mini_project_google_colab

June 15, 2022

##

50.040 Natural Language Processing, Summer 2021

Due 17 June 2021, 5pm

Mini Project

```
[ ]: from google.colab import drive
drive.mount('/content/drive')
```

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

Write your student ID and name

0.0.1 STUDNET ID: 1004335

0.0.2 Name: Dody Senputra

0.0.3 Students with whom you have discussed (if any):

1 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 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.

1.1 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.

1.1.1 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:

$$p(u \mid w, v) = \frac{count(w, v, u)}{count(w, v)}$$

1.1.2 Smoothing the parameters

Add-k Smoothing 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 parameters can be estimated as:

$$p_{add-k}(u) = \frac{count(u) + k}{c + k|V^*|} \tag{1}$$

$$p_{add-k}(u \mid v) = \frac{count(v, u) + k}{count(v) + k|V^*|} \tag{2}$$

$$p_{add-k}(u \mid w, v) = \frac{count(w, v, u) + k}{count(w, v) + k|V^*|} \tag{3}$$

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. #### Interpolation There is another way for smoothing which is named as **interpolation**. In interpolation, we always mix the probability estimates from all the n-gram estimators, weighing and combining the trigram, bigram, and unigram counts. In simple linear interpolation, we combine different order n-grams by linearly interpolating all the models. Thus, we estimate the trigram probability $p(w_n|w_{n-2}, w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a λ:

$$\hat{p}(w_n|w_{n-2}, w_{n-1}) = \lambda_1 p(w_n|w_{n-2}, w_{n-1}) + \lambda_2 p(w_n|w_{n-1}) + \lambda_3 p(w_n) \tag{4}$$

such that the λs sum to 1:

$$\sum_i \lambda_i = 1 \tag{5}$$

In addition, $\lambda_1, \lambda_2, \lambda_3 \geq 0$.

1.1.3 Perplexity

Given a test set D' consisting of sentences $X^{(1)}, X^{(2)}, ..., X^{(|D'|)}$, 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 $X^{(j)}$, and the quality of the language model would be the probability it assigns to the entire set of test sentences, namely:

$$\prod_{j=1}^{|D'|} p(X^{(j)}) \tag{6}$$

Let's define average log_2 probability as:

$$l = \frac{1}{c'} \sum_{j=1}^{|D'|} log_2 p(X^{(j)}) \tag{7}$$

c' is the total number of words in the test set, $|D'|$ is the number of sentences. And the perplexity is defined as:

$$perplexity = 2^{-l} \tag{8}$$

The lower the perplexity, the better the language model.

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

[ ]:
```python
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] != '=']
```

[ ]:
```python
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", '.']
```

1.1.4 Question 1 [code]

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.)
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)

[ ]:
```python
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 = set()
    ngram_dict = {}
    ### YOUR CODE HERE
    for sentence in sents:
        for i in range(len(sentence) - (n - 1)):
            data = tuple(sentence[i : i + n])
            if data in ngram_dict:
                ngram_dict[data] += 1
            else:
                ngram_dict[data] = 1

    ngram_set = list(ngram_dict.keys())
```

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

[ ]:
```
%time unigram_set, unigram_dict = compute_ngram(train_sents, 1)
print('unigram: %d' %(len(unigram_set)),)

%time bigram_set, bigram_dict = compute_ngram(train_sents, 2)
print('bigram: %d' %(len(bigram_set)),)

%time trigram_set, trigram_dict = compute_ngram(train_sents, 3)
print('trigram: %d' %(len(trigram_set)),)
```

```
CPU times: user 1.06 s, sys: 3.26 ms, total: 1.06 s
Wall time: 1.06 s
unigram: 28910
CPU times: user 1.55 s, sys: 34.4 ms, total: 1.59 s
Wall time: 1.59 s
bigram: 577343
CPU times: user 1.38 s, sys: 74.3 ms, total: 1.46 s
Wall time: 1.46 s
trigram: 1344047
```

[ ]:
```
# List 10 most frequent unigrams, bigrams and trigrams as well as their counts.
### YOUR CODE HERE
Counter(unigram_dict).most_common(10)
Counter(bigram_dict).most_common(10)
Counter(trigram_dict).most_common(10)
### END OF YOUR CODE
```

[ ]:
```
[((',', '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)]
```

1.1.5 Question 2 [code]

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". For unigram model, it should be paded as "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 equations in **"Parameter estimation"**. List down the n-grams that have 0 probability.

