Instructor: Vishnu Boddeti
CSE 891-001: Deep Learning
Programming Homework 1
September 29th, 2021

Instructions:

- **Submission:** Only homeworks uploaded to Google Classroom will be graded. Submit solutions in PDF format.
- **Integrity and Collaboration:** You are expected to work on the homeworks by yourself. You are not permitted to discuss them with anyone except the instructor. The homework that you hand in should be entirely your own work. You may be asked to demonstrate how you got any results that you report.
- **Clarifications:** If you have any question, please look at Google Classroom first. Other students may have encountered the same problem, and is solved already. If not, post your question there. We will respond as soon as possible.

In this assignment we will learn about word embeddings and will train a multi-layered perceptron to learn about words. We will consider the task of predicting the next word in a sentence given a sequence of words. You will implement the backpropagation computations for the neural network and analyze the resulting learned word representations. The amount of code you have to write is very short – about 10 lines – but you need to think very carefully to write each line. You will need to first derive the updates mathematically and then implement them using matrix-vector operations in PyTorch.

0 Code and Software Setup The first step is to install Python, PyTorch and other required libraries. After you install Python, see the instructions on this web page to install PyTorch (`https://pytorch.org`). PyTorch provides a tensor library just like Numpy, the code will be based on PyTorch, you are encouraged to look at the documentation.

Starter Code and Data Look at the sentences in `raw_sentences.txt`. It contains the sentences that we will be using for this assignment. These sentences are fairly simple ones and cover a vocabulary of only 250 words (+ 1 special [MASK] token word).
We extracted the 4-grams from this dataset and divided them into training, validation and test sets. To inspect this data, run the following within IPython.

```
import pickle
f = open('data.pk', 'rb')
data_obj = pickle.load(f, encoding='latin1')
```

Here `data_obj` is a Python dictionary which contains the vocabulary, as well as the inputs for all the three data splits. `data['vocab']` is a list of the 251 words in the dictionary. `data['train_inputs']` is a 372,500 $\times$ 4 matrix with each row corresponding to the indices of 4 consecutive context words. The validation and test splits also follow the same structure.

The starter code for this assignment contains the following main files:

- `glove.py`: The main file that you need to run for the first question in this assignment.
- `glove_model.py`: The file that you need to fill in for the first question in this assignment.
- `main.py`: The main file that you need to run to train and evaluate your language models. This file also defines the model that you will be training.
- `layer.py`: Different types of layers in our neural network are implemented here. Each layer is implemented as a class, and each class consists of three functions, `forward`, `backward` and `zero_grad`. You will be implementing the backward functions for all the layers that we need for the language model.

1 Linear Embedding - GLoVe (2pts) In this section we will be implementing a simplified version of GLoVe [1]. Given a corpus with V distinct words, we define the co-occurrence matrix $\boldsymbol{X} \in \mathbb{N}^{V \times V}$ with entries $\boldsymbol{X}_{ij}$ representing the frequency of the i-th word and j-th word in the corpus appearing in the same context - in our case the adjacent words. The co-occurrence matrix can be symmetric (i.e., $\boldsymbol{X}_{ij} = \boldsymbol{X}_{ji}$) if the order of the words do not matter, or asymmetric (i.e., $\boldsymbol{X}_{ij} \neq \boldsymbol{X}_{ji}$) if we wish to distinguish the counts for when i-th word appears before j-th word. GLoVe aims to find a d-dimensional embedding of the words that preserves properties of the co-occurrence matrix by representing the i-th word with two d-dimensional vectors $\boldsymbol{w}_i$, $\tilde{\boldsymbol{w}}_i \in \mathbb{R}^d$, as well as two scalar biases $b_i, \tilde{b}_i \in R$. This objective can be written as[a]:

$$L(\{\boldsymbol{w}_i, \tilde{\boldsymbol{w}}_j, b_i, \tilde{b}_j\}_{i=1}^{V}) = \sum_{i,j=1}^{V} (\boldsymbol{w}_i^T \tilde{\boldsymbol{w}}_j + b_i + \tilde{b}_j - \log \boldsymbol{X}_{ij})^2 \tag{1}$$

When the bias terms are omitted and we tie the two embedding vectors $\boldsymbol{w}_i = \tilde{\boldsymbol{w}}_i$, then GLoVe corresponds to finding a rank-d symmetric factorization of the co-occurrence matrix.

