TRANSFORMER ARCHITECTURE: FROM SELF-ATTENTION TO LLMS

Conceptual + Mathematical + Interview Perspective

Evolution of Generative Models

- **ML → Deep Learning → GANs → Transformers → LLMs**
- ML → feature engineering heavy
- RNN → sequential bottleneck
- GAN → unstable training
- Transformer → parallel attention
- “**Parallelization changed everything**”

What is a Transformer?

- Transformer is a self-attention-based architecture that processes sequences in parallel and captures long-range dependencies efficiently.
- It replaced RNNs and LSTMs in NLP by removing recurrence and using **self-attention**.
- **Highly Scalable**

Big Picture Idea

⚡ Parallel Processing

- Processes all tokens at once
- No sequential bottleneck
- Enables GPU acceleration
- Scales efficiently to large datasets

👁 Self-Attention

- Every word looks at every other word
- Learns relationships dynamically
- Captures contextual meaning
- No fixed window limitation

🔗 Long-Range Dependencies

- Connects distant words directly
- No information fading over time
- Handles complex sentence structure
- Maintains global context

- **Example: "A tiger jumped off the tree because he was thirsty," attention helps resolve that he → tiger.**

Core Idea:

- Instead of reading word-by-word, it reads the whole sentence and decides what matters.

Transformer Architecture Overview

- It consists of:
- **Encoder (stacked N times)**
- **Decoder (stacked N times)**
- Original use: Machine

  Translation

  Example: English → French

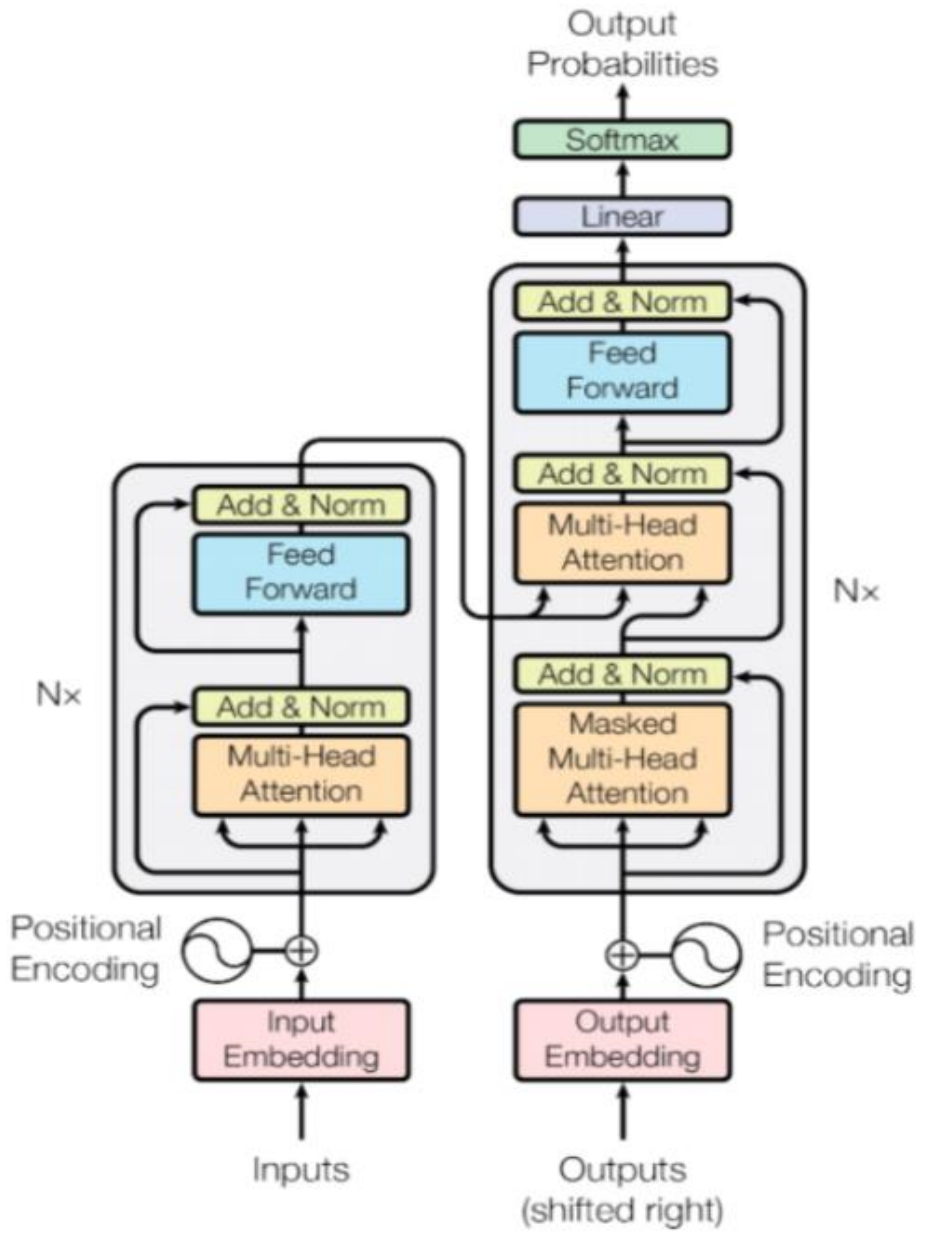
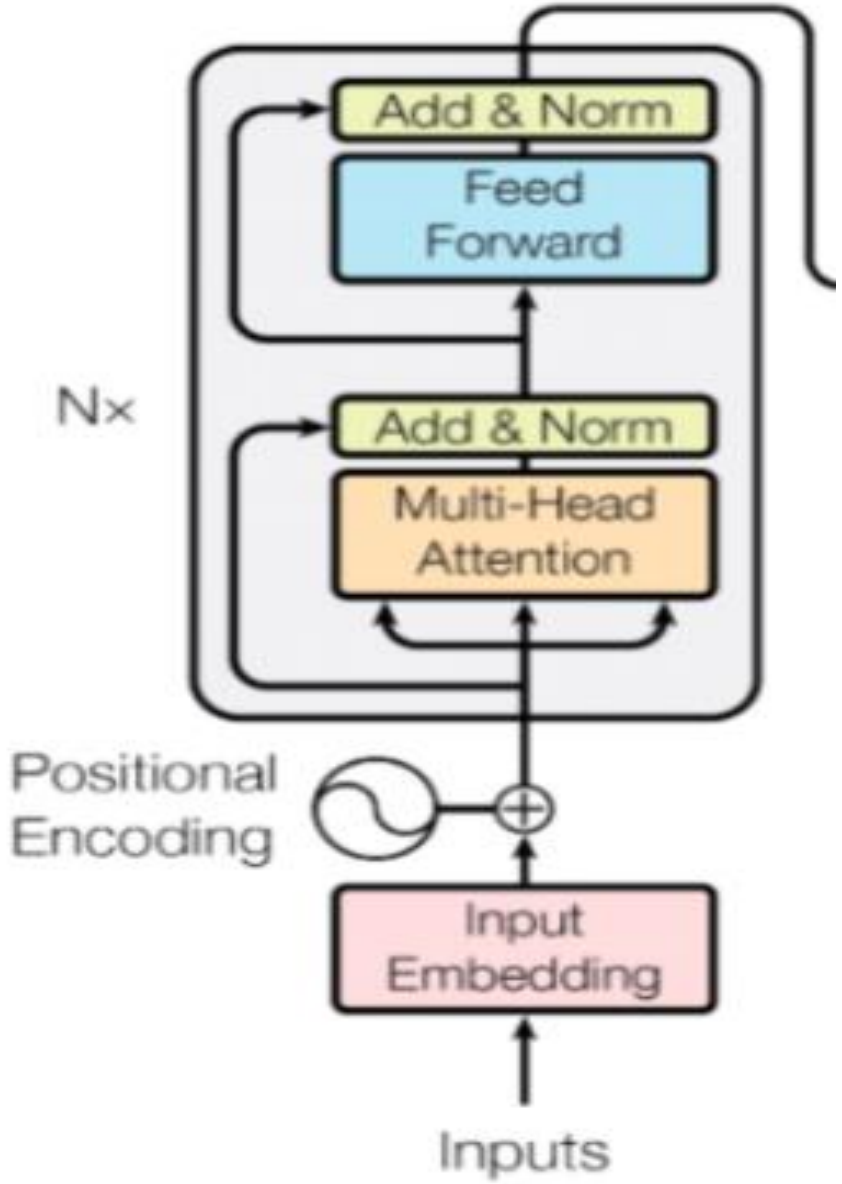


