Can you explain how BERT constructs the embedding for each token?

Each token's embedding in BERT is composed of three components added together:

Token Embedding – Represents the meaning of the token and can be obtained from pre-trained models such as Word2Vec or initialized randomly.

Position Embedding – Since BERT does not inherently capture word order like LSTMs, position embeddings are used to encode the sequential order of tokens in a sentence.

Segment Embedding – Indicates whether a token belongs to the first or second segment of an input sequence. This is primarily used for the Next Sentence Prediction (NSP) task, helping the model determine whether two segments come from the same context.

Can you compare different normalization techniques in the context of LLMs?

Feature Normalization normalizes specific features across the entire dataset to reduce scale differences and improve model stability.

Batch Normalization (BN) operates within each neural network layer, normalizing the same feature dimension across a batch of samples. It is particularly effective in computer vision tasks where the relative differences between samples (e.g., images) are important, as it preserves inter-sample feature relationships within the same channel.

Layer Normalization (LN) normalizes all feature dimensions within a single sample (such as a token) independently of other samples. LN is well-suited for LLMs and Transformer architectures because NLP tasks focus more on the relationships among different feature dimensions within a token rather than between different tokens or samples. The relative values of these dimensions determine the token's semantic representation in the vector space.

What is the purpose of normalization in deep learning?

Mitigates internal covariate shift

Mitigates gradient exploding/vanishing problems

Reduces sensitivity to parameter initialization

Mitigates the influence of outliers, helps decrease the risk of overfitting.

What are the different placements of Layer Normalization in Transformer architectures?

There are three common placements for Layer Normalization in Transformer architectures: Post-LN, Pre-LN, and Sandwich-LN.

In Post-LN setup, Layer Normalization is applied after the residual addition, the sequence is *Attention* → *Add* → *LN* and *Feedforward Network (FFN)* → *Add* → *LN*. The main drawback of this approach is that, in deep models, the gradient norm tends to increase as the network depth grows, which can lead to training instability.

The Pre-LN setup, Layer Normalization is applied before the attention and feedforward layers, the sequence is *LN* → *Attention* → *Add* and *LN* → *FFN* → *Add*. Compared to Post-LN, Pre-LN helps maintain stable gradient norms even in very deep networks, making training more stable and mitigating issues related to gradient explosion or vanishing. However, a slight downside is that models using Pre-LN may achieve slightly lower performance compared to Post-LN in some cases.

The Sandwich-LN approach applies Layer Normalization multiple times within a block, typically following a pattern like *LN* → *Attention* → *LN* → *Add* → *LN* → *FFN* → *LN* → *Add*. This method helps prevent issues like gradient explosion, but it can introduce training instability and, in some cases, may even cause training to collapse.

What is the purpose of residual connections in neural networks?

Mitigates gradient vanishing and exploding problems

Facilitates learning of identity mappings

Enhances model robustness: even if some neurons fail, the information can be maintained.

What are the limitations of Vanilla Attention?

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

High computational complexity $O(N^2)$

Large memory consumption due to storing the full attention matrix

Numerical instability from softmax operations on large values

What are the common variants of Attention mechanisms, and how do they address the limitations of Vanilla Attention?

Sparse Attention allows each query to attend to only a subset of keys instead of all tokens. The selection of keys is predefined, using patterns like sliding windows or learnable sparse masks. It is suitable for tasks with local dependencies, such as time-series forecasting.

Linear Attention approximates softmax attention using kernel functions and reorders matrix multiplications to reduce computational complexity to $O(N)$. However, this approximation can lead to a loss of expressiveness.

Low-Rank Self-Attention reduces both time and space complexity by factorizing the attention matrix into two lower-dimensional subspaces. It is effective for data with low intrinsic dimensionality, such as recommendation systems.

Flash Attention doesn't change the attention formula. Instead, it improves efficiency through mathematical optimizations like tiling (blocking) and optimized softmax calculations, combined with engineering techniques such as kernel fusion and leveraging GPU's fast shared memory. This results in significant speedup (up to 2–4x faster) with lower memory usage but requires GPUs with specific CUDA optimizations.

What are common tokenization approaches?

Byte Pair Encoding (BPE) constructs a subword vocabulary by repeatedly merging the most frequent pairs of characters or subwords in the text. It is simple and efficient, with strong capability to handle OOV (out-of-vocabulary) words. However, it is biased towards high-frequency words and less flexible for low-frequency words. BPE is suitable for space-separated languages like English and is commonly used in GPT-2 and early Transformer models.

WordPiece Encoding builds a subword vocabulary by selecting subword combinations that maximize the likelihood of the training data. It has strong generalization ability and performs well in modeling low-frequency words, but comes with higher training complexity. WordPiece is widely used in pre-trained models like BERT and RoBERTa.

SentencePiece Encoding treats text as a continuous sequence of characters and performs unsupervised subword learning directly on raw text, making it suitable for any language. It works well for languages without spaces (e.g., Chinese, Japanese) but has slower training speed, and the output tokens may be less intuitive, often containing special symbols. SentencePiece is commonly used in multilingual models like T5 and mBART.

Can you explain different types of positional encoding?

