TTIC 31210: Advanced Natural Language Processing
Assignment 1: Language Modeling (70 points)

Instructor: Kevin Gimpel
Assigned: Monday, April 1, 2019
Due: 7:00 pm, Tuesday, April 16, 2019
Submission: email to `kgimpel@ttic.edu`

Submission Instructions

Package your report and code in a single zip file or tarball, name the file with your last name followed by "_hw1", and email the file to `kgimpel@ttic.edu` by the due date and time.

Collaboration Policy

You are welcome to discuss assignments with others in the course, but solutions and code must be written individually. You may modify code you find online, but you must provide attribution and be sure that you understand it!

Overview

In this assignment, you will experiment with various loss functions for training LSTM language models. You will also compare the amount of context used for making word predictions both quantitatively and qualitatively.

Data

You will use a simplified story dataset that I prepared from the ROC story corpus (Mostafazadeh et al., 2016). I converted most names to Bob and Sue and extracted a subset of sentences that only use words from a small vocabulary. The zip file posted on the course webpage contains the following files:

- `bobsue.lm.train.txt`: language modeling training data (LMTRAIN)
- `bobsue.lm.dev.txt`: language modeling development data (LMDEV)
- `bobsue.lm.test.txt`: language modeling test data (LMTEST)
- `bobsue.prevsent.train.tsv`: previous-sentence training data (PREVSENTTRAIN)
- `bobsue.prevsent.dev.tsv`: previous-sentence development data (PREVSENTDEV)
- `bobsue.prevsent.test.tsv`: previous-sentence test data (PREVSENTTEST)
- `bobsue.voc.txt`: file containing vocabulary, with one word type per line

Each line in the `lm` files contains a sentence in a story. Each line in the `prevsent` files contains a sentence in a story followed by a tab followed by the next sentence in the story. Note: the second field in each line of each `prevsent` file is identical to the corresponding line in the corresponding `lm` file. (That is, `cut -f 2 bobsue.prevsent.x.tsv` is the same as `bobsue.lm.x.txt`.) The complete vocabulary is contained in the file `bobsue.voc.txt`, one word per line. You do not have to worry about unknown words in this assignment.

Evaluation

You will train language models in this assignment. However, to evaluate, you should use **word prediction accuracy** as your primary evaluation metric rather than perplexity. We are doing this because perplexity is not helpful when you are trying to compare certain loss functions.

To be more specific, you should evaluate your model's ability to predict each token beyond the start-of-sentence symbol ($\langle$s$\rangle$) in each line of the LMDEV (or LMTEST) file. In the LMDEV file, there are 7957 predictions to make. In the LMTEST file, there are 8059 predictions to make. You can check your numbers of predictions against these to ensure you are computing these accuracies correctly. For each prediction, you should use the previous ground truth words in the sentence. You should not test your ability to predict the initial $\langle$s$\rangle$ at the start of each sentence, but you definitely should test your ability to predict the $\langle$/s$\rangle$ symbol at the end of each sentence!

When you have a softmax layer, the predicted word is the most-probable word in the softmax. When you are using the other losses, the predicted word is the one with the highest score (we will define the score function below in Section 3).

Use LMDEV for early stopping. That is, when you report results, report the LMTEST accuracy for the model that achieves the best accuracy on LMDEV. You should also report the best accuracy achieved on LMDEV. In order to do early stopping and also to answer some of the questions below, you should compute the loss and word prediction accuracy on LMDEV periodically during training. Do this at least once per epoch, and preferably 2 or more times per epoch. We will discuss the PREVSENT files below in Section 5.

Below we will abbreviate this evaluation procedure as EVALLM. To summarize, EVALLM consists of training on LMTRAIN, using LMDEV for early stopping (using word prediction accuracy as the early stopping criterion), and reporting the best word prediction accuracy on LMDEV and the word prediction accuracy on LMTEST using the model that did best on LMDEV.

1. LSTM Language Models Trained with Log Loss (20 points)

First, implement a standard LSTM language model. That is, use an affine transformation on the hidden vector at each time step followed by a softmax to predict the word at that step. Train by minimizing log loss (cross entropy). Evaluate using the EVALLM protocol. Report your results and submit your code. You should be able to reach accuracies of at least 30%.

Notes:

- For these experiments, train for at least 10 epochs.
- If you like, you can experiment with different dimensionalities and architectural variations. But by default, you can just use a single-layer LSTM with dimensionality 200 for both the word embeddings and the LSTM cell/hidden vectors.
- Typically when using a softmax layer/classifier, an affine transform is used, including a bias vector, followed by the softmax function which exponentiates and normalizes the scores to produce a probability distribution over outputs. In my implementation, I did not use a bias vector in order to make the comparison fairer to models trained with other losses that similarly do not use bias parameters. (However, I did not find significant differences in accuracy when using or omitting bias parameters.)

