

```
1 from google.colab import files
2 files.upload()
```

```
1 !unzip -qn Mini\ Project.zip
```

```
1 !cd data && wget -q -nc https://s3.amazonaws.com/research.metamind.io/wikitext/wiki
2 !cd data && unzip -qn wikitext-2-v1.zip
```



▼ 50.040 Natural Language Processing, Summer 2020

****Due 19 June 2020, 5pm****

Mini Project

Write your student ID and name

STUDNET ID: 1002675

Name: Chia Yew Ken

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, \dots, 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:

$$p(x_1, x_2, \dots, x_m)$$

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:

$$p(x_1, x_2, \dots, x_m) = \prod_{i=1}^m p(x_i)$$

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:

$$p(x_0, x_1, x_2, \dots, x_m) = \prod_{i=1}^m p(x_i | x_{i-1})$$

Under the second-order Markovian assumption, the joint probability can be written as:

$$p(x_{-1}, x_0, x_1, x_2, \dots, x_m) = \prod_{i=1}^m p(x_i | x_{i-2}, x_{i-1})$$

Similar to what we did in HMM, we will assume that

$x_{-1} = START, x_0 = START, x_m = STOP$ in this definition, where $START, STOP$ are special symbols referring to the start and the end of a sentence.

▼ Parameter estimation

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

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

- In the unigram model, the parameters can be estimated as:

$$p(u) = \frac{count(u)}{c}$$

, where c is the total number of words in the corpus.

- In the bigram model, the parameters can be estimated as:

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

- In the trigram model, the parameters can be estimated as:

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

```
1 %javascript
2 MathJax.Hub.Config({
3   TeX: { equationNumbers: { autoNumber: "AMS" } }
4 });
```



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:

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

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

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

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.

▼ Perplexity

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

```
1 from collections import Counter, namedtuple
2 import itertools
3 import numpy as np
```

```
1 with open('data/wikitext-2/wiki.train.tokens', 'r', encoding='utf8') as f:
```

```

2     text = f.readlines()
3     train_sents = [line.lower().strip('\n').split() for line in text]
4     train_sents = [s for s in train_sents if len(s)>0 and s[0] != '=']

1 print(train_sents[1])

☞ ['the', 'game', 'began', 'development', 'in', '2010', ',', 'carrying', 'over', '

```

▼ 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)

```

1 def compute_ngram(sents, n):
2     '''
3     Compute n-grams that appear in "sents".
4     param:
5         sents: list[list[str]] --- list of list of word strings
6         n: int --- "n" gram
7     return:
8         ngram_set: set[str] --- a set of n-grams (no duplicate elements)
9         ngram_dict: dict{ngram: counts} --- a dictionary that maps each ngram to it
10        This dict contains the parameters of our ngram model. E.g. if n=2, ngram_di
11
12        You may need to use "Counter", "tuple" function here.
13    '''
14    ngram_set = None
15    ngram_dict = None
16    ### YOUR CODE HERE
17    ngrams = []
18    for tokens in sents:
19        for i in range(0, len(tokens) - n + 1):
20            window = tokens[i:i+n]
21            ngrams.append(tuple(window))
22
23    from collections import Counter
24    counter = Counter(ngrams)
25    ngram_set = set(counter.keys())
26    ngram_dict = counter
27
28    ### END OF YOUR CODE
29    return ngram_set, ngram_dict

```

```
2 unigram_set, unigram_dict = compute_ngram(train_sents, 1)
3 print(len(unigram_set))
```

☞ 28910

```
1 ### ~57xxxx
2 bigram_set, bigram_dict = compute_ngram(train_sents, 2)
3 print(len(bigram_set))
```

☞ 577343

```
1 ### ~134xxxx
2 trigram_set, trigram_dict = compute_ngram(train_sents, 3)
3 print(len(trigram_set))
```

