📌 Transformer Architecture

The Transformer is a deep learning model architecture introduced in the paper **"Attention is All You Need"** (2017). It has revolutionized Natural Language Processing (NLP), computer vision, and AI applications by replacing traditional Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs).

Unlike RNNs, which process data sequentially, Transformers process input in **parallel**, making them **faster** and **more efficient** for large datasets.

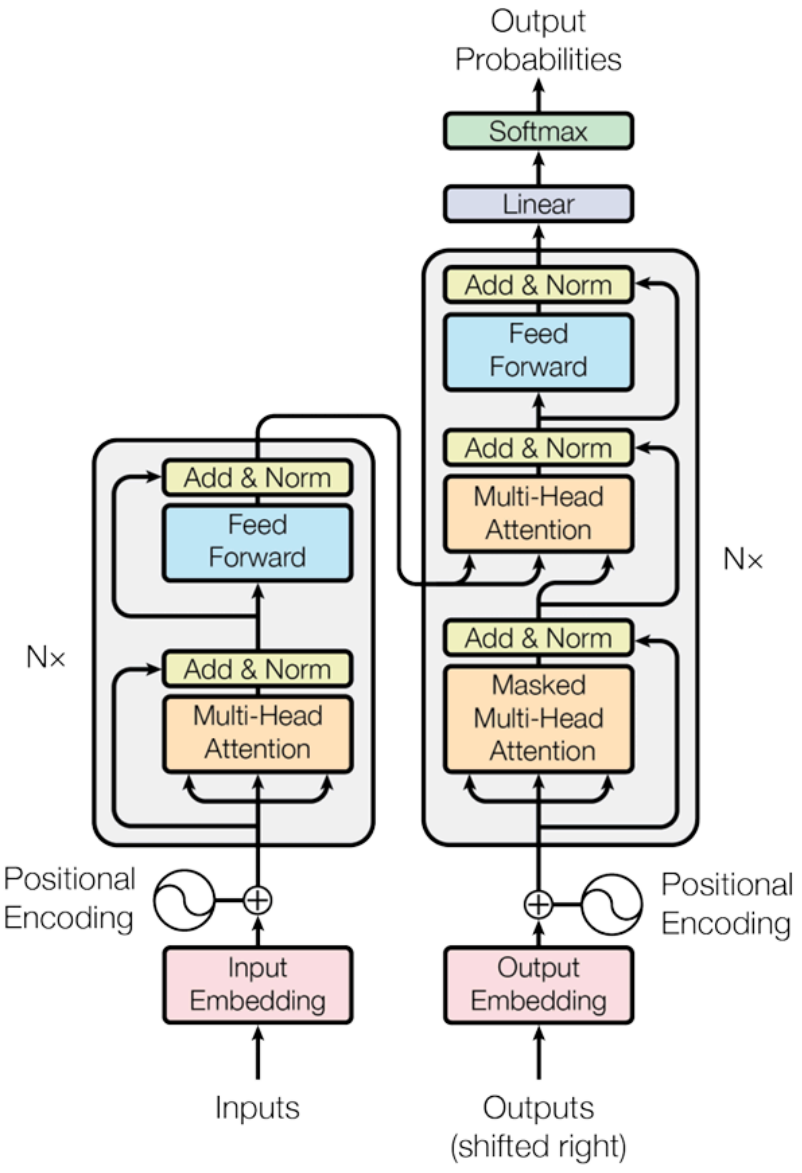


Figure 1: The Transformer - model architecture.

📌 Limitations of RNNs and LSTMs

- Sequential Processing (Slow Training) - Since RNNs pass hidden states sequentially, they cannot be **parallelised**, leading to longer training times.
- Vanishing Gradient Problem - In deep RNNs, **gradients shrink** exponentially while backpropagating through many layers, makes learning ineffective. LSTMs solve this to some extent with **gating mechanisms**, but they still struggle with very long sequences.
- Limited Context Retention - RNNs cannot efficiently retain **long-term context**, whereas Transformers use **self-attention** to maintain relationships between distant words.
- Fixed-Length Memory Bottleneck - If the sequence is long, earlier information **gets forgotten**. Transformers allow direct attention to all tokens at once, **preserving full context**.
- Difficulty in Handling Variable-Length Inputs - Transformers process **variable-length inputs** naturally, making them more flexible.
- High Computational Cost for Long Sequences - Transformers use **self-attention**, which **scales better** with **large datasets**.

