1. Detailed Explanation of Self-Attention

Self-attention is a mechanism that allows a model to look at every word in a sequence simultaneously and decide which other words are relevant to understanding the current word.[1]

How it works (The "Search Engine" Analogy):

Imagine reading the sentence: "The animal didn't cross the street because it was too tired."

To understand what "it" refers to, the model creates three vectors for every word:2

- **Query (Q):** What the current token is looking for (e.g., "it" asks, "what noun do I match?").
- **Key (K):** What the token identifies as (e.g., "animal" says, "I am a noun, subject").
- **Value (V):** The actual content/meaning of the token.[3]

The model calculates a score (dot product) between the Query of the current word and the Keys of all other words.[4]

- "it" (Q) $\times$ "animal" (K) $\rightarrow$ High Score (High Attention).
- "it" (Q) $\times$ "street" (K) $\rightarrow$ Low Score.

These scores are normalized using **Softmax** to create weights.[5] The model then sums up the **Values (V)** weighted by these scores to create a new representation for "it" that is heavily influenced by "animal".[6]

2. Disadvantages of Self-Attention & Solutions

Disadvantage: Quadratic Complexity ([7]$O(N^2)$)[8]

- To calculate attention, every token must compare itself to every other token.[9]
- If you double the sequence length (e.g., 4k to 8k tokens), the computation and memory cost increase by **4x**.
- This makes processing very long documents extremely expensive.[10]

How to Overcome it:

- **Flash Attention:** An optimization that reorders computations to use the GPU's fast memory (SRAM) more efficiently, speeding up training/inference without changing the math.
- **Sparse Attention:** Instead of looking at *all* tokens, the model looks only at a few random ones or a local window.[11]

- **Linear Attention:** Approximates the N^2 operation with a linear $O(N)$ operation using kernel methods (though often with a slight quality drop).

3. What is Positional Encoding?

Transformers process all words simultaneously (in parallel), unlike RNNs which read word-by-word.[12] This means the Transformer has **no inherent sense of order**.[13] To the model, *"Man eats shark"* and *"Shark eats man"* look identical mathematically.

+1

Positional Encoding fixes this by adding a unique vector to each word embedding that represents its position (1st, 2nd, 3rd, etc.).[14]

- Standard Transformers use **Sinusoidal functions** (sine and cosine waves) of different frequencies so the model can learn relative positions (e.g., "word A is 5 positions behind word B").[15]

4. Transformer Architecture in Detail

The Transformer consists of two main blocks (though modern LLMs like GPT often use only one):

1. **Encoder (The "Reader"):**
    - Ingests the input text.
    - Uses **Self-Attention** to understand context.[16]
    - Passes data through a **Feed-Forward Network** (the "brain" that processes info).[17]
    - Output: A rich numerical representation of the text's meaning.[18]
2. **Decoder (The "Writer"):**
    - Generates text one word at a time.[19]
    - Uses **Masked Self-Attention** (it can only look at past words, not future ones).
    - Uses **Cross-Attention** (looks back at the Encoder's output to use the source info).

Note: GPT-style models (Decoder-only) discard the Encoder and just predict the next word based on previous words.

5. Advantages of Transformer vs. LSTM

Feature	LSTM (RNN)	Transformer

Processing	Sequential (Word by word)	Parallel (All words at once)
Speed	Slow training (cannot parallelize)	Fast training (highly parallelizable)
Long Context	Forgets distant info (vanishing gradient)	Remembers distant info perfectly (direct attention)
Distance	Linear path between words	Path length of 1 (immediate access)

6. Local Attention vs. Global Attention

- **Global Attention:** Every token attends to **every other token** in the sequence.[20] (Standard Transformer behavior). Accurate but expensive ([21]$O(N^2)$).[22]
- +1
- **Local (Windowed) Attention:** A token only attends to its neighbors (e.g., 5 words before and after). Fast ($O(N)$), but cannot "see" connections between the start and end of a long book.

7. Why Transformers are Heavy on Compute/Memory & Solutions

The Problem:

- **Parameters:** Billions of weights require massive VRAM (e.g., Llama-3-70B needs ~140GB VRAM just to load).
- **KV Cache:** During text generation, the model must store the Key and Value vectors of past tokens to avoid re-calculating them.[23] This cache grows linearly with context length and eats up memory.[24]
- +1

Solutions:

- **Quantization:** Loading weights in 4-bit or 8-bit integers instead of 16-bit floats (reduces memory by 2x-4x).[25]
- **Grouped Query Attention (GQA):** reduces the size of the KV cache by sharing Key/Value heads across multiple Query heads.

- **Mixture of Experts (MoE):** Only activates a small fraction of parameters (experts) for each token, saving compute.[26]

8. Increasing Context Length

Simply training on longer text crashes memory.[27] Solutions include:

- **RoPE (Rotary Positional Embeddings):** A mathematical trick that allows the model to extrapolate to longer sequences better than standard position embeddings.
- **ALiBi (Attention with Linear Biases):** Biases attention scores based on distance, allowing models to handle lengths longer than they were trained on.
- **Ring Attention:** Splits the long sequence across multiple GPUs so no single GPU runs out of memory.

9. Optimizing for a 100K Token Vocabulary

A large vocabulary (100k+) creates a massive "Embedding Matrix" and "Output Softmax Layer," which are computationally expensive.

Optimizations:

1. **Adaptive Softmax:** Instead of calculating probability for all 100k words every time, group rare words into clusters. Only calculate individual probabilities if the cluster is selected.
2. **Tied Embeddings:** Use the same matrix for both the Input Embeddings and the Output Layer to save parameters.
3. **Subword Tokenization (BPE):** Ensure the 100k vocab is efficient. Don't store "running", "runs", "ran" as separate IDs if "run" + suffixes can cover them.

10. Balancing Vocabulary Size (Computation vs. OOV)

- **Small Vocab (e.g., 10k):** No OOV (Out Of Vocabulary) because you break everything into characters (a, p, p, l, e).
    - *Issue:* Sequences become massive (1 word = 5 tokens), blowing up the N^2 attention cost.
- **Large Vocab (e.g., 100k):** Sequences are short (1 word = 1 token).
    - *Issue:* Massive embedding table parameters; hard to learn rare words.

Best Approach:

Use Byte Pair Encoding (BPE) or Unigram tokenization with a target size of 32k to 64k. This is the "sweet spot" where common words remain single tokens (keeping sequence length short) and rare words are split into subwords (avoiding OOV without bloating the vocab).

11. Types of LLM Architectures & Use Cases

Architecture	Type	Examples	Best For
Encoder-Only	Auto-Encoding	BERT, RoBERTa	**Understanding:** Classification, Sentiment Analysis, Named Entity Recognition. (Can see context from both left and right).
Decoder-Only	Auto-Regressive	GPT-4, Llama 3, Claude	**Generation:** Chatbots, Code generation, Story writing. (Can only see past context).
Encoder-Decoder[28]	Sequence-to-Sequence[29]	T5, BART[30]	**Transformation:** Translation, Summarization.[31] (Reads input, then writes output).