SI 630: Homework 2 – Word Embeddings and Attention

Due: See Canvas

1 Introduction

How do we represent word meaning so that we can analyze it, compare different words' meanings, and use these representations in NLP tasks? One way to learn word meaning is to find regularities in how a word is used. Two words that appear in very similar contexts probably mean similar things. One way you could capture these contexts is to simply count which words appeared nearby. If we had a vocabulary of V words, we would end up with each word being represented as a vector of length $|V|$[1] where for a word w_i, each dimension j in w_i's vector, $w_{i,j}$ refers to how many times w_j appeared in a context where w_i was used.

The simple counting model we described actually words pretty well as a baseline! However, it has two major drawbacks. First, if we have a lot of text and a big vocabulary, our word vector representations become very expensive to compute and store. A 1,000 words that all co-occur with some frequency would take a matrix of size $|V|^2$, which has a million elements! Even though not all words will co-occur in practice, when we have hundreds of thousands of words, the matrix can become infeasible to compute. Second, this count-based representation has a lot of redundancy in it. If "ocean" and "sea" appear in similar contexts, we probably don't need the co-occurrence counts for all $|V|$ words to tell us they are synonyms. In mathematics terms, we're trying to find a lower-rank matrix that doesn't need all $|V|$ dimensions.

Word embeddings solve both of these problems by trying to encode the kinds of contexts a word appears in as a low-dimensional vector. There are many (many) solutions for how to find lower-dimensional representations, with some of the earliest and successful ones being based on the Singular Value Decomposition (SVD); one you may have heard of is Latent Semantic Analysis. In Homework 2, you'll learn about a relatively recent technique, `word2vec`, that outperforms prior approaches for a wide variety of NLP tasks and is *very* widely used. This homework will build on your experience with stochastic gradient descent (SGD) and log-likelihood (LL) from Homework 1. You'll (1) implement a basic version of word2vec that will learn word representations and then (2) try using those representations in intrinsic tasks that measure word similarity and an extrinsic task for sentiment analysis.

For this homework, we've provided skeleton code in Python 3 that you can use to finish the implementation of `word2vec` and comments within to help hint at how to turn some of the math into python code. You'll want to start early on this homework so you can familiarize yourself with the code and implement each part.

This homework has the following learning goals:

[1] You'll often just see the number of words in a vocabulary abbreviated as V in blog posts and papers. This notation is shorthand; typically V is somehow related to vocabulary so you can use your judgment on how to interpret whether it's referring to the set of words or referring to the total number of words.

- Develop your pytorch programming skills through working with more of the library
- Learn how word2vec works in practice
- Learn how one form of attention works
- Improve your advanced data science debugging skills
- Have you work with large corpora
- Learn how to use Weights & Biases
- Evaluate one form of model explanation

This homework is a mix of conceptual and skills based learning. As you get the hang of programming neural networks, you'll be able to teach them to do many more advanced tasks. This homework will hopefully help prepare you by again having you advance your skills while also getting you thinking about what training word embeddings can do for us (as practitioners).

2 Notes

We've made the implementation easy to follow and avoided some of the useful-to-opaque optimizations that can make the code *much* faster.[2] As a result, training your model may take some time. We estimate that on a regular laptop, it might take 30-45 minutes to finish training a single epoch of your model. That said, you can still quickly run the model for $\sim$10K steps in a few minutes and check whether it's working. A good way to check is to see what words are most similar to some high frequency words, e.g., "january" or "good." If the model is working, similar-meaning words should have similar vector representations, which will be reflected in the most similar word lists. We have included this as an automated test which will print out the most similar words.

The skeleton code also includes methods for writing word2vec data in a common format readable by the Gensim library. This means you can save your model and load the data with any other common libraries that work with word2vec. Once you're able to run your model for $\sim$100K iterations (or more), we recommend saving a copy of its vectors and loading them in a notebook to test. We've included an exploratory notebook.

On a final note, this is the most challenging homework in the class. Much of your time will be spent on Task 1, which is just implementing word2vec. It's a hard but incredibly rewarding homework and the process of doing the homework will help turn you into a world-class information and data scientist!

3 Data

For data, we'll be using a sample of cleaned Amazon book reviews that's been shrunk down to make it manageable. This is pretty fun data to use since it lets us use word vectors to probe for

[2] You'll also find a *lot* of wrong implementations of word2vec online, if you go looking. Those implementations will be much slower and produce worse vectors. Beware!

knowledge about how people describe products. If you're very ambitious, we've include a lot of extra data you can use to train. Feel free to see how the model works and whether you can get through a single epoch! We've provided several files for you to use in both the word2vec part and in the downstream classification part:

1. reviews-word2vec.med.txt – **Eventually train your `word2vec` model on this data**
2. reviews-word2vec.tiny.txt – A very small sample of data. Your model won't learn much from this but you can use the file to quickly test and debug your code without having to wait for the tokenization to finish.
3. reviews-word2vec.large.txt – If you have an efficient implementation, try training your word2vec model on this data
4. reviews-word2vec.huge.txt – Lots of data! Running on this data will require some careful performance optimization and starting early
5. sentiment.train.csv – This is the training data for your attention-based classifier in Part 4. You do not need this data for word2vec (nor should you use it)
6. sentiment.dev.csv – Labeled data for evaluating the attention-based classifier in Part 4
7. sentiment.test.csv – Unlabeled test data for evaluating the attention-based classifier in Part 4. You will upload your predictions for this test to Kaggle

4 Task 1: Word2vec

In Task 1, you'll implement parts of `word2vec` in various stages. Word2vec itself is a complex piece of software and you won't be implementing all the features in this homework. In particular, you will implement:

1. Skip-gram negative sampling (you might see this as SGNS)
2. Rare word removal
3. Frequent word subsampling

You'll spend the majority of your time on Part 1 of that list which involves writing the gradient descent part. You'll start by getting the core part of the algorithm up without parts 2 and 3 and running with gradient descent and using negative sampling to generate output data that is incorrect. Then, you'll work on ways to speed up the efficiency and quality by removing overly common words and removing rare words.

