Programming Assignment 1: Learning Distributed Word Representations

Version: 1.2
Changes by Version:
(v1.2)

- (1.3, 1.6) Change notation for gradient from $\frac{\partial L}{\partial \mathbf{W}}$ to $\nabla_{\mathbf{W}} L$
- (1.3, 1.5) Added additional hints

(v1.1)

- (1.2) Clarified operations that can be used
- (1.5) Reference the newly added GloVe gradient checker function
- (1.6) Clarified additional assumptions
- ("What you have to submit") Fixed colour reference to zero point questions

Version Release Date: 2022-02-02
Due Date: Friday, Feb. 4, at 11:59pm
Based on an assignment by George Dahl

Submission: You must submit two files through MarkUs[1]: (1) a PDF file containing your writeup, titled `a1-writeup.pdf`, which will be an export of the iPython notebook, and (2) your code file `a1-code.ipynb`. There will be sections in the notebook for you to write your responses. Your writeup must be typed. Make sure that the relevant outputs (e.g. `print_gradients()` outputs, plots, etc.) are included and clearly visible.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

The teaching assistants for this assignment are Caroline Malin-Mayor and Harris Chan. Post questions on Piazza[3] with the tag `pa1`. Private questions can be emailed to csc413-2022-01-tas@cs.toronto.edu with subject "*[CSC413] PA1 ...*".

[1] `https://markus.teach.cs.toronto.edu/2022-01/courses/16`
[2] `https://uoft-csc413.github.io/2022/assets/misc/syllabus.pdf`
[3] `http://piazza.com/utoronto.ca/winter2022/csc4132516`

Introduction

In this assignment we will learn about word embeddings and make neural networks learn about words. We could try to match statistics about the words, or we could train a network that takes a sequence of words as input and learns to predict the word that comes next.

This assignment will ask you to implement a linear embedding and then the backpropagation computations for a neural language model and then run some experiments to analyze the learned representation. The amount of code you have to write is very short but each line will require you to think very carefully. You will need to derive the updates mathematically, and then implement them using matrix and vector operations in NumPy.

Starter code and data

The starter code is at `https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/a1-code.ipynb`.

