Machine Learning: 11

11 Transformer

Attention

From Sequential Processing to Attention

Traditional sequence models such as RNNs and LSTMs process tokens **sequentially**, meaning that each token must wait for the previous one to be processed. This introduces two fundamental limitations:

1. **Long-range dependencies are hard to model**: information from distant tokens tends to vanish or distort.
2. **Computation is inherently sequential**: parallelization on GPUs is limited.

The Transformer architecture challenges this paradigm with a bold claim: Attention is all you need. Instead of processing tokens one by one, the model allows **each token to directly attend to all other tokens**, regardless of distance. This replaces a chain structure with a **fully connected interaction graph** over the sequence.

😈

Question 27 (Transformer) The computational complexity of a forward pass through a standard, single-headed, encoder-only transformer of depth L, embedding dimension D, sequence length S and using a batch size of N is: $\mathcal{O}(NLSD^2)$

☐ TRUE ■ FALSE

Solution: False. The computational complexity is $\mathcal{O}(NLSD^2 + NLS^2D)$ where the second term comes from the attention mechanism.

Self-Attention: Core Idea

Self-attention allows a model to compute a **context-aware representation** for each token by weighting all other tokens in the sequence relative to it.

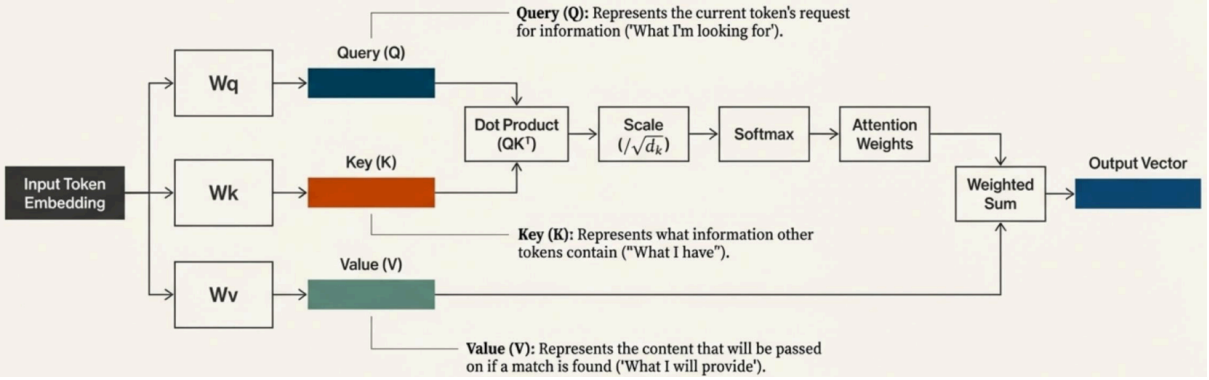
For a given token:

- It asks a question: *What am I looking for?* → **Query (Q)**

- Other tokens advertise what they contain: *What do I have?* → **Key (K)**
- They also provide content to share: *What will I give if selected?* → **Value (V)**

Crucially, **relationships are learned, not hard-coded by position.** This enables:

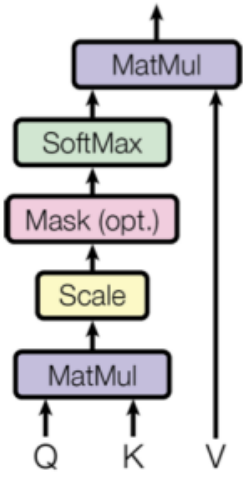
- Explicit modeling of long-range dependencies
- Bidirectional context (e.g., in BERT)
- Full parallel processing over tokens



Scaled Dot-Product Attention

The attention mechanism is mathematically defined as:

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



This computation follows a clear pipeline:

1. Compute similarity scores via dot product: QK^T
2. Scale by $\sqrt{d_k}$
3. Apply softmax to obtain attention weights
4. Compute a weighted sum over values V

Each output vector is thus a **contextualized embedding** that mixes information from all tokens.

Why Scaling by (\sqrt{d_k}) Matters

Without scaling, the dot product $Q \cdot K$ grows with the dimensionality d_k. Assuming independent components with unit variance:

$$\mathrm{Var}(Q \cdot K) = d_k$$

As d_k increases, the logits entering softmax become large in magnitude, causing:

- Softmax saturation
- Extremely peaked distributions
- Vanishing gradients

Dividing by $\sqrt{d_k}$ normalizes the variance back to order 1, keeping training stable.

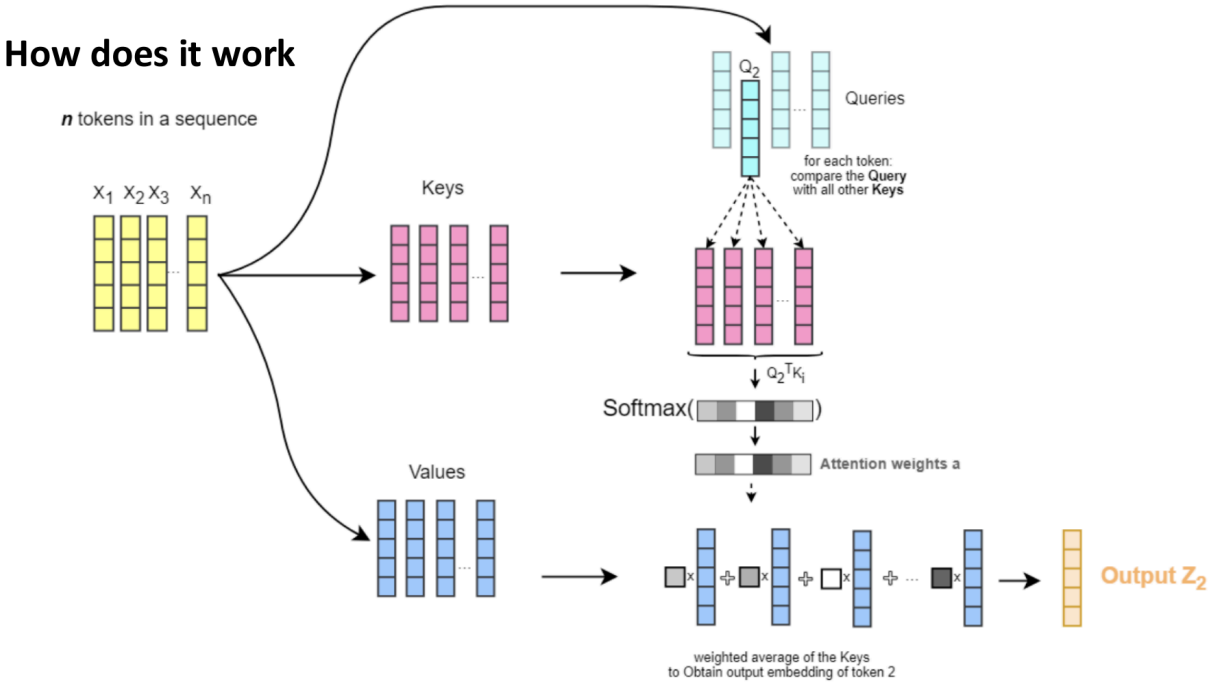
How Self-Attention Works in Practice

Given a sequence of n tokens:

1. Each token embedding is linearly projected into Q, K, V spaces.
2. For each token i, its query Q_i is compared against all keys K_j.
3. Softmax produces attention weights α_{ij}.
4. The output representation is:

$$z_i = \sum_j \alpha_{ij} V_j$$

Thus, **every output token is a weighted mixture of all input tokens**, conditioned on relevance. This is conceptually similar to information retrieval — but learned end-to-end.



Computational Cost and the Quadratic Bottleneck

For a sequence of length n and embedding dimension d:

- Computing attention scores: $O(n^2 d)$
- Softmax: $O(n^2)$
- Weighted sum: $O(n^2 d)$

Overall complexity: $O(n^2 d)$. This quadratic dependence on sequence length becomes the **primary scalability bottleneck**, especially for: Long text, High-resolution images, Video or audio streams. Much of modern research on Transformers is essentially about **escaping this n^2 curse**.

Question 2 Recall that the output of self-attention is given by the following formula:

$$Z = \text{softmax}\left(\frac{XW_QW_K^\top X^\top}{\sqrt{d_k}}\right)XW_V\,,$$

where $X \in \mathbb{R}^{n\times d}, W_Q \in \mathbb{R}^{d\times d_k}, W_K \in \mathbb{R}^{d\times d_k}, W_V \in \mathbb{R}^{d\times d}$ are the data, query, key, and value matrices, respectively. Here, d is the dimensionality of the tokens, d_k is the hidden dimensionality, n is the sequence length. The softmax is applied row-wise. Additionally, the dimensions satisfy $d_k < d < n$. Consider the following linear self-attention modification:

$$Z = \left(XW_QW_K^\top X^\top\right)XW_V\,.$$

Which of the following statements is **FALSE**?

- ☐ The linear self-attention modification can be used to reduce the complexity of the self-attention mechanism to be *linear* in the sequence length.
- ■ Replacing the product $W_QW_K^\top$ with a single matrix $W_{QK} \in \mathbb{R}^{d\times d}$ does not change the class of functions that can be expressed by both formulations of self-attention.
- ☐ The original self-attention mechanism is used in the Transformer model has a *quadratic* complexity in the sequence length.
- ☐ Both the original self-attention and the linear self-attention modification can be used with *causal masking*.
- ☐ The linear self-attention modification does not include normalization and thus does not benefit from a probabilistic interpretation.

Solution: 1. Transformers have quadratic complexity in sequence length. 2. The linear self-attention is one approach to reduce it to linear complexity. 3. The product $W_QW_K^\top$ is a $d \times d$ matrix, but with the rank $\leq d_k$, and replacing it with a single matrix $W_{QK} \in \mathbb{R}^{d\times d}$ can change the expressivity of the model if $d_k < d$. 4. The linear self-attention modification provided here does not include normalization and thus does not benefit from a probabilistic interpretation. 5. It is well-known that transformers can be used with causal masking, and the linear self-attention modification can be used with causal masking as well.

✅ **选项 1**

标准 attention：构造 $QK^\top$：$O(n^2d_k)$

线性 attention（合理重排后）：可以先算 $X^\top X$ 或低秩中间量；避免显式构造 $n \times n$ 的注意力矩阵

❌ **选项 2**

$$W_QW_K^\top \in \mathbb{R}^{d\times d}$$

但它的秩满足：

$$\text{rank}(W_QW_K^\top) \leq d_k \leq d$$

也就是说：$W_QW_K^\top$ **一定是低秩矩阵。**

如果直接用 $W_{QK} \in \mathbb{R}^{d\times d}$，这是一个**完全自由的矩阵**，秩可以是 d。

所以：原表达只能表示 **低秩双线性形式，**替换后可以表示 **任意双线性形式，表达能力发生了变化，而且是增强。**所以这句话说"不会改变函数类"是**错误的。**

✅ **选项 4**

causal masking 本质是：把“未来位置”的贡献置零。无论是 softmax attention 还是线性 attention 都可以通过 mask / 累积和 / 前缀技巧实现。

✅ **选项 5**

原 attention：softmax → 行和为 1；可以解释为“注意力分布”。

线性版本：没有 softmax；没有 normalization；输出只是线性加权和。➡️ **不能解释为概率。**

😈

Question 17 Let $N \in \mathbb{N}$. Which of the following sequence-to-sequence functions $f : \{0,1\}^N \to \{0,1\}^N$ **cannot** be arbitrarily well approximated by a decoder-only transformer without causal masking and without positional embeddings? In the choices below, $f_i(x_1, \ldots, x_N)$ refers to the i^{th} coordinate of $f(x_1, \ldots, x_N)$.

- ☐ $f(x) = \mathbf{0}$, $\forall x \in \{0,1\}^N$, where $\mathbf{0} \in \{0,1\}^N$ is the zero vector.
- ☐ $f : \{0,1\}^N \to \{0,1\}^N$ such that $f_i(x) = x_1 \oplus \cdots \oplus x_{i-1} \oplus x_{i+1} \oplus \cdots \oplus x_N$, where $\oplus$ is the addition modulo 2 operation.
- ■ $f(x) = v$, $\forall x \in \{0,1\}^N$, where $v \in \{0,1\}^N$ is the vector of alternating 0's and 1's: $v = (0,1,0,1,\ldots)$.
- ☐ $f : \{0,1\}^N \to \{0,1\}^N$ such that $f_i(x) = 1 - x_i$.

Solution: A decoder-only transformer without causal masking and without positional encodings cannot implement the function $f(x) = v$, $\forall x \in \{0,1\}^N$, where $v \in \{0,1\}^N$ is the vector of alternating 0's and 1's: $v = (0,1,0,1,\ldots)$. Indeed, such a transformer T can only implement permutation-invariant sequence-to-sequence mappings. Hence, if $T(0,1,\ldots) = (0,1,\ldots)$, then $T(1,0,\ldots) = (1,0,\ldots)$.

Tokenization: Making Everything Look Like Tokens

Attention operates on **arrays of tokens**, not raw data. Therefore, tokenization is the universal abstraction layer. Different modalities use different tokenization strategies:

Text

- Subword tokenization (e.g., BPE, WordPiece)
- Characters or bytes
- Example:

  `"Three guineafowl." → [Th][ree][ ][gui][nea][fowl][.]`

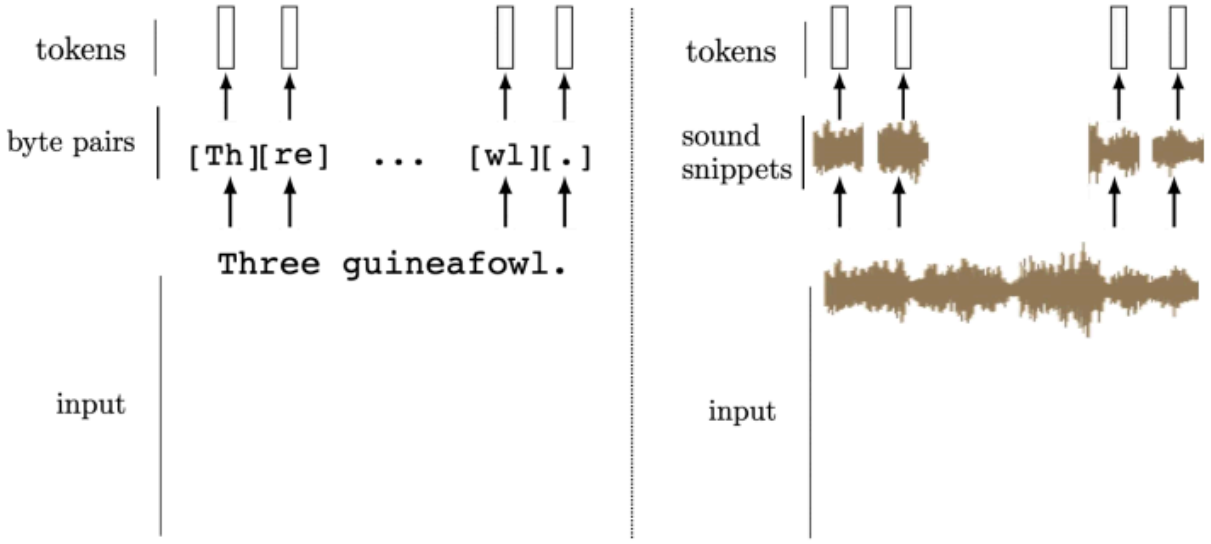
Vision

- Images are split into fixed-size patches
- Each patch is linearly projected into a token embedding
- This enables Vision Transformers (ViT)

Audio

- Waveforms are segmented into time-based snippets
- Each snippet becomes a token

After tokenization: The model no longer “knows” whether input was text, image, or sound; Everything becomes a sequence of vectors in $\mathbb{R}^d$.



Tokens vs Neurons: A Conceptual Shift

Traditional neural networks think in terms of:

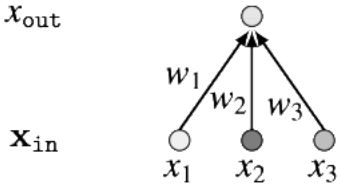
- Arrays of neurons
- Fixed topology

Transformers think in terms of:

- Arrays of tokens
- Dynamic, data-dependent connectivity via attention

Attention does not route signals through fixed edges — it **learns the graph structure on the fly**. This shift is one reason Transformers generalize so well across domains.

Linear combination of neurons

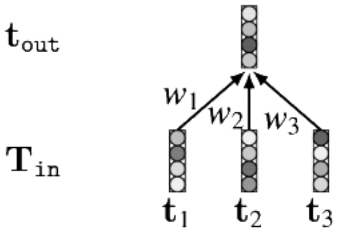


$$x_{\text{out}} = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$x_{\text{out}}[i] = \sum_{j=1}^{N} w_{ij} x_{\text{in}}[j]$$

$$\mathbf{x}_{\text{out}} = \mathbf{W}\mathbf{x}_{\text{in}}$$

Linear combination of tokens



$$\mathbf{t}_{\text{out}} = w_1 \mathbf{t}_1 + w_2 \mathbf{t}_2 + w_3 \mathbf{t}_3$$

$$\mathbf{T}_{\text{out}}[i,:] = \sum_{j=1}^{N} w_{ij} \mathbf{T}_{\text{in}}[j,:]$$

$$\mathbf{T}_{\text{out}} = \mathbf{W}\mathbf{T}_{\text{in}}$$

What Is a Token, Mathematically?

A **token is not a word, a patch, or a neuron**. A token is a **vector-valued entity**, formally:

$$t_i \in \mathbb{R}^d$$

A sequence of tokens forms a matrix:

$$T = \begin{bmatrix} t_1^\top \\ t_2^\top \\ \vdots \\ t_N^\top \end{bmatrix} \in \mathbb{R}^{N \times d}$$

Here:

- N = number of tokens (sequence length)
- d = channel dimension (embedding size)

Transformers operate primarily on this matrix.

