```
#####################################################################
#                                   #
# CS435 Generative AI: Security, Ethics and Governance      #
#                                   #
# Instructor: Dr. Adnan Masood                  #
# Contact:   adnanmasood@gmail.com                  #
#                                   #
# Notebook is MIT Licensed                  #
#####################################################################
```

Teaching Tokenization, Vectors, and Embeddings in LLMs

Introduction

This Jupyter Notebook is designed to explain the concepts of **tokenization**, **vectors**, and **embeddings** in large language models (LLMs). We will explore these topics from five levels of detail (from a very simple, intuitive explanation all the way to a PhD-level perspective). We'll also cover the intuition, brief history, relevant mathematical foundations, illustrative examples, code demonstrations, a step-by-step building of the technology from scratch, real-world applications, and a glossary of terms.

Please **run each cell** in this notebook sequentially to follow the entire lesson.

1. Building an Intuitive Understanding

- **Tokenization**: Splitting long text into small pieces (like breaking a sentence into words or smaller parts) so that a computer can understand it more easily.
- **Vectors**: Numbers in a list that help us describe and compare words and their meanings.
- **Embeddings**: Special lists of numbers that capture the meaning of words or sentences so that a computer can understand them better.
- **Tokenization**: Cutting sentences into the smallest chunks (tokens). These can be whole words or even parts of words, depending on the method.
- **Vectors**: Think of them like coordinates on a map. If you want to show how a word relates to other words, you place it in a space with coordinates (numbers) to find words that are close in meaning.
- **Embeddings**: A way of placing words (or text) in a multi-dimensional space. Words with similar meanings end up near each other in this space, helping computers figure out relationships between words.
- **Tokenization**: In Natural Language Processing (NLP), tokenization can be **word-based** (split on whitespace), **subword-based** (e.g., Byte Pair Encoding/WordPiece), or **character-based**. This helps handle unknown or complex words.
- **Vectors**: Vectors are sequences of numbers representing some property. In NLP, vectors often represent how frequently words appear, or how they might cluster in meaning.
- **Embeddings**: Vector representations learned by neural networks. They reduce high-dimensional text data into smaller, more meaningful representations. Popular methods include **Word2Vec**, **GloVe**, and **Transformer-based embeddings** (e.g., from BERT or GPT).
- **Tokenization**: Advanced tokenization strategies use subword tokenizers (like Byte Pair Encoding in GPT or WordPiece in BERT) that allow efficient handling of rare words or new words ("unseen tokens"). This avoids issues like large vocabulary sizes.
- **Vectors**: Vectors in NLP can be dense (like embeddings) or sparse (like TF-IDF). Dense embeddings capture semantic relationships, while sparse vectors capture raw frequency-based relationships.
- **Embeddings**: They are learned by optimizing an objective that places words (or tokens, subwords, sentences) with similar contexts close together in the embedding space. Contextual embeddings (BERT, GPT, etc.) also consider neighboring words, capturing polysemy (words with multiple meanings) more effectively.
- **Tokenization**: Strategies are informed by information theory (maximizing token coverage while minimizing the total vocabulary). Byte-level BPE or SentencePiece are typical for large-scale LLMs. The design affects downstream performance on tasks.
- **Vectors & Embeddings**:
    - Embeddings can be static (Word2Vec) or contextual (BERT, GPT). Contextual embeddings incorporate attention mechanisms that learn dynamic representations based on the entire sentence.
    - Mathematical foundation often involves matrix factorization (GloVe) or neural predictive tasks (Word2Vec skip-gram, CBOW). Transformers rely on self-attention layers and multi-head attention.
    - Substantial lines of research focus on embedding interpretability, debiasing, and generalization in multi-lingual or multimodal spaces.

---

2. Intuition Behind Tokenization, Vectors, and Embeddings

- **Tokenization**: Imagine reading a book one letter at a time (very slow) versus reading a book word by word (faster and more meaningful). Tokenization helps a computer chunk text into manageable pieces that still preserve meaning.
- **Vectors**: If you want to measure how similar two words are, you need a way to compare them. By converting them into a list of numbers (a vector), we can compute distances (or similarities) between the vectors.
- **Embeddings**: The better the numbers in the vector capture the word's meaning, the more accurate the computer's understanding. Embeddings are these carefully learned numbers, making it so that words like "king" and "queen" end up near each other, while "apple" is farther away.