The starter helper function will download the specific the dataset from `http://www.cs.toronto.edu/~jba/a1_data.tar.gz`. Look at the file `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 have already 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
data = pickle.load(open('data.pk', 'rb'))
```

Now `data` is a Python dict which contains the vocabulary, as well as the inputs and targets for all three splits of the data. `data['vocab']` is a list of the 251 words in the dictionary; `data['vocab'][0]` is the word with index 0, and so on. `data['train_inputs']` is a $372,500 \times 4$ matrix where each row gives the indices of the 4 consecutive context words for one of the $372,500$ training cases. The validation and test sets are handled analogously.

Now look at the notebook ipynb file `a1-code.ipynb`, which contains the starter code for the assignment. Even though you only have to modify a few specific locations in the code, you may want to read through this code before starting the assignment.

1 Linear Embedding – GloVe (3pts)

In this section we will be implementing a simplified version of GloVe [Jeffrey Pennington and Manning]. Given a corpus with V distinct words, we define the co-occurrence matrix $X \in \mathbb{N}^{V \times V}$ with entries 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.*, $X_{ij} = X_{ji}$) if the order of the words do not matter, or *asymmetric* (*i.e.*, $X_{ij} \neq X_{ji}$) 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 $\mathbf{w}_i, \tilde{\mathbf{w}}_i \in \mathbb{R}^d$, as well as two scalar biases $b_i, \tilde{b}_i \in \mathbb{R}$. Typically we have the dimension of the embedding d much smaller than the number of words V. This objective can be written as [4]:

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

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

1.1 GloVe Parameter Count [0pt]

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.

1.2 Expression for the vectorized loss function [0.5pt]

In practice, we concatenate the V embedding vectors into matrices $\mathbf{W}, \tilde{\mathbf{W}} \in \mathbb{R}^{V \times d}$ and bias (column) vectors $\mathbf{b}, \tilde{\mathbf{b}} \in \mathbb{R}^V$, where V denotes the number of distinct words as described in the introduction. Rewrite the loss function L (Eq. 1) in a vectorized format in terms of $\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}, X$. You are allowed to use elementwise operations such as addition and subtraction as well as matrix operations such as the Frobenius norm and/or trace operator in your answer.

Hint: Use the all-ones column vector $\mathbf{1} = [1 \dots 1]^T \in \mathbb{R}^V$. *You can assume the bias vectors are column vectors, i.e. implicitly a matrix with* V *rows and 1 column:* $\mathbf{b}, \tilde{\mathbf{b}} \in \mathbb{R}^{V \times 1}$

1.3 Expression for the vectorized gradient $\nabla_{\mathbf{W}} L$ [0.5pt]

Write the vectorized expression for $\nabla_{\mathbf{W}} L$, the gradient of the loss function L with respect to the embedding matrix $\mathbf{W}$. The gradient should be a function of $\mathbf{W}, \tilde{\mathbf{W}}, \mathbf{b}, \tilde{\mathbf{b}}, X$.

Hint: Make sure that the shape of the gradient is equivalent to the shape of the matrix. You can use the all-ones vector as in the previous question.

Update (v1.2): *Hint: Property 119 in the matrix cookbook*[5] *may be useful. Be careful with transpose.*

[4] We have simplified the objective by omitting the weighting function. For the complete algorithm please see [Jeffrey Pennington and Manning]

[5] `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf`

1.4 Implement Vectorized Loss Function [1pt]

