Pusan National University
Computer Science and Engineering
Technical Report 2025-09

경량 언어 모델 (SLM) 학습을 위한 웹 기반 시각화 플랫폼 설계 및 구현

김명석

염현석

정지윤

지도교수 조준수

목 차

1. 서론

1.1. 연구 배경

최근 몇 년간 인공지능(AI)과 자연어 처리(NLP) 분야는 ChatGPT, 구글 제미나이(Gemini)와 같은 대규모 언어 모델(LLM: Large Language Model)의 등장으로 인해 전례 없는 대중적 관심을 받고 있다. 이들 모델은 단순히 정보를 제공하는 것을 넘어, 자연스러운 대화, 창의적인 글쓰기, 코드 생성 등 광범위한 영역에서 인간과 유사한 성능을 보여주며 AI 기술의 활용 가능성을 극적으로 확장했다.

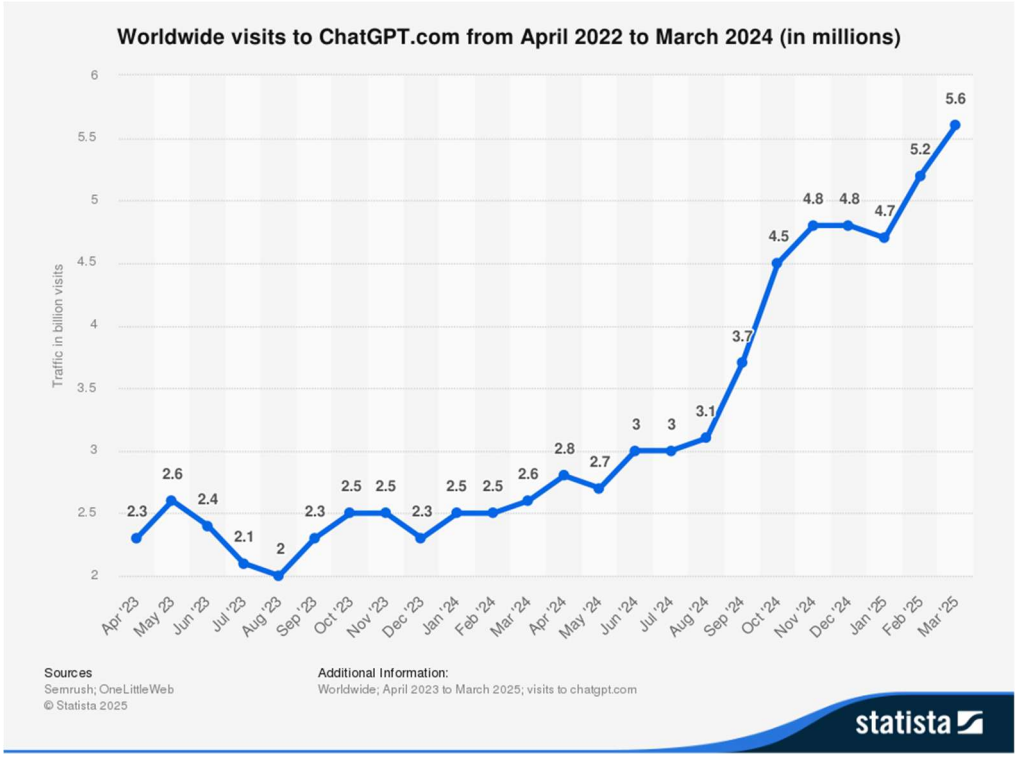

그림 1. GPT 사용자 통계 자료

그림 1의 통계에 따르면 ChatGPT의 전 세계 방문자 수는 2023년 4월 약 23억 회에서 2025년 3월 약 56억 회로 두 배 이상 증가하였다. 이는 LLM이 단순한 기술적 유행을 넘어, 학습·업무·일상 전반에 걸쳐 활용되는 핵심 도구로 자리 잡았음을 보여준다. 이처럼 LLM에 대한 수요와 관심이 커지는 상황에서, 막대한 자원 소모와 운영 비용을 요구하는 LLM 대신 경량 언어 모델(SLM, Small Language Model)에 대한 연구 필요성이 대두되고 있다. SLM은 상대적으로 적은 연산 자원으로도 학습과 추론이 가능해, 제한된 GPU 환경

이나 교육용 실습 플랫폼에서도 활용 가치가 크다. 따라서 실제 교육 및 연구 현장에서 적용 가능한 SLM을 이해하고 구현할 수 있는 기반을 마련하는 것이 본 연구의 중요한 필요성이라 할 수 있다.

1.2. 기존 문제점

대규모 언어 모델(LLM)은 강력한 성능을 보이지만, 교육 현장이나 연구 초심자가 직접 실험하기에는 여러 제약이 따른다. 우선, 모델 설계 복잡성으로 인해 트랜스포머 블록이나 어텐션 구조를 이해하고 코드로 구현하는 과정이 진입장벽이 높다. 또한, 실험 환경 부족 문제도 존재한다. 다양한 모델 아키텍처를 시도하고 성능 변화를 직관적으로 확인하려면 TensorBoard, MLflow 같은 별도 도구를 설치해야 하지만, 초심자에게는 추가 설정과 사용법이 또 다른 부담이 된다.

