SI 630: Homework 2

Student: Conan Wu

Unique name: conanwu

Kaggle username: Conan Wu

CodaLab username: mrcwu

Problem 1:

```
    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 = []
```

```
    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
        print('Reading data and tokenizing')
        with open(file_name, encoding="utf-8") as f:
            text = f.read()

            raw_tokens = self.tokenize(text)
            raw_tokens_without_sw = [word for word in raw_tokens if not word in stopwords.words()]
            all_tokens = [word.lower() for word in raw_tokens_without_sw]

#             all_tokens = [word.lower() for word in raw_tokens]

        # Step 2: Count how many tokens we have of each type
        print('Counting token frequencies')
        freqs = Counter(all_tokens)

        # 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_dict = {}
        for index in range(len(all_tokens)):
            if freqs[all_tokens[index]] < min_token_freq:
                UNK_dict[all_tokens[index]] = index
                all_tokens[index] = '<UNK>'
```

```
        # Step 4: update self.word_counts to be the number of times each word
        # occurs (including <UNK>)
        self.word_counts = Counter(all_tokens)

        # Step 5: Create the mappings from word to unique integer ID and the
        # reverse mapping.
#         word_to_index = {}
#         index_to_word = {}
        n = 0
        for i in Counter(all_tokens).keys():
            self.word_to_index[i] = n
            self.index_to_word[n] = i

            n += 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
        postive_subsample = {}
        sum_word_counts = sum(self.word_counts.values())
        for word in self.word_to_index:

            pw = self.word_counts[word] / sum_word_counts
            pkw = (np.sqrt(pw/0.001) + 1)*(0.001/pw)
            postive_subsample[self.word_to_index[word]] = pkw
```

```
        # 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.

        for word in all_tokens:
            if postive_subsample[self.word_to_index[word]] > np.random.rand():
                self.full_token_sequence_as_ids.append(self.word_to_index[word])

        # 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).


        # 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)))
```

Problem 2:

```
    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 sampling table")

        sum_of_weight = np.sum(np.power(list(self.word_counts.values()), exp_power))
        number_of_instances_dict = {}
        for word in self.word_to_index.keys():
            word_prob = np.power(self.word_counts[word], exp_power) / sum_of_weight
            number_of_instances_dict[word] = int(table_size * word_prob)


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


        # 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, number in number_of_instances_dict.items():
            self.negative_sampling_table += [self.word_to_index[word]] * number
```

Problem 3:

```
window_size = 2
num_negative_samples_per_target = 2

training_data = []

# Loop through each token in the corpus and generate an instance for each,
# adding it to training_data
ftsai_dict = dict(enumerate(corpus.full_token_sequence_as_ids))
for word_index in ftsai_dict:
    instance = ([],[],[])
    # 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

    # UNK
    if ftsai_dict[word_index] == corpus.word_to_index['<UNK>']:
        continue

    # Append Target Word
    instance[0].append(ftsai_dict[word_index])

    # Append Positive Sample
    if word_index < window_size:
        for back in list(range(0, word_index)):
            instance[1].append(ftsai_dict[back])
            instance[2].append(1)
        for beyond in list(range( (word_index + 1), (word_index + window_size + 1) )):
            instance[1].append(ftsai_dict[beyond])
            instance[2].append(1)
```

```
    elif word_index >= len(corpus.full_token_sequence_as_ids) - window_size:
        for back in list(range(word_index - window_size, word_index)):
            instance[1].append(ftsai_dict[back])
            instance[2].append(1)
        for beyond in list(range(word_index + 1, len(corpus.full_token_sequence_as_ids))):
            instance[1].append(ftsai_dict[beyond])
            instance[2].append(1)

    else:
        for back in list(range(word_index - window_size, word_index)):
            instance[1].append(ftsai_dict[back])
            instance[2].append(1)
        for beyond in list(range( (word_index + 1), (word_index + window_size + 1) )):
            instance[1].append(ftsai_dict[beyond])
            instance[2].append(1)

    # Append Negative Sample
    # The window size is for both the left and right, so it's 4 context words for 1 target word(with exceptions of target word at beginning and
    # 2 negatives samples for 1 positive context word in the context window, so in total 4 + 4*2 = 12 context words.
    ns_size = num_negative_samples_per_target*window_size*2 + (window_size*2 - len(instance[1]))
    instance[1].extend(corpus.generate_negative_samples(ftsai_dict[word_index], ns_size))
    instance[2].extend([0 for i in range(ns_size)])

    # transform to np array
    index, contexts, labels = instance
    instance_np = (np.array(index), np.array(contexts), np.array(labels))

    # Append Instance to training_data
    training_data.append(instance_np)
```

