Unlimiformer Analysis

1. Abstract

This paper proposes an extension to the vanilla transformer structure, enabling it to handle unlimited input length. The idea is that the encoder hidden states used in cross-attention can be simplified by using a kNN search over the encoder hidden states. The top-k hidden states are then used in the cross-attention computation, effectively reducing the computational complexity while being a good approximation. As quoted from the paper:

> In preliminary experiments, we found that the top-k attention keys cover more than 99% of the attention mass, and thus attending only to the top-k keys is an accurate approximation of the full, exact, attention.

2. Novelty

The kNN search is an improvement to the approach in *Memorizing Transformer* where the attention computation formula for cross-attention is rearranged in the form:

$$\begin{aligned} QK^T &= (\boldsymbol{h}_d W_q)\,(\boldsymbol{h}_e W_k)^\top \\ &= (\boldsymbol{h}_d W_q)\,W_k^\top \boldsymbol{h}_e{}^\top \\ &= \left(\boldsymbol{h}_d W_q W_k^\top\right) \boldsymbol{h}_e{}^\top \end{aligned}$$

The retrieval objective is to retrieve a set of keys K_{best} to maximize QK_{best}^T. With this formula we can use **only one index storing $\mathbf{h}_e$** across all layers and heads of the decoder. If we use the keys $\mathbf{h}_e W_q$ directly we need to use an index for W_q in every layer and every head, which would be memory and time inefficient due to having to store more indices and spend time initializing them.

Furthermore, the kNN search does not introduce any additional parameters to the model. It can also be used in all stages including training, evaluation and test time inference. The simplest use case quoted from the paper, is as follows:

> As the simplest case, we use a standard fine-tuning regime, where the input is truncated during training. At inference time only, we inject Unlimiformer into the trained model to process full-length inputs.

This benefit is further exploited in the Unlimiformer code implementation. Specifically, the Unlimiformer implementation can be viewed as a plugin to transformer models, meaning they can take any transformer model as a base model. At runtime, the model's methods (e.g. train, eval, generate) will be replaced by hooks that carry out the additional processes needed for the Unlimiformer. For a concrete example, here is a line of code that initializes the base model in a *simple inference*:

```
modelname = "abertsch/unlimiformer-bart-govreport-alternating"
dataset = load_dataset("urialon/gov_report_validation")

tokenizer = AutoTokenizer.from_pretrained("facebook/bart-base")
# This model is a Bart model, not an Unlimiformer model!
model = BartForConditionalGeneration.from_pretrained(modelname)
```

It is worth noticing that instead of creating and initializing from another model instance, the code uses the `BartForConditionalGeneration` class that represents the model for *BART*. This model is then integrated to the Unlimiformer with only modifications to the functions required for evaluation and generation. This plugin-like implementation makes Unlimiformer flexible to use in transformer models.

3. Experiments

The experiments show a certain level of performance gain in long document datasets, which is not surprising, given that the kNN search method is first used partially in *Memorizing Transformer* that achieves better performance. Furthermore, the authors performed an analysis on the impact of input length on entity recall in summarization. On the longest input dataset *BookSum*, they found the following:

> EntMent (Entity Mention Recall) increases almost monotonically with input length, suggesting Unlimiformer exploits the longer inputs to generate better outputs.