Parameters and notation The vocabulary size is V , and the hidden layer size is k. The hidden layer size k is a hyperparameter that will determine the size of our embeddings. The units on these adjacent layers are fully connected. The input is a one-hot encoded vector $\mathbf{x}$, which means for a given input context word, only one out of V units, $\{x_1, \ldots, x_V\}$, will be 1, and all other units are 0. The output layer consists of a number of *context words* which are also V-dimensional one-hot encodings of a number of words before and after the input word in the sequence. So if your input word was word w in a sequence of text and you have a context window[3] ± 2, this means you will have four V-dimensional one-hot outputs in your output layer, each encoding words $w_{-2}, w_{-1}, w_{+1}, w_{+2}$ respectively. Unlike the input-hidden layer weights, the hidden-output layer weights are shared: the weight matrix that connects the hidden layer to output word w_j will be the same one that connects to output word w_k for all context words.

The weights between the input layer and the hidden layer can be represented by a $V \times k$ matrix W and the weights between the hidden layer and each of the output contexts similarly represented as C with the same dimensions. Each row of W is the k-dimension embedded representation v_I of the associated word w_I of the input layer—these rows are effectively the word embeddings we want to produce with word2vec. Let input word w_I have one-hot encoding $\mathbf{x}$ and $\mathbf{h}$ be the output produced at the hidden layer. Then, we have:

$$\mathbf{h} = W^T \mathbf{x} = v_I \tag{1}$$

Similarly, v_I acts as an input to the second weight matrix C to produce the output neurons which will be the same for *all* context words in the context window. That is, each output word vector is:

$$\mathbf{u} = C\mathbf{h} \tag{2}$$

and for a specific word w_j, we have the corresponding embedding in C as $v_j^{'}$ and the corresponding neuron in the output layer gets u_j as its input where:

$$u_j = v_j^{'T} \mathbf{h} \tag{3}$$

Note that in both of these cases, multiplying the one-hot vector for a word w_i by the corresponding matrix is the same thing has simply selecting the row of the matrix corresponding to the embedding for w_i. If it helps to think about this visually, think about the case for the inputs to the network: the one-hot embedding represents which word is the center word, with all other words not being present. As a result, their inputs are zero and never contribute to the activation of the hidden layer (only the center word does!), so we don't need to even do the multiplication. In practice, we typically never represent these one-hot vectors for word2vec as it's much more efficient to simply select the appropriate row.

An *unoptimized*, naive version of word2vec would predict which context word w_c was present given an input word w_I by estimating the probabilities across the whole vocabulary using the softmax function:

$$P(w_c = w_c^* | w_I) = y_c = \frac{\exp(u_c)}{\sum_{i=1}^{V} \exp(u_i)} \tag{4}$$

[3]Typically, when describing a window around a word, we use negative indices to refer to words *before* the target, so a ± 2 window around index i starts at $i - 2$ and ends at $i + 2$ but excludes index i.

This original log-likelihood function is then to maximize the probability that the context words (in this case, $w_{-2}, \ldots, w_{+2}$) were all guessed correctly given the input word w_I. **Note that you are not implementing this function!**

Showing this function raises two important questions (1) why is it still being described and (2) why aren't you implementing it? First, the equation represents an *ideal* case of what the model should be doing: given some positive value to predict for *one* of the outputs (w_c), everything else should be close to zero. This objective is similar to the likelihood you implemented for Logistic Regression: given some input, the weights need to be moved to push the predictions closer to 0 or closer to 1. However, think about how many weights you'd need to update to minimize this particular log-likelihood? For each positive prediction, you'd need to update $|V| - 1$ other vectors to make their predictions closer to 0. That strategy which uses the softmax results a huge computational overhead—despite being the most conceptually sound. The success of word2vec is, in part, due to coming up with a smart way to achieve nearly the same result *without* having to apply the softmax. Therefore, to answer the second question, now that you know what the goal is, you'll be implementing a far more efficient method known as **negative sampling** that will approximate creating a model that minimizes this equation!

If you read the original `word2vec` paper, you might find some of the notation hard to follow. Thankfully, several papers have tried to unpack the paper in a more accessible format. If you want another description of how the algorithm works, try reading Goldberg and Levy [2014][4] or Rong [2014][5] for more explanation. There are also plenty of good blog tutorials for how `word2vec` works and you're welcome to consult those[6] as well as some online demos that show how things work.[7]. There's also a very nice illustrated guide to word2vec `https://jalammar.github.io/illustrated-word2vec/` that can provide more intuition too.

4.1 Getting Started: Preparing the Corpus

Before we can even start training, we'll need to determine the vocabulary of the input text and then convert the text into a sequence of IDs that reflect which input neuron corresponds to which word. Word2vec typically treats all text as one long sequence, which ignores sentences boundaries, document boundaries, or otherwise-useful markers of discourse. We too will follow suit. In the code, you'll see general instructions on which steps are needed to (1) create a mapping of word to ID and (2) processing the input sequence of tokens and covert it to a sequence of IDs that we can use for training. This sequence of IDs is what we'll use to create our training data. As a part of this process, we'll also keep track of all the token frequencies in our vocabulary.

■ **Problem 1.** Modify function `load_data` in the `Corpus` class to read in the text data and fill in the `id_to_word`, `word_to_id`, and `full_token_sequence_as_ids` fields. You can safely skip the rare word removal and subsampling for now.

