Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 5

Due: February 27, 2025 (100 points)

YOUR NAME + LDAP

There are several word embedding vector approaches. In this homework, we will be exploring three of them: (1) collaboratively filtered (SVD) embeddings, (2) the bag of words neural network, and the popular `word2vec` algorithm.

The Dataset: We will make use of nearly 3 million arXiv papers that currently reside on http://arxiv.org. This set of papers has served the scientific community for over 30 years, and the subject matter is diverse, ranging from physics, math, computational biology, and yes, machine learning and artificial intelligence. The dataset `arxiv-titles-dataset.txt` that I provide is *not* shuffled presently; it is ordered according to topics, and it is formatted where each line in the text file is the title of a particular paper.

Question 1: Pre-Processing

The dataset is downloaded directly from `http://arxiv.org`, and is unprocessed, except that I have extracted only the titles, and each line is the unfiltered title of the text.
First, we will create the vocabulary of words and the frequency of their occurrences that we can sample from. Since many of the scientific words are not used frequently, we can cut off those that do not appear in the corpus enough and those that appear too many times. The remaining words will be in our vocabulary and the word frequencies, the first of our return values (`word_freqs`). You decide what these thresholds are; there are no wrong answers. but you will notice a tradeoff between accuracy and computation time.

Second, once we have built our word distributions and determined the appropriate vocabulary, it is beneficial to create an alternative dataset without the out of vocabulary (OOV) words. Fortunately, the dataset is small enough to fit in memory, where each title is an element of a list. Therefore, the second return value (`dataset`) will be a list of titles, where OOV are words removed. Also, a title isn't all that informative without enough words, and so we've added a `min_title_len` to filter out short titles. In later sections, you'll be using a window to look

for context words, which will need to be less than or equal to `min_title_len`.

Write the function `process_data`. Use the below function signature. The file that we'll be processing is the `arxiv-titles.txt`, where each line is the title of a scientific article, which will constitute a single training sample that we will sample from in the subsequent questions.

```
def process_data(filename, min_cnt, max_cnt, min_win = 5, min_letters = 3):
    '''
    Preprocesses and builds the distribution of words in sorted order
    (from maximum occurrence to minimum occurrence) after reading the
    file. Preprocessing will include filtering out:
    * words that have non-letters in them,
    * words that are too short (under minletters)

    Arguments:
    * filename: name of file
    * min_cnt: min occurrence of words to include
    * max_cnt: max occurrence of words to include
    * min_win: minimum number of words in a title after word filtering
    * min_letters: min length of words to include (3)

    Returns:
    * word_freqs: A sorted (max to min) list of tuples of form -
       [(word1, count1), (wordN, countN), ... (wordN, countN)]
    * dataset: A list of strings with OOV words removed -
       ["this is title 1", "this is title 2", ...]
    '''
```

You can test your function out with the following code.

```
word_freqs, dataset = process_data("arxiv_titles.txt", 10, 150000)
```

Plot the distribution of words. This is called a Zipfian distribution, and is common in natural language processing. (Zipf's law is an empirical law stating that when a list of measured values is sorted in decreasing order, the value of the n^{th} entry is often approximately inversely proportional to n.)

Submission Artifacts: `process_data`, `word_freqs`, distribution plot.

Note: In later sections, we will be using a window when sampling each title. It is acceptable for the window size to be effectively larger than it technically should be. For example, consider the sentence: "Many articles of note specify a co-author in the same department", where we've specified a window of size 3 and sampled so that the center word is "note". Let us say that the OOV words are {"of", "a", "in", "the"}. Then, removing OOV words yields "many articles note specify co-author", and it is acceptable to have a context window of 3 surrounding "note" include the words "articles" and "specify".

Question 2: Matrix Factorization

One of the easiest ways to create dense representations of words is by using matrix factorization. In this question, we will need to create an adjacency matrix and then subsequently factorize it.

