# Transformers : Deep Intuition

**What problem are Transformers solving? Why should we care?**

Let’s go back in time a bit.

Imagine you’re typing into Google Translate:

> “How are you?”

Behind the scenes, a machine has to understand not only the meaning of each word, but how they relate **together**. This isn’t just about understanding words — it’s about understanding **sequences** of them.

Historically, we’ve had tools for this:

- **RNNs** (Recurrent Neural Networks): Think of reading one word at a time, remembering the last word.
- **LSTMs**: An improved version with better memory (like sticky notes across a timeline).
- **GRUs**: A lighter variant.

But even with those, we struggled:

- Long-term memory fades (you forget the start of a sentence by the time you’re at the end).
- They’re slow: sequential by nature. You can’t parallelize them well.

Enter: **Transformers (Vaswani et al., 2017)**

*“Attention is All You Need”*

Transformers let us:

- Process **all tokens at once** (parallelized training! )
- Let each word **“look” at all other words** directly
- Learn **contextual representations** with **attention**, not recurrence

Now they power:

- GPT, ChatGPT, BERT, T5, DALL·E, AlphaFold...
- Language translation, summarization, code generation, image captioning, protein folding...

**Transformers didn’t just improve sequence modeling — they redefined it.**

---

# Foundational Knowledge (First Principles)

## 1. What Is a Sequence?

A **sequence** is just an ordered list of elements.

Examples:

- Words in a sentence
- Frames in a video
- Musical notes
- DNA bases

Important point: **order matters**.

> “Dog bites man” ≠ “Man bites dog”

---

## 2. What Is a Representation?

Neural networks don’t understand text — they understand **vectors** (numbers).

So every word gets mapped to a **vector representation**. This is done via:

- **Word embeddings** (like Word2Vec, GloVe, or learned from scratch)

Example:

- “Cat” → [0.3, 1.2, -0.6, ...]
- Think of it as placing “Cat” somewhere in semantic space, close to “Dog”

---

## 3. What Is Attention?

Here’s the key idea:

When processing a word, not all other words are equally important.

> In “The animal didn’t cross the street because it was too tired,”
>
> “it” refers to “animal,” not “street”.

**Attention lets the model learn to focus on relevant words.**

Instead of reading left to right like RNNs, we allow every word to "peek" at every other word.

Mathematically:

> Attention is a weighted sum of all input vectors
>
> — where the weights are learned based on relevance

Analogy:

- Think of each word shooting out “attention beams” to others, saying:

  “How much do I care about you?”

---

## 4. What Is a Neural Network Layer?

A **layer** in a network:

- Takes in vectors (like word embeddings)
- Applies weights + nonlinearity
- Outputs new vectors (transformed representations)

A Transformer is a **stack of these**, where each layer refines your understanding of the input

---

# Main Topic: Explanation in Layers

## Overview: What Is a Transformer?

At the highest level, a Transformer is:

- An **encoder-decoder** architecture (for translation tasks)
- Or just a **stack of encoder blocks** (for GPT-like models)

Each **encoder block**:

1. Applies **self-attention** → each word looks at all other words
2. Applies a **feedforward neural network** to each word individually
3. Uses **residual connections** + **layer normalization**

Let's now zoom in step-by-step.

---

## Step 1: Input Embeddings + Positional Encoding

## Intuition

Words are converted to vectors.

But Transformers **don't have recurrence**, so we must tell them **the position** of each word.

## Formalism

- Input: a sentence of *n* words
- Each word: embedding of size *d_model* (say 512 or 768)

X = [$x_1$, $x_2$, ..., $x_n$] → shape (n, d_model)

Add
**positional encodings**:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

This gives a vector per position — using sinusoids of different frequencies.

## Why sinusoids?

- Fixed, no training needed
- Allow the model to learn relative positions (like “next word”)
- Any shift becomes a linear function — useful for generalization

### Analogy:

It’s like stamping each word with its time-stamp in a rhythmic language — “I’m the 3rd word!” — using sine waves instead of digits.

---

## Step 2: Self-Attention

This is the beating heart of the Transformer.

### Intuition

Each word says: “How much should I care about every other word?”

This creates **context-aware representations**.

### Mechanics

For each input vector **x**, compute three things:

- **Query (Q)**: what am I looking for?
- **Key (K)**: what do I offer?
- **Value (V)**: what information do I carry?