1. **GLoVe Parameter Count (0 pt):** Given the vocabulary size V and embedding dimensionality d, how many trainable parameters does the GLoVe model have? Note that each word in the vocabulary is associated with 2 embedding vectors and 2 biases.

2. **Gradient Expression (1 pt):** Write the expression for $\frac{\partial L}{\partial \boldsymbol{w}_i}$, the gradient of the loss function L with respect to one parameter vector $\boldsymbol{w}_i$. The gradient should be a function of $\boldsymbol{w}$, $\tilde{\boldsymbol{w}}$, b, $\tilde{b}$, $\boldsymbol{X}$ with appropriate subscripts (if any).

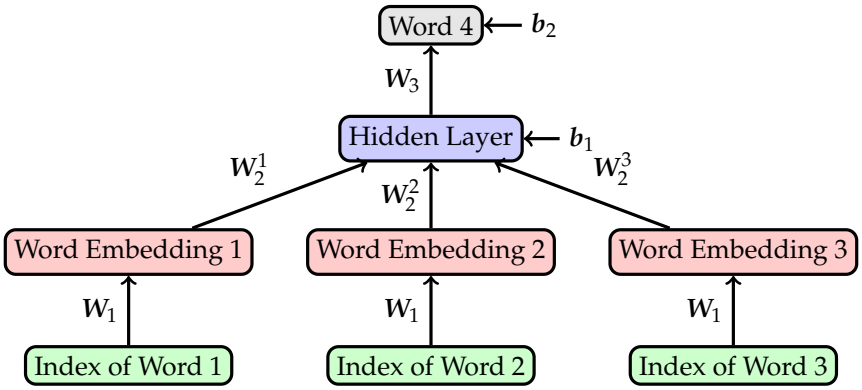
3. **Implementation (1 pt):** Implement the gradient update of GLoVe in `glove.py`. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

4. **Embedding Dimension (0 pt):** Train the both the symmetric and asymmetric GLoVe model with varying dimensionality d. Comment on the results:

   - Which d leads to optimal validation performance for the asymmetric and symmetric models?
   - Why does / doesn't larger d always lead to better validation error?
   - Which model is performing better, and why?

[a] This is a simplified version of the objective by omitting the weighting function. For the complete algorithm please see [1].

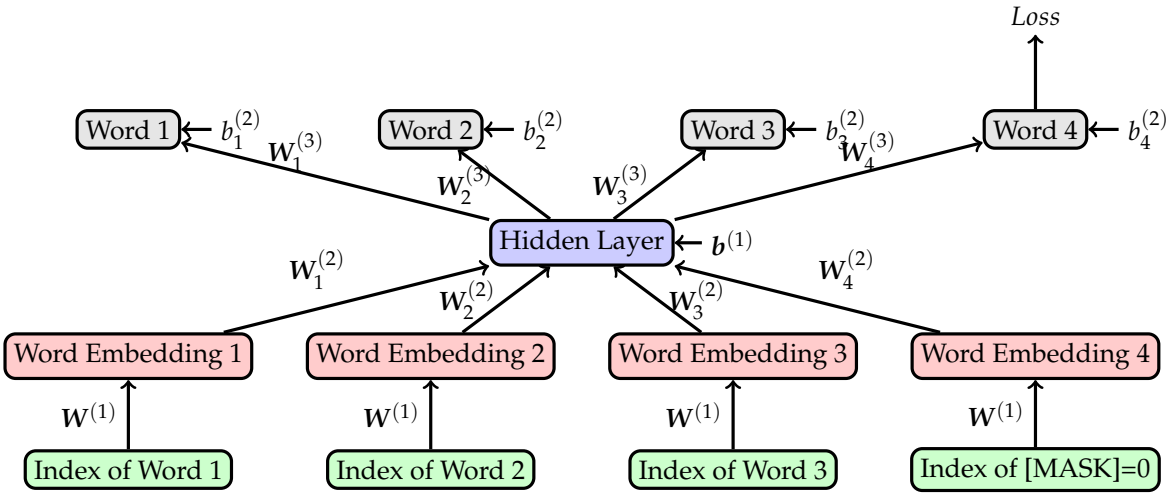
2 Network Architecture (1pts) You will train a neural language model using a multi-layered perceptron like the one in [2] (see below for a pictorial illustration). It receives 3 consecutive words as the input and aims to predict a distribution over the next word (the *target* word). We will train the model using the cross-entropy criterion, which is equivalent to maximizing the probability it assigns to the *target* words in the training set. Hopefully, it will also learn to make sensible predictions for sequences it hasn't seen before.



