Transformer & LLM -Interview-Questions-Answers

1. What is a Transformer?

→ Deep learning architecture using attention mechanisms; processes sequences in parallel, replacing RNNs/LSTMs.

2. What is Self-Attention?

→ Mechanism to weigh importance of each token relative to others in the sequence.

3. Single-Head Attention vs Multi-Head Attention?

→ Single-head → one attention score; Multi-head → multiple attention heads capture diverse relationships.

4. What is Positional Encoding?

→ Adds token position information since Transformer has no recurrence; sinusoidal or learned embeddings.

5. Add & Norm in Transformer

→ Residual connection (`Add`) + LayerNorm to stabilize and accelerate training.

6. Feed Forward Network in Transformer

→ Two linear layers with ReLU in between; applies non-linearity after attention.

7. Encoder in Transformer

→ Processes input sequence; stack of attention + feedforward layers; outputs hidden states.

8. Decoder in Transformer

→ Generates output sequence; includes masked self-attention, encoder-decoder attention, feedforward.

9. Masked Attention in Decoder

→ Prevents positions from attending to future tokens during training.

10. Encoder-Decoder Attention

→ Decoder attends to encoder outputs for context-aware generation.

11. Tokenizer in LLM

→ Converts text to tokens; can be subword (BPE), wordpiece, or character-level.

12. Word2Vec Embeddings

→ Maps words to dense vectors capturing semantic similarity; CBOW or Skip-gram.

13. Embeddings in Transformers

→ Combine token embeddings + positional embeddings as input to encoder/decoder.

14. Generation in LLM

→ Predict next tokens sequentially using sampling or beam search.

15. Quantization in LLM

→ Reduces precision (FP32 → FP16/int8) to save memory and accelerate inference.

16. Sampling Techniques: Top-k, Top-p, Temperature

→ Controls randomness in generation:

- Top-k → choose from top k logits
- Top-p → choose tokens with cumulative probability p
- Temperature → scales logits for diversity

17. Fine-Tuning LLM

→ Adapt pretrained model on task-specific dataset; can use LoRA or QLoRA for efficiency.

18. LoRA (Low-Rank Adaptation)

→ Injects trainable low-rank matrices into attention weights; reduces training cost.

19. QLoRA

→ Quantized LoRA; combines 4-bit quantization + LoRA for memory-efficient fine-tuning.

20. Deployment of LLM

→ Convert model to inference format; use CPU/GPU; optionally quantize for resource efficiency.

21. Encoder-only model

→ e.g., BERT; used for classification, embedding extraction; no autoregressive generation.

22. Decoder-only model

→ e.g., GPT; used for text generation; autoregressive.

23. Encoder-Decoder model

→ e.g., T5; used for translation, summarization; encoder encodes input, decoder generates output.

24. Difference between autoregressive and autoencoding models

→ Autoregressive → predict next token (GPT); Autoencoding → reconstruct masked input (BERT).

25. Attention formula

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

26. Multi-Head Attention formula

→ Concatenate multiple single-head attention outputs → linear projection.

27. Layer Normalization vs Batch Normalization

→ LayerNorm → normalizes across features; used in Transformers; BN not suitable for variable sequence lengths.

28. Generation decoding strategies

→ Greedy, Beam Search, Top-k/top-p sampling, temperature-controlled.

29. Context window in LLM

→ Maximum sequence length; model cannot attend beyond this window.

30. Difference between embeddings and positional embeddings

→ Token embeddings → semantic info; positional embeddings → token order info.

31. Basic Transformer Encoder in PyTorch

```
import torch.nn as nn
encoder_layer = nn.TransformerEncoderLayer(d_model=512, nhead=8)
transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers
=6)
```

32. Transformer Decoder in PyTorch

```
decoder_layer = nn.TransformerDecoderLayer(d_model=512, nhead=8)
transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers
=6)
```

33. Positional Encoding

```
import torch
import math
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0,max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0,d_model,2)*(-math.log(100
00.0)/d_model))
        pe[:,0::2] = torch.sin(position*div_term)
        pe[:,1::2] = torch.cos(position*div_term)
        self.pe = pe.unsqueeze(0)
    def forward(self,x):
        return x + self.pe[:,:x.size(1)]
```

