SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.040 Natural Language Processing, Summer 2021
Mini Project

Due 17 June 2021, 5pm

Mini Project will be graded by Ming Shan

Introduction Language models are very useful for a wide range of applications, e.g., speech recognition and machine translation. Consider a sentence consisting of words $x_1, x_2, \ldots, x_m$, where m is the length of the sentence, the goal of language modeling is to model the probability of the sentence, where $m \geq 1$, $x_i \in V$ and V is the vocabulary of the corpus:

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

In this project, we are going to explore both statistical language model and neural language model on the Wikitext-2 datasets.

Requirements torch/torchtext/nltk/numpy

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.

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:

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

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

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 $X^{(j)}$, 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.

Statistical Language Model (30 points)

Question 1 [code] (1+1 points)

- Implement the function compute_ngram that computes n-grams in the corpus. (Do not take the START and STOP symbols into consideration for now.)
- List 10 most frequent unigrams, bigrams and trigrams as well as their counts.(Hint: use the built-in function .most_common in Counter class)

Question 2 [code] (1+1+2+2 points) 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".

- Implement the pad_sents function.
- Pad "train_sents".
- Apply compute_ngram function to these padded sents.
- 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.

Question 3 [code] (4+5+5 points)

- Implement add_k_smoothing_ngram function to estimate ngram probability with "add-k" smoothing technique.
- Implement interpolation_ngram function to estimate ngram probability with "interpolation" smoothing technique.
- 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.

Question 4 [code][written] (3+3+1 points)

- 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).
- 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).
- Based on the methods and parameters we provide, choose the method that peforms best on the validation data.

Question 5 [code] (1 points) Evaluate the perplexity of the test data "test_sents" based on the best model you choose in "Question 4".

