SI 630: Homework 2: Word2Vec (For Part 4)

This homework will have you implementing word2vec using PyTorch and let you familiarize yourself with building more complex neural networks and the larger PyTorch development infrastructure.

Broadly, this homework consists of a few major parts:

1. Implement a `Corpus` class that will load the dataset and convert it to a sequence of token ids
2. Implement negative sampling to select tokens to be used as negative examples of words in the context
3. Create your dataset of positive and negative examples per context and load it into PyTorch's `DataLoader` to use for sampling
4. Implement a `Word2Vec` class that is a PyTorch neural network
5. Implement a training loop that samples a *batch* of target words and their respective positive/negative context words
6. Implement rare word removal and frequent word subsampling
7. Run your model on the full dataset for at least one epoch
8. Do the exploratory parts of the homework
9. Make a copy of this notebook and change your implementation so it learns word vectors with less bias

After Step 5, you should be able to run your word2vec implementation on a small dataset and verify that it's learning correctly. Once you can verify everything is working, proceed with steps 6 and beyond. **Please note that this list is a general sketch and the homework PDF has the full list/description of to-dos and all your deliverables.**

In [2]:

```
# ! pip install tensorboard
# ! pip install -U gensim
# ! conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

In [12]:

```python
import numpy as np
import pandas as pd
import seaborn as sns
import torch
from torch.utils.data import Dataset, DataLoader
from torch.nn.functional import cosine_similarity

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
from tqdm.auto import tqdm, trange
from collections import Counter
import random
from torch import optim

from torch.utils.tensorboard import SummaryWriter

# Helpful for computing cosine similarity--Note that this is NOT a similarity!
from scipy.spatial.distance import cosine

# Handy command-line argument parsing
import argparse

# Sort of smart tokenization
from nltk.tokenize import RegexpTokenizer

# We'll use this to save our models
from gensim.models import KeyedVectors

#
# IMPORTANT NOTE: Always set your random seeds when dealing with stochastic
# algorithms as it lets your bugs be reproducible and (more importantly) it lets
# your results be reproducible by others.
#
random.seed(1234)
np.random.seed(1234)
torch.manual_seed(1234)
```

```
<torch._C.Generator at 0x267e5623750>
```

Create a class to hold the data

Before we get to training word2vec, we'll need to process the corpus into some representation. The `Corpus` class will handle much of the functionality for corpus reading and keeping track of which word types belong to which ids. The `Corpus` class will also handle the crucial functionality of generating negative samples for training (i.e., randomly-sampled words that were not in the target word's context).

Some parts of this class can be completed after you've gotten word2vec up and running, so see the notes below and the details in the homework PDF.

In [5]:

```
class Corpus:

    def __init__(self):

        self.tokenizer = RegexpTokenizer(r'\w+')

        # These state variables become populated with function calls
        #
        # 1. load_data()
        # 2. generate_negative_sampling_table()
        #
        # See those functions for how the various values get filled in

        self.word_to_index = {} # word to unique-id
        self.index_to_word = {} # unique-id to word

        # How many times each word occurs in our data after filtering
        self.word_counts = Counter()

        # A utility data structure that lets us quickly sample "negative"
        # instances in a context. This table contains unique-ids
        self.negative_sampling_table = []

        # The dataset we'll use for training, as a sequence of unqiue word
        # ids. This is the sequence across all documents after tokens have been
        # randomly subsampled by the word2vec preprocessing step
        self.full_token_sequence_as_ids = None

        self.probability = {}

    def tokenize(self, text):
        '''
        Tokenize the document and returns a list of the tokens
        '''
        return self.tokenizer.tokenize(text)

    def load_data(self, file_name, min_token_freq):
        '''
        Reads the data from the specified file as long long sequence of text
        (ignoring line breaks) and populates the data structures of this
        word2vec object.
        '''


        # Step 1: Read in the file and create a long sequence of tokens for
        # all tokens in the file
        all_tokens = []
        print('Reading data and tokenizing')
        with open(file_name, "r", encoding="utf-8") as file:
            all_tokens = self.tokenize(file.read().lower())

        # Step 2: Count how many tokens we have of each type
        print('Counting token frequencies')
        for t in all_tokens:
            self.word_counts[t] += 1