---

3. Brief History & Underlying Technology

- **Early NLP** used simple bag-of-words or TF-IDF vectors (large sparse vectors). These worked but didn't capture semantic relationships well.
- **Word2Vec (2013)** by Mikolov et al. introduced predictive-based embeddings, popularizing the idea that "you shall know a word by the company it keeps."
- **GloVe (2014)** by Stanford researchers used matrix factorization for global word co-occurrences.
- **Contextual Embeddings (ELMo, BERT, GPT, Transformers)** overcame the limitation of a single embedding per word by incorporating context using attention.
- **Modern LLMs** (e.g., GPT-3, GPT-4, BERT variants) use tokenization with subword approaches plus self-attention to learn highly sophisticated representations.

---

4. The Math Behind It

Word2Vec Skip-Gram Example

You try to predict the *context* words given a *target* word. If your target word is w_t, and the context words are w_{t-1},w_{t+1}$, etc., then you want: $$ P(w_{t-1}, w_{t+1}, ... | w_t) $$

We parameterize w_t with vectors (embeddings), say v_{w_t} for the input (target) and u_{w_c} for the output (context). We maximize the log-likelihood:

$$\max \sum_{t} \sum_{c \in \text{context}(t)} \log P(w_c | w_t)$$

And typically:

$$P(w_c | w_t) = \frac{\exp(u_{w_c}^T v_{w_t})}{\sum_{w'} \exp(u_{w'}^T v_{w_t})}$$

When training, we adjust v_{w_t} and u_{w_c} so that words that appear together frequently will have more similar vectors.

---

5. Intuitive Example + Example Calculations with Code

We'll do a small demonstration of how to turn words into vectors using a simple approach with Python.

Illustrative Example with Simple Counting

- **Words**: ["cat", "dog", "banana"]
- We'll create a simple **co-occurrence** matrix: how many times does each word appear near each other?
    - Suppose "cat" and "dog" appear together 2 times
    - "cat" and "banana" appear together 0 times
    - "dog" and "banana" appear together 1 time

This is overly simplistic, but let's see how it might look in code.

```
# Mock example: constructing a tiny co-occurrence matrix
import numpy as np

words = ["cat", "dog", "banana"]
word_to_index = {word: i for i, word in enumerate(words)}

# Suppose these are our co-occurrence counts
co_occurrences = {
```

```
    ("cat", "dog"): 2,
    ("dog", "cat"): 2,
    ("cat", "banana"): 0,
    ("banana", "cat"): 0,
    ("dog", "banana"): 1,
    ("banana", "dog"): 1
}

matrix = np.zeros((len(words), len(words)))

for (w1, w2), count in co_occurrences.items():
    i = word_to_index[w1]
    j = word_to_index[w2]
    matrix[i,j] = count

print("Co-occurrence Matrix:")
print(matrix)
```

```
Co-occurrence Matrix:
[[0. 2. 0.]
 [2. 0. 1.]
 [0. 1. 0.]]
```

Example Calculations (Weights, Bias, etc.)

- In a neural network, each **word** is transformed into a **weight vector** (its embedding). We might also have biases for certain layers, but typically in embedding layers, we just store the vectors.
- In advanced models, the token embedding is part of a bigger network with attention weights and biases.

---

6. Step-by-Step Example of Creating a Tiny Embedding from Scratch

We'll do a very simplified version of a **Word2Vec-like** process.

Step-by-Step Outline

1. **Collect text**: We'll use a tiny sample corpus.
2. **Tokenize**: We'll split it into words.
3. **Build vocabulary**: Identify all unique tokens.
4. **Initialize embeddings**: Randomly assign each token a small vector of numbers.
5. **Define context window**: For each word, look at its neighbors.
6. **Predict context**: Train the embeddings by predicting which words appear around the target word.
7. **Update embeddings**: Use gradient descent to nudge embeddings for correct predictions.