[ ]:
```
################################################
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']]
```

[ ]:
```
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 = []
    ### YOUR CODE HERE
    for sentence in sents:
        padded_sents.append([START] * np.max([0, n - 1]) + sentence + [STOP])

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

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

[ ]:
```
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)
```

[ ]:
```
len(unigram_set),len(bigram_set),len(trigram_set)
```

[ ]:
```
(28911, 580825, 1363266)
```

[ ]:
```
num_words = sum([v for _,v in unigram_dict.items()])
print(num_words)
```

```
2024702
```

[ ]:
```
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
    if len(ngram) == 1:
        prob = unigram_dic.get(tuple(ngram), 0) / num_words

    elif len(ngram) == 2:
        if unigram_dic.get(tuple([ngram[0]]), 0) == 0:
            return 0
        prob = bigram_dic.get(tuple(ngram), 0) / unigram_dic.
↪get(tuple([ngram[0]]), 0)

    elif len(ngram) == 3:
        if bigram_dic.get(tuple(ngram[:2]), 0) == 0:
            return 0
        prob = trigram_dic.get(tuple(ngram), 0) / bigram_dic.get(tuple(ngram[:
↪2]), 0)
    else:
        prob = "ngram is invalid"
    ### END OF YOUR CODE
    return prob
```

[ ]:
```
print(ngram_prob(ngrams[0], num_words, unigram_dict, bigram_dict, trigram_dict))
print(ngram_prob(ngrams[2], num_words, unigram_dict, bigram_dict, trigram_dict))
print(ngram_prob(ngrams[3], num_words, unigram_dict, bigram_dict, trigram_dict))
```

```
9.960235674499498e-05
0.012683770539060248
0.0867240688826592
```

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

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

1.1.6 Question 3 [code]

1. Implement `add_k_smoothing_ngram` function to estimate ngram probability with `add-k` smoothing technique.
2. Implement `interpolation_ngram` function to estimate ngram probability with `interpolation` smoothing technique.
3. Implement `perplexity` function to compute the perplexity of the corpus "**valid_sents**" according to "**Perplexity**" section. The computation of $p(X^{(j)})$ depends on the n-gram model you choose.

[ ]:
```
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)
```

[ ]:
```
def add_k_smoothing_ngram(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_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:
        s_prob: float --- probability of the "ngram"
    """
    s_prob = None
    V = len(unigram_dic)
    ### YOUR CODE HERE
    if len(ngram) == 1:
        s_prob = (unigram_dic.get(tuple(ngram), 0) + k) / (num_words + k * V)
    elif len(ngram) == 2:
        s_prob = (bigram_dic.get(tuple(ngram), 0) + k) / (
            unigram_dic.get(tuple([ngram[0]]), 0) + k * V
        )
    elif len(ngram) == 3:
        s_prob = (trigram_dic.get(tuple(ngram), 0) + k) / (
            bigram_dic.get(tuple(ngram[:2]), 0) + k * V
        )
    else:
        s_prob = "ngram is invalid"
    ### END OF YOUR CODE
    return s_prob
```

```
[ ]: def interpolation_ngram(ngram, lam, num_words, unigram_dic, bigram_dic,␣
     ↪trigram_dic):
         """
         params:
             ngram: list[str] --- a list that represents n-gram
             lam: list[float] --- a list of length 3.lam[0], lam[1] and lam[2] are␣
     ↪correspondence to trigram, bigram and unigram,repectively.
                                 If len(ngram) == 1, lam[0]=lam[1]=0, lam[2]=1. If␣
     ↪len(ngram) == 2, lam[0]=0. lam[0]+lam[1]+lam[2] = 1.
             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:
             s_prob: float --- probability of the "ngram"
         """
         s_prob = 0
         ### YOUR CODE HERE
         size = len(ngram)
         for i in range(size):
             s_prob += lam[-i - 1] * ngram_prob(
                 ngram[-i - 1 :], num_words, unigram_dic, bigram_dic, trigram_dic
```

```
        )

