CS 287
Assignment 3: (Neural) Language Modeling

Due: Monday, March 7th, 11:59 pm

This assignment focuses on the task of language modeling, a crucial first-step for many natural language applications. In this assignment you will implement several count-based multinomial language models, an influential neural network based language model from the work of Bengio et al. (2003), and an extension to this language model noise contrastive estimation which we discussed in class.

As you complete this assignment, we ask that you submit your results on the test data to the Kaggle competition website at `https://inclass.kaggle.com/c/cs287-hw3` and that you compile your experiences in a write-up based on the template at `https://github.com/cs287/hw_template`.

1 Data and Preprocessing

1.1 Data

The main data for this task is in the `data/` directory. Here is the first few sentences of validation. Each line of the file contains one tokenized sentence.

```
> head data/valid.txt
 consumers may want to move their telephones a little closer to the tv
    set
 <unk> <unk> watching abc 's monday night football can now vote during
    <unk> for the greatest play in N years from among four or five <unk
    > <unk>
 two weeks ago viewers of several nbc <unk> consumer segments started
    calling a N number for advice on various <unk> issues
 and the new syndicated reality show hard copy records viewers '
    opinions for possible airing on the next day 's show
 interactive telephone technology has taken a new leap in <unk> and
    television programmers are racing to exploit the possibilities
 eventually viewers may grow <unk> with the technology and <unk> the
    cost
 but right now programmers are figuring that viewers who are busy
    dialing up a range of services may put down their <unk> control <
    unk> and stay <unk>
```

In addition to we have also included a set of data with a smaller vocabulary (which will be much faster to train on). This data is in `train.1000.txt`. These sentences are identical, but have many more word types replaced with UNK.

We also include dictionary files,

```
> head  data/words.1000.dict
```

This is a mapping from each word in the Penn Treebank to a unique ID. It is important that you use this mapping since it is used for the Kaggle competition data.

1.2 Preprocessing

For this assignment you will write your own preprocessing code in `preprocessing.py`. This code should transform the data representations given above into a format for multiclass prediction. In particular you will be responsible for,

- Mapping the words in the data set to indexes.
- Pre-construct the n-gram windows before word.

Your preprocessing code should include an argument for the size of the n-gram that is used.

Important note: the boundary cases for language modeling are a bit tricky. For each sentence, you must predict the end-of-sentence token represented as "</s>". Conversely you do not predict the start-of-sentence token "<s>" but you do condition on it when predicting words at the beginning of the sentence. Therefore the number of words predicted is size of words in the file + number of sentences (one </s> for each sentence).

For the test set, we have blanked out one word in every sentence, instead writing _#_ where # is the example number . The evaluation, discussed below, will be to give the full distribution over this missing word. Therefore when constructing your `test_input` you should only output the context for these examples.

train_input, train_output, valid_input, valid_output test_input, nwords

2 Code Setup

Write your main code in `HW3.lua`. For this assignment part, you can (and should) use the `nn` library in addition to the standard Torch library.

2.1 Prediction and Evaluation

Be sure you are able to read and output the text data and that you understand the format. Also be sure you are able output your classification results on the test data as a text file. Check that you can upload these to the Kaggle.

The Kaggle submission for this assignment takes on a more complicated form. For each position in the test data, we would like you to output a complete distribution over a given set of words.

For instance the first challenges is given the context C, give the distribution over the words listed above.

```
Q my outright recovery operations grew combustion enthusiastic deadly
   california breathing broadcasters unable tenders practical jacob
   lying premiere longer vermont-slauson seemingly tumbled shipment
   monday expenditure s miners multiples art combat increasingly
   kasparov lab questionable bold promotion regularly texans eighth gaf
    sinking N of excitement window municipalities poverty soliciting
   pressure neighboring accommodate
C no it was n't black _1_
```

In your output file you should give the distribution $p(w_6|w_1, \ldots, w_5)$ renormalized over the 50 word candidate set given above. It will look something like this.

```
ID,Class1,Class2,Class3,...
1,0.2,0.001,0.0012,...
```

Here `Class1` refers to the first word in `words.dict`, `Class2` to the second, etc.

Worst-case if you are not able to scale the full language model, you can still submit to Kaggle using reduced vocabulary training. If you do this, describe the method you use for handling words in the candidate set that are outside of your vocabulary.

2.2 Hyperparameters

Several of the models described have explicit hyperparameters that you will need to tune. It is your responsibility to cleanly separate these out from the models themselves and expose as command-line options. This makes it much easier to run experiments and to utilize experimental scripts.

2.3 Logging and Reporting

As part of the write-up, you will need to report on the training and predictive accuracy of your models. To make this possible, your code should report on various metrics of the model both at training and test time. We will leave it up to you on which metrics to log, but we recommend reporting training speed, training set NLL, training set predictive accuracy, and validation predictive accuracy. It is your responsibility to convince us that the model is correctly training.

3 Models

For this assignment you will implement the three models. We will warm up with a simple count-based language model, then we will add smoothing, and finally we will move on to the full neural network model from Bengio et al. (2003) and finally implement noise contrastive estimation (Gutmann and Hyvärinen, 2010).

3.1 Count-Based Language Model

To start, modify your code from HW1 to implement a simple of n-gram multinomial estimate of $p(w_i|w_{i-n+1}, \ldots w_{i-1})$. Use three different strategies to estimate this distribution,

1. Simple maximum likelihood estimation.
2. Estimation with Laplace smoothing (α).
3. Estimation with Witten-Bell.
4. Estimation with (Modified)Kneser-Ney smoothing .

Use this distribution to calculate the perplexity of the development set with n-grams 2 and 3. The final smoothing technique is discussed in the lecture notes. Report the results for in terms of perplexity on the development set.

3.2 Neural Network Language Models (NNLMs)

To compare, we will also implement a neural network language model for this problem. Unfortunately when using a CPU it is too inefficient to train on this full data set. The issue comes from the partition function, which requires $O(|\mathcal{V}|)$ time to compute each step. To avoid this issue, we have provided a reduced vocabulary data set `data/train.1000.txt` to start with. In the next section we will get to the full data set.

Note the following correspondences,

Math	Torch
sparse $\boldsymbol{xW}$	nn.LookupTable
log softmax	nn.LogSoftMax
tanh	nn.TanH
$L_{cross-entropy}()$	nn.ClassNLL
dense $\boldsymbol{xW} + \boldsymbol{b}$	nn.Linear

The documentation (`https://github.com/torch/nn`) is quite good for all of these models (although it is possible the notation may be slightly different than class).

The training and evaluation code for the model takes a similar form to HW2. Similar tips apply here .

- Train using minibatches (we used size 32). This will significantly speed up training.
- Keep track of the average loss as you train.
- Run on the development set each iteration to see the progress of the model.
- You should not have to run for more than 20 epochs on this data. If you are seeing large changes beyond that, you may have an issue.
- One simple way of regularizing language models is to renormalize the word embeddings to have a max $\ell 2$ norm of 1 after each epoch. We recommend this technique instead of other types of regularization.

3.3 Neural Network Model with Noise Constrastive Estimation

As a final step, retrain your model with NCE as described in the class notes and the papers posted online (the work of Mnih and Kavukcuoglu (2013) is particularly clear).

NCE will require using a slightly different setup than our neural network models so far. The base of the model will be identical. However instead of using a standard softmax and criterion you will need to manually calculate the loss and the gradient with respect to the model, by sampling. You can then use the standard Torch modules to backprop this gradient through the model. To use the learned model, you can either add a softmax to get a full distribution, or simply limit the model to the 50 words in the set.

3.4 Word Embeddings

Once you have trained your model, you can examine the word embeddings that the model produces by exporting the parameters of your `nn.LookupTable`. Write out these parameters to HDF5 and load them into back into Python. There are several interesting experiments that are often run on these embeddings. We ask that you demonstrate the correctness of your approach with some of these methods.

- For a subset of words compute the k-nearest neighbors under dot-product and cosine similarity. What does this show? How do the metrics differ?
- Graph a subset of the word vectors by projecting to 2-dimensions. The simplest method is to use PCA, but there are a number of other techniques. Scikit-Learn has a number of methods for doing this which you can explore through the tutorial at `http://scikit-learn.org/stable/auto_examples/manifold/plot_lle_digits.html`.

3.5 Additional Experiments

Once these models are constructed, you should also report on additional experiments on these data sets. We will leave this aspect open-ended, but suggestion include:

- Include additional features in the input of the model, for instance, capitalization or suffixes. Does this have any effect on the perplexity of the system.
- Experiment with different architectures (Replicate the different setups in Bengio et al. (2003)).
- Modern language modeling experiments are often run on a GPU, an architecture particularly well-suited for computing large partition functions. If you have access to a NVidia GPU, we encourage you to tree these experiments. Follow the instructions on the Torch website for running on this system. It is usually as easy as calling :cuda on your model.
- Experiment with "babbling", i.e. sample from the distribution proposed by the model and feed it back in. What does the generated text look like? What issues does it have?

4 Report and Submission

For your write-up, follow the report template at `https://github.com/cs287/hw_template`. Be sure to include a link to your code, Kaggle ID, and reports on your results.

In addition to submitting your Kaggle results, we also expect you to report on your experimental process. This should include data tables, graphs and discussion of any issues that you may run into.

References

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

Gutmann, M. and Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *International Conference on Artificial Intelligence and Statistics*, pages 297–304.

Mnih, A. and Kavukcuoglu, K. (2013). Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in Neural Information Processing Systems*, pages 2265–2273.