1.2.1. LLM 교육 문제 1 - 성능 확인 환경 부족

현재 교육 현장이나 개인 학습 차원에서 딥러닝 모델을 직접 구축하고 실험해 보는 방식은 대개 다음과 같은 제한이 있다. 첫째, 텍스트나 영상 강의 위주의 이론 학습과 실제 코딩 실습의 간극이 크다. 이론을 이해하더라도, 이를 바로 코드로 구현하는 과정에서 난관에 부딪히는 경우가 많아 모델 구조 설계나 하이퍼파라미터 튜닝 등에 대한 실질적인 학습 기회가 제한된다. 둘째, 다양한 딥러닝 아키텍처를 실시간으로 비교하거나 실험 결과를 시각적으로 확인하기 쉽지 않다. 모델의 레이어나 블록을 유연하게 추가·변경하고, 그에 따른 성능 변화를 즉시 확인하며 학습할 수 있는 환경이 부족하다. 일반적으로는 TensorBoard, MLflow 등의 도구를 별도로 설치·활용해야 하며, 이와 같은 도구들에 대한 이해도와 추가 설정 작업이 필요하다. 초보자나 교육 대상자 입장에서는 모델 성능 변화를 직관적으로 모니터링하고 여러 실험 결과를 한눈에 비교·분석하기가 쉽지 않아, 반복 학습 및 피드백 과정이 원활하지 않다.

1.2.2. LLM 교육 문제 2 - 플랫폼 부족

이로 인해 학습자나 연구 초심자들은 직관적이고 시각화된 방식으로 딥러닝 모델을 조립하고, 성능을 비교 · 분석해보는 경험을 충분히 누리지 못하고 있다. 이에 따라

'Transformer Block'을 비롯한 여러 딥러닝 레이어를 드래그 앤 드롭(Drag & Drop) 방식으로 쉽게 설계하고, 모델을 직접 학습 및 비교·분석할 수 있는 웹 기반의 학습·실험 플랫폼이 필요하다. 그러나 현재 이러한 플랫폼이 충분히 갖춰져 있지 않아, 원하는 모델을 실제로 구성해 보고 성능을 개선하는 과정을 단일 인터페이스에서 수행하기가 어려운 실정이다. 또한 여러 라이브러리와 환경 설정을 개별적으로 익혀야 하므로, 학습자가 모델 구조의 변화가 성능에 미치는 영향을 직관적으로 파악하기까지 많은 시간과 노력이 소요되고 있다.

1.2.3. 데이터셋 구성

아울러 데이터셋을 다루는 문제도 있다. 딥러닝 실습을 위해서는 일정 규모 이상의 학습 데이터셋을 준비해야 하는데, 이를 직접 수집·전처리하기 위해선 추가적인 기술적·시간적 비용이 발생한다. 기존 오픈소스 데이터셋을 활용한다고 하더라도, 웹 인터페이스 상에서 간편하게 불러와 쓰기가 쉽지 않고, 각기 다른 형식의 데이터셋을 재구성하는 과정이 필요하다. 이러한 어려움은 딥러닝 교육 및 실험의 진입 장벽을 더욱 높이고, 학습자들이 모델 설계나 성능 평가에 집중하기보다는 환경 세팅 및 데이터 준비에 대부분의 시간을 할애하게 만드는 원인이 된다.

1.2.4. 비효율성

결과적으로, 초심자나 학생들이 이론과 실제 코딩을 매끄럽게 연결하여 딥러닝 모델을 설계·학습·평가해보는 통합적인 경험을 얻기 어려운 상황이다. 또한 이미 어느 정도 경험이 있는 연구자나 개발자도 간단한 시제품 수준의 모델을 빠르게 구성하고 성능을 확인해보려 할 때, 매번 환경 설정과 모델 구현 과정을 반복해야 하는 비효율을 겪게 된다. 이는 딥러닝 학습과 연구 과정에서 발생하는 시간·노력·자원 낭비를 야기하며, 나아가 창의적인 모델 실험과 효율적인 교육 환경 구축을 저해하는 주요한 문제로 작용하고 있다.

1.3. 연구 목표

본 프로젝트의 주된 목표는 웹 환경에서 직관적이고 체계적으로 딥러닝 모델을 설계하고 학습·비교할 수 있는 플랫폼을 구축하는 것이다. 이를 통해 모델 최적화를 실시간으로 실험하고 다양한 성능 지표를 시각적으로 확인함으로써, 사용자(교육 대상자 및 연구 초심

자)가 딥러닝 개념을 빠르고 정확하게 이해할 수 있도록 돕는 것을 최종 목적으로 삼는다. 구체적으로 다음과 같은 세부 목표를 설정한다.