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

```
[ ]: add_k_prob = add_k_smoothing_ngram(
         ngrams[5], 0.01, num_words, unigram_dict, bigram_dict, trigram_dict
     )
     interpolation_prob = interpolation_ngram(
         ngrams[5], [0.6, 0.3, 0.1], num_words, unigram_dict, bigram_dict,␣
      ↪trigram_dict
     )
     print(ngrams[5])
     print(add_k_prob, interpolation_prob)
```

```
['a', 'number', 'of']
0.5088395909967428 0.7282499450128033
```

```
[ ]: def perplexity(
         n,
         method,
         num_words,
         valid_sents,
         unigram_dic,
         bigram_dic,
         trigram_dic,
         k=0,
         lam=[0, 0, 1],
     ):
         """
          params:
             n: int --- n-gram model you choose
             method: int ---- method == 0, use add_k_smoothing; method != 0, use␣
      ↪interpolation method.
             num_words: int --- total number of words
             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";
             k: float --- The parameter of add_k_smoothing
             lam: list[float] --- a list of length 3. The parameter of␣
      ↪interpolation.
         return:
             ppl: float --- perplexity of valid_sents
```

```
    """
    ppl = None
    ### YOUR CODE HERE
    prob = 0
    ngrams, ngram_dict = compute_ngram(valid_sents, n)
    for ngram in ngrams:
        if method == 0:
            prob += (
                np.log2(
                    add_k_smoothing_ngram(
                        ngram,
                        k,
                        num_words,
                        unigram_dic,
                        bigram_dic,
                        trigram_dic,
                    )
                )
                * ngram_dict[ngram]
            )
        else:
            prob += (
                np.log2(
                    interpolation_ngram(
                        ngram,
                        lam,
                        num_words,
                        unigram_dic,
                        bigram_dic,
                        trigram_dic,
                    )
                )
                * ngram_dict[ngram]
            )
    ppl = 2 ** (prob / num_words * -1)
    ### END OF YOUR CODE
    return ppl
```

[ ]:

```
perplexity(
    1,
    0,
    num_words,
    uni_valid_sents,
    unigram_dict,
    bigram_dict,
    trigram_dict,
    k=0.1,
```

```
    lam=[0, 0, 1.0],
)
```

[ ]: 2.01675089390637

1.1.7 Question 4 [code][written]

1. Based on add-k smoothing method, try out different $k \in [0.0001, 0.001, 0.01, 0.1, 0.5]$ and different n-gram model (unigram, bigram and trigram). Find the model and k that gives the best perplexity on "**valid_sents**" (smaller is better).
2. Based on interpolation method, try out different λ where $\lambda_1 = \lambda_2$ and $\lambda_3 \in [0.1, 0.2, 0.4, 0.6, 0.8]$. Find the λ that gives the best perplexity on "**valid_sents**" (smaller is better).
3. Based on the methods and parameters we provide, choose the method that peforms best on the validation data.

[ ]:
```
n = [1, 2, 3]
k = [0.0001, 0.001, 0.01, 0.1, 0.5]
### YOUR CODE HERE (add-k smoothing method)
valid_sents
aggregated_valid_sents = [uni_valid_sents, bi_valid_sents, tri_valid_sents]
best = dict(ppl=10e10, n=-1, k=-1)
for choosen_n in n:
    for choosen_k in k:
        ppl = perplexity(
            choosen_n,
            0,
            num_words,
            aggregated_valid_sents[choosen_n - 1],
            unigram_dict,
            bigram_dict,
            trigram_dict,
            k=choosen_k,
        )
        if ppl < best["ppl"]:
            best["ppl"] = ppl
            best["n"] = choosen_n
            best["k"] = choosen_k

print(best)
### END OF YOUR CODE
```

{'ppl': 1.9078036007269192, 'n': 2, 'k': 0.001}

[ ]:
```
n = [1, 2, 3]
lambda_3 = [0.1, 0.2, 0.4, 0.6, 0.8]
### YOUR CODE HERE (interpolation method)
best = dict(ppl=10e10, lam=[], n=-1)
```

```python
aggregated_valid_sents = [uni_valid_sents, bi_valid_sents, tri_valid_sents]
for choosen_n in n:
    for lambda_1 in lambda_3:
        for lambda_2 in lambda_3:
            lam = [lambda_1, lambda_1, lambda_2]
            if sum(lam) != 1:
                continue
            ppl = perplexity(
                choosen_n,
                1,
                num_words,
                aggregated_valid_sents[choosen_n - 1],
                unigram_dict,
                bigram_dict,
                trigram_dict,
                lam=lam,
            )
            if ppl < best["ppl"]:
                best["ppl"] = ppl
                best["lam"] = lam
                best["n"] = choosen_n

print(best)
### END OF YOUR CODE
```

```
{'ppl': 1.8281413356300262, 'lam': [0.4, 0.4, 0.2], 'n': 3}
```