📌 Key Features/Benefits of Transformer Architecture

- Parallel Processing - Unlike RNNs, which process data sequentially, Transformers process the entire **input sequence at once**. This makes them significantly **faster** and **more scalable**.
- Self-Attention Mechanism - Helps capture long-range dependencies in text. Can focus on **relevant words** in a sentence, regardless of their **position**.
- No Vanishing Gradient - Since there is no recurrent structure, Transformers avoid **vanishing gradients** and learn better representations.
- Dynamic Sequence Handling - No need for **fixed-size memory** (unlike LSTMs). Works well with variable-length inputs.
- Scalability to Large Datasets - Models like **BERT, GPT, and T5** are built using Transformers and can be **fine-tuned** on massive datasets.

✒️ Why Matrices Don't Naturally Capture Sequence/Order?

- A matrix multiplication treats **tokens as a bag of vectors**, performing **dot product** similarity.
- Matrix operations are permutation-invariant—meaning, **swapping two rows (tokens) does not change how the operation works**.
- No **sequential dependencies** are built into the attention mechanism itself.

✒️ Self Attention Matrix - The **Self-Attention Matrix** is the core component of the Transformer model that helps it understand relationships between words (tokens) in a sentence, **regardless of their position**. Unlike RNNs, which process sequences **step-by-step**, Self-Attention allows Transformers to process **all tokens at once** while determining their contextual importance.

A **Self-Attention Matrix** in transformers like **BERT, GPT, and other attention-based models** captures the relationships between tokens in a sequence. It is an $n \times n$ **matrix**, where n **is the number of tokens** in a sentence, and each value represents how much attention one token pays to another.

✒️ Understanding Self-Attention in a Matrix

Given an input sequence: "AI is powerful", tokenized as:
X **= ["AI", "is", "powerful"]**
A self-attention matrix A for this sequence might look like this:

$$A = \begin{bmatrix} 0.1 & 0.3 & 0.6 \\ 0.2 & 0.5 & 0.3 \\ 0.4 & 0.4 & 0.2 \end{bmatrix}$$

- The rows represent the **query token**.
- The columns represent the **key tokens**.
- Each value in the matrix represents the **attention score between tokens**.

For example

- A(1,3) = 0.6 means that token "AI" pays 60% of its attention to "powerful".

- A(2,2) = 0.5 means the token "is" attends 50% to itself.

✒️ Why is Self-Attention Important?

- Captures Long-Range Dependencies - **"The cat sat on the mat, and it was happy."** The word **"it" refers to "cat"**. Self-Attention helps the model understand this connection, even if **"it" is far from "cat"**.
- Handles Variable-Length Sequences - Unlike RNNs, Transformers don't require **fixed-length inputs**.
- Parallel Computation - Since attention can be computed for all words **simultaneously**, Transformers train much faster than RNNs/LSTMs.

📌 Why Not Add Word Position as a Third Dimension?

At first, it might seem logical to represent word **position as an additional dimension** in the input tensor. For example, if words are represented as **vectors (embeddings)**, we could try to append position information as another dimension, like: **Word Representation = [Embedding, Position]**

However, this approach has fundamental drawbacks, which is why Transformers use Positional Encoding instead. For example:

- It Treats Position as an Independent Feature - If we add word position as another dimension in the embedding, the model treats it **separately from word meaning**. This contradicts the idea that word position should **influence word relationships**.
- Does Not Capture Relative Word Order - Transformers rely on **self-attention**, which processes all words simultaneously. If we simply add a position dimension, the model won't know **how positions relate to each other**.
- Limits Generalization to Variable-Length Sentences - If the position is just an **extra dimension**, the model may overfit to specific sentence lengths. In reality, sentences have **variable lengths**, and we want the model to handle them all.
- Self-Attention Mechanism is Position-Agnostic - Self-Attention **does not inherently consider word order**; it treats all words equally. If we just add position as an extra dimension, **self-attention won't be used effectively**.

✒️ Positional Encoding - Since Transformers don't have a built-in sense of order (like RNNs do), **we encode positional information directly into the word embeddings using sinusoidal functions.**