We'll do a miniature version below with a made-up dataset.

```
# Step-by-step tiny example of a Word2Vec-like approach
import numpy as np
import random

# 1. Tiny sample corpus
corpus = [
    "i love dogs",
    "i love cats",
    "cats and dogs are friends"
]

# 2. Tokenize
tokens = [sentence.split() for sentence in corpus]
all_tokens = [word for sent in tokens for word in sent]

# 3. Build vocabulary
vocab = list(set(all_tokens))
vocab_size = len(vocab)
word_to_idx = {w:i for i, w in enumerate(vocab)}
idx_to_word = {i:w for w,i in word_to_idx.items()}

# 4. Initialize embeddings
embedding_dim = 5  # let's pick 5 for illustration
embeddings = np.random.rand(vocab_size, embedding_dim)

# 5. Define context window
window_size = 2

def get_context_pairs(tokens, window_size):
```

```
    pairs = []
    for sent in tokens:
        for i, word in enumerate(sent):
            target_idx = word_to_idx[word]
            start = max(i - window_size, 0)
            end = min(i + window_size + 1, len(sent))
            for j in range(start, end):
                if j != i:
                    context_word = sent[j]
                    context_idx = word_to_idx[context_word]
                    pairs.append((target_idx, context_idx))
    return pairs

pairs = get_context_pairs(tokens, window_size)

print("Vocabulary:", vocab)
print("Index mapping:", word_to_idx)
print("Initial Embeddings Shape:", embeddings.shape)
print("Context Pairs (target_idx, context_idx):")
print(pairs[:10], "...")
```

```
Vocabulary: ['love', 'are', 'and', 'dogs', 'i', 'cats', 'friends']
Index mapping: {'love': 0, 'are': 1, 'and': 2, 'dogs': 3, 'i': 4, 'cats': 5, 'friends': 6}
Initial Embeddings Shape: (7, 5)
Context Pairs (target_idx, context_idx):
[(4, 0), (4, 3), (0, 4), (0, 3), (3, 4), (3, 0), (4, 0), (4, 5), (0, 4), (0, 5)] ...
```

6.1. (Optional) Training Loop

A full training loop would do something like:

1. For each `(target_idx, context_idx)` pair:
2. Compute similarity (e.g., dot product) between `embeddings[target_idx]` and `embeddings[context_idx]`.
3. Compare with the desired outcome (we want it to be high if it's a correct pair).
4. Backpropagate and update weights.

We'll do a toy training iteration with a simplistic approach.

```
import math

# We'll define a simple training approach with a dot product + naive gradient
learning_rate = 0.01
def train_one_epoch(pairs, embeddings, learning_rate=0.01):
    for target_idx, context_idx in pairs:
        # Dot product
        v_target = embeddings[target_idx]
        v_context = embeddings[context_idx]

        # For simplicity, let's say we want dot product to be close to 1 for real pairs
        # error = 1 - (v_target dot v_context)
        dot = np.dot(v_target, v_context)
        error = 1.0 - dot

        # gradient wrt v_target = -error * v_context
        grad_target = -error * v_context
        grad_context = -error * v_target

        # update embeddings
        embeddings[target_idx] -= learning_rate * grad_target
        embeddings[context_idx] -= learning_rate * grad_context

for epoch in range(5):
    train_one_epoch(pairs, embeddings, learning_rate)

print("Embeddings after 5 epochs of naive training:")
print(embeddings)
```

```
Embeddings after 5 epochs of naive training:
[[0.11417021 0.4582453  0.64568259 0.60502272 0.26755421]
 [0.52797195 0.87401378 0.00862019 0.87433186 0.693066  ]
 [0.1001251  0.37867523 0.0379155  0.3306993  0.71148489]
 [0.31325816 0.94913331 0.56535772 0.12093341 0.07828871]
 [0.32606061 0.81851743 0.07463199 0.42005152 0.40715168]
 [0.33973293 0.37842223 0.30000209 0.86412263 0.63203676]
 [0.72580473 0.61781131 0.66487452 0.48082485 0.19365697]]
```

This is a very crude and simplistic version, but it illustrates how embeddings get updated toward a representation where words co-occurring often end up having a higher dot product.

---

7. What Illustrative Problem Does This Solve?

Example

Text similarity: If we can embed sentences, we can measure how similar two sentences are by comparing their vectors. For instance, “I love dogs” and “I like puppies” might have very close embeddings, indicating similar meaning.

---

8. Real-World Problems Solved by This Technology

- **Search/Information Retrieval**: If you type a query, the system can embed the query and find documents with similar embeddings.
- **Recommendation Systems**: Embeddings can help match user interests to items.
- **Chatbots & Virtual Assistants**: LLMs use embeddings to generate context-aware, coherent responses.