[107]:
```
# testing
training_data[0:2]
```

[107]:
```
[(array([0]),
  array([    2,     4,  6700,    83, 15087,  7760,    91,  5778, 22387,
          2281,  9652,    33]),
  array([1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])),
 (array([2]),
  array([    0,     4,     5,  4582,   264, 13069,  1384, 14401,   172,
           185,    39,  1456]),
  array([1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]))]
```

Problem 4:

```
class Word2Vec(nn.Module):

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

        # Save what state you want and create the embeddings for your
        # target and context words
        self.target_embeddings = nn.Embedding(vocab_size, embedding_size)
        self.context_embeddings = nn.Embedding(vocab_size, embedding_size)

        # 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/vocab_size)

    def init_emb(self, init_range):

        # Fill your two embeddings with random numbers uniformly sampled
        # between +/- init_range
        # writing style 1
        self.target_embeddings.weight.data.uniform_(-init_range, init_range)
        # writing style 2
        nn.init.uniform_(self.context_embeddings.weight, -init_range, init_range)
```

Problem 5:

```
    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).
        '''
```

```
        w1 = self.target_embeddings(target_word_id)
        w2 = self.context_embeddings(context_word_ids)


        return torch.sigmoid(torch.bmm(w1, torch.transpose(w2, 1, 2)))
```

Problem 6:

```
: # TODO: Set your training stuff, hyperparameters, models, tensorboard writer etc. here
  # variables
  batch_size = 512
  epochs = 1
  learning_rate = 5e-5
  vocab_size = len(corpus.index_to_word)
  embedding_size = 50

  # models and calculations
  model = Word2Vec(vocab_size, embedding_size)
  # model.to(device)
  loss_criterion = torch.nn.BCELoss()
  optimizer = optim.AdamW(model.parameters(), lr=learning_rate)
  dataloader = DataLoader(training_data, 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 trange(epochs):

      loss_sum = 0

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

          # TODO: Fill in all the training details here
```

```
        # 1.forward
        predict = model.forward(target_ids, context_ids)

        # 2.loss
        loss = loss_criterion(torch.squeeze(predict.float()), torch.squeeze(labels.float()))

        # 3.backward
        loss.backward()

        # 4.optimizer step: update the weight
        optimizer.step()

        # 5.optimizer to 0
        optimizer.zero_grad()


        # 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.
        loss_sum += loss.item()
        if step % 100 == 99:
            writer.add_scalar("Loss", loss_sum, step)
            print(f'loss:{loss_sum}')
            loss_sum = 0.0


        # TODO: it can be helpful to add some early stopping here after
        # a fixed number of steps (e.g., if step > max_steps)
#         if step == 10:
#             break


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

Problem 7:

```
100% 1/1 [01:05<00:00, 65.80s/it]
100% 2673/2673 [01:05<00:00, 41.25it/s]
loss:69.31462073326111
loss:69.31277620792389
loss:69.30656826496124
loss:69.295195043087
loss:69.27805954217911
loss:69.25612568855286
loss:69.23010969161987
loss:69.19601953029633
loss:69.15738046169281
loss:69.11603105068207
loss:69.07040053606033
loss:69.01686191558838
loss:68.97091031074524
loss:68.91052913665771
loss:68.85177862644196
loss:68.79254031181335
loss:68.73297220468521
loss:68.65701222419739
loss:68.58906674385071
loss:68.52516949176788
loss:68.44770222902298
loss:68.3714371919632
loss:68.29748958349228
loss:68.2175024151802
loss:68.13225609064102
loss:68.06096345186234
[9]: Word2Vec(
       (target_embeddings): Embedding(23362, 50)
       (context_embeddings): Embedding(23362, 50)
     )
```

Problem 8:

```
# Step 2: Count how many tokens we have of each type
print('Counting token frequencies')
freqs = Counter(all_tokens)

# 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_dict = {}
for index in range(len(all_tokens)):
    if freqs[all_tokens[index]] < min_token_freq:
        UNK_dict[all_tokens[index]] = index
        all_tokens[index] = '<UNK>'
```

Problem 9:

```
# 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
postive_subsample = {}
sum_word_counts = sum(self.word_counts.values())
for word in self.word_to_index:

    pw = self.word_counts[word] / sum_word_counts
    pkw = (np.sqrt(pw/0.001) + 1)*(0.001/pw)
    postive_subsample[self.word_to_index[word]] = pkw
```

Problem 10:

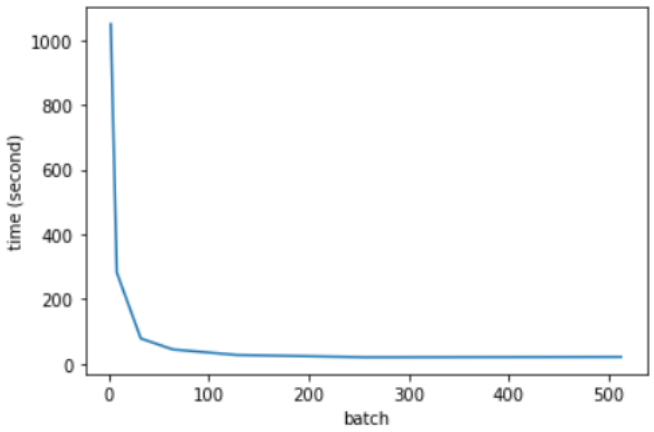
```
# Step 7: process the list of tokens (after min-freq filtering) to fill
# a new list self.full_token_sequence_as_ids where
#
# (1) we probabilistically choose whether to keep each *token* based on the
# subsampling probabilities (note that this does not mean we drop #
# an entire word!) and
#
# (2) 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.

for word in all_tokens:
    if postive_subsample[self.word_to_index[word]] > np.random.rand():
        self.full_token_sequence_as_ids.append(self.word_to_index[word])
```

Problem 11:

```
# 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.
loss_sum += loss.item()
if step % 100 == 99:
    writer.add_scalar("Loss", loss_sum, step)
    print(f'loss:{loss_sum}')
    loss_sum = 0.0
```

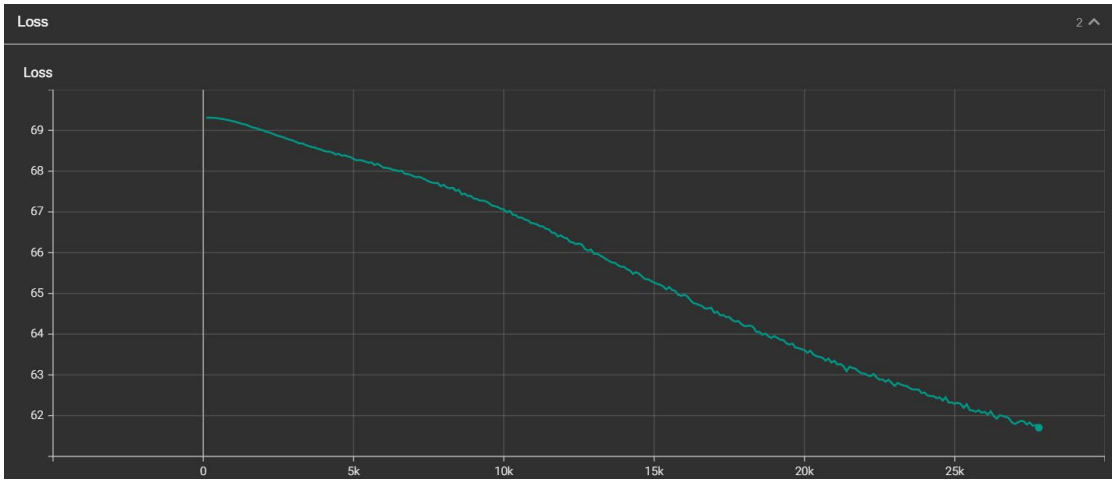
Problem 12:



Eventually I choose 512 as my batch size. This plot shows that if we choose larger batch size, the running time will be faster usually. However, it also depends on the computer's capacity. In this 10k file testing, the running time difference between 256 and 512 batch sizes is not so obvious. It may be that the batch size has already reach my computer's capacity.

Problem 13:

```
100% 1/1 [05:11<00:00, 311.05s/it]
100% 27843/27843 [05:10<00:00, 70.94it/s]
loss:69.3146453499794
loss:69.31335526704788
loss:69.30974572896957
loss:69.30360662937164
loss:69.29560059309006
loss:69.28436970710754
loss:69.27128565311432
loss:69.25725227594376
loss:69.24095022678375
loss:69.22346431016922
loss:69.20337927341461
loss:69.18216985464096
loss:69.15972077846527
loss:69.14504426717758
loss:69.11404347419739
loss:69.08343422412872
loss:69.06522476673126
loss:69.04661893844604
loss:69.0188764333725
loss:68.9977176785469
loss:68.9646812081337
loss:68.95001369714737
loss:68.921362221241
loss:68.888174533844
loss:68.86259293556213
loss:68.84748816490173
loss:68.82060241699219
```



Problem 14:

```
[30]: from gensim.models import KeyedVectors
      from gensim.test.utils import datapath

[31]: word_vectors = KeyedVectors.load_word2vec_format('output_vec_2.kv', binary=False)

[32]: word_vectors['the']

[32]: array([ 0.18802835,  0.29028285, -0.24105756, -0.00692955, -0.3637075 ,
             -0.1391922 ,  0.05505604, -0.02201102, -0.01318244,  0.08471593,
             -0.28282005, -0.10890935,  0.022376  , -0.24436377,  0.3307738 ,
              0.3046057 ,  0.00712686, -0.22134596, -0.2952898 ,  0.2708089 ,
             -0.18050997,  0.10042962,  0.30910814,  0.12133314,  0.3583124 ,
              0.368024  ,  0.2599017 , -0.27963793, -0.14033063, -0.07809907,
             -0.32484785, -0.04139069,  0.31014144,  0.27934352, -0.15740025,
             -0.20978579, -0.2917006 , -0.2008799 , -0.18805508, -0.3411105 ,
              0.23283759,  0.01215435,  0.21576694, -0.03702924, -0.20067894,
             -0.13088608,  0.25922978,  0.00064003, -0.29993364, -0.05171195],
            dtype=float32)
```

Problem 15:

cow	cattle	0.981979
book	story	0.994583
winter	summer	0.987903
taxi	cab	0.980690
tree	maple	0.444423
bed	bedroom	0.997630
roof	ceiling	0.912390
disease	infection	0.932271
arm	shoulder	0.871682
sheep	lamb	0.980650

All of these words have either close relations or very similar ideas. However, the scores seem to be a little unmatched what I expected. For example, I feel like the relation between maple and tree is the same as arm and shoulder, but the scores show a big difference (almost twice).

Problem 16:

After the unbiased training. I got the analogy output:
I try to find the analogy of “man, woman, doctor”

```
[50]: get_analogy('man', 'woman', 'doctor')

[50]: 'mcgill'
```

McGill is an university in Canada. It has a high female to male ratio. (70.7% according to Wiki)

And I try to find the analogy of "science, study, major"

```
[84]: get_analogy('science', 'study', 'major')

[84]: 'play'
```

Sounds very funny. The only study/major without science is playing.

And "school, university" without "study" means graduated.

```
[93]: get_analogy('study', 'school', 'university')

[93]: 'graduated'
```

However, some words analogies may not able to see a clear idea:

```
[92]: get_analogy('study', 'man', 'university')

[92]: 'returning'
```

Reference:

Discussed and co-worked on the assignment with Yun Lee, Michelle Cheng, and Ariel Chang