☞ 1344047

```
1 # List 10 most frequent unigrams, bigrams and trigrams as well as their counts.
2 for d in [unigram_dict, bigram_dict, trigram_dict]:
3     print(d.most_common(10))
4     print()

--NORMAL--
```

☞ [((('the',), 130519), ((',',), 99763), (('.',), 73388), (('of',), 56743), (('<unk',), 17242), (('in', 'the'), 11778), ((',', 'and'), 11643), (('.', 'the'), 1393), ((',', '<unk>', ','), 950), (('<unk>', ',', '<unk>'), 1393), ((',', '<unk>', 'the'), 1393), ((',', '<unk>', ','), 950), (('<unk>', ',', '<unk>'), 1393)]

▼ 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.

```

1 #####
2 ngrams = list()
3 with open(r'data/ngram.txt','r') as f:
4     for line in f:
5         ngrams.append(line.strip('\n').split())
6 print(ngrams)
7 #####

```

☞ [['the', 'computer'], ['go', 'to'], ['have', 'had'], ['and', 'the'], ['can', 'se

```

1 START = '<START>'
2 STOP = '</STOP>'
3 #####
4 def pad_sents(sents, n):
5     '''
6     Pad the sents according to n.
7     params:
8         sents: list[list[str]] --- list of sentences.
9         n: int --- specify the padding type, 1-gram, 2-gram, or 3-gram.
10    return:
11        padded_sents: list[list[str]] --- list of padded sentences.
12    '''
13    padded_sents = None
14    ### YOUR CODE HERE
15    front = [START] * (n - 1)
16    back = [STOP]
17    padded_sents = [front + tokens + back for tokens in sents]
18    ### END OF YOUR CODE
19    return padded_sents

```

```

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

```

```

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

```

```

1 ### (28xxx, 58xxxx, 136xxxx)
2 len(unigram_set),len(bigram_set),len(trigram_set)

```

☞ (28911, 580825, 1363266)

```

1 ### ~ 200xxxx; total number of words in wikitext-2.train
2 num_words = sum([v for _,v in unigram_dict.items()])
3 print(num_words)

```

☞ 2024702

```
1 def ngram_prob(ngram, num_words, unigram_dic, bigram_dic, trigram_dic):
2     '''
3     params:
4         ngram: list[str] --- a list that represents n-gram
5         num_words: int --- total number of words
6         unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram to i
7         bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram to i
8         trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram to i
9     return:
10         prob: float --- probability of the "ngram"
11     '''
12     prob = None
13     ### YOUR CODE HERE
14     tup = tuple(ngram)
15     n = len(tup)
16     fn = {
17         1: lambda x: unigram_dic[x] / num_words,
18         2: lambda x: bigram_dic[x] / unigram_dic[x[:-1]],
19         3: lambda x: trigram_dic[x] / bigram_dic[x[:-1]],
20     }[n]
21     prob = fn(tup)
22     ### END OF YOUR CODE
23     return prob
```

```
1 ### ~9.96e-05
2 ngram_prob(ngrams[0], num_words, unigram_dict, bigram_dict, trigram_dict)
```

☞ 9.960235674499498e-05

```
1 ### List down the n-grams that have 0 probability.
2 print([
3     ng for ng in ngrams
4     if ngram_prob(ng, num_words, unigram_dict, bigram_dict, trigram_dict) == 0
5 ])
```