Implement the `loss_GloVe()` function of GloVe in `a1-code.ipynb`. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. Note that you need to implement both the loss for an *asymmetric* model (from your answer in question 1.2) and the loss for a *symmetric* model which uses the same embedding matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ for the first and second word in the co-occurrence, i.e. $\tilde{\mathbf{W}} = \mathbf{W}$ and $\tilde{\mathbf{b}} = \mathbf{b}$ in the original loss.

Hint: You may take advantage of NumPy's broadcasting feature[6] *for the bias vectors*

1.5 Implement the gradient update of GloVe [1pt]

Implement the `grad_GloVe()` function which computes the gradient of GloVe in `a1-code.ipynb`. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. Again, note that you need to implement the gradient for both the symmetric and asymmetric models.

Update (v1.1): We added a gradient checker function using finite difference called `check_GloVe_gradients()`. You can run the specified cell in the notebook to check your gradient implementation for both the symmetric and asymmetric models before moving forward.

Once you have implemented the gradient, run the following cell marked by the comment `### TODO: Run this cell ###` in order to train an asymmetric and symmetric GloVe model. The code will plot a figure containing the training and validation loss for the two models over the course of training. **Include this plot in your write up**.

Update (v1.2): *Hint: In the symmetric model case, you can use what you have derived for the asymmetric model case. For example, consider a function* $f(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$*, where* $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$*. If we define* $\mathbf{a} = \mathbf{x}$ *and* $\mathbf{b} = \mathbf{x}$*, where* $\mathbf{b} \in \mathbb{R}^d$*, then*

$$\begin{aligned} \nabla_{\mathbf{x}} f &= \nabla_{\mathbf{a}} f + \nabla_{\mathbf{b}} f && (2)\\ &= \mathbf{b} + \mathbf{a} && (3)\\ &= \mathbf{x} + \mathbf{x} && (4)\\ &= 2\mathbf{x} && (5) \end{aligned}$$

1.6 Effects of a buggy implementation [0pt]

Suppose that during the implementation, you initialized the weight embedding matrix $\mathbf{W}$ and $\tilde{\mathbf{W}}$ with the same initial values (i.e., $\mathbf{W} = \tilde{\mathbf{W}} = \mathbf{W}_0$). The bias vectors were also initialized the same, i.e., $\mathbf{b} = \tilde{\mathbf{b}} = \mathbf{b}_0$. Assume also that in this case, the co-occurrence matrix is also symmetric: $X_{ij} = X_{ji}$

What will happen to the values of $\mathbf{W}$ and $\tilde{\mathbf{W}}$ over the course of training? Will they stay equal to each other, or diverge from each other? Explain your answer briefly.

Hint: Consider the gradient $\nabla_{\mathbf{W}} L$ *versus* $\nabla_{\tilde{\mathbf{W}}} L$

[6] https://numpy.org/doc/stable/user/basics.broadcasting.html

1.7 Effects of embedding dimension [0pt]

Train the both the symmetric and asymmetric GloVe model with varying dimensionality d. Comment on the results:

1. Which d leads to optimal validation performance for the asymmetric and symmetric models?
2. Why does / doesn't larger d always lead to better validation error?
3. Which model is performing better (asymmetric or symmetric), and why?

2 Neural Language Model Network architecture (1pt)

In this assignment, we will train a neural language model like the one we covered in lecture and as in Bengio et al. [2003]. However, we will modify the architecture slightly, inspired by the Masked Language Modeling (MLM) objective introduced in BERT [Devlin et al., 2018]. The network takes in N consecutive words, where one of the words is replaced with a `[MASK]` token[7]. The aim of the network is to predict the masked word in the corresponding output location. See Figure 1 for the diagram of this architecture.

[7] In the original BERT paper, they mask out 15% of the tokens randomly.

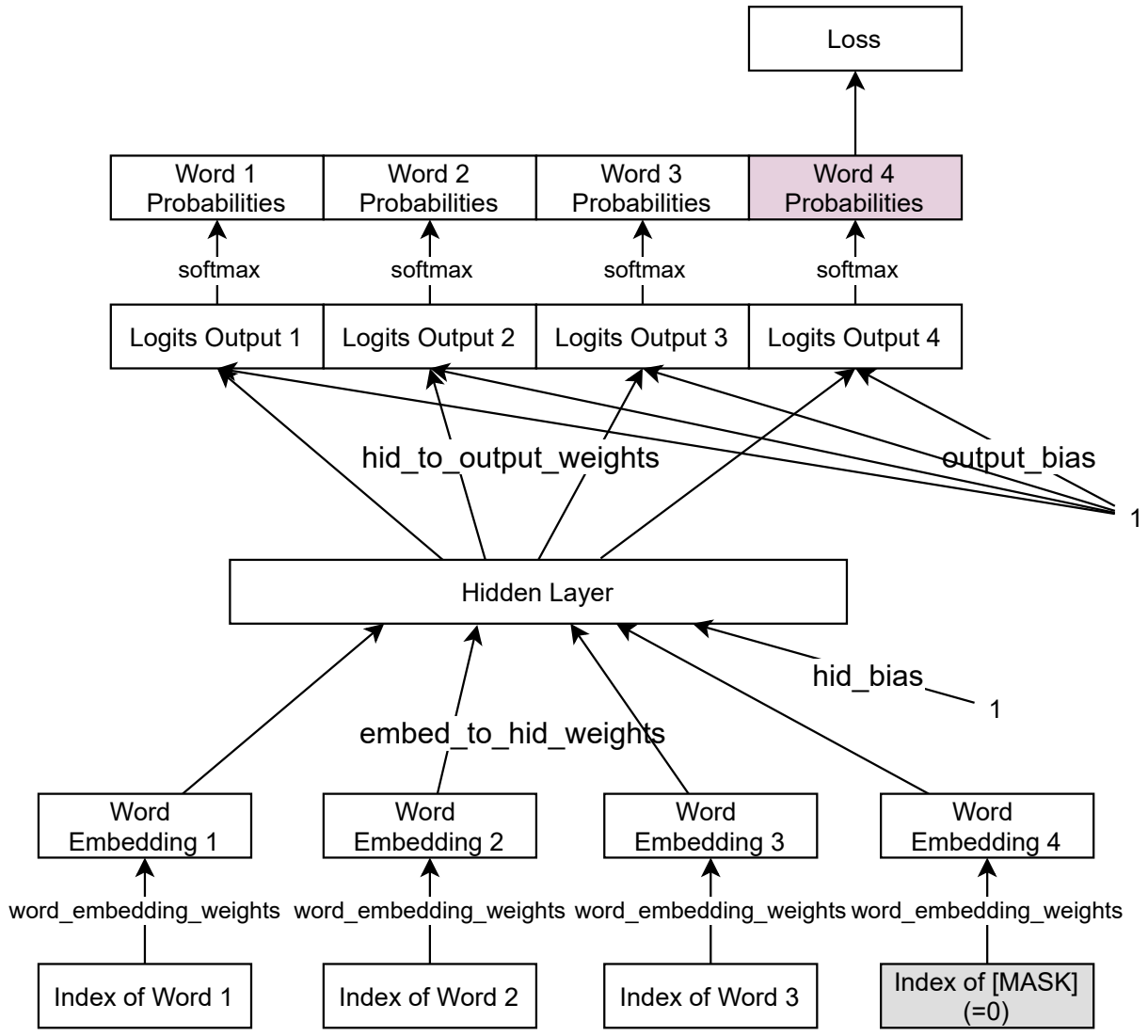

Figure 1: A simplified architecture with N words input and N words output. During training, we mask out one of the input words by replacing it with a `[MASK]` token, and try to predict the masked out word in the corresponding position in the output. Only that output position is used in the cross entropy loss.

The network consists of an input layer, embedding layer, hidden layer and output layer. The input consists of a sequence of N consecutive words, with each word given as integer valued indices. (*e.g.*, the 250 words in our dictionary are arbitrarily assigned integer values from 0 to 249.) The embedding layer maps each word to its corresponding vector representation. Each of the N context words are mapped independently using the same `word_embedding_weights` matrix. The embedding layer has $N \times D$ units, where D is the embedding dimension of a single word. The embedding layer is fully connected to the hidden layer with H units, which uses a logistic nonlinearity. The hidden layer in turn is connected to the logits output layer, which has $N \times V$ units. Finally, softmax over V logit output units is applied to each consecutive V logit output units, where V is the number of words in the dictionary (including the `[MASK]` token).[8]

2.1 Number of parameters in neural network model [0.5pt]

The trainable parameters of the model consist of 3 weight matrices and 2 sets of biases. 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 $V \gg H > D > N$?[9] Explain your reasoning.

2.2 Number of parameters in n-gram model [0.5pt]

Another method for predicting the next words is an *n-gram model*, which was mentioned in Lecture 3[10]. If we wanted to use an n-gram model with the same context length $N - 1$ as our network[11], we'd need to store the counts of all possible N-grams. If we stored all the counts explicitly and suppose that we have V words in the dictionary, how many entries would this table have?

2.3 Comparing neural network and n-gram model scaling [0pt]

How do the parameters in the neural network model scale with the number of context words N versus how the number of entries in the n-gram model scale with N? Which model has a more compact representation for the words?

3 Training the Neural Network (2pts)

In this part, you will learn to implement and train the neural language model from Figure 1. As described in the previous section, during training, we randomly sample one of the N context words

[8] For simplicity we will include the `[MASK]` token in the output softmax as well.

[9] The symbol $\gg$ means "much greater than"

[10] https://uoft-csc413.github.io/2022/assets/slides/lec03.pdf

[11] Since we mask 1 of the N words in our input

to replace with a `[MASK]` token. The goal is for the network to predict the word that was masked, at the corresponding output word position. In our implementation, this `[MASK]` token is assigned the index 0 in our dictionary. In practice, it is more efficient to take advantage of parallel computing hardware, such as GPUs, to speed up training. Instead of predicting one word at a time, we achieve parallelism by lumping N output words into a single output vector that can be computed by a single matrix product. Now, the hidden-to-output weight matrix `hid_to_output_weights` has the shape $NV \times H$, 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. Note here that the softmax is applied in chunks of V as well, to give a valid probability distribution over the V words[12]. Only the output word positions that were masked in the input are included in the cross entropy loss calculation:

$$C = -\sum_{i}^{B}\sum_{n}^{N}\sum_{v}^{V} m_n^{(i)}(t_{v+nV}^{(i)} \log y_{v+nV}^{(i)}) \tag{6}$$

Where:

- $y_{v+nV}^{(i)}$ denotes the output probability prediction from the neural network for the i-th training example for the word v in the n-th output word. Denoting z as the output logits, we define the output probability y as a softmax on z over contiguous chunks of V units (see also Figure 1):

$$y_{v+nV}^{(i)} = \frac{e^{z_{v+nV}^{(i)}}}{\sum_l^V e^{z_{l+nV}^{(i)}}} \tag{7}$$

- $t_{v+nV}^{(i)} \in \{0, 1\}$ is 1 if for the i-th training example, the word v is the n-th word in context
- $m_n^{(i)} \in \{0, 1\}$ 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

Now, you are ready to complete the implementation in the notebook `https://colab.research.google.com/github/uoft-csc413/2022/blob/master/assets/assignments/a1-code.ipynb`, you will implement a method which computes the gradient using backpropagation. The `Model` class contains several important methods used in training:

- `compute_activations` computes the activations of all units on a given input batch
- `compute_loss_derivative` computes the gradient with respect to the output logits $\frac{\partial C}{\partial z}$

[12]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

- `evaluate` computes the average cross-entropy loss for a given set of inputs and targets

You will need to complete the implementation of two additional methods to complete the training, and print the outputs of the gradients.

3.1 Implement Vectorized Loss [0.5pt]

Implement a vectorized `compute_loss` function, which computes the total cross-entropy loss on a mini-batch according to Eq. 6. Look for the `## YOUR CODE HERE ##` comment for where to complete the code. The docstring provides a description of the inputs to the function.

3.2 Implement gradient with respect to parameters [1pt]

`back_propagate` is the function which computes the gradient of the loss with respect to model parameters using backpropagation. It uses the derivatives computed by `compute_loss_derivative`. Some parts are already filled in for you, but you need to compute the matrices of derivatives for `embed_to_hid_weights` and `output_bias`. These matrices have the same sizes as the parameter matrices. Look for the `## YOUR CODE HERE ##` comment for where to complete the code.

In order to implement backpropagation efficiently, you need to express the computations in terms of matrix operations, rather than `for` loops. You should first work through the derivatives on pencil and paper. First, apply the chain rule to compute the derivatives with respect to individual units, weights, and biases. Next, take the formulas you've derived, and express them in matrix form. You should be able to express all of the required computations using only matrix multiplication, matrix transpose, and element-wise operations — no `for` loops! If you want inspiration, read through the code for `Model.compute_activations` and try to understand how the matrix operations correspond to the computations performed by all the units in the network.

Hints: Your implementations should also be similar to `hid_to_output_weights_grad`, `hid_bias_grad` *in the same function call.*

3.3 Print the gradients [0.5pt]

To make your life easier, we have provided the routine `check_gradients`, which checks your gradients using finite differences. You should make sure this check passes (prints OK for the various parameters) before continuing with the assignment. Once `check_gradients` passes, call `print_gradients` and include its output in your write-up.

3.4 Run model training [0 pt]

Once you've implemented the gradient computation, you'll need to train the model. The function `train` in `a1-code.ipynb` implements the main training procedure. It takes two arguments:

- `embedding_dim`: The number of dimensions in the distributed representation.

- `num_hid`: The number of hidden units

For example, execute the following:

```
model = train(16, 128)
```

As the model trains, the script prints out some numbers that tell you how well the training is going. It shows:

- The cross entropy on the last 100 mini-batches of the training set. This is shown after every 100 mini-batches.
- The cross entropy on the entire validation set every 1000 mini-batches of training.

At the end of training, this function shows the cross entropies on the training, validation and test sets. It will return a `Model` instance.

4 Bias in Word Embeddings (2pts)

Unfortunately, stereotypes and prejudices are often reflected in the outputs of natural language processing algorithms. For example, Google Translate is more likely to translate a non-English sentence to "*He* is a doctor" than "*She* is a doctor" when the sentence is ambiguous. In this section, you will explore how bias[13] enters natural language processing algorithms by implementing and analyzing a popular method for measuring bias in word embeddings.

4.1 WEAT method for detecting bias [1pt]

Word embedding models such as GloVe attempt to learn a vector space where semantically similar words are clustered close together. However, they have been shown to learn problematic associations, e.g. by embedding "man" more closely to "doctor" than "woman" (and vice versa for "nurse"). To detect such biases in word embeddings, Caliskan et al. [2017] introduced the Word Embedding Association Test (WEAT). The WEAT test measures whether two *target* word sets (e.g. {programmer, engineer, scientist, ...} and {nurse, teacher, librarian, ...}) have the same relative association to two *attribute* word sets (e.g. {man, male, ...} and {woman, female ...}).[14]

Formally, let A, B be two sets of attribute words. Then

$$s(w, A, B) = \text{mean}_{a \in A} \cos(\vec{w}, \vec{a}) - \text{mean}_{b \in B} \cos(\vec{w}, \vec{b}) \tag{8}$$

[13] In AI and machine learning, **bias** generally refers to prior information, a necessary prerequisite for intelligent action. However, bias can be problematic when it is derived from aspects of human culture known to lead to harmful behaviour, such as stereotypes and prejudices.

[14] There is an excellent blog on bias in word embeddings and the WEAT test at `https://developers.googleblog.com/2018/04/text-embedding-models-contain-bias.html`

measures the association of a target word w with the attribute sets - for convenience, we will call this the WEAT association score. A positive score means that the word w is more associated with A, while a negative score means the opposite. For example, a WEAT association score of 1 in the following test $s(\text{“}programmer\text{”}, \{man\}, \{woman\}) = 1$, implies the “programmer” has a stronger association to $\{man\}$. For reference, the cosine similarity between two word vectors $\vec{a}$ and $\vec{b}$ is given by:

$$\cos(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\| \vec{a} \| \| \vec{b} \|} \tag{9}$$

In the notebook, we have provided example target words (in sets X and Y) and attribute words (in sets A and B). You must implement the function `weat_association_score()` and compute the WEAT association score for each target word.

4.2 Reasons for bias in word embeddings [0pt]

Based on the results of the WEAT test, do the pretrained word embeddings associate certain occupations with one gender more than another? What might cause word embedding models to learn certain stereotypes and prejudices? How might this be a problem in downstream applications?

4.3 Analyzing WEAT

While WEAT makes intuitive sense by asserting that closeness in the embedding space indicates greater similarity, more recent work Ethayarajh et al. [2019] has further analyzed the mathematical assertions and found some drawbacks this method. Analyzing edge cases is a good way to find logical inconsistencies with any algorithm, and WEAT in particular can behave strangely when A and B contain just one word each.

4.3.1 1-word subsets [0.5pts]

In the notebook, you are asked to find 1-word subsets of the original A and B that reverse the association between some of the occupations and the gendered attributes (change the sign of the WEAT score).