1.3.1. 드래그 & 드롭을 활용한 직관적인 모델 설계

마우스 드래그 & 드롭으로 트랜스포머 블록이나 선형 레이어 등 다양한 레이어를 쉽게 배치하고 연결할 수 있는 그래픽 기반 인터페이스를 제공한다. 이를 통해 초심자도 복잡한 딥러닝 구조를 시각적으로 이해하며 설계 과정을 체험할 수 있다.

생성된 모델 그래프에서 레이어 간 연결 관계, 하이퍼파라미터 설정 등을 자유롭게 수정·확장할 수 있도록 하여, 다양한 실험 시나리오에 대응할 수 있게 한다.

1.3.2. 빠른 학습 및 비동기 처리를 통한 사용자 경험 강화

웹 서버와 모델 학습 서버를 분리하거나 병렬 연산을 적극적으로 활용해 학습 속도를 향상시킨다. 이를 통해 사용자는 빠른 피드백 루프를 확보하고, 반복 실험을 손쉽게 진행할 수 있다.

모델 훈련이 백그라운드에서 이루어지도록 설계해, 웹 인터페이스가 멈추거나 지연되지 않도록 한다. 사용자는 다른 설정을 변경하거나, 이미 학습된 다른 모델과 성능을 비교하는 등 병행 작업이 가능해진다.

1.3.3. 시각화된 학습 결과 및 모델 비교 제공

학습 과정에서의 손실 곡선, 정확도, F1 점수 등 주요 성능 지표를 그래프 형태로 시각화하여, 모델이 어떻게 학습되고 있는지를 직관적으로 파악할 수 있도록 한다.

서로 다른 모델 구조(예: 레이어 개수 차이, 트랜스포머 블록 포함 여부 등)를 한 화면에서 비교 분석할 수 있게 하여, 어떤 요소가 성능 개선에 영향을 미치는지 빠르게 확인할 수 있도록 한다.

1.3.4. 교육 목적 최적화 및 접근성 강화

데이터셋 업로드, 모델 구성, 학습, 평가 및 시각화 결과 확인까지 모든 과정을 한 웹사이트에서 통합적으로 처리할 수 있도록 하여, 복잡한 환경 설정이나 라이브러리 설치 부담을 최소화한다.

초심자도 다양한 모델 구조를 시도해보며 학습 과정에서 얻은 피드백을 토대로 모델을 개선해나갈 수 있게 한다. 이를 통해 딥러닝 개념과 최적화 과정을 자연스럽게 체득할 수 있는 학습 환경을 조성한다.

1.3.5. 효율적 자원 활용 및 시스템 확장성

GPU, CPU 등 컴퓨팅 자원을 동적으로 할당·관리하여 사용자가 여러 명이 동시에 실습하더라도 웹사이트가 무리 없이 동작하도록 설계한다.

향후 추가되는 딥러닝 레이어(예: RNN, CNN 등)나 새로운 시각화·분석 기능을 손쉽게 통합할 수 있도록 시스템 구조를 모듈화하고, 확장성을 고려하여 개발한다.

2. 연구 배경

2.1. SLM의 필요성과 가능성

SLM은 적은 메모리와 연산 자원으로도 학습과 추론을 수행할 수 있어, 개인 연구자나 중소 규모의 프로젝트에서도 활용이 가능하다. 특히 GPU 자원이 제한된 환경에서도 실험과 최적화를 진행할 수 있다는 장점이 있다.

또한 SLM은 특정 분야에 맞춰 도메인 특화 모델로 쉽게 커스터마이징할 수 있으며, 파인튜닝(Fine-tuning)과 지식 증류(Knowledge Distillation) 기법을 활용하면 LLM에 근접한 성능을 확보할 수도 있다.

이러한 특성 덕분에 교육, 의료, 금융, 법률 등 다양한 분야에서 비용 효율적인 언어 모델로 활용되고 있다.

2.2. 기존 연구와의 차별성

본 연구는 기존의 딥러닝 교육·연구 환경과 비교했을 때 다음과 같은 차별성을 지닌다.

첫째, 통합 환경 제공이다. 기존에는 데이터셋 준비, 모델 설계, 학습, 평가, 시각화를 위해 여러 도구와 환경 설정을 따로 거쳐야 했다. 그러나 본 연구에서 개발한 플랫폼은 이러한 과정을 모두 하나의 웹 인터페이스에서 처리할 수 있도록 통합하였다. 사용자는 별도의 복잡한 툴 설치나 환경 설정 없이 웹 환경에서 손쉽게 모델 실험을 진행할 수 있다.

둘째, 직관적인 모델 설계 방식이다. 트랜스포머 블록, 임베딩 층, 어텐션 레이어 등 핵심 구성 요소를 드래그 앤 드롭(Drag & Drop) 방식으로 배치·연결할 수 있게 함으로써, 비전공자나 초심자도 복잡한 모델 구조를 직관적으로 이해하고 설계할 수 있다. 이는 기존의 코드 중심 학습 방식과 달리, 시각적·체험적 학습 효과를 제공한다는 점에서 교육적 의미가 크다.

