DS-GA 1011 Fall 2019
N-Gram and Neural Language Modeling

In this assignment, you will build and analyze language models using two approaches: n-gram language modeling and neural language modeling with a recurrent neural network.

1 N-Gram Language Modeling

In this section you will improve the n-gram language model that we saw in the lab. In particular, you will implement a smoothing technique called **interpolation**.

1.1 Setup

We provide code in the accompanying notebook (`lm_homework.ipynb`) for the following:

- Data loading (`load_wikitext`)
- Additive smoothing (`NGramAdditive`)
- Evaluating perplexity (`perplexity()`)

This code is very similar to the code from the lab. You will implement a new class (`NGramInterpolation`) based on the description below.

1.2 Interpolation

Recall that n-gram language models use **smoothing** techniques to allocate probability to n-grams that do not occur in the training corpus.

For instance, in the lab we estimated a 3-gram language model and found that the 3-gram `(my, pet, lion)` did not occur in the training corpus, and hence received zero probability under the language model. To fix this, in the lab we used *additive smoothing*.

Several alternatives[1] use the idea that although an n-gram may not appear in the corpus (e.g. `(my, pet, lion)`), lower-order $(n-k)$-grams contained within (e.g. `(pet, lion)`, `(lion)`, `...`) may appear.

A resulting technique, called ***interpolation***, involves estimating an n-gram model *as well as* separate $(n-1)$-gram, $(n-2)$-gram, etc. models. You will implement a simple variant called **linear interpolation**:

$$p_n^{\text{interp}}(w_t|w_{t-n+1}^{t-1}) = \lambda_n p_n(w_t|w_{t-n+1}^{t-1}) + \lambda_{n-1} p_{n-1}(w_t|w_{t-n+2}^{t-1}) + \ldots + \lambda_1 p_1(w_t) + \lambda_0 p_0, \quad (1)$$

$$\sum_{j=0}^{n} \lambda_j = 1, \quad (2)$$

$$\lambda_j \geq 0 \text{ for all } j. \quad (3)$$

where p_n is a n-gram model estimated with maximum likelihood (the counting approach we used in the lab), and $p^0 = \frac{1}{|\text{vocab}|}$ ensures that sequences with unseen words receive non-zero probability.

[1] See Jurafsky & Manning 3.4 for additional background `https://web.stanford.edu/~jurafsky/slp3/3.pdf`.

1.2.0.1 λ parameters For simplicity, we will use a single λ parameter per n-gram order as shown above.

With interpolation, a lower order model is a special case of a higher order model. For instance, a 2-gram interpolation model is a 3-gram interpolation model with $\lambda_3 = 0$. Thus for a given n-gram order, there is a setting of $(\lambda_0, \ldots, \lambda_n)$ that performs *at least as well as* the best lower order model.

Your task is to implement n-gram linear interpolation models for $n = \{2, 3, 4\}$, and find settings of $(\lambda_0, \ldots, \lambda_n)$ such that:

1. All three interpolation models outperform the additive smoothing model.
2. Performance improves as a function of n.

The additive smoothing model is given in `lm_homework.ipynb` (`NGramAdditive`), and performance is measured by validation perplexity. For λ parameters, it should be sufficient use manual selection.[2]

Please report the results in `lm_homework.ipynb`, including the λ values you chose for each model.

1.3 Extra Credit: Kneser-Ney Smoothing

The task above showed that interpolation can improve n-gram language models, but choosing good λ parameters is crucial. An alternative method that works well in practice is **Kneser-Ney Smoothing**. You can read more about Kneser-Ney smoothing in section 5.3.1 of the course lecture notes, (Chen and Goodman 1996), or Jurafsky & Manning 3.5, and implement this method for extra credit.

2 Neural Language Modeling with a Recurrent Neural Network

2.1 LSTM and Hyper-parameters

Your first task is to train a neural recurrent language model on **Wikitext-2**. This task is similar to what was covered in the lab. However, instead of a vanilla RNN, you will use an **LSTM** (see `nn.LSTM`[3]).

1. Train a LSTM language model and evaluate its training and validation perplexity. Your goal is to train a model which performs at least as well as the model from the lab in terms of validation perplexity; to measure this baseline, you should train a RNN (`nn.RNN`) using similar hyper-parameters as those used in the lab. Report the results in `lm_homework.ipynb`.
2. There are several hyper-parameters which can affect performance, such as the number of parameters, amount of regularization, etc. Find and choose two of these, and for each one **produce a plot** showing the effect of varying the hyper-parameter. Two options are:
   - A plot with the hyper-parameter varying on the x-axis, and the best validation perplexity achieved during training on the y-axis.
   - A two-part plot showing the training and validation losses over the course of training, for varying hyper-parameter settings.

   Include a brief (1-2 sentence) discussion of the results. Report the results in `lm_homework.ipynb`.

2.2 Learned Embeddings

Next you will analyze the word embeddings that the model learns.

[2] Make sure that $\sum_{j=0}^{n} \lambda_j = 1$.

[3] For additional background see 4.2.3 of the lecture note (`https://github.com/nyu-dl/NLP_DL_Lecture_Note/blob/master/lecture_note.pdf`).

1. Choose a set of 5 words, e.g. {`the, run, dog, where, quick`}, and for each word in the set, find the 10 'closest' words and 10 'furthest' words according to cosine similarity of vectors from the trained LSTM model's `nn.Embedding` layer.[4]
2. Use **UMAP**[5] to visualize the model's `nn.Embedding` layer. In particular, run UMAP on the entire `nn.Embedding` weight matrix, and plot the points corresponding to the 100 words selected in part (1).[6]. We provide code in `lm_homework.ipynb` and the lab which demonstrates UMAP.
3. Repeat steps (1) and (2) with the model's projection layer (the `nn.Linear` weight matrix of size $|\text{vocab}| \times d$).

   Show the words you selected, the plots, and briefly discuss the results, for instance: Are the 'closest' and 'furthest' words interpretable? Where are these words in the plots you produced? Are there interpretable patterns in the plots? Do the embedding visualizations substantially differ from the projection layer visualizations? Negative results are also good to discuss.

2.3 Scoring

One use of a language model is to *score* sequences. **Your task** is to:

1. Implement a function which returns the log probability ('score') of a sequence according to your model.
2. Report the 10 sequences from the validation set with the highest scores, and 10 sequences with the lowest scores, according to the best LSTM model that you trained above.
3. Modify one of the high-scoring sequences from above by substituting a few (e.g. 1-3) of its tokens with other tokens from the vocabulary. Find a modified sequence with a lower score and a modified sequence with a higher score, and explain why you think the score changed in both cases.

2.4 Sampling

In the lab, we saw how to sample sequences from n-gram language models. In this section, you will implement **sampling** for your recurrent language model. Here is pseudo-code of this process:

$h_0 \leftarrow 0$;
$x_0 \leftarrow$ `<bos>`;
while x *is not* `<eos>` **do**
  $(p, h_{t+1}) = RNN(h_t, x_t)$;
  $x_{t+1} \sim \text{multinomial}(p)$;
Result: sampled sequence $(x_1, x_2, \ldots, x_T)$

Where p is a $|\text{vocab}|$-dimensional vector giving conditional probabilities of each possible next-token. The typical way of obtaining these probabilities is by projecting the RNN output and taking the `softmax()`. For multinomial(p) see `https://pytorch.org/docs/stable/torch.html#torch.multinomial`.

Your task is to:

1. Implement sampling.
2. Sample 1,000 sequences from the best LSTM model you trained above.
3. Compare the sampled sequences against 1000 sequences from the validation set in terms of **number of unique tokens** and **sequence length**.
4. Choose 3 sampled sequences and discuss their properties, for instance: Can you tell that these 3 sequences are model-generated rather than human-generated? Do these samples stay on topic? Are they grammatically correct?

[4] See `https://pytorch.org/docs/master/nn.functional.html?highlight=cosinetorch.nn.functional.cosine_similarity`.
[5] `https://umap-learn.readthedocs.io/en/latest/`
[6] There may be fewer than 100 words if there is overlap between the 'closest' and 'furthest' words, which is fine.

Bibliography

Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics. doi: 10.3115/981863.981904. URL https://doi.org/10.3115/981863.981904.