Lecture 10: What are Token Embeddings?

1. The Place in the Workflow

- **Context:** This lecture focuses on **Step 3** of the Large Language Model (LLM) training pipeline.
- **The Pipeline:**
    1. **Input Text:** (e.g., "This is an example").
    2. **Tokenization:** Breaking text into individual units/tokens (Step 1).
    3. **Token IDs:** Converting tokens into specific integer IDs (Step 2).
    4. **Token Embeddings:** Converting Token IDs into vectors (Step 3),.
- **Terminology:** While often called "Word Embeddings," the term **"Token Embeddings"** is more accurate because modern tokenizers (like Byte Pair Encoding) handle sub-words and characters, not just whole words.

2. The Problem with Simple Numbers

To train a neural network, text must be converted into numbers. However, simple methods fail to capture meaning:

- **Random Integer Assignment:** If we arbitrarily assign "Cat" = 34 and "Kitten" = -13, the model sees no mathematical relationship between them, even though they are semantically related.
- **One-Hot Encoding:** Creating a massive vector of zeros with a single "1" for the specific word.
    - **Failure:** This also fails to encode semantic relationships. There is no way to know from a one-hot vector that "Dog" and "Puppy" are closer in meaning than "Dog" and "Banana".

3. The Solution: Vector Embeddings

The goal is to exploit the inherent information present in text—specifically that words carry meaning and related words should be mathematically closer to each other.

- **The Concept:** Every token is encoded as a **vector** in a high-dimensional space.
- **The "Feature" Analogy:** Dr. Dander explains this using a hypothetical 5-dimensional vector where dimensions represent features like "Has a tail," "Is eatable," or "Is a pet".
    - **Dog/Cat:** Both would have high values for "Has a tail" and "Is a pet." Their vectors would look similar.
    - **Apple/Banana:** Both would have high values for "Is eatable." Their vectors would look similar.
    - **Differentiation:** A "Dog" vector would be mathematically distant from a "Banana" vector because their feature values (e.g., "Is eatable" vs. "Has a tail") would effectively cancel out or differ significantly.

4. Semantic Arithmetic (King + Woman - Man = ?)

The lecture demonstrates the power of embeddings using a pre-trained "Word2Vec" model (trained on Google News data) to show that vectors essentially encode logic.

- **Vector Arithmetic:** If you take the vector for **King**, add the vector for **Woman**, and subtract the vector for **Man**, the resulting vector is mathematically closest to **Queen**,.
    - *Interpretation:* The model successfully captured the concept of gender (removing "masculinity" and adding "femininity") and royalty.
- **Distance as Meaning:** The magnitude of the difference between vectors indicates semantic similarity.
    - *Close:* "Nephew" and "Niece" (Difference $\approx$ 1.96).
    - *Far:* "Semiconductor" and "Earthworm" (Difference $\approx$ 5.67),.

5. Implementing the Embedding Layer (Technical Details)

How is this actually built in an LLM like GPT-2?

- **The Matrix Dimensions:** To create the embedding matrix, you need two numbers:
    1. **Vocabulary Size:** How many unique tokens exist? (GPT-2: 50,257).
    2. **Embedding Dimension:** How big is the vector for each token? (GPT-2 small: 768 dimensions).
- **The Embedding Matrix:** This results in a massive matrix of size **[50,257 $\times$ 768]**.
    - There is one row for every token ID.
    - Each row contains 768 weights representing that token's vector.
- **The Lookup Mechanism:**
    - The Embedding Layer is technically a **Lookup Table**,.
    - If the model receives Token ID `3`, it simply goes to **Row 3** of the matrix and retrieves those specific weights. It does not perform a calculation; it performs a retrieval,.

6. Training the Embeddings

- **Initialization:** When training begins, the embedding matrix is initialized with **small random values**. It does not know that "Cat" and "Kitten" are related yet,.
- **Optimization:** These weights are optimized via **backpropagation** during the LLM training process itself. As the model learns to predict the next word, it simultaneously adjusts the values in the embedding matrix so that semantically similar words end up with similar vectors,.

7. Embedding Layer vs. Linear Layer

The lecture notes a technical nuance regarding PyTorch implementation:

- **Equivalence:** An Embedding Layer produces the exact same output as a Neural Network Linear Layer (`nn.Linear`) that takes a one-hot encoded vector as input.
- **Efficiency:** We use `torch.nn.Embedding` (Lookup Table) instead of `torch.nn.Linear` because the Linear layer would involve multiplying a massive matrix by a vector of

mostly zeros (one-hot), which is computationally wasteful. The Lookup Table skips the zeros and grabs the specific row directly, making it much more efficient.

8. Next Step: Positional Embeddings

The lecture concludes by noting a limitation: Current token embeddings capture *meaning* but ignore *position*. "The cat sat on the mat" contains the same words as "The mat sat on the cat," so the embeddings would be identical despite the meaning changing. The next lecture will introduce **Positional Embeddings** to solve this.