Assignment 1:
Word Embeddings - Properties, Meaning and Training

Welcome to the first assignment of ECE 1786! This assignment engages you in the first major topic of the course - the properties, meaning, viewing, and training of word embeddings (also called word vectors). This assignment must be done individually. The specific learning objectives in this assignment are:

1. To set up the computing environment used in this course.

2. To learn word embedding properties, and use them in simple ways.

3. To translate vectors into understandable categories of meaning.

4. To understand how embeddings are created, using two methods - the Skip Gram method, and the Skip Gram with Negative Sampling (SGNS) method, and to get a sense of trade-offs in the creation of embeddings.

Deadline: Monday, September 25, 2023 at 9:00pm
Late Penalty: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted between 1 hour and 24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted.

What To Submit

You should submit the following to the Quercus website for this course, under Assignment 1:

1. A single PDF document, called `Assign1.pdf` which answers *every* question posed in Sections 1 through 4 of this assignment. You should number your answer to each question in the form `Question X.Y`, where X is the section of this assignment, and Y is the numbered element in the question. **You should include the specific written question itself** and then provide your answer.

2. You should submit separate code files for the code that you wrote for each section, as specified in each section, with the specified file name. For example, Section 1 Part 2 asks you to submit a python (text) file named `A1P1_2.py`. Note that you're required to submit python files, *not* notebook (`.ipynb`)files.

Before You Begin: Set Up Your Computing Environment

Set up your computing environment according to the document **Course Software Infrastructure and Background** provided with this assignment. Note that the pre-requisite of this course requires that you have experience and knowledge of software systems like (or exactly the same) as described in that document.

1 Properties of Word Embeddings [15 points]

In the first lecture of this course, we discuss properties of word embeddings/vectors, and use a PyTorch notebook to explore some of their properties. You can retrieve that code in notebook provided with this assignment called `A1_Section1_starter.ipynb`. It illustrates the basic properties of the GloVe embeddings [1]. As a way to gain familiarity with word vectors, do each of the following:

1. Read the documentation of the `Vocab` class of Torchtext that you can find here: https://torchtext.readthedocs.io/en/latest/vocab.html and then read the `A1_Section1_starter.ipynb` code. Run the notebook and make sure you understand what each step does.

2. Write a new function, similar to `print_closest_words` called `print_closest_cosine_words` that prints out the N-most (where N is a parameter) similar words using `cosine similarity` rather than euclidean distance. Provide a table that compares the 10-most cosine-similar words to the word 'dog', in order, alongside to the 10 closest words computed using euclidean distance. Give the same kind of table for the word 'computer.' Looking at the two lists, does one of the metrics (cosine similarity or euclidean distance) seem to be better than the other? Explain your answer. Submit the specific code for the `print_closest_cosine_words` function that you wrote in a separate Python file named `A1P1_2.py`. [2 points]

3. The section of `A1_Section1_starter.ipynb` that is labelled **Analogies** shows how relationships between pairs of words is captured in the learned word vectors. Consider, now, the word-pair relationships given in Figure 1 below, which comes from Table 1 of the Mikolov[2] paper. Choose one of these relationships, but not one of the ones already shown in the starter notebook, and report which one you chose. Write and run code that will generate the second word given the first word. Generate 10 more examples of that same relationship from 10 other words, and comment on the quality of the results. Submit the specific code that you wrote in a separate Python file, `A1P1_3.py`. [4 points]

4. The section of `A1_Section1_starter.ipynb` that is labelled **Bias in Word Vectors** illustrates examples of bias within word vectors in the notebook, as also discussed in class. Choose a context that you're aware of (different from those already in the notebook), and see if you can find evidence of a bias that is built into the word vectors. Report the evidence and the conclusion you make from the evidence. [2 points]

5. Change the the embedding dimension (also called the vector size) from 50 to 300 and re-run the notebook including the new cosine similarity function from part 2 above. How does the euclidean difference change between the various words in the notebook when switching from d=50 to d=300? How does the cosine similarity change? Does the ordering of nearness change? Is it clear that the larger size vectors give better results - why or why not? [5 points]

6. There are many different **pre-trained** embeddings available, including one that tokenizes words at a sub-word level[3] called `FastText`. These pre-trained embeddedings are available from Torchtext. Modify the notebook to use the FastText embeddings. State any changes that you see in the **Bias** section of the notebook. [2 points]

Table 1: *Examples of five types of semantic and nine types of syntactic questions in the Semantic-Syntactic Word Relationship test set.*

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

Figure 1: Mikolov's Pairwise Relationships

2   Computing Meaning From Word Embeddings [12 points]

Now that we've seen some of the power of word embeddings, we can also feel the frustration that the individual elements/numbers in each word vector do not have a meaning that can be interpreted or understood by humans. It would have preferable that each position in the vector correspond to a specific *axis of meaning* that we can understand based on our ability to comprehend language. For example the "amount" that the word relates to *colour* or *temperature* or *politics*. This is not the case, because the numbers are the result of an optimization process that does not drive each vector element toward human-understandable meaning.

