🔧 FAISS 실습 - 커널 충돌 & 메모리 부족 문제 트러블슈팅

> **작성일:** 2025-09-26
> **작성자:** Jay
> **소요 시간:** 4시간 30분
> **최종 결과:** `HuggingFace Embeddings 초경량 모델 해결`

1. 문제 상황

- **목표:**
  - `FAISS 벡터 저장소` 학습 중 **벡터 검색 시스템** 구축 필요
  - Google Gemini 임베딩 모델(3072차원)을 사용한 **FAISS 인덱스** 생성
  - **`LangChain + FAISS + Google Embeddings`** 환경에서 유사도 검색 구현
- **환경:**

```
- `Python`: 3.13.5 (pyenv 가상환경)
- `LangChain`: 최신 버전
- `임베딩 모델`: `models/gemini-embedding-001` (3072차원)
- `벡터 저장소`: FAISS (Facebook AI Similarity Search)
- `개발환경`: Google Colab / Jupyter Notebook
```

2. 시도한 실패 방법들

2.1 Google Gemini 임베딩 방식 시도 ❌

```
from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain_community.vectorstores import FAISS

# Google Gemini 임베딩 (3072차원)
embeddings = GoogleGenerativeAIEmbeddings(
    model="models/gemini-embedding-001",
    task_type="retrieval_document",
    google_api_key=os.getenv("GOOGLE_API_KEY")
)

# FAISS 벡터스토어 생성 시도
vectorstore = FAISS.from_texts(
    texts=documents,
    embedding=embeddings
)
```

실패 원인:

- 🚫 **높은 차원수**: 3072차원으로 인한 **과도한 메모리 사용량**
- 🔥 **메모리 오버플로우**: 시스템 메모리 사용률 **78-80%** 상태에서 처리 불가
- 💥 **커널 충돌**: `index.add(np.array(doc_embeddings))` 부분에서 반복적 크래시

2.2 OpenAI 임베딩 시도 ❌

```
from langchain_openai import OpenAIEmbeddings

# OpenAI 임베딩 (1536차원)
embeddings_openai = OpenAIEmbeddings(
    model="text-embedding-3-small",
    openai_api_key=os.getenv("OPENAI_API_KEY")
)
```

실패 원인:

- 💸 **API 요금 부족**: OpenAI 크레딧 소진으로 사용 불가능
- 🌐 **네트워크 의존성**: 인터넷 연결 필수
- 📊 **여전한 고차원**: 1536차원으로 여전히 메모리 부담

2.3 기본 HuggingFace 임베딩 시도 ❌

```
from langchain_community.embeddings import HuggingFaceEmbeddings

# 기본 HuggingFace 모델 (768차원)
embeddings = HuggingFaceEmbeddings()
```

실패 원인:

- ⚠️ **Deprecation 경고**: `LangChain 0.2.2`에서 deprecated 상태
- 📦 **패키지 충돌**: 구버전 패키지 사용으로 인한 호환성 문제
- 🐍 **Import 에러**: 올바르지 않은 import 경로

3. 핵심 문제 진단

3.1 메모리 부족 & 차원의 저주 🦠

- **현상:** 고차원 임베딩으로 인한 **시스템 메모리 오버플로우**
- **Google Gemini**: 3072차원 → 과도한 메모리 소비
- **OpenAI**: 1536차원 → 여전히 높은 메모리 사용량
- **추정 원인:** Colab/Jupyter 환경의 제한된 RAM에서 대용량 벡터 처리 불가

3.2 FAISS 커널 충돌 패턴 🚨

```
Document 'hello world' embedded in 0.36 seconds
Document 'goodbye world' embedded in 0.37 seconds
# ↑ 여기까지는 성공
# ↓ index.add() 호출 시점에서 커널 크래시 발생
현재 셀 또는 이전 셀에서 코드를 실행하는 동안 Kernel이 충돌했습니다.
```

- **충돌 지점**: `index.add(np.array(doc_embeddings))` 실행 시
- **메모리 패턴**: 임베딩 생성은 성공하지만 FAISS 인덱스 추가 시 실패
- **재현성**: 동일한 지점에서 반복적으로 발생

4. 해결 방향 전환

4.1 초경량 HuggingFace Embeddings 발견 💡

핵심 아이디어: 고차원 API 기반 모델 대신 **저차원 로컬 모델** 사용

- **Google Gemini**: 3072차원 → **8배 메모리 사용**
- **OpenAI**: 1536차원 → **4배 메모리 사용**
- **MiniLM-L6-v2**: 384차원 → **기준 메모리 사용** ✅

4.2 최신 LangChain 패키지 적용 🔄

```
# 🎯 패키지 업데이트 & 설치
pip install -U langchain-huggingface sentence-transformers faiss-cpu
pip install -U langchain langchain-community
```

핵심 변경사항:

- `langchain_community.embeddings.HuggingFaceEmbeddings` ❌
- `langchain_huggingface.HuggingFaceEmbeddings` ✅

5. 최종 성공 구현

5.1 완성된 초경량 FAISS 시스템

```
import gc
import numpy as np
import time
from langchain_huggingface import HuggingFaceEmbeddings  # ✅ 최신 import
from langchain_community.vectorstores import FAISS
import warnings

# 경고 무시
```

```
warnings.filterwarnings("ignore")

def ultra_light_faiss_updated():
    """최신 LangChain으로 수정된 초경량 FAISS"""

    print("🚀 LangChain으로 초경량 임베딩 시작...")

    try:
        # 1️⃣ 최신 HuggingFace Embeddings (384차원!)
        embeddings = HuggingFaceEmbeddings(
            model_name="sentence-transformers/all-MiniLM-L6-v2",
            model_kwargs={'device': 'cpu'},
            encode_kwargs={'normalize_embeddings': True}
        )

        print("✅ 최신 임베딩 모델 로딩 완료!")
        print("🔪 임베딩 차원: 384 (Google Gemini 대비 1/8 절약!)")

        # 2️⃣ 테스트 문서
        documents = [
            "자연어 처리는 컴퓨터가 인간의 언어를 이해하는 기술입니다.",
            "머신러닝은 데이터로부터 패턴을 학습하는 방법입니다.",
            "딥러닝은 신경망을 이용한 머신러닝 기법입니다.",
            "FAISS는 효율적인 유사도 검색을 위한 라이브러리입니다.",
            "벡터 데이터베이스는 임베딩을 저장하고 검색하는 시스템입니다."
        ]

        # 3️⃣ FAISS 벡터스토어 생성 (성공!)
        print("🔄 FAISS 벡터스토어 생성 중...")
        start_time = time.time()

        vectorstore = FAISS.from_texts(
            texts=documents,
            embedding=embeddings,
            metadatas=[{"id": i, "source": f"doc_{i}"} for i in
range(len(documents))]
        )

        end_time = time.time()
        print(f"✅ FAISS 생성 완료! (소요시간: {end_time-start_time:.2f}초)")

        return vectorstore, embeddings

    except Exception as e:
        print(f"❌ 오류 발생: {e}")
        gc.collect()
        return None, None
```

5.2 유사도 검색 & 고급 기능 구현

```
def faiss_advanced_operations(vectorstore):
    """FAISS 고급 기능 시연"""

    # 4️⃣ 유사도 검색 테스트
    print("\n🔍 유사도 검색 테스트...")
    query = "딥러닝과 머신러닝의 차이점"

    # 기본 유사도 검색
    results = vectorstore.similarity_search_with_score(query, k=3)

    print("🎉 검색 결과:")
    for i, (doc, score) in enumerate(results):
        similarity = 1 - score
        print(f"  {i+1}. 유사도: {similarity:.4f}")
        print(f"      내용: {doc.page_content}")
        print(f"      메타데이터: {doc.metadata}\n")

    # 5️⃣ 문서 추가 (add_documents)
    from langchain_core.documents import Document

    new_docs = [
        Document(
            page_content="Transformer는 어텐션 메커니즘을 사용하는 딥러닝 모델입니
다.",
            metadata={"source": "new_doc_1"}
        )
    ]

    vectorstore.add_documents(new_docs, ids=["transformer_doc"])
    print("✅ 새 문서 추가 완료!")

    # 6️⃣ 문서 삭제 테스트
    test_ids = vectorstore.add_texts(
        ["삭제 테스트용 문서"],
        metadatas=[{"source": "test"}],
        ids=["delete_test"]
    )

    vectorstore.delete(test_ids)  # 삭제
    print("✅ 문서 삭제 완료!")

    return vectorstore
```

5.3 저장 및 로드 시스템

```
def faiss_save_load_system(vectorstore, embeddings):
    """FAISS 저장 및 로드 시스템"""

    # 7️⃣ 로컬 저장
    vectorstore.save_local(
```

```
        folder_path="../09_VectorStore/faiss_light_index",
        index_name="faiss_index"
    )
    print("💾 FAISS 인덱스 저장 완료!")

    # 8️⃣ 로컬에서 로드
    loaded_vectorstore = FAISS.load_local(
        folder_path="../09_VectorStore/faiss_light_index",
        index_name="faiss_index",
        embeddings=embeddings,
        allow_dangerous_deserialization=True
    )

    # 로드 검증
    test_query = "자연어 처리"
    results = loaded_vectorstore.similarity_search(test_query, k=1)
    print(f"✅ 로드 검증 완료: {results[0].page_content[:50]}...")

    return loaded_vectorstore
```

