR CODE
```

for n = 1 the best k that gives the smallest perp: (0.1, 840.7347306258201)
for n = 2 the best k that gives the smallest perp: (0.1, 788.5223577024464)
for n = 3 the best k that gives the smallest perp: (0.1, 5079.078804814675)

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 [300]:
```
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)
```

['robert', '<unk>', 'is', 'an', 'english', 'film', ',', 'television', 'and', 'theatre', 'actor', '.', 'he', 'had', 'a', 'guest', '@-@', 'starring', 'role', 'on', 'the', 'television', 'series', 'the', 'bill', 'in', '2000', '.', 'this', 'was', 'followed', 'by', 'a', 'starring', 'role', 'in', 'the', 'play', 'herons', 'written', 'by', 'simon', 'stephens', ',', 'which', 'was', 'performed', 'in', '2001', 'at', 'the', 'royal', 'court', 'theatre', '.', 'he', 'had', 'a', 'guest', 'role', 'in', 'the', 'television', 'series', 'judge', 'john', '<unk>', 'in', '2002', '.', 'in', '2004', '<unk>', 'landed', 'a', 'role', 'as', '"', 'craig', '"', 'in', 'the', 'episode', '"', 'teddy', "'s", 'story', '"', 'of', 'the', 'television', 'series', 'the', 'long', 'firm', ';', 'he', 'starred', 'alongside', 'actors', 'mark', 'strong', 'and', 'derek', 'jacobi', '.', 'he', 'was', 'cast', 'in', 'the', '2005', 'theatre', 'productions', 'of', 'the', 'philip', 'ridley', 'play', 'mercury', 'fur', ',', 'which', 'was', 'performed', 'at', 'the', 'drum', 'theatre', 'in', 'plymouth', 'and', 'the', '<unk>', '<unk>', 'factory', 'in', 'london', '.', 'he', 'was', 'directed', 'by', 'john', '<unk>', 'and', 'starred', 'alongside', 'ben', '<unk>', ',', 'shane', '<unk>', ',', 'harry', 'kent', ',', 'fraser', '<unk>', ',', 'sophie', 'stanton', 'and', 'dominic', 'hall', '.']
['<START>', 'robert', '<unk>', 'is', 'an', 'english', 'film', ',', 'television', 'and', 'theatre', 'actor', '.', 'he', 'had', 'a', 'guest', '@-@', 'starring', 'role', 'on', 'the', 'television', 'series', 'the', 'bill', 'in', '2000', '.', 'this', 'was', 'followed', 'by', 'a', 'starring', 'role', 'in', 'the', 'play', 'herons', 'written', 'by', 'simon', 'stephens', ',', 'which', 'was', 'performed', 'in', '2001',

'at', 'the', 'royal', 'court', 'theatre', '.', 'he', 'had', 'a', 'guest', 'role', 'in', 'the', 'television', 'series', 'judge', 'john', '<unk>', 'in', '2002', '.', 'in', '2004', '<unk>', 'landed', 'a', 'role', 'as', '"', 'craig', '"', 'in', 'the', 'episode', '"', 'teddy', "'s", 'story', '"', 'of', 'the', 'television', 'series', 'the', 'long', 'firm', ';', 'he', 'starred', 'alongside', 'actors', 'mark', 'strong', 'and', 'derek', 'jacobi', '.', 'he', 'was', 'cast', 'in', 'the', '2005', 'theatre', 'productions', 'of', 'the', 'philip', 'ridley', 'play', 'mercury', 'fur', ',', 'which', 'was', 'performed', 'at', 'the', 'drum', 'theatre', 'in', 'plymouth', 'and', 'the', '<unk>', '<unk>', 'factory', 'in', 'london', '.', 'he', 'was', 'directed', 'by', 'john', '<unk>', 'and', 'starred', 'alongside', 'ben', '<unk>', ',', 'shane', '<unk>', ',', 'harry', 'kent', ',', 'fraser', '<unk>', ',', 'sophie', 'stanton', 'and', 'dominic', 'hall', '.']
['<START>', '<START>', 'robert', '<unk>', 'is', 'an', 'english', 'film', ',', 'television', 'and', 'theatre', 'actor', '.', 'he', 'had', 'a', 'guest', '@-@', 'starring', 'role', 'on', 'the', 'television', 'series', 'the', 'bill', 'in', '2000', '.', 'this', 'was', 'followed', 'by', 'a', 'starring', 'role', 'in', 'the', 'play', 'herons', 'written', 'by', 'simon', 'stephens', ',', 'which', 'was', 'performed', 'in', '2001', 'at', 'the', 'royal', 'court', 'theatre', '.', 'he', 'had', 'a', 'guest', 'role', 'in', 'the', 'television', 'series', 'judge', 'john', '<unk>', 'in', '2002', '.', 'in', '2004', '<unk>', 'landed', 'a', 'role', 'as', '"', 'craig', '"', 'in', 'the', 'episode', '"', 'teddy', "'s", 'story', '"', 'of', 'the', 'television', 'series', 'the', 'long', 'firm', ';', 'he', 'starred', 'alongside', 'actors', 'mark', 'strong', 'and', 'derek', 'jacobi', '.', 'he', 'was', 'cast', 'in', 'the', '2005', 'theatre', 'productions', 'of', 'the', 'philip', 'ridley', 'play', 'mercury', 'fur', ',', 'which', 'was', 'performed', 'at', 'the', 'drum', 'theatre', 'in', 'plymouth', 'and', 'the', '<unk>', '<unk>', 'factory', 'in', 'london', '.', 'he', 'was', 'directed', 'by', 'john', '<unk>', 'and', 'starred', 'alongside', 'ben', '<unk>', ',', 'shane', '<unk>', ',', 'harry', 'kent', ',', 'fraser', '<unk>', ',', 'sophie', 'stanton', 'and', 'dominic', 'hall', '.']

In [301]:
```
### YOUR CODE HERE
perplexity(2, 0.1, num_words,bi_test_sents, unigram_dict, bigram_dict, trigram_dict)
### END OF YOUR CODE
```

Out[301]: 715.5058643689783

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](https://pytorch.org) 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 [302]:
```
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 /home/ubuntu/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[302]: <torch._C.Generator at 0x7f78a3b0e4b0>

