SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing, Summer 2020

Due 19 June 2020, 5pm

Mini Project

Write your student ID and name

STUDENT ID: 1001781

Name: Krishna Penukonda

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 $x_1, x_2, \ldots, x_m$, 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 (https://blog.einstein.ai/the-wikitext-long-term-dependency-language-modeling-dataset/)](https://blog.einstein.ai/the-wikitext-long-term-dependency-language-modeling-dataset/) 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 $x_{-1} = START, x_0 = START, x_m = STOP$ in this definition, where $START, STOP$ are special symbols referring to the start and the end of a sentence.

In [1]:
```
%%javascript
MathJax.Hub.Config({
  TeX: { equationNumbers: { autoNumber: "AMS" } }
});
```

In [2]:
```
from collections import Counter, namedtuple
import itertools
import numpy as np
```

In [3]:
```
with open('data/wikitext-2/wiki.train.tokens', 'r', encoding='utf8') as f
:
    text = f.readlines()
    train_sents = [line.lower().strip('\n').split() for line in text]
    train_sents = [s for s in train_sents if len(s)>0 and s[0] != '=']
```

In [4]:
```
print(train_sents[1])
```

```
['the', 'game', 'began', 'development', 'in', '2010', ',', 'carrying', 'over', 'a', 'large', '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 [5]:
```
def compute_ngram(sents, n):
    '''
    Compute n-grams that appear in "sents".
    param:
        sents: list[list[str]] --- list of list of word strings
        n: int --- "n" gram
    return:
        ngram_set: set{str} --- a set of n-grams (no duplicate elements)
        ngram_dict: dict{ngram: counts} --- a dictionary that maps each ngram to its number occurence in "sents";
        This dict contains the parameters of our ngram model. E.g. if n=2, ngram_dict={('a','b'):10, ('b','c'):13}

        You may need to use "Counter", "tuple" function here.
    '''
    ngram_set = None
    ngram_dict = None
    ### YOUR CODE HERE
    ngrams = [[tuple(sent[i: i + n]) for i in range(len(sent) - n + 1)] for sent in sents]
    ngrams = list(itertools.chain.from_iterable(ngrams))
    ngram_dict = Counter(ngrams)
    ngram_set = set(ngram_dict.keys())
    ### END OF YOUR CODE
    return ngram_set, ngram_dict
```

In [6]:
```
### ~28xxx
unigram_set, unigram_dict = compute_ngram(train_sents, 1)
print(len(unigram_set))
```

```
28910
```

In [7]:
```
### ~57xxxx
bigram_set, bigram_dict = compute_ngram(train_sents, 2)
print(len(bigram_set))
```

```
577343
```

In [8]:
```
### ~134xxxx
trigram_set, trigram_dict = compute_ngram(train_sents, 3)
print(len(trigram_set))
```

```
1344047
```

In [9]:
```
# List 10 most frequent unigrams, bigrams and trigrams as well as their c
ounts.
print("Most frequent N-grams")
print("=" * 25)
print("Unigrams:")
print(unigram_dict.most_common(10))
print("-" * 50)
print("Bigrams:")
print(bigram_dict.most_common(10))
print("-" * 50)
print("Trigrams:")
print(trigram_dict.most_common(10))
```

```
Most frequent N-grams
=========================
Unigrams:
[(('the',), 130519), ((',',), 99763), (('.',), 73388), (('of',), 56743),
(('<unk>',), 53951), (('and',), 49940), (('in',), 44876), (('to',), 3946
2), (('a',), 36140), (('"',), 28285)]
--------------------------------------------------
Bigrams:
[(('of', 'the'), 17242), (('in', 'the'), 11778), ((',', 'and'), 11643),
(('.', 'the'), 11274), ((',', 'the'), 8024), (('<unk>', ','), 7698), (('t
o', 'the'), 6009), (('on', 'the'), 4495), (('the', '<unk>'), 4389), (('an
d', 'the'), 4331)]
--------------------------------------------------
Trigrams:
[((',', 'and', 'the'), 1393), ((',', '<unk>', ','), 950), (('<unk>', ',',
'<unk>'), 901), (('one', 'of', 'the'), 866), (('<unk>', ',', 'and'), 81
9), (('.', 'however', ','), 775), (('<unk>', '<unk>', ','), 745), (('.',
'in', 'the'), 726), (('.', 'it', 'was'), 698), (('the', 'united', 'state
s'), 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 [10]:
```python
##############################################
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'], ['ca
n', 'sea'], ['a', 'number', 'of'], ['with', 'respect', 'to'], ['in', 'ter
ms', 'of'], ['not', 'good', 'bad'], ['first', 'start', 'with']]
```