---

9. How to Solve a Real-World Problem Using This Tech

1. **Identify your text data** (e.g., a collection of reviews, articles, or queries).
2. **Tokenize** using a robust tokenizer (e.g., from Hugging Face).
3. **Get embeddings** for each piece of text (words, sentences, paragraphs) using a pre-trained model or train your own.
4. **Perform your task** (e.g., retrieve similar documents, cluster text by topic, power a Q&A chatbot) using vector operations like **cosine similarity**.

---

10. Questions to Illustrate the Use of This Tech

Here are some example questions:

1. **How do you measure similarity between two word embeddings?**
2. **What happens if a word never appears in training?**
3. **How do subword tokenizers solve the out-of-vocabulary problem?**
4. **What is the role of attention in creating contextual embeddings?**
5. **Can embeddings be updated after deployment?**

Answers and Illustrative Code Examples

Q1: How do you measure similarity between two word embeddings?

- Common approach: **Cosine similarity**. We compute:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \, ||\mathbf{v}||}$$

```
# Example: Cosine similarity
def cosine_similarity(u, v):
    return np.dot(u, v) / (np.linalg.norm(u)*np.linalg.norm(v) + 1e-7)

# Let's compare 'i' and 'love' from our toy embeddings (if they exist)
if "i" in vocab and "love" in vocab:
    i_embed = embeddings[word_to_idx["i"]]
    love_embed = embeddings[word_to_idx["love"]]
    sim = cosine_similarity(i_embed, love_embed)
    print("Cosine similarity between 'i' and 'love':", sim)
else:
    print("'i' or 'love' not in vocab!")
```

```
Cosine similarity between 'i' and 'love': 0.7483117703055256
```

Q2: What happens if a word never appears in training?

- If a word (token) never appears, the model can’t learn an embedding directly. **Out-of-vocabulary** words can be handled with **subword tokenizers** or a special "unknown" token embedding.

Q3: How do subword tokenizers solve the out-of-vocabulary problem?

- They break a rare word into **subword units**, so even if the entire word is unseen, the pieces likely appear in training.

Q4: What is the role of attention in creating contextual embeddings?

- **Attention** weighs how important each token is to every other token in a sentence, leading to embeddings that reflect the context. Words are interpreted differently depending on surrounding words.

Q5: Can embeddings be updated after deployment?

- Yes. **Continual learning** or further fine-tuning can update embeddings if the model architecture/framework supports it.