In [303]:

```
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_to
ken=STOP)
train, valid, test = WikiText2.splits(TEXT)
```

In [304]:

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

Vocabulary size: 28905

In [305]:

```
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=B
PTT_LEN,
                                                                  repeat=Fal
se)
```

In [306]:

```
#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 [307]:

```
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_si
ze)
        return:
            normalized_score: torch.FloatTensor of shape (sequence_len, ba
tch_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 No
ne. 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

        embeddings = self.embedding(batch_sents)
        out, hidden = self.rnn(embeddings, hidden)
        output = self.linear(out)
        normalized_score = F.log_softmax(output, dim=-1)

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

In [308]:

```
def train(model, train_iter, valid_iter, vocab_size, criterion, optimizer,
 num_epochs):
    # num_epoch: entire data set is passed forward and backward thru RNN once
    for n in range(num_epochs):
        print(n)
        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 No
ne
            ### YOU CODE HERE (~5 lines)

            #clear gradients before each instance
            optimizer.zero_grad()
            output, _ = model(text)

            # compute loss, grad and update params using optimizer.step()
            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, val_loss))
```

In [309]:

```
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, hidden=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 [310]:
```
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 [311]:
```
train(LM, train_iter, valid_iter, vocab_size, criterion, optimizer, num_ep
ochs)
```

0
Epoch: 1, Training Loss: 6.0688, Validation Loss: 5.1867
1
Epoch: 2, Training Loss: 5.4029, Validation Loss: 4.9612
2
Epoch: 3, Training Loss: 5.1289, Validation Loss: 4.8638
3
Epoch: 4, Training Loss: 4.9559, Validation Loss: 4.8165
4
Epoch: 5, Training Loss: 4.8306, Validation Loss: 4.7866
5
Epoch: 6, Training Loss: 4.7318, Validation Loss: 4.7675
6
Epoch: 7, Training Loss: 4.6504, Validation Loss: 4.7564
7

