Auto-Regressive MLM on Trees – A Tutorial

Jeevesh Juneja

January 12, 2021

1 Task

In social media platforms like Reddit, we often see the formation of a tree-like structure of comments, rooted at the first post(”comment tree”). Henceforth, by ”post” we refer to the OP’s post and by ”comment” we refer to any of the following comments in the entire tree. A ”top level comment” refers to a comment that is a direct comment on the OP’s post and will lie at depth 1 in the comment tree. A sequence of post and following comments $\{p = c_0, c_1, c_2, ..., c_n\}$ is called a thread if p is the post and c_i is a child of c_{i-1} for all $0 \leq i \leq n$. For every comment, c, there is a unique thread, T_c that ends with it. Our task now is to predict the masked tokens in any comment(c), as in [Dev+19], while additionally, conditioning this prediction on all the comments before c in T_c. That is, the prediction of masked tokens of any comment is conditioned on the post and all the comments on the path from the post to the comment in the comment tree.

2 Model Architecture

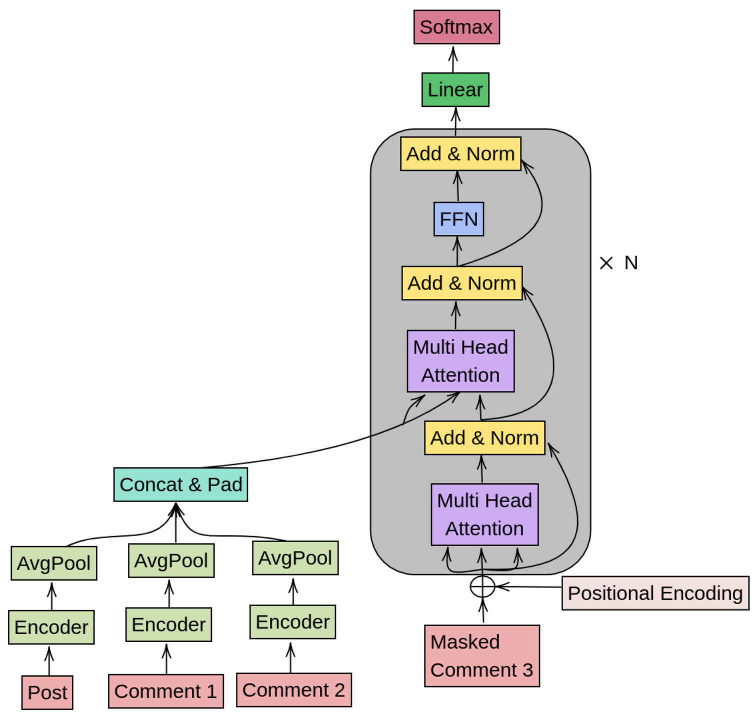

Figure 1: Model architecture: The input to a model is a thread, ending at comment 3. The model has to predict the masked tokens in comment 3. The Multi Head attention blocks are the same, as in [Vas+17].

2.1 Encoding Context

The model consists of an encoder part, that encodes each of the previous comment in a thread, using an encoder identical in architecture to the transformer encoder of [Vas+17]. This encoder outputs a contextual embedding for each position of the input. We pool the embeddings over all positions obtaining a fixed length vector corresponding to each post/comment we want to condition our prediction on. The maximum number of embeddings we can condition on, is a hyper-parameter(cl), of the model. It is set to the same value as the maximum possible number of tokens in a post/comment. After obtaning all the fixed length embeddings, we concatenate together fixed-length feature embedding of each previous post/comment, and pad the tensor along the comment/post dimension to length cl. Note here that although, each post/comment is embedded into a fixed size vector, the context **is not** a fixed size vector. The size of the context, increases linearly with the number of comments to condition on, until number of comments becomes cl, after which it becomes constant, equal to cl.

2.2 The Decoder and Conditioning

The last comment of the thread is masked in the same way as [Dev+19]. The conditioning mechanism is simple multi-head attention mechanism with the embeddings of the masked comment acting as the queries and the context (matrix of size $cl \times d_{model}$) prepared by the encoder acts as keys and values. The decoder architecture is similar to the transformer decoder of [Vas+17], except the first multi-head attention is not masked. We predict the tokens at the masked position and back-propagate the loss.

3 Training and Biases

3.1 A First Approach

The above model and task seem to fit nicely together at first. It seems obvious, to pick all possible threads one by one, and run the model, as described above on each of those threads. But, it would be quite computationally expensive to embed all the previous comments just to predict a few mask tokens in the last one. We will end up running the encoder $|T_c|$ times to predict masked tokens in 1 comment.

A general way to resolve this, is to adopt an approach somewhat similar to back-propagating loss at each step of RNNs rather than just the last. That is, when you have embedded all the elements of the thread T_c, using the encoder, then don't just back-propagate loss from the last position, but from all intermediate positions too. So, we mask tokens in c_i and predict those masked tokens conditioned on $(p, c_1, c_2 ... c_{i-1})$, for all i. Doing this, allows us to predict masked tokens for $|T_c| - 1$ while running the encoder $|T_c|$ times.

3.2 The Bias in Training

But this approach, introduces a bias if we go on iterating over all threads. Two threads, may differ only at the last element. When we back-propagate losses for each of the position in each of the threads, we will be back-propagating loss for masking in the first $|T_c| - 1$ comments twice. In general, the number of times the masked tokens of a comment are predicted will drop exponentially with the depth of the comment in the comment tree. Thus, the comments closer to root are given more preference.

To resolve this, we go further, and decide to move to the domain of comment trees rather than threads. We embed each post/comment in an entire comment tree and then predict masked tokens of each comment(C) in the tree, conditioned on the embeddings of all the previous elements in the unique thread T_C, corresponding to C. We add all the losses, then back-propagate. There is another good side-effect of this approach, the embedding of each comment gets gradients from all the comments in the sub-tree rooted at that node. Only when all the gradients are accumulated, do we take a step. This ensures that the embedding moves in a direction that is beneficial to all the comments in the sub-tree. The encoder got gradients from all nodes in the previous approach too, but the gradients here are less stochastic.

3.3 The Middle Path

But this approach, is quite memory-consuming, especially in case of large comment trees, for example with around 500 comments. So, to tackle that, we break big comment trees into smaller trees. Each of which consists of a single thread(T_c) up-to some depth, followed by the entire sub-tree rooted at the last comment c. The total number of comments in this new tree is bounded by a hyper-parameter('max-tree-size'). If the size of tree is larger than this, then the thread is made longer and sub-tree at a greater depth. Making 'max-tree-size' smaller moves us from the comment tree based approach back to the thread based approach.

References

[Dev+19] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: `1810.04805 [cs.CL]`.

[Vas+17] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: `1706.03762 [cs.CL]`.