[4] `https://arxiv.org/pdf/1402.3722.pdf`
[5] `https://arxiv.org/pdf/1411.2738.pdf`
[6] E.g., `http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/`
[7] `https://ronxin.github.io/wevi/`

4.2 Negative sampling

For a target word, the nearby words in the context form the positive example for training our prediction model. Rather than train word2vec like a regular mutliclass classification model (which uses the softmax function to predict outputs[8]), word2vec uses a small number of randomly-selected words as negative examples.[9] These negative examples are referred to as the *negative samples*. The negative samples are chosen using a unigram distribution raised to the $\frac{3}{4}$ power: Each word is given a weight equal to its frequency (word count) raised to the $\frac{3}{4}$ power. The probability for a selecting a word is just its weight divided by the sum of weights for all words. The decision to raise the frequency to the $\frac{3}{4}$ power is fairly empirical and this function was reported in their paper to outperform other ways of biasing the negative sampling towards infrequent words.

Computing this function each time we sample a negative example is expensive, so one important implementation efficiency is to create a table so that we can quickly sample words. We've provided some notes in the code and your job will be to fill in a table that can be efficiently sampled.[10]

■ **Problem 2.** Modify function `generate_negative_sampling_table` to create the negative sampling table.

4.3 Generating the Training Data

Once you have the tokens in place, the next step is get the training data in place to actually train the model. Say we have the input word "fox" and observed context word "quick". When training the network on the word pair ("fox", "quick"), we want the model to predict an output of 1 signalling this word ("quick") was present in the context.

With negative sampling, we are will randomly select a small number of negative examples (let's say 2) for each positive example to update the weights for. (In this context, a negative example is one for which we want the network to output a 0 for). When updating the model (later), our parameters will be updated on our current ability to predict 1 for the positive examples and 0 for the negative examples.

To generate the training, you'll iterate through all token IDs in the sequence. At each time step, the current token ID will become the *target word*. You'll use the `window_size` parameter to decide how many nearby tokens should be included as positive training examples.

The original word2vec paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets. In this assignment, you will update

[8]When using the softmax, you would update the parameters (embeddings) for all the words after seeing each training instance. However, consider how many parameters we have to adjust: for one prediction, we would need to change $|V|N$ weights—this is expensive to do! Mikolov *et al.* proposed a slightly different update rule to speed things up. Instead of updating all the weights, we update only a small percentage by updating the weights for the predictions of the words in context and then performing *negative sampling* to choose a few words at random as negative examples of words in the context (i.e., words that shouldn't be predicted to be in the context) and updating the weights for these negative predictions.

[9]There is another formulation of word2vec that uses a hierarchical softmax to speed up the softmax computation (which is the bottleneck) but few use this in practice.

[10]Hint: In the slides, we showed how to sample from a multinomial (e.g., a dice with different weights per side) by turning it into a distribution that can be sampled by choosing a random number in [0,1]. You'll be doing something similar here.

with 2 negative words per context word. This means that if your context window selects four words, you will randomly sample 8 words as *negative examples* of context words. We recommend keeping the negative sampling rate at 2, but you're welcome to try changing this and seeings its effect (we recommend doing this *after* you've completed the main assignment).

Note: There is one important PyTorch-related wrinkle that you will need to account for, which is described in detail in the code.

■ **Problem 3.** Generate the list of training instances according to the specifications in the code.

4.4 Define Your word2vec Network

Now that the data is ready, we can define our PyTorch neural network for word2vec. Here, we will not use layers but instead use PyTorch's `Embedding` class to keep track of our target word and context word embeddings.

■ **Problem 4.** Modify the `init_weights` function to initialize the values in the two `Embedding` objects based on the size of the vocabulary $|V|$ and the size of the embeddings. Unlike in logistic regression where we initialized our β vector be zeros, here, we'll initialize the weights to have small non-zero values centered on zero and sampled from (`-init_range`, `init_range`).[11]

The next step is to update the `forward` function, which takes as input some target word and context words and predicts 0 or 1 for whether each context word was present. Formally, for some target word vector v_t and context word vector v_c, word2vec makes its predictions as

$$\sigma(v_t \cdot v_c) \tag{5}$$

where σ is the sigmoid function (like in Homework 1). Word2vec aims to learn parameters (its two embedding matrices) such that this function is maximized for positive examples and minimized for negative examples.

■ **Problem 5.** Modify the `forward` function

4.5 Train Your Model

Once you have the data in the right format, you're ready to train your model! You will need to implement the core training loop like you did in Homework 1, where you iterate over all the instances in a single epoch and potentially train for multiple epochs.

One key difference this time is that you will use *batching*. In Homework 1 we had a stark contrast between (1) full gradient descent where a single step required us to compute the gradient with respect to all the data and (2) stochastic gradient descent where take a step based on the prediction error for a single instance. However, there is a middle ground! Often we can improve the gradient by computing it with respect to a few instances instead of just one. Analogously, consider if you wanted to know if you were on the right track, it can help to ask a few folks, but you don't need to ask everyone (and asking just one person could be risky and send you on the wrong track). Batched gradient descent is the same way.

[11] Why initialize this way? Consider what would happen if our initial matrices were all zero and we had to compute the inner product of the word and context vectors. The value would always be zero and the model would never be able to learn anything!

Conveniently, PyTorch works nearly seamlessly with batching. We can tell the `DataLoader` class our batch size and it will return a random sample of instances of that size. The code you write for the `forward` function will also work with a batch too with no modifications (most of the time). This behavior is even better for us because often computers are much faster at larger computations—especially GPUs—so trying to do the forward/backward passes for an entire batch is often just as fast as doing them for a single instance.

Note: One caveat to things just working is that sometimes your forward-pass code will be set up so that it can't work with batching. The code hints and description in the notebook will hopefully help you avoid these, but we're also here to support you in Piazza.