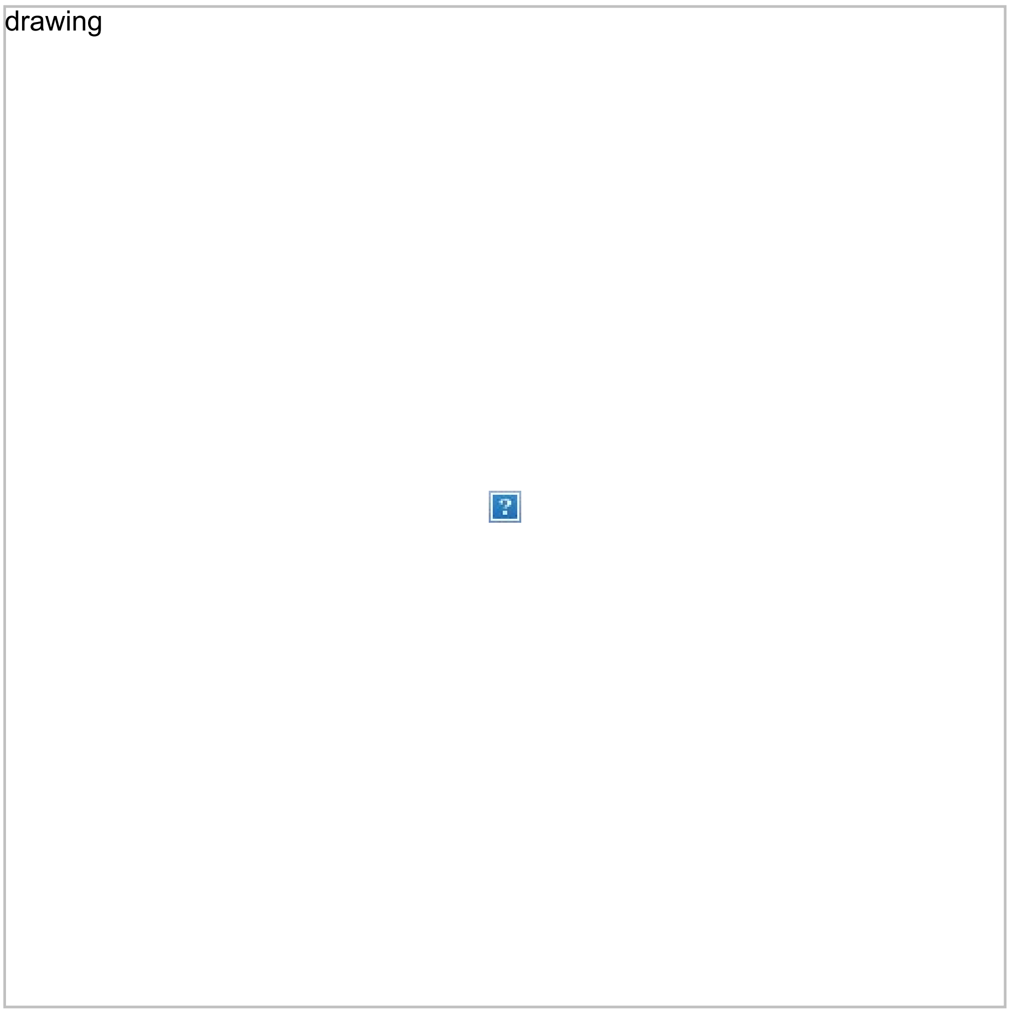
Epoch: 8, Training Loss: 4.5805, Validation Loss: 4.7505
8
Epoch: 9, Training Loss: 4.5198, Validation Loss: 4.7475
9
Epoch: 10, Training Loss: 4.4664, Validation Loss: 4.7483

In [312]:
```
# < 150
test(LM, vocab_size, criterion, test_iter)
```

Out[312]: 99.45588303288754

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`.



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 [313]:
```
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', 'm
ember', 'of',...]
    '''

    model.eval()

    # Defines a vocabulary object that will be used to numericalize a field.
    ID = TEXT.vocab.stoi[start_token]
    strings = [start_token]
    hidden = None

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

    # Returns a tensor filled with the scalar value 1, with the shape defined by the variable argument size, ID.
    leading = torch.ones(1,1)
    leading = leading.long().to(device) * ID

    # iterate through input
    for i in range(max_len):
        outputs, hidden = model(leading, hidden)
        leading = torch.argmax(outputs[-1,:,:], dim=-1)
        inp = leading.cpu().detach().numpy()[0]
        output = TEXT.vocab.itos[inp]
#       print(output)
        if strings[-1] == '<eos>':
            break
        else:
            strings.append(output)


        leading.unsqueeze_(0)

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

In [314]:

```
# 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):
    pass
```

In [315]:

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

Out[315]:

```
['he',
 'was',
 'a',
 '``',
 '<',
 'unk',
```

'>',
'``',
'and',
'``',
'<',
'unk',
'>',
'``',
'.']

In [316]:
```
word_beam_search(LM, 'he', 64, 1)
```

char-level LM

Question 7 [code]

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

In [317]:
```
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 = list(string)
    ### END OF YOUR CODE
    return char_list
```

In [318]:
```
test_str = 'test test test'
char_tokenizer(test_str)
```

Out[318]:
```
['t', 'e', 's', 't', ' ', 't', 'e', 's', 't', ' ', 't', 'e', 's', 't']
```

In [319]:
```
CHAR_TEXT = data.Field(lower=True, tokenize=char_tokenizer ,init_token='<S
TART>', eos_token='<STOP>')
ctrain, cvalid, ctest = WikiText2.splits(CHAR_TEXT)
```

In [320]:
```
CHAR_TEXT.build_vocab(ctrain)
print('Vocabulary size:', len(CHAR_TEXT.vocab))
```

Vocabulary size: 247

In [321]:
```
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, c
valid, ctest),
                                                                batch_size
=BATCH_SIZE,
                                                                bptt_len=B
PTT_LEN,
                                                                repeat=Fal
se)
```

In [322]:
```
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(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)

    def forward(self, batch_sents, hidden=None):
        ### YOUR CODE HERE

        normalized_score = None
        hidden = hidden

        embeddings = self.embedding(batch_sents)
        out, hidden = self.rnn(embeddings, hidden)
        output = self.linear(out)
        normalized_score = F.log_softmax(output, dim=-1)

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

In [323]:
```
def char_train(model, train_iter, valid_iter, criterion, optimizer, vocab_
size, num_epochs):
    # num_epoch: entire data set is passed forward and backward thru RNN once
    for n in range(num_epochs):
```

```
        print(n)
        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

            optimizer.zero_grad()
            outputs, _ = model(text, hidden=None)
            loss = criterion(outputs.view(-1, vocab_size), targets.view(-1
))
            loss.backward()
            optimizer.step()

            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, hidden=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, val_loss))
```

In [324]:
```
def char_test(model, vocab_size, test_iter, criterion):
    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, hidden = model(text, hidden=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
```

```
        ppl = math.exp(test_loss)

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

In [325]:
```
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 [326]:
```
char_train(CLM, ctrain_iter, cvalid_iter, char_criterion, char_optimizer,
char_vocab_size, num_epochs)
```

0
Epoch: 1, Training Loss: 1.8334, Validation Loss: 1.5410
1
Epoch: 2, Training Loss: 1.5417, Validation Loss: 1.4394
2
Epoch: 3, Training Loss: 1.4703, Validation Loss: 1.3956
3
Epoch: 4, Training Loss: 1.4337, Validation Loss: 1.3699
4
Epoch: 5, Training Loss: 1.4101, Validation Loss: 1.3531
5
Epoch: 6, Training Loss: 1.3936, Validation Loss: 1.3410
6
Epoch: 7, Training Loss: 1.3812, Validation Loss: 1.3319
7
Epoch: 8, Training Loss: 1.3714, Validation Loss: 1.3245
8
Epoch: 9, Training Loss: 1.3634, Validation Loss: 1.3183
9
Epoch: 10, Training Loss: 1.3569, Validation Loss: 1.3132

In [327]:
```
#<10
char_test(CLM, char_vocab_size, ctest_iter, char_criterion)
```

Out[327]: 3.683521135213296

In [328]:
```
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

    last_one_hot = torch.ones(1,1)
    last_one_hot = last_one_hot.long().to(device) * ID

    # iterate through input
    for i in range(max_len):
        outputs, hidden = model(leading, hidden)
        leading = torch.argmax(outputs[-1,:,:], dim=-1)
        inp = leading.cpu().detach().numpy()[0]
        output = TEXT.vocab.itos[inp]
#        print(output)
        if strings[-1] == '<eos>':
            break
        else:
            strings.append(output)


        leading.unsqueeze_(0)

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

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.

In [ ]: