SI 630: Homework 2 Parts 1-3: Word2Vec

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. Save vectors and word-indexing data for later use in training a classifier

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.**

Estimated performance times

We designed this homework to be run on a laptop-grade CPU, so no GPU is required. If your primary computing device is a tablet or similar device, this homework can also be *developed* on that device but then run on a more powerful machine in the Great Lakes cluster (for free). Such cases are the exception though. Following, we report on the estimated times from our reference implementation for longer-running or data-intensive pieces of the homework. Your timing may vary based on implementation design; major differences in time (e.g., 10x longer) usually point to a performance bug.

- Reading and tokenizing: ~5 seconds
- Subsampling and converting to token ids: ~15 seconds
- Generating the list of training examples: ~15 minutes
- Training one epoch: ~12 minutes

A quick note: the code that generates training instances is probably slower than it needs to be based on how we've described the process. The instructions make generating training tuples conceptually easier by breaking it into steps. *However*, the steps introduce a subtle performance bug from sampling random numbers one at a time when we really need a lot of them (particularly for the negative samples). Fixing this performance but will cause the training data to be generated in ~20% of the time but makes the code a bit ugly to read. Since you only need to generate the training instances once (and could even save these to disk if needed) we've kept the instructions as is. However, if you decide to scale up to the bigger corpora, you might dig into the cell that calls `generate_negative_samples`.

In [ ]:
```python
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader

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

import pickle
import time
```

```
#
# 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)
```

Out[ ]:
```
<torch._C.Generator at 0x28a98b82190>
```

In [ ]:
```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("Running on the GPU")
else:
    device = torch.device("cpu")
    print("Running on the CPU")
```

```
Running on the GPU
```

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 [ ]:
```
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

    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')

        f = open(file_name, "r")
        data = f.read()
        f.close()
        all_tokens = self.tokenize(data)
```

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

        self.word_counts.update(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")

        sorted_tokens = self.word_counts.most_common()
        self.word_counts['<UNK>'] = 0

        for token, count in sorted_tokens:
            if (count < min_token_freq):
                self.word_counts['<UNK>'] += count
                del(self.word_counts[token])

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

        # Step 5: Create the mappings from word to unique integer ID and the
        # reverse mapping.

        index = 0

        for key in self.word_counts:
            self.word_to_index[key] = index
            self.index_to_word[index] = key
            index += 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


        subsampling_table = np.zeros(len(self.word_to_index))
        size = len(all_tokens)

        for key in self.word_counts:
            subsampling_table[self.word_to_index[key]] = count / size

        subsampling_table = (np.power(subsampling_table / 0.001, 0.5) + 1) * 0.001 / subsampling_table

        # 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.

        self.full_token_sequence_as_ids = [0 for _ in range(len(all_tokens))]

        for i, token in enumerate(all_tokens):
            if token not in self.word_to_index:
                token = '<UNK>'

            if (np.random.rand(1)[0] < subsampling_table[self.word_to_index[token]]):
                self.full_token_sequence_as_ids[i] = self.word_to_index[token]
            else:
                self.full_token_sequence_as_ids[i] = -1

        # 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)))

    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")

        sampling_table = np.zeros(len(self.word_to_index))

        for (token, count) in self.word_counts.items():
            sampling_table[self.word_to_index[token]] = count

        sampling_table[self.word_to_index['<UNK>']] = 0

        sampling_table = np.power(sampling_table, exp_power)
        sampling_table = sampling_table / np.linalg.norm(sampling_table) * table_size
        sampling_table = np.asarray(sampling_table, dtype = int)


        # 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 = np.zeros(np.sum(sampling_table), dtype = 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" ]
        #

        prev = 0

        for word, count in enumerate(sampling_table):
            self.negative_sampling_table[prev:prev + count] = word
            prev += count


    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

        for _ in range(num_samples):
            sample_id = np.random.choice(self.negative_sampling_table, 1)[0]
            while (sample_id == cur_context_word_id):
                sample_id = np.random.choice(self.negative_sampling_table, 1)[0]

            results.append(sample_id)


        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 [ ]:
```
corpus = Corpus()
corpus.load_data('reviews-word2vec.med.txt', 5)
corpus.generate_negative_sampling_table()
```

```
Reading data and tokenizing
Counting token frequencies
Performing minimum thresholding
Loaded all data from reviews-word2vec.med.txt; saw 10607824 tokens (35193 unique)
Generating 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 [ ]:
```
window_size = 2
num_negative_samples_per_target = 2

maximum_num = window_size * 2 * (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 index, token_id in enumerate(corpus.full_token_sequence_as_ids):