Figure 1: The Transformer - model architecture.

Left Side = Encoder Stack

- Left block contains:

1. Multi-Head Attention
2. Add & Norm
3. Feed Forward
4. Add & Norm
5. Repeated N times



Right Side = Decoder Stack

- Right block contains:

- 1 Masked Multi-Head Attention

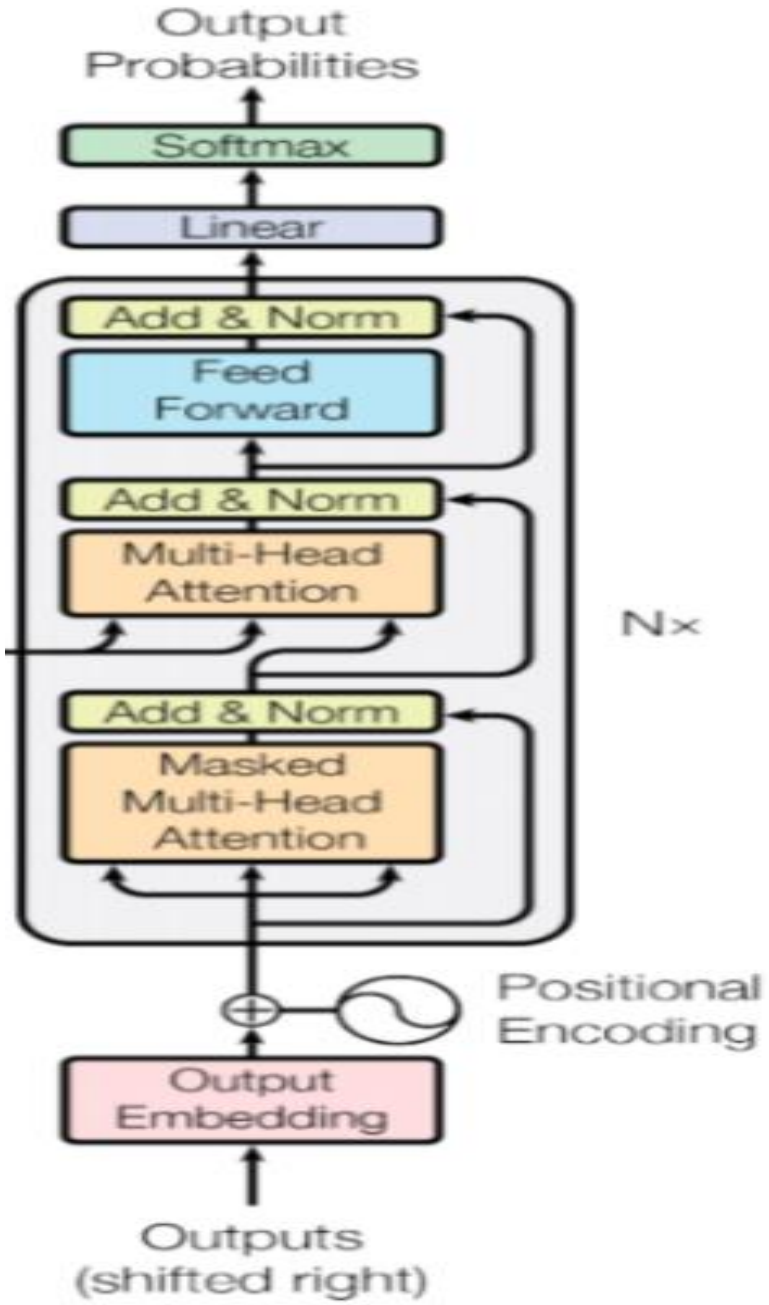
  2 Add & Norm

  3 Multi-Head Attention (Cross-Attention with Encoder output)

  4 Feed Forward

  5 Add & Norm

  6 Repeated N times



Difference Between Encoder & Decoder

Encoder	**Decoder**
Full self-attention	Masked self-attention
No masking	Cannot see future tokens
No cross-attention	Has cross-attention to encoder output

Why Masked Attention in Decoder?

Because during training:

When predicting token at position t:

It should NOT see tokens at t+1, t+2...

So masking ensures:

$$P(x_t | x_1, x_2, \ldots, x_{t-1})$$

Important Interview Insight

- Now here is something interviewers love asking:
- **GPT uses which part?**
- GPT = **Decoder-only Transformer**
- It removes the entire encoder side.
- **BERT uses which part?**
- BERT = **Encoder-only Transformer**
- It removes the decoder side.