☞ [['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).

```
1 with open('data/wikitext-2/wiki.valid.tokens', 'r', encoding='utf8') as f:
2     text = f.readlines()
3     valid_sents = [line.lower().strip('\n').split() for line in text]
4     valid_sents = [s for s in valid_sents if len(s)>0 and s[0] != '=']
5
6 uni_valid_sents = pad_sents(valid_sents, 1)
7 bi_valid_sents = pad_sents(valid_sents, 2)
8 tri_valid_sents = pad_sents(valid_sents, 3)
```

```
1 def smooth_ngram_prob(ngram, k, num_words, unigram_dic, bigram_dic, trigram_dic):
2     '''
3     params:
4         ngram: list[str] --- a list that represents n-gram
5         k: float
6         num_words: int --- total number of words
7         unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram to
8         bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram to
9         trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram to
10    return:
11        s_prob: float --- probability of the "ngram"
12    '''
13    s_prob = 0
14    V = len(unigram_dic) + 1
15    ### YOUR CODE HERE\
16    tup = tuple(ngram)
17    n = len(tup)
18    fn = {
19        1: lambda x: (unigram_dic[x] + k) / (num_words + k*V),
20        2: lambda x: (bigram_dic[x] + k) / (unigram_dic[x[:-1]] + k*V),
21        3: lambda x: (trigram_dic[x] + k) / (bigram_dic[x[:-1]] + k*V),
22    }[n]
23    s_prob = fn(tup)
24    ### END OF YOUR CODE
25    return s_prob
```

```
1 ### ~ 9.31e-05
```

```
2 smooth_ngram_prob(ngrams[0], 0.5, num_words, unigram_dict, bigram_dict, trigram_dic
```

```
☞ 9.311950336264874e-05
```

```
1 def perplexity(n, k, num words, valid_sents, unigram dic, bigram dic, trigram dic):
```



```

2     '''
3     compute the perplexity of valid_sents
4     params:
5         n: int --- n-gram model you choose.
6         k: float --- smoothing parameter.
7         num_words: int --- total number of words in the traning set.
8         valid_sents: list[list[str]] --- list of sentences.
9         unigram_dic: dict{ngram: counts} --- a dictionary that maps each 1-gram to
10        bigram_dic: dict{ngram: counts} --- a dictionary that maps each 2-gram to i
11        trigram_dic: dict{ngram: counts} --- a dictionary that maps each 3-gram to
12    return:
13        ppl: float --- perplexity of valid_sents
14    '''
15    ppl = None
16    ### YOUR CODE HERE
17    ngrams = []
18    for tokens in valid_sents:
19        for i in range(0, len(tokens) - n + 1):
20            window = tokens[i:i+n]
21            ngrams.append(tuple(window))
22
23    losses = [
24        smooth_ngram_prob(ng, k, num_words, unigram_dic, bigram_dic, trigram_dic)
25        # ngram_prob(ng, num_words, unigram_dic, bigram_dic, trigram_dic)
26        for ng in ngrams
27    ]
28    # loss = np.sum(np.log(losses)) / num_words
29    loss = np.mean(np.log(losses))
30    ppl = np.exp(-loss)
31    ### END OF YOUR CODE
32    return ppl

```

1 ### ~ 840

2 perplexity(1, 0.1, num_words, uni_valid_sents, unigram_dict, bigram_dict, trigram_d

☞ 833.464705004929

```

1 n = [1,2,3]
2 k = [0.1, 0.3, 0.5, 0.7, 0.9]
3 ### YOUR CODE HERE
4 ppl_best = 1e9
5 n_best = None
6 k_best = None
7 for n_value in n:
8     for k_value in k:
9         ppl = perplexity(n_value, k_value, num_words, uni_valid_sents, unigram_dict
10        print(dict(n_value=n_value, k_value=k_value, ppl=ppl))
11        if ppl < ppl_best:
12            ppl_best, n_best, k_best = ppl, n_value, k_value
13
14 print(dict(n_best=n_best, k_best=k_best, ppl_best=ppl_best))