34. Tokenizer example (HuggingFace)

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('gpt2')
tokens = tokenizer("Hello world", return_tensors="pt")
```

35. Embedding layer

```
embedding = nn.Embedding(num_embeddings=10000, embedding_dim=512)
x_embed = embedding(tokens['input_ids'])
```

36. Multi-Head Attention in PyTorch

```
mha = nn.MultiheadAttention(embed_dim=512, num_heads=8)
output, attn = mha(x_embed, x_embed, x_embed)
```

37. Add & Norm

```
import torch.nn.functional as F
residual = x_embed + output
normed = F.layer_norm(residual, residual.size()[1:])
```

38. Feed Forward Layer

```
ff = nn.Sequential(nn.Linear(512,2048), nn.ReLU(), nn.Linear(2048,51
2))
x_out = ff(normed)
```

39. Simple Encoder-only model

```
from transformers import BertModel
encoder = BertModel.from_pretrained('bert-base-uncased')
outputs = encoder(**tokens)
```

40. Simple Decoder-only model

```
from transformers import GPT2LMHeadModel
decoder = GPT2LMHeadModel.from_pretrained('gpt2')
outputs = decoder(**tokens)
```

41. Encoder-Decoder model (T5)

```
from transformers import T5ForConditionalGeneration
model = T5ForConditionalGeneration.from_pretrained('t5-small')
```

42. Generate text (GPT2)

```
output = decoder.generate(**tokens, max_length=50, do_sample=True, to
p_k=50, temperature=0.7)
```

43. Fine-tune with LoRA (HuggingFace PEFT)

```
from peft import LoraConfig, get_peft_model
config = LoraConfig(r=8, lora_alpha=32, target_modules=["q_proj","v_p
roj"])
lora_model = get_peft_model(decoder, config)
```

44. Quantize model to 8-bit

```
import torch.quantization
quantized_model = torch.quantization.quantize_dynamic(decoder, {nn.Li
near}, dtype=torch.qint8)
```

45. Apply Top-k / Top-p decoding

```
output = decoder.generate(**tokens, do_sample=True, top_k=40, top_p=
0.9, temperature=0.8)
```

46. Calculate total parameters

```
sum(p.numel() for p in decoder.parameters() if p.requires_grad)
```

47. Convert tokens to text

```
generated_text = tokenizer.decode(output[0], skip_special_tokens=Tru
e)
```

48. Masked attention example

```
seq_len = tokens['input_ids'].size(1)
mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1).bool()
```

```
LoraConfig(r=8, lora_alpha=32, target_modules=["q_proj","v_proj"])
```

1️⃣ `r=8` (Rank of LoRA)

- **What it is:** The rank of the low-rank decomposition matrices.
- **Explanation:** LoRA approximates a large weight matrix (W) as (W + \Delta W), where (\Delta W = A \cdot B) with (A \in \mathbb{R}^{d \times r}) and (B \in \mathbb{R}^{r \times d}).
- **Effect:** Lower `r` → fewer trainable parameters, less expressive adaptation. Higher `r` → more expressive but more memory usage.

2️⃣ `lora_alpha=32` (Scaling Factor)

- **What it is:** A scaling factor applied to the LoRA updates.
- **Explanation:** The update is scaled as: [W' = W + \frac{\alpha}{r} A B] Here, `lora_alpha` = α. This balances the magnitude of LoRA updates relative to the pretrained weights.

3️⃣ `target_modules=["q_proj","v_proj"]`

- **What it is:** Specifies which layers/weight matrices LoRA should modify.
- **Explanation:**
  - `"q_proj"` → query projection matrix in attention layers.
  - `"v_proj"` → value projection matrix in attention layers.
- **Effect:** LoRA only trains these selected modules while keeping all other weights frozen, drastically reducing the number of trainable parameters.

✅ Summary

Parameter	Purpose
`r`	Low-rank dimension of trainable LoRA matrices
`lora_alpha`	Scaling factor applied to LoRA updates
`target_modules`	List of layer names to apply LoRA (e.g., attention q/v)

In [ ]:

```

```