Phase 1

Q1. What is language modelling?

- **Language Modelling** is the task of predicting the next token in a sequence by generating a probability distribution over the vocabulary, given the preceding tokens. This is commonly formulated as maximizing the likelihood of the sequence using the chain rule of probability.
- It is foundational in training generative models like GPT and is a core example of sequence generation models.

Q2. What is self-supervised pretraining?

- In **Self-Supervised Pretraining**, a neural network is trained using supervision signals generated automatically from unlabeled data, without human annotation. The model creates pseudo-labels and learns from them.
- For example, in masked language modeling or next-token prediction, the model learns to predict masked or next words using surrounding context. This approach allows large-scale training from raw text and has become the dominant method for pretraining LLMs.

Q3. Why is pretraining more hardware-efficient for Transformer- or attention-based models compared to RNN-based models?

- **Transformers**, especially their attention layers, enable significant parallelism because each token can attend to others simultaneously within a sequence, and operations like multi-head attention are inherently parallelizable.
- **RNNs**, in contrast, are inherently sequential — each token depends on the previous one — which prevents effective parallelization.
- This makes Transformers more hardware-efficient during both training and inference.

Q4. What is the difference between encoder-only and decoder-only models, and why are decoder-only models more popular?

- **Encoder-only models** (e.g., BERT) are designed for sequence encoding tasks. They take the entire input sequence and often use masking during pretraining (like masked language modeling) to learn rich contextual representations. These models are typically used for classification or understanding tasks and are usually part of larger systems.
- **Decoder-only models** (e.g., GPT) fall under sequence generation. They generate text one token at a time based on previously generated tokens and are trained via next-token prediction. They are popular because their architecture naturally supports autoregressive generation, which is well-suited for a wide range of tasks through prompting or instruction tuning, including question answering, summarization, and dialogue generation.

Q5. Suppose the vocabulary consists of only three words: Apple, Banana, and Cherry. During decoder-only pretraining, the model outputs the probability distribution Prθ(· | 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?

- For Cherry, (I have used natural log here)

Loss = - (0 * log(0.1) + 0 * log(0.7) + 1 * log(0.2)) = 1.6094

- For Banana,
  Loss = - (0 * log(0.1) + 1 * log(0.7) + 0 * log(0.2)) = 0.3567

Q6. What are zero-shot learning, few-shot learning, and in-context learning?

- **Zero-shot learning** refers to the ability of a model to perform a task without having seen any explicit examples during training or prompting. It relies on the model's general understanding learned during pretraining and is usually achieved via task instructions embedded in the prompt.
- **Few-shot learning** involves providing a few examples of input-output pairs (called demonstrations) in the prompt before the actual input. This allows the model to generalize the pattern and perform the task better.
- **In-context learning** is the broader capability behind both zero-shot and few-shot learning. It refers to the model's ability to learn how to perform a task from the context provided in the prompt (instructions, examples, etc.), without any weight updates or fine-tuning. This is a key emergent property of large decoder-only models like GPT-3 and beyond

Q7. Why is fine-tuning necessary? What is instruction fine-tuning?

- **Fine-tuning** is the process of training a model further on a smaller dataset (than the pretraining dataset) to adapt the model to a specific task (eg: question answering). It is necessary as the model after pre-training is trained on a very large unlabelled dataset and cannot provide good predictions without knowing relevant "context".
- In classical NLP, models are used as components of other systems. But, LLMs being generative models are complete end-to-end NLP systems by themselves. Their generative nature turns every NLP task into a text generation task, thereby solving the desired overall NLP task.
- Thus, the finetuning of LLMs consists of a mechanism to "activate" the learned knowledge during the pre-training phase for it to generate relevant text. This type of finetuning is very different than finetuning classical NLP models (like BERT) where training is done on a smaller labelled dataset.
- So, the process of activating the learned knowledge during pre-training by fine-tuning the pretrained model parameters using instruction-following data is called **Instruction Fine-tuning**. This is done by defining a loss function for a specific sample (y_s, x_s), the best parameters for that sample are found, and the error gradients are back-propagated only for the pre-trained network parts corresponding to y_s – while leaving the rest of the model network un-changed.

Q8. What is tokenization? What is a word embedding layer?

- **Tokenization** is the process of separating text into basic units called tokens, given a character sequence and defined document unit. Tokens are instances of a sequence of characters in some particular document that are grouped together as a useful semantic unit for processing.
  Eg: Text: 'Friends, Romans, Countrymen, lend me your ears'
  Tokens: 'Friends', 'Romans', 'Countrymen', 'lend', 'me', 'your', 'ears'

- **Word Embeddings** are distributed representation of words, where they are represented as low-dimensional real-valued vectors (instead of discrete variables). This makes it possible to compute the meanings of words in a continuous representation space, as they are dense vectors capturing semantic meaning and relationships. **Word Embedding Layers** are the layers in a network that maps the one-hot encoded representation of a word to a dense embedding vector in continuous space, while learning this embedding process.
  Eg: Country-Capital: Canada-Ottawa

Q9. What is position embedding? What kind of position embedding method is used in Llama models?

- Word Embeddings by themselves provide semantic meaning but lack order awareness **Position Embedding** adds positional information to a Word Embedding to form the full Embedding.
  Eg: 'The dog chased the cat'
  Without a Position Embedding, each word is embedded independently, and the model can't tell if the dog chased the cat or the other way around. With a Position Embedding, the order matters now, and the two things in the following aren't the same:
  'dog chased the cat' != 'cat chased the dog'
  The order matters now.
- Position Embeddings are trainable/learnable, just like the Word Embeddings. Position Embeddings are learned representations. In contrast, Position Encodings are deterministic, are derived from a formula. Either can be used to add positional information to a Word Embedding.
- Llama model use **Rotary Positional Encoding (RoPE)**, which is deterministic and not learnable. RoPE integrates explicit relative position into the self-attention mechanism of Llama models. It allows the models to understand how the tokens are related to each other, dynamically.
- Instead of 'adding' positional vectors to a word embedding, it 'rotates' Q and K vectors based on their position after projection and just before the attention dot-product.

Q10. What is the difference between multi-head attention and grouped-query attention? Which type of attention mechanism is used in Llama models?

- In Multi-head Attention (MHA), every head has a Q, K and V matrix. In contrast, **Grouped-Query Attention (GQA)** has a number of K and V groups. The Q matrices are distributed amongst these KV groups.
  Eg: Considering 12 heads, for MHA: there are 12 Q, K, V-matrices for each head
  while, for GQA: if we set the number of KV groups to 4, then there are 3 Q-matrices for each KV group.
  If number of KV groups = number of heads in MHA, then only 1 Q-matrix exists for each KV group, which turns the GQA into MHA.
- MHA provides the best performance, but is memory-intensive for inference. In contrast, GQA performs slightly worse, but provides large drop in KV-cache size (as there are fewer K, V-matrices in total).

- GQA is sort of a middle ground between MHA and Multi-Query Attention (MQA). MQA allocates all Q matrices to just 1 K, V-matrices. MQA provides the largest drop in KV cache size, but performs the poorest too.
- Some Llama use GQA attention mechanism, while others (smaller parameter variants) use MHA
  Eg of Llama variants using GQA:
  1. Llama 2 34B, number of KV groups = 8, Q matrices = 64
  2. Llama 2 70B, number of KV groups = 8, Q matrices = 64
  3. Llama 3 8B, number of KV groups = 8, Q matrices = 32
  4. Llama 3 70B, number of KV groups = 8, Q matrices = 64

  Eg of Llama variants using MHA:

  1. Llama 1 7B, 12B, 30B, 65B
  2. Llama 2 7B, 13 B

Q11. What kind of activation function is used in Llama models?

- Llama models use the **SwiGLU activation function**: Swish-Gated Linear Unit
- Defined as:
  y = W_3 @ [element-wise_product(SiLU(x @ W_1, x @ W_2))]
  …where, SiLU(): Sigmoid Linear Unit activation (element-wise sigmoid)
  W_q, W_2 and W_3: learnable weights
  x: input and y: output

Q12. What is layer normalization? What is the difference between layer normalization and batch normalization?

- **Layer Normalization** is the re-scaling of a set of feature activations (outputs) within each sample. The mean and variance across the feature dimension of that sample is calculated, followed by applying learned scale Gamma and shift Beta.
- **Batch Normalization** is the re-scaling of each feature (channel) across each batch (mini-batch) of data. For every feature, the mean and variance over all samples in that batch is calculated, followed by applying learned scale Gamma and shift Beta.
- **Differences**:
  1. Both normalization methods differ in terms of what dimension they normalize over.
  2. While Batch Normalization is usually used for CNNs and fully-connected networks, Layer Normalization is used in RNNs, Transformers and other sequence models where batch sizes vary greatly.
  3. For Batch Normalization, batch statistics are used during training for normalization and running averages during inference. While for Layer Normalization, the same normalization computation is done during training as well as inference.

Q13. What is the auto-regressive generation process, and how is a decoding strategy used during text generation?

- The **Autoregressive Generation** process is a process of the decoder, where given an input, the model predicts the next token, and then the next prediction is made by taking the combined input in the next step.
- This takes place for the decoder half in encoder-decoder models, and for the decoder itself for decoder-only models. A causal mask is applied for both model groups. If considering encoder-decoder models (eg: Transformer), each prediction step still involves a full pass through the Transformer layers.
  Eg:
  1. Input: The cat sat -> Prediction: on
  2. Input: The cat sat on -> Prediction: the
  3. Input: The cat sat on the -> Prediction: mat
- Need for a **Decoding Strategy**: At each step to generate the next token, the output softmax provides a probability distribution over all tokens. Right after this, we need to decide how to select the next token given this probability distribution over all tokens, while knowing all previous tokens in the sequence. Hence, a decoding strategy is required.
- Common decoding strategies are:
  - Greedy
  - Top-k / Top-p sampling
  - Beam Search
  - Contrastive / Speculative decoding
- Two examples for these common decoding strategies:-
  Top-k Sampling:
  - The top 'k' tokens are kept, their probabilities are re-normalized (so they sum to 1) and a sample is chosen from this clipped renormalized distribution.
  - A common choice for 'k' is 50 (usually choose an integer between 1 to 100)
  - It is a relatively trivial decoding strategy, while drawbacks being that 'k' size is fixed, so its not adaptive based on number of tokens.
  - Eg: for k=3, following probability is obtained after output softmax,
    "cat": 0.4, "dog": 0.3, "bird": 0.2, "elephant": 0.1
    Here, we keep "cat", "dog" and "bird"
    Re-normalizing, "cat": 0.444, "dog": 0.333, "bird": 0.222
    Now, choose a sample among these 3 tokens. This is a weighted distribution, so it is more likely that we select "cat".

  Greedy:

  - The top-1 token, i.e. the token with the highest probability is always selected.
  - No sampling is done, simply the top-1 or the most probable token is selected
  - Eg: output softmax gives the following probability distribution,
    "cat": 0.4, "dog": 0.3, "bird": 0.2, "elephant": 0.1
    Simply choose the highest, i.e "cat".