Fares Ahmed Moustafa

F.ahmed2270@nu.edu.eg

Task 1 — LLMs as Foundational NLP Components

1.1 The Theoretical Shift

1) LLM as Vectorizer / Embedder

A. Contextual vs. Static Embeddings

Static embeddings (Word2Vec, GloVe)

- Each word type has **one fixed vector**.
- Cannot differentiate senses of **polysemous** words.

Example:

- "I deposited money in the bank."
- "The boat reached the bank of the river."

→ Word2Vec produces the **same vector** for *"bank"* in both sentences.

Contextual embeddings (BERT, GPT-style models)

- The embedding depends on the **entire sentence**.
- Different senses are captured automatically.

Example:

- "I deposited money in the bank." → *financial meaning*
- "The boat reached the bank of the river." → *geographic meaning*

→ BERT produces **two different vectors** for *"bank"*.

B. Common Techniques to Create Sentence/Document Embeddings

1. [CLS] Token Embedding

- Use the final hidden state of the **[CLS]** token.
- Built-in summary representation for classification.

2. Mean Pooling Over Token Embeddings

- Compute contextual vectors for all tokens, then **average them**.
- Simple, stable, widely used.

3. Specialized Models (Sentence-BERT, SimCSE)

- Fine-tuned using **contrastive / siamese** objectives.
- Produces embeddings optimized for **semantic similarity** tasks.

---

2) LLM as Tokenizer

A. Subword Tokenization (BPE, WordPiece)

Why needed:

Word-level tokenizers struggle with:

- Huge vocabulary size
- Rare words
- Misspellings
- Morphological variations
- **OOV (Out-of-Vocabulary) words**

Subword tokenizers split rare words into meaningful components:

Examples:

- bioinformatics → **bio + informatics**
- unbelievability → **un + believ + ability**

✔ Ensures **every word** is representable → solves OOV.

B. Common Special Tokens

- **[CLS]** — Classification / sentence-level representation
- **[SEP]** — Separates sequences (e.g., question | answer)
- **[PAD]** — Padding token
- *(Others: [MASK], ,)*

1.2 Performance and Application

1) Trade-offs: LLM Embeddings vs. Traditional Methods

A. Accuracy / Semantic Understanding

LLM embeddings (Sentence-BERT, BERT):

- Capture **context**, **semantics**, **polysemy**
- Excellent for semantic search, similarity, paraphrase detection

Traditional (TF-IDF / GloVe):

- Lexical, not semantic
- Good when exact words matter

B. Computational Cost

LLMs:

- Heavy computation per sentence
- Often require **GPUs**

TF-IDF / GloVe:

- Very fast
- CPU-friendly

C. Memory Footprint

LLMs:

- Dense vectors (256–1024 dims) → **large storage**

TF-IDF:

- Sparse vectors → **memory efficient**

D. Interpretability

TF-IDF / GloVe:

- Interpretable weights

LLMs:

- Dense latent vectors → hard to interpret

---

2) Real-World Application Where LLM Embeddings Excel

Semantic Search / Document Retrieval

Example: enterprise knowledge base search.

Why LLM embeddings are necessary:

Users phrase queries in ways that **do not share keywords** with documents.

LLM embeddings:

- Capture **semantic meaning**
- Retrieve relevant documents even with **low word overlap**

Example:

Query: "set OAuth for API"
Retrieved: "configure token-based authentication"

→ TF-IDF would fail unless exact keywords match. """

Task 2: Practical Application Examples

Tokenization and Encoding

```python
from transformers import BertTokenizer

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

text = "The algorithm utilizes unigrammar to parse text. This method
improves efficiency."

encoding = tokenizer(text, return_tensors='pt')

tokens = tokenizer.tokenize(text)
token_ids = encoding["input_ids"]
attention_mask = encoding["attention_mask"]

print("Tokens:", tokens)
print("\nToken IDs:", token_ids)
print("\nAttention Mask:", attention_mask)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/
_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your
settings tab (https://huggingface.co/settings/tokens), set it as
secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to
access public models or datasets.
  warnings.warn(
```

{"model_id":"61fe8b39724744b184de3f582154829a","version_major":2,"vers
ion_minor":0}

{"model_id":"2459de723f2b44cfa49d97fe502cb0e6","version_major":2,"vers
ion_minor":0}

{"model_id":"c18af95b743544f6a696ffa8d1348a1e","version_major":2,"vers
ion_minor":0}