Q 2.1: Create an Adjacency Matrix

Create an adjacency matrix where entry (i, j) is the number of times word i co-occurs with word j in a single article title within a specified maximum window size (defaulted to ten). The maximum window size is simply used to make computation feasible, where we would only increment a count if i and j are both within `max_win`. Assuming you have built the word distributions, you will need to create and pass in `word2index` as a dictionary of vocabulary words and their indices. We will expect the adjacency matrix to be in the order specified by `word2index`, the dictionary that assigns a word to the index.

```
def create_adjacency(dataset, word2index, win = 10):
    '''
    Builds an adjacency matrix based on word co-occurrence within a window.

    Args:
        dataset: List of processed titles
        word2index: Dictionary mapping word to index
        win: The window size for co-occurrence.

    Returns:
        adjacency_matrix: A NumPy array representing the adjacency matrix.
    '''
```

Submission Artifacts: `create_adjacency`

Q 2.2: Create SVD Word Vectors

Using the adjacency matrix calculated in Q2.1 and parameters specifying the minimum and maximum index of the singular values, write a function that creates an embedding space with a specified number of components, determined by `embedding_dim` = `max_index` - `max_index`. Now, using the, we can create word embeddings. You may use any `numpy` or `scipy` library for the matrix decomposition (e.g., `eig`, `svd`, or `svds`). For example,

```
from scipy.sparse.linalg import svds

def train_svd(adjacency_matrix, min_index = 3, max_index = 103):
    """
    Creates an embedding space using SVD on the adjacency matrix.

    Args:
        adjacency_matrix: The adjacency matrix.
        embedding_dim: The desired dimensionality of the embedding space.

    Returns:
        A NumPy array representing the embedding space (num_words x embedding_dim)
    """
```

After training, save out your vectors into `assignment5.pkl` as `V_svd`. In `assignment5.pdf`, print out the ten nearest neighbors to "neural", "machine", "dark", "string", and "black".

Submission Artifacts: `train_svd`, `V_svd`, nearest neighbors to specified words

Note: The larger the matrix you have, the longer the decomposition will take. Test your word vectors out by finding the nearest neighbors to any given word. For example, one of the closest words to the word "*neural*" should "*networks*". You will save these word embeddings to a file, labeled `V_svd`.

Question 3: Bag of Words

For this question, we will be using a traditional neural network to predict the word context with a full cross-entropy classification cost function. We will use a simple, two layer neural network. You can use your own code from a previous homework, or you can use a library like `Torch` or `Tensorflow`.

Q 3.1: Create a Window Sampler

For any given word in our vector space, we will say that it is close to the words that surround it, i.e., its *context*, and far away from the words that don't commonly appear with it. To do this, we need to encode **the word** and **its context** as input and output vectors. For this, we will use a one-hot vector for any word and multi-hot vectors for its context, which we define as the window surrounding the word. In this question, you will write a sampling function that returns the input and output vectors, randomly sampled from the titles.

One Hot and Multi-Hot Encoding

The most common way to encode words are sparse one-hot encodings, where the representation of a word is a $|V|$-dimensional vector of all zeros except for the index corresponding to the word, where the value is one. For example, let's say that I have a small vocabulary that consists of the following words, organized according to the following indices.

index	0	1	2	3	4	5	6
word	bee	comes	flower	nectar	flies	back	honey

Then, the sparse one-hot encoding of the word `nectar` can be written as follows.

$$\mathbf{x} = [0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0]^T \tag{3.1}$$

If we want to encode *multiple* words, then the *multi*-hot encoding of the multiple words {flower, flies} with the above vocabulary is

$$\mathbf{y} = [0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0]^T \tag{3.2}$$

For our sampler, we would like two vectors: a word (as a one-hot vector) and its context (as a multi-hot vector) from a window. (We *could* say that the entire title in `arxiv_titles.txt` is the context, but that would be too computationally inefficient and could take a long time optimizing.) For words that do not appear in the vocabulary, we do not need to encode them.

Let's say we are randomly sampling a window of size 5 from the sentence below with the above dictionary.

```
The bee comes to the flower for nectar and flies back to make honey.
```

If we land on `nectar`, then the word and context vectors will be as described in (3.1) and (3.2), respectively.