Let us Continue

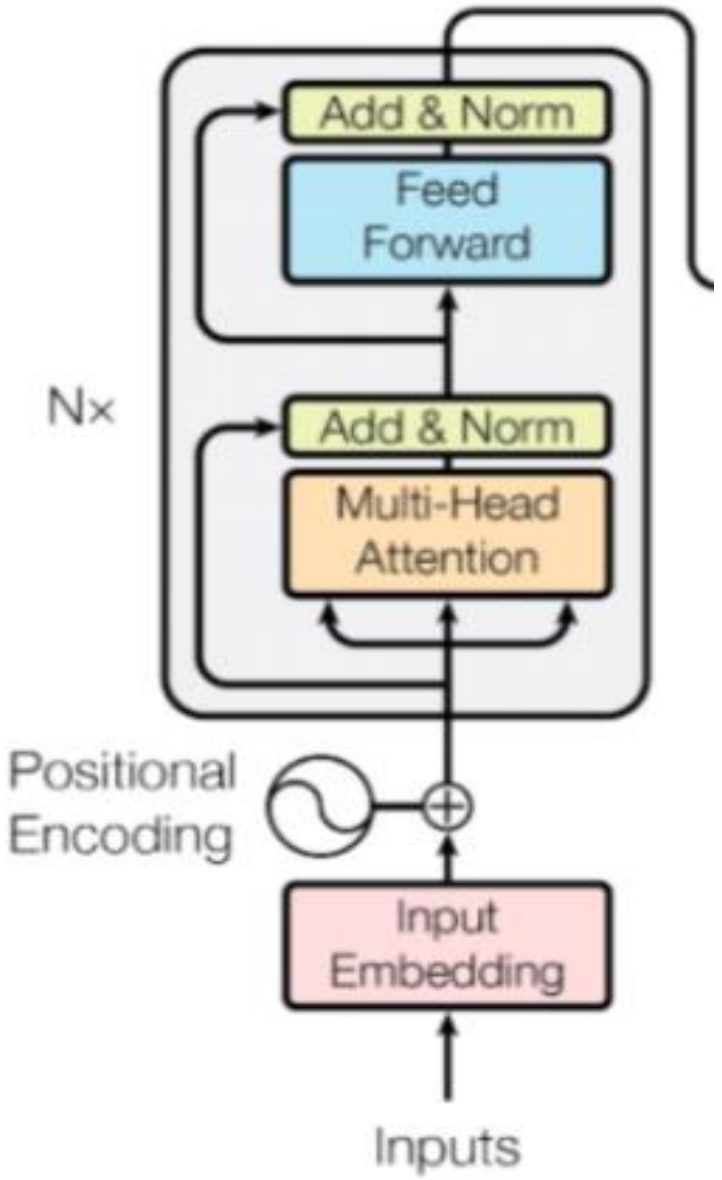
Transformer Block Structure

- **Each Encoder Layer contains:**

1. Multi-Head Self-Attention
2. Add & LayerNorm
3. Feed Forward Network
4. Add & LayerNorm

- **Decoder Layer adds:**

1. Masked Self-Attention
2. Cross Attention



Deep Dive: Multi-Head Attention Explained Step-by-Step

: It start with an input text to model.....

Journey Begins – Input to the Transformer

- We start with a sentence.
- The sentence is broken into tokens using a tokenizer.
- Each token is converted into a numerical vector using embeddings.
- Since Transformer does not understand order naturally, we add positional encoding.
- Now we have a matrix of shape (sequence length × embedding size).
- This matrix becomes the input to the Attention mechanism.

Enter Multi-Head Attention

- Instead of using a single attention mechanism, we use multiple attention heads.
- Think of each head as an expert.
- Every head looks at the same sentence but from a different perspective.
- One head may focus on grammar.
- Another may focus on long-distance relationships.
- Another may focus on contextual meaning.

Creating Query, Key, and Value

- For each head, the input embedding's(X) are projected into three different spaces:
  - Query → WQ → Q
  - Key → WK → K
  - Value → WV → V
- These projections are done using learnable weight matrices.
- Initially, these weights are randomly initialized (like in deep learning).
- During training, the model learns how to adjust them.

What Happens Inside Each Head?

- The Query of every token is compared with the Key of every other token.
- This comparison tells us how strongly tokens are related.
- The scores are scaled to maintain numerical stability.
- Then a softmax function converts scores into attention weights.
- These weights tell us:
- How much attention one token should pay to others.

Producing the Attention Output

- The attention weights are multiplied with the Value vectors.
- Each token becomes a weighted combination of all other tokens.
- The output now contains contextual understanding.
- The meaning of each word is updated based on the entire sentence.

Combining Multiple Heads

- Each head produces its own contextual output.
- These outputs are concatenated together.
- A final linear layer combines information from all heads.
- Now we return to the original embedding dimension.

Residual Connection and Normalization

- The original input is added back to the attention output.
- This is called a residual connection.
- It helps stabilize deep learning and prevents gradient problems.
- Then Layer Normalization is applied.
- This ensures training remains stable.

Feed Forward Network (FFN)

- After attention, the data passes through a Feed Forward Network.
- This network works independently on each token.
- It introduces non-linearity.
- It allows the model to learn complex feature transformations.
- Again, residual connection and normalization are applied.

What Have We Achieved?

- Each token now understands:
- Its own meaning
- Its relationship with other tokens
- Its contextual importance
- This completes one Transformer Encoder layer.
- Multiple such layers are stacked for deeper understanding.

Final Prediction (If Decoder)

- In language models, the final output is projected to vocabulary size.
- A softmax function converts it into probabilities.
- The model predicts the next word.

- **Multi-Head Attention allows a model to understand a sentence by letting multiple “experts” analyze relationships between words from different perspectives simultaneously.**

Step 1: Tokenization

- Let we have a Input sentence: **“Deep Learning is powerful"**
- Very first convert this into **tokens**
- **["Deep", "learning", "is", "powerful"]**
- If sequence length = n
- n=4

Step 2: Token Embedding

- Convert each token is a → **embedding vector.**
- **Assume** a **d_model** with = 4 (dimension)
- **Example:** then embeddings for each token
- **Deep** → [1, 0, 1, 0]
- **learning** → [0, 1, 0, 1]
- **is** → [1, 1, 0, 0]
- **powerful** → [0,0,1,1]

$$X \in \mathbb{R}^{(n \times d_{model})}$$

$$X \in \mathbb{R}^{4 \times 512}$$

- **Dimension = number of elements in one token embedding vector**

What is d_model in Real Models?

Model	d_model
BERT Base	768
GPT-2 Small	768
GPT-3	12288
LLaMA 7B	4096

- So instead of 4 numbers, each token might be represented by 4096 floating-point values.

Why Do We Need d_model?

- Each token embedding must capture:
- **Semantic meaning**
- **Syntactic role**
- **Contextual relationships**
- **Position information (after adding positional encoding)**
- **More dimensions → richer representation**

  But also → more computation

Example

- If sentence: "The cat sat"
- And d_model = 4
- We represent each word as:
- The → [1, 0, 1, 0]
- cat → [0, 1, 0, 1]
- sat → [1, 1, 0, 0]

- So shape becomes:
- [seq_len × d_model]
- 3 × 4
- Seq_len = no of tokens in input sequence

Step 2: Add Positional Encoding

- **We add positional encoding because attention has no sense of order.**
- Transformers process all tokens **in parallel**.
- Unlike RNN, they do not process : The → cat → sat step by step
- Instead they process: [The, cat, sat] at the same time
- So the model does not inherently know order.
- Without positional encoding: "The cat sat“ and "Sat cat the“ would look identical to the model.

So What Is Positional Encoding?

- It is a vector added to each token embedding that tells the model:
- “This token is at position 0”

  “This token is at position 1”

  “This token is at position 2”
- So Final Input = Token Embedding + Positional Encoding
- X = X + PE

Two Types of Positional Encoding

- **Learned Positional Embeddings**
- **Fixed (Sinusoidal) Positional Encoding**

Learned Positional Embeddings

- Model learns position vectors like normal embeddings.
- Used in: GPT and BERT
- Think of it like this:
- Just as words have embeddings,
- Positions also have embeddings.

- If max sequence length = 512

Then we create:

- Position 0 → Vector

Position 1 → Vector

Position 2 → Vector

...

Position 511 → Vector

Sinusoidal Positional Encoding

- Used in:
  - *Attention Is All You Need* (Vaswani et al.)
- **What is it?**
- Instead of learning positions, we compute them using sine and cosine functions.
- Each position in the sequence gets a unique pattern generated using:
- **sine for even dimensions**
- **cosine for odd dimensions**
- The frequency varies across dimensions.
- This creates a deterministic positional signal.

Defined mathematically as:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Where:

- pos = token position
- i = embedding dimension index
- d_model = embedding size

- **To Encode Relative Position Information**
- Sine and cosine have a mathematical property:
- Sin(a+b) can be expressed using Sin(a),Cos(a)
- This allows the model to: Learn relative distance between tokens.
- Meaning:
- If token A is 5 positions away from token B, the model can infer that using these periodic patterns.

Example (d_model = 4)

Position 0:

cpp Copy code

```
[ sin(0), cos(0), sin(0), cos(0) ]
= [ 0, 1, 0, 1 ]
```

Position 1:

cpp Copy code

```
[ sin(1/10000^0), cos(1/10000^0), sin(1/10000^(2/4)), cos(1/10000^(2/4)) ]
```

Each position gets a unique pattern.

- **They Create Unique Smooth Patterns**
- Each position gets a unique vector like:
- Position 0 → [0, 1, 0, 1, ...]

  Position 1 → slightly different

  Position 2 → slightly different

- **They Generalize to Longer Sequences**
- If we used simple numbers like:
- Position 1 → [1,0,0,0]

  Position 2 → [2,0,0,0]
- Model would not generalize well.
- But sine/cosine are periodic and bounded:
- $-1 \leq value \leq 1$.
- So model can extrapolate beyond training length.

Learned vs Sinusoidal (Intuition)

Sinusoidal	**Learned**
Fixed mathematical formula	Trainable parameters
Can extrapolate to longer sequences	Limited to trained max length
No additional learning	More flexible

Final Input

Now the vector contains:

- **semantic meaning**
- **position information**

Suppose:

Token embedding:

```bash
cat → [0.5, 0.2, 0.8, 0.1]
```

Positional encoding:

```css
position 2 → [0.1, 0.9, 0.3, 0.7]
```

Final input:

```csharp
[0.6, 1.1, 1.1, 0.8]
```

What is positional encoding?

- **Positional encoding is a technique used in Transformers to inject sequence order information into token embeddings. Since self-attention processes tokens in parallel and is permutation-invariant, positional encodings allow the model to understand the relative and absolute positions of tokens in a sequence.**

python

```python
import torch
import math

def positional_encoding(seq_len, d_model):
    pos = torch.arange(seq_len).unsqueeze(1)
    i = torch.arange(d_model).unsqueeze(0)
    angle_rates = 1 / torch.pow(10000, (2 * (i//2)) / d_model)
    angle_rads = pos * angle_rates
    pe = torch.zeros(seq_len, d_model)
    pe[:, 0::2] = torch.sin(angle_rads[:, 0::2])
    pe[:, 1::2] = torch.cos(angle_rads[:, 1::2])
    return pe
```

Step 3: Self-Attention (Core of Transformer)

This is the most important part.

For each token, we compute:

$$Q = XW^Q$$

$$K = XW^K$$

$$V = XW^V$$

Where:

- Q = Query
- K = Key
- V = Value
- W are learned weight matrices

- To measure attention score we are starting with three different **representations of the same token**,
- Each used for a different role in attention.
- **Intuition:** Think of each word as:
- **Query** → What am I looking for?
- **Key** → What do I contain?
- **Value** → What information do I pass forward?

Example Intuition

- Sentence: "The dog chased the cat"
- When computing attention for "chased":
- Its **Query** asks: “Who is the subject?”
- It compares against **Keys** of other tokens.
- It pulls **Values** from relevant tokens (like "dog").

