Fixing the Cost of Embeddings in RAG Systems

Day 1 of 9

Table of Contents

1. Introduction

Embeddings are crucial components of Retrieval Augmented Generation (RAG) systems, enabling the conversion of text into numerical vectors that capture semantic meaning.

These vectors facilitate tasks such as document retrieval and similarity search.

Next page

However, embeddings come with hidden costs that can impact the efficiency and scalability of RAG systems.

In this Day 1 tutorial of the "Fixing the Hidden Cost of Embeddings in RAG" series, we'll explore the fundamentals of embeddings in RAG systems, with a particular focus on the often-overlooked costs associated with them.

We'll also demonstrate how to generate embeddings using OpenAI's ADA model.

2. What are Embeddings?

Embeddings are high-dimensional vectors representing data (typically text) in a continuous vector space.

These vectors capture the semantic relationships between different pieces of data, allowing similar data points to be close to each other in the vector space.

2.1 Example of Word Embeddings

Consider the words "king," "queen," "man," and "woman." Word embeddings might represent these words as follows:

```
king   = [0.5, 0.8, -0.3, 0.7]
queen  = [0.4, 0.9, -0.2, 0.6]
man    = [0.2, 0.6, -0.5, 0.5]
woman  = [0.3, 0.7, -0.4, 0.6]
```

The idea is that the difference between the embeddings of "king" and "queen" should be similar to the difference between "man" and "woman," capturing the gender relationship.

2.2 Embeddings in RAG Systems

In RAG systems, embeddings represent both the query and the documents or passages in a vector space.

When a query is received, it is converted into an embedding, and the system retrieves documents with embeddings that are closest to the query embedding.

3. Understanding Embedding Costs

Embeddings, while powerful, introduce several hidden costs that can affect the performance and scalability of RAG systems:

3.1 Storage Costs

Embeddings are typically high-dimensional vectors, which means storing a large number of embeddings can consume significant storage space.

This can lead to higher infrastructure costs in large-scale applications.

3.2 Compute Costs

Generating embeddings requires computational resources, particularly when using large models.

This cost becomes significant when embeddings need to be generated in real-time or for large datasets.

3.3 Retrieval Costs

Searching for similar embeddings in a large vector space is computationally intensive.

The cost of retrieving the nearest neighbors grows with the number of embeddings and their dimensionality.

3.4 Maintenance Costs

Embeddings may need to be updated or re-generated as new data is added or as the underlying models evolve.

This incurs additional compute and storage costs.

4. Detailed Sample Code

We'll now set up a Google Colab environment to generate embeddings using OpenAI's ADA model and explore the associated costs.

4.1 Install Required Libraries

```
# Install the required libraries
!pip install openai

import openai
import numpy as np
```

Explanation: **We install the openai Python package, which allows us to interact with OpenAI's API to generate embeddings using the ADA model.**

4.2 Set Up OpenAI API Key

Before proceeding, you'll need to set up your OpenAI API key.

You can obtain this from the OpenAI platform and securely store it in your Colab environment.

```
import os

# Set your OpenAI API key here
os.environ["OPENAI_API_KEY"] =
                    "your-api-key-here"
```

Explanation: Replace "your-api-key-here" with your actual OpenAI API key. This key is required to authenticate requests to the OpenAI API.

4.3 Generate Embeddings Using OpenAI's ADA Model

```
def get_embedding(text, model="text-embedding-ada-002"):
    response = openai.Embedding.create(
        input=[text],
        model=model
    )
    embedding = response['data'][0]['embedding']
    return np.array(embedding)

# Example text
text = "This is an example sentence for generating
embeddings."
# Generate embedding for the example text
embedding = get_embedding(text)
print("Embedding shape:", embedding.shape)
print("Embedding vector:", embedding)
```

Explanation: We define a function get_embedding that takes a text input and generates its embedding using the OpenAI ADA model.