All are linear projections:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Then compute attention scores:

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

Where:

- $QK^T$: dot product of each query with each key → similarity

- divide by $\sqrt{d_k}$ : to prevent large dot products (stabilizes gradients)
- SoftMax : makes weights sum to 1 (like attention weights)
- multiply by V : gather actual info from relevant words

### Example

If you're processing the word "it", and Q has high similarity to the K of "animal", the attention output will lean heavily on V of "animal".

---

## Step 3: Multi-Head Attention

Instead of doing attention once, we do it multiple times with different **heads**.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Each head learns a **different aspect** of attention:

- One head might learn syntax (subject-verb)
- Another might learn semantic roles (who did what)

### Analogy:

Imagine you're at a party, and you assign multiple "attention advisors":

- One listens for tone
- Another for content
- Another for who’s being addressed

You aggregate all their insights.

---

## Step 4: Feedforward Network (FFN)

After attention, each word’s vector is passed **independently** through a 2-layer neural net:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

This adds **nonlinearity** and transforms representations further.

---

### Step 5: Residuals + Layer Norm

To stabilize learning and let gradients flow, each sub-layer has:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

These **residual connections** help prevent “forgetting” information.

---

### Full Encoder Block:

1. Multi-head self-attention
2. Add + LayerNorm
3. Feedforward
4. Add + LayerNorm

Repeat for **N layers** (e.g., 6 or 12 or 96 in GPT-4)

---

## Mathematics and Mechanics

We now revisit the most critical formula:

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

**Every term matters**:

- $Q \in \mathbb{R}^{n \times d_k}$ : What each word is querying
- $K \in \mathbb{R}^{n \times d_k}$: What each word represents
- $QK^T \in \mathbb{R}^{n \times n}$: Every pairwise similarity
- Divide by $\sqrt{d_k}$: Normalize variance
- SoftMax: Turn scores into probabilities
- V: Pulls in actual data (context vector)

---

# Code Intuition (Minimal PyTorch)

Each component is written as a minimal class or function in PyTorch, assuming `torch` and `torch.nn` are already imported.

## 1. Scaled Dot-Product Attention

```
def attention(Q, K, V, mask=None):
    # Q, K, V: (batch, heads, seq_len, d_k)
    d_k = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / math.sqrt(d_k)  # (B, h, seq, seq)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))
    weights = F.softmax(scores, dim=-1)
    return weights @ V  # (B, h, seq, d_k)
```

> Intuition: Each word queries all others, scores them by dot-product similarity, and retrieves relevant content via weighted sum.

## 2. Multi-Head Attention Module

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
```

```
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, x, mask=None):
        B, T, D = x.size()
        # Project inputs to Q, K, V
        Q = self.W_q(x).view(B, T, self.num_heads, self.d_k).transpose(1, 2)
        K = self.W_k(x).view(B, T, self.num_heads, self.d_k).transpose(1, 2)
        V = self.W_v(x).view(B, T, self.num_heads, self.d_k).transpose(1, 2)

        # Apply attention
        attn = attention(Q, K, V, mask)  # (B, h, T, d_k)
        out = attn.transpose(1, 2).contiguous().view(B, T, D)
        return self.W_o(out)
```

> Multiple heads allow the model to learn different types of relationships in parallel.

---

## 3. Feedforward Network (Position-wise MLP)

```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model)
        )

    def forward(self, x):
        return self.net(x)
```

> Typically, d_ff is 4x d_model (e.g., 2048 for 512).

---

## 4. Positional Encoding (Sinusoidal)

```
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, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-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)  # shape: (1, max_len, d_model)

    def forward(self, x):
        return x + self.pe[:, :x.size(1)].to(x.device)
```

> This injects sequence order information into word embeddings.

---

## 5. Transformer Block (Encoder Layer)

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, num_heads)
        self.ff = FeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        x = x + self.attn(self.norm1(x), mask)
```

```
        x = x + self.ff(self.norm2(x))
        return x
```

> Residual connections + normalization = stability and training speed.

---

## 6. Stack of Transformer Layers

```
class TransformerEncoder(nn.Module):
    def __init__(self, num_layers, d_model, num_heads, d_ff):
        super().__init__()
        self.layers = nn.ModuleList([
            TransformerBlock(d_model, num_heads, d_ff)
            for _ in range(num_layers)
        ])

    def forward(self, x, mask=None):
        for layer in self.layers:
            x = layer(x, mask)
        return x
```

> This is the backbone of models like BERT and GPT.