```

15 ### END OF YOUR CODE

```
➞ {'n_value': 1, 'k_value': 0.1, 'ppl': 833.464705004929}
   {'n_value': 1, 'k_value': 0.3, 'ppl': 833.865817488325}
   {'n_value': 1, 'k_value': 0.5, 'ppl': 834.3114466705376}
   {'n_value': 1, 'k_value': 0.7, 'ppl': 834.7976589099709}
   {'n_value': 1, 'k_value': 0.9, 'ppl': 835.3210490687667}
   {'n_value': 2, 'k_value': 0.1, 'ppl': 804.6918648337755}
   {'n_value': 2, 'k_value': 0.3, 'ppl': 1152.8672381011695}
   {'n_value': 2, 'k_value': 0.5, 'ppl': 1399.8530753304933}
   {'n_value': 2, 'k_value': 0.7, 'ppl': 1604.0183827086553}
   {'n_value': 2, 'k_value': 0.9, 'ppl': 1782.6628376645583}
   {'n_value': 3, 'k_value': 0.1, 'ppl': 5805.930887540093}
   {'n_value': 3, 'k_value': 0.3, 'ppl': 8124.104705953042}
   {'n_value': 3, 'k_value': 0.5, 'ppl': 9530.813230188001}
   {'n_value': 3, 'k_value': 0.7, 'ppl': 10569.446528522254}
   {'n_value': 3, 'k_value': 0.9, 'ppl': 11397.840234405312}
   {'n_best': 2, 'k_best': 0.1, 'ppl_best': 804.6918648337755}
```

▼ 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).

```
1 with open('data/wikitext-2/wiki.test.tokens', 'r', encoding='utf8') as f:
2     text = f.readlines()
3     test_sents = [line.lower().strip('\n').split() for line in text]
4     test_sents = [s for s in test_sents if len(s)>0 and s[0] != '=']
5
6 uni_test_sents = pad_sents(test_sents, 1)
7 bi_test_sents = pad_sents(test_sents, 2)
8 tri_test_sents = pad_sents(test_sents, 3)

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

```
➞ 731.992244548119
```

▼ Neural Language Model (RNN)



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

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

▼ Question 5 [code]

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

```
1 import torchtext
2 import torch
3 import torch.nn.functional as F
4 from torchtext.datasets import WikiText2
5 from torch import nn, optim
6 from torchtext import data
7 from nltk import word_tokenize
8 import nltk
9 nltk.download('punkt')
10 torch.manual_seed(222)
```

```
↳ [nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
<torch._C.Generator at 0x7f7fab7c7570>
```

```
1 def tokenizer(text):
2     '''Tokenize a string to words'''
3     return word_tokenize(text)
4
5 START = '<START>'
6 STOP = '<STOP>'
7 #Load and split data into three parts
8 TEXT = data.Field(lower=True, tokenize=tokenizer, init_token=START, eos_token=STOP)
9 train, valid, test = WikiText2.splits(TEXT)
```

```
↳ downloading wikitext-2-v1.zip
wikitext-2-v1.zip: 100%|██████████| 4.48M/4.48M [00:00<00:00, 6.80MB/s]
extracting
```

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

```
↳ Vocabulary size: 28908
```

```

1 BATCH_SIZE = 64
2 # the length of a piece of text feeding to the RNN layer
3 BPTT_LEN = 32
4 # train, validation, test data
5 train_iter, valid_iter, test_iter = data.BPTTIterator.splits((train, valid, test),
6                                                                batch_size=BATCH_SIZE,
7                                                                bptt_len=BPTT_LEN,
8                                                                repeat=False)

```

```

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

```

```

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

```

```

1 class LangModel(nn.Module):
2     def __init__(self, lang_config):
3         super(LangModel, self).__init__()
4         self.vocab_size = lang_config['vocab_size']
5         self.emb_size = lang_config['emb_size']
6         self.hidden_size = lang_config['hidden_size']
7         self.num_layer = lang_config['num_layer']
8
9         self.embedding = None
10        self.rnn = None
11        self.linear = None
12
13        ### TODO:
14        ### 1. Initialize 'self.embedding' with nn.Embedding function and 2 variables
15        ### 2. Initialize 'self.rnn' with nn.LSTM function and 3 variables we have
16        ### 3. Initialize 'self.linear' with nn.Linear function and 2 variables
17        ### Reference:
18        ### https://pytorch.org/docs/stable/nn.html
19
20        ### YOUR CODE HERE (3 lines)
21        self.embedding = nn.Embedding(self.vocab_size, self.emb_size)
22        self.rnn = nn.LSTM(self.emb_size, self.hidden_size, self.num_layer)
23        self.linear = nn.Linear(self.hidden_size, self.vocab_size)
24        ### END OF YOUR CODE
25
26    def forward(self, batch_sents, hidden=None):
27        '''
28        params:
29            batch_sents: torch.LongTensor of shape (sequence_len, batch_size)
30        return:
31            normalized_score: torch.FloatTensor of shape (sequence_len, batch_size,
32            ...

```