셋째, 실험 결과 비교와 시각화 강화이다. MLflow를 활용하여 학습 과정에서의 손실값, 정확도, F1 점수 등을 자동으로 기록하고 그래프 형태로 시각화함으로써, 사용자는 서로 다른 모델 구조 간 성능 변화를 직관적으로 파악할 수 있다.

3. 연구 내용

3.1. 언어 모델 구축 준비 및 단계적 설계

본 연구에서는 토크나이저를 GPT-2 및 LLaMA 계열 모델에서 제공하는 기존 오픈소스

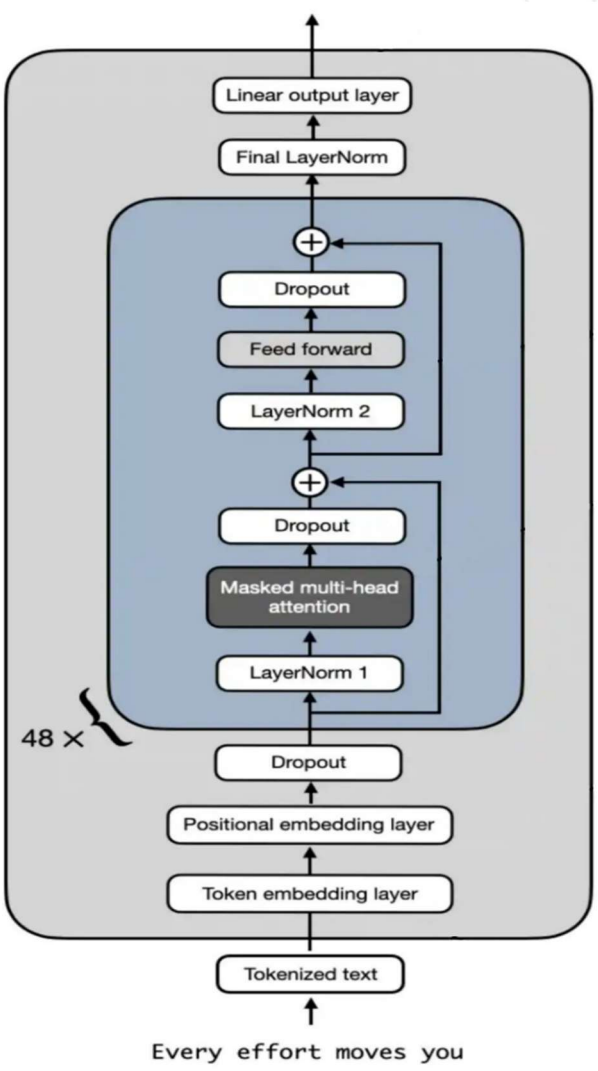

그림 2. 모델 구현 방법

토크나이저를 활용하였다. 이를 통해 학습 데이터가 안정적으로 토큰화되어 모델 입력으로 변환되도록 하였으며, 연구의 초점은 토크나이저 자체 개발이 아닌 모델 구조 설계와 학습 과정의 시각화에 맞추었다.

학습 데이터는 토크나이저를 거쳐 시퀀스 단위로 변환되며, 설정된 context length에 따

라 분할된다. 이 과정에서 stride 기법을 적용하여 연속된 시퀀스 간 문맥 손실을 최소화하였다. 이후 배치 단위로 묶어 효율적인 학습이 가능하도록 구성하였다.

모델은 JSON 기반의 layer_stack을 사용해 정의된다. 임베딩 레이어, 포지셔널 임베딩, 트랜스포머 블락, 피드포워드 레이어 등의 요소를 모듈화하였으며, 사용자가 웹 인터페이스에서 설정한 값에 따라 PyTorch 모델을 동적으로 생성할 수 있도록 설계하였다. 이 방식은 다양한 실험 조건에 따라 모델 구조를 손쉽게 확장·변경할 수 있는 장점을 가진다.

학습 과정은 PyTorch를 기반으로 하며, Celery와 Redis를 활용한 비동기 처리 구조로 설계되었다. 이를 통해 모델 학습이 웹 인터페이스와 독립적으로 진행되며, 사용자는 실시간으로 상태를 확인할 수 있다. 또한 학습 지표와 로그는 MLflow에 자동 기록되어, 실험 간 성능 변화를 추적·비교할 수 있다.

최종적으로 손실 곡선, 정확도 등 주요 성능 지표를 시각화하여 사용자에게 제공한다. MLflow UI를 통해 다양한 모델 아키텍처 및 하이퍼파라미터 조건에서의 학습 결과를 직관적으로 비교·분석할 수 있다.

