EE508 Project Phase - 1

Shaun Almeida
Asmita Mohanty

1. What is language modeling?

Language modeling is a core concept in natural language processing (NLP) that involves creating statistical or computational models capable of understanding and generating human language. At its heart, a language model estimates the probability distribution over sequences of words, which enables it to predict the likelihood of a given sequence or the next word in a sequence.

2. What is self-supervised pre training?

In deep learning, pre-training refers to the process of optimizing a neural network before it is further trained/tuned and applied to the tasks of interest. This approach is based on an assumption that a model pre-trained on one task can be adapted to perform another task. As a result, we do not need to train a deep, complex neural network from scratch on tasks with limited labeled data. Instead, we can make use of tasks where supervision signals are easier to obtain. This reduces the reliance on task-specific labeled data, enabling the development of more general models that are not confined to particular problems.
Self-training is a concept where a model is iteratively improved by learning from the pseudo labels assigned to a dataset. To do this, we need some seed data to build an initial model. This model then generates pseudo labels for unlabeled data, and these pseudo labels are subsequently used to iteratively refine and bootstrap the model itself.
Unlike the standard self-training method, self-supervised pre-training in NLP does not rely on an initial model for annotating the data. Instead, all the supervision signals are created from the text, and the entire model is trained from scratch. A well-known example of this is training sequence models by successively predicting a masked word given its preceding or surrounding words in a text. This enables large-scale self-supervised learning for deep neural networks, leading to the success of pre-training in many understanding, writing, and reasoning tasks.

3. Why is pre training more hardware-efficient for Transformer- or attention-based models compared to RNN-based models?

Transformer-based models and their associated pre-training methods tend to be more hardware-efficient compared to RNN-based models largely because of their architectural design, which permits a high degree of parallelization and better utilization of modern accelerator hardware like GPUs and TPUs. RNNs (including their variants like LSTMs or GRUs) process sequences step by step. This inherent sequential dependency forces each computation step to wait for the previous one to complete, greatly reducing the ability to leverage parallelism. In contrast, transformers use self-attention mechanisms that allow the model to consider all tokens simultaneously. This means that computations for different tokens in a sequence can be executed in parallel, taking full advantage of the parallel processing capabilities of GPUs and TPUs.

Self-Supervised Objectives:

Pre-training tasks like masked language modeling (used in models like BERT) can be applied simultaneously across the entire sequence, enabling transformers to fully exploit hardware parallelism. The self-supervised pre-training objectives in transformer models are designed to work with large amounts of unstructured text where each token can be processed independently (or in parallel) for the mask prediction task.

Reduced Training Time:

Because transformer models can process entire sequences in parallel, the overall training time decreases significantly when compared to RNNs. This is especially beneficial during the pre-training phase, where the volume of data is typically enormous. The ability to efficiently utilize large batches and parallel computation means that transformers scale more effectively to massive datasets.

4. What is the difference between encoder-only and decoder-only models, and why are decoder-only models more popular?

- Encoder-only models:
    - It consists of encoder stack of the transformer & doesn't have a decoder
    - It will output the context vectors i.e contextual representations of the input tokens
    - It typically follows a global self-attention mechanism where the input tokens attend to all the tokens i.e they are bi-directional, having information from all the inputs & processes all tokens simultaneously
    - Gives a deep contextualized embedding

- Decoder-only models:
    - It consists of decoder stack of transformer & doesn't have encoder, therefore there is no context embedding as external input
    - It uses a masked self-attention mechanism, i.e each token at current time step will attend to all the previously generated tokens or all the previous hidden states of the decoder
    - It is an autoregressive generator, generating one token at a time/sequentially & is hence uni-directional
    - They cannot process all tokens simultaneously

- Popularity of decoder-only models:
    - Primary reason is that they can predict/generate tokens without relying on any context or specific task, unlike encoder-only models
    - Best suited for generation/token prediction, has more flexibility & has wide-spread applicability such as in text summarization, translation, content creation, code generation etc, making them more general purpose & better at scalability

5. Suppose the vocabulary consists of only three words: Apple, Banana, and Cherry. During decoder-only pretraining, the model outputs the probability distribution $Pr\theta(\cdot \mid x0,...,xi) = (0.1,0.7,0.2)$. If the correct next word is Cherry, represented by the one-hot vector (0,0,1), what is the value of the log cross-entropy loss? What is the loss value if the correct next word is Banana instead?

When correct next word is Cherry, CEL = 1.609
When correct next word is Banana, CEL = 0.357

6. What are zero-shot learning, few-shot learning, and in-context learning?

- Zero-Shot learning:
    - A prompt-based learning which doesn't require traditional learning. LLM is directly applied on unseen tasks to make predictions & the prompts are repetitively adjusted to improve the predictions.
    - No task demonstration is given. LLM just receives the prompt or instructions & then makes predictions. It implicitly relies on how powerful the pre-training is so that it can make better predictions.
- Few-Shot learning:

    - A prompt-based learning where the prompt contains sufficient demonstrations (relatively small). LLM learns from this newly added experience & then makes predictions.
    - If LLM pre-training is powerful already, few-shot learning can enable it to address complex problems & make predictions with just a few demonstrations.
- In-context learning:
    - A prompt based learning where the LLM is provided a "context" that gives a description of a specific information/task before instructing it to perform it. This "context" serves as a premise which is provided along with a demonstration to enable the model to learn better.
    - This is efficient during inference, since the LLM doesn't need to do pre-training, it reorganizes the knowledge learned from the pre-trained model without fine-tuning or requiring additional training

7. Why is fine-tuning necessary? What is instruction fine-tuning?

Pre-trained models are generally trained on broad and diverse datasets, which allows them to learn a wide array of linguistic patterns and representations. However, these models are not necessarily specialized for any single downstream task (e.g., sentiment analysis, machine translation, or summarization). Fine-tuning adjusts the model's parameters on a task-specific dataset so that its predictions are better aligned with the requirements of that task. Even though a pre-trained model has learned a lot about language, direct application on a task without further tuning may result in suboptimal performance. Fine-tuning refines the model weights to minimize errors (using loss functions like cross entropy) on the specific task, leading to improvements in accuracy, relevance, and overall performance.

Instruction fine-tuning: Unlike traditional fine-tuning, which is aimed at improving performance on a specific task, instruction fine-tuning trains the model to follow directives given in natural language. The training data consists of prompts (or instructions) paired with desired outputs or responses. This helps the model understand what the user is asking for and generate responses that align with the specified instructions. Instruction fine-tuned models are often exposed to a variety of prompts during training. This exposure helps the model generalize better, allowing it to handle a

range of tasks and queries effectively without having to be re-trained for each specific task.

8. What is tokenization? What is a word embedding layer?

Tokenization:
It's a foundational NLP process of converting unstructured data into structured data or smaller units called “tokens” that can be processed by models. The token can be a token of words, characters, subwords, sentences or morphological terms, which are broken down from a text. Tokenization refers to the feature extraction from raw text that helps in downstream computational algorithms for classification, summarization, translation etc.

Word embedding layer:
Word embedding refers to a dense vector representation of a given word. It is a collection of real numbers, usually in a continuous, lower dimension space. Word embedding layer refers to the neural network that is an embedding matrix, that converts the input token to a vector, which the neural network can understand. This matrix is learnable, i.e, its weights get updated during training, which, in turn produces more meaningful vectors that the neural network can understand.

9. What is position embedding? What kind of position embedding method is used in Llama models?

Since the transformer model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension as the embeddings, so that the two can be summed.

LLaMA models use rotary positional embeddings (RoPE). Instead of adding a fixed or learned positional vector to each token embedding, RoPE applies a rotational transformation to the token embeddings. This rotation encodes the relative positions of tokens, which means that the attention mechanism can naturally compute relative position relationships.

10. What is the difference between multi-head attention and grouped-query attention? Which type of attention mechanism is used in Llama models?

Multi-head attention (MHA):

- The input X is projected into query Q, key K to retrieve value V across multiple heads, i.e there are multiple projections or embeddings of (Q,K,V) which is unique for each head. Each head will have a unique query which is mapped to a unique (K,V) pair. No two queries can share the same KV space.
  $Q_1 \rightarrow K_1, V_1$
  $Q_2 \rightarrow K_2, V_2$
- The computation of each attention head is done in parallel since the heads are independent of each other & later the outputs are concatenated with each other & linearly transformed to form a larger attention head which contains rich combinations from each head. Therefore, there is representational diversity among different heads.
- It is much faster because the weight projection matrix for each head has a lower dimension compared to the full dimension in single-head attention, thereby making computation faster.