In [11]:
```python
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 = [list(itertools.chain([START] * (n - 1), sent, [STOP]
if n > 1 else [])) for sent in sents]
    ### END OF YOUR CODE
    return padded_sents
```

In [12]:
```python
uni_sents = pad_sents(train_sents, 1)
bi_sents = pad_sents(train_sents, 2)
tri_sents = pad_sents(train_sents, 3)
```

In [13]:
```python
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 [14]:
```python
### (28xxx, 58xxxx, 136xxxx)
len(unigram_set),len(bigram_set),len(trigram_set)
```

Out[14]:
```
(28910, 580825, 1363266)
```

In [15]:
```python
### ~ 200xxxx; total number of words in wikitext-2.train
num_words = sum([v for _,v in unigram_dict.items()])
print(num_words)
```

```
2007146
```

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 [16]:
```
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_dicts = [unigram_dic, bigram_dic, trigram_dic]
    ngram = tuple(ngram)
    n = len(ngram)
    numerator = ngram_dicts[n - 1].get(ngram, 0)
    denominator = ngram_dicts[n - 2].get(ngram[:n - 1], 0) if n > 1 else
num_words
    prob = numerator / denominator
    ### END OF YOUR CODE
    return prob
```

In [17]:
```
### ~9.96e-05
ngram_prob(ngrams[0], num_words,unigram_dict, bigram_dict, trigram_dict)
```

Out[17]:
```
9.960235674499498e-05
```

In [18]:
```
### List down the n-grams that have 0 probability.
from pprint import pprint
zero_prob_ngrams = [ngram for ngram in ngrams if ngram_prob(ngram, num_wo
rds, unigram_dict, bigram_dict, trigram_dict) == 0]
print("Zero-probability n-grams:")
pprint(zero_prob_ngrams)
```

```
Zero-probability n-grams:
[['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 [19]:
```
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)
```

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:

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

where $k \in (0, 1)$ is the parameter of this approach, and $|V^*|$ is the size of the vocabulary V^*,here $V^* = V \cup STOP$. 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.

In [20]:
```
def smooth_ngram_prob(ngram, k, num_words, unigram_dic, bigram_dic, trigr
am_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 = 0
    V = len(unigram_dic) + 1
    ### YOUR CODE HERE\、
    ngram_dicts = [unigram_dic, bigram_dic, trigram_dic]
    ngram = tuple(ngram)
    n = len(ngram)
    numerator = ngram_dicts[n - 1].get(ngram, 0)
    denominator = ngram_dicts[n - 2].get(ngram[:n - 1], 0) if n > 1 else
num_words
    s_prob = (numerator + k) / (denominator + k * V)
    ### END OF YOUR CODE
    return s_prob
```

In [21]:
```
### ~ 9.31e-05
smooth_ngram_prob(ngrams[0], 0.5, num_words, unigram_dict, bigram_dict, t
rigram_dict)
```

Out[21]: 9.311982452086402e-05

Perplexity

Given a test set D' consisting of sentences $X^{(1)}, X^{(2)}, \ldots, X^{(|D'|)}$, each sentence $X^{(j)}$ consists of words $x_1^{(j)}, x_2^{(j)}, \ldots, x_{n_j}^{(j)}$, 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:

$$\prod_j^{D'} p(X^{(j)})$$

Let's define average log2 probability as:

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

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 [22]:
```python
from math import log, exp

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 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
    log_prob = sum(
        log(smooth_ngram_prob(sentence[i: i + n], k, num_words, unigram_dic, bigram_dic, trigram_dic))  # Log-probability of the n-gram
        for sentence in valid_sents  # Loop over sentences
        for i in range(len(sentence) - n + 1)  # Loop over n-grams in each sentence
    )
    ppl = exp(-log_prob / sum(len(s) for s in valid_sents))
    ### END OF YOUR CODE
    return ppl
```