Where Do Tokens Come From?

Tokens are produced by a **tokenization operator**:

$$t_i = W_{\text{tokenize}}(x_i)$$

What x_i is depends on modality:

- **Text**: subwords / bytes / characters
- **Vision**: image patches
- **Audio**: waveform segments

After tokenization, **modality disappears**. Everything becomes a sequence of vectors in $\mathbb{R}^d$. This is why Transformers are modality-agnostic.

Is Patch Tokenization "Good Enough"?

Patch tokenization (e.g. ViT) produces:

$$T \in \mathbb{R}^{K \times d}$$

But a problem emerges: **global information is not explicitly represented**.

Solution: introduce a **global / classification token** ([CLS]):

- A learnable vector
- Appended to the token sequence
- Aggregates information through attention

Attention Is Permutation-Invariant

Self-attention satisfies:

$$\text{Attention}(PT) = P\ \text{Attention}(T)$$

for any permutation matrix P. Meaning:

- Tokens have **no inherent order**
- "The dog bites the man" and "The man bites the dog" look identical

Position Encoding

Why We Cannot Encode Positions as Discrete Labels

Discrete indices are: Non-smooth, Non-comparable, Non-generalizable beyond training length. Neural networks require **continuous geometry**. So position must be encoded as a **vector**, not an ID.

Sinusoidal Positional Encoding: The Core Idea

Positional Encoding (PE) maps position (pos) to a vector:

$$PE(pos, 2i) = \sin\left(\frac{pos}{N^{2i/d}}\right)$$
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{N^{2i/d}}\right)$$

Interpretation:

- Each dimension is a sinusoid with a different frequency
- Low dimensions encode coarse position
- High dimensions encode fine position

Why Sinusoids Are Powerful

Three crucial properties:

1. **Smoothness**: nearby positions have similar encodings
2. **Extrapolation**: works beyond training sequence length
3. **Relative position recoverability**:

$$PE(pos + k) \text{ can be expressed as a linear function of } PE(pos)$$

This aligns perfectly with dot-product attention.

Final Assembly: What a Transformer Actually Sees

Each input token is:

$$\tilde{t}_i = \text{Embedding}(x_i) + \text{PositionalEncoding}(i)$$

The Transformer **never sees raw words, pixels, or positions**. It only sees **vectors that encode content + location**.

Multi-Head Attention（MHA）

1. Why Single-Head Self-Attention Is Not Enough

Consider the sentence:

> The animal didn’t cross the street because it was too tired.

The pronoun **“it”** is ambiguous. A single self-attention head produces **one attention distribution**, meaning it commits to **one dominant relational interpretation**. Language, however, is inherently multi-faceted:

- Polysemy(多义性) (e.g., *bank*)
- Coreference (e.g., *it*, *his*)
- Long-range dependencies
- Syntactic(句法) vs semantic roles
- Negation, temporal order, discourse structure

A single attention head risks **representational tunnel vision**: it learns *one* way to relate tokens, when language requires *many simultaneously*.

Core Idea of Multi-Head Attention (MHA)

Multi-Head Attention replaces one attention mechanism with **multiple parallel attention heads**.

Each head:

- Has its own W_Q^h, W_K^h, W_V^h
- Projects tokens into a different subspace
- Computes attention independently

Formally, for head h:

$$\text{head}_h = \text{Attention}(QW_Q^h, KW_K^h, VW_V^h)$$

The outputs are then concatenated and linearly projected:

$$\text{MHA}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)W_O$$

“A Council of Experts”: What Heads Actually Learn

Different heads tend to specialize in different relational patterns:

- Coreference resolution
- Syntactic dependency
- Semantic similarity
- Entity tracking

- Negation scope
- Long-range vs local context

No head is *assigned* a role. Specialization **emerges** because different projection matrices induce different similarity geometries. So MHA is best understood as: A set of parallel, low-dimensional relational lenses over the same token set.

Heads Are Not "Multiple Outputs" — They Are Multiple Bases

A common misunderstanding:

> "Multi-head attention means we attend multiple times."

More precise:

- Each head attends in a **different subspace**
- Each head uses **lower-dimensional vectors**
- Together they span a richer representation space

If $d_{\text{model}} = 64$ and $H = 8$, then each head works in $d_{\text{head}} = 8$.

This prevents:

- One head from dominating all dimensions
- Early softmax saturation
- Representation collapse

Computational Structure (What Actually Runs on the GPU)

Each head performs:

1. Score computation:

$$S_h = Q_h K_h^T \quad \Rightarrow O(n^2 d_{\text{head}})$$

2. Softmax normalization
3. Weighted sum:

$$O_h = A_h V_h$$

All heads run **fully in parallel**.

The quadratic cost comes from token–token interaction, not the number of heads.

This is why:

- MHA increases **expressiveness**
- But does not asymptotically worsen $O(n^2)$

Why MHA Is Expressively More Powerful

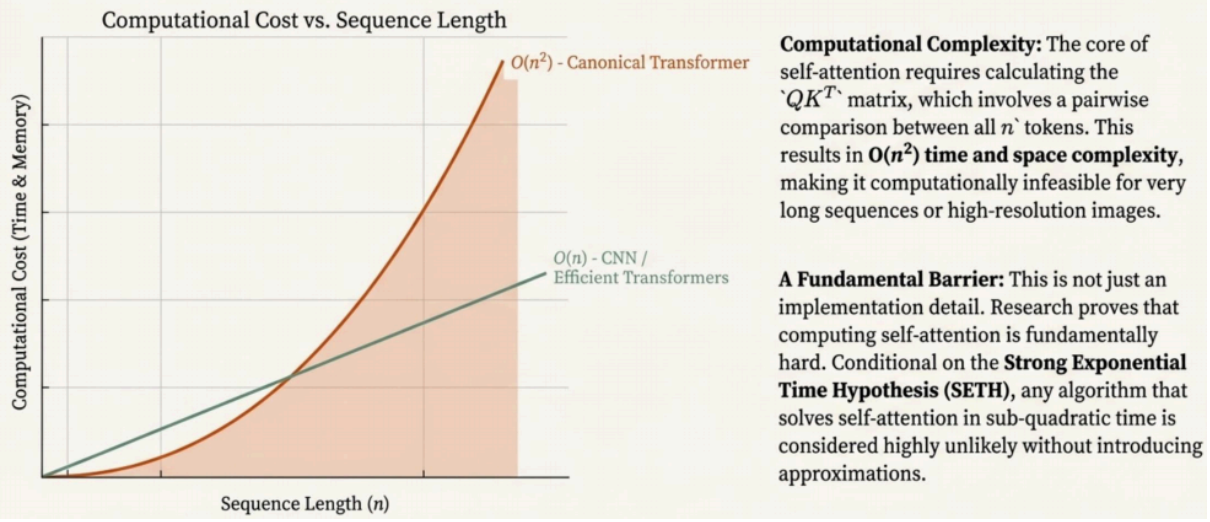
A single-head attention layer computes something close to a **rank-1 relational pattern**.

Multi-head attention approximates a **mixture of relational operators**:

- Different heads ≈ different adjacency matrices
- Concatenation ≈ feature-level composition
- Output projection ≈ learned fusion

From a linear algebra perspective: MHA expands the space of representable token–token interaction matrices. This is why removing MHA almost always hurts performance, even if depth is increased.

The Unavoidable Cost: Quadratic Complexity



Hence the explosion of:

- Sparse attention
- Low-rank attention

- Linear attention
- Performer / Nyström / FlashAttention (engineering-level relief)

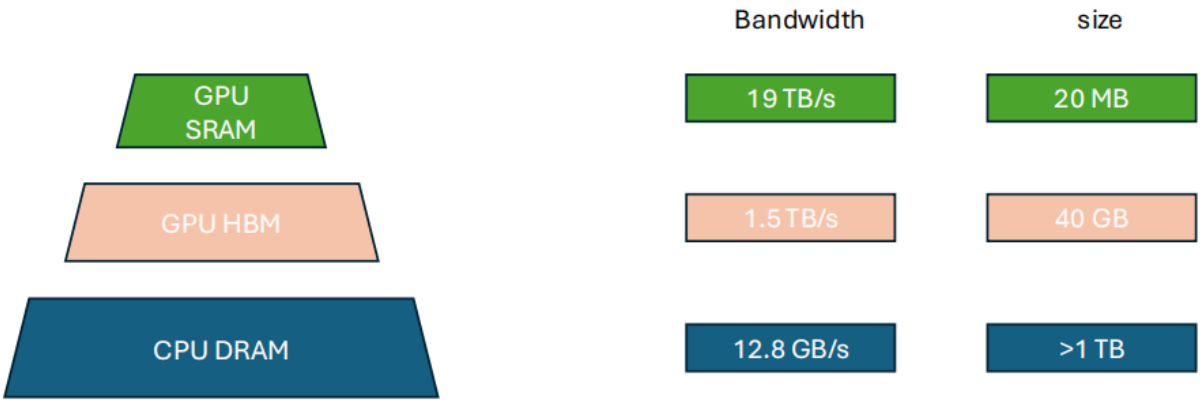
The Real Bottleneck: Not FLOPs, but Memory Traffic

Modern GPUs can perform matrix multiplications extremely fast. The true bottleneck of attention lies in **data movement**, not arithmetic. In standard attention, the following sequence happens:

1. Load Q, K from HBM
2. Compute and write $S = QK^T$ to HBM
3. Load S, apply softmax, write A back to HBM
4. Load A and V, compute $O = AV$, write output

Each step involves **round trips between GPU cores and High-Bandwidth Memory (HBM)**. Key observation: The same data is loaded and stored multiple times, even though computation itself is cheap.

GPU Memory Hierarchy: Why This Hurts

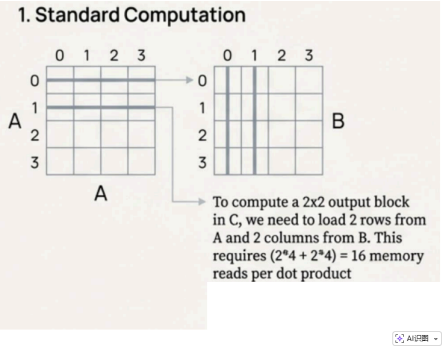


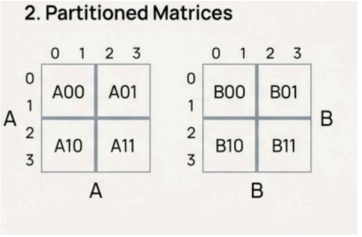
SRAM is: Extremely fast, Extremely small; while HBM is: Large, Still orders of magnitude slower. Thus: Every unnecessary HBM access dominates runtime.

First Weapon: Tiling (Blocking)

The top-left corner of the figure shows the drawback of the naïve approach: to compute one output element c_{ij}, we need an entire row of A and an entire column of B. Suppose we want to compute a 2×2 output block of matrix C (for example, block C_{00}). The naïve method means that each of these four elements must separately read one row from A and one column from B. As a

result, much of the data is repeatedly fetched from slow memory with almost no reuse, causing an explosion in memory access operations. The compute units (responsible for multiply-accumulate operations) often sit idle, waiting for data.





