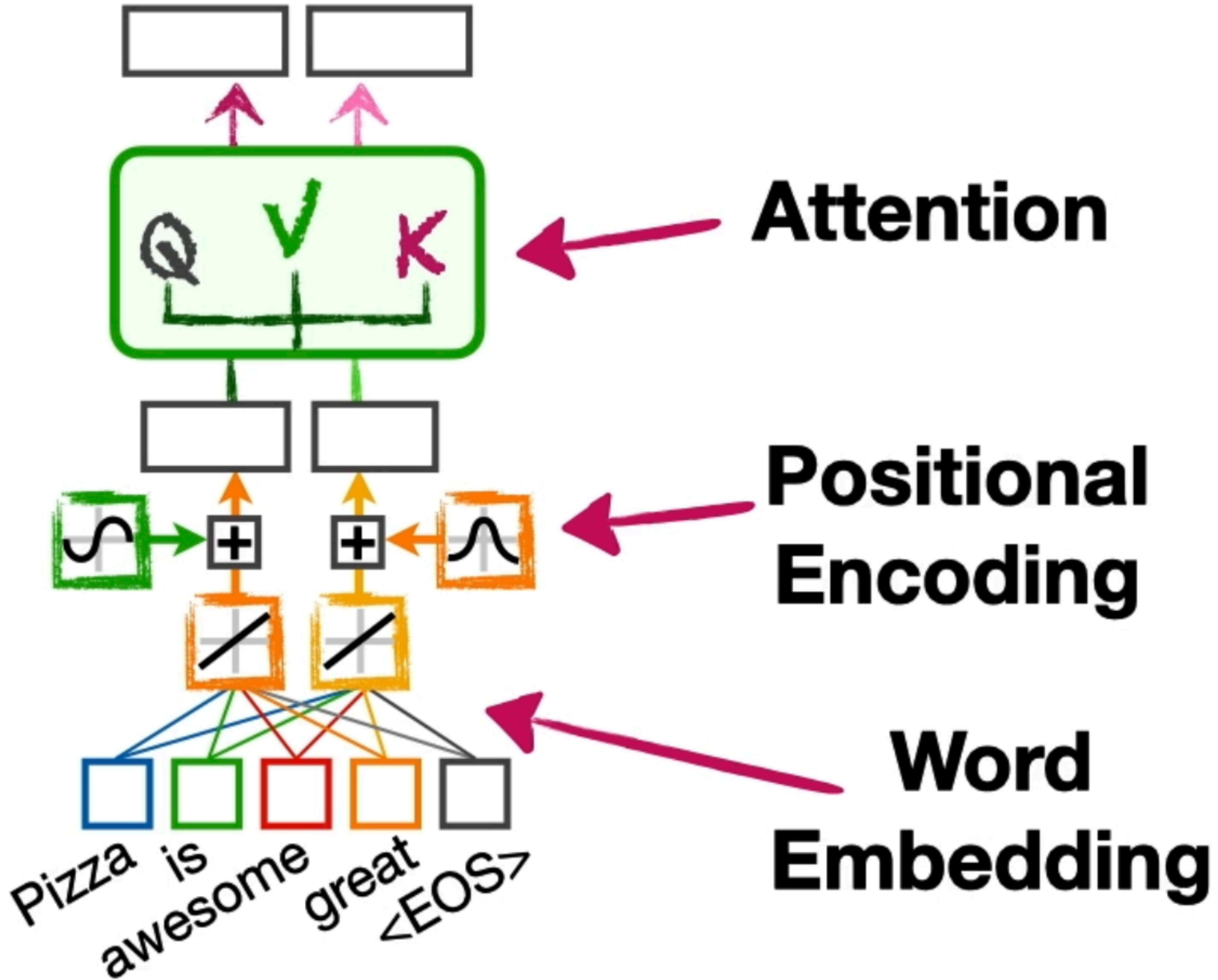
DeepLearning.AI (Transformerrs)

Aug_30

ChatGPT is built on the Transformer architecture.

Transformers are based on three key ideas:

1. **Token embeddings**
   Convert inputs (words, subwords, symbols, punctuation) into numeric vectors.
2. **Positional encoding**
   keep track of word order—for example:
   “Elahe eats pizza” ≠ “Pizza eats Elahe.”
3. **Self-attention**
   Each token attends to every token in the sentence (including itself), measuring relevance and building context-aware representations.



