CSE P 517: Homework 5
Neural Text Generation

Due Sun March 14 2021 11:59pm

Submission instructions As usual, submit 2 files on Canvas — **Code** (`HW5.zip`) and **Report** (`HW5.pdf`). The usual submission guidelines apply.

Logistics While this assignment involves large neural architectures, you will not need to train or fine-tune them, and evaluation should run quickly on `Azure` or most CPUs.

1 Neural Text Generation with Transformers

In HW4, you got a taste of generating text with a large neural network (GPT-2). Using models like GPT-2 [7] to generate text is one of the hottest areas in NLP today.

```
HUMAN WRITTEN PROMPT:
In a shocking finding, scientist discovered a herd of unicorns living in a remote,
previously unexplored valley, in the Andes Mountains.  Even more surprising to there
searchers was the fact that the unicorns spoke perfect English.
MACHINE CONTINUES:
The scientist named the population, after their distinctive horn, Ovid's Unicorn.
These four-horned, silver-white unicorns were previously unknown to science.  Now,
after almost two centuries, the mystery of what sparked this odd phenomenon is finally
solved.  Dr.  Jorge Perez, an evolutionary biologist from the University of La Paz,
and several companions, were exploring the Andes Mountains when they found a small
valley, with no other animals or humans.  Perez noticed that the valley had what appeared
to be a natural fountain, surrounded by two peaks of rock and silver snow
```

Before models like this, RNNs were the standard. In HW4 we found RNNs often wrote sentences that aren't even coherent. Generating a creative and contentful text using NLP models was (and still is) impressive and exciting. You can try https://talktotransformer.com/ for additional examples.

But how do models like GPT-2[7] or GPT-3[2] actually *generate text*? To understand this, we should think about these models as a blackbox function P which takes text as input (e.g. the HUMAN WRITTEN PROMPT from above) and outputs probabilities for what words[1] might come next. We can consider how a model might have written the example above. Let's start with generating the *first word*:

[1]To be precise, what *tokens* might come next. What is the difference between a word and a token? Many words (e.g. "The") are tokens, but some longer words are broken into easier-to-predict pieces. E.g. "scientist" → ["scient", "ist"]; these are tokens. We don't need UNK as in HW1, because we define a set of tokens that can be combined into any text we'll see! We will be using tokens from byte pair encoding (link) specifically. This won't affect your code but *it is interesting*.

$$P(PROMPT) = \begin{bmatrix} 0.5 \\ 0.05 \\ 0.03 \\ 0.08 \\ 0.000003 \\ 0.11 \\ ... \end{bmatrix}$$

Where each entry in this vector (we'll call it $\mathbf{p_{prompt}}$) is the probability of some word coming next: $\mathbf{p_{prompt}[i]}$ is the probability of word t_i. We can imagine in this example that t_0 is "The". $\mathbf{p_{prompt}[0] = 0.5}$ so the model gives a 50% chance that the next word is "The" (very high probability–and indeed, this is the word the model chooses).

There are a couple of very simple options for choosing what word comes next: either, take the *highest probability word*. This must always be the best word because the model gave it the highest probability, right? (SPOILER: **no**). Another option is just *sampling from* $\mathbf{p}$. In the example above, this would mean 50% chance of picking t_0 ("The"), 5% chance of picking t_1, 3% change of t_2 and so on. This tends to be a better option, but in practice we often do something in between. Below, we describe multiple common solutions for decoding (generating text).

Greedy Decoding Mentioned above, this is simply taking the highest probability word t from the distribution $P(input) = \mathbf{p}$:

$$t = \text{argmax}_i \ \mathbf{p[i]} \tag{1}$$

In the example, this would mean taking "The" to be the next word because it has the highest probability. In this case, it seems like a good decision. However, greedy decoding often gives bad generations. Choosing an *interesting* next word is often not very high probability, and the highest probability option is often something bland or generic.

Pure Sampling This is sampling a word based on the distribution defined by our model, $\mathbf{p}$. This is the most "in agreement" with what the model is telling us. As we discussed above, there's a 50% "The" is picked in this case, which seems reasonable. However, the model often gives a bit too much likelihood to low probability things, which can result in strange and nonsensical generations when using pure sampling.

top$_k$ Sampling We often use something between greedy decoding and pure sampling, not just taking the highest probability word but completely avoiding (truncating) some lower-probability options. We don't want to fall into the trap of greedy decoding, where the most probable word is often bland and uninformative, but on the other hand pure sampling often gives too much probability to *very strange* generations. A common solution is to **truncate** the distribution, where we set lower probabilities to 0 and renormalize. Intuitively, if we're giving a very low probability to some word t, it might as well be zero.[2]

In top$_k$ sampling, we truncate (set probability to 0) any words not in the top-k (by probability):

$$p'[i] = \begin{cases} p[i] / \sum_{i \text{ s.t. } t_i \in V_k} p[i] & \text{if } t_i \in V_k \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Where V_k is the set of the top-k highest probability words. To be precise, we set the probability of any words not in V_k to 0, then renormalize $\mathbf{p}$, so that it still sums to 1 (valid probability).

top$_k$ sampling was quite popular in the past, but has become much less common with the invention of top$_p$ (or *nucleus*) sampling.

[2]If there are many very low probability words, then it unintuitively becomes very likely that we sample a low-probability word–even though they are individually unlikely, their combined probability is high.

top$_p$ Sampling top$_p$ sampling (which comes from Professor Choi's group! [3]) is one of the most popular ways to generate text. Much like top$_k$ sampling, we **truncate** low probability words. However, instead of keeping the k best words to have nonzero probability, top$_p$ keeps enough of the top words to cover at least probability p. Mathematically, this can be defined as taking the smallest set of words V_p such that:

$$\sum_{i \text{ s.t. } t_i \in V_p} \mathbf{p}[i] \geq p$$

In practice, this means we add words t_i to V_p, starting with the highest probability word and going down, until the total probability of words in V_p is at least p.

After we've defined this set of words that will have non-zero probability, top$_p$ sampling works just like top$_k$:

$$p'[i] = \begin{cases} p[i] / \sum_{i \text{ s.t. } t_i \in V_p} p[i] & \text{if } t_i \in V_p \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

top$_p$ sampling was found to work a lot better than past approaches (read the paper!) But this still depends on the application. We will explore this in Q2.

1.1 Warmup: Written Answer

To get familiar with different decoding techniques, we will begin by applying them by hand. Consider 2 different probability vectors (i.e. of the same form as $P(PROMPT)$ above). REMEMBER: you should renormalize probability after setting some values to 0.

$$\mathbf{p}_1 = \begin{bmatrix} 0.25 \\ 0.34 \\ 0.32 \\ 0.09 \end{bmatrix} \quad \mathbf{p}_2 = \begin{bmatrix} 0.20 \\ 0.04 \\ 0.53 \\ 0.11 \\ 0.02 \\ 0.10 \end{bmatrix}$$

1. Apply top$_k$ truncation with $k = 1$.
$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{p}_2 = ?$$
2. top$_k$ truncation with $k = 3$.
$$\mathbf{p}_1 = ? \quad \mathbf{p}_2 = ?$$
3. top$_p$ truncation with $p = 0.35$.
$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 0.515 \\ 0.485 \\ 0 \end{bmatrix} \quad \mathbf{p}_2 = ?$$
4. top$_p$ truncation with $p = 0.8$.
$$\mathbf{p}_1 = ? \quad \mathbf{p}_2 = ?$$
5. What setting of k will give us greedy decoding? pure sampling? (hint: this may depend on the model or $\mathbf{p}$)
6. What setting of p will give us greedy decoding? (hint: this will depend on $\mathbf{p}$) pure sampling?

1.2 Implementing Decoding Strategies

In this section, you will be implementing the generation strategies from above.

setup For this section, you will follow the same setup as for HW4, specifically using the `hw4` conda environment (feel free to rename this to `hw5` if you'd like). No further setup is required besides this, all code you will need is included in hw5.zip. Aren't you glad we made you set this up early?

1.2.1 Greedy Decoding

Navigate to Q1.2.1 in `generation_utils.py` and implement greedy decoding. Follow the hints that are included there.

1.2.2 top$_k$ Sampling

Navigate to Q1.2.2 in `generation_utils.py` and implement top$_k$ sampling. Follow the hints that are included there. This is a bit more complicated than greedy decoding, so you can test your implementation by running `python test_top_k.py`. If it runs without errors, then your top$_k$ truncation is correctly implemented (NOTE: you still need to sample correctly after truncation. This is not tested).

1.2.3 top$_p$ Sampling

Navigate to Q1.2.3 in `generation_utils.py` and implement top$_p$ sampling. Follow the hints that are included there. To test your implementation, run `python test_top_p.py`. If it runs without errors, then the truncation step of top$_p$ is correctly implemented! (NOTE: you still need to sample correctly after truncation. This is not tested)

1.3 Delivarables

- (5pts) Answers to Q1.1, 0.5 pts for each answer.
- (5pts) Implementations for Q1.2: (1pt) for greedy and (2pts) each for top$_k$ and top$_p$.

2 Generating With Neural Models

In this section, we will try generating using our implementations from Q1! We will start with something familiar, the summarization problem from HW4. Here, we give you a trained model and you can try different versions of decoding and see what works best!

Next, we try something completely different: writing stories. With a *different* model trained to write stories, you will be generating the last sentence for short stories.

Overall, you will be answering 2 questions here. First, which generation strategy gets the best *automatic* scores (i.e. BLEU and METEOR) on summarization and story writing? Second, do you agree with the automatic metrics?

2.1 Summarization

We will be considering a similar summarization task to HW4[5], although with a different test set. Unlike HW4, we provide you with a **trained** model, and instead of training the model as in HW4, you will be trying out different decoding (generation) strategies from Q1 with different parameters, to see which works the best.

You should try greedy decoding, pure sampling, top$_k$ and top$_p$. For top$_k$ and top$_p$ you should try 2 - 4 values for the decoding parameter. You can choose these yourself, or use the following (standard) values: $k = 4, 16, 64$ for top$_k$, $p = 0.3, 0.5, 0.8$ for top$_p$. Feel free to use these values or any others you choose!

To generate summaries, use the following command:

```
python generate_summaries.py -version <version> -top_p <p> -top_k <k>
```

-version should be one of [greedy, sample, top_k, top_p]. You only need to specify -top_k or -top_p (truncation parameters) if using these versions.

Once you have generated with various settings, run `python evaluate_summaries.py` to get BLEU [6] and METEOR [1] scores for each strategy. As discussed in HW4, these indicate how close the machine generation is to what a human wrote (the higher, the better). Report these scores in your write-up.

Finally, compare the summaries written by a few different generation strategies (you can see input documents in `data/XSUM_test_inp.txt` and generations in `generations_XSUM`). Can you tell the difference in terms of quality? Do you agree with what the automatic scores tell us about which one is best? Any answer is reasonable, as long as you support it with examples.

2.2 Story Writing

You will be doing essentially the same thing as above, but for story writing. We will consider the ROC stories dataset [4], which contains simple 5-sentence stories. We have trained a model for you that takes the first 4 sentences of such stories, and tries to finish them (i.e. write the 5th sentence). You will again be using the different generation strategies from Q1. The best strategies **might not** be the same between both tasks, depending on the values you pick.

You should try greedy decoding, pure sampling, top_k and top_p. For top_k and top_p you should try 2 - 4 values for the decoding parameter. You can choose these yourself, or use the following (standard) values: $k = 4, 16, 64$ for top_k, $p = 0.3, 0.5, 0.8$ for top_p. Feel free to use these values or any others you choose!

To generate stories, use the following command:

```
python generate_stories.py -version <version> -top_p <top_p> -top_k <top_k>
```

-version should be one of [greedy, sample, top_k, top_p]. You only need to specify -top_k or -top_p (truncation parameters) if using these versions.

Once you run all of these generations, run `python evaluate_stories.py` to get BLEU [6] and METEOR [1] scores for each strategy. As discussed in HW4, these indicate how close the machine generation is to what a human wrote (the higher, the better). Report these scores in your write-up.

Finally, compare the story endings written by a few different strategies/settings for generation (you can see input stories in `data/ROC_test_inp.txt` and generations in `generations_ROC`). Can you tell the difference in quality? Do you agree with what the automatic scores tell us about which one is best? Any answer is reasonable, as long as you support it with examples.

2.3 Short Answer

The metrics we use here for scoring (BLEU and METEOR) are based on how close the machine generation is to what a human wrote (the *ground truth*). Specifically, how many words, sub-strings, and synonyms are shared between the machine generation and ground truth.

In summarization, the ground truth is a human written summary for the same news article, and in story writing this is the ending a human wrote for the same story. We tend to find lower values of METEOR and BLEU for story-writing tasks, indicating less agreement with the human-written text. Why might this be? Does this mean BLEU and METEOR are less suited for scoring machine-written stories than summaries?

2.4 Delivarables

- (2pts) Report the scores for your generations in both tasks.

- (2pts) Provide discussion mentioned above for both tasks.
- (1pt) short answer for Q2.3

References

[1] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[3] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *ArXiv*, abs/1904.09751, 2019.

[4] 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, 2016.

[5] Shashi Narayan, Shay B Cohen, and Mirella Lapata. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. *arXiv preprint arXiv:1808.08745*, 2018.

[6] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[7] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.