$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$
$$C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11}$$
$$C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10}$$
$$C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11}$$

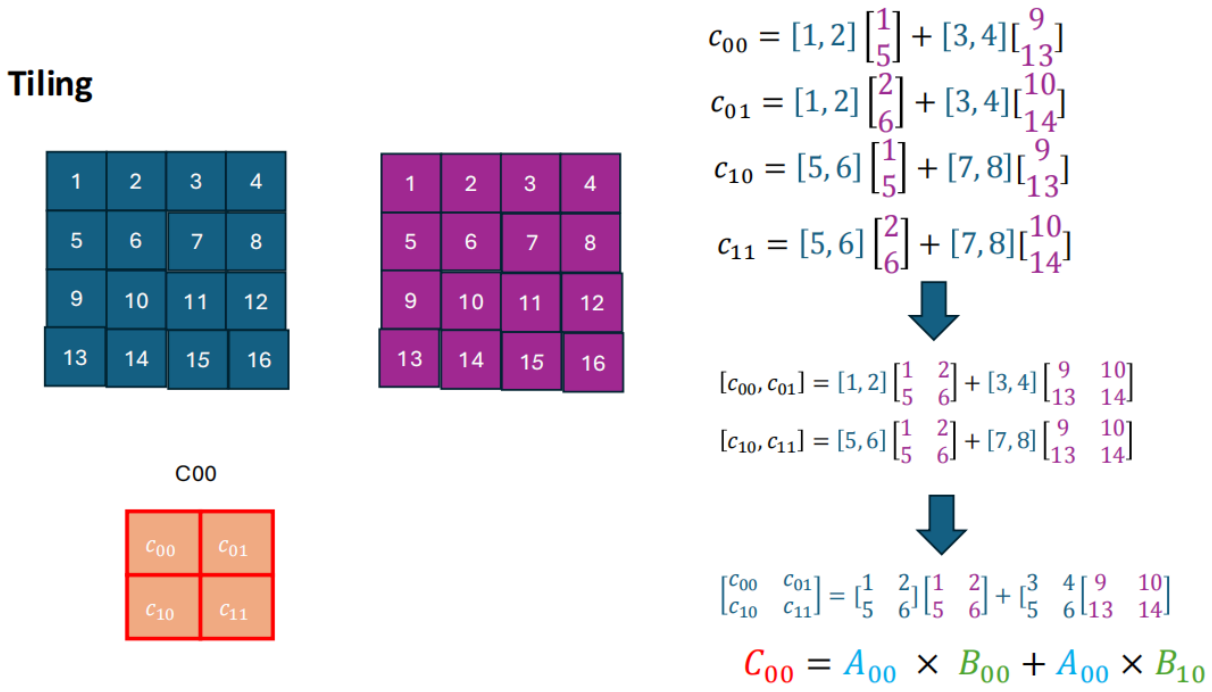
On the right, the partitioned matrices divide both A and B into 2×2 large blocks:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Matrix multiplication also holds at the block level (which is crucial):

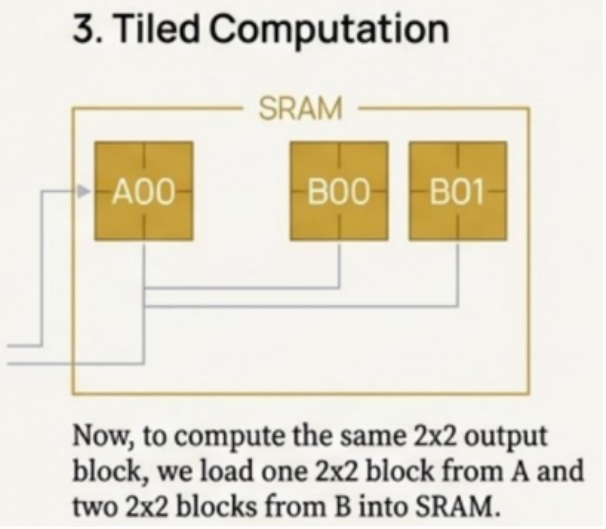
$$C_{00} = A_{00}B_{00} + A_{01}B_{10}$$
$$C_{01} = A_{00}B_{01} + A_{01}B_{11}$$
$$C_{10} = A_{10}B_{00} + A_{11}B_{10}$$
$$C_{11} = A_{10}B_{01} + A_{11}B_{11}$$

To compute the top-left block C_{00}, we only need to multiply the top-left block of A, A_{00}, with the top-left block of B, B_{00}, and then add the product of A_{01} and B_{10}. This allows us to focus on small blocks, load these few blocks into faster memory at once, perform a large number of multiply-accumulate operations within them, and then move on to the next set of blocks.



Therefore, the same block A_{00} can be paired multiple times with different block columns of B, allowing it to be loaded into fast memory once and reused repeatedly. Symmetrically, the loaded block B_{00} can also be reused by different blocks of A (such as A_{00} and A_{10}).

In the diagram on the right, these blocks are loaded into faster on-chip memory. Compared with the naïve approach, since each block in fast memory can be reused multiple times (roughly on the order of M times), the total amount of global memory access can be reduced to about $1/M$ of the original. This is what people often refer to as making the operator *compute-bound* (increasing the compute-to-memory-access ratio) rather than *memory-bound*.



Why Softmax Breaks Tiling

Softmax is **not a local operation**. To compute:

$$\text{softmax}(x_i) = \frac{e^{x_i - m}}{\sum_j e^{x_j - m}}$$

We need two **global properties** of the entire row:

1. The maximum value $m = \max_j x_j$ (for numerical stability)
2. The sum $\sum_j e^{x_j - m}$

This creates a dependency across *all tokens in the row*. Implication: **You cannot finalize any softmax output until you have seen every element.** This is why naïve attention requires:

- Writing S to HBM
- Reading it again for softmax
- Writing A back

Key Insight: Rewriting Softmax as an Online Algorithm

The breakthrough idea is simple but deep: **Softmax does not need to be computed in two full passes if we maintain the right running state.** Define a running maximum m_i and running sum d_i:

$$m_i = \max(m_{i-1}, x_i)$$

$$d_i = d_{i-1} \cdot e^{(m_{i-1} - m_i)} + e^{(x_i - m_i)}$$

After processing all elements:

$$\text{softmax}(x_i) = \frac{e^{x_i - m_N}}{d_N}$$

We eliminate the dependency on a pre-computed global maximum. This is **Online Softmax**.

Why Online Softmax Changes Everything

Online softmax enables:

- Processing attention **block by block**
- Keeping intermediate state (max, sum) in SRAM
- Avoiding materializing S and A in HBM

So instead of:

> Compute S → store → load → softmax → store → load → multiply

We can:

> Load a block → update running softmax → multiply with V → accumulate output

This unlocks **full tiling for attention**.