6. 최종 성공 결과🎉

6.1 처리 결과 통계

```
🚀 LangChain으로 초경량 임베딩 시작...
✅ 최신 임베딩 모델 로딩 완료!
📏 임베딩 차원: 384 (Google Gemini 대비 1/8 절약!)
🔄 FAISS 벡터스토어 생성 중...
✅ FAISS 생성 완료! (소요시간: 1.48초)

🎉 검색 결과:
  1. 유사도: 0.5892
     내용: 딥러닝은 신경망을 이용한 머신러닝 기법입니다.
     메타데이터: {'id': 2, 'source': 'doc_2'}

  2. 유사도: 0.4567
     내용: 머신러닝은 데이터로부터 패턴을 학습하는 방법입니다.
     메타데이터: {'id': 1, 'source': 'doc_1'}

💾 인덱스 저장 완료!
🎉🎉🎉 최신 LangChain으로 FAISS 완료! 🎉🎉🎉
```

6.2 메모리 사용량 비교

임베딩 모델	차원	메모리 사용량	커널 충돌
Google Gemini	3072	🔥🔥🔥🔥🔥🔥🔥🔥	❌ 충돌

임베딩 모델	차원	메모리 사용량	커널 충돌
OpenAI Ada-002	1536	🔥 🔥 🔥 🔥	❌ 충돌
HuggingFace 기본	768	🔥 🔥	⚠ 경고
MiniLM-L6-v2	384	🔥	✅ 성공

6.3 구현 완료 기능들

✅ **FAISS 벡터스토어 생성**: `from_documents`, `from_texts`
✅ **유사도 검색**: `similarity_search`, `similarity_search_with_score`
✅ **문서 관리**: `add_documents`, `add_texts`, `delete`
✅ **메타데이터 필터링**: 소스별 검색 기능
✅ **저장/로드**: `save_local`, `load_local`
✅ **메모리 최적화**: 가비지 컬렉션, 배치 처리

7. 교훈 및 성과 🎓

7.1 기술적 교훈

- **차원의 저주**: 고차원 벡터는 메모리 사용량을 기하급수적으로 증가시킴
- **로컬 vs API**: 네트워크 기반 API보다 로컬 모델이 더 안정적이고 비용 효율적
- **패키지 버전 관리**: LangChain 생태계의 빠른 변화에 대응하는 최신 패키지 사용 중요
- **메모리 관리**: 대용량 데이터 처리 시 명시적 메모리 정리 필수

7.2 문제해결 역량 향상

- **체계적 분석**: 메모리 사용량, 차원 수, 패키지 버전을 종합적으로 고려
- **대안 모색**: API 부족 → 로컬 모델, 고차원 → 저차원으로 전환
- **성능 최적화**: 384차원으로 8배 메모리 절약하면서 성능 유지
- **안정성 확보**: 커널 충돌 없는 안정적인 시스템 구축

7.3 학습 성과

- **FAISS 완전 정복**: 벡터 저장소 생성부터 고급 기능까지 전 과정 마스터
- **임베딩 모델 선택**: 용도별 최적 모델 선택 기준 확립
- **LangChain 생태계**: 최신 패키지 구조와 import 방식 완전 이해
- **메모리 효율성**: 제한된 환경에서의 최적화 기법 습득

8. 향후 개선 방안 🚀

8.1 성능 최적화

- **더 작은 모델**: `paraphrase-MiniLM-L3-v2` (17MB) 테스트
- **양자화**: 모델 압축을 통한 추가 메모리 절약
- **병렬 처리**: 대용량 문서 처리 시 멀티프로세싱 적용

8.2 기능 확장

- **하이브리드 검색**: 키워드 + 벡터 검색 결합
- **실시간 업데이트**: 동적 문서 추가/삭제 시스템
- **분산 처리**: 여러 FAISS 인덱스 병합 및 관리

8.3 안정성 향상

- **자동 백업**: 주기적 인덱스 저장 시스템
- **오류 복구**: 부분 실패 시 복구 메커니즘
- **모니터링**: 메모리 사용량 실시간 추적

9. 핵심 코드 정리

9.1 최종 해결 코드

```
# 🎯 핵심 해결 방법: 초경량 HuggingFace + 최신 패키지
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS

embeddings = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",  # 384차원
    model_kwargs={'device': 'cpu'},
    encode_kwargs={'normalize_embeddings': True}
)

vectorstore = FAISS.from_texts(
    texts=documents,
    embedding=embeddings
)
```

9.2 사용법

```
# 유사도 검색
results = vectorstore.similarity_search("검색어", k=3)

# 문서 추가
vectorstore.add_texts(["새 문서"], ids=["new_id"])

# 저장/로드
vectorstore.save_local("./faiss_index")
loaded_db = FAISS.load_local("./faiss_index", embeddings=embeddings,
                             allow_dangerous_deserialization=True)
```

🏆 **결론: Google API 요금 부족과 메모리 충돌을 초경량 로컬 모델로 해결하여 FAISS 벡터 검색 시스템 완전 정복!**

핵심 성공 요인: 차원 축소(3072→384) + 로컬화 + 최신 패키지 + 체계적 메모리 관리