        # Step 3: Replace all tokens below the specified frequency with an <UNK>
        # token.
        #
        # NOTE: You can do this step later if needed
        print("Performing minimum thresholding")
        unk_token = set()
        for word in self.word_counts.keys():
            if self.word_counts[word] < min_token_freq:
                unk_token.add(word)
        for i in range(len(all_tokens)):
            if all_tokens[i] in unk_token:
                all_tokens[i] = "<UNK>"

        # Step 4: update self.word_counts to be the number of times each word
        # occurs (including <UNK>)
        print('Updating token frequencies after filtering <UNK>')
        self.word_counts.clear()
        for t in all_tokens:
            self.word_counts[t] += 1

        # Step 5: Create the mappings from word to unique integer ID and the
        # reverse mapping.
        print("Create words and id mapping")
        i = 0
        for word in self.word_counts.keys():
            self.word_to_index[word] = i
            self.index_to_word[i] = word
            i += 1

        # Step 6: Compute the probability of keeping any particular *token* of a
        # word in the training sequence, which we'll use to subsample. This subsampling
        # avoids having the training data be filled with many overly common words
        # as positive examples in the context
        print("Compute probability of subsampling")
        total_num = sum(self.word_counts.values())
        for word in self.word_counts.keys():
            p = self.word_counts[word] / total_num
            self.probability[word] = ((p / 0.001) ** 0.5 + 1) * (0.001 / p)

        # Step 7: process the list of tokens (after min-freq filtering) to fill
        # a new list self.full_token_sequence_as_ids where
        #
        # (1) we probabilistically choose whether to keep each *token* based on the
        # subsampling probabilities (note that this does not mean we drop
        # an entire word!) and
        #
        # (2) all tokens are convered to their unique ids for faster training.
        #
        # NOTE: You can skip the subsampling part and just do step 2 to get
        # your model up and running.

        # NOTE 2: You will perform token-based subsampling based on the probabilities in
        # word_to_sample_prob. When subsampling, you are modifying the sequence itself
        # (like deleting an item in a list). This action effectively makes the context
        # window  larger for some target words by removing context words that are common
        # from a particular context before the training occurs (which then would now include
        # other words that were previously just outside the window).
        print("Create all token ID list")
        self.full_token_sequence_as_ids = []
        for t in all_tokens:
            if random.random() <= self.probability[t]: # subsampling
                self.full_token_sequence_as_ids.append(self.word_to_index[t])
        # Helpful print statement to verify what you've loaded
        print('Loaded all data from %s; saw %d tokens (%d unique)' \
              % (file_name, len(self.full_token_sequence_as_ids),
                 len(self.word_to_index)))

    def generate_negative_sampling_table(self, exp_power=0.75, table_size=1e6):
        '''
        Generates a big list data structure that we can quickly randomly index into
        in order to select a negative training example (i.e., a word that was
        *not* present in the context).
        '''


        # Step 1: Figure out how many instances of each word need to go into the
        # negative sampling table.
        #
        # HINT: np.power and np.fill might be useful here
        print("Generating negative sampling table")
        negative_sampling_prob = {}
        denominator = sum(np.power(list(self.word_counts.values()), 3/4))
        for word in self.word_counts.keys():
            p = self.word_counts[word] ** 0.75 / denominator
            negative_sampling_prob[word] = p

        # Step 2: Create the table to the correct size. You'll want this to be a
        # numpy array of type int
        self.negative_sampling_table = []

        # Step 3: Fill the table so that each word has a number of IDs
        # proportionate to its probability of being sampled.
        #
        # Example: if we have 3 words "a" "b" and "c" with probabilites 0.5,
        # 0.33, 0.16 and a table size of 6 then our table would look like this
        # (before converting the words to IDs):
        #
        # [ "a", "a", "a", "b", "b", "c" ]
        for word in self.word_counts.keys():
            n = int(negative_sampling_prob[word] * table_size)
            self.negative_sampling_table += n * [self.word_to_index[word]]


    def generate_negative_samples(self, cur_context_word_id, num_samples):
        '''
        Randomly samples the specified number of negative samples from the lookup
        table and returns this list of IDs as a numpy array. As a performance
        improvement, avoid sampling a negative example that has the same ID as
        the current positive context word.
        '''

        results = []