In your implementation we recommend starting with these default parameter values:

- batch size = 16 (you can go higher too if your computer supports it, which will speed things up!)
- $k = 50$ (embedding size)
- $\eta = 5e - 5$ (learning rate)
- window ± 2
- min_token_freq $= 5$
- epochs $= 1$
- optimizer = AdamW

You can experiment around with other values to see how it affects your results. Your final submission should use a batch size > 1. For more details on the equations and details of word2vec, consult Rong's paper [Rong, 2014], especially Equations 59 and 61.

■ **Problem 6.** Modify the cell containing the training loop to complete the required PyTorch training process. The notebook describes in more details all the steps

■ **Problem 7.** Check that your model actually works. We recommend running your code on the `reviews-word2vec.med.txt` file for one epoch. After this much data, your model should know enough for common words that the nearest neighbors (words with the most similar vectors) to words like "january" will be month-related words. We've provided code at the end of the notebook to explore. Try a few examples and convince yourself that your model/code is working.

Once you're finished here, you're not yet ready to run everything but you're close!

4.6 Implement stop-word and rare-word removal

Using all the unique words in your source corpus is often not necessary, especially when considering words that convey very little semantic meaning like "the", "of", "we". As a preprocessing step, it can be helpful to remove any instance of these so-called "stop words".

Note that when you remove stop words, you should keep track of their position so that the context doesn't include words outside of the window. This means that a sentence with "my big *cats* of the kind that..." if you have a context window of ± 2, then you would only have "my" and "big" as context words (since "of" and "the" get removed) and not include "kind."

4.6.1 Minimum frequency threshold.

In addition to removing words that are so frequent that they have little semantic value for comparison purposes, it is also often a good idea to remove words that are so *infrequent* that they are likely very unusual words or words that don't occur often enough to get sufficient training during SGD. While the minimum frequency can vary depending on your source corpus and requirements, we will set `min_count = 5` as the default in this assignment.

Instead of just removing words that had less than `min_count` occurrences, we will replace these all with a unique token <UNK>. In the training phase, you will skip over any input word that is <UNK> but you will still keep these as possible context words.

■ **Problem 8.** Modify function `load_data` to convert all words with less than `min_count` occurrences into <UNK> tokens. Modify your dataset generation code to avoid creating a training instance when the target word is <UNK>.

4.6.2 Frequent word subsampling

Words appear with varying frequencies: some words like "the" are very common, whereas others are quite rare. In the current setup, most of our positive training examples will be for predicting very common words as context words. These examples don't add much to learning since they appear in many contexts. The word2vec library offers an alternative to ensure that contexts are more likely to have meaningful words. When creating the sequence of words for training (i.e., what goes in `full_token_sequence_as_ids`), the software will randomly drop words based on their frequency so that more common words are less likely to be included in the sequence. This subsampling effectively increases the context window too—because the context window is defined with respect to `full_token_sequence_as_ids` (not the original text), dropping a nearby common words means the context gets expanded to include the next-nearest word that was not dropped.

To determine whether a token in `full_token_sequence_as_ids` should be subsampled, the word2vec software uses this equation to compute the probability $p_k(w_i)$ of a token for word w_i being kept in for training:

$$p_k(w_i) = \left(\sqrt{\frac{p(w_i)}{0.001}} + 1 \right) \cdot \frac{0.001}{p(w_i)} \tag{6}$$

where $p(w_i)$ is the probability of the word appearing in the corpus initially. Using this probability, each occurrence of w_i in the sequence is randomly decided to be kept or removed based on $p_k(w_i)$.

■ **Problem 9.** Modify function `load_data` to compute the probability $p_k(w_i)$ of being kept during subsampling for each word w_i.

■ **Problem 10.** Modify function `load_data` so that after the initial `full_token_sequence_as_ids` is constructed, tokens are subsampled (i.e., removed) according to their probability of being kept $p_k(w_i)$.

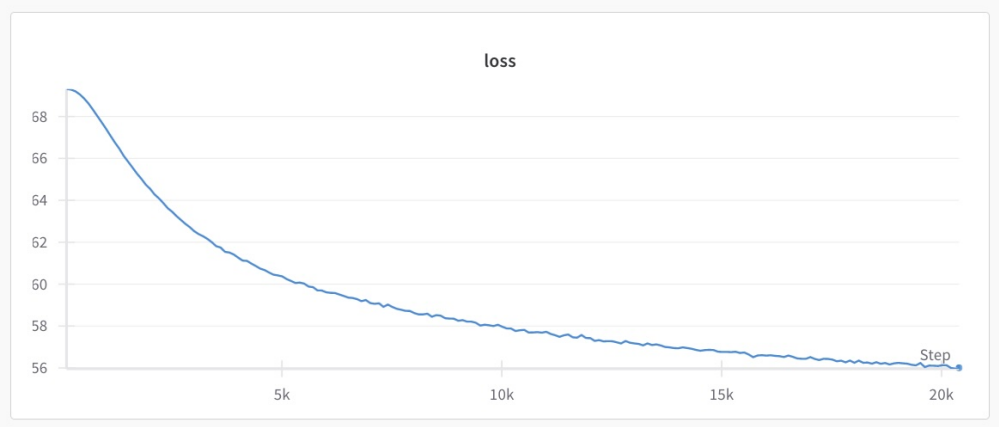

Figure 1: An example `wandb` run from the reference solution where the running sum of loss is reported every 100 steps (i.e., the sum of those steps' loss) across one epoch on the training data. Hovering over any point shows the loss at that time. As you can see, after one epoch the model as learned something but has probably not fully converged!