{"model_id":"fe17c5a5cf18410d9836c735ecbcba85","version_major":2,"vers
ion_minor":0}

```
Tokens: ['the', 'algorithm', 'utilizes', 'un', '##ig', '##ram',
'##mar', 'to', 'par', '##se', 'text', '.', 'this', 'method',
'improves', 'efficiency', '.']

Token IDs: tensor([[  101,  1996,  9896, 21852,  4895,  8004,  6444,
7849,  2000, 11968,
          3366,  3793,  1012,  2023,  4118, 24840,  8122,  1012,
102]])

Attention Mask: tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1]])
```

Demonstration

1 Token list

- The tokenizer produced:

['the', 'algorithm', 'utilizes', 'un', '##ig', '##ram', '##mar', 'to', 'par', '##se', 'text', '.', 'this', 'method', 'improves', 'efficiency', '.']

This is the **subword tokenization** of your input text.
Each element is a token that BERT recognizes or splits from unknown words.

---

2 How the tokenizer handled “unigrammar”

The rare word `unigrammar` is **not in BERT’s vocabulary**, so the WordPiece tokenizer splits it into subword tokens:

"un" + "##ig" + "##ram" + "##mar"

- `"un"` → first subword
- `"##ig"` → continuation of previous subword
- `"##ram"` → continuation
- `"##mar"` → final continuation

This ensures **no unknown token ([UNK]) is used**, allowing BERT to represent rare words via combinations of known subwords.

---

3 Difference between Token IDs and Attention Mask

Component	Meaning
Token IDs	`[101, 1996, 9896, 21852, 4895, 8004, 6444, ... , 102]`
Numerical representation of each token in the BERT vocabulary; input to the model.	
Attention Mask	`[1, 1, 1, 1, 1, 1, 1, ...]`
Binary mask indicating which tokens are real input tokens (1)	

Component	Meaning
vs padding tokens (0). Ensures the model ignores padding during attention computation.	

Contextual Vectorization

```python
import torch
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertModel.from_pretrained("bert-base-uncased")

sentA = "The salesman gave a strong pitch for the new product."
sentB = "The baseball player threw the final pitch of the game."

inputsA = tokenizer(sentA, return_tensors="pt")
inputsB = tokenizer(sentB, return_tensors="pt")

with torch.no_grad():
    outputsA = model(**inputsA)
    outputsB = model(**inputsB)

embA = outputsA.last_hidden_state[0]
embB = outputsB.last_hidden_state[0]

tokensA = tokenizer.tokenize(sentA)
tokensB = tokenizer.tokenize(sentB)

idxA = tokensA.index("pitch")
idxB = tokensB.index("pitch")

vecA = embA[idxA]
vecB = embB[idxB]

print("Vector for 'pitch' in Sentence A (first 10 dims):", vecA[:10])
print("Vector for 'pitch' in Sentence B (first 10 dims):", vecB[:10])
```

{"model_id":"7ad2d7628ee24f458aea660dfe4389db","version_major":2,"version_minor":0}

```
Vector for 'pitch' in Sentence A (first 10 dims): tensor([-0.2971, -
0.3703,  0.3951,  0.0531, -0.3009, -0.1872, -0.2699,  0.5252,
        -0.1097, -0.3569])
Vector for 'pitch' in Sentence B (first 10 dims): tensor([-0.1211, -
0.5794,  0.0948,  0.3067, -0.5679,  0.1957,  0.7515,  0.4390,
         0.2467,  0.1445])
```

```
The two vectors for 'pitch' are different because BERT produces
context-dependent embeddings.
In Sentence A, 'pitch' means a sales presentation, while in Sentence
B, 'pitch' refers to a baseball throw.
The surrounding words influence the embedding, capturing the different
meanings.

This contextual difference is important for downstream tasks like:
- Text classification: distinguishing marketing vs sports content
- Semantic clustering: grouping words by meaning
- Information retrieval: ensuring searches return contextually
relevant results
```

- **The two vectors for 'pitch' are different because BERT produces context-dependent embeddings.**
- **In Sentence A, 'pitch' means a sales presentation, while in Sentence B, 'pitch' refers to a baseball throw.**

The surrounding words influence the embedding, capturing the different meanings.

This contextual difference is important for downstream tasks like:

- Text classification: distinguishing marketing vs sports content
- Semantic clustering: grouping words by meaning
- Information retrieval: ensuring searches return contextually relevant results