Each head has:

$$W_Q, W_K, W_V$$

But their sizes are:

$$W_Q \in \mathbb{R}^{d_{model} \times d_k}$$

$$W_K \in \mathbb{R}^{d_{model} \times d_k}$$

$$W_V \in \mathbb{R}^{d_{model} \times d_v}$$

Size of Query(Q), Key(K) and Value(V)

X = (5 x 8)

W^Q = (8 x 4)

Q = (5 x 4)

K = (5 x 4)

V = (5 x 4)

- **If seq_len = 5**
- **If d_k = 4**
- **Q, K V :** Same size — but NOT same meaning

$$W^Q \neq W^K \neq W^V$$

$$QK^T = (5 \times 4)(4 \times 5) = (5 \times 5)$$

$$(m \times n) \times (n \times p) = (m \times p)$$

How are Q, K, V roles decided?

- Q, K, and V are produced by separate learned projection matrices. Their roles are architecturally defined — Queries are used to compute compatibility with Keys via dot product, and Values are weighted and summed to produce the attention output.

Important Relationship

- In multi-head attention:
- If: d_model = 512 and heads = 8
- Then each head gets:
- d_k = d_model / heads = 512 / 8 = 64 dimension
- So:
- Each head works in a **64-dimensional subspace**.
- This is crucial in interviews.

- **Shape of Input Matrix**
- Each token has 8 dimensions.
- There are 5 tokens. So input matrix shape:(5×8)
- **Value of d_k per head**
- d_k = d_model / heads =8/2 = 4
- Each head works on a 4-dimensional subspace.
- **Shape of Q After Projection (Per Head)**
- **Q = X * W ^ Q→ weight of Query**

- Where: shape of X is (5 × 8)
- shape of W^Q = (8 × 4) ← because each head projects to 4 dimensions
- So: (5×8)×(8×4)=(5×4)
- So shape of Q per head = (5 × 4) same for K and V

Let Me Ask You Something Important

- If: seq_len = 5, d_model = 8, heads = 2
- What will be:
- 1 **Shape of input matrix**?

  2 **Value of d_k per head**?

  3 **Shape of Q after projection (per head)?**
- Try answering — this is exactly how interviewers test depth.

Attention Score Shape

- We compute: **$Q.K^T$**
- **$Q = (5 \times 4)$, $K^T = (4 \times 5)$**
- **So: $(5\times4)\times(4\times5)=(5\times5)$**
- **K^T = Tranpose of K**

Component	Shape
Input X	(5 × 8)
d_k	4
Q per head	(5 × 4)
Attention scores	(5 × 5)
Output per head	(5 × 4)
Final concatenated output	(5 × 8)

How to make Attention Calculation

- **Assume: d_k = 2**
- **Let**

Q =

```
csharp


[1 0]
[0 1]
[1 1]
```

K =

```
csharp


[1 0]
[0 1]
[1 1]
```

Very First: Compute Score =Q.K^T

$$QK^T =$$

```
[1 0]   [1 0 1]
[0 1] x [0 1 1]
[1 1]
```

Result:

```
[1 0 1]
[0 1 1]
[1 1 2]
```

Then : Scale by √d_k

Since d_k = 2

$$\sqrt{2} \approx 1.41$$

Divide each element.

Why Divide by sqrt{d_k} ?

- **Without scaling:**
- Dot product grows large when d_k increases
- Softmax saturates
- Gradients vanish.
- So we scale

$$\frac{QK^T}{\sqrt{d_k}}$$

Later: Apply Softmax Row-wise

- This converts scores into probabilities.
- Example first row:
- [1, 0, 1] → softmax → [0.42, 0.15, 0.42]
- This means: Token 1 attends mostly to token 1 and 3.

$$\text{Attention Weights} = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

$$(n \times n)$$

Multiply by V

Output=Attention Weights·V

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

$$(n \times n) \cdot (n \times d_v)$$

$$(n \times d_v)$$

What's the intuition behind softmax in attention?

- Softmax converts the raw attention scores (dot products of Query and Key) into a normalized probability distribution over tokens. This ensures the weights sum to 1, allowing the model to compute a weighted sum of Values reflecting relative importance.

How weight are decided

- All three matrices W^Q, W^K, W^V are initialized randomly.
- Initialization method is same (e.g., Xavier / Glorot).
- Avoid vanishing / exploding gradients
- Maintain variance across layers
- So initially they are statistically similar.
- But the updating is different.

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

Notice:

- Q appears on the left of dot product.
- K appears on the right.
- V is multiplied after softmax.

That asymmetric usage creates different gradient signals.

- So during backpropagation:
- W^Q receives gradients from the left side of dot product.
- W^K receives gradients from the right side.
- W^V receives gradients from final weighted sum.
- Even if initialized identically, their gradient paths are different.
- So after just a few updates, they diverge.

Later need to Concatenate heads

- Combine h heads into one [4×8] matrix.

- Final output is weighted sum of Value vectors.
- **Why This Is Powerful**
- Each word can now look at:
- Previous words
- Next words
- Itself
- All at once.

If we have:

$$h = 8$$

Each head output:

$$(n \times 64)$$

Concatenate:

$$(n \times 512)$$

Back to original dimension.

Multi-Head Attention

- Instead of doing this once, we do it multiple times (e.g., 8 heads).
- Each head learns different relationships:
- Head 1 → grammar

  Head 2 → subject-verb

  Head 3 → long-range
- Then we concatenate all heads.