Let's say that we've read in the data into a list of strings, and the vocabulary, can you write a function that randomly samples a title, randomly samples a window inside that title (check if the window size is odd), and returns a one-hot vector of the center word and multi-hot vector of the words surrounding it (excluding the center word)? You can ignore words that are not in the vocabulary.

```
def sample_bow(dataset, word2index, win=5):
    '''
    Randomly samples a title and a window within that title, returning
    one-hot and multi-hot vectors.

    Args:
        dataset: A list of preprocessed titles.
        word2index: A dictionary of words and their indices
        win: The size of the context window.

    Returns:
        content_vector: the one-hot vector for the center word
        context_vector: multi-hot vector for window around center
    '''
```

Submission Artifacts: `sample_bow`

Q 3.2: Create BOW Word Vectors

For this question, we will train and evaluate the neural network shown in Fig 3.1. To keep things simple, do *not* use a bias vector for the first dense layer. For the neural network itself, feel free to use the code that you wrote in an earlier assignment (in Python) or any library you would like, but make sure to plot out your loss function for both training and validation / test sets to ensure that you are not overfitting. You don't need to use mini-batches; you can just do simple stochastic gradient descent. Please use the sampler created in Q3.1.

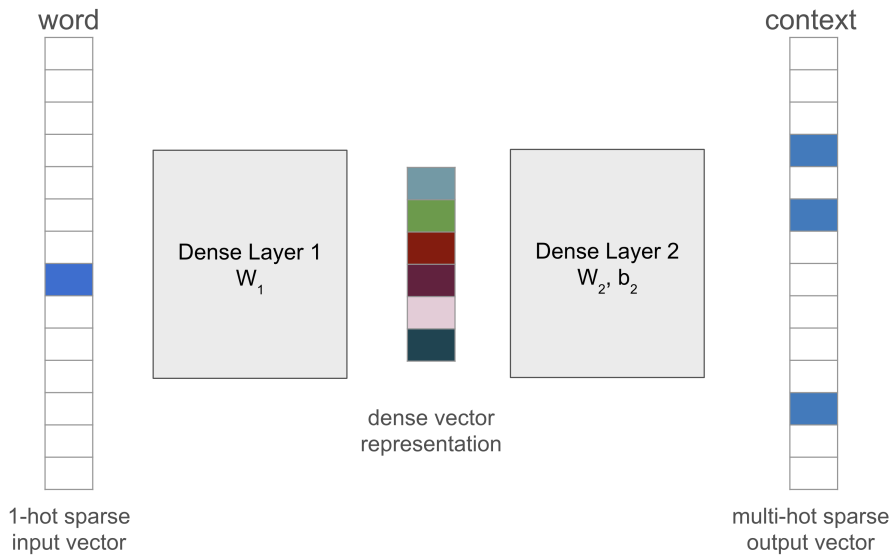

Figure 3.1: Bag of Words Neural Network

```
def train_bow(dataset, word2index, iters,
              win = 5, embedding_dim = 100, learning_rate=0.01):
    """
    Creates an embedding space using SVD on the adjacency matrix.

    Args:
        dataset: A list of preprocessed titles.
        word2index: Dictinoary assigning word to index
        win: The size of the context window for sampling.
        iters: Number of iterations to run for
        embedding_dim: The desired dimensionality of the embedding space.
        learning_rate: Learning rate or any other DNN params with defaults.
                       The autograder won't touch this.

    Returns:
        V_bow: an array representing the embedding space (num_words x
               embedding_dim)
        List of losses (to print out)
    """
```

When you've finished, your word embeddings will be the weight matrix W_1, which you will need to submit along with your code and loss function plots for training, evaluation, and testing. Why can you use the columns (or rows, depending on your implementation) as your word embeddings? Put your explanation in `assignment5.pdf`. Save out your vectors into `assignment5.pkl` as `V_bow`, taking care not to overwrite `V_svd`. In `assignment5.pdf`, print out the ten nearest neighbors to "neural", "machine", "dark", "string", and "black".