```

32     ...
33     normalized_score = None
34     hidden = hidden
35     ### TODO:
36     ###     1. Feed the batch_sents to self.embedding
37     ###     2. Feed the embeddings to self.rnn. Remember to pass "hidden" into
38     ###         use "hidden" when implementing greedy search.
39     ###     3. Apply linear transformation to the output of self.rnn
40     ###     4. Apply 'F.log_softmax' to the output of linear transformation
41     ###
42     ### YOUR CODE HERE
43     x = self.embedding(batch_sents)
44     x, hidden = self.rnn(x, hidden)
45     x = self.linear(x)
46     x = F.log_softmax(x, dim=-1)
47
48     normalized_score = x
49     ### END OF YOUR CODE
50     return normalized_score, hidden

```

```

1 def train(model, train_iter, valid_iter, vocab_size, criterion, optimizer, num_epochs):
2     for n in range(num_epochs):
3         train_loss = 0
4         target_num = 0
5         model.train()
6         for batch in train_iter:
7
8             text, targets = batch.text.to(device), batch.target.to(device)
9             loss = None
10
11             ### we don't consider "hidden" here. So according to the default settings
12             ### YOU CODE HERE (~5 lines)
13             optimizer.zero_grad()
14             outputs, hidden = model(text)
15             loss = criterion(outputs.view(-1, vocab_size), targets.view(-1))
16             loss.backward()
17             optimizer.step()
18             ### END OF YOUR CODE
19             #####
20             train_loss += loss.item() * targets.size(0) * targets.size(1)
21             target_num += targets.size(0) * targets.size(1)
22
23         train_loss /= target_num
24
25         # monitor the loss of all the predictions
26         val_loss = 0
27         target_num = 0
28         model.eval()
29         for batch in valid_iter:
30             text, targets = batch.text.to(device), batch.target.to(device)
31
32             prediction,_ = model(text)

```

```

33         loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))
34
35         val_loss += loss.item() * targets.size(0) * targets.size(1)
36         target_num += targets.size(0) * targets.size(1)
37     val_loss /= target_num
38
39     print('Epoch: {}, Training Loss: {:.4f}, Validation Loss: {:.4f}'.format(n+

```

```

1 def test(model, vocab_size, criterion, test_iter):
2     '''
3     params:
4         model: LSTM model
5         test_iter: test data
6     return:
7         ppl: perplexity
8     '''
9     ppl = None
10    test_loss = 0
11    target_num = 0
12    with torch.no_grad():
13        for batch in test_iter:
14            text, targets = batch.text.to(device), batch.target.to(device)
15
16            prediction,_ = model(text)
17            loss = criterion(prediction.view(-1, vocab_size), targets.view(-1))
18
19            test_loss += loss.item() * targets.size(0) * targets.size(1)
20            target_num += targets.size(0) * targets.size(1)
21
22    test_loss /= target_num
23
24    ### Compute perplexity according to "test_loss"
25    ### Hint: Consider how the loss is computed.
26    ### YOUR CODE HERE(1 line)
27
28    # test_loss is float, not torch.Tensor
29    ppl = np.exp(test_loss)
30    ### END OF YOUR CODE
31    return ppl

```