4.7 Using Weights & Biases

As you might guess, training word2vec on a lot of data can take some time. This waiting process will be increasingly true as you train larger and larger models (not just word2vec). However, the larger pytorch ecosystem provides some fantastic tools for you, the practitioner, to monitor the progress. In this subtask, you'll be using one of those tools, `Weights & Biases` (`wandb`), that allows you to log how your model is doing and then you can connect to the `wandb` website and see the plot. Figure 1 shows an example of the `wandb` plot for our reference implementation after one epoch of training. Here, we've just recorded a running sum of the loss every 100 steps. You will want to do the same. This will help you see how quickly your model is converging.

If you train multiple models, `wandb` will show all of their training plots so you can see how your choice in hyperparameters affects training speed and which model as learned the most (has the lowest loss). In practice, many people use `wandb` to determine when to stop training after seeing at their model has effectively converged.

■ **Problem 11.** Add `wandb` logging to your training loop so that you keep track of the sum of the losses for the past 100 steps and record the value with `wandb`. You will need to register for a free `wandb` account and then log into that account on your computer (e.g., on the command line) so that the `wandb` library can know how and where to post the results.

4.8 Train Your Final Model

All the pieces are now in place and you can verify the model has learned something. For your final vectors, we'll have you train on *at least* one epoch. Before you do that, we'll have you do one quick exploration to see how batch size impacts training speed.

■ **Problem 12.** Try batch sizes of 2, 8, 32, 64, 128, 256, 512 to see how fast each step (one batch worth of updates) is and the total estimated time. For this, you'll set the parameter and then run the training long enough to get an estimate for both with `tqdm` wrapped around your batch iterator.

You do not need to finish training for the full epoch. Make a plot where batch size is on the x-axis and the tqdm-estimated time to finish one epoch is on the y-axis. (You may want to log-scale one or both of the axes). You can try other batch sizes too in this plot if you're curious. In your write up, describe what you see. What batch size would you choose to maximize speed? Side note: You might also want to watch your memory usage, as larger batches can sometimes dramatically increase memory.

■ **Problem 13.** Train your model on at least one epoch worth of data. You are welcome to change the hyperparameters as you see fit for your final model (although batch size must be > 1. Record the full training process and save a picture of the `wandb` plot from your training run in your report. We need to see the plot. It will probably look something like Figure 1.

4.9 Optional Exercises

Once you get word2vec working, if you are *really curious or excited by word2vec*, we've included a few optional exercises or extension you could try out at the very end of the assignment in the notebook. There is no extra credit for these tasks but they will help provide a lot of insight into model building.

5 Task 2: Save Your Outputs

Once you've finished training the model for at least one epoch, save your vector outputs. The rest of the homework will use these vectors so you don't have even re-run the learning code (until the very last part, but ignore that for now). Task 2 is here just so that you have an explicit reminder to save your vectors. We've provided a function to do this for you.

6 Task 3: Qualitative Evaluation of Word Similarities

Once you've learned the word2vec embeddings from how a word is used in context new we can use them! How can we tell whether what it's learned is useful? As a part of training, we put in place code that shows the nearest neighbors, which is often a good indication of whether words that we think are similar end up getting similar representations. However, it's often better to get a more quantitative estimate of similarity. In Task 3, we'll begin evaluating the model by hand by looking at which words are most similar another word based on their vectors.

Here, we'll compare words using the *cosine similarity* between their vectors. Cosine similarity measures the angle between two vectors and in our case, words that have similar vectors end up having similar (or at least related) meanings.

■ **Problem 14.** Load the model (vectors) you saved in Task 2 by using the Jupyter notebook provided (or code that does something similar) that uses the Gensim package to read the vectors. Gensim has a number of useful utilities for working with pretrained vectors.

■ **Problem 15.** Pick 10 target words and compute the most similar for each using Gensim's function. Qualitatively looking at the most similar words for each target word, do these predicted word seem to be semantically similar to the target word? Describe what you see in 2-3 sentences. **Hint:** For maximum effect, try picking words across a range of frequencies (common, occasional, rare words).

■ **Problem 16.** Given the analogy function, find five interesting word analogies with your word2vec model. For example, when representing each word by word vectors, we can generate the following equation, king - man + woman = queen. In other word, you can understand the equation as queen - woman = king - man, which mean the vectors similarity between queen and women is equal to king and man. What kinds of other analogies can you find? (**NOTE:** Any analogies shown in the class recording cannot be used for this problem.) What approaches worked and what approaches didn't? Write 2-3 seconds in a cell in the notebook.

7 Task 4: Using Word Vectors with Attention for Classification

Once you have completed all other steps, *only* then start on Task 4!

Hopefully Task 3 has shown you that your word vectors have learned something. But what exactly do we do with the vectors? In Task 4, you'll try using your vectors in a downstream task: Classifying documents.

Before we get to the details, let's think about about some of the logistics for how a person might use word vectors in a classification by comparing with what you did in Homework 1 with a bag of words (BoW) representation. In the BoW representation, you have a *fixed-length* vector that represents which words are in the document. Even if you extended these features to include bigrams or other kinds of features like who is the author, the vector length would still stay the same if we added more text to the document—adding more words to a document only increase the counts in the document's BoW vector.

What might we do if we have word vectors instead of word counts? Well, one way to think of a simple BoW vector is a sum of the one-hot vectors of the word in the document (e.g., if a word appears seven times, we'd sum its one-hot vector to get a value of 7 in that word's index in the BoW vector). We might take an analogous approach to working with word vectors. To represent a document, we could take the *sum* of the word vectors. This would give us a fixed-length vectors! More words in the document means we just add them to the sum—but the vector length stays the same! In practice, most approaches take the *average of the word vectors* to get a sense of "what kind of content is in this document?" This can work well in practice (as you might see later).