Based on the methods and parameters we provide, choose the method that peforms best on the validation data (**write your answer**):

1.1.8 Question 5 [code]

Evaluate the perplexity of the test data **test_sents** based on the best model you choose in **Question 4**.

[ ]:
```python
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)
```

[ ]:
```python
### YOUR CODE HERE
lam = [0.4, 0.4, 0.2]
n = (3,)
num_test_words = sum([len(s) for s in test_sents])
```

```
perplexity(
    3, 1, num_test_words, tri_test_sents, unigram_dict, bigram_dict,␣
 ↪trigram_dict, lam=lam
)
### END OF YOUR CODE
```

[ ]: 137.97200474401626

1.2 Neural Language Model

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.

1.2.1 Question 6 [code]

- Implement the `__init__` function in `LangModel` class. *Note: the code implementation should allow switching between unidirectional LSTM and bidirectional LSTM easily*
- Implement the `forward` function in `LangModel` class.
- Complete the training code in train function and the testing code in test function.
- Train two models - **Unidirectional LSTM** and **Bidirectional LSTM**. Compute the perplexity of the test data "test_iter" using the trained models. The test perplexity of both trained models should be below 150.

Important Note: Make sure that "torchtext <= 0.11", as newer version might have torchtext.legacy removed

[ ]:
```
!pip install torchtext==0.11.2
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting torchtext==0.11.2
  Downloading torchtext-0.11.2-cp37-cp37m-manylinux1_x86_64.whl (8.0 MB)
     |                  | 8.0 MB 4.9 MB/s
Collecting torch==1.10.2
  Downloading torch-1.10.2-cp37-cp37m-manylinux1_x86_64.whl (881.9 MB)
     |                  | 834.1 MB 1.2 MB/s eta
0:00:39tcmalloc: large alloc 1147494400 bytes == 0x38e96000 @  0x7fed9172d615
0x592b76 0x4df71e 0x59afff 0x515655 0x549576 0x593fce 0x548ae9 0x51566f 0x549576
0x593fce 0x548ae9 0x5127f1 0x598e3b 0x511f68 0x598e3b 0x511f68 0x598e3b 0x511f68
0x4bc98a 0x532e76 0x594b72 0x515600 0x549576 0x593fce 0x548ae9 0x5127f1 0x549576
0x593fce 0x5118f8 0x593dd7
     |                  | 881.9 MB 1.8 kB/s
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-
packages (from torchtext==0.11.2) (4.64.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
```

```
packages (from torchtext==0.11.2) (2.23.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.11.2) (1.21.6)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch==1.10.2->torchtext==0.11.2)
(4.2.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.11.2)
(1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.11.2)
(3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchtext==0.11.2) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.11.2)
(2022.5.18.1)
Installing collected packages: torch, torchtext
  Attempting uninstall: torch
    Found existing installation: torch 1.11.0+cu113
    Uninstalling torch-1.11.0+cu113:
      Successfully uninstalled torch-1.11.0+cu113
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.12.0
    Uninstalling torchtext-0.12.0:
      Successfully uninstalled torchtext-0.12.0
ERROR: pip's dependency resolver does not currently take into account all

the packages that are installed. This behaviour is the source of the following

dependency conflicts.

torchvision 0.12.0+cu113 requires torch==1.11.0, but you have torch 1.10.2 which

is incompatible.

torchaudio 0.11.0+cu113 requires torch==1.11.0, but you have torch 1.10.2 which

is incompatible.
Successfully installed torch-1.10.2 torchtext-0.11.2
```

[ ]:
```
import torchtext
import torch
import torch.nn.functional as F
from torchtext.legacy.datasets import WikiText2
from torch import nn, optim
from torchtext.legacy import data
from nltk import word_tokenize
import nltk
import math
```

```
nltk.download('punkt')
torch.manual_seed(222)
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

[ ]: <torch._C.Generator at 0x7fd54e9bc070>