        # Create a list and sample from the negative_sampling_table to
        # grow the list to num_samples, avoiding adding a negative example that
        # has the same ID as the current context_word
        while len(results) < num_samples:
            negative_sample = random.sample(self.negative_sampling_table, 1)
            if negative_sample not in cur_context_word_id:
                results.append(negative_sample[0])
        return results
```

Create the corpus

Now that we have code to turn the text into training data, let's do so. We've provided several files for you to help:

- `wiki-bios.DEBUG.txt` -- use this to debug your corpus reader
- `wiki-bios.10k.txt` -- use this to debug/verify the whole word2vec works
- `wiki-bios.med.txt` -- use this when everything works to generate your vectors for later parts
- `wiki-bios.HUGE.txt.gz` -- *do not use this* unless (1) everything works and (2) you really want to test/explore. This file is not needed at all to do your homework.

We recommend startin to debug with the first file, as it is small and fast to load (quicker to find bugs). When debugging, we recommend setting the `min_token_freq` argument to 2 so that you can verify that part of the code is working but you still have enough word types left to test the rest.

You'll use the remaining files later, where they're described.

In the next cell, create your `Corpus`, read in the data, and generate the negative sampling table.

In [6]:

```
corpus = Corpus()
corpus.load_data('wiki-bios.med.txt', 2)
corpus.generate_negative_sampling_table()
```

```
Reading data and tokenizing
Counting token frequencies
Performing minimum thresholding
Updating token frequencies after filtering <UNK>
Create words and id mapping
Compute probability of subsampling
Create all token ID list
Loaded all data from wiki-bios.med.txt; saw 17831398 tokens (189001 unique)
Generating negative sampling table
```

Generate the training data

Once we have the corpus ready, we need to generate our training dataset. Each instance in the dataset is a target word and positive and negative examples of contexts words. Given the target word as input, we'll want to predict (or not predict) these positive and negative context words as outputs using our network. Your task here is to create a python `list` of instances.

Your final training data should be a list of tuples in the format ([target_word_id], [word_id_1, ...], [predicted_labels]), where each item in the list is a list:

1. The first item is a list consisting only of the target word's ID.
2. The second item is a list of word ids for both context words and negative samples
3. The third item is a list of labels to predicted for each of the word ids in the second list (i.e., `1` for context words and `0` for negative samples).

You will feed these tuples into the PyTorch `DatasetLoader` later that will do the converstion to `Tensor` objects. You will need to make sure that all of the lists in each tuple are `np.array` instances and are not plain python lists for this `Tensor` converstion to work.

In [7]:

```
window_size = 2
num_negative_samples_per_target = 2
context_size = (2 * window_size) * (1 + num_negative_samples_per_target)

training_data = []

# Loop through each token in the corpus and generate an instance for each,
# adding it to training_data
for i in tqdm(range(len(corpus.full_token_sequence_as_ids))):
    target_id = [corpus.full_token_sequence_as_ids[i]]
    if corpus.index_to_word[target_id[0]] == "<UNK>":
        continue
    context_id = [] # a list of context word ids according to current target
    context_label = [] # a list of ccontext label according to current contect id
    # generate positive sample
    for j in range(i - window_size, i + window_size + 1):
        if j < 0 or j == i or j >= len(corpus.full_token_sequence_as_ids):
            continue
        context_id.append(corpus.full_token_sequence_as_ids[j])
        context_label.append(1.0)
    # generate negative sample
    negative_sample_size = context_size - len(context_id)
    negative_samples = corpus.generate_negative_samples(context_id, negative_sample_size)
    context_id += negative_samples
    context_label += negative_sample_size * [0.0]
    # appending an entry of training sample to training data
    training_data.append(
        (np.array(target_id), np.array(context_id), np.array(context_label))
    )

    # TIPS:
    # For exach target word in our dataset, select context words
    # within +/- the window size in the token sequence

    # For each positive target, we need to select negative examples of
    # words that were not in the context. Use the num_negative_samples_per_target
    # hyperparameter to generate these, using the generate_negative_samples()
    # method from the Corpus class

    # NOTE: this part might not make sense until later when you do the training
    # so feel free to revisit it to see why it happens.
    #
    # Our training will use batches of instances together (compare that
    # with HW1's SGD that used one item at a time). PyTorch will require
    # that all instances in a batches have the same size, which creates an issue
    # for us here since the target wordss at the very beginning or end of the corpus
    # have shorter contexts.
    #
    # To work around these edge-cases, we need to ensure that each instance has
    # the same size, which means it needs to have the same number of positive
    # and negative examples. Since we are short on positive examples here (due
    # to the edge of the corpus), we can just add more negative samples.
    #
    # YOUR TASK: determine what is the maximum number of context words (positive
    # and negative) for any instance and then, for instances that have fewer than
    # this number of context words, add in negative examples.
    #
    # NOTE: The maximum is fixed, so you can precompute this outside the loop
    # ahead of time.
```

```
100%                    17831398/17831398 [05:29<00:00, 53004.48it/s]
```

Create the network

We'll create a new neural network as a subclass of `nn.Module` like we did in Homework 1. However, *unlike* the network you built in Homework 1, we do not need to used linear layers to implement word2vec. Instead, we will use PyTorch's `Emedding` class, which maps an index (e.g., a word id in this case) to an embedding.

Roughly speaking, word2vec's network makes a prediction by computing the dot product of the target word's embedding and a context word's embedding and then passing this dot product through the sigmoid function (σ) to predict the probability that the context word was actually in the context. The homework write-up has lots of details on how this works. Your `forward()` function will have to implement this computation.

In [8]:

```
class Word2Vec(nn.Module):

    def __init__(self, vocab_size, embedding_size, context_size):
        super(Word2Vec, self).__init__()

        # Save what state you want and create the embeddings for your
        # target and context words
        self.vocab_size = vocab_size
        self.embedding_size = embedding_size # feature
        self.context_size = context_size # (2 * window_size) * (1 + num_negative_samples_per_target)
        self.target_embeddings = nn.Embedding(vocab_size, embedding_size)
        self.context_embeddings = nn.Embedding(vocab_size, embedding_size)
        self.sig = nn.Sigmoid()

        # Once created, let's fill the embeddings with non-zero random
        # numbers. We need to do this to get the training started.
        #
        # NOTE: Why do this? Think about what happens if all the embeddings
        # are all zeros initially. What would the predictions look like for
        # word2vec with these embeddings and how would the updated work?

        self.init_emb(init_range=0.5/self.vocab_size)

    def init_emb(self, init_range):

        # Fill your two embeddings with random numbers uniformly sampled
        # between +/- init_range
        self.target_embeddings.weight.data.uniform_(-init_range, init_range)
        self.context_embeddings.weight.data.uniform_(-init_range, init_range)

    def forward(self, target_word_id, context_word_ids):
        '''
        Predicts whether each context word was actually in the context of the target word.
        The input is a tensor with a single target word's id and a tensor containing each
        of the context words' ids (this includes both positive and negative examples).
        '''

        # NOTE 1: This is probably the hardest part of the homework, so you'll
        # need to figure out how to do the dot-product between embeddings and return
        # the sigmoid. Be prepared for lots of debugging. For some reference,
        # our implementation is three lines and really the hard part is just
        # the last line. However, it's usually a matter of figuring out what
        # that one line looks like that ends up being the hard part.

        # NOTE 2: In this homework you'll be dealing with *batches* of instances
        # rather than a single instance at once. PyTorch mostly handles this
        # seamlessly under the hood for you (which is very nice) but batching
        # can show in weird ways and create challenges in debugging initially.
        # For one, your inputs will get an extra dimension. So, for example,
        # if you have a batch size of 4, your input for target_word_id will
        # really be 4 x 1. If you get the embeddings of those targets,
        # it then becomes 4x50! The same applies to the context_word_ids, except
        # that was alreayd a list so now you have things with shape
        #
        #    (batch x context_words x embedding_size)
        #
        # One of your tasks will be to figure out how to get things lined up
        # so everything "just works". When it does, the code looks surprisingly
        # simple, but it might take a lot of debugging (or not!) to get there.
        # NOTE 3: We *strongly* discourage you from looking for existing
        # implementations of word2vec online. Sadly, having reviewed most of the
        # highly-visible ones, they are actually wrong (wow!) or are doing
        # inefficient things like computing the full softmax instead of doing
        # the negative sampling. Looking at these will likely leave you more
        # confused than if you just tried to figure it out yourself.