3.2. 시스템 구조

3.2.1. 개발언어

구분	언어	활용 목적
백엔드	**Python**	FastAPI 서버 개발, Celery 워커, PyTorch 학습 파이프라인 구현
프론트엔드	**TypeScript**	React 기반 웹 UI 개발, 모델 설계 캔버스(ReactFlow), 상태 관리
스타일링	**Tailwind CSS**	UI 스타일링, 반응형 웹 구현
설정/배포	**YAML / Dockerfile**	Docker 컨테이너 설정, 서비스 실행 환경 관리

표 1. 사용 언어 및 역할

3.2.2. 개발 도구

구분	도구	활용 목적
프레임워크 라이브러리	FastAPI, React, ReactFlow, PyTorch,	API 서버, UI 구성, 모델 학습, 데이터셋 처리

	HuggingFace Datasets	
비동기 처리	Celery, Redis	분산 학습 태스크 실행, 학습 상태 이벤트 전달
실험 관리	MLflow	학습 지표 및 하이퍼파라미터 기록, 실험 결과 비교
환경 관리	Docker	컨테이너 기반 실행 및 배포
개발 환경	Vite	프론트엔드 개발 환경 및 빌드

표 2. 개발 도구 및 활용 방법

3.2.3. 프론트엔드 (Front-end)

본 프로젝트의 프론트엔드는 React + ReactFlow + Tailwind를 기반으로 구현되었다. React는 컴포넌트 기반 UI 설계 방식을 제공하여 복잡한 화면을 모듈 단위로 분리·재사용할 수 있고, 유지보수와 확장성이 뛰어나다. ReactFlow는 노드 기반의 시각화 라이브러리로, 사용자가 직접 토큰 임베딩, 포지셔널 임베딩, 트랜스포머 블락 등을 드래그 앤 드롭 방식으로 배치하고 연결할 수 있도록 지원한다. 이를 통해 초심자도 직관적으로 모델 구조를 설계할 수 있다. Tailwind는 유틸리티 클래스 기반 CSS 프레임워크로, UI를 빠르게 구성하면서도 일관된 스타일을 유지할 수 있게 하여 개발 속도와 편의성을 높였다.

프론트엔드 화면은 크게 모델 설계 캔버스, 학습 제어 패널, 실험 결과 조회 페이지로 구성된다. 사용자는 설계된 구조를 JSON으로 변환해 백엔드에 전달하고, 학습 요청을 보낼 수 있으며, 학습 상태와 결과를 UI를 통해 확인할 수 있다.

3.2.4. 백엔드 (Back-end)

백엔드는 FastAPI 프레임워크를 기반으로 구축되었으며, 비동기 처리를 지원하여 학습 요청과 상태 조회를 효율적으로 처리할 수 있다. 서버는 Docker 컨테이너로 배포 가능하며, 확장성과 이식성을 고려해 설계되었다. 주요 구성 요소는 다음과 같다:

1. **REST API 서버 (FastAPI)**
    - /train, /events/{task_id}, /stop/{task_id}, /completed-models 등의 REST API를 제공한다. /train API는 사용자가 설계한 JSON과 학습 설정을 입력받아

Celery 태스크로 등록하고, /events/{task_id}는 Redis Pub/Sub을 통해 전달된 학습 상태를 SSE 형식으로 스트리밍한다. 또한 /stop/{task_id}를 통해 학습 중단 요청을 처리하며, /completed-models에서는 학습 완료된 모델 목록을 조회할 수 있다. 이러한 API 설계로 인해 사용자 요청과 학습 연산이 효율적으로 분리되었다.

메서드	엔드포인트	설명
POST	/train	모델 구조(JSON)와 설정(config)을 입력받아 Celery 학습 태스크 실행
GET	/events/{task_id}	Redis Pub/Sub 기반 SSE로 학습 진행 상태 스트리밍
POST	/stop/{task_id}	특정 학습 태스크 중단
POST	/inference	학습 완료된 모델을 불러와 텍스트 생성 수행
GET	/completed-models	학습 완료된 모델(.pt) 목록 및 구조(JSON) 확인

표 3. API 명세

2. 비동기 태스크 실행 (Celery + Redis)

- 워커는 Redis를 브로커로 사용하여 백엔드와 통신하며, 데이터셋 로딩, 배치 구성, 손실 계산, 옵티마이저 업데이트 등 학습 과정을 처리한다. JSON 기반 모델 구조는 factory.py를 통해 PyTorch 모듈로 동적으로 변환되며, CustomSequential 클래스를 활용해 Residual Connection과 병렬 연결을 지원한다.

3. 실험 추적 (MLflow 연동)

- 하이퍼파라미터, 손실 곡선, 모델 구조, 체크포인트 파일 등이 자동으로 저장되며, 사용자는 MLflow UI를 통해 서로 다른 실험 결과를 비교할 수 있다. 이를 통해 단순한 학습 기록을 넘어, 다양한 설계 선택이 성능에 어떤 영향을 미치는지 체계적으로 분석할 수 있도록 하였다.