    # 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.


    word_ids = []
    predicted_labels = []
    context_ids = []

    if (corpus.index_to_word[token_id] == '<UNK>'):
        continue

    for context_index in range(index - window_size, index + window_size + 1):
        if not (context_index != index and context_index in range(len(corpus.full_token_sequence_as_ids))):
            continue
        context_id = corpus.full_token_sequence_as_ids[context_index]

        if (context_id == -1):
            continue

        context_ids.append(context_id)

        word_ids.append(context_id)
        predicted_labels.append(1)

        negative_samples = corpus.generate_negative_samples(context_id, num_negative_samples_per_target)

        word_ids += negative_samples
        predicted_labels += [0 for _ in range(num_negative_samples_per_target)]


    if (len(word_ids) < maximum_num):
        num_samples = maximum_num - len(word_ids)

        negative_samples = corpus.generate_negative_samples(np.random.choice(context_ids, 1)[0], num_samples)

        word_ids += negative_samples
        predicted_labels += [0 for _ in range(num_samples)]

    training_data.append((np.array([token_id], dtype=int), np.array(word_ids, dtype=int), np.array(predicted_lal
```

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 [ ]:

```
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.vocab_size = vocab_size
        self.embedding_size = embedding_size

        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(0.5)

    def init_emb(self, init_range):

        # Fill your two embeddings with random numbers uniformly sampled
        # between +/- init_range

        print("Initilizing embedding layers")

        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


        # TODO: Implement the forward pass of word2vec

        target_embedding = self.target_embeddings(target_word_id)
        context_embedding = self.context_embeddings(context_word_ids)


        return torch.sigmoid(torch.bmm(context_embedding, torch.unsqueeze(target_embedding.squeeze(), 2))).squee
```

In [ ]:

```
x = torch.tensor([[[1, 2]], [[3, 4]]])
y = torch.tensor([[[1, 1], [1, 1], [2, 2], [2, 2]], [[3, 3], [3, 3], [4, 4], [4, 4]]])

print(torch.unsqueeze(x.squeeze(), 2).size())
print(y.size())

torch.bmm(y, torch.unsqueeze(x.squeeze(), 2))
```

```
torch.Size([2, 2, 1])
torch.Size([2, 4, 2])
```

```
Out[ ]: tensor([[[ 3],
          [ 3],
          [ 6],
          [ 6]],

         [[21],
          [21],
          [28],

          [28]]])
```

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 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.

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 code in other notebook included to explore the vectors and do simple vector analogies.

```
In [ ]: # TODO: Set your training stuff, hyperparameters, models, tensorboard writer etc. here

model = Word2Vec(len(corpus.word_to_index), 50).to(device)
loader = DataLoader(dataset = training_data, batch_size = 16, shuffle = True)
writer = SummaryWriter()
criterion = torch.nn.BCELoss()
optimizer = torch.optim.AdamW(model.parameters(), lr = 5e-5)

# 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(1):

    loss_sum = 0

    # TODO: use your DataLoader to iterate over the data
    for step, data in enumerate(tqdm(loader, desc = "Progressing")):

        # 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
        target_ids = target_ids.to(device)
        context_ids = context_ids.to(device)
        labels = labels.to(device)


        # TODO: Fill in all the training details here
        labels = labels.to(torch.float32)

        optimizer.zero_grad()
        prediction = model(target_ids, context_ids)

        loss = criterion(prediction, labels)
        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.

        if step % 100 == 99:
            writer.add_scalar("Loss/train", loss_sum, step + epoch * len(loader))
            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)



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

```
Initilizing embedding layers
  0%|          | 0/1 [00:00<?, ?it/s]
Progressing:   0%|          | 0/653980 [00:00<?, ?it/s]
```

Out[ ]:
```
Word2Vec(
  (target_embeddings): Embedding(35193, 50)
  (context_embeddings): Embedding(35193, 50)
)
```

Problem 12

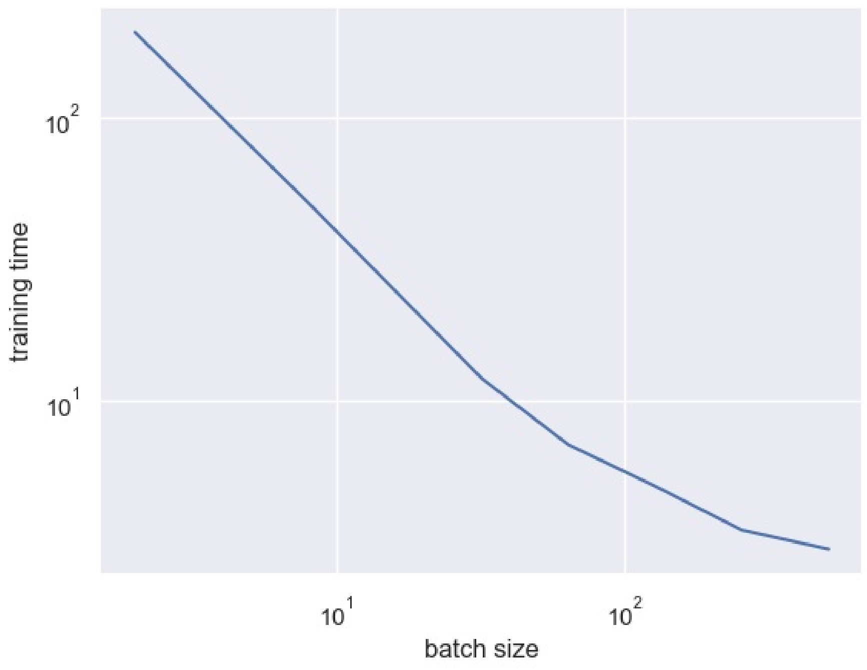
In [ ]:
```
import matplotlib.pyplot as plt

batch_size = [2, 8, 32, 64, 128, 256, 512]
training_time = [200, 50, 12, 7, 5, 3.5, 3]

plt.plot(batch_size, training_time)
plt.xscale('log')
plt.yscale('log')
plt.xlabel('batch size')
plt.ylabel('training time')
plt.show()



## To maximize speed, the largest batches should be chosen, but I tend to use a smaller batch to ensure the eff
```