        # NOTE 4: There many ways to implement this, some more efficient
        # than others. You will want to get it working first and then
        # test the timing to see how long it takes. As long as the
        # code works (vector comparisons look good) you'll receive full
        # credit. However, very slow implementations may take hours(!)
        # to converge so plan ahead.


        # Hint 1: You may want to review the mathematical operations on how
        # to compute the dot product to see how to do these

        # Hint 2: the "dim" argument for some operations may come in handy,
        # depending on your implementation

        # Hint 3: printing the shape of the tensors can come in very handy when
        # debugging to see where things aren't lining up

        # TODO: Implement the forward pass of word2vec
        target_embedding = self.target_embeddings(target_word_id)
        context_embedding = self.context_embeddings(context_word_ids)

        return self.sig(torch.bmm(context_embedding.reshape(-1, self.context_size, self.embedding_size),
                                  target_embedding.reshape(-1, self.embedding_size, 1)))
```

Train the network!

Now that you have data in the right format and a neural network designed, it's time to train the network and see if it's all working. The trainin code will look surprisingly similar at times to your pytorch code from Homework 1 since all networks share the same base training setup. However, we'll add a few new elements to get you familiar with more common training techniques.

For all steps, be sure to use the hyperparameters values described in the write-up.

1. Initialize your optimizer and loss function
2. Create your network
3. Load your dataset into PyTorch's `DataLoader` class, which will take care of batching and shuffling for us (yay!)
4. Create a new `SummaryWriter` to periodically write our running-sum of the loss to a tensorboard
5. Train your model

Two new elements show up. First, we'll be using `DataLoader` which is going to sample data for us and put it in a batch (and also convert the data to `Tensor` objects. You can iterate over the batches and each iteration will return all the items eventually, one batch at a time (a full epoch's worth).

The second new part is using `tensorboard` . As you might have noticed in Homework 1, training neural models can take some time. TensorBoard (https://www.tensorflow.org/tensorboard/) is a handy web-based view that you can check during training to see how the model is doing. We'll use it here and periodically log a running sum of the loss after a set number of steps. The Homework write up has a plot of what this looks like. We'll be doing something simple here with tensorboard but it will come in handy later as you train larger models (for longer) and may want to visually check if your model is converging. TensorBoard was initially written for another deep learning framework, TensorFlow, but proved so useful it was ported to work in PyTorch too and is easy to integrate (https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html).

To start training, we recommend training on the `wiki-bios.10k.txt` dataset. This data is small enough you can get through an epoch in a few minutes (or less) while still being large enough you can test whether the model is learning anything by examining common words. Below this cell we've added a few helper functions that you can use to debug and query your model. In particular, the `get_neighbors()` function is a great way to test: if your model has learned anything, the nearest neighbors for common words should seem reasonable (without having to jump through mental hoops). An easy word to test on the `10k` data is "january" which should return month-related words as being most similar.

NOTE: Since we're training biographies, the text itself will be skewed towards words likely to show up biographices--which isn't necessary like "regular" text. You may find that your model has few instances of words you think are common, or that the model learns poor or unusual neighbors for these. When querying the neighbors, it can help to think of which words you think are likely to show up in biographies on Wikipedia and use those as probes to see what the model has learned.

Once you're convinced the model is learning, switch to the `med` data and train your model as specified in the PDF. Once trained, save your model using the `save()` function at the end of the notebook. This function records your data in a common format for word2vec vectors and lets you load the vectors into other libraries that have more advanced functionality. In particular, you can use the gensim (https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html) code in other notebook included to explore the vectors and do simple vector analogies.

In [9]:

```
# run this in command line!
# ! tensorboard --logdir=runs # enable tensor board
# http://localhost:6006/
```

In [39]:

```
def cosine_similarity_for_bias(model, word_to_index, word_one, word_two):
    try:
        word_one_index = word_to_index[word_one]
        word_two_index = word_to_index[word_two]
    except KeyError:
        return 0

    embedding_one = model.target_embeddings(torch.LongTensor([word_one_index]).cuda())
    embedding_two = model.target_embeddings(torch.LongTensor([word_two_index]).cuda())
    similarity = 1 - abs(float(cosine(embedding_one.detach().cpu().numpy(),
                                      embedding_two.detach().cpu().numpy())))
    return similarity

