FNLP Tutorial 4

1 Parallelization of transformers

One of the greatest advantages of the Transformer model is that it is possible to parallelise its computation, and therefore train a model on much larger training datasets (for example, for pre-training) with less compute.

1. Consider an encoder-decoder RNN model, following up on the question before. Can the computation of the encoder be easily parallelised with respect to the number of tokens (meaning, can you break the input and the computation into chunks such that they run in parallel)? Explain why or why not.

   Solution An RNN encoder needs to compute the activation of step i, before being able to compute the activation at step $i+1$. Therefore, RNN computation is not easily parallelisable. New RNN methods, such as state-space models try to overcome some of these issues by building the RNN in such a way that it has certain properties that allow more efficient compute ("associative scan") [1].

2. Consider the Transformer encoder layer. Explain why its computation can be paralleised over tokens *per layer*. Can computation be easily parallelised across layers?

   Solution The computation of one Transformer layer activation can be done in parallel. More specifically, the computation of the activation for token i requires integrating information from the *previous* layer of all other tokens, but not from the *current* layer. This also means that the computation cannot be easily parallelisable across layers.

2 Self-attention mechanism

1. The QKV attention can be viewed as an operation on a *query* vector $q \in \mathbf{R}^d$, a set of *value* vectors $\{v_1, \ldots, v_n\}$, $v_i \in \mathcal{R}^d$, and a set of *key* vectors $\{k_1, \ldots, k_n\}$, $k_i \in \mathcal{R}^d$, specified as follows:

$$c = \sum_{i=1}^{n} \alpha_i v_i \tag{1}$$

$$\alpha_i = \frac{\exp(k_i^T q)}{\sum_{j=1}^{n} \exp(k_j^T q)} \tag{2}$$

   with $\alpha_i, \ldots, \alpha_n$ termed the "attention weights". Observe that the output $c \in \mathbf{R}^d$ is an average over the value vectors weighted to α.

   (a) Try to construct a set of queries $\{q_1, \ldots, q_n\}$, values and keys so that the output c for an incoming query q_i is always equal to *some* value vector v_j, i.e. $\forall q_i \rightarrow \exists j \in \{1, \ldots, n\}$ *s.t.* $c = v_j$. Explain what properties about the sets of values, keys and queries would be desirable to achieve this.

   (b) Now, let's move to the case with only two key and value vectors, e.g. $n = 2$. What query vector would have exactly the same attention to both value vectors (i.e. $\alpha_1 = \alpha_2 = 0.5$)?

Solution

(a) We could have that any incoming query vector is always orthogonal to all the key vectors except one. If the key vectors are orthogonal between themselves (e.g. they form an orthogonal base/subspace), then we could have query vectors that are another key vector multiplied by a constant. In this case, we could pick any value vectors.

(b) We have that $k_1^T q = k_2^T q$, therefore $k_1^T q - k_2^T q = (k_1^T - k_2^T)q = 0$ and we have that q must be orthogonal to the difference vector $k_1 - k_2$.

3 KV Cache

In an auto-regressive Language Model (LM), only one token gets produced at a time by conditioning on previously generated tokens. A key component of these models is the self-attention mechanism, which allows each token to attend to others in the sequence, determining how much influence each token should have on the next prediction.

1. Consider the sequence of $N = 3$ tokens *The cat is*. Write the matrix form of self-attention, specifying the key, query and value vectors for each token.

   Solution Let's say we have the matrix of embeddings

   $$X = \begin{bmatrix} x_{the} \\ x_{cat} \\ x_{is} \end{bmatrix} \tag{3}$$

   We have the query, key, value matrices $Q = XW_Q$, $V = XW_V$, $K = XW_K$. The self-attention equation is:

   $$\text{Attention} = \text{softmax}(\frac{QK^T}{\sqrt{d}})V \tag{4}$$

   where d is the embedding dimension.

2. For an output sequence of length N, what is the time complexity of the self-attention mechanism?

   Solution From the self-attention equation, the dominant term is the multiplication of QK^T, which leads to a $O(N^2)$ complexity.

3. Imagine we generated the sequence *The cat is*, and want to generate the next token. What new operations do we need to perform now?

   Solution Instead of computing self-attention for all tokens, we only need to compute it over the last token. This means, we can avoid the matrix multiplication QK^T: Attention = $\text{softmax}(\frac{qK^T}{\sqrt{d}})V$, where $q = x_{is}W_Q$.

4. What is the time complexity of, given a sequence of N tokens, generating an additional token?

   Solution We need to multiply the query of the last token q with K, therefore, the time complexity is $O(N)$.

5. Imagine that we decide that, for every new token that we compute, we will store all the vectors that will be reused to compute new tokens. What vectors will we store?

Solution For each new token that we produce, we will store the key and value vectors. This means that we will have the K, V matrices in memory, which is commonly known as KV Cache.

6. This caching mechanism, typically known as KV Cache, significantly speeds up text generation. However, KV cache requires additional memory. How would this scale with context length?

   Solution We need to store the K, V matrices, which are $N \cdot d$ each. Therefore, memory scales $O(N)$ with context length. For very long contexts, this can become a limiting factor in deployment, as the GPU memory requirements grow substantially.

References

[1] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces.* arXiv preprint arXiv:2312.00752, 2024. https://arxiv.org/abs/2312.00752