The embedding is returned as a NumPy array. We then generate an embedding for an example sentence and print its shape and vector.

4.4 Understanding Storage Costs

```
# Simulate storing a large number of embeddings

# Let's assume we have a million embeddings
num_embeddings = 1000000
embedding_dim = embedding.shape[0]

# Calculate the storage required in bytes

# 4 bytes per float32 value
storage_cost = num_embeddings * embedding_dim * 4

# Convert to MB
storage_cost_MB = storage_cost / (1024**2)

print(f"Estimated storage cost for {num_embeddings}
                embeddings: {storage_cost_MB:.2f} MB")
```

Explanation: Here, we simulate the storage cost of a large number of embeddings by calculating the space required to store 1 million embeddings, assuming each element in the embedding vector is a 32-bit float.

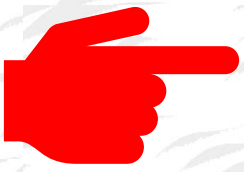
The result is converted to megabytes (MB) for better understanding.

5. Conclusion

In this introductory tutorial, we explored the basics of embeddings in RAG systems and uncovered the hidden costs associated with their use.

Understanding these costs is crucial for building efficient and scalable RAG systems.

Next page



In the next tutorial, we will dive deeper into storage costs and considerations, where we will discuss techniques to reduce storage needs without compromising the effectiveness of your embeddings.

Stay tuned for Day 2!

Link of collab Notebook

Fixing_the_Cost_of_Embeddings_in_RAG_Systems_Day_1

August 22, 2024

1 LLM Text Masking to Protect Sensitive Data: Day-1

1.1 Table of Contents

#**1. Introduction** ###Embeddings are crucial components of Retrieval Augmented Generation (RAG) systems, enabling the conversion of text into numerical vectors that capture semantic meaning.

###These vectors facilitate tasks such as document retrieval and similarity search.

###However, embeddings come with hidden costs that can impact the efficiency and scalability of RAG systems.

###In this Day 1 tutorial of the "Fixing the Hidden Cost of Embeddings in RAG" series, we'll explore the fundamentals of embeddings in RAG systems, with a particular focus on the often-overlooked costs associated with them.

###We'll also demonstrate how to generate embeddings using OpenAI's ADA model.

#**2. What are Embeddings?** ###Embeddings are high-dimensional vectors representing data (typically text) in a continuous vector space.

###These vectors capture the semantic relationships between different pieces of data, allowing similar data points to be close to each other in the vector space.

##**2.1 Example of Word Embeddings** Consider the words "king," "queen," "man," and "woman." Word embeddings might represent these words as follows:

[ ]:
```
king   = [0.5, 0.8, -0.3, 0.7]
queen  = [0.4, 0.9, -0.2, 0.6]
```

```
man    = [0.2, 0.6, -0.5, 0.5]
woman  = [0.3, 0.7, -0.4, 0.6]
```

The idea is that the difference between the embeddings of "king" and "queen" should be similar to the difference between "man" and "woman," capturing the gender relationship.

##**2.2 Embeddings in RAG Systems** In RAG systems, embeddings represent both the query and the documents or passages in a vector space.

When a query is received, it is converted into an embedding, and the system retrieves documents with embeddings that are closest to the query embedding.

#**3. Understanding Embedding Costs** ###Embeddings, while powerful, introduce several hidden costs that can affect the performance and scalability of RAG systems:

1. Storage Costs Embeddings are typically high-dimensional vectors, which means storing a large number of embeddings can consume significant storage space. This can lead to higher infrastructure costs in large-scale applications.

2. Compute Costs Generating embeddings requires computational resources, particularly when using large models. This cost becomes significant when embeddings need to be generated in real-time or for large datasets.

3. Retrieval Costs Searching for similar embeddings in a large vector space is computationally intensive. The cost of retrieving the nearest neighbors grows with the number of embeddings and their dimensionality.