Step 4: Add & Norm

- Residual Connection (Skip Connection)
- Formula: Output= X+ Attention(X)
- What Is It? We add the original input X back to the output of the sublayer.
- So instead of:
- Output=Attention(X) We do:
- Output= X+ Attention(X)

- **Why Do This?**
- Because deep networks suffer from:
- Vanishing gradients
- Degradation problem (performance worsens as depth increases)
- **Residual connections allow:**
- Prevent gradient degradation
- Identity mapping if needed
- Easier optimization

- **Intuition**

- If attention learns something useful → good

  If it learns nothing useful → model can fallback to original X

- So training becomes stable.

- **Layer Normalization**
- After adding: X + Attention(X)
- We apply: LayerNorm(...)

- **What Does LayerNorm Do?**
- For each token vector:
- x^=(x−μ)\σ
- Where: μ = mean of features
- σ = standard deviation
- It normalizes across features

- **Why LayerNorm?**
- Because:
- Attention outputs can vary in scale.
- Deep stacking can cause unstable values.
- Keeps distribution stable.

BatchNorm	**LayerNorm**
Across batch	Across features
Bad for NLP	Good for NLP

Since NLP sequences vary in length → LayerNorm is preferred.

Step 5: Feed Forward Network (FFN)

After attention block, each token passes through:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

So:

- First expand dimension
- Apply non-linearity
- Project back

$$W_1 : (512 \rightarrow 2048)$$

$$W_2 : (2048 \rightarrow 512)$$

What Is Happening Here?

It is a 2-layer fully connected neural network.

Step 1:

$$xW_1 + b_1$$

Projects from:

$$d_{model} \rightarrow d_{ff}$$

Usually:

$$d_{ff} = 4 \times d_{model}$$

Example:

If:

$$d_{model} = 512$$

Then:

$$d_{ff} = 2048$$

Step 2:

ReLU:

$$max(0, x)$$

Adds non-linearity.

Step 3:

$$W_2$$

Projects back:

$$d_{ff} \rightarrow d_{model}$$

Important Insight

- Attention mixes information **between tokens**.
- Feed Forward processes information **within each token**.
- Attention = interaction
- FFN = transformation

Step 6: Add & Norm Again

After FFN:

$$\text{Output} = LayerNorm(X + FFN(X))$$

Again:

- Residual
- Stabilization

This completes one Transformer Encoder Block.

- In Encoder → No final softmax.
- In Decoder → Use softmax, after linear projection to vocabulary size..

Final layer:

$$\text{Linear}(d_{model} \rightarrow vocab\_size)$$

Then:

$$\text{Softmax}$$

Space Complexity

$$O(n^2)$$

Correct.

Because attention matrix is:

$$(n \times n)$$

Memory grows quadratically with sequence length.

That is the core limitation of Transformers.

Why Two Add & Norm Blocks?

Each encoder layer has:

1. Attention sublayer
2. Feed Forward sublayer

Each sublayer gets its own residual + normalization.

Visual Flow of One Encoder Layer

Input (x)

↓

Multi-Head Attention (Attention X)

↓

Add & Norm Z1= LayerNorm(X + Attention(X))

↓

Feed Forward FFN(Z1)=max(0,Z1W1+b1)W2+b2

↓

Add & Norm Output=LayerNorm(Z1+FFN(Z1))

↓

Output The Feed Forward network works independently per token

Why does Transformer expand dimension in FFN? dmodel to 4×dmodel

- The intermediate expansion increases the representational capacity of the model, allowing richer nonlinear transformations before projecting back to the original embedding size. This improves expressiveness without increasing sequence length.

Why do we need Add & Norm and FFN in Transformer?

- Residual connections ensure stable gradient flow in deep architectures, while Layer Normalization stabilizes activations. The Feed Forward network adds non-linearity and increases representational capacity by transforming token features independently after attention.

What does each of the 8 heads capture?

1 Subject–verb relationship "cat" ↔ "sat"

2 Nearby word context (local attention) "on" ↔ "mat"

3 Semantic similarity between nouns "cat" ↔ "mat"

4 Article–noun binding "the" ↔ "cat", "the" ↔ "mat"

5Verb's focus on objects "sat" ↔ nearby nouns

6 Long-range dependencies "cat" ↔ "mat" (start ↔ end)

7 Emphasis or negation cues Useful in complex sentences with "not", "never", etc.

8 Word position & rhythm (positional encoding) "sat" knows it's in the middle, not at the end or start

Beyond Transformer: Modern LLM Ecosystem

- GPT vs BERT
- Fine-tuning & PEFT
- Sampling strategies
- Evaluation
- Bias

Encoder vs Decoder

- **Encoder**(BERT)
- Self-attention (full access)
- Sees full sentence at once
- **Decoder (GPT)**
- Predicts next word
- Masked self-attention (cannot see future)
- Cross-attention (attends to encoder output)

Practical Difference in Real World

Use Case	GPT	BERT
ChatGPT	✅	❌
Google Search ranking	❌	✅
Text generation	✅	❌
Text classification	Possible but not ideal	Excellent

Core Architecture Difference

Feature	GPT	BERT
Transformer Type	Decoder-only	Encoder-only
Attention Type	Masked Self-Attention	Full Self-Attention
Context Direction	Left → Right	Bidirectional
Primary Goal	Text Generation	Text Understanding

Why GPT is Decoder-Only

- GPT only uses:
- Masked Self-Attention

  Feed Forward

  Stacked N layers
- No encoder.
- Because text generation doesn’t need separate encoder.