Using the average word vector to represent a document is promising but also seems a bit flawed when we think of which kinds of words are contributing to the vector. Why should the vector for "the" contribute just as much as the vector for "amazing"? In our bag-of-words representation, we tried to mitigate this with re-weighting the BoW vector with techniques like TF-IDF (note: you didn't do this in Homework 1 but we briefly talked about in class). We could try doing something similar with our word vectors but this raises a question: which kind of re-weighting should we use for our classification task? How do we know which words to weight more or less? The answer is at the heart of new approaches to deep learning: *let's learn the weighting!*

In Homework 2, you're going to try a powerful technique called *attention* that is at the heart of many algorithms we'll see later. For now, we're going to try implementing a very simple version of attention. Let's start by thinking about how we might weight word vectors. Let w_i be a word vector for the ith word in a document and let v be a vector that knows about our task and tells us if w_i is related to our task. For now, we won't talk about where v comes from, but just assume the vector contains information about which words are important to the task. To actually quantify how we will use v to measure each word's importance, we'll say that we get a score of each word's relevance for the task by taking the dot product $v \cdot w_i$. Ideally, words that are very relevant should get large dot product values, while irrelevant words get low (negative) values.

How do we use this v vector in practice? Let's say we have some document $d = [w_1, w_2, \ldots, w_n]$. We want to figure out how relevant/important each of those vectors are to our task so that we can compute a weighted average for our document representation. We start by computing the dot-product of v with the vector for each word in our document, $r = [v \cdot w_1, v \cdot w_2, \ldots, v \cdot w_n]$. We'll call this vector r for relevance, where r_i is the relevance score for the ith word in the document. This r vector contains all sorts of values, some very large, some small, and some positive and negative! How do we figure out how to get a re-weighting? We can't simply multiple these values in r by each word's vector—what would it mean to add a negative vector? Instead, we'll use a familiar tool similar to what we saw in Homework 1: the softmax function. Remember the softmax function is a generalization of the sigmoid for more than one class. In essence, if we pass some vector through a softmax function, it produces a probability distribution on the output (the values sum to 1). In our case, we will calculate $\text{Softmax}(r_i) = \frac{\exp(r_i)}{\sum_{j=1\ldots n} \exp(r_j)}$. We'll call the vector output of this softmax a for attention, where a_i is how much we should "attend" to the ith word in the document; since attention is a probability distribution, $0 \leq a_i \leq 1$ and $\sum_{i=1\ldots n} a_i = 1$.

Now that we have an attention vector a, we can compute our new re-weighted document vector as $d = \sum_i a_i w_i$. Note that since a_i is a scalar and w_i is a vector, the multiplication inside the summation is going to scale the value of the vector. We can then use this re-weighted vector for classification, just we did in Homework 1. For a binary classification task, we would define some linear layer X and compare $P(\text{class} = 1) = \sigma(dX)$.

So where do we get the seemingly-magic v that knows about our task? *We will learn v*!! Just like we learned our β vector in Homework 1, we will create an embedding that, through back-propagation, learns how to weight the words in our document to give an weight-average document vector d that performs best on the classification.

Now comes the big twist: What if there are *multiple* kinds of words we should be paying attention to for our task? For example, maybe we need to pay attention to sentiment words *and* pay attention to action verbs? Asking a single v vector to pay attention to many things is asking a lot. In these cases, NLP approaches use *multiple* v vectors. Typically, we call these v vectors different "attention heads"—you can think of each attention head as having eyes looking out for different things. In Homework 4, we'll use four attention heads. Each head gets its own v vector, which we'll denote as v^j (that's not exponentiation), that gets updated separately through backpropagation.

There are a few different ways to combine the outputs of the different attention heads. For example, we could simply average them! However, we'll try to maximize the information from each head by concatenating them, so our final "document representation" is a vector of length $4|d|$.

Let's sum up the steps you'll need to do to build this model:

- Load in the embeddings from your word2vec model and the word-to-index mapping

- Create a new neural network model in PyTorch called `DocumentAttentionClassifier`. The model will have its own Embedding, four attention heads, and a linear layer for predicting the output class from the attention-weighted vectors
- In the `init` function, load in the pre-trained word2vec embeddings into your classifier's Embeddings. We'll use these to start the model knowing something about words.
- Initialize your attention heads randomly
- In the forward pass...
  - For each head in the attention, compute the vector r as the dot-product of the head with the embedding for each word in the document. For best performance, see if you can do with without a for-loop!
  - Compute the softmax of r to get the attention vector a for each attention head
  - Compute the weighted average of the document's vectors using the a vector for each attention head. This should give you four different versions of d
  - Concatenate those four vectors into one vector (stack them side by side)
  - Pass this concatenated vector through a linear layer to get the output activation. Then compute the sigmoid of the activation to get the probability of the class. This last process could look very similar to your logistic regression classifier code.
  - The `forward` function should return the probability *and* the attention vectors for each head. You will need these later.

The hard part of Part 4 is implementing the model. The code for the forward pass can be quite simple (as few as six lines) and involves only a few pytorch operations, most of which are matrix multiplications. We encourage you think of the forward pass as a puzzle that you need to line the pieces up in a particular way.

We *strongly* encourage you to try building a "toy forward pass" as a jupyter cell that just does the steps on some simple input vectors you define (e.g., three word vectors of length 4). This will let you work through each part of the forward pass computation and verify that each step is producing the output you expect—e.g., is my r vector the right shape, or does my a vector contain a probability distribution over all the words for each head? Getting these steps working outside the model will often let you drop in the pipeline into the forward function and have it just work.

A final implementation note, depending on how you implement it, some of these torch functions may be useful: `bmm` (which is also called with the `@` opreator), `softmax`, `view`, `concat`, `squeeze`.

The above sketch is the general idea for the classifier. In addition, you'll need to implement the following too, some of which will look similar to our word2vec training:

- Write code to load in the training, development, and testing data into separate `DataLoader` instances
- Write the core training loop that iterates over the training data in random order
- Write code to report the sum of the prediction loss every 500 steps to `wandb`. Like word2vec, this will help you tell whether the model is learning. *In addition*, every 5000 steps, report the model's F1 on the development set.