Absolute Positional Encoding assigns a unique position vector to each token based on its absolute position. It's simple and easy to implement, commonly used in early models like BERT. However, it doesn't generalize well to longer sequences beyond the training range and lacks the ability to capture relative token relationships.

Relative Positional Encoding focuses on the distance between tokens, modeling how tokens relate to each other instead of their fixed positions. This improves generalization, especially for variable-length sequences, and is effective in tasks like machine translation (used in Transformer-XL, T5). The downside is added computational complexity due to pairwise position calculations in attention.

Rotary Positional Encoding (RoPE) encodes positions by rotating token embeddings based on their positions. It implicitly captures both absolute and relative positions, making it highly effective for long-context models like LLaMA and GPT-NeoX. RoPE generalizes well to longer sequences but involves more complex mathematical operations.

What's your understanding of different Transformer-based language model architectures?

Encoder-only models (e.g., BERT, RoBERTa, DeBERTa, ALBERT) include only the Transformer encoder. They use bidirectional self-attention to capture complex semantic relationships, excelling in tasks like text classification. However, they are not suitable for text generation due to the absence of a decoder.

Decoder-only models (e.g., GPT series, BLOOM) include only the decoder and use an autoregressive mechanism to predict the next token, making them ideal for text generation. However, their reliance on unidirectional context limits overall semantic understanding.

Encoder-Decoder (Seq2Seq) models (e.g., T5, BART, mBART, original Transformer) combine an encoder and a decoder, making them suitable for sequence-to-sequence tasks like machine translation, summarization, and grammar correction, though they have higher computational costs.

How have models improved upon BERT?

Several models have improved BERT by modifying pretraining tasks and architecture.

For pretraining tasks, RoBERTa enhances robustness by removing the Next Sentence Prediction task and using dynamic masking, ELECTRA improves sample efficiency with Replaced Token Detection task. XLM extends BERT's multilingual capabilities through Translation Language Modeling task.

For architecture, ALBERT increases efficiency with parameter sharing and embedding factorization, DeBERTa improves semantic understanding with disentangled attention and enhanced relative position encoding.

Can you explain the different types of Parameter-Efficient Fine-Tuning (PEFT) methods?

Parameter-Efficient Fine-Tuning (PEFT) methods can be categorized into three types: adding extra parameters, updating a subset of parameters, and reparameterization-based approaches.

The first category, adding extra parameters, introduces small tunable components while keeping the base model mostly frozen. Adapters insert feed-forward networks into transformer layers, allowing strong task-specific adaptation but requiring architectural modifications. Prompt Tuning optimizes soft prompt embeddings prepended to the input, making it highly efficient for large-scale models, though it may struggle with complex tasks. Prefix Tuning modifies self-attention states with additional prefix tokens, offering more flexibility while remaining parameter-efficient, especially for generative models.

The second category, updating only a subset of parameters, fine-tunes select pre-trained weights instead of adding new layers. BitFit, which updates only bias terms, is extremely lightweight with minimal compute cost but lacks the adaptability of methods that introduce new tunable components.

The third category, reparameterization-based approaches, optimizes how weight updates are stored to reduce memory and compute costs. LoRA replaces full-rank weight updates with smaller low-rank matrices, making fine-tuning more efficient. AdaLoRA improves this by dynamically adjusting rank allocation based on task importance. QLoRA further reduces memory usage by combining LoRA with 4-bit quantization, enabling fine-tuning on resource-constrained hardware.

Overall, adding extra parameters provides strong generalization but requires modifications, subset parameter tuning is extremely lightweight but limited in flexibility, and reparameterization-based methods strike a balance between efficiency and performance.

What kind of prompt engineering you did?

I used Chain of Thought (CoT) prompting to guide the model step-by-step: First, extract key behavioral attributes. Then, determine the risk or intent. Finally, identify underlying themes.

I also considered Few-Shot Learning, providing labeled examples to improve pattern recognition, and Self-Consistency, prompting the model to apply different reasoning approaches—such as impact-based vs. intent-based analysis—and consolidate the results.

While Few-Shot Learning enhanced generalization and Self-Consistency reduced variance, CoT was the most effective.

How do you prevent overfitting when training a model?

To prevent overfitting during training, I use weight decay, dropout, and early stopping.

Weight decay (L2 regularization) helps prevent overfitting by adding a penalty term to large weights. I implement this using the AdamW optimizer with weight_decay=0.01.

Dropout randomly deactivates neurons, reducing over-reliance on specific features. I typically set hidden_dropout_prob and attention_probs_dropout_prob to 0.1.

Early stopping prevents memorizing noise by monitoring validation loss and stopping training if it stagnates for 3–5 epochs.

Imbalanced Data

One-Class Learning trains a model solely on normal or anomalous samples, capturing their distribution to detect deviations.

Resampling balances the dataset through undersampling (reducing normal samples) or oversampling (increasing anomalies via duplication or synthetic data).

Loss function optimization improves detection by assigning higher weights to anomalies, forcing the model to focus on harder-to-classify cases.

Ensemble learning also helps—boosting increases the weight of misclassified anomalies in each iteration, while bagging can be combined with undersampling to improve representation.

`Finally, unsupervised methods identify outliers that don’t fit normal clusters, reducing the impact of imbalance and improving anomaly detection.