4.3.2 How word frequency affects embedding similarity [0.5pts]

Next, consider this fact about word embeddings, which has been verified empirically and theoretically: The squared norm of a word embedding is linear in the log probability of the word in the training corpus. In other words, the more common a word is in the training corpus, the larger the norm of its word embedding. Following this fact, we will show how one may exploit the WEAT

association score for a specific word embedding model. Let us start with three word embedding vectors: a target word $\mathbf{w}_i$ and two attributes $\{\mathbf{w}_j\}$, $\{\mathbf{w}_k\}$.

$$\begin{aligned} s(\mathbf{w}_i, \{\mathbf{w}_j\}, \{\mathbf{w}_k\}) &= cos(\mathbf{w}_i, \mathbf{w}_j) - cos(\mathbf{w}_i, \mathbf{w}_k) \\ &= \frac{\mathbf{w}_i^\top \mathbf{w}_j}{\|\mathbf{w}_i\|\|\mathbf{w}_j\|} - \frac{\mathbf{w}_i^\top \mathbf{w}_k}{\|\mathbf{w}_i\|\|\mathbf{w}_k\|} \end{aligned}$$

Remember the GloVe embedding training objective from Part 1 in this assignment. Assume tied weights and ignore the bias units, we can write the training loss as following:

$$\text{Simplified GloVe} \quad L(\{\mathbf{w}_i\}_{i=1}^V) = \sum_{i,j=1}^{V} (\mathbf{w}_i^\top \mathbf{w}_j - \log X_{ij})^2,$$

where X_{ij} denotes the number of times word i and word j co-occured together in the training corpus. In the special case, X_{ii} denotes the number of times word i appeared in the corpus. When this model reaches zero training loss, the inner product of the GloVe embedding vectors will simply equal to the entries in the log co-occurrence matrix $\log X$. We can then express the WEAT association score in terms of the original co-occurrence matrix:

$$s(\mathbf{w}_i, \{\mathbf{w}_j\}, \{\mathbf{w}_k\}) = \frac{1}{\sqrt{logX_{ii}}} \left(\frac{logX_{ij}}{\sqrt{logX_{jj}}} - \frac{logX_{ik}}{\sqrt{logX_{kk}}} \right)$$

Briefly explain how this fact might contribute to the results from the previous section when using different attribute words. Provide your answers in no more than three sentences.

Hint: The paper cited above is a great resource if you are stuck.

4.3.3 Relative association between two sets of target words [0 pts]

In the original WEAT paper, the authors do not examine the association of individual words with attributes, but rather compare the relative association of two sets of target words. For example, are insect words more associated with positive attributes or negative attributes than flower words.

Formally, let X and Y be two sets of target words of equal size. The WEAT test statistic is given by:

$$s(X, Y, A, B) = \sum_{x \in X} s(x, A, B) - \sum_{y \in Y} s(y, A, B) \tag{10}$$

Will the same technique from the previous section work to manipulate this test statistic as well? Provide your answer in no more than 3 sentences.

What you have to submit

For reference, here is everything you need to hand in. The zero point questions (in black below) will not be graded, but you are more than welcome to include your answers for these as well in the submission. See the top of this handout for submission directions.

- A PDF file titled `a1-writeup.pdf` containing the following:
  - Part 1: Questions 1.1,1.2, 1.3, 1.6, 1.7. Completed code for `loss_GloVe()` (1.4) and `grad_GloVe()` function and output plot in 1.5.
  - Part 2: Questions 2.1, 2.2, 2.3.
  - Part 3: Completed code for `compute_loss()` (3.1), `back_propagate()` (3.2) functions, and the output of `print_gradients()` (3.3)
  - Part 4: Questions 4.2, 4.3.2, 4.3.3. Completed code for `weat_association_score()`, and outputs (4.1), 1-word subsets with outputs (4.3.1).
- Your code file `a1-code.ipynb`

References

Richard Socher Jeffrey Pennington and Christopher D Manning. Glove: Global vectors for word representation. Citeseer.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

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.

Aylin Caliskan, Joanna J Bryson, and Arvind Narayanan. Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186, 2017.

Kawin Ethayarajh, David Duvenaud, and Graeme Hirst. Understanding undesirable word embedding associations. *arXiv preprint arXiv:1908.06361*, 2019.