Grouped-Query attention (GQA):

- Multiple queries are grouped together per attention head, with each group sharing the same K,V pairs. Instead of a single query per head, now there will be a set of queries per head mapped to a certain (K, V) pair.
  Group1: $[Q_1, Q_2, Q_3] \rightarrow [K_1, V_1]$
  Group2: $[Q_4, Q_5, Q_6] \rightarrow [K_2, V_2]$
- The number of attention heads will reduce since queries are grouped together; however, as queries in a group share the same (K,V) pair, the head outputs of the queries in a group are not independent anymore & may have some correlation. This results in less/no representational diversity across multiple heads.
- Since the number of attention heads are now reduced, computationally GQA is more efficient than MHA with a slight trade-off in performance due to lesser independence in the outputs. It is also more memory efficient as it requires lesser memory due to lesser heads or KV projections.

Attention Mechanism in Llama: Llama uses Grouped-Query Attention (GQA) because llama models are large with billions of parameters & will require significant memory & compute needs. Using GQA will require lesser KV projection with fewer attention heads that helps in faster inference, lesser compute & lesser memory needs. This allows large models to operate & scale efficiently without compromise on quality.

11. What kind of activation function is used in Llama models?

LLaMA models use SwiGLU activation function, SwiGLU is an activation function designed to enhance the expressiveness and training dynamics of feed-forward layers—especially in large transformer architectures. It's a variant of the traditional Gated Linear Unit (GLU) that combines gating mechanisms with a "Swish" non-linearity.

In a typical GLU, the input is split into two halves. One half (often denoted x_a) is processed linearly, and the other half (x_b) is passed through a gating function—commonly a sigmoid—to control the information flow. The output is computed as:

$$GLU(x) = x_a + \sigma(x)_b$$

SwiGLU replaces the sigmoid in the gating component with the Swish activation function. Swish is defined as:

$$swish(x) = x \times \sigma(x)$$
$$SwiGLU(x) = x_a \times swish(x_b)$$

Here, the swish multiplication combines a linear transformation with a non-linear, smooth gating signal. The Swish function is known for maintaining a balance between non-linearity and gradient flow, which can lead to better performance and training stability than using a simple sigmoid.

12. What is layer normalization? What is the difference between layer normalization and batch normalization?

Layer Normalization:

- Normalizes the output across the feature dimension within each token/sample & not batch; applied to the output of every sub-layer (after multi-head & after feed-forward)

- Standardization ensures the gradients don't depend too much on the scales of the parameters (weights, activations)
- Results in faster convergence & training, more stability
- Primarily used in sequence models like transformers

Difference between Layer & Batch Normalization:

- In Batch Normalization the output features across the tokens/samples in a batch. It is primarily used in CNNs
- Layer normalization doesn't rely on batch dimension unlike batch normalization.
- Transformers that use the seq2seq mechanism, they use inputs of varying sequence lengths which are padded as "padding" tokens to have the same length for all tokens. But batch norm cannot compute mean or variance statistics for padded tokens & hence cannot normalize these varying lengths of sequences, therefore, its not used in transformers.
- Batch norm can break token independence since it normalizes across all tokens in a batch whereas layer norm is local to the token & normalizes it independently from other tokens.
- Batch norm cannot follow the auto-regressive mechanism because during inference batch size is 1 since only 1 token is processed, so it breaks as there is no other token to normalize with, which does not happen in layer norm as they normalize one token only independent of others.

13. What is the auto-regressive generation process, and how is a decoding strategy used during text generation?

Auto-regressive generation:

- In the auto-regressive generation process, the model generates one token/output at a time by taking previously generated outputs/tokens as inputs. Once a token is generated, it appends to the input sequence & then generates the next token; this process continues till it hits the end of sequence length.
- The self-attention mechanism of transformers enables the tokens to attend to all the tokens. However, in the decoders, masked or causal self-attention is used such that the decoder doesn't see the future tokens & focuses only on the past tokens.

Decoding process:

- The decoding strategy used in text generation is that the decoder will generate a probability distribution for the next token over a fixed vocabulary given the predictions or contexts of previously generated tokens.
- The choice of selecting the next token follows separate strategies such as choosing based on:
    - Highest Probable token: Deterministic, Greedy approach
    - Top-K most probable tokens: Stochastic approach; to add more diversity in the output
    - Tokens with highest cumulative probability: Deterministic, Beam search approach