Assignments

Computational Semantics for Natural Language Processing

Spring 2022

March, 2022

(Assignment 1)

Dept. of Computer Science, ETH Zürich

Mrinmaya Sachan

http://www.mrinmaya.io/teaching_csnlp22

1 Skip-Gram (20)

Consider a word $w \in V_W$ and its context $c \in V_C$, where V_W is the word vocabulary[1] and V_C is the context vocabulary. Let the collection[2] of observed word and context pairs be $\mathcal{D}$. We denote the number of times the pair (w, c) appears in $\mathcal{D}$ as $\#(w, c)$. Similarly, $\#(w) = \sum_{c' \in V_C} \#(w, c')$ and $\#(c) = \sum_{w' \in V_W} \#(w', c)$ are the number of times w and c occur in $\mathcal{D}$, respectively.

Each word $w \in V_W$ is associated with a vector $\vec{w} \in \mathbb{R}^d$ and similarly each context $c \in V_C$ is represented as a vector $\vec{c} \in \mathbb{R}^d$, where d is the embedding's dimensionality. $W \in \mathbb{R}^{|V_W| \times d}$ and $C \in \mathbb{R}^{|V_C| \times d}$ are matrices with their rows W_i (C_i) being the vector embedding of the ith word in the corresponding vocabulary V_W (V_C).

1. Consider a word-context pair (w, c). We model the probability $P(D = 1|w, c)$ that (w, c) come from the observed data $\mathcal{D}$ as a binary classification problem:

$$P(D = 1|w, c) = \sigma(\vec{w} \cdot \vec{c}) = \frac{1}{1 + \exp(-\vec{w} \cdot \vec{c})}$$

   Consider the log-likelihood function shown below. Can we use this log likelihood function as the objective function? Why? (**3**)

$$\ell = \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) \log \sigma(\vec{w} \cdot \vec{c})$$

2. As a result, we randomly draw negative contexts c_N, and the objective for a pair (w, c) is then to maximize $P(D = 1|w, c)$ for (w, c) while maximizing $P(D = 0|w, c)$ for negative samples. Let k be the number of negative samples, and c_N be drawn according to the empirical unigram distribution $P_D(c) = \frac{\#(c)}{|D|}$. Write down the objective function for a single pair and all pairs (hint: use σ, P_D, and $\mathbb{E}_{c_N \sim P_D}[\cdots]$.). (**4**)

3. Assuming that each product $\vec{w} \cdot \vec{c}$ is independent of the others, extract the local objective $\ell(w, c)$ for a specific (w, c) pair from your objective function ℓ. (**3**)

4. Let $x = \vec{w} \cdot \vec{c}$. Find $x^* = \arg\max_x \ell$ by setting its partial derivative with respect to x to zero. (**6**)

5. In the case of $k = 1$, show x^* is exactly the pointwise mututal information (PMI) between w and c. (**1**)

[1] A vocabulary is a set of **distinct** words.

[2] A collection is a multiset, i.e., a set in which **multiplicity** is significant.

6. Assume we have W^* and C^* with their rows all being optimized as above. What will $M = W^* \cdot C^{*\top}$ be? What does the result imply? (**3**)

2 Multi-prototype Word Embeddings (20)

Following the notations of Q1, given a pair of word and context (w, c), assuming w has N_w word senses (prototypes), $W \in \mathbb{R}^{|V_W| \times N_w \times d}$ is now a 3D matrix. We introduce a latent variable $h_w \in \{1, \cdots, N_w\}$. where $h_w = k$ if w means its k-th prototype. Let $\pi_{wk} = P(h_w = k|w)$ denote the prior probability of prototypes. We model $P(D = 1|w, c)$ (the probability that (w, c) come from the observed data $\mathcal{D}$) as a mixture model:

$$\begin{aligned} P(D = 1|w, c) &= \sum_{k=1}^{N_w} P(h_w = k|w) P(D = 1|w, h_w = k, c) \\ &= \sum_{k=1}^{N_w} \pi_{wk} P(D = 1|w, h_w = k, c). \end{aligned}$$

We will now use an expectation-maximization (EM) algorithm to train the multi-prototype word embeddings. The EM algorithm is a popular algorithm used to find local maximum likelihood parameters of a model that involves unobserved latent variables (sense-prototypes in our case). The algorithm alternates between performing an expectation (E) step, and a maximization (M) step. If you are not familiar with the EM algorithm, you can find more details here: https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm.