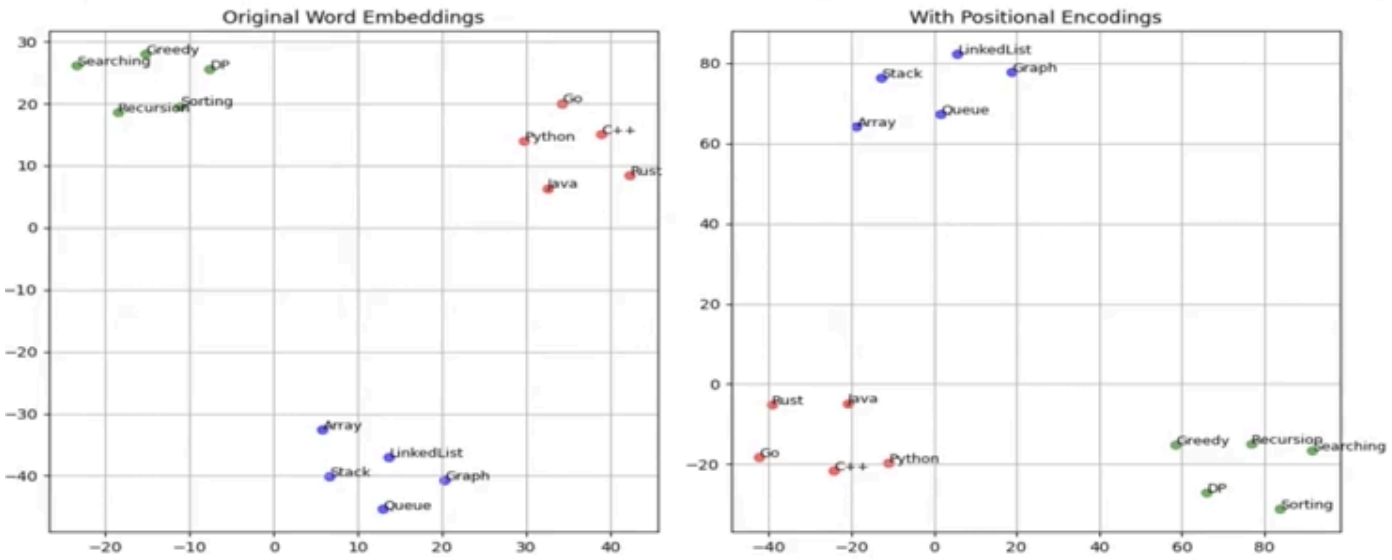
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- pos = Position of the token in the sequence (0, 1, 2, ...).
- i = The dimension index (half for sine, half for cosine).
- d = Total embedding dimension (e.g., 512 in original Transformer).
- 10000 = A scaling factor to ensure smooth variations.

✒️ Why Sinusoidal Functions?

Relative positioning was introduced so that nearby words have similar encodings. They **generalize to unseen sequence lengths** better than fixed position indices.

✒️ Original Embedding & Positional Encoding



✒️ Understanding the Relationship Between Input Embedding and Positional Encoding in k-Dimensional Space

In Transformers, the final input to the self-attention mechanism is the sum of two components: Final Embedding = Input Embedding + Positional Encoding

- **Input Embedding** - A k-dimensional dense vector for each token, capturing semantic meaning.
- **Positional Encoding (PE)** - A k-dimensional vector that injects positional information into the model.

📌 What are Query and Key Vectors?

Every token in the input sequence is transformed into three vectors:

- Query (Q) $\rightarrow$ Represents what this token is searching for in other tokens.
- Key (K) $\rightarrow$ Represents the properties of this token that determine whether it should be attended to.

These vectors are learned representations obtained from the original input embeddings.

$$Q = XW_Q, \quad K = XW_K$$

where:

- X is the **input embedding** matrix of size $(n \times d)$.
- W_Q, W_K are **learnable weight matrices** that transform embeddings into queries and keys.
- Q, K are the transformed matrices of size $(n \times d_k)$.

📌 How Are They Used in Self-Attention?

The core idea of self-attention is to compare queries (Q) and keys (K) across all tokens in the sequence. The attention score for a token pair (i, j) is calculated as:

$$\text{Score}(i, j) = Q_i \cdot K_j^T$$

This dot product:

- Measures the similarity between the query of token i and the key of token j.
- If Score(i, j) is high, token j is important for token i, meaning token i should attend more to token j.

📌 Example

Consider the sentence:

> "AI is powerful"

Step 1: Convert Each Token to Query & Key Vectors

Let's assume our embedding dimension is **4**, and we have the following transformed vectors:

Query Vectors (Q)

$$Q = \begin{bmatrix} 0.2 & 0.5 & 0.8 & 0.1 \\ 0.7 & 0.1 & 0.3 & 0.5 \\ 0.4 & 0.6 & 0.9 & 0.2 \end{bmatrix}$$

Key Vectors (K)

$$K = \begin{bmatrix} 0.3 & 0.7 & 0.2 & 0.6 \\ 0.5 & 0.2 & 0.9 & 0.4 \\ 0.8 & 0.3 & 0.6 & 0.7 \end{bmatrix}$$

Step 2: Compute Attention Scores ($Q \cdot K^T$)

To find how much attention each word pays to others, we compute the **dot product** between each query (Q) and all keys (K):

$$A = QK^T = \begin{bmatrix} (0.2, 0.5, 0.8, 0.1) \cdot K^T \\ (0.7, 0.1, 0.3, 0.5) \cdot K^T \\ (0.4, 0.6, 0.9, 0.2) \cdot K^T \end{bmatrix}$$

✒️ Why Do We Take Softmax of $Q{\cdot}KT$ in Self-Attention?