In your implementation we recommend starting with these default parameter values:

- batch size = 1 (**Note that this is different from word2vec! See the optional exercises in the code for details)**
- $\eta = 5e - 5$ (learning rate)
- epochs $= 1$ (2+ is recommended
- optimizer = AdamW

■ **Problem 17.** Build your attention-based classification model like the above. Your new model should use `wandb` to track both the loss and F1

■ **Problem 18.** Train the classifier for at least one epoch on the provided training data. You should include a plot for the loss and the F1. Longer training is encouraged but not required.

■ **Problem 19.** What if we didn't want to change the embeddings during the training process—i.e., we could keep our word meaning fixed and just train the network? This idea is known as *freezing* some parameters and can sometimes be a very helpful practice if you have a large set of pretrained vectors but your classifiers training data is small. In that setting, if you update the vectors during classifier-training, only some of the vectors get changed but the rest (for words not in the classifier training data) keep their old values from the word2vec pre-training, which can lead to the model not knowing how to interpret them as well. For this problem *turn off gradient descent on the word vectors* and retrain for one epoch. This training should be much faster since fewer parameters need updating. Show the plots of the loss and performance. In a few sentences, describe whether you think we should freeze our word vectors in this setting or not.[12]

■ **Problem 20.** Predict the classifications for the test set and upload the scores to Kaggle. Be sure to include your Kaggle username in your report so we can give you credit.

■ **Problem 21.** What is our attention doing? We've provided handy helpful function to visualize each head's attention distributions across the input. Your tasks are:

1. Generate at least four "interesting" attention plots from text in the dev data, at least two for for each class, and describe why you think the plots are interesting.
2. Using what you've observed from the visualizations, write a short paragraph describing what you think the attention heads are looking for. Describe any differences you see between heads and whether there are any patterns in terms of what they focus on. We encourage looking at many examples to get a sense of behavior (e.g., try randomly sampling text and visualizing the attention).
3. Try to fool the classifier by either writing an example that the model predicts incorrectly or directly looking for a mistake on the dev data. Show what the attention is looking at in this item. In a few sentences, describe whether the attention is looking at the right thing and whether the attention is a good explanation for why the classifier made the prediction for this item.

7.1 Optional Exercises

Once you get classifier model working, if you are *really curious or excited by attention or building models*, we've included a few optional exercises or extension you could try out at the very end of

[12] For those curious, you can try varying the amount of training data and seeing how performance changes relative to whether the vectors are frozen or not.

the jupyter notebook for that part (not in this PDF). There is no extra credit for these tasks but they will help provide a lot of insight into model building and how to improve/extend attention. Some of them are relatively easy too!

8 Hints

1. Start early; this homework will take time to debug and you'll need time to wait for the model to train for Task 1 before moving on to Tasks 2-4 which use its output or modify the core code.

2. Try implementing the forward pass on toy data that you've designed and manually step through each operation to verify it is doing what you expect

3. Once you get the model implemented, run on a small amount of data at first to get things debugged. We've included some small datasets to help you try this training.

4. Each notebook contains estimated performance times for different parts. Your performance will likely vary but if your times are *much* longer, that usually points to some performance bug.

5. If your main computer is a tablet, you can definitely develop and debug the code on the tablet but you might want to consider using Great Lakes for training the final models. Talk to the instructors if you think you might need this.

6. If you're using a newer Mac computer that has an M1 (or later) processor, you can speed things up by using its onboard GPU-like processor. Once you get your model implemented, you can say `model = model.to("mps")` which should tell torch to do the computation for that model using the M1's faster math processing. We haven't tested this extensively and the performance times aren't reported with this, so your mileage (and bugs) may vary—but it could be useful to try once things are working!

9 Submission

Please upload the following to Canvas by the deadline:

1. Your jupyter notebooks for Tasks 1, 3, and 4 (as code)

2. A separate PDF document that contains these answers, labeled by problem number. Do not submit a PDF copy of your jupyter notebook, as this will receive a penalty (slower to grade). Be sure to include what your username is on Kaggle. All written/plotting problems should be clearly numbered so we can search for them during grading. We reserve the right to score answers as zero if we can't easily find them during grading.

We reserve the right to run any code you submit; **code that does not run or produces substantially different outputs will receive a zero**.

10 Academic Honesty

Unless otherwise specified in an assignment all submitted work must be your own, original work. Any excerpts, statements, or phrases from the work of others must be clearly identified as a quotation, and a proper citation provided. Any violation of the University's policies on Academic and Professional Integrity may result in serious penalties, which might range from failing an assignment, to failing a course, to being expelled from the program. Violations of academic and professional integrity will be reported to Student Affairs. Consequences impacting assignment or course grades are determined by the faculty instructor; additional sanctions may be imposed.

Copying code from any existing word2vec implementation is considered grounds for violation of Academic Integrity and will receive a zero. Code generated through automated or AI-assisted tools, such as Co-Pilot will also receive a zero.

11 Optional Tasks

`Word2vec` has spawned many different extensions and variants. For anyone who wants to dig into the model more, we've included a few optional tasks here. **Before attempting any of these tasks, please finish the rest of the homework and then save your code in a separate file so if anything goes wrong, you can still get full credit for your work.** These optional tasks are intended entirely for educational purposes and no extra credit will be awarded for doing them.

11.1 Optional Task 1: Modeling Multi-word Expressions

In your implementation `word2vec` simply iterates over each token one at a time. However, words can sometimes be a part of phrases whose meaning isn't conveyed by the words individually. For example "White House" is a specific concept, which in NLP is an example of what's called a **multiword expression**.[13] In our particular data, there are *lots* of multi-word expressions. As biographies a lot of people are born in the United States, which ends up being modeled as "united" and "states"—not ideal! We'll give you two ideas.