---

## 7. Embedding Layer + Positional Encoding

```
class InputEmbedding(nn.Module):
    def __init__(self, vocab_size, d_model):
        super().__init__()
        self.token_embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model)
```

```
    def forward(self, x):
        return self.pos_encoding(self.token_embedding(x))
```

---

### Example Usage

```
x = torch.randint(0, vocab_size, (batch_size, seq_len))  # (B, T)
embed = InputEmbedding(vocab_size, d_model)
encoded = embed(x)  # (B, T, d_model)
output = TransformerEncoder(num_layers=6, d_model=512, num_heads=8, d_
ff=2048)(encoded)
```

### From Here to GPT:

For GPT-like models, just:

- Use **causal masking** in attention to prevent future lookahead.
- Add **language modeling head** on top: `nn.Linear(d_model, vocab_size)`

---

## Broader Connections

- **BERT**: Bi-directional Transformer encoder (no decoder)
- **GPT**: Decoder-only Transformer (uses causal attention mask)
- **T5**: Encoder-decoder Transformer trained on text-to-text tasks
- **Vision Transformers (ViT)**: Transformers on image patches

Transformers generalize across:

- Text
- Images
- Audio

- Proteins
- Code

They’re now the **foundation model** of modern AI.

---

## Mental Models & Intuition Recap

- A Transformer is like a **meeting room of experts** — every word gets to hear every other word.
- Attention is a **spotlight** — it decides where to look.
- Multiple heads are like **multiple eyes** looking at different things.
- Residuals + layer norms are like **glue and polish** — keep things stable and smooth.

---

## Key Takeaways Summary

- Transformer = Self-attention + FFN + Residuals + LayerNorm
- No recurrence; full parallelism
- Attention = weighted sum of values, based on query-key similarity
- Multi-head: multiple perspectives
- Positional encodings inject order info
- Foundation for modern AI models (GPT, BERT, T5, ViT)

## Mind Map

attachment:cf773a58-258f-46d7-bbd2-86d321318002:transformer_minmap.pdf

## Audio Podcast for overview

attachment:e205ee4d-1ca0-432c-bc15-97e8f2ec4096:The_Illustrated_Transformer_Explained.wav

---

# Frequently Asked Questions

Starting with easy to the most difficult

Easy

▼ What is a Transformer model in deep?

A Transformer is a deep learning architecture that relies entirely on self-attention mechanisms to model relationships between input tokens, allowing for parallel processing and improved long-range dependency modeling.

▼ What are the main components of a Transformer?

The main components are Multi-Head Self-Attention, Feedforward Neural Networks, Positional Encoding, and Residual Connections with Layer Normalization.

▼ What is the purpose of positional encoding in Transformers?

It provides information about the position of tokens in the sequence since Transformers lack recurrence or convolution.

▼ What is self-attention?

Self-attention is a mechanism where each token in a sequence attends to all other tokens to compute a context-aware representation.

▼ What is the difference between self-attention and cross-attention?

Self-attention attends within the same sequence, while cross-attention attends to another sequence, often used in encoder-decoder models.

▼ Why are Transformers preferred over RNNs for NLP tasks?

Transformers allow parallel computation, handle long-range dependencies better, and scale efficiently.

▼ What is the shape of the attention matrix in self-attention?

It is (sequence_length × sequence_length), representing attention scores between all token pairs.

▼ What is the benefit of multi-head attention?

It allows the model to jointly attend to information from different representation subspaces at different positions.

▼ How is softmax used in attention?

Softmax normalizes the attention scores so they sum to 1 and can be interpreted as probabilities.

▼ What is the output dimension of the self-attention mechanism?

It is the same as the input dimension, typically d_model, after passing through the output linear projection.

Medium

▼ How does scaled dot-product attention work?

It computes the dot product between query and key vectors, scales it by the square root of the dimension, applies softmax, and then uses it to weight the value vectors.

▼ Why do we divide by $\sqrt{(}d_k)$ in attention?

To prevent large dot product values which can push softmax into regions with very small gradients.

▼ How are queries, keys, and values computed in a Transformer?

They are obtained by applying separate learned linear projections to the input embeddings.

