Mixture-of-Tokens: Efficient LLMs Through Cross-Example Aggregation

Main Idea: traditional sparse MoE has some drawbacks - the router decision is discrete, making it not fully differentiable for training, which can cause training instabilities; the load balancing between experts is also not guaranteed, which leads to the need to apply methods such as using an auxiliary loss or adding random noise to training inputs, which do not guarantee solving this challenge. MoT tries to improve on the traditional MoE architecture, providing a fully differentiable strategy which automatically results in load balancing.

OBS: MoT is compatible with both masked and causal LLM training and inference (fill missing tokens and autoregressive language modeling).

Issues with MoE

- The router is discrete, which causes training instabilities since small changes in the input may cause big changes in the gradient (if the small change in the input results in a different router selection). This makes the training process not fully differentiable. Using a weighted average of the selected experts to form the outputs seems to help with this but is not an optimal solution.
- There is no guarantee that the MoE will distribute loads evenly among experts. A capacity factor (CF) can be set for minimizing token dropping, but this does not help with load balancing and increases memory requirements. This prompts the use of an auxiliary loss, which is, again, not ideal.

- Most studies done with MoE are not compatible with autoregressive decoding (take soft MoE for example).

MoT Algorithm

1. The first step is to pass the input tokens (all of them) through a router/controller (linear layer) and apply a SoftMax to get the token importance scores for each expert.
    a. $Imp_{weights} = Softmax\big(Linear(tokens)\big)$
        i. Where $Imp_{weights}$ is a matrix with each input token as a row and each expert as a column.
        ii. Each column sums up to 1, so each expert has its own designated router.
2. Then, the tokens are mixed by their importance weights, forming a mix of tokens for each expert.
    a. So, the token mix passed to expert i is -> $\sum_t^T token_t * imp_{weights_i}$
3. After having the mix of tokens for each expert, the next step is to pass the expert's mix of tokens through its respective FFN.
4. To obtain the final output for a specific expert, we need to scale the expert output based on the importance for each token in its mix:
    a. Final_output (for token t and a given expert) = expert output * imp_weight for token t
5. The final output for a given token t is then the sum of all the final outputs of each expert for that token t.

When doing this process for decoding, having to recompute each token multiple times seems inefficient, so a strategy to group tokens needs to be employed.

- The authors group tokens according to their position in a sequence (1st tokens grouped together, 2nd tokens grouped together, etc.). This way, for a given batch, each sequence can be computed in parallel, token-by-token.

Experiments

- The authors compare a GPT-like model to a MoT model (Transformer architecture with all feed-forward layers replaced with MoT layers).
- The MoT model shows promising pre-training results, achieving the vanilla Transformer's final loss in 1/4th of the training steps and 1/3 of the training time.

My takeaways:

- Intuition about the MoT algorithm:
    - Calculating the importance vector of the input tokens is done at the expert level. This means that the input tokens are passed through the router for expert n, which will give the importance weights of each token for that expert. This is calculating how the mix of tokens which is passed to each expert will be weighted.
    - MoT sounds like Expert Choice Routing in terms of the expert choosing the importance to give to each token (in Expert Choice, however, the method used to

determine if the token will be sent to an expert is given by the affinity or importance weight given by the expert, while in MoT every token is considered by every expert).

    - To get a final output, each token looks at the output of each expert, and considers how much importance to give to each expert's output based on the importance the expert gave it.

- Although it is possible to do natural language generation with this approach, it seems to be very inefficient since generation of tokens cannot be done in parallel for all input tokens in the same sequence, while this approach takes all input tokens in the same sequence in consideration during inference and performs a forward pass in all experts.
    - Highly impractical in its given form for language generation. The design presented only works at a batch-level.
    - These limitations create the need for future research to make this approach practical.
- For now, this is only a training strategy, but does not work for inference.