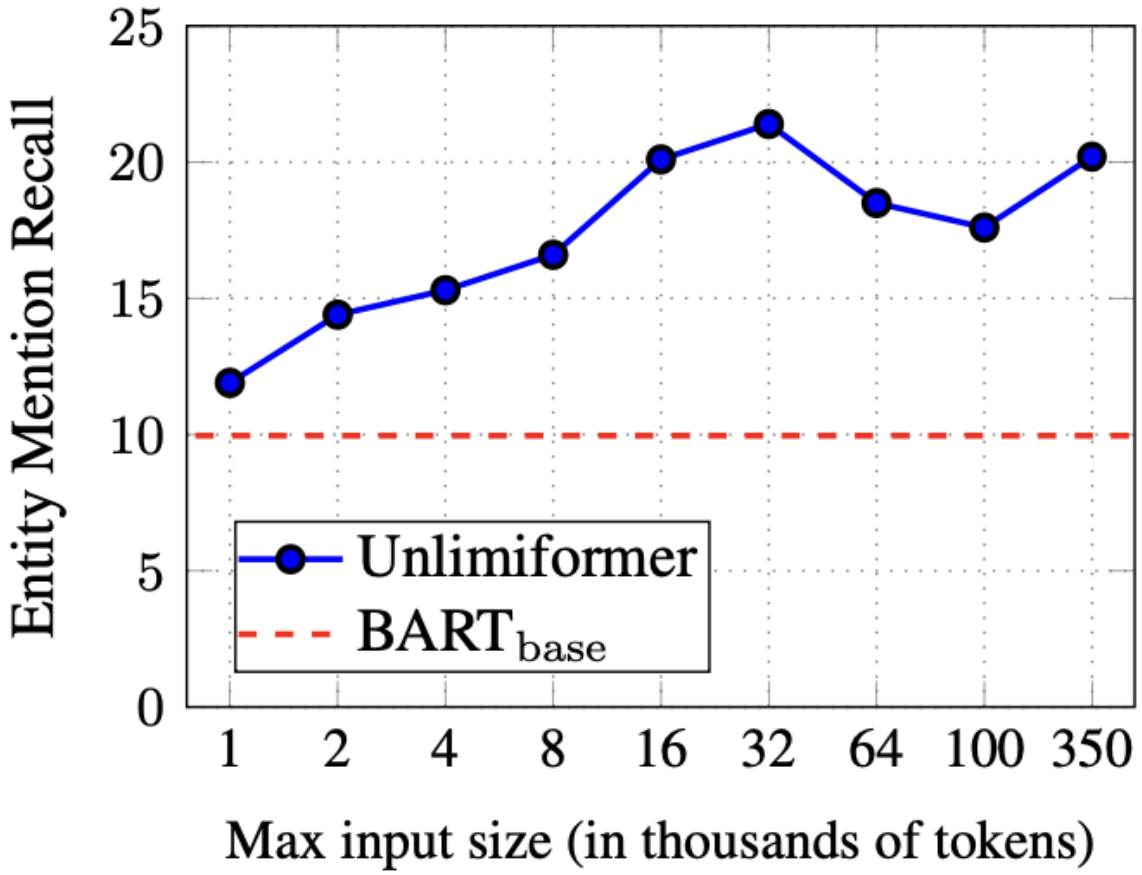


Figure 3: As the maximum datastore size increases, the entity recall generally increases. At all datastore sizes, Unlimiformer outperforms the baseline (BART, in red).

4. Limitations

Training configurations

The training configurations were not compared in a way where one setting changes while the other stay the same. Specifically, in section 3.2 of the paper, three training

methods used are:

> ***Random-encoded training:*** At each training step, the full (longer-than-context-window) training example is encoded in chunks; then, the keys for each decoder layer are chosen randomly from the encoded hidden states. This weakly simulates a nearest-neighbors search, but is computationally cheaper.
>
> ***Retrieval training:*** At each training step, the keys for each decoder head and layer are selected using a kNN search. When inputs are longer than 16k tokens, we truncated the input to 16k tokens at training time due to GPU memory requirements. This training approach is the closest to the test-time computation.
>
> ***Alternating training:*** In this approach we alternate batches of *Random-encoded training* and *Retrieval training*. *Retrieval training* is identical to the test-time setting, while *Random-encoded* introduces regularization that makes the model attend to non-top-k keys as well.

In **random-encoded training**, the encode step uses full length input and the encoded hidden states are selected randomly. In **retrieval training**, the encode step truncates input and the encoded hidden states are kNN retrieved. The **alternating training** is just a combination of the previous two.

With settings that differ in multiple aspects, it is hard to locate which component contributes to performance gain the most. For instance, we cannot conclude that kNN search works by observing better results using **retrieval training** than **random-encoded training**, since their encoding methods are also different.

Datasets

Only three datasets are used in the experiments which might not be convincing enough to say the method proposed works. It would be useful to explore results on other long document datasets.