```
[ ]: 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)
```

```
downloading wikitext-2-v1.zip

100%|      | 4.48M/4.48M [00:00<00:00, 22.4MB/s]

extracting
```

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

```
Vocabulary size: 28907
```

```
[ ]: BATCH_SIZE = 64
     # the length of a 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)
```

```
[ ]: #Generate a batch of train data
     batch = next(iter(train_iter))
     text, target = batch.text, batch.target
     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])
```

[ ]:
```
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.bidirectional = lang_config["bidirectional"]

        self.embedding = None
        self.lstm = None
        self.linear = None

        ### TODO:
        ###    1. Initialize 'self.embedding' with nn.Embedding function and 2␣
↪variables we have initialized for you
        ###    2. Initialize 'self.lstm' with nn.LSTM function and 4 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.lstm = nn.LSTM(self.emb_size, self.hidden_size, self.num_layer,␣
↪bidirectional=self.bidirectional)
        self.linear = nn.Linear(self.hidden_size * (1 + self.bidirectional),␣
↪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.lstm. Remember to pass "hidden"␣
↪into self.lstm, even if it is None. But we will
        ###         use "hidden" when implementing greedy search.
        ###      3. Apply linear transformation to the output of self.lstm
        ###      4. Apply 'F.log_softmax' to the output of linear transformation
        ###
        ### YOUR CODE HERE (4 lines)
        embeddings = self.embedding(batch_sents)
        output, hidden = self.lstm(embeddings, hidden)
        output = self.linear(output)
        normalized_score = F.log_softmax(output, dim=2) #dody check dimension␣
↪again
        ### END OF YOUR CODE
        return normalized_score, hidden
```

[ ]:
```
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, val_loss
            )
        )
```

[ ]:
```
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
```

[ ]:
```
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,
    'bidirectional': False
}

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))
```

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

```
Epoch: 1, Training Loss: 6.0577, Validation Loss: 5.1698
Epoch: 2, Training Loss: 5.3880, Validation Loss: 4.9414
Epoch: 3, Training Loss: 5.1200, Validation Loss: 4.8541
Epoch: 4, Training Loss: 4.9522, Validation Loss: 4.8108
Epoch: 5, Training Loss: 4.8313, Validation Loss: 4.7831
Epoch: 6, Training Loss: 4.7345, Validation Loss: 4.7646
Epoch: 7, Training Loss: 4.6525, Validation Loss: 4.7536
Epoch: 8, Training Loss: 4.5823, Validation Loss: 4.7464
Epoch: 9, Training Loss: 4.5210, Validation Loss: 4.7452
Epoch: 10, Training Loss: 4.4664, Validation Loss: 4.7462
```

[ ]:
```
test(LM, vocab_size, criterion, test_iter)
```

[ ]: 99.30294279118066

[ ]:
```
config = {
    'vocab_size':vocab_size,
    'emb_size':128,
    'hidden_size':128,
    'num_layer':1,
    'bidirectional': True
}

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

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

[ ]:
```
train(biLSTM, train_iter, valid_iter, vocab_size, criterion, optimizer,␣
 ↪num_epochs)
```

```
Epoch: 1, Training Loss: 3.1737, Validation Loss: 1.3003
Epoch: 2, Training Loss: 0.9979, Validation Loss: 0.6368
Epoch: 3, Training Loss: 0.5144, Validation Loss: 0.4418
Epoch: 4, Training Loss: 0.3228, Validation Loss: 0.3504
Epoch: 5, Training Loss: 0.2295, Validation Loss: 0.3047
Epoch: 6, Training Loss: 0.1833, Validation Loss: 0.2809
Epoch: 7, Training Loss: 0.1582, Validation Loss: 0.2686
Epoch: 8, Training Loss: 0.1423, Validation Loss: 0.2609
Epoch: 9, Training Loss: 0.1308, Validation Loss: 0.2565
Epoch: 10, Training Loss: 0.1217, Validation Loss: 0.2535
```

[ ]:
```
test(biLSTM, vocab_size, criterion, test_iter)
```

[ ]:
```
1.2745250712413583
```

1.2.2 Question 7 [code][written]

When we use trained language model to generate a sentence given a start token, we can choose `greedy 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`

