GA1
Lexically Constrained Beam Search for Machine Translation

Anupreet Singh
UMBC, Campus ID- UK43298
email-anuprel1@umbc.edu
https://github.com/anupreetsingh/GA1

1. Algorithm Discussion

Beam Search
Beam search is a heuristic search algorithm used extensively for Natural Language Processing(NLP) tasks that involve sequence generation like Machine Translation, Summarization and Caption generation.

A common approach used before Beam Search that has now become much less attractive was *greedy search*. In greedy search for sequence generation tasks the algorithm would select the most probable output token at each time step. This equation from [1] provides a clearer understanding of the concept:

$$y'_t = \arg\max_{y_i \in \{v\}} p(y_i \mid x; \{y_0, \ldots, y_{t-1}\})$$

Where

y'_t is the most probable output token at time-step t conditioned on the input sequence(x) and the output tokens up to that point($\{y_0, \ldots . y_{t-1}\}$).

y_i is a token from the vocabulary of our Language Model.

So y'_t is chosen because it is appears to be the most likely output token at that time, as it may have the greatest logit score and hence the greatest probability value. But the problem with this approach is that we run the risk of ending up with a globally sub optimal output in pursuit of going for the locally optimal output.

Now, Beam search solves this problem by suggesting to keep track of k(i.e. Beam width) most likely Hypothesis at each time step. A hypothesis a potential output sequence of tokens generated by a language model at a given time step. Here is a description of how Beam search handles sequence generation:

- Initially, the model begins with an empty hypothesis, meaning it has no generated words yet for the output sentence.
- At the first step(Time step 1), the model generates a set of possible starting words, each forming a one-word hypothesis. Then beam search will select only the k highest-probability starting words and store each one as a distinct one word hypothesis.
- In the next step(Time step 2), for each of these one-word hypothesis, all possible two-word hypothesis are generated based on the model's vocabulary.Then, across all these expansions, only the top scoring k two-word hypotheses are selected and kept track of for further steps.
- This is continued until k complete candidate hypothesis are obtained and the one with the highest probability is given as output.

A question may arise, if Keeping track of k most probable hypothesis at each time step will result in a more optimal solution then is it good to have a bigger value of k? Well, no. This is because increasing k after a certain extent basically leads the approach closer to an exhaustive exploration and scoring of every possible sequence.

In beam search, although after each time step the hypothesis are pruned down to k but we still need to calculate:
Number of sequences scored at each time step=k*V, where V is the size of the vocabulary.
Total number of sequences scored for an output of length T=k*V*T

For an exhaustive approach,
Total Number of sequence scored=Total number of possible sequences=V^T

Scoring V^T is an unmanageable task for most real world scenarios. So Beam search with a reasonably small value of k gives the best possible solution and acts as good compromise between getting a global suboptimal solution using greedy search and taking on intractable undertaking using an exhaustive approach.

Lexically Constrained Beam Search
Lexically constrained beam search is an extension of the traditional beam search algorithm, specifically designed to include user defined lexical constraints in sequence generation output. For the purposes of Machine translation, implementing Lexical constraints typically involve ensuring that certain words or phrases appear in the translated sequence. These constraints could be Hard(must appear) and Soft(preferred but not mandatory). In our implementation we only apply hard constraints. There are multiple possible approaches to implementing lexical constraint in a standard beam search but I mostly referred back to the one discussed in Hokamp & Liu, 2016[1].

2. Implementation Discussion

Dataset:
The dataset chosen for observing the final results of machine translation of English-Russian pairs is WMT-16 [2]. In the beginning of the project development cycle the WMT-19 dataset [3] was considered as it is almost 30 times larger than WMT-16 so the model may have been a little better fine-tuned and result in generating better tokens at each step which would have led to overall Better translation results for the test/validation splits.

Model:
The pre-trained model selected for most optimal results is Google mT5 Base[4].

The two methods to generate an English to Russian translation using mT5-base are:
1. Use a prefix in the english sentence indicating the task(i.e. translated English to Russian) which turns out performs extremely poorly so the second method is the basically the only option to get meaningful results.
2. Fine tune the mT5-base Model to the WMT-16 dataset so that it implicitly learns what to do by updating its parameters.

After Tokenizing the train split of the WMT-16 dataset and then fine tuning the mT5-base model to it. The problem at this point arose that using this fine tuned mT5-base model for anything after this point in development would have actually needed a much better local system with a dedicated GPU for the computation to be time feasible. And it could not be done at google collab because of the issues discussed in the hurdles section below. So for all the processing and obtaining results beyond this point we use the Helsinki model [5] as it is already fine tuned for the task of English-Russian translation and would give us meaningful results to interpret our implementation of lexically constrained beam search given the time constraint of the semester.

Developing a basic Beam Search for Machine Translation:
The .generate function from the model was used get translation of the validation/test split of WMT-16 dataset. This is used as baseline for comparing it to our scratch_beam_search function. For the .generate function a lot of possible combinations of max_length, num_beams and early_stopping were tried, and the instance with max_length=128, num_beams=4 and early_stopping=True gave the best BLEU score for the translation. So we use the same instances for the scratch_beam_search function.

The scratch_beam_search function uses an implementation of beam search from scratch by choosing the top-k tokens with the highest score from the logits derived by the first forward pass of the model. It keeps track of these candidate 1 token hypotheses in k beams.Then In subsequent steps it passes the k candidate hypotheses in the models forward pass to get the top k best score from any of the possible expansion tokens for the k hypotheses. The top k scorers are then tracked as the new hypotheses in k beams. Since early_stopping is set True for our implementation, this process is continued till the beam-1(top scoring beam) for a sequence generates the EOS token, at which point the hypotheses in beam 1 is considered complete and returned as output of the translation of the input sequence.

BLEU Scores for the Translation of Eval and Test split of the data using .generate and scratch beam search are given below:

For Validation set:
1. **Using .generate with beam search:**
BLEU = 27.11 56.9/32.9/21.0/13.8 (BP = 1.000 ratio = 1.027 hyp_len = 57419 ref_len = 55920)
2. **Using Scratch_beam_search Function:**
BLEU = 27.48 57.4/33.3/21.3/14.0 (BP = 1.000 ratio = 1.012 hyp_len = 56568 ref_len = 55920)

For Test Set:
1. **Using .generate with beam search:**
BLEU = 26.23 56.4/32.2/20.2/12.9 (BP = 1.000 ratio = 1.004 hyp_len = 62252 ref_len = 62014)
2. **Using Scratch_beam_search Function:**
BLEU = 26.39 56.9/32.6/20.5/13.1 (BP = 0.993 ratio = 0.993 hyp_len = 61605 ref_len = 62014)

Since we observe similar scores for the test and validation dataset, we consider our scratch_beam_search translation a suitable baseline to be used for implementing lexical constraint in the output sequence.

Identifying Lexical Constraints:
As discussed in segment 4.1 of Hokamp and liu [1] we simulate user interaction by choosing phrases of up to three tokens(tri-grams) from the reference translation that are missing from the current machine translation for a sequence.

For first iteration we get baseline translation and identify subsequence in reference translation missing from hypothesis translation(baseline in this case), use it as a constraint to get translation through our constrained_beam_search(CBS) function.

For second iteration we get the translation from first iteration and identify subsequence in reference translation missing from hypothesis translation(first iteration in this case), and then add that as second constraint in CBS to get translation and so on and so forth.This is why the constraints for each sequence is store in a list and we keep adding to the list in each iteration.

Implementing Lexical Constraints:
Now in the code I define a constrained_translate function that tokenizes the model_generated_list and target_reference list.Then runs a loop in which it calls find_missing_tokens function to return the missing tokens for the sequence in current iteration to be used as constraints and adds them to them to the constraint list for the current sequence. Then it calls constrained_beam_search to include the all constraints in the list for that sequence to the output. Then it adds the translation of that sequence to the constrained_translated_list.

3. Results and Abilities of Constrained Beam Search:
In the outputs of every single sequence it is observed that the constrained_beam_search function always successfully implements the constraints specified at the required positions of an output sequence. On a sequence-by-sequence observation we almost always see either the same or better BLEU score compared to baseline(scratch_beam_search function).

A peculiar observation is that the overall BLEU score for the entire test split does not actually change much, even reduces. I think this is because the way the Helsinki model generate tokens during each decoding step is different than the tokenizer. During forward pass the model generates larger token_ids but lesser in number and the tokenizer uses smaller token_ids but more in number to represent the same sentence, which is causing loss in information hence offsetting the bleu score increase from implementing the constraints. This discrepancy is further discussed in detail in the last section of the code in the Jupyter Notebook. The Last cell of the submitted python notebook evaluates BLEU score for different sequences and verifies that the constraints are indeed being added, which could be explored at the readers discretion.

BLEU score for eval split translation using constrained_beam_search:
1. **First Iteration of Implementing Constraints:**
BLEU = 26.34 56.7/32.2/20.2/13.0 (BP = 1.000 ratio = 1.005 hyp_len = 56172 ref_len = 55920)
2. **Second Iteration of Implementing Constraints:**
BLEU = 26.60 57.2/32.5/20.5/13.2 (BP = 1.000 ratio = 1.010 hyp_len = 56485 ref_len = 55920)
3. **Third Iteration of Implementing Constraints:**
26.47 57.1/32.3/20.4/13.1 (BP = 1.000 ratio = 1.009 hyp_len = 56443 ref_len = 55920)
4. **Fourth Iteration of Implementing Constraints:**
26.32 57.1/32.2/20.2/12.9 (BP = 1.000 ratio = 1.011 hyp_len = 56549 ref_len = 55920)

4. Implementation Hurdles

Computation Resources
The most substantial Hurdle throughout the entire development process has been trying to find work arounds for my computation limitations, since my personal system does not have a dedicated GPU, getting the results for the entire validation and test set are the hardest especially for subsequent iterations of constraints. For many segments in this implementation, I tried to work around this by checking that I was getting the desired result for one single sequence, then trying 10 random sequences from the validation split of the dataset, but I always had to resort to keep going back to google collab for actual computation of the entire split.

Now 2 major problems with Google Collab is that its usage limit gets hit at random times within an hour of computation and then your runtime gets disconnected deleting all the state information that would have been required for further segments. And since it only allows google drive storage which is very limited in size, I was constantly figuring out whether and how to save or not save something on drive.

Which is why I would either keep figuring out ways to save state information required for further steps or just get back to square one. But many a times you don't know which state information you would need further down the line and you don't save it from the current runtime which again forces you to wait for the next Google Collab limit and try to get the required element. Only if the usage limit is not reached before getting the output was I able to make any inference about whether what I was currently doing was right or wrong for the scale of the dataset and then be able to return back to actually making progress in the implementation.

This kind of Trouble shooting was not only an inconvenience but also took up a massive chunk out of the actual implementation time. If the computation resources were continuously available the implementation would have definitely been of a much greater scope than it currently is.

References

1. Chris Hokamp and Qun Liu. 2017. Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1535–1546, Vancouver, Canada. Association for Computational Linguistics.
2. "WMT 2016 Russian-English Dataset," *Hugging Face*, https://huggingface.co/datasets/wmt/wmt16/viewer/ru-en.
3. "WMT 2019 Russian-English Dataset," *Hugging Face*, https://huggingface.co/datasets/wmt/wmt19/viewer/ru-en.
4. "Google mT5 Base Model," *Hugging Face*, https://huggingface.co/google/mt5-base.
5. "Helsinki-NLP OPUS-MT English-Russian Model," *Hugging Face*, https://huggingface.co/Helsinki-NLP/opus-mt-en-ru.