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

In [ ]:
```
get_neighbors(model, corpus.word_to_index, "recommend")
```

```
  0%|          | 0/35193 [00:00<?, ?it/s]
```

Out[ ]:
```
[{'word': 'enjoy', 'score': 0.7896250486373901},
 {'word': 'suggest', 'score': 0.7717113494873047},
 {'word': 'buy', 'score': 0.7643194198608398},
 {'word': 'finish', 'score': 0.7450742125511169},
 {'word': 'purchase', 'score': 0.7308987379074097},
 {'word': 'recommended', 'score': 0.7307265996932983},
 {'word': 'purchased', 'score': 0.729428768157959},
 {'word': 'figured', 'score': 0.7288399934768677},
 {'word': 'bought', 'score': 0.7205524444580078},
 {'word': 'give', 'score': 0.7155219316482544}]
```

In [ ]:
```
get_neighbors(model, corpus.word_to_index, "son")
```

```
  0%|          | 0/35193 [00:00<?, ?it/s]
```

```
Out[ ]: [{'word': 'daughter', 'score': 0.7873198986053467},
         {'word': 'course', 'score': 0.7200912833213806},
         {'word': 'husband', 'score': 0.6906254887580872},
         {'word': 'money', 'score': 0.6792412400245667},
         {'word': 'interest', 'score': 0.6782874464988708},
         {'word': '10', 'score': 0.6774787902832031},
         {'word': 'liking', 'score': 0.6729979515075684},
         {'word': 'structure', 'score': 0.6718194484710693},
         {'word': 'friends', 'score': 0.66325843334198},
         {'word': 'kids', 'score': 0.6596392393112183}]
```

Save your vectors for the gensim inspection part!

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 [ ]: 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]).to(device)).detach().cpu().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)
```

Save your vectors / data for the pytorch classifier in Part 4!

We'll be to using these vectors later in Part 4. We want to save them in a format that PyTorch can easily use. In particular you'll need to save the *state dict* of the embeddings, which captures all of its information.

```
In [ ]: save(model, corpus, "target_embedding_layer.txt")

          0%|          | 0/35193 [00:00<?, ?it/s]
```

We will also need the mapping from word to index so we can figure out which embedding to use for different words. Save the `corpus` objects mapping to a file using your preferred format (e.g., pickle or json).

```
In [ ]: import json

        with open("word_to_index.txt", 'w') as fp:
            json.dump(corpus.word_to_index, fp)
```

```
In [ ]: with open("index_to_word.txt", 'w') as fp:
            json.dump(corpus.index_to_word, fp)
```

```
In [ ]: torch.save(model.state_dict(), "word2vec.pt")
```

```
In [ ]: print(model.state_dict())
```

```
OrderedDict([('target_embeddings.weight', tensor([[-0.0217,  0.5202,  0.1789,  ...,  0.1694,  0.5171, -0.5530],
        [ 0.2293,  0.2541,  0.3720,  ...,  0.4038,  0.7722, -0.3952],
        [ 0.0727,  0.1542,  0.3411,  ..., -0.3772,  0.6162, -0.3807],
        ...,
        [ 0.3089, -0.2826, -0.2459,  ..., -0.2614,  0.0298, -0.0543],
        [-0.0730,  0.2149, -0.3193,  ...,  0.2372,  0.1421, -0.2747],
        [-0.1399,  0.0855, -0.3225,  ..., -0.1515,  0.1149,  0.3646]],
       device='cuda:0')), ('context_embeddings.weight', tensor([[ 0.1375, -0.0995, -0.1206,  ...,  0.3940,  0.3
641, -0.1690],
        [-0.2580,  0.2129, -0.0762,  ...,  0.1075, -0.1935, -0.1343],
        [ 0.3007, -0.0550,  0.1550,  ...,  0.0902, -0.0632,  0.5644],
        ...,
        [ 0.1848, -0.2809, -0.4599,  ...,  0.2448, -0.0728,  0.2149],
        [-0.1576, -0.1928, -0.2484,  ..., -0.0818,  0.0380, -0.0238],
        [-0.2838,  0.4269,  1.1801,  ..., -0.1211,  1.3603, -0.8584]],
       device='cuda:0'))])
```