▼ What is layer normalization and why is it used?

Layer normalization normalizes the inputs across the features to stabilize and accelerate training.

▼ What is the role of residual connections in Transformers?

They help in training deeper networks by preserving the gradient flow and allowing gradient backpropagation through identity paths.

▼ How does a decoder block differ from an encoder block?

Decoder blocks use masked self-attention to prevent attending to future tokens, and also include cross-attention to encoder outputs.

▼ Why is attention considered parallelizable?

Unlike RNNs, attention mechanisms do not rely on sequential processing and can operate on all tokens simultaneously.

▼ What is masking in Transformers?

Masking is used to prevent the model from attending to certain positions, like padding tokens or future tokens in decoder.

▼ How is positional encoding implemented mathematically?

Using sinusoidal functions: sin(pos/10000^(2i/d_model)) and cos(pos/10000^(2i/d_model)).

▼ What is the function of the feedforward network in a Transformer block?

It processes each token independently using a two-layer fully connected neural network.

▼ What does the softmax in attention normalize over?

It normalizes over the sequence length dimension (typically the key dimension).

▼ What is causal attention?

Causal attention masks future tokens to ensure the prediction of a token depends only on past tokens.

▼ Why are multi-head attention outputs concatenated?

Each head captures different relations; concatenation combines these before projecting back to original dimension.

▼ What is the purpose of the final linear layer in attention?

It maps the concatenated multi-head outputs back to the original model dimension.

▼ How does attention handle variable-length sequences?

Attention works on matrices and can handle variable lengths using masking to ignore padding.

▼ How are Transformer parameters trained?

Via backpropagation using gradient descent and variants like Adam optimizer.

▼ What is label smoothing and why is it used?

Label smoothing prevents the model from becoming overconfident by assigning a small probability to all other classes.

▼ What is the intuition behind attention scores?

They represent how much focus a token should place on each other token.

▼ How can Transformers model long-range dependencies better than RNNs?

Through self-attention, where each token has direct access to every other token.

▼ What is the computational complexity of self-attention?

It is $O(n^2 \cdot d)$, where n is the sequence length and d is the model dimension.

▼ How is the decoder initialized in Transformer models like GPT?

With learned token embeddings and positional encodings, and uses causal attention.

▼ How does the Transformer architecture allow parallel training?

By using attention mechanisms that don't depend on sequential token processing.

▼ What are the key differences between BERT and GPT?

BERT uses bidirectional encoder-only architecture, GPT uses unidirectional decoder-only architecture.

▼ Why is dropout used in Transformers?

To regularize the model and prevent overfitting by randomly deactivating neurons during training.

▼ What is a learned positional embedding?

An embedding where positional vectors are learned during training instead of fixed sinusoidal values.

▼ What is the effect of increasing the number of attention heads?

It increases model capacity but also memory and compute cost.

▼ What happens if we remove positional encoding?

The model loses the sense of token order, which is essential for language understanding.

▼ How does Transformer handle out-of-vocabulary tokens?

Typically through subword tokenization like BPE or WordPiece.

▼ Why is attention sometimes referred to as a weighted average?

Because it outputs a weighted sum (average) of values, where weights are computed via softmax over similarities.

▼ What are the limitations of vanilla Transformers?

Quadratic memory and compute cost with sequence length, inability to scale to very long sequences efficiently.

Advanced

▼ Explain how multi-head attention enhances learning capacity compared to single-head attention.

Multi-head attention allows the model to focus on different subspaces or types of relationships simultaneously. Each head can learn to capture different patterns, such as syntactic or semantic relations, enabling richer representations.

▼ What are the computational bottlenecks of Transformers and how can they be addressed?

The main bottleneck is the quadratic time and memory complexity of the self-attention mechanism (O(n^2)). Solutions include efficient variants like Linformer, Longformer, Performer, and sparse attention models that reduce complexity to linear or sub-quadratic.

▼ How does the Transformer architecture handle padding tokens during attention?

Padding masks are applied to the attention scores, masking out positions corresponding to padding tokens by setting their scores to negative infinity before the softmax operation.

▼ What is the role of layer normalization in Transformer blocks and where is it applied?

Layer normalization stabilizes training by normalizing activations. It's typically applied before or after sub-layers (Pre-LN or Post-LN). Pre-LN is more stable for deep Transformers.

