Tokenization and Embeddings: Connecting Concepts

1 Introduction

In the realm of natural language processing (NLP) and multimodal AI, tokenization and embeddings represent two fundamental but distinct processes that work in tandem to transform human-interpretable information into machine-processable representations. This article explores the intricate relationship between these concepts, highlighting their similarities, differences, and the methods by which they are developed and trained.

2 Fundamental Concepts

2.1 What is Tokenization?

Tokenization is the process of breaking down text (or other data) into smaller units called tokens. These tokens serve as the atomic units of processing for machine learning models. In traditional NLP, tokens often correspond to words, subwords, or characters, while in multimodal contexts, tokens might represent image patches, audio segments, or video frames.

2.2 What are Embeddings?

Embeddings are dense vector representations of tokens in a continuous vector space. These numerical representations capture semantic relationships between tokens, enabling mathematical operations on otherwise discrete symbolic data. Embeddings transform sparse, high-dimensional one-hot encodings into dense, lower-dimensional vectors that encode meaning.

3 The Relationship Between Tokenization and Embeddings

3.1 Sequential Relationship

In typical machine learning pipelines, tokenization precedes embedding:

1. **Tokenization**: Raw input → Discrete tokens
2. **Embedding**: Discrete tokens → Continuous vector representations

```python
# Example of the sequential relationship
from transformers import BertTokenizer, BertModel
import torch

# Initialize tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Input text
text = "Tokenization precedes embedding in the NLP pipeline."

# Step 1: Tokenization
tokens = tokenizer(text, return_tensors="pt")
print(f"Tokenized IDs: {tokens['input_ids'][0][:10]}...")

# Step 2: Embedding
with torch.no_grad():
    embeddings = model(**tokens).last_hidden_state

print(f"Embedding shape: {embeddings.shape}")
print(f"First token embedding (CLS): {embeddings[0][0][:5]}...")
```

3.2 Conceptual Similarities

1. **Representation Learning**: Both tokenization and embedding are forms of representation learning, transforming raw data into more useful formats.
2. **Dimensionality Management**: Both manage dimensionality—tokenization by discretizing continuous input, embeddings by compressing sparse representations.
3. **Contextual Adaptation**: Modern approaches to both tokenization and embedding are increasingly context-aware.
4. **Transferability**: Both can be transferred across tasks and domains.

3.3 Key Differences

1. **Nature of Representation**:
   - Tokenization: Discrete, symbolic representation
   - Embeddings: Continuous, numerical representation

2. **Information Density**:
    - Tokenization: May lose information through segmentation
    - Embeddings: Compress information while preserving semantic relationships
3. **Mathematical Operations**:
    - Tokenization: Limited mathematical operations possible
    - Embeddings: Support rich mathematical operations (distance, similarity, arithmetic)
4. **Reversibility**:
    - Tokenization: Often reversible (tokens → original text)
    - Embeddings: Generally irreversible (vectors → exact tokens)

```python
# Demonstrating reversibility difference
from transformers import BertTokenizer
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Original text
text = "Embeddings capture semantic relationships."

# Tokenization is reversible
tokens = tokenizer.encode(text)
recovered_text = tokenizer.decode(tokens)
print(f"Original: {text}")
print(f"Recovered: {recovered_text}")

# Embeddings are not reversible
# Let's create a simple embedding space for demonstration
word_to_embedding = {
    'embeddings': np.array([0.1, 0.2, 0.3]),
    'capture': np.array([0.4, 0.5, 0.6]),
    'semantic': np.array([0.7, 0.8, 0.9]),
    'relationships': np.array([0.15, 0.25, 0.35])
}

# If we have a new vector, we can find the closest word but not recover exactly
new_vector = np.array([0.12, 0.22, 0.32])
similarities = {word: cosine_similarity([vec], [new_vector])[0][0]
                for word, vec in word_to_embedding.items()}
```

```
most_similar = max(similarities.items(), key=lambda x: x[1])
print(f"New vector is most similar to: {most_similar[0]}
    with similarity {most_similar[1]:.4f}")
```

4 Development and Training Approaches

4.1 Tokenizer Development

4.1.1 Rule-based Tokenizers

The simplest tokenizers are rule-based, using predefined patterns:

```
# Simple rule-based tokenizer
def simple_tokenizer(text):
    # Remove punctuation and split on whitespace
    import re
    text = re.sub(r'[^\w\s]', '', text)
    return text.split()

text = "Hello, world! How are you today?"
tokens = simple_tokenizer(text)
print(tokens)  # ['Hello', 'world', 'How', 'are', 'you', '
    today']
```

Rule-based tokenizers require no training but lack adaptability to new domains or languages.

5 Mathematical Foundations of Embeddings

5.1 Word2Vec Skip-gram Model

The Skip-gram model maximizes the probability of context words given a target word:

$$\mathcal{L} = \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq 0} \log P(w_{t+j}|w_t)$$

where:

- c is the context window size
- w_t is the target word
- w_{t+j} are context words

The probability is computed using softmax:

$$P(w_O|w_I) = \frac{\exp({v'_{w_O}}^T v_{w_I})}{\sum_{w \in V} \exp({v'_w}^T v_{w_I})}$$

where v_w and v'_w are the input and output vectors for word w.

5.2 GloVe (Global Vectors)

GloVe learns word vectors by minimizing:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

where:

- X_{ij} is the co-occurrence count between words i and j
- $f(x)$ is a weighting function
- w_i, $\tilde{w}_j$ are word vectors
- b_i, $\tilde{b}_j$ are bias terms

5.3 BERT's Contextual Embeddings

BERT uses masked language modeling with attention:

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

The final embedding for token i is:

$$h_i^L = \text{LayerNorm}(\text{FFN}(\text{MultiHead}(h_i^{L-1})))$$

where:

- h_i^L is the embedding at layer L
- FFN is a feed-forward network
- MultiHead combines multiple attention heads

5.4 Cross-modal Contrastive Learning

For multimodal embeddings, the contrastive loss is:

$$\mathcal{L} = -\log \frac{\exp(s(x, y^+)/\tau)}{\sum_{y \in Y} \exp(s(x, y)/\tau)}$$

where:

- $s(x, y)$ is the similarity between embeddings
- y^+ is the positive (matching) example
- τ is the temperature parameter

Algorithm (Contrastive Learning):

1. For each batch:
    - Generate embeddings for all modalities
    - Compute similarities between pairs
    - Update embeddings to maximize similarity of matching pairs
    - While maintaining distance between non-matching pairs

5.5 Dynamic Token Embeddings

Adaptive token embeddings use a gating mechanism:

$$e_t = g_t \odot e_t^{\text{static}} + (1 - g_t) \odot e_t^{\text{dynamic}}$$

where:

- g_t is a context-dependent gate
- e_t^{static} is the pre-trained embedding
- e_t^{dynamic} is the context-dependent embedding

Algorithm (Dynamic Embedding):

1. For each token t:
    - Compute context representation
    - Generate dynamic embedding
    - Calculate gate values
    - Combine static and dynamic components