In the self-attention mechanism of Transformers, we compute the attention scores by taking the dot product of Query (Q) and Key (K) matrices:

$$A = Q \cdot K^T$$

However, this raw score matrix A needs to be transformed into a probability distribution. This is where Softmax comes in.

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

The **Softmax function** is defined as:

$$\text{softmax}(A_i) = \frac{e^{A_i}}{\sum_j e^{A_j}}$$

- **Convert Raw Scores into Probabilities** - The dot product produces unbounded values (positive or negative). Applying Softmax normalizes these values into a range [0,1] and ensures they sum to 1. This makes it possible to interpret them as relative importance scores.
- **Focus on Relevant Tokens** - A higher softmax value means more focus on that token. A lower softmax value means less focus. This enables dynamic attention, meaning the model learns which words matter more based on context.
- **Prevent Large Magnitude Explosions** - The dot product grows proportionally to the embedding dimension leading to large numbers. The division operation prevents the values from becoming too large, which would make softmax outputs very sharp (close to 0 or 1). This ensures a stable gradient flow during training.

📌 Example

Let's say we have a query-key similarity matrix:

$$A = \begin{bmatrix} 3 & 1 & 0 \\ 2 & 4 & 1 \\ 1 & 3 & 5 \end{bmatrix}$$

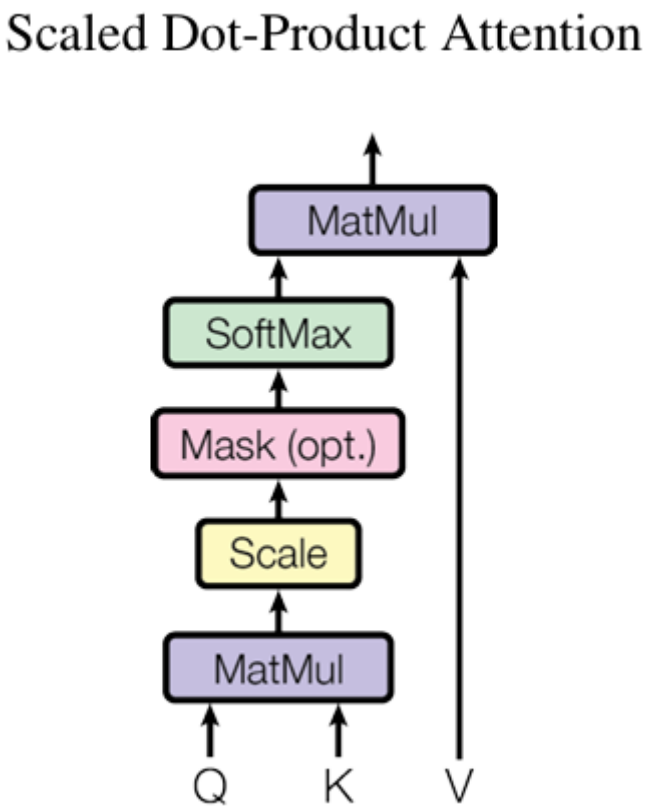
Without **softmax**, we have raw scores. After **applying softmax row-wise**, it transforms into:

$$\text{Softmax}(A) = \begin{bmatrix} 0.84 & 0.11 & 0.05 \\ 0.15 & 0.78 & 0.07 \\ 0.02 & 0.12 & 0.86 \end{bmatrix}$$

Interpretation:

- **Row 1:** Token 1 pays **84% of its attention** to itself, **11% to Token 2**, and **5% to Token 3**.
- **Row 2:** Token 2 pays **78% of its attention** to Token 2 and **only 7% to Token 3**.
- **Row 3:** Token 3 pays **86% of its attention** to Token 3.

📌 Scaled Dot-Product Attention



📌 What is the Value Vector (V) in Self-Attention?

In the self-attention mechanism of Transformers, every token in a sequence is mapped to three vectors:

- Query (Q) – Represents what a token is looking for in others.
- Key (K) – Represents what each token has to offer.
- Value (V) – Contains the actual information that will be used to generate the final output.

Unlike Query and Key, which are used to compute attention scores, Value (V) determines what content gets passed on to the next layer.