▼ Why does GPT use causal (autoregressive) masking, and how is it implemented?

Causal masking ensures that each token can only attend to previous tokens and not future ones. It's implemented by masking the upper triangular part of the attention matrix (i.e., future positions).

▼ How is gradient flow affected by residual connections in deep Transformer models?

Residual connections improve gradient flow by providing a shortcut path for gradients to backpropagate through, helping mitigate the vanishing gradient problem and allowing deeper architectures to train effectively.

▼ Explain the concept and use of attention bias in large language models.

Attention bias refers to learned or static biases added to the attention scores before softmax. These biases can encode structural priors, segment information, or cache-relative positions (as in GPT-3/4).

▼ What is rotary positional embedding (RoPE), and how does it compare to sinusoidal or learned encodings?

RoPE encodes relative position using rotation matrices applied to Q and K vectors. It enables better extrapolation to longer contexts and preserves linearity for position-relative operations, unlike fixed or learned embeddings.

▼ How does FlashAttention improve Transformer efficiency?

FlashAttention is a memory-efficient and numerically stable implementation of attention that computes softmax in-place and in fused kernels, significantly speeding up training and inference while reducing memory usage.

▼ Describe the architecture changes and motivation behind Transformer-XL.

Transformer-XL introduces segment-level recurrence and relative positional encoding to improve long-context modeling. It caches hidden states across segments, enabling learning dependencies beyond the fixed-length context window.

Expert

▼ Explain the key innovation in the Attention is All You Need paper.

The core innovation is the replacement of recurrence with self-attention mechanisms, allowing parallelization and improved long-range dependency modeling.

▼ How does memory compression affect attention in long-context Transformers?

Memory compression reduces the sequence length by summarizing previous tokens, reducing computation but potentially losing detail.

▼ What are the differences between encoder-only, decoder-only, and encoder-decoder Transformer architectures?

Encoder-only (e.g., BERT) encodes input context, decoder-only (e.g., GPT) generates text autoregressively, and encoder-decoder (e.g., T5) translates input to output, combining both mechanisms.

▼ What is relative positional encoding and how is it implemented?

Relative encoding focuses on the distance between tokens, not their absolute positions, often implemented by modifying attention scores based on token offsets.

▼ What is the role of the output linear projection in multi-head attention?

It maps the concatenated outputs of all attention heads back to the model's original dimensionality.

▼ What causes instability in training deep Transformer models?

Common causes include vanishing gradients, poor initialization, attention collapse, and overfitting due to small batch sizes.

▼ What is the impact of model width vs. depth on Transformer performance?

Wider models can learn more parallel patterns, while deeper models learn more complex hierarchies. Optimal trade-off depends on task, data, and training scale.

▼ What is parameter sharing in Transformers and where is it used?

Sharing weights across layers (like in ALBERT) reduces model size and encourages consistent representations.

▼ How does ALiBi positional encoding differ from traditional encodings?

ALiBi biases attention scores based on relative position directly in logits, enabling extrapolation to longer sequences without embeddings.

▼ What are MoE (Mixture-of-Experts) Transformers and how do they work?

MoE layers route each token to a subset of specialized feedforward networks (experts), enabling sparse computation and scalability.

▼ How is attention visualization used in model interpretability?

By inspecting attention matrices, we can see which tokens influence others, helping interpret how the model processes relationships.

▼ Why do larger Transformers generalize better?

Larger capacity enables them to model complex distributions, given sufficient regularization and training data.

▼ What is the trade-off between batch size and learning rate in large Transformer training?

Larger batch sizes require proportionally larger learning rates to maintain gradient signal strength.

▼ How do Transformers avoid attending to padding tokens?

They apply a padding mask that zeroes out attention weights to padding tokens before softmax.

▼ What is gradient checkpointing and how is it used in large Transformer training?

It saves memory by recomputing intermediate activations during backward pass instead of storing them all.

▼ What is attention collapse and how can it be avoided?

It happens when attention heads converge to similar patterns, reducing diversity. Solutions include head dropout and regularization.

▼ What is prefix tuning in parameter-efficient fine-tuning?

Prefix tuning appends learnable tokens to the attention keys/values, allowing task adaptation without updating core model weights.

▼ What is the role of temperature in Transformer-based generation?