We can, however, make use of the methods shown in Section 1 above to measure the amount of meaning in specific categories of our choosing, such as colour. Suppose that we want to know how much a particular word/embedding relates to colour. One way to measure that could be to determine the cosine similarity between the *word embedding* for colour and the word of interest. We might expect that a word like 'sky' or 'grass' might have elements of colour in it, and that 'purple' would have more. However, it may also be true that there are multiple meanings to a single word, such as 'colour', and so it might be better to define a category of meaning by using several words that, all together, define it with more precision.

For example, a way to define a category such as colour would be to use that word itself, and together with several examples, such 'red', 'green', 'blue', 'yellow.' Then, to measure the "amount" of colour in a specific word (like 'sky') you could compute the average cosine similarity between `sky` and each of the words in the category. Alternatively, you could average the vectors of all the words in the category, and compute the cosine similarity between the embedding of `sky` and that average embedding. In this section, use the d=50 GloVe embeddings that you used in Section 1.

Do the following:

1. Write a PyTorch-based function called compare_words_to_category that takes as input:

sun	moon	winter
rain	cow	wrist
wind	prefix	ghost
glow	heated	cool

Table 1: Vocabulary to Test in Section 2

- The *meaning category* given by a set of words (as discussed above) that describe the category, and

- A given word to 'measure' against that category.

The function should compute the cosine similarity of the given word in the category in two ways:

(a) By averaging the cosine similarity of the given word with every word in the category, and

(b) By computing the cosine similarity of the word with the average embedding of all of the words in the category.

Submit the specific code that you wrote in a separate Python file, `A1P2_1.py`. [2 points]

2. Let's define the *colour* meaning category using these words: "colour", "red", "green", "blue", "yellow." Compute the similarity (using both methods (a) and (b) above) for each of these words: "greenhouse", "sky", "grass", "azure", "scissors", "microphone", "president" and present them in a table. Do the results for each method make sense? Why or why not? What is the apparent difference between method 1 and 2? [4 points]

3. Create a similar table for the meaning category *temperature* by defining your own set of category words, and test a set of 10 words that illustrate how well your category works as a way to determine how much temperature is "in" the words. You should explore different choices and try to make this succeed as much as possible. Comment on how well your approach worked. [4 points]

4. Use these two categories (colour & temperature) to create a new word vector (of dimension 2) for each of the words given in Table 1, in the following way: for each word, take its (colour, temperature) cosine similarity numbers (try both methods and see which works better), and apply the `softmax` function to convert those numbers into a probabilities. Plot each of the words in two dimensions (one for colour and one for temperature) using matplotlib. Do the words that are similar end up being plotted close together? Why or why not? [2 points]

3 Training A Word Embedding Using the Skip-Gram Method on a Small Corpus [16 points]

In lecture 2, we described the Skip Gram method of training word embeddings [2]. In this Section you are going to write code to use that method to train a very small embedding, for a very small vocabulary on very small corpus of text. The goal is to gain some insight into the general notion of how embeddings are produced in a neural net training context. The corpus you are going to use

was provided with this assignment, in the file `SmallSimpleCorpus.txt`.

Your task is to write complete code (given some starter code) to train, test and display the trained embeddings, using the *skip gram* method. (In Section 4 you'll use a different, more efficient method). You can find the starter code in the provided file `A1_Section3_starter.ipynb`.