📌 How is the Value Vector Computed?

Similar to Query and Key, the Value vector is obtained by transforming the input embeddings:

$$V = XW_V$$

- X = Input embeddings of tokens ($n \times d$ matrix).
- W(V) = Learnable weight matrix for values ($d \times d(v)$).
- V = Value matrix of shape ($n \times d(v)$).

📌 Example

Let's assume we have a **sentence:**

> **"AI is powerful"**

Each token is mapped to Query, Key, and Value vectors.

Query Matrix (Q)

$$Q = \begin{bmatrix} 0.2 & 0.5 & 0.8 \\ 0.7 & 0.1 & 0.3 \\ 0.4 & 0.6 & 0.9 \end{bmatrix}$$

Key Matrix (K)

$$K = \begin{bmatrix} 0.3 & 0.7 & 0.2 \\ 0.5 & 0.2 & 0.9 \\ 0.8 & 0.3 & 0.6 \end{bmatrix}$$

Value Matrix (V)

$$V = \begin{bmatrix} 0.1 & 0.4 & 0.7 \\ 0.3 & 0.8 & 0.2 \\ 0.5 & 0.9 & 0.6 \end{bmatrix}$$

4. How Value Vectors Influence the Output

1. We **compute attention scores** using QK^T.
2. Apply **Softmax** to get a probability distribution.
3. Multiply these **attention weights with** V.

$$Z = \text{Attention Weights} \cdot V$$

✒️ Key Takeaways

- Query (Q) asks: "What am I looking for?"
- Key (K) answers: "Do I match what you're looking for?"
- Value (V) provides: "Here's the actual information you need!"

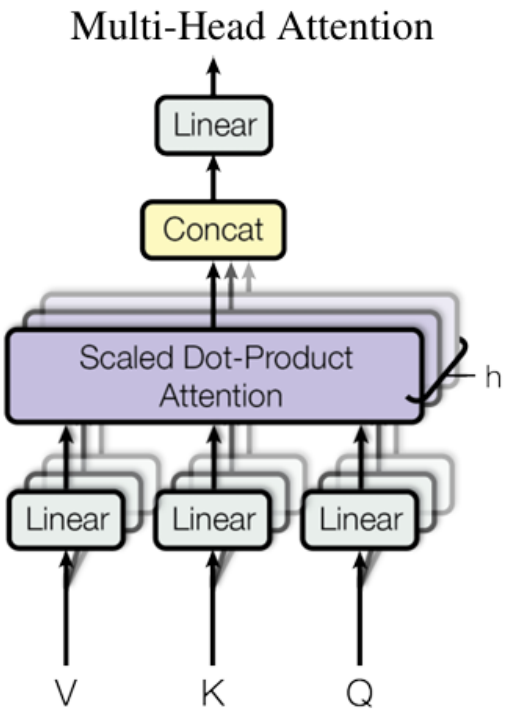
📌 Multi-Head Attention

Multi-Head Attention (MHA) is an extension of the self-attention mechanism used in Transformers (e.g., BERT, GPT, T5). Instead of using a single attention function, the model splits the input into multiple "heads", allowing it to learn different aspects of relationships in the data simultaneously.

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

✒️ Why Do We Need Multi-Head Attention?

A single self-attention mechanism can only capture one type of relationship between words at a time. However, language is complex, and different words relate to each other in multiple ways (e.g., syntactic, semantic).



✒️ Why is Transformer Architecture Known as an Encoder-Decoder Architecture?

The Transformer architecture (introduced in "Attention Is All You Need", 2017) is often referred to as an Encoder-Decoder Architecture because it consists of two main components:

- **Encoder** → Processes the input sequence and generates a meaningful representation (context).
- **Decoder** → Takes this representation and generates an output sequence (e.g., translated text).

This structure is similar to traditional sequence-to-sequence (seq2seq) models, which are commonly used for tasks like machine translation, text summarization, and question-answering.

💡 **Example Use Cases:**

Task	Encoder Input	Decoder Output
Translation	"I love AI"	"J'aime l'IA"
Summarization	"Transformers use self-attention..."	"Transformers rely on attention"
Question Answering	"What is AI?" + Context	"AI is..."

✒️ Steps

1. Input tokenization & embedding + Positional encoding to retain word order.
2. Multi-headed attention
    - Each token attends to every other token using a self-attention mechanism.
    - Compute Q, K and V matrices (learnt/training).
    - Different relationships - Multiple attention heads.
3. Normalization layer
4. Feedforward network - 2 fully connected layers (2 dense layers)
5. Normalization layer

Steps 2-5 are repeated. This is the overview of an Encoder.