Temperature controls randomness in sampling; higher values yield more diverse outputs, lower ones make outputs more deterministic.

▼ How does beam search differ from greedy decoding?

Beam search keeps multiple candidate sequences, allowing exploration of better global sequences than greedy decoding.

▼ What is the significance of zero-shot and few-shot learning in Transformers?

It demonstrates model generalization – learning tasks without gradient updates (zero-shot) or with minimal examples (few-shot).

▼ How does the attention mechanism relate to kernel methods?

Attention can be interpreted as a learned kernel over the input space, computing pairwise similarities and weighted aggregation.

▼ Why do attention weights not always align with linguistic intuition?

Attention is optimized for loss, not human interpretability. It may learn task-relevant but linguistically unaligned patterns.

▼ What are the implications of scale (GPT-3/4) on training dynamics?

Larger models exhibit emergent capabilities, need stable optimization techniques, and benefit from massive pretraining data.

▼ What is loss scaling in mixed-precision Transformer training?

It's a technique to avoid underflow in float16 by scaling up the loss before backpropagation and scaling gradients down after.

▼ Explain the architecture and benefits of Reformer.

Reformer uses locality-sensitive hashing for attention and reversible layers to reduce memory and compute.

▼ What are the drawbacks of attention-only models like GPT?

They may struggle with local features, lack recurrence-like state retention, and require more compute for long contexts.

▼ What are the key differences between Vision Transformers (ViT) and CNNs?

ViT processes image patches as tokens, allowing global attention, while CNNs focus on local receptive fields and spatial hierarchies.

▼ How does retrieval-augmented generation (RAG) use Transformers?

RAG retrieves relevant documents to condition the decoder, combining external knowledge with generative capabilities.

▼ How is positional bias learned in large-scale decoders?

Models learn attention bias patterns through position-related initializations or embedding layers, sometimes without explicit encodings.

▼ What is the function of the token type embedding in BERT?

It helps distinguish between segments (e.g., sentence A vs. sentence B) in tasks like next sentence prediction.

▼ What is model parallelism and how does it apply to Transformers?

Model parallelism splits large Transformer layers across devices, enabling training models too large for one GPU.

▼ Why is attention softmax applied after masking?

To ensure masked positions have zero probability and don't affect the result.

▼ What is residual dropout and how is it different from standard dropout?

Residual dropout is applied to sublayer outputs before adding to residual path, helping regularize each residual computation.

▼ What are dynamic position biases and how are they used in attention?

They learn attention biases as a function of relative positions dynamically, improving generalization to unseen sequence lengths.

▼ What are the challenges in fine-tuning large pretrained Transformers?

Overfitting, catastrophic forgetting, computational cost, and instability with small datasets.

▼ Explain the LayerDrop technique in Transformer training.

LayerDrop randomly drops entire layers during training for robustness and regularization, encouraging redundancy tolerance.

▼ What are some causes of mode collapse in Transformer generation?

Overfitting, high repetition penalties, and poor temperature/top-k settings can lead to repetitive or generic outputs.

▼ How does top-k sampling work in Transformer decoding?

It samples from the top k most likely next tokens instead of the full distribution, reducing tail risks.

▼ How do large Transformers learn arithmetic reasoning?

By abstracting patterns over sequences of digits and their operations, often requiring specific data exposure.

▼ What are attention heads and why might some be pruned during inference?

Each head is a separate attention subspace; redundant heads may be pruned for efficiency without much accuracy loss.

▼ How does tokenization affect Transformer training and inference?

It impacts vocabulary coverage, sequence length, model compression, and generalization to unseen words.

▼ How do language models represent syntax and semantics internally?

Through distributed representations in attention layers and hidden states, implicitly capturing grammar and meaning.

▼ What is the role of positional embeddings in multilingual Transformers?

They help distinguish between structurally different sentence patterns across languages while retaining sequence order.

▼ How do Transformer models generalize to unseen sequences?

By learning abstract representations and relational patterns, not memorization, across token sequences.

▼ What are the effects of context window length on model performance?

Longer contexts improve coherence and reasoning but increase memory and computation costs.

▼ What is the intuition behind head diversity in attention?

Different heads specialize in capturing unique relational patterns, improving model expressiveness and robustness.

▼ How are embedding matrices tied in Transformers and why?

Tying input and output embeddings reduces parameter count and can improve performance by enforcing consistency.