In [23]:
```python
### ~ 840
perplexity(1, 0.1, num_words, uni_valid_sents, unigram_dict, bigram_dict, trigram_dict)
```

Out[23]: 840.7347306260672

In [24]:
```
n = [1,2,3]
k = [0.1, 0.3, 0.5, 0.7, 0.9]
### YOUR CODE HERE
best_perp = float("inf")
best_params = None
n_valid_sents = [uni_valid_sents, bi_valid_sents, tri_valid_sents]
for n_choice, k_choice in itertools.product(n, k):
    perp = perplexity(n_choice, k_choice, num_words, n_valid_sents[n_choi
ce - 1], unigram_dict, bigram_dict, trigram_dict)
    print(f"(n, k)={n_choice, k_choice} | Perplexity: {perp}")
    if perp < best_perp:
        best_perp = perp
        best_params = n_choice, k_choice
print(f"==============================\nBest params: (n, k) = {best_param
s}")
### END OF YOUR CODE
```

```
(n, k)=(1, 0.1) | Perplexity: 840.7347306260672
(n, k)=(1, 0.3) | Perplexity: 841.1427277007833
(n, k)=(1, 0.5) | Perplexity: 841.5959678977988
(n, k)=(1, 0.7) | Perplexity: 842.0904494791431
(n, k)=(1, 0.9) | Perplexity: 842.6227084910905
(n, k)=(2, 0.1) | Perplexity: 739.5817358287826
(n, k)=(2, 0.3) | Perplexity: 1061.3982617381348
(n, k)=(2, 0.5) | Perplexity: 1289.149126033837
(n, k)=(2, 0.7) | Perplexity: 1477.1909399569556
(n, k)=(2, 0.9) | Perplexity: 1641.5907324604584
(n, k)=(3, 0.1) | Perplexity: 4773.6491283133
(n, k)=(3, 0.3) | Perplexity: 6676.617325724715
(n, k)=(3, 0.5) | Perplexity: 7831.228458055187
(n, k)=(3, 0.7) | Perplexity: 8684.056079368764
(n, k)=(3, 0.9) | Perplexity: 9364.604903329062
==============================
Best params: (n, k) = (2, 0.1)
```

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).

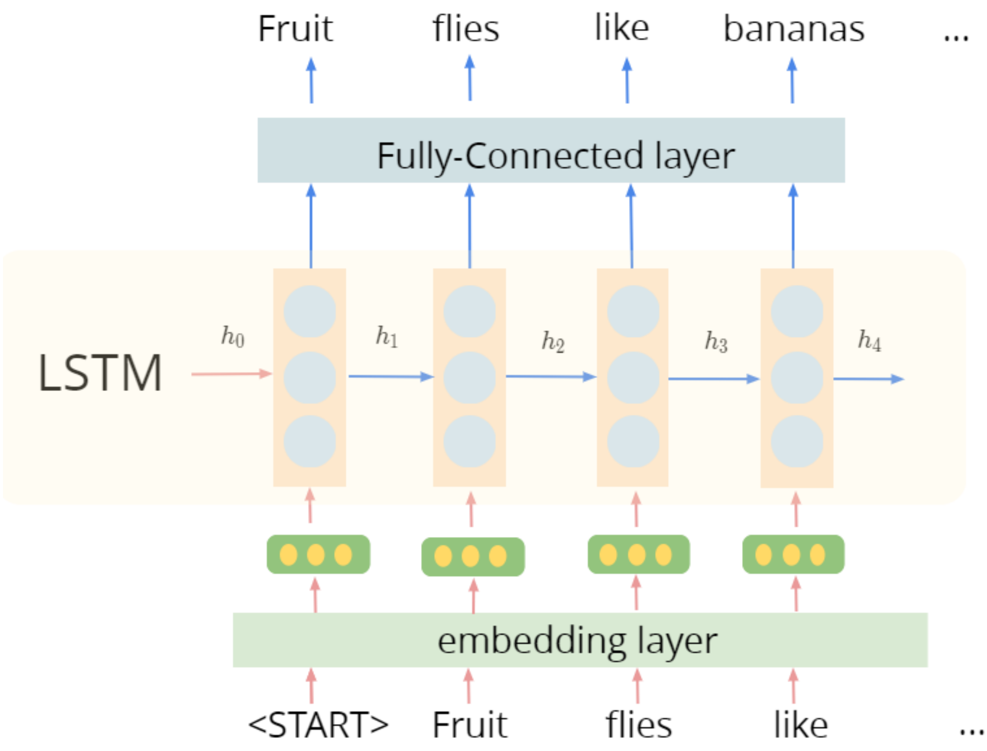
In [25]:
```
with open('data/wikitext-2/wiki.test.tokens', 'r', encoding='utf8') as f:
    text = f.readlines()
    test_sents = [line.lower().strip('\n').split() for line in text]
    test_sents = [s for s in test_sents if len(s)>0 and s[0] != '=']

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

In [26]:
```
### YOUR CODE HERE
perplexity(2, 0.1, num_words,bi_test_sents, unigram_dict, bigram_dict, tr
igram_dict)
### END OF YOUR CODE
```

Out[26]:
```
689.392959094 4014
```

Neural Language Model (RNN)



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/tutorials/)](https://pytorch.org/tutorials/) and [torchtext (https://torchtext.readthedocs.io/en/latest/index.html#)](https://torchtext.readthedocs.io/en/latest/index.html#) 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 [27]:
```
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/krishna/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[27]:
```
<torch._C.Generator at 0x7fd2de4b2dd0>
```

