Nvidia DLI NLP Session Notes

November 5, 2023

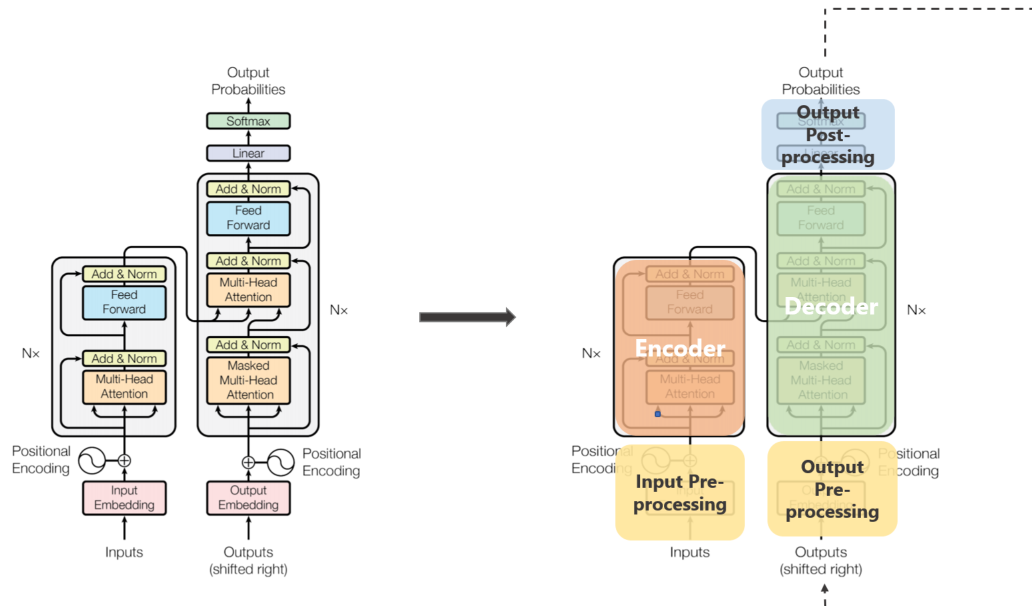

Figure 1: Diagram of the transformer model as well as a high level labelling of the steps, found on google images

Tokenizer:

- Takes text, returns numerical representation made of "tokens" (tokens are elements that represent parts of a word)
- Easy to train a tokenizer for a specific purpose

Embedding:

- Takes tokenized text, returns a different numerical representation that the network can use
- some embeddings will take context and semantic meaning into account, which will cause related words to be closer together

- Embeddings make tokenized representations less sparse (think bag of words with like 50 million possibilites, hella sparse)
- Transformers use simple learned embeddings
  * A matrix of size $|\text{vocab}| \times d_{\text{model}}$
  * Trained with transformer
  * In original transformer implementation, weights for input embedding, output embedding and linear layer were the same
- Vocab size and d_{model} are hyper parameters

Positional Encoding:

- Since there is no recurrent or convolutional units, positional encoding is used to infer order
- Same size as embedding so that they can be summed
- In the original paper, positional encoding was done with a combination of sine and cosine functions

Transformer Encoder:

- Encodes sentence into hidden state vector

Attention:

- Focuses on *important* words

  E.g. My [dog] loves playing fetch with a tennis [ball]
  * *dog* will have low attention to *tennis*
  * *ball* will have high attention to *tennis* and *playing*

Self Attention:

- 3 important components; Query, Key and Value
  * Each has its own weight matrix
  * For each word, Q, K, V are generated by multiplying word representation with its respective weight matrix

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

- How are Q, K, V generated?
  1. Take embedding of word with (potentially randomly initialized) matrix
  2. Let $X = [X_1, \dots, X_n]$ be a matrix where each column is the embedding of words $1, \dots, n$
  3. Set $Q = XW^Q$, $K = XW^K$, $V = XW^V$
  4. Set self attention matrix $Z = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$

Multi-head Attention:

- Same as regular attention, except multiple attention layers at once

Transformer Decoder:

- Input is encoded vector from transformer encoder
- outputs one word at a time

BERT:

- A model that only looks at the encoding part of the transformer model
- only learns a good representation of the text

- loss evaluated with:
  - fill in the blanks, where we remove some percentage of words in a sentence (15% for example) and tell BERT to guess the missing words
  - Self supervised learning, loss does not require self annotated data

BERT Wordpiece Tokenizers:
- Splits a word or token into smaller pieces

E.g Tokenization

Characters: `'t', 'o', 'k', 'e', 'n', 'i', 'z', 'a', 't', 'i', 'o', 'n'`

Words: `tokenization`

Subwords: `"token", "##ization"`

Wordpiece Algorithm:
1. Split word into char tokens
2. Build language model with above tokens
3. Generate new tokens by combining 2 with high liklihood
4. Repeat until desired vocab size is reached

Pretraining:
- On the fly preprocessing
  - Train and validation should have format: `[WORD][SPACE][WORD]...`
- OOV (Out of Vocabulary):
  - Replace OOV with a token like `[UNK]`
  - Split OOV at the character level
  - Tokenize into subwords

NeMo:
- Build around neural models
- Based on pytorch lightning
  - 2 main components:
    1. `LightningModule`
    2. `Trainer`
- Every NeMo module has an example config file and training script

Data Prep:
- Data needs to be in the following format before training:
  `[WORD][SPACE][WORD]...[TAB][LABEL]`
- Header needs to be removed

Optimization and Performance:
- Pytorch JIT / Torch script
- ONNX Runtime
- ONNX Tensor RT
- Tensor RT

Triton Server:
- Supports:

Tensorflow Graph

Tensorflow Saved Model

Caffe 2 Exports

Custom models

Quantization:
- A method to reduce size of an LLM

E.g `[0.34 3.27 5.6]`

- `[0.34, 3.27, 5.6]` $\xrightarrow{\text{quantization}}$ `[64, 134, 217]` $\xrightarrow{\text{dequantization}}$ `[0.41, 3.62, 5.29]`
- Helps with computation

Concurrent Model Execution:
- Allows multiple models to run in paralell
- Triton handles this automatically (shameless plug)

Dynamic Batching:
- No overhead for parameter storage or fetching
- Better GPU utilization

Scheduling Strategies:
- Batches are inferred at each request
- Choice of scheduler or batcher depend on:
  - Stateful or Stateless nature of workload
  - Model is isolated or part of a pipeline

Stateless Inference:
  - Option 1: Distribute request to all instances (preferred when states are known and understood)
  - Option 2: Dynamic Batching

Stateful:
  - State is maintained between inferences
  - Option 1: Direct
  - Option 2: Oldest