```

1 num_epochs=10
2 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3 vocab_size = len(TEXT.vocab)
4
5 config = {'vocab_size':vocab_size,
6          'emb_size':128,
7          'hidden_size':128,
8          'num_layer':1}
9
10 LM = LangModel(config)
11 LM = LM.to(device)

```

```

12
13 criterion = nn.NLLLoss(reduction='mean')
14 optimizer = optim.Adam(LM.parameters(), lr=1e-3, betas=(0.7, 0.99))

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


↳ Epoch: 1, Training Loss: 6.0691, Validation Loss: 5.1777
Epoch: 2, Training Loss: 5.4015, Validation Loss: 4.9643
Epoch: 3, Training Loss: 5.1293, Validation Loss: 4.8661
Epoch: 4, Training Loss: 4.9561, Validation Loss: 4.8139
Epoch: 5, Training Loss: 4.8310, Validation Loss: 4.7835
Epoch: 6, Training Loss: 4.7311, Validation Loss: 4.7640
Epoch: 7, Training Loss: 4.6476, Validation Loss: 4.7501
Epoch: 8, Training Loss: 4.5766, Validation Loss: 4.7423
Epoch: 9, Training Loss: 4.5150, Validation Loss: 4.7384
Epoch: 10, Training Loss: 4.4606, Validation Loss: 4.7392

1 # < 150
2 test(LM, vocab_size, criterion, test_iter)

↳ 99.15262130357378

```

▼ Question 6 [code]

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

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

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

```

1 def word_greedy_search(model, start_token, max_len):
2     '''
3     param:
4         model: nn.Module --- language model
5         start_token: str --- e.g. 'he'
6         max_len: int --- max number of tokens generated
7     return:
8         strings: list[str] --- list of tokens, e.g., ['he', 'was', 'a', 'member', '
9     '''
10    model.eval()
11    ID = TEXT.vocab.stoi[start_token]
12    strings = [start_token]
13    hidden = None
14
15    ### You may find TEXT.vocab.itos useful.

```

```

16     ### YOUR CODE HERE
17     outputs = [torch.full(size=(1,), fill_value=ID, dtype=torch.long)]
18     outputs[0] = outputs[0].to(device)
19
20     with torch.no_grad():
21         for _ in range(max_len):
22             inputs = outputs[-1].unsqueeze(dim=0)
23             logits, hidden = model(inputs, hidden)
24             outputs.append(logits[-1,:,:].argmax(dim=-1))
25
26     for o in outputs[1:]:
27         i = o.item()
28         s = TEXT.vocab.itos[i]
29         if s == "<eos>":
30             break
31         else:
32             strings.append(s)
33     ### END OF YOUR CODE
34     return strings

```



```

1 # BeamNode = namedtuple('BeamNode', ['prev_node', 'prev_hidden', 'wordID', 'score',
2 # LMNode = namedtuple('LMNode', ['sent', 'score'])
3
4 def word_beam_search(model, start_token, max_len, beam_size):
5     pass

```



```

1 word_greedy_search(LM, 'he', 64)

```

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


```

1 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

```

1 def char_tokenizer(string):
2     '''
3     param:
4         string: str --- e.g. "I love this assignment"
5     return:
6         char list: list[str] --- e.g. ['I', ' ', 'l', 'o', 'v', 'e', ' ', ' ', 't', 'h', 'i', 's', ' ', 'a', 's', 's', 'i', 'g', 'n', 'm', 'e', 'n', 't']

```



```

6     char_list = list(string)
7     '''
8     char_list = None
9     ### YOUR CODE HERE
10    return list(string)
11    ### END OF YOUR CODE
12    return char_list

```

```

1 test_str = 'test test test'
2 char_tokenizer(char_tokenizer(test_str))

```

```

☞ ['t', 'e', 's', 't', ' ', 't', 'e', 's', 't', ' ', 't', 'e', 's', 't']