Resource requirements

This point is not exactly a limitation but a reminder of the computing resources required for training:

> In our experiments, we were able to use a GPU index for input examples exceeding 500k tokens (on GPUs no larger than 48 GBs), but this may be a concern when using smaller GPUs or larger models.

5. Conclusion

This paper presents an approach to extend transformer models with a non-parametric kNN search component. The method has an advantage of reducing time complexity and being mountable on any base transformer model. The open-sourced code also seems to be quite comprehensive, and supports running on huggingface which makes reproducing results much easier and manageable.

Thoughts on the paper:

The methods proposed here are quite practical since they are able to apply kNN search to every decoder layer, unlike the *Memorizing Transformer* approach which only uses kNN on one decoder layer. Additionally, the Unlimiformer implementation is flexible and can be used with different transformer models. If we want to test Unlimiformer with a different dataset, we can simply finetune a model like BART on that dataset and use it as a base model for Unlimiformer.

However, the methods proposed in the paper are all based on previous methods. There are no fundamental innovations in aspects like model structure and training methods. Moreover the authors did not conduct sufficient experiments to comprehensively examine the impact of each component in the model, hence it is difficult to determine which methods work and which ones don’t.

In my opinion, this model falls under the same category as models like LongT5, Longformer, etc., since they all propose extensions to make the attention computation more efficient while still being a good approximation. Based on the original project proposal, I would consider adding Unlimiformer to be another one worth exploring, but it

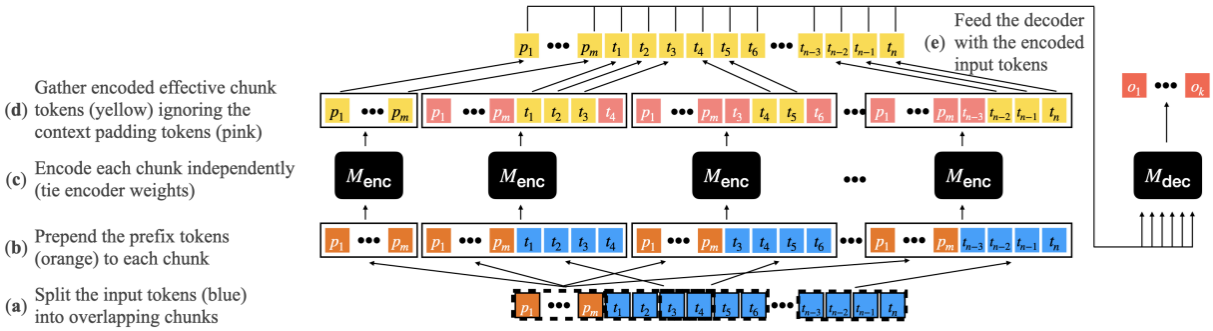
might not be that useful to study it in detail. Nevertheless, there are some investigations I can think of that may be interesting:

- Do more experiments suggested in the Limitation section
    - Conducting a thorough experiment to investigate which component works best will provide insights into the effectiveness of each component of Unlimiformer. However, it might be a time-consuming process to run that many experiments.
- Investigate the kNN component
    - The idea of using a kNN component appears in many papers, and faiss is one of the best-performing libraries used for kNN search. Therefore, the characteristics of such kNN components can be worth studying. There are many *approximate search algorithms* available in faiss that have different advantages and use cases. In addition, retrieving from and potentially updating indices are also important in model performance. In section 2.4 of the paper *Atlas*, the authors analyzed the computational overhead imposed by actions like index update, which could be expensive in model training. Studying such topics can provide insight into the amount of time required when using kNN approaches.

Appendix

SLED structure

In Unlimiformer the authors also used an approach from SLED to efficiently encode long inputs. The idea is to split the long input into overlapping chunks and use an encoder with a limited context length to encode each chunk. Subsequently, only the central tokens within each chunk are selected and combined to obtain the final representation. This approach is beneficial because the central tokens retained in each encoded chunk have a contextual reference around them, which is a better way compared to encoding them independently. The structure is presented as follows (copied from the original paper):



This approach is used in some settings of the Unlimiformer training and testing, and perhaps it would be interesting to see whether it can boost performance.