Next, we will derive the EM algorithm for multi-protoype word embeddings with a series of subquestions:

1. Write the likelihood $P(\mathcal{D})$ and log-likelihood $L(\mathcal{D})$ for all pairs. (**2**)

2. Now we introduce a binary latent variable $M_{wkc} \in \{0, 1\} := \mathbb{1}\{h_w = k\}$, where $\mathbb{1}\{\}$ is the indicator function. Write the joint likelihood $P(\mathcal{D}, M)$ and log-likelihood $L(\mathcal{D}, M)$. (**4**)

3. E-step: In this step, we create a function for the expectaion of the log-likelihood evaluated using the current estimate for the parameters. Compute $Q(\Theta) := \mathbb{E}_{M|\mathcal{D}} L(\mathcal{D}, M)$, given $\mathbb{E}_{M|\mathcal{D}} M_{wkc} =: \gamma_{wkc}$, and state what are the parameters Θ. (**4**)

4. Derive γ_{wkc} as an expression of $\{\pi_{wk'}\}_{k'=1}^{N_w}$ and $\{P(D = 1|w, h_w = k', c)\}_{k'=1}^{N_w}$ from its definition (hint: use Bayes' theorem). (**5**)

5. M-step: In this step, we compute parameters maximizing the expected log likelihood found in the E step. Please compute $\pi^*_{wk} := \arg\max_{\pi_{wk}} Q(\Theta)$ (hint: use Lagrange multiplier). (**5**)

3 Hierarchical Softmax and Huffman Coding (20)

Hierarchical softmax is another strategy to approximate the expensive softmax computation when training word embeddings, aside from *negative sampling* that we have already introduced in class.

Given a document D with $|D|$ words, and its vocabulary set V with size $|V|$, the frequency of word w is defined as $p(w) = \frac{\#w}{|D|}$, where $\#w$ denotes the number of times that the word w appears in the document. For the Skip-gram model mentioned above, we can calculate a score for center word w with respect to its context word c e.g., $s(w, c) = \vec{w} \cdot \vec{c}$. Using softmax function, we have the (empirical) probability:

$$P_w(c) = \frac{\exp\{(s(w, c))\}}{\sum_{c \in V} \exp\{(s(w, c))\}}.$$

Now for each (w, c) pair, the time complexity of calculating the probablity is $O(|V|)$.

Since the vocabulary size $|V|$ could be very large, we can use **hierarchical softmax** as an **approximation** of the probability to improve time efficiency. Specifically, we construct a binary tree, whose leaf nodes are used to represent all words in the vocabulary. Then, we can traverse the tree to compute the probability of a word. The probability of each word is computed as the product of the probability of choosing the branch that is on the path from the root to the word. i.e., the calculation of the $P_w(c)$ is approximated as

$$P_w(c) = \prod_{j=1}^{L(c)-1} P_w((c, j) \to (c, j + 1)),$$

where $L(c)$ represents the depth of node c in the tree, (c, j) denotes the node of depth j (on the path from root to leaf node c), and $P_w((c, j) \to (c, j+1))$ represent the probablility of transition from (c, j) to $(c, j+1)$. By traversing the tree, The time complexity of calculating $P_w(c)$ is reduced from $O(|V|)$ to $O(L(c))$ (i.e., here it's $O(\log |V|)$), since the number of parameters on which $P_w(c)$ depends is only **proportional to the depth** of the tree **instead of all the leaf nodes**. Please answer following questions:

1. **Binary tree for hierarchical softmax.** Let's start with a simple example. In this question, assume $|V| = 2^{10} = 1024$. We construct a complete binary tree for hierarchical softmax, whose $|V|$ leaf nodes correspond to $|V|$ words. For the constructed binary tree, how many inner nodes and edges does it have? For the probability of one pair (w, c), with the help of the binary tree, write down the number of operations that we need to compute $P_w(c)$ (i.e., what is $L(c) - 1$). Now extend this to the general condition where the vocabulary size is $|V|$. Answer the question again (here you can use big O notation $O(\cdot)$) (**6=3+3**).