3.3. 토크나이저 활용

본 연구에서는 다양한 언어 모델 구조를 실험하기 위하여 GPT-2, LLaMA-2, LLaMA-3에 대응하는 토크나이저를 구현하였다. 토크나이저는 텍스트를 정수 시퀀스로 변환하는 전처리 단계로, 이후 임베딩 층을 통해 고차원 벡터로 매핑되어 모델의 입력으로 사용된다. 각 모델별 구현 방식은 다음과 같다.

3.3.1. GPT-2 토크나이저

OpenAI에서 제공하는 byte-level BPE (Byte Pair Encoding) 기반 토크나이저를 사용하였다. 구현 시 tiktoken.get_encoding("gpt2")를 호출하여 어휘 집합(vocabulary) 50,257개를 로드하였다. 이 방식은 모든 유니코드 문자를 바이트 단위로 분해하여 토큰화하므로 <unk> 토큰이 불필요하며, 희귀 단어나 신조어도 안정적으로 처리할 수 있다.

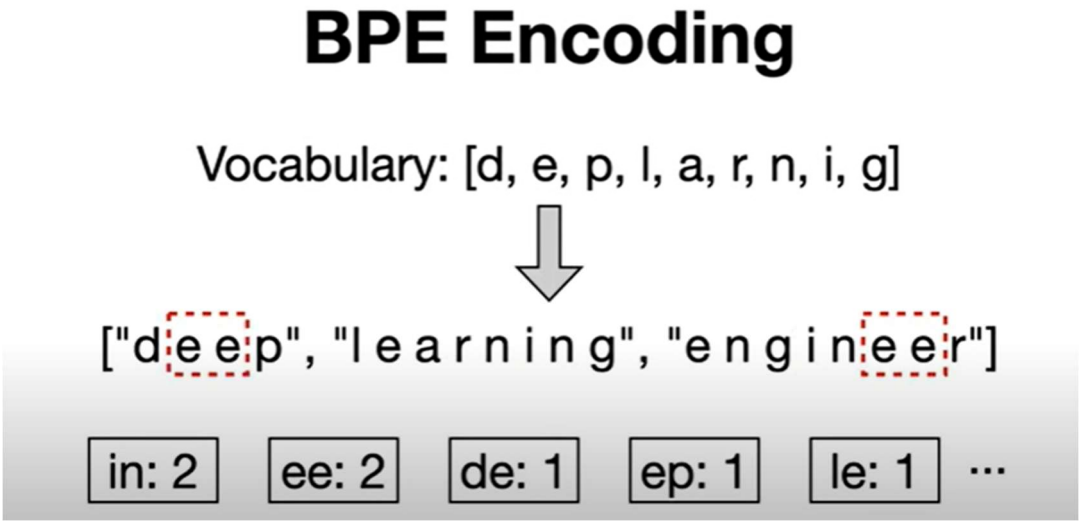

그림 3. BPE encoding 방법

3.3.2. LLaMA-2 토크나이저

Meta에서 공개한 LLaMA-2 모델은 SentencePiece 기반 토크나이저를 사용한다. 본 연구에서는 해당 .model 파일을 불러와 초기화하였으며, 약 32K 어휘 크기를 가진 서브워드 단위 토큰화를 수행하였다. SentencePiece는 공백을 독립적으로 처리(▁ 기호)하기 때

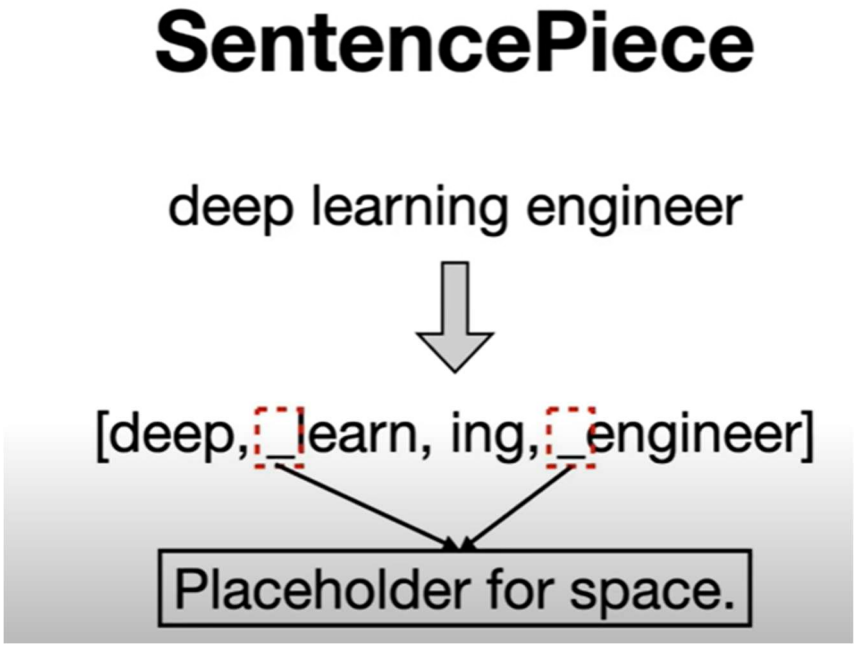

그림 4. Sentence piece 라이브러리

문에 다국어 환경에서도 안정적으로 동작하며, 원문 복원(디토크나이즈) 과정이 용이하다.