1. First, read the file `SmallSimpleCorpus.txt` so that you see what the sequence of sentences is. Recalling the notion "you shall know a word by the company it keeps," find **three** pairs of words that this corpora implies have similar or related meanings. For example, 'he' and 'she' are one such example – which you cannot use in your answer! [1 point]

2. The `prepare_texts` function in the starter code is given to you and fulfills several key functions in text processing, a little bit simplified for this simple corpus. Rather than full tokenization (covered in Section 4 below, you will only *lemmatize* the corpus, which means converting words to their *root* - for example the word "holds" becomes "hold", whereas the word "hold" itself stays the same (see the Jurafsky [4] text, section 2.1 for a discussion of lemmatization). The `prepare_texts` function performs lemmatization using the `spaCy` library, which also performs *parts of speech* tagging. That tagging determines the type of each word such as noun, verb, or adjective, as well as detecting spaces and punctuation. Jurafsky [4] Section 8.1 and 8.2 describes parts-of-speech tagging. The function `prepare_texts` uses the parts-of-speech tag to eliminate spaces and punctuation from the vocabulary that is being trained.

   Review the code of `prepare_texts` to make sure you understand what it is doing. Write the code to read the corpus `SmallSimpleCorpus.txt`, and run the `prepare_texts` on it to return the text (lemmas) that will be used next. Check that the vocabulary size is 11. Which is the most frequent word in the corpus, and the least frequent word? What purpose do the v2i and i2v functions serve? [2 points]

3. Write a new function called `tokenize_and_preprocess_text` the skeleton of which is given in the starter code, but not written. It takes the lemmatized small corpus as input, along with `v2i` (which serves as a simple, lemma-based tokenizer) and a window size `window`. You should write it so that its output should be the Skip Gram training dataset for this corpus: pairs of words in the corpus that "belong" together, in the Skip Gram sense. That is, for every word in the corpus a set of training examples are generated with that word serving as the (target) input to the predictor, and all the words that fit within a window of size `window` surrounding the word would be predicted to be in the "context" of the given word. The words are expressed as tokens (numbers). To be clear, this definition of `window` means that only odd numbers of 3 or greater make sense. A window size of 3 means that there are maximum 2 samples per target word, using the one word before and the one word after.

   For example, if the corpus contained the sequence `then the brown cow said moo`, and if the current focus word was `cow`, and the window size was `window=3`, then there would be two training examples generated for that focus word: (`cow`, `brown`) and (`cow`, `said`). You must generate all training examples across all words in the corpus within a window of size `window`. Test that your function works, and show with examples of output (submitted) that it does. [2 points]

4. Next you should define the model to be trained, the skeleton for which is give in the starter code class `Word2vecModel`. Portions of the weights in this model, once trained, provides the trained embeddings we are seeking. Recall that the input to the model is a token (a number) representing which word in the vocabulary is being predicted *from*. The output of the model is of size `|V|`, where `V` is the size of the vocabulary, and each individual output in some sense represents the probability of that word being the correct output. That prediction is based directly on the embedding for each word, and the embeddings are quantities being determined during training. Set the embedding size to be 2, so that will be the size of our word embeddings/vectors. What is the total number of parameters in this model with an embedding size of 2 - counting all the weights and biases? Submit your code for the Word2VecModel class in the file `A1P3_4.py`. [2 points]

5. Write the training loop function, given in skeleton form in the starter code as function `train_word2vec`. It should call the function `tokenize_and_preprocess_text` to obtain the data and labels, and split that data to be 80% training and 20% validation data. It should use a Cross Entropy loss function, a batch size of 4, a `window` size of 5, and 50 Epochs of training. Using the default Adam optimizer, find a suitable learning rate, and report what that is. Show the training and validation curves (loss vs. Epoch), and comment on the apparent success (or lack thereof) that these curves suggest. Submit your code for the training function `train_word2vec` in the file `A1P3_5.py` [4 points]

6. For your best learning rate, display each of the embeddings in a 2-dimensional plot using Matplotlib. Display both a point for each word, and the word itself. Submit this plot, and answer this question: Do the results make sense, and confirm your choices from part 1 of this Section? What would happen when the window size is too large? At what value would `window` become too large for this corpus? [5 points]

7. Run the training sequence twice - and observe whether the results are identical or not. Then set the random seeds that are used in, separately in `numpy` and `torch` as follows (use any number you wish, not necessary 43, for the seed):

```
np.random.seed(43)
torch.manual_seed(43)
```

Verify (and confirm this in your report) that the results are always the same every time you run your code if you set these two seeds. This will be important to remember when you are debugging code, and you want it to produce the same result each time.