2. **Better than binary tree: Huffman tree.** Consider the binary tree in part 1. Intuitively, if we put words with high frequency closer to the root node (and leave words with low frequencies farther away from the root node), we can further reduce our number of operations (i.e., $L(c)$). Now start with a simple example with $|V| = 4$: $[p(w_1) = 0.1; p(w_2) = 0.1; p(w_3) = 0.3; p(w_4) = 0.5]$, try to construct a binary tree that has the **least** expected number of operations. Draw the tree and write down the expected number of operations (i.e., $\mathbb{E}[L(c) - 1]$). This binary tree (may not be unique) is called **Huffman tree**. Argue that in the Huffman tree, the expected number of operations for each word $E[L(c) - 1]$ is greater than or equal to $E_{w_i}[-\log(p(w_i))]$. Now use this result to give the **least** expected number of operations in the general case with a vocabulary size $|V|$. Here you can use big O notation $O(\cdot)$. (**9=2+2+5**).

3. **Huffman tree for hierarchical softmax.** Next, we consider the frequency of words in the document to further improve time efficiency. Assume the distribution of word w in the vocabulary is uniform ($\#w_i = \#w_j \quad \forall w_i, w_j \in V$). And for each word $w \in V$, it has $m \times \#w$ context words (note that $\#w \geq 1$). In question 1, we considered the time complexity of hierarchical softmax operation for one pair (w, c).

Now write down the time complexity of the softmax operation for the whole document with a (randomly constructed) binary tree (here you can use big O notation $O(\cdot)$). Extend the word distribution to the general condition using a Huffman tree, write down the time complexity of the softmax operation for the whole document (here you can use the big O notation $O(\cdot)$).

Given the derived complexity, explain why we can expect the Huffman tree to help. (Hint: Recall the relationship between the depth of a leaf node and the frequency of a word) (**5=2+2+1**).

4 FastText Embeddings (20)

Let's recap the FastText model. Similar to the Skip-gram model, the loss function we are trying to optimize (for a single context word) is given as follows:

$$\mathcal{L}(w_t, w_c, N_K) = -\log(\sigma(s(w_t, w_c))) - \sum_{n \in N_k} \log(\sigma(-s(w_t, w_n)))$$

$$s(w_1, w_2) = \sum_{g \in G_w} z_g^T u_c$$

Make sure you understand how this expression is written using the sigmoid (σ) is similar to the one in the paper. Extending this to every context word, we get the objective as:

$$J(w_t, w_{c_1}, w_{c_2} \dots w_{c_t}) = \sum_{c \in C_t} \mathcal{L}(w_t, w_c, N_K)$$

The embeddings for the context and negatively sampled words are from the *outside* embeddings U while the embeddings of the n-grams from the center word are drawn from the *inside* hash embeddings Z.

Let's look at some elementary techincal questions about Skip-gram and FastText before we move on to implement them.

1. You train a vanilla Skip-gram model and a fastText model (with parameters as described in the paper) on a corpus of text. You train these models to have embeddings of size 128. Your corpus has 40000 unique words. Your model is upperbounded by $2e6$ subword tokens.

   (a) How many parameters are there in your Skip-gram model? **(1)**

   (b) How many parameters are there in your FastText model? **(1)**

2. Before we get to the implementation we need to calculate the derivative of the loss function.

   (a) Calculate $\frac{\partial \mathcal{L}}{\partial u_c}$ and $\frac{\partial \mathcal{L}}{\partial u_n}$ **(3)**

   (b) Calculate $\frac{\partial \mathcal{L}}{\partial z_g}$ **(3)**

3. Download the code from https://polybox.ethz.ch/index.php/s/pSN67mjEfSjRKtQ and follow the instructions:

   (a) Install Anaconda[3] or Miniconda[4] and create a virtual environment by running the following commands in the project directory:
   `conda env create -f env.yml`
   `conda activate a1`

   (b) We will start with `word2vegenc.py`. Go ahead and fill out the code for for `negSamplingLossAndGradient`, `Skip-gram`. You will need to calculate the loss and the gradients. **(4 × 2)**

   (c) Test out the gradients by running `python word2vec.py`

   (d) Now let's run FastText for 20000 iterations, do this by `python run.py` (this will take around 3-4 hours, so please plan accordingly)

   (e) The code will generate a plot, paste that in your assignment, along with the plot submit all the python files (`*.py`) in a zip file.