Submission Artifacts: `train_bow`, `V_bow`, loss function plot, answer to why W_1 can be used as the set of word vectors you saved (i.e., `V_bow`), nearest neighbors to specified words

Question 4: Word2Vec

In this question, we are going to implement the popular word2vec paper, which originally was written in C. There are two main components that make `word2vec` an effective optimization. These two components comprise the curricula for this week's homework assignment:

1. **Positive and Negative Sampling**
   - Skip-gram positive sampling from context.
   - Negative sampling from the word distribution
2. **Optimizing vectors in matrices V_i and V_o**
   - You will maximize similarity between v_i and v_o.
   - You will maximize the distance between v_i and each v_n

In *word2vec*, we optimize two sets of vectors, which we store in matrices V_i and V_o, both of size $\mathbb{R}^{d \times |V|}$, where d is a chosen dimensionality of the vectors (e.g., 100) and $|V|$ is the size of the vocabulary. The **first set of vectors** is stored in the input matrix V_i, where each column $\mathbf{v}_i$ is a vector sampled from target words. When training / optimizing, we will be randomly sampling titles from the dataset, and then we'll randomly sample any word in the title to obtain the target vector $\mathbf{v}_i$. The **second set of vectors** is stored in the output matrix V_o, whose columns comprise the context that complements the target words. For each randomly sampled target vector, you will also randomly sample both a positive context vector $\mathbf{v}_o$ within a window of the target word and multiple negative irrelevant vectors $\{\mathbf{v}_n\}$ from this matrix.

In order to optimize vectors in the matrices V_i and V_o, we will be pulling those vectors belonging to words that we say are similar (i.e., in a given window) closer to each other closer. Simultaneously, we will be pushing those vectors belonging to words that we say are dissimilar (any word from the distribution of all words) further apart. This process is a much simpler version of *noise contrastive estimation*, because we are getting signal by contrasting our targets with noise.

Because we are deriving gradients, you may use `numpy` only. Please *do not use any special libraries* like `Torch` or `Tensorflow`.

Q 4.1: Negative Sampling

For this question, we will write `sample_w2v` to prepare us for training the `word2vec` algorithm. We will need to sample the target word, a single word from the window, i.e. the positive context, and multiple *negative samples*, i.e. the negative context. In the original paper, we obtain the negative samples from a function of the distribution of words *excluding* the words in the window, notably $U(w)^{\frac{3}{4}}$ for $w \notin W(\text{target})$. The function they use is the distribution you calculated in Q1 normalized to one raised to the $\frac{3}{4}$ power. In practice for this homework and for expediency, we can sample from the entire uniform distribution (including the target and words in the window), since the sampled words are unlikely to contain the positive samples.

```
def sample_w2v(data, word2index, neg_samples=5, win=10):
    '''
    Randomly samples a title and a window within that title, returning
    one-hot and multi-hot vectors.
```

```
    Args:
        dataset: A list of preprocessed titles.
        word2index: A dictionary of words and their indices
        neg_samples: Number of negative samples
        win: The size of the context window.

    Returns:
        wi - target vector index
        wo - context vector index
        Wn - negative vectors index
    '''
```

Submission Artifacts: `sample_w2v`

Q 4.2: Gradient Derivation

In Mikolov's original paper in equation (4), the loss function for `word2vec` can be written as follows:

$$\mathcal{J}(\mathbf{v}_i, \mathbf{v}_o, \{\mathbf{v}_n\}) = \log \sigma(\mathbf{v}_i^T \mathbf{v}_o) + \sum_{\mathbf{v}_n \in P_n}^{k} \log \sigma(-\mathbf{v}_i^T \mathbf{v}_n) \tag{4.1}$$

where the vector corresponding to the target word is $\mathbf{v}_i$, the vector corresponding to a word in the target's context is $\mathbf{v}_o$, and there are k vectors $\mathbf{v}_n$ that have been negatively sampled from the words that don't appear in the target word's context, i.e., $w_n \sim P_n(w)$. (In your case $P_n(w) = U(w)$, but that's irrelevant to your derivations.) Derive the gradients with respect to $\boldsymbol{v}_i$ and $\boldsymbol{v}_o$, for any given i and o.