In [28]:

```
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_t
oken=STOP)
train, valid, test = WikiText2.splits(TEXT)
```

In [29]:

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

```
Vocabulary size: 28906
```

In [30]:

```
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, vali
d, test),
                                                                  batch_siz
e=BATCH_SIZE,
                                                                  bptt_len=
BPTT_LEN,
                                                                  repeat=Fa
lse)
```

In [31]:

```
#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 [32]:

```python
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 varia
bles 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.RNN(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_s
ize)
        return:
            normalized_score: torch.FloatTensor of shape (sequence_len, b
atch_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 "hi
dden" 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 transfo
rmation
        ###
        ### YOUR CODE HERE
        embed = self.embedding(batch_sents)
        out, hidden = self.rnn(embed, hidden)
        normalized_score = F.log_softmax(self.linear(out))
        ### END OF YOUR CODE
        return normalized_score, hidden
```

In [33]:
```
from tqdm.auto import tqdm

def train(model, train_iter, valid_iter, vocab_size, criterion, optimizer
, num_epochs):
    for n in range(num_epochs):
        print(f"Epoch {n + 1} / {num_epochs}")
        train_loss = 0
        target_num = 0
        model.train()
        for batch in tqdm(train_iter):
            text, targets = batch.text.to(device), batch.target.to(device
)
            loss = None

            ### we don't consider "hidden" here. So according to the defa
ult setting, "hidden" will be None
            ### YOU CODE HERE (~5 lines)
            model.zero_grad()
            prediction, _ = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.vie
w(-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.vie
w(-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 [39]:
```
from math import exp

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.vie
w(-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 = exp(test_loss)
        ### END OF YOUR CODE
        return ppl
```

In [35]:
```
num_epochs=10
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device = "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 [36]:
```
train(LM, train_iter, valid_iter, vocab_size, criterion, optimizer, num_e
pochs)
```

```
Epoch 1 / 10

/home/krishna/miniconda3/envs/ai/lib/python3.7/site-packages/ipykernel_la
uncher.py:45: UserWarning: Implicit dimension choice for log_softmax has
been deprecated. Change the call to include dim=X as an argument.