4. What do you think are the advantages a subword approach such as fastText has over a character embedding-based approach? **(2)**

5. Can you think of a few examples where subword information might hurt the embedding's performance? **(2)**

5 Coding: LSTM vs. GRU (20)

LSTM and GRU are two popular models designed to prevent vanishing gradient problem. GRUs are simpler than LSTMs, which control the flow of information like the LSTM unit. From empirical experience, training a GRU is faster than that of LSTM. However, LSTM and its variants outperform GRUs if the training data are sufficient. More details can be found in [Kaiser and Sutskever(2016), Yin et al.(2017)Yin, Kann, Yu, and Schütze, Chung et al.(2014)Chung, Gülçehre, Cho, and Bengio]. In this question, you will first implement an LSTM and a GRU, and then evaluate them on a small dataset for sentiment analysis: IMDB[5].

1. Prepare your environment. Download the code here and follow the instructions below:

   (a) Prepare your python3 environment, and install PyTorch 1.8.0[6]

   (b) install torchtext (`conda install torchtext=0.9.0 -c pytorch`).

   (c) install tokenizer (`conda install spacy=3.2 -c conda-forge` and `python -m spacy download en_core_web_sm`).

2. Complete the code of GRUCell and LSTMCell in the file `assignment_code.py`. The missing parts are highlighted. Please don't change other parts of the code, otherwise TAs may not be able to run the code in your submission.

   (a) Read the comments in the code first. Cells are the backbone of recurrent neural networks. A recurrent layer contains a cell object. The cell implements the computations of one single step, while the recurrent layer calls the cell and performs recurrent computations (with loop), as illustrated during exercise session 3.

   (b) Complete the code in classes `GRUCell_assignment` and `LSTMCell_assignment`. Each class contains two functions: initialization `__init__()` and forward `forward()`. Below are the definitions of the two cells. You can also refer to the PyTorch documentation[7].

[3] https://docs.anaconda.com/anaconda/install/
[4] https://docs.conda.io/en/latest/miniconda.html
[5] https://ai.stanford.edu/ amaas/data/sentiment/
[6] link's here. Please try to install 1.8.0 version as we have not tested our code with other `torchtext` libraries.
[7] LSTMCell: https://pytorch.org/docs/stable/generated/torch.nn.LSTMCell.html. GRUCell:https://pytorch.org/docs/stable/generated/torch.nn.GRUCell.html

GRUCell:

$$
\begin{aligned}
r &= \sigma(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \\
z &= \sigma(W_{iz}x + b_{iz} + W_{hz}h + b_{hz}) \\
n &= \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \\
h' &= (1 - z) * n + z * h
\end{aligned}
$$

LSTMCell:

$$
\begin{aligned}
i &= \sigma(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= \sigma(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + b_{ig} + W_{hg}h + b_{hg}) \\
o &= \sigma(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * \tanh(c')
\end{aligned}
$$

where $*$ is the Hadamard product (element-wise product).

3. Test your code. The code can be run either on GPUs or on CPUs.

   (a) Test your GRU code with python command `python src/run_gru.py`. It will take around 5 minutes with either GPU or CPU. **(5)**

   (b) Test your LSTM code with python command `python src/run_lstm.py`. It will take around 5 minutes with either GPU or CPU. **(5)**

   (c) Run python command `python src/run_results.py` to get the final results. It will take around 15 minutes with either GPU or CPU and the output will be saved in `results.json`. The results of first three models: "torch_RNN", "torch_GRU", "torch_LSTM" are PyTorch versions. Compare your models with theirs. The accuracies of your models should be similar to theirs. It is normal that your models are slower than theirs[8].

4. Put your results (`results.json`) in your assignment submission[9]. Try to describe and analyze your results (several sentences are good enough), e.g., which model converges faster (LSTM or GRU), which model is more time efficient. Submit the python file (`assignment_code.py`). **(10)**

[8] As they use cuDNN to speed up: https://developer.nvidia.com/blog/optimizing-recurrent-neural-networks-cudnn-5/.
[9] Don't worry if you see a very low accuracy. **This problem is not graded based on accuracy**.

References

[Chung et al.(2014)Chung, Gülçehre, Cho, and Bengio] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[Kaiser and Sutskever(2016)] Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In *ICLR*, 2016.

[Yin et al.(2017)Yin, Kann, Yu, and Schütze] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of CNN and RNN for natural language processing. *CoRR*, abs/1702.01923, 2017.