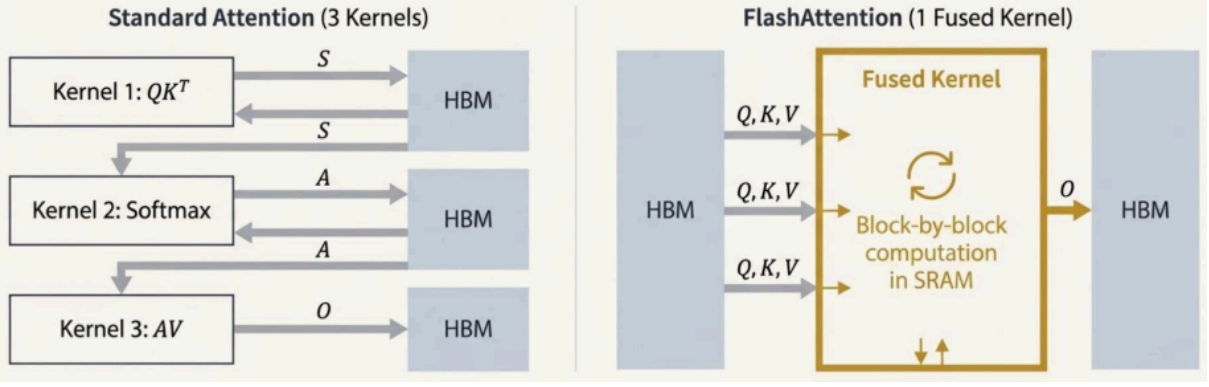
Putting It Together: Attention as a Streaming Algorithm

Rewritten attention loop:

1. Initialize running max and sum
2. For each block of keys:
   - Compute partial QK^T
   - Update online softmax state
   - Accumulate partial output O
3. Finalize normalization

Everything fits into: SRAM; One streaming pass over K/V. This is the intellectual core behind **FlashAttention** and related kernels.

What Is Still Missing After Online Softmax?



At this point, we already know how to compute softmax **without storing the full score matrix.** We maintain: a running maximum m_i; a running normalizer d'_i such that after seeing all keys:

$$a_i = \frac{e^{x_i - m_N}}{d'_N} \quad \text{where } x_i = qk_i^T$$

However, standard attention output is:

$$O = \sum_{i=1}^{N} a_i v_i$$

Problem: If we wait until the end to compute a_i, we must either (1) store all v_i, or (2) re-read them from HBM. That would **kill the memory advantage**. We want to update the output **incrementally**, just like softmax:

$$O_i \text{ from } O_{i-1}$$

In other words: Can we express the partial attention output after seeing i keys without revisiting earlier data? This is the final missing piece.

Let:

$$O'_i = \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} v_j$$

This is the *correct* attention output **restricted to the first i keys**, normalized with the *current* max m_i. We want a recurrence(循环):

$$O'_i = f(O'_{i-1}, x_i, v_i)$$

From the definition of $d'i$:

$$d'_i = d'_{i-1} \cdot e^{m_{i-1} - m_i} + e^{x_i - m_i}$$

Now split the sum:

$$O'_i = \sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{d'_i} v_j - \frac{e^{x_i - m_i}}{d'_i} v_i$$

Rewrite the first term using m_{i-1}:

$$\sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d'_{i-1}} \cdot \frac{d'_{i-1}}{d'_i} \cdot e^{m_{i-1} - m_i} v_j$$

But that sum is **exactly** O'_{i-1}. So we obtain the recurrence:

$$O'_i = O'_{i-1} \cdot \frac{d'_{i-1}}{d'_i} \cdot e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{d'_i} v_i$$

What This Means Algorithmically

We now have **three running states**, all small and SRAM-resident:

- m_i: running max
- d'_i: running softmax normalizer
- O'_i: running attention output vector

The final algorithm becomes:

```
m = -∞
d = 0
O = 0

for each block of keys/values:
    compute x = q k^T
    m_new = max(m, max(x))
    d_new = d * exp(m - m_new) + sum(exp(x - m_new))
    O = O * (d / d_new) * exp(m - m_new)
        + sum(exp(x - m_new) * v) / d_new
    m = m_new
    d = d_new

return O
```

Everything happens **online, block by block**.

Why This Enables FlashAttention

Now we can finally explain the diagram:

Standard Attention (3 Kernels)

- Kernel 1: compute $S = QK^T$, write to HBM
- Kernel 2: softmax $S \rightarrow A$, write to HBM
- Kernel 3: compute AV, write to HBM

Memory traffic: **O(N²)**

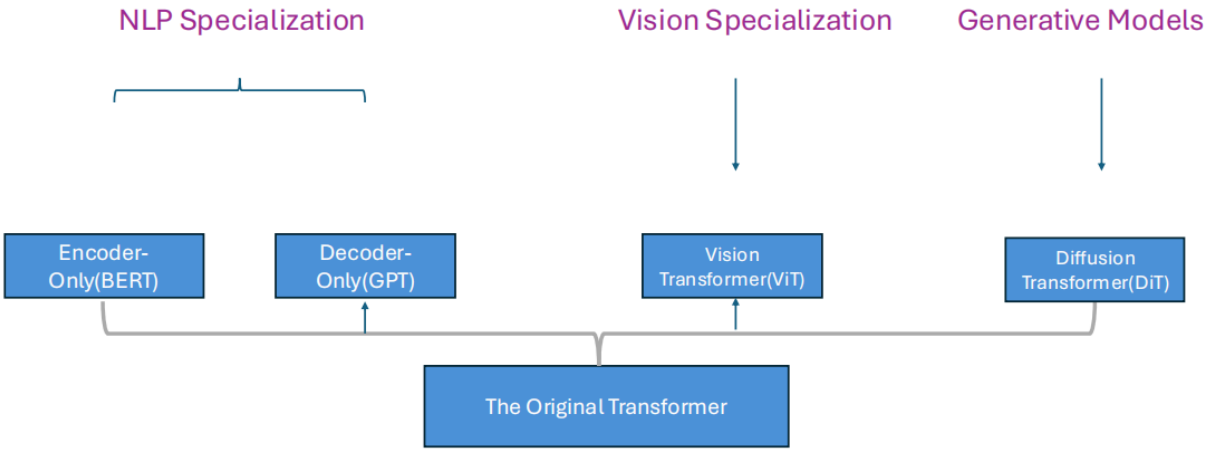
FlashAttention (1 Fused Kernel)