```

```

1 CHAR_TEXT = data.Field(lower=True, tokenize=char_tokenizer ,init_token='<START>', e
2 ctrain, cvalid, ctest = WikiText2.splits(CHAR_TEXT)

```

```

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

```

```

☞ Vocabulary size: 247

```

```

1 BATCH_SIZE = 32
2 # the length of a piece of text feeding to the RNN layer
3 BPTT_LEN = 128
4 # train, validation, test data
5 ctrain_iter, cvalid_iter, ctest_iter = data.BPTTIterator.splits((ctrain, cvalid, ct
6                                                                    batch_size=BATCH_SI
7                                                                    bptt_len=BPTT_LEN,
8                                                                    repeat=False)

```

```

1 class CharLangModel(nn.Module):
2     def __init__(self, lang_config):
3         ### YOUR CODE HERE
4         super().__init__()
5         self.model = LangModel(lang_config)
6
7     def forward(self, batch_sents, hidden=None):
8         ### YOUR CODE HERE
9         return self.model(batch_sents, hidden)

```

```

1 def char_train(model, train_iter, valid_iter, criterion, optimizer, vocab_size, num
2     ### YOUR CODE HERE
3     return train(model, train_iter, valid_iter, vocab_size, criterion, optimizer, n

```

```

1 def char_test(model, vocab_size, test_iter, criterion):
2     ### YOUR CODE HERE
3     return test(model, vocab_size, criterion, test_iter)

```

```

1 num_epochs=10
2 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3 char_vocab_size = len(CHAR_TEXT.vocab)
4
5 config = {'vocab_size':char_vocab_size,
6           'emb_size':128,
7           'hidden_size':128,
8           'num_layer':1}
9
10 CLM = CharLangModel(config)
11 CLM = CLM.to(device)
12
13 char_criterion = nn.NLLLoss(reduction='mean')
14 char_optimizer = optim.Adam(CLM.parameters(), lr=1e-3, betas=(0.7, 0.99))

```

```

1 char_train(CLM, ctrain_iter, cvalid_iter, char_criterion, char_optimizer, char_voca

```

```

↳ Epoch: 1, Training Loss: 1.8419, Validation Loss: 1.5488
Epoch: 2, Training Loss: 1.5470, Validation Loss: 1.4423
Epoch: 3, Training Loss: 1.4730, Validation Loss: 1.3961
Epoch: 4, Training Loss: 1.4351, Validation Loss: 1.3709
Epoch: 5, Training Loss: 1.4113, Validation Loss: 1.3537
Epoch: 6, Training Loss: 1.3945, Validation Loss: 1.3416
Epoch: 7, Training Loss: 1.3816, Validation Loss: 1.3317
Epoch: 8, Training Loss: 1.3712, Validation Loss: 1.3234
Epoch: 9, Training Loss: 1.3628, Validation Loss: 1.3166
Epoch: 10, Training Loss: 1.3559, Validation Loss: 1.3110

```

```

1 # <10
2 char_test(CLM, char_vocab_size, ctest_iter, char_criterion)

```

```

↳ 3.6844402887921337

```

```

1 def char_greedy_search(model, start_token, max_len):
2     '''
3     param:
4         model: nn.Module --- language model
5         start_token: str --- e.g. 'h'
6         max_len: int --- max number of tokens generated
7     return:
8         strings: list[str] --- list of tokens, e.g., ['h', 'e', ' ', 'i', 's',...]
9     '''
10    model.eval()
11    ID = CHAR_TEXT.vocab.stoi[start_token]
12    strings = [start_token]
13    hidden = None
14
15    ### You may find CHAR_TEXT.vocab.itos useful.
16    ### YOUR CODE HERE
17    strings = word_greedy_search(model, start_token, max_len)
18    ### END OF YOUR CODE

```

```
19     return strings
```

```
1 TEXT = CHAR_TEXT
2 print("".join(char_greedy_search(CLM, "h", 64)))
```

```
↳ he state , and the story , and the story , and the story , and th
```

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.