# def bias_measuring_function(model, corpus):
#     man_woman_sim = cosine_similarity_for_bias(model, corpus.word_to_index, "man", "woman")
#     return 1 - max(man_woman_sim, 0)
```

In [53]:

```
# TODO: Set your training stuff, hyperparameters, models, tensorboard writer etc. here
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# learning parameters
batch_size = 512
vocab_size = len(corpus.word_counts)
embedding_size = 50
context_size = (2 * window_size) * (1 + num_negative_samples_per_target) # 12
window_size = 2

min_token_freq = 5
lr = 1e-4
num_epoch = 5
max_step = None
lambda1 = 0.8
lambda2 = 0.2

model = Word2Vec(vocab_size, embedding_size, context_size)
criterion = nn.BCELoss()
optimizer = optim.AdamW(model.parameters(), lr = lr)

# init the model
model = model.to(device)
train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True)
writer = SummaryWriter()

# HINT: wrapping the epoch/step loops in nested tqdm calls is a great way
# to keep track of how fast things are and how much longer training will take

for epoch in range(num_epoch):
    print("Epoch ", epoch)
    last_loss_sum = 0
    loss_sum = 0

    # TODO: use your DataLoader to iterate over the data
    for step, data in enumerate(tqdm(train_dataloader)):

        # NOTE: since you created the data as a tuple of three np.array instances,
        # these have now been converted to Tensor objects for us
        target_ids, context_ids, labels = data

        # for GPU training
        target_ids = target_ids.cuda()
        context_ids = context_ids.cuda()
        labels = labels.cuda()

        # TODO: Fill in all the training details here
        optimizer.zero_grad()
        outputs = torch.reshape(model(target_ids, context_ids), (-1, context_size))
        loss = lambda1 * criterion(outputs.float(), labels.float()) + \
            lambda2 * (1 - cosine_similarity_for_bias(model, corpus.word_to_index, "man", "woman"))
        loss_sum += loss
        loss.backward()
        optimizer.step()

        # TODO: Based on the details in the Homework PDF, periodically
        # report the running-sum of the loss to tensorboard. Be sure
        # to reset the running sum after reporting it.
        step_sum = step + epoch * len(train_dataloader)
        if step % 1000 == 0 and step_sum != 0:
            last_loss_sum = loss_sum
            writer.add_scalar('Loss/train', loss_sum, step_sum)
            loss_sum = 0

        # TODO: it can be helpful to add some early stopping here after
        # a fixed number of steps (e.g., if step > max_steps)
        if max_step and step_sum > max_step:
            print("Reach the max step. Early stop.")
            break
        if step_sum % 100 == 0 and step_sum != 0:
            if abs(last_loss_sum - loss_sum) <= 0.001:
                print("The model converge. Early stop.")
                break

# once you finish training, it's good practice to switch to eval.
model.eval()
```

```
Epoch  0
```

```
100%                                          34658/34658 [04:23<00:00, 157.91it/s]
```

```
Epoch  1
```

```
100%                                          34658/34658 [03:48<00:00, 158.74it/s]
```

```
Epoch  2
```

```
100%                                          34658/34658 [03:45<00:00, 121.20it/s]
```

```
Epoch  3
```

```
100%                                          34658/34658 [03:43<00:00, 155.51it/s]
```

```
Epoch  4
```

```
100%                                          34658/34658 [03:57<00:00, 149.56it/s]
```

```
Word2Vec(
  (target_embeddings): Embedding(189001, 50)
  (context_embeddings): Embedding(189001, 50)
  (sig): Sigmoid()
)
```

Verify things are working

Once you have an initial model trained, try using the following code to query the model for what are the nearest neighbor of a word. This code is intended to help you debug

In [59]:

```
def get_neighbors(model, word_to_index, target_word):
    """
    Finds the top 10 most similar words to a target word
    """
    outputs = []
    for word, index in tqdm(word_to_index.items(), total=len(word_to_index)):
        similarity = compute_cosine_similarity(model, word_to_index, target_word, word)
        result = {"word": word, "score": similarity}
        outputs.append(result)