3.3.3. LLaMA-3 토크나이저

LLaMA-3는 기존 SentencePiece 대신 128K 어휘 규모의 BPE 기반 토크나이저를 도입하였다. 그러나 공식 토크나이저가 공개되지 않았기 때문에, 본 연구에서는 대체 방법으로 OpenAI의 tiktoken 라이브러리에서 제공하는 "cl100k_base" 인코딩을 활용하였다. 이는 GPT-4 계열 모델에서 사용되는 어휘 체계로, 구조적으로 LLaMA-3의 토크나이저와 유사하다. 이를 통해 동일한 입력 문장을 더 적은 토큰 수로 분할할 수 있어, 제한된 자원 환경에서도 긴 컨텍스트 창을 효율적으로 활용할 수 있도록 하였다.

구분	GPT-2 Tokenizer	LLaMA-2 Tokenizer	LLaMA-3 Tokenizer (대체: cl100k_base)
방식	BPE (Byte Pair Encoding), tiktoken `"gpt2"`	SentencePiece (Unigram/BPE 혼합), `tokenizer.model` 사용	BPE 기반, tiktoken `"cl100k_base"` 사용 (GPT-4 계열과 유사)
어휘 크기	약 50K	약 32K	약 128K
특징	영어 위주 최적화, 간단하고 빠름	다국어 지원, LLaMA-2 공개 모델과 동일	긴 문맥 처리 및 다국어 지원 강화, 최신 모델에서 활용
장점	가볍고 빠른 학습/실험에 적합	LLaMA-2와 동일 환경 재현 가능	LLaMA-3 호환성 확보, 긴 컨텍스트 학습 적합
단점	긴 문맥·다국어에서 한계	SentencePiece 설치 필요, 속도 느림	실제 Meta LLaMA-3 tokenizer와 완벽 동일은 아님

표 4. 토크나이저 비교 표

이와 같이 각 모델의 토크나이저를 실제 환경에 맞추어 구현함으로써, 연구자는 동일한 플랫폼 내에서 GPT-2, LLaMA-2, LLaMA-3 기반 실험을 유연하게 수행할 수 있다. 특히

LLaMA-3의 경우, 공식 배포 토크나이저의 부재를 tiktoken 기반 인코딩으로 보완함으로써 최신 모델 구조의 특성을 반영하였다.

3.4. 토큰 임베딩과 포지셔널 임베딩

3.4.1. 토큰 임베딩

토큰 임베딩(Token Embedding)은 텍스트를 토크나이저를 통해 정수 ID로 변환한 뒤, 이를 고차원 벡터 공간으로 매핑하는 과정이다. 각 토큰은 임베딩 행렬에서 자신만의 벡터 표현을 갖게 되며, 이러한 벡터는 모델이 의미적 유사성을 학습하는 데 사용된다. 본 프로젝트에서는 GPT-2, LLaMA-2, LLaMA-3 토크나이저를 통해 생성된 정수 시퀀스를 토큰 임베딩 레이어에 입력하여 학습에 활용하였다.

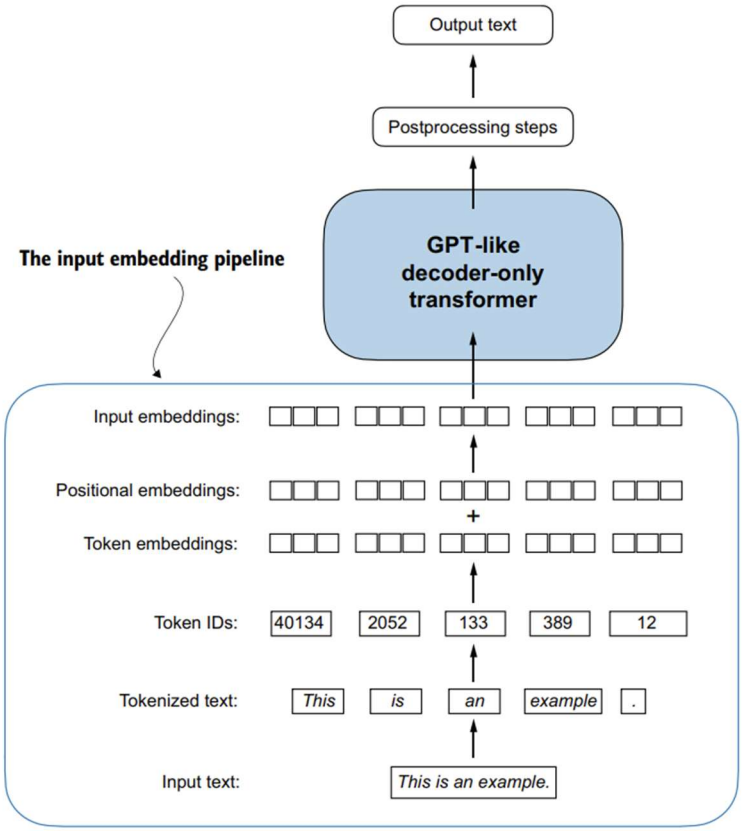

그림 5. 토큰 임베딩 및 포지셔널 임베딩 과정

본 연구에서 정의한 JSON 기반 모델 구조에서 토큰 임베딩 레이어는 아래와 같은 키들을 가진다.

