Lecture 11: The Importance of Positional Embeddings

1. The Missing Piece: Context and Order

While the previous lectures covered **Token Embeddings** (Step 3), which capture the semantic meaning of words (e.g., relating "cat" to "kitten"), this step alone is insufficient for language understanding.

- **The Problem:** Token embeddings capture *what* a word is, but not *where* it is.
- **The Example:** Consider two sentences: *"The cat sat on the mat"* vs. *"On the mat the cat sat"*.
    - In both sentences, the word "cat" has the same Token ID and, consequently, the exact same vector representation from the embedding layer.
    - However, the position of "cat" changes the sentence structure and potential meaning. Without positional information, the model sees "cat" identically in both cases.
- **The Solution:** We must inject additional information about the **order** of tokens into the model. This step is often viewed as "Step 3.5" in the data processing pipeline, resulting in the final **Input Embeddings**.

2. Types of Positional Embeddings

There are two primary methods to encode position, and the choice depends on the architecture (e.g., Transformer vs. GPT).

A. Absolute Positional Embedding

- **Mechanism:** A unique embedding vector is assigned to each specific position in the input sequence (e.g., a vector for Position 1, a different vector for Position 2).
- **Process:** This unique position vector is **added** to the token embedding vector.
    - *Formula:* Final Vector = Token Embedding (X) + Positional Embedding (Y).
    - *Result:* The word "cat" at position 2 will have a different final mathematical representation than "cat" at position 5, because the added positional vectors differ.
- **Usage:** This method is used by OpenAI's **GPT-3 and GPT-4**, as well as the original Transformer paper.
- **Optimization:** Unlike the original 2017 Transformer paper which used fixed mathematical formulas (sinusoidal/cosine functions), GPT models **learn** these positional values. They are initialized randomly and optimized via backpropagation during training, just like the token embeddings.

B. Relative Positional Embedding

- **Mechanism:** Instead of assigning a unique vector to a fixed slot (1, 2, 3), the model learns the **distance** or relationship between tokens.

- **Usage:** This is useful for very long sequences where the exact position matters less than how far apart specific words are from each other.

3. Hands-On Implementation (Dimensions & Broadcasting)

Dr. Dander emphasizes understanding the tensor dimensions to demystify the coding process. The implementation assumes a batch size of 8, a context length of 4, and an embedding dimension of 256.

- **Step 1: The Token Embedding Tensor**
  - We process a batch of data containing 8 sequences.
  - Each sequence has a context length of 4 tokens.
  - Each token is converted into a 256-dimensional vector.
  - **Resulting Tensor Shape:** ``.
- **Step 2: The Positional Embedding Layer**
  - We only need to encode positions for the maximum context length the model sees at one time.
  - **Rows:** Equal to the context length (4).
  - **Columns:** Equal to the embedding dimension (256). *Note: The dimension must match the token embedding for addition to work.*
  - **Resulting Tensor Shape:** ``.
- **Step 3: Creating Input Embeddings (Broadcasting)**
  - We add the Token Tensor `and the Position Tensor`.
  - **Broadcasting:** Python automatically duplicates (broadcasts) the `` position tensor 8 times to match the batch size of the token tensor.
  - *Logic:* The positional vector for "Position 1" is the same for *every* sequence in the batch.
  - **Final Output:** A tensor of shape `` that now contains both semantic and positional information.

4. Summary of the Data Pipeline

The lecture concludes by updating the workflow map to include this crucial step:

1. **Input Text** (Raw)
2. **Tokenization** (Text $\rightarrow$ Token IDs)
3. **Token Embeddings** (IDs $\rightarrow$ Vectors with Semantic Meaning)
4. **Positional Embeddings** (Injecting Order)
5. **Input Embeddings** (Token Vector + Position Vector $\rightarrow$ Final Input to LLM).

Analogy: The Musical Score

To understand why we add positional embeddings to token embeddings, imagine a **musical score**.

- **Token Embeddings are the Notes:** A "C-sharp" (the token) has a specific sound (semantic meaning). However, a pile of "C-sharp" notes thrown on a table is just noise; it has no melody.
- **Positional Embeddings are the Time Signature and Measure:** This tells the musician *when* to play the note.
- **The Combination:** When you place the "C-sharp" (Token) at the "start of the second measure" (Position), you create music.
- **The Result:** Just as the same note played at different times changes the melody, the same word ("cat") placed at different positions changes the sentence's meaning. The LLM needs both the "sound" of the word and its "timing" to understand the language.