Epoch: 1, Training Loss: 2.6148, Validation Loss: 2.3341
Epoch 2 / 10

Epoch: 2, Training Loss: 2.2647, Validation Loss: 2.2283
Epoch 3 / 10

Epoch: 3, Training Loss: 2.1226, Validation Loss: 2.1889
Epoch 4 / 10

Epoch: 4, Training Loss: 2.0304, Validation Loss: 2.1741
Epoch 5 / 10

Epoch: 5, Training Loss: 1.9619, Validation Loss: 2.1687
Epoch 6 / 10

Epoch: 6, Training Loss: 1.9074, Validation Loss: 2.1722
Epoch 7 / 10

Epoch: 7, Training Loss: 1.8622, Validation Loss: 2.1782
Epoch 8 / 10

Epoch: 8, Training Loss: 1.8238, Validation Loss: 2.1885
Epoch 9 / 10

Epoch: 9, Training Loss: 1.7897, Validation Loss: 2.2011
Epoch 10 / 10

Epoch: 10, Training Loss: 1.7605, Validation Loss: 2.2150
```

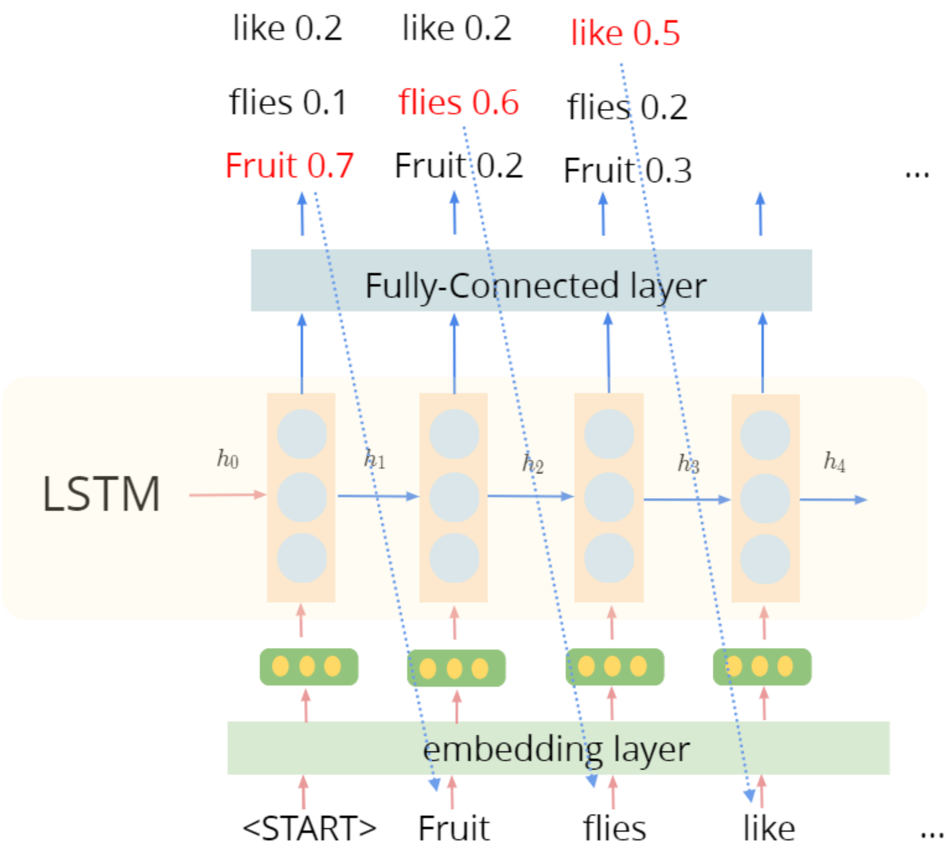
In [40]:
```
# < 150
test(LM, vocab_size, criterion, test_iter)
```

```
/home/krishna/miniconda3/envs/ai/lib/python3.7/site-packages/ipykernel_la
uncher.py:45: UserWarning: Implicit dimension choice for log_softmax has
been deprecated. Change the call to include dim=X as an argument.
```

Out[40]:
```
8.902204649954376
```

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 [ ]:
```
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 OF YOUR CODE
    return strings
```

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

In [ ]:
```
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 [47]:
```
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 = list(string)
    ### END OF YOUR CODE
    return char_list
```

In [44]:
```
test_str = 'test test test'
char_tokenizer(test_str)
```

Out[44]:
```
['t', 'e', 's', 't', ' ', 't', 'e', 's', 't', ' ', 't', 'e', 's', 't']
```

In [45]:
```
CHAR_TEXT = data.Field(lower=True, tokenize=char_tokenizer ,init_token='<
START>', eos_token='<STOP>')
ctrain, cvalid, ctest = WikiText2.splits(CHAR_TEXT)
```