In Option 1 of Optional Task 1, we've provided a list of common multi-word expressions in our data on Canvas (`common-mwes.txt`). Update your program to read these in and during the `load_data` function, use them to group multi-world expressions into a single token. You're free to use whatever way you want, recognizing that not all instances of a multi-word expression are actually a single token, e.g., "We were united states the leader." This option is actually fair easy and a fun way to get multi-word expressions to show up in the analogies too, which leads to lots of fun around people analogies.

Option 2 is a bit more challenging. Mikolov *et al.* describe a way to automatically find these phrases as a preprocessing step to `word2vec` so that they get their own word vectors. In this option, you will implement their phrase detection as described in the "Learning Phrases" section of Mikolov et al. [2013].[14]

[13] `https://en.wikipedia.org/wiki/Multiword_expression`
[14] `http://arxiv.org/pdf/1310.4546.pdf`

11.2 Optional Task 2: Better UNKing

Your current code treats all low frequency words the same by replacing them with an <UNK> token. However, many of these words could be collapsed into specific types of unknown tokens based on their prefixes (e.g., "anti" or "pre") or suffixes (e.g., "ly" or "ness") or even the fact that they are numbers or all capital letters! Knowing something about the context in which words occur can still potentially improve your vectors. In Optional Task 2, try modifying the code that replaces a token with <UNK> with something of your own creation.

11.3 Optional Task 3: Some performance tricks for Word2Vec

Word2vec and deep learning in general has many performance tricks you can try to improve how the model learns in both speed and quality. For Optional Task 3, you can try two tricks:

- **Dropout**: One useful and deceptively-simple trick is known as *dropout*. The idea is that during training, you randomly set some of the inputs to zero. This forces the model to not rely on any one specific neuron in making its predictions. There are many good theoretical reasons for doing this [e.g., Baldi and Sadowski, 2013]. To try this trick out, during training (and only then!), when making a prediction, randomly choose a small percentage (10%) of the total dimensions (e.g., 5 of the 50 dimensions of your embeddings) and set these to zero before computing anything for predictions.

- **Learning Rate Decay**: The current model uses the same learning rate for all steps. Yet, as we learn the vectors are hopefully getting better to approximate the task. As a result, we might want to make a *smaller* change the vectors as time goes on to keep them close to the values that are producing good results. This idea is formalized in a trick known as *learning rate decay* where as training continues, you gradually lower the learning rate in hopes that the model converges better to a local minima. There are many (many) approaches to this trick, but as an initial idea, try setting a lower bound on the learning rate (which could be zero!) and linearly decrease the learning rate with each step. You might even do this after the first epoch. If you want to get fancier, you can try to only start decreasing the learning rate when the change in log-likelihood becomes smaller, which signals that the models is converging, but could still potentially be fine-tuned a bit more.

11.4 Optional Task 4: Incorporating Synonyms

As a software library, word2vec offers a powerful and extensible approach to learning word meaning. Many follow-up approaches have extended this core approach with aspects like (1) added more information on the context with additional parameters or (2) modifying the task so that the model learns multiple things. In Optional Task 4 you'll try one easy extension: Using external knowledge!

Even though word2vec learns word meaning from scratch, we still have quite a few resources around that can tell us about words. One of those resources which we'll talk much more about in Week 12 (Semantics) is WordNet, which encodes word meanings in a large knowledge base. In particular, WordNet contains information on which word meanings are synonymous. For example, "couch" and "sofa" have meanings that are synonymous. In Optional Task 4, we've provided you

with a set of synonyms (`synonyms.txt`) that you'll use during training to encourage word2vec to learn similar vectors for words with synonymous meanings.

How will we make use of this extra knowledge of which woulds should have similar vectors? There have been many approaches to modifying word2vec, some which are in your weekly readings for the word vector week. However, we'll take a simple approach: during training, if you encounter a token that has one or more synonyms, replace that token with a token sampled from among the synonymous tokens (which includes that token itself). For example, if "state" and "province" are synonyms, when you encounter the token "state" during training, you would randomly swap that token for one sampled from the set ("state", "province"). Sometimes, this would keep the same token, but other times, you force the model to use the synonym—which requires that synonym's embedding to predict the context for the original token. Given more epochs, you may even predict the same context for each of the synonyms. One of the advantages of this approach is that if a synonyms word shows up much less frequently (e.g., "storm" vs. "tempest"), the random swapping may increase the frequency of the rare word and let you learn a similar vector for both.

Update the `main` function to take in an optional file with a list of synonyms. Update the `train` function (as you see fit) so that if synonyms are provided, during training tokens with synonyms are recognized and randomly replaced with a token from the set of synonyms (which includes the original token too!)

Train the synonym-aware model for the same number of epochs as you used to solve Task 3 and save this model to file.

As you might notice, the `synonyms.txt` has synonyms that only make sense in some contexts! Many words have multiple meanings and not all of these meanings are equally common. More over a word can have two parts of speech (e.g., be a noun and a verb), which word2vec is unaware of when modeling meaning. As a result, the word vectors you learn are effectively trying to represent *all* the meanings for a word in a *single* vector—a tough challenge! The synonyms we've provided are a initial effort of identifying common synonyms, yet even these may shift the word vectors in unintended ways. In the next problem, you'll assess whether your changes have improved the quality of the models.

Load both the original (not-synonym-aware) model and your new model into a notebook and examine the nearest neighbors of some of the same words. For some of the words in the `synonyms.txt` file, which vector space learns word vectors that have more reasonable nearest neighbors? Does the new model produce better vectors, in your opinion? Show at least five examples of nearest neighbors that you think help make your case and write at least two sentences describing why you think one model is better than the other.

References

Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26: 2814–2822, 2013.

Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.