    # Sort by highest scores
    neighbors = sorted(outputs, key=lambda o: o['score'], reverse=True)
    return neighbors[1:11]

def compute_cosine_similarity(model, word_to_index, word_one, word_two):
    '''
    Computes the cosine similarity between the two words
    '''
    try:
        word_one_index = word_to_index[word_one]
        word_two_index = word_to_index[word_two]
    except KeyError:
        return 0

    embedding_one = model.target_embeddings(torch.LongTensor([word_one_index]))
    embedding_two = model.target_embeddings(torch.LongTensor([word_two_index]))
    similarity = 1 - abs(float(cosine(embedding_one.detach().numpy(),
                                      embedding_two.detach().numpy())))
    return similarity
```

In [61]:

```
# get_neighbors(model.cpu(), corpus.word_to_index, "michigan")
```

Save your model!

Once you have a fully trained model, save it using the code below. Note that we only save the `target_embeddings` from the model, but you could modify the code if you want to save the context vectors--or even try doing fancier things like saving the concatenation of the two or the average of the two!

In [71]:

```
def save(model, corpus, filename):
    '''
    Saves the model to the specified filename as a gensim KeyedVectors in the
    text format so you can load it separately.
    '''

    # Creates an empty KeyedVectors with our embedding size
    kv = KeyedVectors(vector_size=model.embedding_size)
    vectors = []
    words = []
    # Get the list of words/vectors in a consistent order
    for index in trange(model.target_embeddings.num_embeddings):
        word = corpus.index_to_word[index]
        vectors.append(model.target_embeddings(torch.LongTensor([index])).detach().numpy()[0])
        words.append(word)

    # Fills the KV object with our data in the right order
    kv.add_vectors(words, vectors)
    kv.save_word2vec_format(filename, binary=False)

model_name = "third_model"
save(model.cpu(), corpus, "./models/" + model_name + ".kv")
```

```
  0%|          | 0/189001 [00:00<?, ?it/s]
```

FINAL PART: DO THIS LAST AND READ CAREFULLY

Before you start this part, you need to have a fully working solution and completed the exploratory part of the assignment.

Once you are ready, create a copy of your working notebook and call it `Debiased Word2Vec.ipynb` **. Do not do this part in your working code for the assignment!!!**

Seriously, save your code in a new file and then start reading the rest of these instructions there.

Ok, hopefully you're reading these in a new file... For this last part of the assignment, we're going to *change* how word2vec learns at a fundamental level.

As you might have noticed in your exploratory analysis, the word2vec model learns to weird and sometimes biased associations between words. In particular, your word2vec model has likely learned some unfortunate gender biases, e.g., that the vector for "nurse" is closer to "woman" than "man". The algorithm itself isn't to blame since it is learning these from a corpus (here, Wikipedia biographies) that contain biases already based on how people write. Wikipedia is (http://markusstrohmaier.info/documents/2015_icwsm2015_wikipedia_gender.pdf) well (http://dcs.gla.ac.uk/~mounia/Papers/wiki_gender_bias.pdf) known (https://www.academia.edu/download/64856696/genderanalysisofWikipediabiostext_self_archived.pdf) for having gender biases in how it writes about men and women.

Side note: Some times this bias-learning behavior is useful: We can use word2vec to uncover these biases and analyze their trends, like this PNAS paper did for looking at bias in news writing along multiple dimensions of identity (https://www.pnas.org/content/pnas/115/16/E3635.full.pdf)

In this last part of the homework, we'll ask how we might try to *prevent* these biases by modifying the training. You won't need to solve this problem by any means, but the act of trying to reduce the biases will open up a whole new toolbox for how you (the experimenter/practioner) can change how and what models learn.

There are many potential ways to *debias* word embeddings so that their representations are not skewed along one "latent dimension" like gender. In this homework, you'll be trying one of a few different ideas for how to do it. **You are not expected to solve gender bias! This part of the assignment is to have to start grappling with a hard challenge but there is no penalty for doing less-well!**

One common technique to have models avoid learning bias is similar to another one you already—**regularization**. In Logistic Regression, we could use L2 regularization to have our model avoid learning β weights that are overfit to specific or low-frequency features by adding a regularizer penalty where the larger the weight, the more penalty the model paid. Recall that this forces the model to only pick the most useful (generalizable) weights, since it has to pay a penalty for any non-zero weight.

In word2vec, we can adapt the idea to think about whether our model's embeddings are closer or farther to different gender dimensions. For example, if we consider the embedding for "president", ideally, we'd want it to be equally similar to the embeddings for "man" and "woman". One idea then is to penalize the model based on how uneven the similarity is. We can do this by directly modifying the loss:

```
loss = loss_criteron(preds, actual_vals) + some_bias_measuring_function(model)
```

Here, the `some_bias_measuring_function` function takes in your model as input and returns how much bias you found. Continuing our example, we might implement it in pseudocode as

```
def some_bias_measuring_function(model):
    pres_woman_sim = cosine_similarity(model, "president", "woman")
    pres_man_sim = cosine_similarity(model, "president", "man")
    return abs(pres_woman_sim - pres_man_sim)
```

This simple example would penalize the model for learning a representation of "president" that is more simular to one of the two words. Of course, this example is overly simple. Why just "president"? Why just "man" and "woman"? Why not other words or other gender-related words or other gender identities??

Another idea might be to just make the vectors for "man" and "woman" be as similar as possible:

```
def some_bias_measuring_function(model):
    # cosine similarity is in [-1,1] but we mostly expect it in [0,1]
    man_woman_sim = cosine_similarity(model, "man", "woman")
    # penalize vectors that are not maximally similar, and avoid the edge case
    # of negative cosine similarity
    return 1 - max(man_woman_sim, 0)
```

All of this works in practice because PyTorch is fantastic about tracking the gradient with respect to the loss. This ability lets us easily define a loss function so that our word2vec model (1) learns to predict the right context words while (2) avoids learning biases. If we compare this code to the numpy part of Homework 1, it's easy to see how powerful PyTorch can be as an experimenter for helping you control what and how your models learn!

Your task is to expand this general approach by coming up with an extension to word2vec that adds some new term to the `loss` value that penalizes bias in the gender dimension. There is no right way to do this and even some right-looking approaches may not work—or might word but simultaneously destroy the information in the word vectors (all-zero vectors are unbiased but also uninformative!).

Suggestion: You may need to weight your bias term in the loss function (remember that $\lambda_1 x_1 + \lambda_2 x_2$ interpolation? This is sort of similar) so that your debiasing regularizer doesn't overly penalize your model.

Once you have generated your model, record word vector similarities for the pairs listed on canvas in `word-pair-similarity-predictions.csv` where your file writes a result like

```
word1,word2,sim
dog,puppy,0.91234123
woman,preseident,0.81234
```

You'll record the similarity for each pair of words in the file and upload it to CodaLab, which is kind of like Kaggle but lets use a custom scoring program. We'll evaluate your embeddings based on how unbiased they are and how much information they still capture after debiasing. **Your grade does not depend on how well you do in CodaLab, just that you tried something and submitted.** However, the CodaLab leaderboard will hopefully provide a fun and insightful way of comparing just how much bias we can remove from our embeddings.

The CodaLab link will be posted to Piazza

In [54]:

```python
sim_pred_df = pd.read_csv("word_pair_similarity_predictions.csv")
for i, row in tqdm(sim_pred_df.iterrows()):
    sim_pred_df["sim"][i] = cosine_similarity_for_bias(model, corpus.word_to_index, row["word1"], row["word2

sim_pred_df
```

```
1630/? [00:12<00:00, 147.91it/s]
```

```
C:\Users\JUNQIC~1\AppData\Local\Temp/ipykernel_5308/2647726112.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning
-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  sim_pred_df["sim"][i] = cosine_similarity_for_bias(model, corpus.word_to_index, row["word1"], row["wor
d2"])
```

	word1	word2	sim
0	old	new	0.266472
1	smart	intelligent	0.797016
2	hard	difficult	0.437628
3	happy	cheerful	0.876806
4	hard	easy	0.703117
...	...	...	...
1625	relatives	sister	0.633990
1626	relatives	she	0.345794
1627	relatives	her	0.569001
1628	relatives	hers	0.887537
1629	relatives	daughter	0.491659

1630 rows × 3 columns

In [55]:

```python
# save to csv
sim_pred_df.to_csv('word_pair_similarity_predictions.csv', index=False)
```

In [ ]: