ECE 587 – Hardware/Software Co-Design
Lecture 21 Large Language Models

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

April 2, 2025

Outline

Reading Assignment

- This lecture: Large Language Models
  - Attention Is All You Need, Vaswani et al. `https://arxiv.org/abs/1706.03762`
  - Llama `https://github.com/meta-llama/llama-models`
- We will study state-of-the-art hardware accelerators and interconnection networks for the rest of the semester.

Outline

Large Language Models

Llama Models

Language Models Overview

- Tokenization: convert text into sequences of tokens
- Embedding: represent tokens as vectors
- Encoder $C' = E(C, x)$: input one token at a time
- Decoder $(Pr, C') = D(C)$: output probability of next token
- Autoregression: $(Pr, C') = D(C, x^{-1}, x^{-2}, \ldots, x^{-N})$
  - Make use of previously generated output tokens.
- Challenges
  - How can we design encoders and decoders as neural networks?
  - How to define loss functions to train models?
  - How to obtain data for training?

Decoder-only Models

$$Pr_{N+1} = D(x_1, x_2, \ldots, x_N)$$

- When the window size N is large enough, the whole input sequence can be included as if they are generated first.
  - Let's rename the symbols to be consistent with literatures.
- Introduce special tokens to indicate end of input.
  - Prompt the decoder to generate actual output tokens.
- No need to use encoder and context any more.
  - Context, similar to state in a FSM, makes it difficult to parallelize the computations, in particular for training where a lot of data need to be consumed efficiently.

Considerations for Training

$$(Pr_2, Pr_3, \ldots, Pr_{N+1}) = D(x_1, x_2, \ldots, x_N)$$

- The decoder model actually predict probability Pr_2, Pr_3, $\ldots$ for known tokens x_2,x_3,$\ldots$ in addition to the next token.
  - A model architecture matching lengths of input and output.
- A loss function can be defined between actual tokens $(x_2, \ldots, x_{N+1})$ and predictions $(Pr_2, \ldots, Pr_{N+1})$.
  - Masking: ensure that probabilites are only computed from previous tokens, like how we read a sentence word by word.
  - For example, Pr_2 should only depend on x_1, and Pr_N should only depend on $(x_1, \ldots, x_{N-1})$ but not x_N.
- Learn D from vast amount of text via unsupervised learning, without the need to label data by human beings.
- How to build neural networks for D?

Attention: Query

- Attention: a neural network layer that allows to extract data from a sequence of arbitrary length.
- Query q: a vector representing a pattern of interests.
  - Assume q to have the same size as x_i, i.e. both are $d \times 1$ vectors. Then the inner product $q^T x_i$ is a scalar representing how similar q and x_i are.
- Use inner products to score tokens: $(q^T x_1, q^T x_2, \ldots, q^T x_N)$
  - Token with higher score will contribute more to extracted data.
  - Use softmax to calculate weights for each token and extracted data as a weighted summation of all tokens.
- Attention with query: softmax$(q^T \boldsymbol{X}^T)\boldsymbol{X}$
  - $\boldsymbol{X}$ is a matrix with N rows $x_1^T, \ldots, x_N^T$, and d columns.
  - $q^T \boldsymbol{X}^T$ gives a $1 \times N$ row vector and so does softmax.
  - softmax$(q^T \boldsymbol{X}^T)\boldsymbol{X}$ extracts a $1 \times d$ row vector from the input sequence of arbitrary length with the given query q.

Attention: Keys and Values

- What if we would like to have more flexibility so both query and output could have a different size?
- Keys: $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^K$ where $\boldsymbol{W}^K$ are the weights
  - Query with the key instead of the tokens.
  - Assume $\boldsymbol{W}^K$ is a $d \times d_k$ matrix.
  - $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^K$ is a $N \times d_k$ matrix.
- The scores and weights become softmax$(q^T \boldsymbol{K}^T)$
  - q will have a matching size of $d_k \times 1$.
  - $q^T \boldsymbol{K}^T$ gives a $1 \times N$ row vector and so does softmax.
- Values: $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^V$ where $\boldsymbol{W}^V$ are the weights
  - Extract data as weighted summation of value instead of tokens.
  - Assume $\boldsymbol{W}^V$ is a $d \times d_v$ matrix.
  - $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^V$ is a $N \times d_v$ matrix.
- Attention: softmax$(q^T \boldsymbol{K}^T)\boldsymbol{V}$, a $1 \times d_v$ row vector

Self-Attention

- Is it possible to use multiple queries and how to obtain them?
  - Yes and we can obtain them from the input sequence itself.
- Queries: $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^Q$ where $\boldsymbol{W}^Q$ are the weights
  - Query the input sequence with itself.
  - $\boldsymbol{W}^Q$ is a $d \times d_k$ matrix and $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^Q$ is a $N \times d_k$ matrix.
  - Each row of $\boldsymbol{Q}$ is a query and there are N queries.
- $\boldsymbol{Q}\boldsymbol{K}^T$ computes scores between the N queries and N keys.
  - Each row contains scores for a single query with all keys.
  - We can apply softmax row by row to obtain weights.
- Self-Attention: $\text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V}$, a $N \times d_v$ matrix.
  - $\boldsymbol{Q}\boldsymbol{K}^T$ is scaled by $\sqrt{d_k}$ as its elements get larger when each query and key becomes longer.
  - Learn all the weights $\boldsymbol{W}^Q$,$\boldsymbol{W}^K$,$\boldsymbol{W}^V$ during training.

Masking

$$\text{Self-Attention}(\boldsymbol{X}) = \text{softmax}(\frac{\boldsymbol{QK}^T}{\sqrt{d_k}})\boldsymbol{V}$$

- Self-Attention($\boldsymbol{X}$) outputs a $N \times d_v$ matrix, which can be treated as an output sequence with the same length as $\boldsymbol{X}$.
  - $\boldsymbol{QK}^T$ is a $N \times N$ matrix.
  - An element at ith row and jth column of $\boldsymbol{QK}^T$ controls how the input j contributes to the output i.
- For masking, output i should only depends on input $1, \ldots, i$.
  - Set elements in $\boldsymbol{QK}^T$ with $i < j$ to $-\infty$ before softmax.
- For inference, masking enables the use of KV cache so that one can compute Pr_{N+1} efficiently for the next token.
  - No need to recalculate $Pr_2, \ldots, Pr_N$ for previous tokens.

Multi-Head Attention

$$\text{head}_i = \text{Self-Attention}_i(\boldsymbol{X})$$

$$\text{MultiHead}(\boldsymbol{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\boldsymbol{W}^O$$

- Learn multiple (h) sets of $(\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V)$
- Each generate a $N \times d_v$ matrix as output using self-attention.
- Concatenate the outputs into a $N \times hd_v$ matrix.
- Learn the matrix $\boldsymbol{W}^O$ of size $hd_v \times d$ as the output weights so the overall output has the same size $N \times d$ as the input.
- Multi-head attention provide a lot of opportunities for parallelization.
- When input and output are of the same size, we can stack many of the same layers for a deeper and larger model.

Positional Encoding

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}^Q, \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}^K, \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}^V, \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V}$$

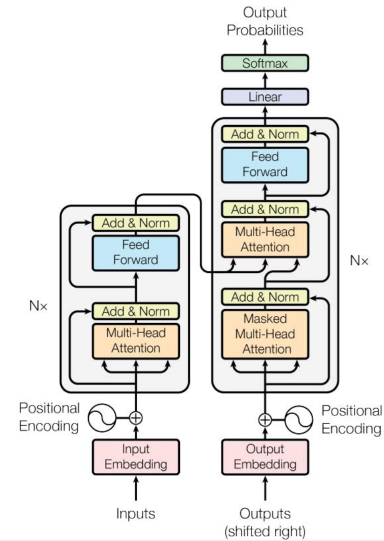
- Reorder the input sequence results in reordering rows of $\boldsymbol{X}$.
- Rows of $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ will be reordered the same way.
  - Though their values remain same.
- $\boldsymbol{Q}\boldsymbol{K}^T$ will be reordered in a way such that the output of softmax is only a reordering of the original one.
- Not correct since words mean differently at different locations
  - E.g. "You own me \$100" and "I own you \$100".
- Choose a sequence of vectors to represent the N positions and add them to $\boldsymbol{X}$, or to $\boldsymbol{Q}$ and $\boldsymbol{K}$.
- While attention can handle arbitrary sequence lengths, the need to maintain positional information makes it difficult to use a different sequence length than that used for training.

Position-wise Feed-Forward Networks (FFN)

$$\text{MultiHead}(\boldsymbol{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\boldsymbol{W}^O$$

- The output of $\text{MultiHead}(\boldsymbol{X})$ as a $N \times d$ matrix can be viewed as a sequence of N row vectors.
- Introduce additional non-linearity and capacity by transforming individual output vectors identically.
- Make use of multiple fully connected (MLP) layers, e.g.
$$\text{FFN}(\boldsymbol{y}) = ReLU(\boldsymbol{y}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2$$
  - $\boldsymbol{y}$ is a row vector from the output of multi-head attention.
  - Learn weights and bias's $\boldsymbol{W}_1$, $\boldsymbol{b}_1$, $\boldsymbol{W}_2$, $\boldsymbol{b}_2$ during training.
  - The same set of $\boldsymbol{W}_1$, $\boldsymbol{b}_1$, $\boldsymbol{W}_2$, $\boldsymbol{b}_2$ are used for all rows.

Transformer



(Figure 1, Attention Is All You Need, Vaswani et al., 2017)

- The original transformer model contains both encoder and decoder.
- Stack of FFN and attention layers.
  - With layer normalizations and residual connections.
- Probabilites are generated at each output position identically.
  - First, a linear layer transform the output vector of size d into a vector of size M.
  - Then, apply softmax to obtain the probabilities at this position.
- Remove encoder related parts to obtain a decoder-only transformer.

Outline

Large Language Models

Llama Models

Llama Models

- Llama (Large Language Model Meta AI)
  - Open and efficient foundation language models
  - Llama 2 (2023): up to 70B parameters with window size $N = 4096$
  - Various Llama 3 versions (2024): up to 405B parameters with window size $N = 128k$
- A decoder-only (autoregressive) transformer model.
  - Reference implementation for inference is provided in PyTorch.
  - Trained models (weights) are available for download after signing an agreement with Meta.
  - A lot of open-source implementations to support quantization, efficient CPU inference, fine tuning, etc.

Example: LLaMA-2 13B

- Tokenization: $M = 32000$ different tokens.
- Embedding: each token vector has a size of $d = 5120$.
  - $32000 * 5120 \approx 160M$ parameters for embedding.
- 40 layers of FFN and attention
  - Each attention layer has $h = 40$ heads and $d_k{=}d_v{=}\frac{5120}{40}{=}128$.
    - $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$, $\boldsymbol{W}^V$ have the same size $5120 * 128 \approx 650K$.
    - $\boldsymbol{W}^O$ has a size of $5120 * 5120 \approx 26M$.
    - All 40 sets $\boldsymbol{W}^Q$, $\boldsymbol{W}^K$, $\boldsymbol{W}^V$, plus $\boldsymbol{W}^O$, have $650K * 3 * 40 + 26M \approx 104M$ parameters.
  - Each FFN has two fully-connected layers that map a vector of size 5120 to size 13824 and then back to size 5120.
    - Three $13824 * 5120$ matrices with $212M$ parameters: one each for the two layers, and one additional for gated activation.
  - $(104M + 212M) * 40 \approx 12.6B$ parameters for 40 layers.
- Output linear layer: $32000 * 5120 \approx 160M$ parameters
- All together: $160M + 12.6B + 160M \approx 13B$ parameters.