Coding Assignment #2

In this assignment, you will implement a neural language model that produces word embeddings based on the original [Word2Vec](https://arxiv.org/abs/1301.3781) paper.

Your corpus for learning word embeddings will be the top 30 most downloaded Project Gutenberg books at the time of assignment creation (don't worry, there's a script). Your model will learn over a vocabulary of 3000 tokens, and will be evaluated in vitro via the language modeling task for which it's being trained, as well as in vivo via the analogical reasoning task we discussed in class. The analogy dataset is curated such that with a vocab size of 3000 and the given books corpus, every analogy word will be represented with a token in the downstream evaluation.

The starter code handles a lot of what you've learned already: turning raw book files into sentences, tokenization, and encoding sentences into machine-readable vectors. The analogical reasoning evaluation is also already implemented.

What you'll need to do is turn your encoded sentences into data pairs (inputs and outputs) that your model can learn from, as well as implement the model itself.

Install some packages

```
# first create a virtualenv
virtualenv -p $(which python3) ./venv

# activate virtualenv
source ./venv/bin/activate

# install packages
pip3 install -r requirements.txt

# download books [creates a local directory called 'books/']
chmod +x get_books.sh
./get_books
```

Train and evaluate model

The training file will throw some errors out of the box. You will need to fill in the TODOs before anything starts to train.
While debugging, consider taking a small subset of the data and inserting break statements in the code and print the values of your variables.

```
Train:
python3 train.py \
    --analogies_fn analogies_v3000_1309.json \
    --data_dir books/ \

Evaluation:
python train.py \
    --analogies_fn analogies_v3000_1309.json \
```

```
    --data_dir books/ \
    --downstream_eval

# add any additional argments you may need
```

Grading

In this assignment you **may not** use HuggingFace libraries and implementations. Anything in the base `torch` library is fair game, however.
This assignment will be scored out of 30 points on the *correctness* and *documentation detail and accuracy*. Note that the report is weighted higher this time; the Report is an opportunity for you to explain your code, choices, and experiments. The point breakdown is:

- [ ] (5pt) Turning the encoded input data into input/output tensors for your model
- [ ] (15pt) Implementation of EITHER the CBOW model or the Continuous Skip-Gram model and associated training paradigm; you can use a fixed` context window for skip-gram rather than sampling
- [ ] (10pt) *Report* your results through an .md file in your submission; discuss your implementation choices and document the performance of your model (both training and validation performance, both in vitro and in vivo) under the conditions you settled on (e.g., what hyperparameters you chose) and discuss why these are a good set. If you implemented any bonus experiments, detail those in clearly delineated sections as well. Finally, in this report I'd also like you to do a little bit of analysis of the released code and discuss what's going on. In particular, what are the in vitro and in vivo tasks being evaluated? On what metrics are they measured? If there are assumptions or simplifications that can affect the metrics, what are they and what might go "wrong" with them that over- or under-estimate model performance?

Remember that each coding assignment will be worth 30 points, for a total of 90 points towards coding assignments which account for 25% of your course grade. The remaining 10 points to round out to 100 are based on discussions we have in class during coding assignment "debrief" sessions (i.e., participation points).

Submission checklist

- Submit a link to your public github repository containing your solution for the homework.
- Your repository should contain:
    1. .py files provided with your solution implementation along with any additional files you may need
    2. `.md` file (see above)
    3. 4 images: `training_loss.png, training_acc.png, validation_loss.png, validation_acc.png`
    4. The text output of the in vivo evaluation for analogies, though this would be prettier as graphs across choices, of course.

Available Bonus Points

You may earn up to 10pt of *bonus points* (and ONLY up to 10) by implementing the following bells and whistles that explore further directions. For these, you will need to compare the performance of the base model against whatever addition you try. Add those details to your report. If you implement bonus items, your base code implementing the main assignment must remain intact and be runnable still. I have ordered these by my rough estimate of the implementation and experiment difficulty of each.

- [ ] (*5pt*) Analyze the performance of your learned embeddings against the much larger-scale-pretraining [word2vec](https://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/), [GLoVE](https://nlp.stanford.edu/projects/glove/), or [FastText](https://fasttext.cc/) embeddings. You may have to modify the gensim vector reading or do some preprocessing to shove pretrained vectors into the right format for the in vivo eval. Your comparison should include experiments that back hypotheses about performance differences you observe.
- [ ] (*5pt*) We have hypothesized that the context window size affects syntactic versus semantic performance. Evaluate that hypothesis with your model by varying the context window and looking for relationships to syntax versus semantic analogical task performance.
- [ ] (*10pt*) Perform a detailed analysis of the data itself or of your model's performance on the data. This bonus is very open ended, and points will be based on the soundness of the implementation as well as the insights gained and written up in the report. For example, you could cluster learned embeddings to look for emergent relationships, analyze which analogies your model gets wrong with high confidence (e.g., most probability mass on the wrong choice at prediction time) and see, qualitatively, if you can identify systematic misclassifications of that type and hypothesize about why they happen, etc.
**- [ ] (*10pt*) Implement the OTHER of CBOW or Continuous Skip-Gram model and compare the performance of the two with the same context window sizes.
**- [ ] (*5pt-10pt*) Add the context window sampling scheme we discussed in class so that your context window varies. Detail your implementation choices carefully and compare performance against a fixed-size window. Note that to make this change for CBOW, you'll have to introduce some tricks to your encoding and switch from SUM to AVERAGE as your aggregation so that different context window sizes produce embeddings of the same magnitude. (Up to 5pt each for implementation to each of CBOW versus Skip-Gram).
- [ ] (*10pt*) Context windows near sentence boundaries are tricky to handle. What if you change the encoding approach to consider words on the other side of sentence boundaries? You'll have to change the encoding functions to get adjacent sentences, but then you can try letting the context window spill between sentence boundaries. How does this affect performance?