4. Maintenance Costs Embeddings may need to be updated or re-generated as new data is added or as the underlying models evolve. This incurs additional compute and storage costs.

2 4. Detailed Sample Code

We'll now set up and generate embeddings using OpenAI's ADA model and explore the associated costs.

##**4.1 Install Required Libraries** In RAG systems, embeddings represent both the query and the documents or passages in a vector space.

[ ]:
```
# Install required libraries

!pip install openai==0.28
```

```
Requirement already satisfied: openai==0.28 in /usr/local/lib/python3.10/dist-
packages (0.28.0)
Requirement already satisfied: requests>=2.20 in /usr/local/lib/python3.10/dist-
packages (from openai==0.28) (2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages
(from openai==0.28) (4.66.5)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-
packages (from openai==0.28) (3.10.5)
Requirement already satisfied: charset-normalizer<4,>=2 in
```

```
/usr/local/lib/python3.10/dist-packages (from requests>=2.20->openai==0.28)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests>=2.20->openai==0.28) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.20->openai==0.28)
(2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.20->openai==0.28)
(2024.7.4)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (2.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp->openai==0.28) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (6.0.5)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp->openai==0.28) (1.9.4)
Requirement already satisfied: async-timeout<5.0,>=4.0 in
/usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (4.0.3)
```

###**Explanation:** We install the 'openai' Python package, which allows us to interact with OpenAI's API to generate embeddings using the ADA model.

##**4.2 Set Up OpenAI API Key** Before proceeding, you'll need to set up your OpenAI API key. You can obtain this from the OpenAI platform and securely store it in your Colab environment.

[ ]:
```python
import openai
import numpy as np
import os

# Set your OpenAI API key here
openai.api_key = "your OpenAI API key here"
```

##**4.3 Generate Embeddings Using OpenAI's ADA Model**

[ ]:
```python
def get_embedding(text, model="text-embedding-ada-002"):
    response = openai.Embedding.create(
        input=[text],
        model=model
    )
    embedding = response['data'][0]['embedding']
    return np.array(embedding)

# Example text
```

```
text = "This is an example sentence for generating embeddings."

# Generate embedding for the example text
embedding = get_embedding(text)
print("Embedding shape:", embedding.shape)
print("Embedding vector:", embedding)
```

```
Embedding shape: (1536,)
Embedding vector: [-0.02978859  0.00220844  0.01431259 … -0.00223212
-0.01335211
 -0.02399862]
```

###**Explanation:** We define a function get_embedding that takes a text input and generates its embedding using the OpenAI ADA model. The embedding is returned as a NumPy array. We then generate an embedding for an example sentence and print its shape and vector.

##**4.4 Understanding Storage Costs**

[ ]:
```
# Simulate storing a large number of embeddings
num_embeddings = 1000000  # Let's assume we have a million embeddings
embedding_dim = embedding.shape[0]

# Calculate the storage required in bytes
storage_cost = num_embeddings * embedding_dim * 4  # 4 bytes per float32 value
storage_cost_MB = storage_cost / (1024**2)  # Convert to MB

print(f"Estimated storage cost for {num_embeddings} embeddings:␣
 ↪{storage_cost_MB:.2f} MB")
```

```
Estimated storage cost for 1000000 embeddings: 5859.38 MB
```

###**Explanation:** Here, we simulate the storage cost of a large number of embeddings by calculating the space required to store 1 million embeddings, assuming each element in the embedding vector is a 32-bit float. The result is converted to megabytes (MB) for better understanding.

3 5. Conclusion

###In this introductory tutorial, we explored the basics of embeddings in RAG systems and uncovered the hidden costs associated with their use.

###Understanding these costs is crucial for building efficient and scalable RAG systems.

###In the next tutorial, we will dive deeper into storage costs and considerations, where we will discuss techniques to reduce storage needs without compromising the effectiveness of your embeddings.

###Stay tuned for Day 2!