Example: Hotel Database Analogy

- Imagine a hotel has a database with two columns:
  Key: surname (e.g., *Elahe*)
  Value: room number (e.g., *203*)
- A guest says their name at reception:
  - What the receptionist hears and types in is the **query** (e.g., *Ilagi*).
  - The system compares this query against all stored **keys** (surnames in the list).
  - It finds the closest match and ranks possible candidates.
- Finally, the system retrieves the **value** (the room number) associated with the best-matching key.

👉 This is the same idea used in **attention mechanisms**:

- **Keys** = stored reference entries
- **Query** = what you're searching with
- **Values** = the information you want to retrieve

Attention Equation (Step by Step)

1. **Dot product similarity**
    - The dot product between a **query** and a **key** gives an **unscaled measure of similarity**.
    - This is closely related to **cosine similarity**, except cosine similarity normalizes for vector length.
2. **Scaling**
    - To avoid very large values when the key dimension (d_k) is high, we scale by \frac{1}{\sqrt{d_k}}
3. **Softmax**
    - Apply the **softmax function** to the scaled scores.
    - This converts them into a distribution that looks like percentages (all positive, sum to 1).
4. **Weighted sum of values**
    - Each score weights the corresponding **value vector**.
    - The final output is a **weighted combination** of values, where more relevant keys contribute more.

👉 In short:
Attention = Softmax((Query • Key) / $\sqrt{d_k}$) × Value

Important Parameters in PyTorch Attention Code

- **d_model**
  - Defines the size of the weight matrices used to create **queries, keys, and values**.
  - Represents the number of **embedding values per token**.
  - Example: In *Attention Is All You Need*, d_model = 512.
  - If d_model = 2, each token is represented by a 2-dimensional vector.
- **in_features & out_features** (in linear layers like nn.Linear)
  - in_features: number of **rows** in the weight matrix (*input dimension*).
  - out_features: number of **columns** in the weight matrix (*output dimension*).
  - Used in projection matrices **W_Q, W_K, W_V** for queries, keys, and values.
- **Bias term**
  - In the original Transformer paper, **no additional bias** is added when computing attention weights.
  - Some PyTorch implementations may include bias by default, but it’s not part of the original design.

👉 In summary:

- d_model = embedding dimension
- in_features / out_features = shape of projection matrices
- Bias = omitted in the original Transformer attention

Self-Attention Types in Transformers

- **Encoder-only Transformers (e.g., BERT)**
  - Use **full self-attention**: every token can attend to **all tokens** in the input.
  - Best for **understanding** tasks (classification, embeddings, etc.).
- **Decoder-only Transformers (e.g., GPT models)**
  - Use **masked self-attention**: each token can only attend to **itself and previous tokens**, not future ones.
  - This masking ensures the model generates text **sequentially** and doesn’t peek at words it hasn’t produced
  - Best for **generation** tasks.
- **Encoder–Decoder Transformers (e.g., original Transformer, T5)**
  - **Encoder:** full self-attention (context building).
  - **Decoder:** masked self-attention + cross-attention (decoder attends to encoder outputs).
  - Best for **translation, summarization, sequence-to-sequence tasks**.

…is that **Self-Attention**, can look at words before *and* after the word of interest...

…in contrast, **Masked Self-Attention** ignores the words that come after the word of interest.

The good news is that the only difference between the equation for **Self-Attention**…

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

…and the equation for **Masked Self-Attention**…

…is that we add a new matrix, ***M*** for **Mask**, to the scaled similarities.

$$MaskedAttention(Q, K, V, M) = SoftMax\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

Multi-head attention:

- Instead of doing this once, the model creates **multiple sets of Queries, Keys, and Values** using different learned weight matrices.
- Each "head" learns to capture **different kinds of relationships**.
  - Head 1 might focus on **subject–verb** connections.
  - Head 2 might focus on **long-distance dependencies**.
  - Head 3 might focus on **word order nuances**, etc.
- Each head produces its own attention output.
- These outputs are **concatenated** and passed through a final linear layer.

Why it's useful

- A single attention head might miss some patterns.
- With multiple heads, the model can **look at the sentence in parallel from different perspectives**.
- This gives the Transformer a much richer representation of meaning.

👉 In short:

Multi-head attention = many attention mechanisms running in parallel, each focusing on different aspects of the sequence, then combining results.