[ ]:
```
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(strings)
    i = len(strings)
    while i < max_len and ID != end:
        text = torch.LongTensor([[ID]]).to(device)
```

```
        output, hidden = model(text, hidden)
        # to get the maximum output
        ID = torch.argmax(output).item()
        strings.append(TEXT.vocab.itos[ID])
        i += 1
    ### END OF YOUR CODE
    print(strings)
```

[ ]:
```
word_greedy_search(LM, 'he', 64)
```

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

Review Question: Based on your understanding, can we use the Bidirectional LSTM for this language generation (decoding) task? Explain why? write your explanation:

No, eventhough we can force to use BiLSTM for language generation, it may not be as performant as uni-LSTM. Since BiLSTM has both forward LSTM and backward LSTM, only the forward LSTM will hold the useful long term data from the past. The backward LSTM will not have the future data as the future has not been generated yet.

1.2.3 Question 8 [code][written]

- We will use the hidden vectors (the working memory) of LSTM as the contextual embeddings. Implement `contextual_embedding` function.
- Use the `contextual_embedding` function to get the contextual embeddings of the word "sink" in four sequences "wood does not sink in water", "a small water leak will sink the ship", "there are plates in the kitchen sink" and "the kitchen sink was full of dirty dishes". Then calculate the cosine similarity of "sink" from each pair of sequences. Assume that w_1 and w_2 are embeddings of "sink" in sequences "wood does not sink in water" and "a small water leak will sink the ship" respectively. The cosine similarity can be calculated as

$$similarity = cos(\theta) = \frac{w_1^{\mathrm{T}} w_2}{||w_1||_2 ||w_2||_2} \tag{9}$$

Give the explanation of the results.

[ ]:
```
def contextual_embedding(model, sentence):
    """
    params:
        model: nn.Module --- language model
        sentence -- list[str]: list of tokens, e.g., ['I', 'am',...]
    return:
```

```
        embeddings -- numpy array of shape (length of sentence, word embedding␣
↪size)
    """
    model.eval()

    ### YOUR CODE HERE
    seq = []
    for word in sentence.split():
        seq.append([TEXT.vocab.stoi[word]])
        if word == "sink":
            break
    seq = torch.LongTensor(seq).to(device)
    _, (h, __file__) = model(seq)

    embeddings = h[0]
    ### END OF YOUR CODE
    return embeddings
```

[ ]:
```
sink_seq1 = "wood does not sink in water"
sink_seq2 = "a small water leak will sink the ship"
sink_seq3 = "there are plates in the kitchen sink"
sink_seq4 = "the kitchen sink was full of dirty dishes"
### YOUR CODE HERE

ts = []
# ts.append(contextual_embedding(biLSTM, sink_seq1))
# ts.append(contextual_embedding(biLSTM, sink_seq2))
# ts.append(contextual_embedding(biLSTM, sink_seq3))
# ts.append(contextual_embedding(biLSTM, sink_seq4))
ts.append(contextual_embedding(LM, sink_seq1))
ts.append(contextual_embedding(LM, sink_seq2))
ts.append(contextual_embedding(LM, sink_seq3))
ts.append(contextual_embedding(LM, sink_seq4))

cos = torch.nn.CosineSimilarity(dim=1, eps=1e-6)

for i in range(4):
  for j in range(i,4):
    if i != j:
      print(i+1,j+1,cos(ts[i],ts[j]))
### END OF YOUR CODE
```

```
1 2 tensor([0.7897], device='cuda:0', grad_fn=<DivBackward0>)
1 3 tensor([0.5950], device='cuda:0', grad_fn=<DivBackward0>)
1 4 tensor([0.5736], device='cuda:0', grad_fn=<DivBackward0>)
2 3 tensor([0.6297], device='cuda:0', grad_fn=<DivBackward0>)
2 4 tensor([0.5674], device='cuda:0', grad_fn=<DivBackward0>)
```

```
3 4 tensor([0.8567], device='cuda:0', grad_fn=<DivBackward0>)
```

write your explanation:

Review Question: Based on your understanding, can we use the Bidirectional LSTM for this contextual embedding task? Explain why? write your explanation:

Yes, BiLSTM will actually give you the contextual embedding from 2 different perspective, forward and backward. And this will actually tell the context of 'sink' better than LSTM

1.2.4 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.

1.2.5 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.