4   Skip-gram with Negative Sampling [14 points + 2 bonus points]

One big issue with the pure skip gram training method presented in the Section 3 is that it is quite slow, largely because the output of the model is the size of the vocabulary, and normal vocabularies are in the range of 5,000 or more words. An alternative is called *skip gram with negative sampling*, or SGNS for short. Here, rather than predict a word that is contextually associated with another, we build a binary predictor that says whether two words belong together as part of the same context. To make such a binary predictor, the training data has to contain both positive examples (as in Section 3) and *negative* examples, hence the 'negative sampling' in the SGNS name. Section 6.8

of the Jurafsky text [4] gives a detailed description of this approach.

The goal in this Section to gain more understanding into how embeddings are made, using a larger embedding size and a much larger corpus, with a much larger vocabulary. The corpus you are going to use was provided with this assignment, in the file `LargerCorpus.txt`. It comes from a book called "Illustrated History of the United States Mint." We note that if you tried to train a word embedding of reasonable size on this corpus, using the method of Section 3, it would be very slow!

Starter code is provided in the file `A1_Section4_starter.ipynb`.

1. Take a quick look through `LargerCorpus.txt` to get a sense of what it is about. Give a 3 sentence summary of the subject of the document. [1 point]

2. The `prepare_texts` function in the starter code is a more advanced version of that same function given in Section 3. Read through it and make sure you understand what it does. What are the functional differences between this code and that of the same function in Section 3? [1 point]

3. Write the code to read in `LargerCorpus.txt` and run `prepare_texts` on it. Determine the number of words in the text, and the size of the filtered vocabulary, and the most frequent 20 words in the filtered vocabulary, and report those. Of those top 20 most frequent words, which one(s) are unique to the subject of this particular text? [1 point]

4. Write the function `tokenize_and_preprocess_text` which generates both positive and negative samples for training in the following way: first, use `w2i` to create a tokenized version of the corpus. Next, for every word in the corpus, generate: a set of positive examples within a window of size `window` similar to the example generation in Section 3. Set the label for each positive example to be $+1$, in the list `Y` produced by the function. **Also**, for each word, generate the same number of negative samples as positive examples, by choosing that number of randomly-chosen words from the entire corpus. (You can assume randomly chosen words are very likely to be not associated with the given word). Set the label of the negative examples to be -1. Submit your code for this function in the file named `A1P4_4.py`. How many total examples were created? [2 points]

5. OPTIONAL: The training can be made more efficient by reducing the number of examples for the most frequent words, as the above method creates far more examples connected to those words than are necessary for successful training. Revise your function to reduce the number of examples. Submit your code for this function in the file named `A1P4_5.py`, and state how many examples remain for the corpus using this reduction. [2 bonus points]

6. Write the model class, from the given skeleton class `SkipGramNegativeSampling`. The model's input are the tokens of two words, the word and the context. The model stores the embedding that is being trained (just one embedding per token), which is set up in the `_init_` method. The output of the forward function is a binary prediction, but it should only compute the raw prediction of the network (which is the dot product of the the two embeddings of the input tokens). Submit your model class in the file `A1P4_6.py`. [2 points]

7. Write the training function, given in skeleton form in the starter code as the function `train_sgns`. This function should call the `tokenize_and_preprocess_text` function to

obtain the data and labels, and split that data to be 80% training and 20% validation data. It should use an embedding size of 8, a window size of 5, a batch size of 4, and 30 Epochs of training. The loss function should be $\log(\sigma(prediction))$ for positive examples, and $\log(\sigma(-prediction))$ where σ is the sigmoid function. Since it is possible for the prediction to be 0, to prevent the log function from having an infinite result, we typically add a small constant the output of the sigmoid to prevent this - typically `1e-5`.

Using the default Adam optimizer, find a suitable learning rate, and report what that is. Show the training and validation curves vs. Epoch, and comment on the apparent success (or lack thereof) that these curves suggest. Submit your training function in the file `A1P4_7.py`. [4 points]

8. Write a function that reduces the dimensionality of the embeddings from 8 to 2, using principle component analysis (PCA) as shown in the partially-written function `visualize_embedding`. Since we cannot visualize the embeddings of the entire vocabulary, the function is written to select a range of the most frequent words in the vocabulary. (Too frequent are not interesting, but too infrequent are also less interesting). Comment on how well the embeddings worked, finding two examples each of embeddings that appear correctly placed in the plot, and two examples where they are not. [3 points]

References

[1] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *https://arxiv.org/abs/1301.3781*, 2013.

[3] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[4] Dan Jurafsky and James H. Martin. Speech and language processing : an introduction to natural language processing, computational linguistics, and speech recognition, 3rd edition draft. *https://web.stanford.edu/ jurafsky/slp3/*, 2023.