The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of 3 consecutive words, provided as integer valued indices, i.e., the 250 words in our dictionary are arbitrarily assigned a unique integer between 0 and 249. The embedding layer maps each word to its corresponding vector representation. This layer has $3 \times d$ units, where d is the embedding dimension, and functions as a look-up table. We will share the same look-up table for all the 3 positions, so we will learn a single common word embedding matrix for each context position. The embedding layer is connected to the hidden layer, which uses a sigmoid loss activation function. The hidden layer is connected to the output layer, and the output layer is a softmax over the V words in our dictionary.

1. The trainable parameters of the model consist of 3 weight matrices and two bias vectors. Assuming that we have V words in the dictionary, use N words as our input context, a D-dimensional word embedding and a hidden layer with H units. What is the total number of trainable parameters in the model as a function of V,N,D,H? In the diagram given above, which part of the model (i.e., `word_embbeding_weights`, `embed_to_hid_weights`, `hid_to_output_weights`, `hid_bias`, or `output_bias`) has the largest number of trainable parameters if we have the constraint that $V \gg H > D > N$? Explain your reasoning.

3 Training Neural Networks (5pts) We will modify the architecture slightly from the previous section, inspired by BERT [3]. Instead of having only one output, the architecture will now take in $N = 4$ context words, and also output predictions for $N = 4$ words. See below for a diagram of this architecture.



During training, a word is randomly sampled from among the N context words and replaced with a [MASK] token. The network is tasked to predict the word that was masked, at the corresponding output word position. This [MASK] token is assigned the index 0 in our dictionary. The weights $\boldsymbol{W}^{(2)} \in \mathbb{R}^{NV \times H}$ as the output layer has NV neurons, where the first V output units are for predicting the first word, then the next V are for predicting the second word, and so on. The output of the network is a probability distribution over V for each output word[a]. Only the output word positions that were masked in the input are included in the cross entropy loss calculation: $C = -\sum_{i=1}^{B}\sum_{n=1}^{N}\sum_{j=1}^{V} m_n^{(i)}(t_{n,j}^{(i)} \log y_{n,j}^{(i)})$, where $y_{n,j}^{(i)}$ denotes the output probability prediction from the neural network for the i-th training example for the word j in the n-th output word, and $t_{n,j}^{(i)}$ is 1 if for the i-th training example, the word j is the n-th word in context. Finally, $m_n^{(i)} \in 0, 1$ is a mask that is set to 1 if we are predicting the n-th word position for the i-th example (because we had masked that word in the input), and 0 otherwise.

In this part of the assignment, you will implement a method to compute the gradients using backpropagation. In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than `for` loops. You should compute the derivatives on pencil and paper using chain rule and then express them in terms of matrix operations. Check out the `forward` functions within `layer.py` to understand how each layer can be implemented using matrix operations.

Once you have implemented the gradient computation, you will need to train the model. The training procedure is implemented in `main.py`. Be sure to set the appropriate parameter values in `args`. As the model trains it will print the aggregate loss and accuracy of the trained model at the end of each epoch.