---

11. A Sample Exercise

Below is a code skeleton for students to complete. We will generate embeddings using a **HuggingFace** Transformer model (like DistilBERT) to see how it works in practice. You will fill in the missing parts (marked as `TODO`).

```
# TODO code sample
## Install Transformers if not available
# !pip install transformers sentencepiece --quiet  # Uncomment if needed

from transformers import AutoTokenizer, AutoModel
import torch

# 1. TODO: Choose a pretrained model name, e.g. 'distilbert-base-uncased'
model_name = 'distilbert-base-uncased'  # e.g. 'distilbert-base-uncased'

# 2. Load tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name)

# 3. Load model
model = AutoModel.from_pretrained(model_name)
model.eval()

# 4. TODO: Provide some example sentences
sentences = [
    # "I love machine learning.",
    # "Dogs are awesome pets.",
    # Add your own...
    "TODO: ADD A SENTENCE",
    "Blah blah blah",
    "Another sentence",
    "Yet another sentence",
    "If I could save time in a bottle, the first thing that I'd like to do",
    "Is to save every day 'til eternity passes away",
    "Just to spend them with you",
    "If I could make days last forever",
    "If words could make wishes come true",
    "I'd save every day like a treasure and then",
    "Again, I would spend them with you",
    "But there never seems to be enough time",
    "To do the things you want to do",
    "Once you find them",
    "I've looked around enough to know",
    "That you're the one I want to go through time with",
    "If I had a box just for wishes",
    "And dreams that had never come true",
    "The box would be empty",
    "Except for the memory of how",
    "They were answered by you",
    "But there never seems to be enough time",
    "To do the things you want to do",
    "Once you find them",
    "I've looked around enough to know",
    "That you're the one I want to go through time with",
    "Till all the seas gang dry, my dear, And the rocks melt with the sun; I will love thee still, my dear, While the sands of life shall run.",
]

for sentence in sentences:
    # 5. Tokenize sentence
    inputs = tokenizer(sentence, return_tensors='pt')

    # 6. Forward pass to get hidden states
    with torch.no_grad():
        outputs = model(**inputs)

    # 7. Get the [CLS] token embedding (for DistilBERT, it's the first hidden state)
    # or you can average the token embeddings, etc.
    last_hidden_state = outputs.last_hidden_state
    # Typically, we might take the embedding of the first token or average:
    sentence_embedding = torch.mean(last_hidden_state, dim=1)
```

```
print(f"Sentence: {sentence}")
print("Embedding (first 5 values):", sentence_embedding[0][:5])
```

```
tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
config.json:   0%|          | 0.00/483 [00:00<?, ?B/s]
vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
2025-01-28 18:18:12.071138: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to register cuFFT factory: Attempting to register factory f
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1738070292.094561   24560 cuda_dnn.cc:8310] Unable to register cuDNN factory: Attempting to register factory for plugin cuDNN when one has
E0000 00:00:1738070292.101114   24560 cuda_blas.cc:1418] Unable to register cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
2025-01-28 18:18:12.123724: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in pe
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]
Sentence: TODO: ADD A SENTENCE
Embedding (first 5 values): tensor([-0.0258, -0.2476, -0.1363, -0.0953,  0.1093])
Sentence: Blah blah blah
Embedding (first 5 values): tensor([ 0.7752,  0.2569,  0.6226, -0.0606, -0.0516])
Sentence: Another sentence
Embedding (first 5 values): tensor([ 0.2550, -0.2537, -0.1267,  0.0377, -0.0992])
Sentence: Yet another sentence
Embedding (first 5 values): tensor([ 0.2824, -0.1910,  0.0803, -0.0420, -0.1538])
Sentence: If I could save time in a bottle, the first thing that I'd like to do
Embedding (first 5 values): tensor([ 0.1773,  0.1798, -0.0224, -0.0708,  0.1748])
Sentence: Is to save every day 'til eternity passes away
Embedding (first 5 values): tensor([ 0.3206,  0.0039,  0.5546, -0.1785,  0.2097])
Sentence: Just to spend them with you
Embedding (first 5 values): tensor([ 0.4350,  0.0317,  0.2756,  0.2262, -0.0903])
Sentence: If I could make davs last forever
```

Hints

1. **model_name**: Try `'distilbert-base-uncased'` or `'bert-base-uncased'`.
2. **Sentences**: Add a few short sentences that you want to test.
3. **Embedding**: Observe how the numbers change for different sentences.

```
Sentence: To do the things you want to do
```

12. Glossary

- **Tokenization**: The process of splitting text into tokens (subwords, words, or characters).
- **Embedding**: A dense vector that represents a piece of text. Learned during training.
- **Vector**: A list/array of numbers. In NLP, used to represent words/documents in numeric form.
- **Context Window**: The set of neighboring words around a target word.
- **Word2Vec**: Early neural embedding method (Mikolov et al.).
- **GloVe**: Global Vectors for Word Representation (Stanford). Uses matrix factorization of co-occurrences.
- **Attention**: A mechanism (esp. in Transformers) that helps the model weigh how important each token is with respect to others.
- **Transformer**: A neural network architecture using self-attention. BERT, GPT, etc. are based on Transformers.
- **LLM (Large Language Model)**: A huge neural model trained on massive text data to understand/generate language.
- **Cosine Similarity**: A common measure for how similar two vectors are, based on angle rather than magnitude.
- **Out-of-Vocabulary (OOV)**: Words not present in the training set's vocabulary.
- **Subword Tokenizer**: Tokenizes at subword or character level to handle new/rare words effectively.
- **[CLS] Token**: A special token used by BERT-like models, often used as a representation for the entire sequence.

End of Notebook

Congratulations! You now have an understanding of how tokenization, vectors, and embeddings work in LLMs, from basic concepts to advanced details. Feel free to experiment and modify the code in this notebook.

For more reading, check out original papers:

- Word2Vec Paper (Mikolov et al., 2013)
- GloVe Paper (Pennington et al., 2014)
- Attention is All You Need (Vaswani et al., 2017)

Keep exploring and have fun!

```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
  colab_check = "Yes" if 'google.colab' in sys.modules else "No"
  mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in reversed(range(0, 48, 8)))
  print("+++ Acknowledgement +++")
  print(f"Executed on: {datetime.datetime.now()}")
  print(f"In Google Colab: {colab_check}")
```

```
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-28 18:20:31.230523
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```