Write your solution to:

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial \mathbf{v}_i} &= ?? \\ \frac{\partial \mathcal{J}}{\partial \mathbf{v}_o} &= ?? \\ \frac{\partial \mathcal{J}}{\partial \mathbf{v}_n} &= ??, \quad \forall n \end{aligned} \tag{4.2}$$

Submission Artifacts: Derivations for $\frac{\partial \mathcal{J}}{\partial \mathbf{v}_i}$, $\frac{\partial \mathcal{J}}{\partial \mathbf{v}_o}$, $\frac{\partial \mathcal{J}}{\partial \mathbf{v}_n}$

Q 4.3: Gradient Implementation

Write a function that calculates the gradients of $\mathbf{v}_i$, $\mathbf{v}_o$, and the relevant $\mathbf{v}_n$'s. Recall that $\mathbf{v}_i \in V_i$ and both $\mathbf{v}_o$ and set of $\mathbf{v}_n$'s are all in V_n. Please use the following signature for our auto-graders.

```
def w2vgrads( vi, vo, Vns ):
  """
```

```
  This function implements the gradient for all vectors in
  input matrix Vi and output matrix Vo.

  Args:
    vi:  Vector of shape (d,), a sample in the input word
         vector matrix
    vo:  Vector of shape (d,), a positive sample in the output
         word vector matrix
    vns: Vector of shape (d, k), k negative samples in the
         output word vector matrix

  Returns:
    dvi, dvo, dVns: the gradients of J with respect to vi
                    and vo.
  """
```

Submission Artifacts: `w2v_grads`

Q 4.4: Create W2V Embeddings

For this question, we will be doing gradient ascent for several epochs. You will need to use the functions that you wrote in Q4.1 and Q4.3, i.e. your sampler and gradient implementation, effectively applying gradients to V_i and V_o for the samples that you've randomly obtained.

```
def train_w2v(dataset, word2index, iters, negsamps = 5,
              win = 5, embedding_dim = 100, learning_rate=0.01):
    """
    Creates an embedding space using SVD on the adjacency matrix.

    Args:
        dataset: A list of preprocessed titles.
        word2index: Dictinoary assigning word to index
        iters: Number of iterations to run for
        negsamps: Number of negative samples
        win: The size of the context window for sampling.
        embedding_dim: The desired dimensionality of the embedding space.
        learning_rate: Learning rate or any other DNN params with defaults.
                       The autograder won't touch this.

    Returns:
        V_w2v: an array representing the embedding space (num_words x
               embedding_dim)
        List of losses (to print out)
    """
```

You will note that this is *much* faster than your optimization in Q3. Play with your learning rate and see what works. Keep can track your objective function $\mathcal{J}$ by plotting it every so often. To keep computations at a minimum during optimization, one trick is to plot only the objective function for the positive portion of $\mathcal{J}$. The word vectors are those stored

in V_i; go ahead and qualitatively analyze them with nearest neighbors of words. Save your vectors into `assignment5.pkl` as `V_w2v`, taking care not to overwrite `V_bow` and `V_svd`. In `assignment5.pdf`, print out the ten nearest neighbors to "neural", "machine", "dark", "string", and "black".

Submission Artifacts: `train_w2v`, `V_w2v`, loss function plot, nearest neighbors to specified words

Submission Instructions

There are several artifacts that you will need to upload to Gradescope. You can consolidate these into three files total. These include:

Code All your code saved in a Python file called `assignment5.py`. We will be expecting multiple different functions in `assignment.py`, including:

- Q1 - `process_data`
- Q2.1 - `create_adjacency`
- Q2.2 - `train_svd`
- Q3.1 - `sample_bow`
- Q3.2 - `train_bow`
- Q4.1 - `sample_w2v`
- Q4.3 - `w2v_grads`
- Q4.4 `train_w2v`

Data your model parameters saved in a pickle file called `assignment5.pkl`. We will be expecting multiple parameters with the following embeddings, which you can save with the following code.

```
data = {
    'word_freqs': <YOUR-DISTRIBUTION>'
    'V_svd': <YOUR-SVD-EMBEDDINGS, Vi>
    'V_bow': <YOUR-BAG-OF-WORD-EMBEDDINGS, Vi>
    'V_w2v': <YOUR-WORD2VEC-EMEDDINGS, Vi>,
}

with open('assignment5.pkl', 'wb') as f:
    pickle.dump(data, f)
```

Plots and Derivations your plots and derivations in a PDF in a file called `assignment5.pdf`. Plot the distribution of words.

- Q1 - Plots of the distribution of words
- Q2.2 - Nearest neighbors to words

- Q3.2 - Plots of loss function, answer to why W_1 can be used as word vectors, nearest neighbors to words
- Q4.2 - Gradient derivations
- Q4.4 - Plots of loss function, nearest neighbors to words