In [46]:
```
CHAR_TEXT.build_vocab(ctrain)
print('Vocabulary size:', len(CHAR_TEXT.vocab))
```

```
Vocabulary size: 247
```

In [48]:
```
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_siz
e=BATCH_SIZE,
                                                                bptt_len=
BPTT_LEN,
                                                                repeat=Fa
lse)
```

In [56]:
```
class CharLangModel(nn.Module):
    def __init__(self, lang_config):
        ### YOUR CODE HERE
        super().__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.RNN(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
        embed = self.embedding(batch_sents)
        out, hidden = self.rnn(embed, hidden)
        normalized_score = F.log_softmax(self.linear(out))
        return normalized_score, hidden
```

In [50]:
```python
def char_train(model, train_iter, valid_iter, criterion, optimizer, vocab
_size, num_epochs):
    ### YOUR CODE HERE
    for n in range(num_epochs):
        print(f"Epoch {n + 1} / {num_epochs}")
        train_loss = 0
        target_num = 0
        model.train()
        for batch in tqdm(train_iter):
            text, targets = batch.text.to(device), batch.target.to(device
)
            loss = None

            ### we don't consider "hidden" here. So according to the defa
ult setting, "hidden" will be None
            ### YOU CODE HERE (~5 lines)
            model.zero_grad()
            prediction, _ = model(text)
            loss = criterion(prediction.view(-1, vocab_size), targets.vie
w(-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.vie
w(-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 [53]:
```
def char_test(model, vocab_size, test_iter, criterion):
    ### YOUR CODE HERE
    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.vie
w(-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 = exp(test_loss)
        ### END OF YOUR CODE
        return ppl
```

In [57]:
```
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 [58]:
```
char_train(CLM, ctrain_iter, cvalid_iter, char_criterion, char_optimizer,
char_vocab_size, num_epochs)
```

```
Epoch 1 / 10

/home/krishna/miniconda3/envs/ai/lib/python3.7/site-packages/ipykernel_la
uncher.py:17: UserWarning: Implicit dimension choice for log_softmax has
been deprecated. Change the call to include dim=X as an argument.

Epoch: 1, Training Loss: 3.5908, Validation Loss: 3.3970
Epoch 2 / 10

Epoch: 2, Training Loss: 3.4502, Validation Loss: 3.3523
Epoch 3 / 10

Epoch: 3, Training Loss: 3.4204, Validation Loss: 3.3330
Epoch 4 / 10

Epoch: 4, Training Loss: 3.4059, Validation Loss: 3.3232
Epoch 5 / 10

Epoch: 5, Training Loss: 3.3968, Validation Loss: 3.3162
Epoch 6 / 10

Epoch: 6, Training Loss: 3.3904, Validation Loss: 3.3105
Epoch 7 / 10

Epoch: 7, Training Loss: 3.3856, Validation Loss: 3.3060
Epoch 8 / 10

Epoch: 8, Training Loss: 3.3821, Validation Loss: 3.3026
Epoch 9 / 10

Epoch: 9, Training Loss: 3.3792, Validation Loss: 3.3003
Epoch 10 / 10

Epoch: 10, Training Loss: 3.3769, Validation Loss: 3.2981
```

In [59]:
```
# <10
char_test(CLM, char_vocab_size, ctest_iter, char_criterion)
```

```
/home/krishna/miniconda3/envs/ai/lib/python3.7/site-packages/ipykernel_la
uncher.py:17: UserWarning: Implicit dimension choice for log_softmax has
been deprecated. Change the call to include dim=X as an argument.
```

Out[59]:
```
26.839968565005446
```

```
In [ ]: 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

            ### 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 (https://colab.research.google.com/notebooks/welcome.ipynb)](https://colab.research.google.com/notebooks/welcome.ipynb), 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 (https://colab.research.google.com/notebooks/io.ipynb)](https://colab.research.google.com/notebooks/io.ipynb).

In addition, Microsoft also provides the online platform [Azure Notebooks (https://notebooks.azure.com/help/introduction)](https://notebooks.azure.com/help/introduction) for research of data science and machine learning, there are free trials for new users with credits.