Why Transformers Beat RNNs

RNN	Transformer
Sequential	Parallel
Vanishing gradients	Stable
Hard long dependencies	Excellent long dependencies

- A Transformer is a deep learning architecture that uses self-attention instead of recurrence to process sequences in parallel, enabling better modeling of long-range dependencies and scalability.

What are fine-tuning techniques?

- Full fine-tuning: Update all weights → expensive.
- PEFT (Parameter-Efficient Fine-Tuning): Only train small added modules.
- Examples:
  - **Adapters:** Small bottleneck layers.
  - **LoRA:** Low-rank matrices for attention layers.
  - **QLoRA:** Quantized LoRA for memory efficiency.
  - **Soft Prompting:** Learnable prompt embeddings.

PEFT: Parameter-Efficient Fine-Tuning

- **Why?** Large models (GPT-3, LLaMA) have **billions of parameters** — too costly to fine-tune all.
- Idea: Freeze base model.
- Insert **lightweight, trainable modules.**
- Techniques:
- **Adapters:** Add small layers (e.g., bottleneck MLP).
- **LoRA:** Approximate weight updates with two low-rank matrices:

$$\Delta W \approx AB^T$$

where A and B are low-rank.

Example: Fine-tuning a 7B model with LoRA:

```
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForCausalLM


base_model = AutoModelForCausalLM.from_pretrained("llama-7b")
lora_config = LoraConfig(r=8, alpha=16, target_modules=["q_proj", "v_proj"])
peft_model = get_peft_model(base_model, lora_config)
```

- Benefits:
- Fine-tune only ~1% of parameters.
- Much cheaper, faster.

Top-K vs Top-P (nucleus) sampling?

- **Top-K:** pick next token from top k probable tokens.
- **Top-P:** pick from smallest set of tokens whose probabilities sum ≥ p.

- What is **self-attention vs cross-attention**?
- Self-attention: attends to tokens within the same sequence.
- Cross-attention: decoder attends to encoder output in seq2seq.

- **What does "autoregressive" mean?**
- Output is generated token-by-token, each conditioned on previous outputs. GPT is an autoregressive decoder-only model.

  - Given sequence: "The cat"
  - Model computes $P(\text{sat}|\text{The cat})$, then next $P(\text{on}|\text{The cat sat})$, and so on.

Mathematically:

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t|x_1, ..., x_{t-1})$$

- **How are the outputs of heads combined?**
- Each head produces a projection of dim dk. Concatenate all h heads [head1;head2;...;headh] and pass through final linear projection to get output of size dmodel.

How do we benchmark agents?

- Beyond output — measure trajectory, tool accuracy, fairness, bias.

What's the difference between fine-tuning and prompt-tuning?

- **Fine-tuning:**
- Retrain part or all of the model weights on your task.
- More resource-intensive, better for highly domain-specific needs.
- **Prompt-tuning / PEFT:**
- Keep base model frozen, only learn small prompt vectors (or adapters like LoRA).
- Much cheaper, faster — especially with large models.
- **When to use?**
- Prompt-tuning if budget and data are limited.
- Fine-tuning if you need full control and have resources.

How are LLMs evaluated?

- **Evaluation:**
- Automatic metrics: BLEU, ROUGE, F1 — for text overlap.
- Human evaluation: quality, relevance, helpfulness.
- Recently: LLM-as-a-judge frameworks.
- **Biases:**
- Social biases (gender, race stereotypes).
- Overconfidence in wrong answers.
- Data-driven biases (over-represented groups).

Flow Summary

1. Tokenize
2. Embed → (n × d_model)
3. Add Positional Encoding
4. Project to Q, K, V
5. Compute Scaled Dot-Product Attention
6. Softmax
7. Multiply with V
8. Concatenate heads
9. Linear projection
10. Residual + LayerNorm
11. FFN
12. Residual + LayerNorm
13. (Decoder only) Linear → Softmax

- What is **self-attention vs cross-attention**?
- Self-attention: attends to tokens within the same sequence.
- Cross-attention: decoder attends to encoder output in seq2seq.

- **What does "autoregressive" mean?**
- Output is generated token-by-token, each conditioned on previous outputs. GPT is an autoregressive decoder-only model.

  - Given sequence: "The cat"
  - Model computes $P(\text{sat}|\text{The cat})$, then next $P(\text{on}|\text{The cat sat})$, and so on.

Mathematically:

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t|x_1, ..., x_{t-1})$$

- **How are the outputs of heads combined?**
- Each head produces a projection of dim dk. Concatenate all h heads [head1;head2;...;headh] and pass through final linear projection to get output of size dmodel.

Diffusion Models different from GANs?

- Diffusion models generate data by starting from random noise and iteratively denoising it to produce realistic outputs.

  GANs, on the other hand, use a generator and discriminator in a minimax game to produce outputs.

  Diffusion models are more stable and produce high-quality images, while GANs are harder to train.