**A. Embedding**

Before transformers can begin processing and understanding language, they must first convert raw input — usually a sequence of words, subwords, or characters — into a numerical form that the model can work with. This critical first step is known as **embedding**. It's the bridge between human-readable language and machine-understandable data. Without embeddings, a transformer model would not be able to interpret the semantic and syntactic information contained in natural language inputs.

**1. Why Embedding?**

Raw text (like "The cat sat on the mat") is meaningless to a machine. Computers operate on numbers, not letters or words. While we might see meaningful patterns in words, models like transformers only see sequences of tokens (like "The", "cat", "sat", etc.) that must be converted into numerical vectors.

Initially, a simple method like one-hot encoding was used in early NLP systems. However, one-hot vectors are sparse (mostly zeroes), don't scale well with vocabulary size, and lack any notion of similarity between words. For example, the one-hot vectors for "cat" and "dog" are completely different and show no indication that the words are semantically related. This is where **embedding** comes in.

**2. What Is an Embedding?**

An **embedding** is a learned representation of text in a continuous vector space where semantically similar words are mapped to nearby points. More formally, each token (a word or subword) is mapped to a dense, fixed-size vector of real numbers, typically with dimensions ranging from 128 to 1024.

For instance:

* "cat" → [0.12, -0.25, 0.88, ..., 0.04]
* "dog" → [0.10, -0.21, 0.83, ..., 0.06]

These numbers don’t mean much on their own, but collectively they capture useful properties of the token, including its meaning, grammatical role, and relationships to other words.

These vectors are stored in an **embedding matrix** a large lookup table that transforms token IDs into vectors. When the transformer processes an input, it uses this matrix to map each token to its corresponding embedding.

**3. Tokenization and Embedding Lookup**

Before embedding can happen, the input text needs to be **tokenized**. Tokenization is the process of splitting the text into smaller units usually words, subwords, or characters. Most modern transformer-based models like BERT or GPT use subword tokenization methods such as Byte Pair Encoding (BPE) or WordPiece.

Example:

* Input: "Transformers are amazing"
* Tokens: ["Transform", "##ers", "are", "amazing"]
* Token IDs: [10345, 2098, 2024, 12093]

These IDs are then used to index into the embedding matrix, producing a sequence of embedding vectors.

**4. Positional Embeddings**

One important thing to note is that transformers, unlike recurrent models, have **no inherent sense of order**. They process input as a set rather than a sequence, which means the model doesn’t know that "cat sat" is different from "sat cat" unless we explicitly encode positional information.

To solve this, transformers add **positional embeddings** to token embeddings. These positional vectors encode the position of each token in the sequence and are added (element-wise) to the token embeddings. This gives the model some notion of the order in which tokens appear.

**5. How Are Embeddings Learned?**

Initially, embeddings are just random vectors. During training, the transformer adjusts them using backpropagation to minimize the loss function (i.e., how wrong its predictions are). Over time, this allows the model to learn meaningful representations.

For example, during training on large datasets, the model learns that:

* "king" and "queen" are similar
* "Paris" and "France" are related
* "run" and "ran" have similar roots but indicate different tenses

This learned knowledge gets encoded in the embedding vectors.

In some architectures (like BERT), embeddings also include **segment embeddings**, which indicate whether a token belongs to sentence A or sentence B (useful in tasks like question-answering). These are also added to the token and positional embeddings.

So, the final embedding is often a **sum of three components**:

1. Token embedding
2. Positional embedding
3. Segment embedding (if applicable)

**6. Visualization and Intuition**

Imagine plotting all the token embeddings in 3D space (after dimensionality reduction). Words with similar meanings would cluster together — "man", "boy", "father" might be near each other, and far from unrelated words like "apple" or "table". These relationships help the model generalize and make sense of patterns in language.

**7. Importance of Good Embeddings**

High-quality embeddings are the foundation for the transformer’s performance. If two words with very different meanings are embedded similarly, the model may make mistakes. On the other hand, good embeddings allow the model to capture complex relationships, analogies, and syntactic roles.

Moreover, embeddings can be:

* **Static** (like GloVe or Word2Vec): Same embedding for a word regardless of context.
* **Contextual** (as in transformers): Embeddings change depending on the surrounding words. This is more powerful and allows the same word to take on different meanings in different contexts.

Example:

* "He went to the **bank** to withdraw cash." → "bank" = financial institution
* "She sat on the **bank** of the river." → "bank" = side of a river

In transformers, thanks to self-attention and deep layers, the embedding of "bank" is informed by the words around it.

**B. Self-Attention**

After converting input text into embeddings, the next fundamental component in a transformer model is **self-attention**. This mechanism enables the model to determine which words in a sentence are relevant to one another, regardless of their position. Self-attention is critical to the transformer's ability to understand context and capture relationships between words.

**1. Purpose of Self-Attention**

Self-attention allows the model to weigh the importance of each word in the input sequence relative to others. For every word, the model calculates how much it should "attend" to all other words when generating its contextual representation. This helps the model understand the meaning of a word based on its surrounding context.

For example, in the sentence:

“The animal didn’t cross the road because **it** was tired,”

the model needs to understand that “it” refers to “animal.” Self-attention enables this by assigning a higher weight to the word “animal” when processing the word “it.”

**2. How Self-Attention Works**

Each input token is projected into three different vectors:

* **Query (Q)**
* **Key (K)**
* **Value (V)**

These are computed by multiplying the token embeddings by learned weight matrices. The attention score between two tokens is calculated by taking the dot product of the query vector of one token with the key vector of another.

The scores are then normalized using the softmax function, producing **attention weights**. These weights determine how much focus the model should place on each word when computing the final output for a given token.

Finally, the output for each token is the weighted sum of the value vectors from all other tokens. This process is repeated for every token in the sequence, allowing each one to incorporate contextual information from the entire input.

**3. Multi-Head Attention**

In practice, transformers use **multi-head attention**, where the self-attention mechanism is applied multiple times in parallel using different sets of weight matrices. Each "head" captures different types of relationships (e.g., syntactic, semantic, positional). The results from all heads are concatenated and passed through a final linear transformation.

This allows the model to attend to various aspects of the input simultaneously, enriching its understanding of the language.

**4. Advantages of Self-Attention**

* **Global Context Understanding**: Each token has access to all other tokens in the input sequence.
* **Parallel Processing**: Unlike recurrent models, self-attention can process all tokens simultaneously, leading to faster training.
* **Handling Long-Range Dependencies**: The model can capture relationships between distant words effectively.

**5. Limitations**

Despite its effectiveness, self-attention has a computational cost that grows quadratically with sequence length, which can be a challenge for processing long texts. To address this, newer transformer variants introduce more efficient attention mechanisms.