- Load blocks of Q, K, V
- Compute scores, softmax normalization, and output accumulation **inside SRAM**
- Write only the final O to HBM

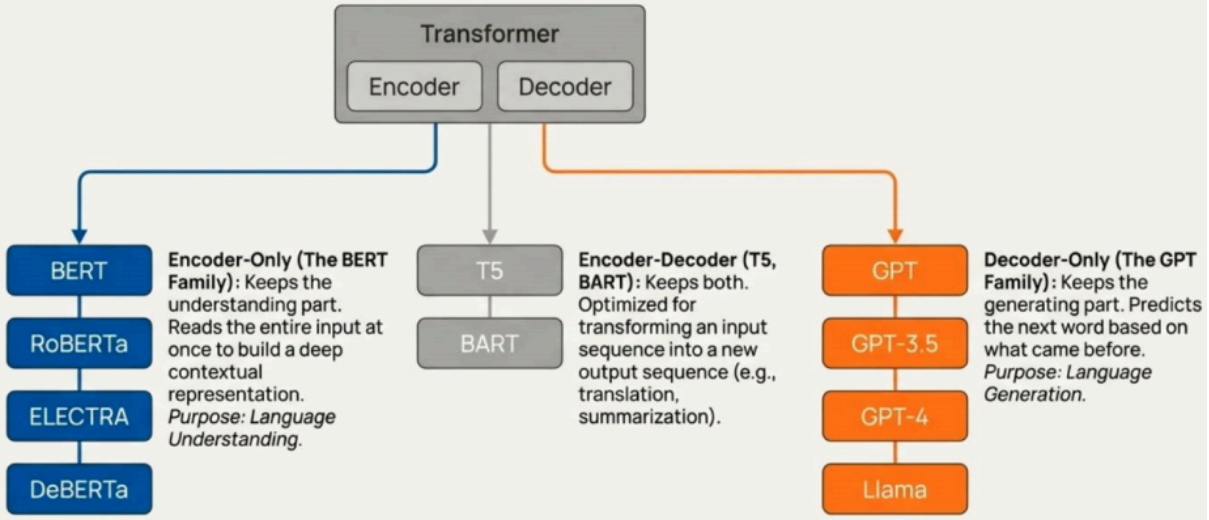
Memory traffic: **O(N)**

One Attention, Many Specializations

The original Transformer introduced **a general attention-based computation framework**. Everything that followed is **specialization, not reinvention**.



NLP: Encoder vs Decoder Is About Information Flow



Encoder-Only Models (BERT family)

BERT-style models use **bidirectional self-attention**. Each token can attend to: Left context; Right context; Itself. This is implemented via a **full attention mask**.

Objective: Masked Language Modeling (MLM); Sentence-level understanding

This makes BERT ideal for: Classification; NER; QA (extractive); Semantic understanding

Decoder-Only Models (GPT family)

GPT-style models use **causal (unidirectional) attention**. Each token can attend only to: Itself; Tokens to its left. Implemented via a **triangular attention mask**.

Encoder–Decoder Models (T5, BART)

These combine both worlds:

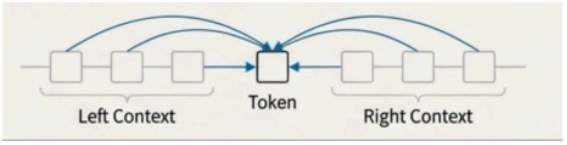
- Encoder: bidirectional understanding
- Decoder: causal generation
- Cross-attention bridges them

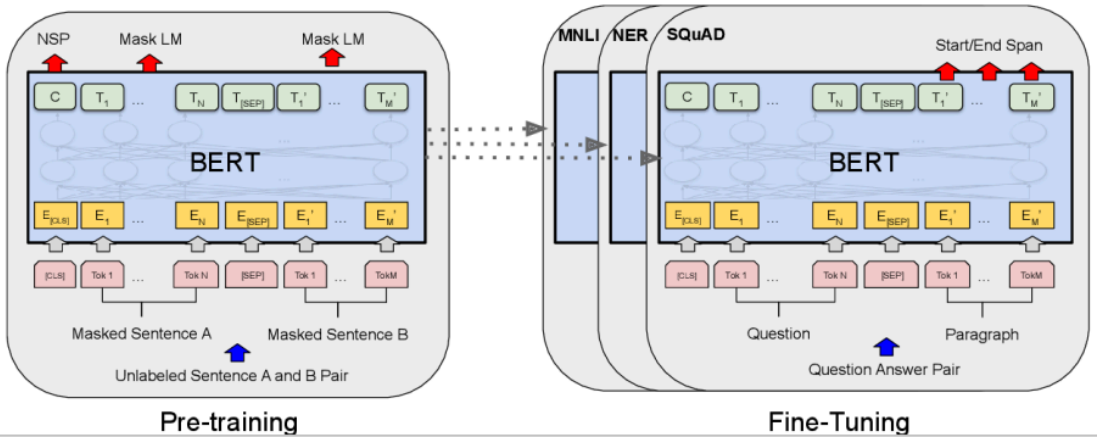
BERT vs GPT: Same Attention, Different Geometry

Aspect	BERT (Encoder)	GPT (Decoder)
Attention	Bidirectional	Causal
Mask	Full grid	Triangular
Objective	Understanding	Generation

Aspect	BERT (Encoder)	GPT (Decoder)
Analogy	Speed reader	Writer
Can see future?	Yes	No

BERT (Bidirectional-Encoder Representation Transformer)





GPT