- Randomly initialize the word embeddings and learn them along with the other model parameters.
- Since you will be using word embeddings for the inputs to the LSTM, and an affine transformation / softmax layer on the hidden vector for making predictions, the "input" and "output" word embeddings will be distinct. That is, they are not tied.
- For optimization, use Adam, stochastic gradient descent, or any other optimizer you wish. For my experiments, I used Adam with default hyperparameters. Document in your write-up what you use along with the optimizer hyperparameter values.
- You are encouraged to use a deep learning toolkit for this assignment (though you are welcome to implement everything from scratch if you prefer). To simplify things, you don't need to use mini-batching, though of course you can if you like. With my DyNet implementation, running each experiment took a few minutes per epoch on my laptop without mini-batching.

2. Error Analysis (15 points)

2.1. (5 points) Let's define an **error** as a pair $\langle y_g, y_p \rangle$ where y_g is the ground truth word, y_p is the model's prediction, and $y_g \neq y_p$. Implement the capability to print the word prediction errors made by your model. Submit your code.

2.2. (5 points) List the top 35 most frequent errors made by your model on LMTEST, including their counts.[1]

2.3. (5 points) Inspect the common errors to identify common error categories. You do not have to categorize every error, but you should spend some time looking at the errors to determine if there are groups of related errors. To help yourself identify error categories, think about the following as you look through the errors: Why do you think the model might have predicted what it predicted? How could the model have done better? Was the model "close"? If so, in what way was the model close? After identifying error categories, label each of the top 35 most frequent error types with one or more of your categories. (Note: I found about 6 or 7 error categories among the top 35 most frequent error types, including one "catch-all" category for those that were difficult to categorize.) Discuss what the model does well and what it does poorly.

3. Binary Log Loss Implementation and Experimentation (20 points)

Now you'll experiment with a different loss function for training your word prediction model: binary log loss with negative sampling.

Before describing the loss, let's introduce some notation. We use $\boldsymbol{y}$ to refer to a sentence and we use y_t to refer to the word at position t in $\boldsymbol{y}$. We assume that $y_1 = \langle \mathrm{s} \rangle$, the special start-of-sentence token, and that $y_n = \langle / \mathrm{s} \rangle$, the end-of-sentence token where $n = |\boldsymbol{y}|$ is the length of $\boldsymbol{y}$. When feeding word y to the LSTM, we define the input embedding function that returns the embedding of y by $emb_i(y)$. We'll use $\mathbf{h}_t$ to denote the LSTM hidden vector at position t (which you used in Section 1 above to predict y_t using a softmax layer). Given the above notation, we will define the loss as follows:

$$\min_{\boldsymbol{\theta}} \sum_{\boldsymbol{y} \in D} \sum_{t=2}^{|\boldsymbol{y}|} \left(-\log \sigma \left(\text{score}(y_t, \mathbf{h}_t) \right) - \frac{1}{|NEG|} \sum_{y' \in NEG} \log \left(1 - \sigma \left(\text{score}(y', \mathbf{h}_t) \right) \right) \right) \tag{1}$$

[1] Ordinarily it's not ideal to use test sets for error analysis because it shows us too much about the test set, thereby contaminating it, but it's ok in this case because (1) this is not a standard test set used by others, and (2) for self-supervised tasks like language modeling we can always create more test sets.

where $\boldsymbol{\theta}$ contains all model parameters, D is the training set of sentences, σ is the logistic sigmoid function ($\sigma(x) = \frac{1}{1+e^{-x}}$), NEG is a set of **negative samples**, and $\text{score}(y, \mathbf{h})$ outputs a scalar score under the model for outputting word y from hidden vector $\mathbf{h}$. Note that t begins at 2 because $y_1 = \langle \text{s} \rangle$ always, so it doesn't need to be predicted. For the form of the score function, use the following:

$$\text{score}(y, \mathbf{h}) = emb_o(y)^\top \mathbf{h} \tag{2}$$

where $emb_o(y)$ is the "output" embedding function, which should be distinct from the "input" word embeddings emb_i used as input to the LSTM. (Note: the score function defined above does not include the "bias" weight for word y. Biases are typically included in softmax layers used for classification. I didn't find the inclusion of a bias to make a significant difference in the results, so I omitted it in all of my experiments for this assignment. You can choose to use or omit bias parameters in your experiments; just be sure to report what you do.)

To form NEG, you will experiment with several methods. First, simply generate samples from a uniform distribution over word types (entries in the vocabulary), including $\langle / \text{s} \rangle$. We will refer to this scheme as UNIF. Mikolov et al. (2013) proposed other ways of choosing negative samples. They used the unigram distribution (UNIG) as well as "flattened" versions of it with flattening of varying degree. Where the unigram probability of a word y is denoted $P(y)$, the f-flattened unigram probability $P_f(y)$ is:

$$P_f(y) \propto P(y)^f \tag{3}$$

That is, raise all unigram probabilities to f and renormalize to ensure that P_f is a probability distribution. We will refer to this as UNIG-f. Note that UNIG-f corresponds to UNIG when $f = 1$ and it corresponds to UNIF when $f = 0$.

3.1. (5 points) Implement training for this loss and submit your code.

3.2. (5 points) Run EVALLM using the above loss. To form NEG, sample r words uniformly at random from the vocabulary (UNIF). Experiment with three choices of r: $r = 20$, $r = 100$, and $r = 500$. For all experiments with binary log loss, train for at least 20 epochs.

3.3. (5 points) Now, with $r = 20$, form NEG by sampling from UNIG-f with various values of f. Experiment with f values between 0 and 1. Can you find an f value that improves test accuracy over UNIF? As in 3.2, train for at least 20 epochs.

3.4. (5 points) One of the primary motivations for binary log loss with negative sampling is training time. But there are several ways to measure this, such as sentences processed per second ("#sents/sec"), number of sentences processed in order to reach maximum LMDEV accuracy ("#sents for max acc"), and wall-clock time needed to reach maximum LMDEV accuracy ("time for max acc"). Discuss these three efficiency measures for log loss and binary log loss with negative sampling. Under which efficiency measure(s) is one better than the other?

4. Using a Larger Context (15 points)

You will now train models that use additional context in the form of the previous sentence in the story. The PREVSENTTRAIN, PREVSENTDEV, and PREVSENTTEST files contain two consecutive sentences in a story in a tab-separated format. The words you need to predict are those in the second sentence in each line in the PREVSENTDEV and PREVSENTTEST files; those second sentences are identical to the lines in the LMDEV and LMTEST files, so this will let you see the additional contribution to accuracy from

having the previous sentence.

Train an LSTM language model with log loss as in Section 1. Do not compute a loss for the words in the previous sentence. Simply input the previous sentence to your LSTM, one word at a time, including its sentence start and end symbols. This will produce hidden vectors for each word, but you should not use them to make any predictions. Then, input the sentence start symbol (y_1) of the second sentence to the LSTM and compute the log loss of predicting (y_2). Proceed as in Section 1 until the end of the second sentence, computing log losses for each prediction in the second sentence.

When you run experiments in this section, do the following: Train on PREVSENTTRAIN using log loss, use PREVSENTDEV for early stopping, and report your word prediction accuracy on PREVSENTTEST. Below we will abbreviate this as EVALPREV.

4.1. (6 points) Implement the procedure described above and submit your code. Run EVALPREV and report the test accuracy. Hopefully it's higher than what you found in Section 1!

4.2. (3 points) List the 35 most frequent errors on PREVSENTTEST using the model from 4.1.

4.3. (6 points) Categorize the errors like you did in Section 2. Compare the most frequent errors to those you observed in Section 2 for the model without the previous sentence. What sorts of errors are being corrected when given larger context? What sorts of errors still remain?

5. Extra Credit: Hinge Loss Implementation and Experimentation (5 points)

For extra credit, implement and experiment with the following contrastive hinge loss:

$$\min_{\boldsymbol{\theta}} \sum_{\boldsymbol{y} \in D} \sum_{t=2}^{|\boldsymbol{y}|} \sum_{y' \in NEG} \max(0, 1 - \text{score}(y_t, \mathbf{h}_t) + \text{score}(y', \mathbf{h}_t)) \tag{4}$$

where the notation is defined as in Section 3.

Implement training for this loss (submit your code) and run EVALLM using this loss. To form NEG, sample r words uniformly at random from the vocabulary (UNIF). Experiment with three choices: $r = 20$, $r = 100$, and $r = 500$. Then, with a single value for r, form NEG by sampling from UNIG-f with various values of f. Experiment with f values between 0 and 1. Can you find an f value that improves test accuracy over UNIF? In all cases, train for at least 20 epochs.

References

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc., 2013.

Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. A corpus and cloze evaluation for deeper understanding of commonsense stories. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 839–849, San Diego, California, June 2016. Association for Computational Linguistics.