CS 533: Natural Language Processing **(Due: 02/18/20)**

Assignment 2

Instructor: Karl Stratos

- 2 problems: total 30 points (40 with bonus) (18 (+10) + 12)
- No collaboration
- Due by 11:59pm of the due date, no late submission accepted
- Use the provided LaTeX assignment template to write the answers. Upload the code as well.

Problem 1: Log-Linear Models

(2 + 2 + 2 + 2 + 4 + 2 + 2 + 2 (+10) = 18 (+10) points)

You will experiment with a log-linear language model (without BOS and EOS symbols for simplicity). Download the starter kit provided. Use the same training and validation data from A1 (put them in `data/` directory). The files `assignment2.py` and `util.py` contain partially complete code. We will retain the convention from A1 and use the top-10,000 vocab size under the NLTK tokenizer with lowercasing.

1. Take a look at `extract_features` in `util.py`. Basic feature extractors are defined for your benefit. Implement `basic_features1_suffix3` that extracts features in `basic_features1` plus suffixes of length up to 3 of `window[-2]`. You can ignore suffixes whose lengths exceed the length of the word. For example, given a window `[have, an, apple]`, we should get feature types that look like `c-1s1=n∧w=apple` and `c-1s2=an∧w=apple`. Note that the feature string convention is unimportant.

2. Use 10% of the training data and 10% of the validation data (default) and report the number of feature types you extract using `basic_features1`, `basic_features1_suffix3`, and `basic_features2`. Note that we are "cheating" by including features extracted from the validation data converted to indices in our cache so that things are faster at evaluation.

3. Implement `softmax`, which gets a vector and returns the softmax transformation of the vector. For numerical stability, use the following formula:

$$\text{softmax}_i(v) = \frac{e^{v_i - v_{\max}}}{\sum_j e^{v_j - v_{\max}}}$$

   where $v_{\max} = \max_k v_k$. This way we prevent the exponential from exploding. Why is this the same as the usual definition of softmax?

4. Implement `compute_probs`. The cache `self.fcache[window]` contains precomputed feature indices corresponding to the given window. The cache `self.x2ys[tuple(x)]` contains all words y that follow x seen in the data. You must use these caches to speed up computation. Recall that the entry of the score vector `q_` corresponding to target word y is

$$[\texttt{q\_}]_{\text{ind}(y)} = w^\top \phi(x, y) = \sum_{i=1:\ \phi_i(x,y)=1}^{d} w_i$$

5. Implement the gradient update in `do_epoch`. For computational reasons, the update must be sparse: you should never iterate over all words or features, but only iterate over cached ones found in `self.x2ys` and `self.fcache`.

6. If you implement the gradient update correctly you should be able to reduce the training loss relatively fast (for 10% of data). Try `lr` $= \{0.1, 0.5, 1, 2, 4, 8\}$ with `basic_features1` and report the best validation perplexity in 10 epochs. It should not be much larger than 150.
7. For your best model, report top-10 feature types that have been assigned the highest weight. What do they mean?
8. For your best model, try 10 random seeds and report the mean and standard deviation of the validation perplexities.
9. (Bonus): train your model using 100% of the training and validation data with your choice of features (you may design different feature extractors, such as one including the suffixes and prefixes up to length 3) and full hyperparameter tuning over the learning rate and seed. Report the best validation perplexity. You must provide a command that we can simply type and reproduce your result. Top 3 students will get 10 extra points on this assignment.

Problem 2: Feedforward Neural Language Model

(2 + 2 + 2 + 2 + 2 + 2 = 12 points)

You will experiment with a simple feedforward neural language model. The file `assignment2_nlm.py` contains partially complete code. Unfortunately, training large neural models is infeasible without GPUs. Hence the goal of this problem is not to obtain state-of-the-art perplexity but to become familiar with fitting a neural model using PyTorch. To this end, we will **train and test on the same data**, 10% of the validation data from A1, with vocab size 1,000, and explore the model's capacity to fit training data. This toy configuration is set as default arguments. You will only need to play with `wdim` (word embedding dimension), `hdim` (hidden state dimension), `nlayers` (number of layers), `lr` (learning rate), and possibly `epochs` (number of training epochs), `B` (batch size), and `seed` (seed).

Most of the code is provided for you. Take a look at class `FFLM`. It has a word embedding dictionary `self.E` and a feedforward layer `self.FF`. Examine class `FF` to see what options you can control in the layer. Given previous words $x = (x_1, \ldots, x_n)$ (already converted to indices) the model defines a distribution over the next word as

$$p_{Y|X}(y|x) = \mathrm{softmax}_y\left(\mathrm{FF}\left(\begin{bmatrix} E_{x_1} \\ \vdots \\ E_{x_n} \end{bmatrix}\right)\right)$$

1. Define `self.FF` with correct specifications. Only specify the input dimension, hidden state dimension, output dimension, and number of layers.
2. Implement (batch-version) `forward` in `FFLM`. You'll want to use `view` to get the correct shape. The logits just mean values to be passed to softmax (or sigmoid), which can be directly used for predefined losses like CrossEntropyLoss.
3. Explore the impact of the number of parameters in a linear model as follows. Set `nlayers` $= 0$. Fix all default arguments except `wdim`, `hdim`, and `lr`. Thus you will train a linear layer using 3 previous words with batch size 16 up to 10 epochs. Vary `wdim` $=$ `hdim` (tie for simplicity) across $\{1, 5, 10, 100, 200\}$ and for each choice optimize over `lr` $= 0.00001, 0.00003, 0.0001, 0.0003, 0.001$. Report the optimized perplexity (y-axis) at each dimension (x-axis).
4. Repeat the previous exercise with `nlayers = 1`. Thus you will train a feedfoward network with 1 ReLU hidden layer in otherwise the same setting.
5. You will see that adding nonlinearity doesn't automatically make training loss smaller given the same number of epochs. One hypothesis is that it takes more updates for bigger models to converge, but when they do they can achieve a smaller training loss. Verify or refute this hypothesis by running the linear and nonlinear models to convergence (e.g., set `epochs` $= 1000$) using `wdim` $=$ `hdim` $= 30$ and your optimized learning rates from the previous exercises.
6. The code also prints nearest neighbors of trained word embeddings (in cosine similarity). Explore these neighbors. Do they always/sometimes/never make sense? What do you think is the reason?