CARTE ML Bootcamp Lab 5-1:
Word Embeddings - Properties, Meaning and Training

This lab engages you in the properties, meaning, viewing, and training of word embeddings (also called word vectors). The specific learning objectives in this assignment are:

1. To learn word embedding properties, and use them in simple ways.
2. (optional) To translate vectors into understandable categories of meaning.
3. To understand how embeddings are created, using the Skip Gram method.

Your Computing Environment: Google Colab

With a basic google account you can use Google Colab which provides free and fairly good performance computing without the need to do much set up of your environment. When logged in to your google account, go to https://colab.research.google.com. To learn how to write python code into a Google Colab notebook, read and follow the following links:

1. What is Colab?
2. Near the bottom of *What is Colab?* click on these links:
   - Overview of Colaboratory
   - Guide To Markdown
   - Guide to Local Files, Drive, Sheets and Cloud Storage

Colab also has most of the libraries/frameworks we need for this course already installed, including **torch** and **torchtext** (https://pytorch.org/tutorials/beginner/text_sentiment_ngrams_tutorial.html) and **spaCy** (https://spacy.io/). However, to use the english library of spaCy in Colab, you'll need to import it (every time) with the following code in the notebook (the exclamation mark causes the code to run in the shell containing your code):

```
!python -m spacy download en
```

1 Properties of Word Embeddings

In the first part of this day, we have discussed properties of word embeddings/vectors, and use a PyTorch notebook to explore some of their properties. You can retrieve that code in and upload to google colab the notebook called `lab_5_1_vectors.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:

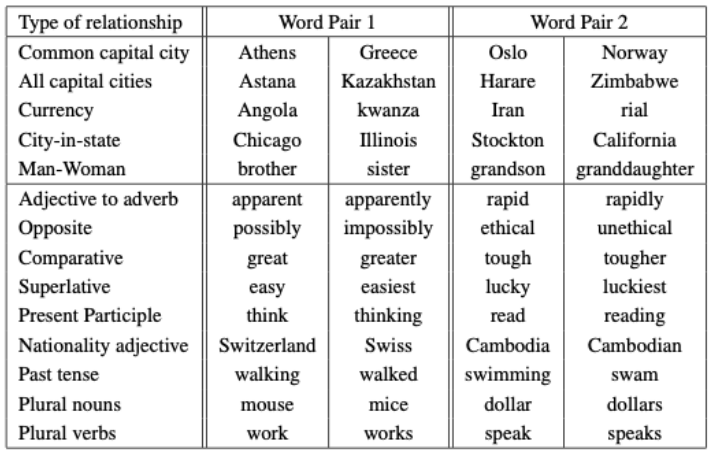
1. Read the documentation of the `Vocab` class of Torchtext that you can find here: https://torchtext.readthedocs.io/en/latest/vocab.html and then read the `lab_5_1_vectors.ipynb` code. Run the notebook and make sure you understand what each step does.

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.

3. The section of `lab_5_1_vectors.ipynb` that is labelled **Analogies** shows how a relationships between pairs of words that 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.

4. 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?

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 Optional: Computing Meaning From Word Embeddings

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.

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

Table 1: Vocabulary to Test in Section 2

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:
   - 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.

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", "purple", "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?

3 Training A Word Embedding Using the Skip-Gram Method on a Small Corpus

The lecture this morning described the Skip Gram method of training word embeddings [2]. In this Section you are going to revew 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. The corpus you are going to use was provided with this assignment, in the file `SmallSimpleCorpus.txt`, and was also shown in the lecture.

Your task is to read the complete code to train, test and display the trained embeddings, using the *skip gram* method, you can find the code in the notebook `lab_5_1_train_word_vectors`.

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!

2. Read the `prepare_texts` function in the code given to you fulfills several key functions in text processing, a little bit simplified for this simple corpus. Rather than full tokenization it only *lemmatizes* 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 [3] 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 [3] 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. Review the code that reads 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?

3. The function called `tokenize_and_preprocess_text` takes the lemmatized small corpus as input, along with `v2i` (which serves as a simple, lemma-based tokenizer) and a window size `window`. 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. Add a little code so that you can see the dataset that is produced.

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)`. The code generates all training examples across all words in the corpus within a window of size `window`. Run this code and check its output (that you printed out).

4. Review the code in function `Word2vecModel`. Part of this model ultimately provides the trained embeddings/vectors, and you can see these are defined and initialized to random numbers in the line:

   ```
   self.embedding = torch.nn.Parameter(torch.rand(vocab_size, embedding_size)) # * 0.01)
   ```

   The subsequent line defines the prediction model, which is a simple fully-connected linear neural network. 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. Note also that more than one example is presented to the network at once during training, a number called the *batch size*.

5. The training loop function, `train_word2vec` calls the function `tokenize_and_preprocess_text` to obtain the data and labels, and splits that data to be 80% training and 20% validation data. It uses Cross Entropy loss function described in the lecture, a batch size of 4, a `window` size of 5, and 50 Epochs of training. It uses the Adam optimizer, and a learning rate of 1x10^{-3}. [Jiading/Mohamed, add in something that includes the training curves.] Run the training code.

6. Run the code that displays each of the embeddings in a 2-dimensional plot using Matplotlib. 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?

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.

8. Run the nearest neighbours code to see if the results also make sense.

4 Training A Single-Neuron Classifier to Determine if a Sentence is Objective or Subjective

You may have to leave this exercise to the afternoon, depending on which of the above exercises you chose to do. The purpose of this exercise is to review the code for training a simple network(just a single neuron) to determine if a sentence is objective or subjective.

1. Load the notebook `lab_5_1_Obj_Sub_Classify` into google colab, and also load the data file `data.tsv` as instructed near the top of the notebook.
2. Take a quick look at the file data.tsv to see which sentences were labelled subjective (1) and which objective (0). (The 1's are in the first half of the file)
3. Read through each of the code blocks, getting a rough sense of what is going on by reading the comments. You see code functions that split the dataset, in the tab-separated file data.tsv into training, validation and test sets. Perhaps look closest at the code block call "Classifier model" class where you can see the torch.nn.Linear class being used to instantiate a single neuron with `embedding_size` inputs and just 1 output.
4. Run the code up to and including the block titled "instantiate the model and execute the fit function." Observe the training curves – training loss and validation loss, and the accuracy of the test set as the training proceeds. What is the final accuracy?
5. Run the next block, titled 'interesting experiment.' Here we point out that the number of parameters in this model is equal to the `embedding_size`, and if you think about it, that vector is being used as the deciding point, somehow, of the average input vector. So, it is interesting to see what the `print_closest_cosine_words` are for this vector. The code prints those out - what do you observe about those words.
6. Run the remaining code, which installs the `gradio` package and sets up an easy-to-use interface with the trained model, that you can type in any sentence and have the model decide if it is subjective or objective. To use the interface, just type in the sentence and click the `classify` button. You may find that it is better at classifying longer sentences, as that is the nature of the dataset it was trained on.

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