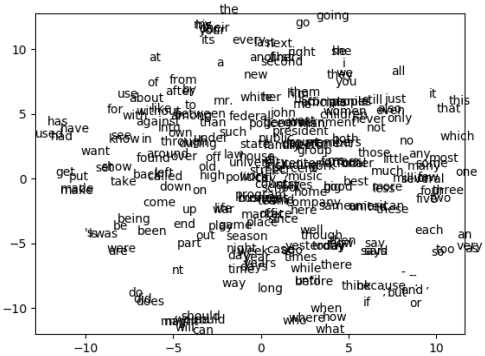
To demonstrate that you have correctly implemented the gradient computation, please include the following with your assignment submission:

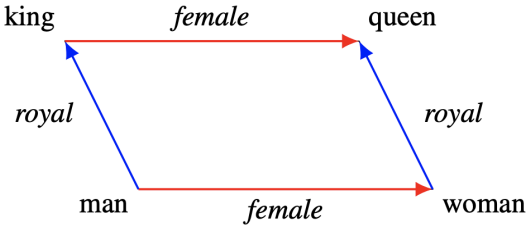
- You will submit the `layer.py` file, you do not need to modify any of the code except the parts that we have asked you to implement.

[a]For simplicity we also include the [MASK] token as one of the possible prediction even though we know the target should not be this token

4 Analysis (2pts) In this part, you will perform arithmetic calculations on the word embeddings learned from previous models and analyze the representation learned by the networks with t-SNE plots. You should first train a GLoVe and a neural network model with a 16-dimensional embedding and 128 hidden units, as discussed in the previous question. You will use these trained models for the remainder of this section. Be sure to load your trained model weights for this analysis. The following methods can be used to analyze the model after the training is done:

- `tsne_plot`: Creates a 2-D plot of the distributed representation space using an algorithm called t-SNE (if you do not know what this is for the assignment, you are encouraged to read more about it on your own). Nearby points in the 2-D space are meant to correspond to nearby points in the 16-D word embedding space. From the learned model, you can create pictures of the kind shown below.





- `display_nearest_words`: Shows the words whose embedding vectors are nearest to the given word
- `word_distance`: Computes the distance between the embeddings of two given words.

Using these methods, please answer the following question.

1. **1pt** Running the `glove.py` file will create a few plots, (a) 2-dimensional visualization for the trained model Look at the plot and find a few clusters of related words. What do the words in each cluster have in common? (b) 2-dimensional visualization for the GloVe model. How do the t-SNE embeddings for both models compare? (c) 2-dimensional visualization of the learned representation. How does this compare to the t-SNE embeddings? Please answer in 2 sentences for each question and show the plots in your submission.
2. **Word Analogy Arithmetic (1pt):** A word analogy f is an invertible transformation that holds over a set of ordered pairs S iff $\forall(x,y) \in s, f(x) = y \wedge f^{-1}(y) = x$. When f is of the form $x \mapsto x + r$, it is a linear word analogy. Arithmetic operators can be applied to vectors generated by language models. There is a famous example: $\overrightarrow{king}$ - $\overrightarrow{man}$ + $\overrightarrow{women} \approx \overrightarrow{queen}$. As shown in figure above, these linear word analogies form a parallelogram structure in the vector space [4]. We will explore a property of linear word analogies. We will use the embeddings from the symmetric, asymmetrical GloVe model, and the neural network model to perform arithmetics.

4 Analysis Continued
Please answer the following questions:

1. **[1pt]** Perform arithmetic on words *he*, *him*, *her*, using: (1) symmetric, (2) averaging asymmetrical GloVe embedding, (3) concatenating asymmetrical GloVe embedding, and (4) neural network word embedding. That is, we are trying to find the closet word embedding vector to the vector *emb(he) - emb(him) + emb(her)*. For each sets of embeddings, you should list out: (1) what the closest word that is not one of those three words, and (2) the distance to that closest word. Is the closest word she? Compare the results with the tSNE plots.
2. **Extra Point [1pt]** Pick another quadruplet from the vocabulary which displays the parallelogram property (and also makes sense sementically) and repeat the above procedures. Compare and comment on the results from arithmetic and tSNE plots.

Submission Here is everything you need to submit.

- A PDF file titled `programming-assignment-1-msunetid.pdf` containing the following:
  - The answers for Q1, Q2, Q4.
  - Optional: What part of this homework was the most illuminating, and the part that was difficult?
- Your code file `layer.py` and `glove_model.py`

References

[1] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] Kawin Ethayarajh, David Duvenaud, and Graeme Hirst. Towards understanding linear word analogies. *arXiv preprint arXiv:1810.04882*, 2018.