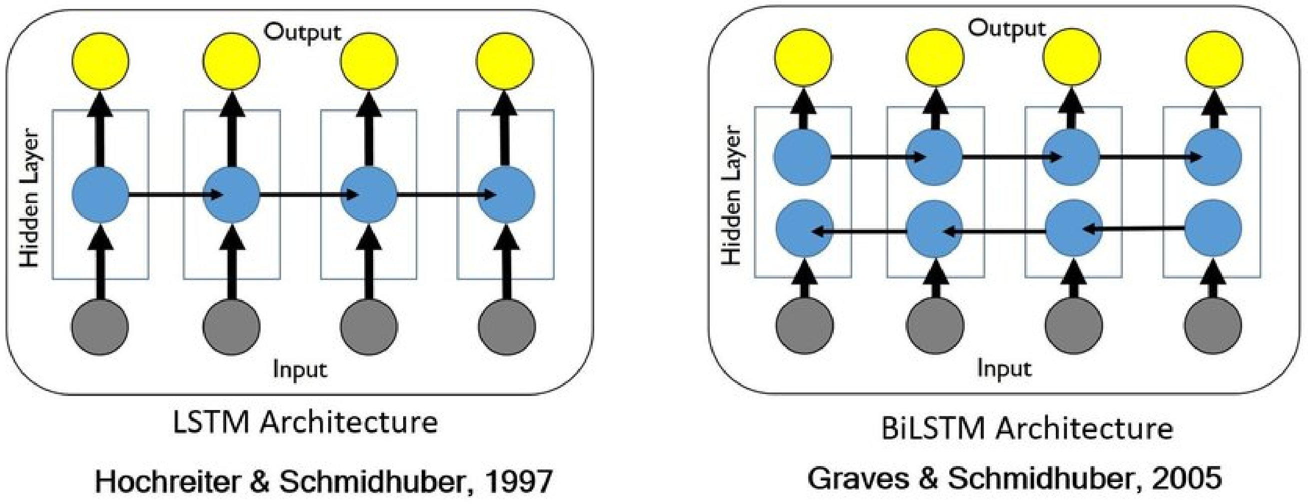

Figure 1: LSTM Language Model

Neural Language Model (20 points) Suppose we have a training set $(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}) \in \mathcal{D}, j \in \{1, 2, ..., |\mathcal{D}|\}$, where $\mathbf{x}^{(j)} = (x_0^{(j)}, x_1^{(j)}, ..., x_n^{(j)})$ is the input sentence and $\mathbf{y}^{(j)} = (x_1^{(j)}, x_2^{(j)}, ..., x_{n+1}^{(j)})$ is the sentence our model will predict. Typically, $x_i^{(j)} \in \mathbb{R}^{|V|}$ is a one-hot vector, where $|V|$ is the size of the vocabulary. $x_0^{(j)}$ is a special "START" token and $x_{n+1}^{(j)}$ is a special "STOP" token. A neural language model (RNN) will be trained to predict the next word x_{t+1} given the current word x_t and history information of words h_{t-1}.

$$p(x_1, x_2, ..., x_{n+1} \mid x_0) = \prod_{t=0}^{n} p(x_{t+1} \mid x_t, ..., x_0) = \prod_{t=0}^{n} p(x_{t+1} \mid h_t) \tag{9}$$

where $h_t = \text{RNN}(x_t, h_{t-1})$ denotes the history information of words. h_0 is usually set to be a zero vector.

In a neural language model, we first map a sequence of words $x_1^{(j)}, ..., x_n^{(j)}$ to word embeddings (we don't consider special token for simplicity), $e_1^{(j)}, ..., e_n^{(j)}$, where $e_i^{(j)} \in \mathbb{R}^k$. Then, we feed those embeddings into a RNN model (a LSTM in the project), obtaining a sequence of hidden states $h_1^{(j)}, h_2^{(j)}, ..., h_n^{(j)}$, where $h_i^{(j)} \in \mathbb{R}^d$. Next, we apply a linear transformation $W \in \mathbb{R}^{|V| \times d}$ to those hidden states, yielding $\hat{h}_1^{(j)}, \hat{h}_2^{(j)}, ..., \hat{h}_n^{(j)}$, where $\hat{h}_i^{(j)} \in \mathbb{R}^{|V|}$. We apply "softmax" function to each $\hat{h}_i^{(j)}$, yielding a probability distribution $\hat{y}_i^{(j)} \in \mathbb{R}^{|V|}, i \in [1, 2, ..., n]$ over the vocabulary V. $\hat{y}_i^{(j)}$ represents the probability of a certain word at position i. Finally, we compute the "Negative Log Likelihood Loss" between model's predictions $\hat{y}_i^{(j)}, i \in \{1, 2, ..., n\}$ and gold targets $\mathbf{y}^{(j)}$.

Question 6 [code] (3+3+2+2 points) We will create a LSTM language model as shown in figure 1 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.

- 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.

Question 7 [code] (4+1 points)

- Implement the word_greedy_search function returns a generated sentence based on an arbitrary word token. This function takes an arbitrary token and a model as input.
- Run the "word_greedy_search" using the start token "he" and the **Unidirectional LSTM** (trained in Question 7).
- Based on your understanding, can we use the **Bidirectional LSTM** for this language generation (decoding) task? Explain why?

Question 8 [code] (3+1+1 points)

- 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 $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ 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

$$similarity = cos(\theta) = \frac{\boldsymbol{w}_1^{\mathrm{T}} \boldsymbol{w}_2}{||\boldsymbol{w}_1||_2 ||\boldsymbol{w}_2||_2} \tag{10}$$

  Give the explanation of the results.
- Based on your understanding, can we use the **Bidirectional LSTM** for this contextual embedding task? Explain why?

How to submit

1. Fill up you student ID and name in the Jupyter Notebook.
2. Click the Save button at the top of the Jupyter Notebook.
3. Select Cell - All Output - Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File – Download as – PDF via LaTeX
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see! Submit your PDF on eDimension.