키	역할 (의미)	예시
type	레이어 종류 지정	"tokenEmbedding"
data.id	노드 고유 ID (연결·디버깅용)	"tokenEmbedding-1"
data.vocabSize	전체 어휘 집합 크기	32000 (GPT-2: 50257, LLaMA-2: 32000)
data.embDim	각 토큰을 매핑할 임베딩 벡터 차원	768
data.inDim	(선택) 입력 차원. 일반적으로 embDim과 동일	768
data.outDim	(선택) 출력 차원. 일반적으로 embDim과 동일	768

표 5. 토큰 임베딩 레이어 JSON 키 정리

3.4.2. 포지셔널 임베딩

트랜스포머 모델은 병렬 연산을 통해 입력 토큰을 처리하기 때문에 순서 정보를 스스로 인식하지 못한다. 따라서 각 토큰이 문장에서 차지하는 위치 정보를 추가해주는 과정이 필요하다. 본 프로젝트에서는 이러한 목적을 위해 포지셔널 임베딩 모듈을 별도로 구현하고, 사용자가 JSON 기반 layer_stack에서 원하는 방식을 선택해 모델에 적용할 수 있도록 설계하였다.

포지셔널 임베딩은 토큰 임베딩으로 변환된 벡터에 위치 정보를 더하거나 변환을 가해, 모델이 단어의 의미뿐 아니라 단어 간의 순서적 관계까지 학습할 수 있도록 한다. 본 프로젝트에서 구현한 방식은 다음과 같다.

1. **Sinusoidal Embedding (사인/코사인 기반, GPT-2 계열에서 사용)**
    - 위치를 수학적 주기 함수인 사인(sin)과 코사인(cos)으로 변환하여 생성한다.
    - 각 차원마다 서로 다른 주기를 부여하여, 위치마다 고유한 패턴을 갖도록 설계된다.
    - 학습 파라미터가 필요 없기 때문에 메모리 효율적이며, 입력 길이가 달라져도 일반화가 가능하다.
    - 즉, 학습하지 않아도 되는 "고정형 위치 정보"로서, 실험적 교육 환경에 적

합하다.

2. **Learned Positional Embedding (학습형, LLaMA-2 계열에서 사용)**
    - 위치마다 고유한 벡터를 두고, 학습 과정에서 모델이 직접 최적화한다.
    - 데이터에 따라 위치 표현을 유연하게 조정할 수 있어, 모델 성능 향상에 기여할 수 있다.
    - 단, 새로운 입력 길이나 학습되지 않은 시퀀스 범위에서는 일반화가 어려울 수 있다.
    - 교육적 측면에서는, "고정형 vs 학습형" 차이를 실험적으로 비교할 수 있는 좋은 예시가 된다.
3. **Rotary Embedding (RoPE, 회전형, LLaMA-3 계열에서 사용)**
    - 쿼리(Query)와 키(Key) 벡터 자체에 회전 변환을 적용해 위치 정보를 주입한다.
    - 위치에 따라 벡터 공간에서의 각도가 변하므로, 토큰 간 상대적 거리와 순서를 자연스럽게 반영할 수 있다.
    - 긴 문맥 처리에 특히 강력하며, 최신 대규모 언어 모델들이 선호하는 방식이다.
    - RoPE는 단순히 위치 벡터를 더하는 것이 아니라 벡터 공간에서의 기하학적 변환을 이용한다는 점에서 다른 방법들과 구분된다.

이와 같은 포지셔널 임베딩은 토큰 임베딩과 결합되어 트랜스포머 블록으로 전달된다. 따라서 동일한 텍스트라도 위치가 달라지면 모델 입력이 달라지며, 이는 문맥과 순서 정보가 모델 내부 연산에 반영됨을 의미한다.

키	역할 (의미)	예시
type	레이어 종류 지정	"positionalEmbedding"
data.id	노드 고유 ID (연결·디버깅용)	"posEmbedding-1"
data.embDim	임베딩 벡터 차원 (Token Embedding과 동일)	768
data.ctxLength	문맥 길이 (최대 입력 토큰 수)	1024
data.method	위치 인코딩 방식 선택	"sinusoidal", "learned", "rope"

data.inDim	(선택) 입력 차원. 보통 embDim과 동일	768
data.outDim	(선택) 출력 차원. 보통 embDim과 동일	768

표3. Positional Embedding Layer JSON 키 정리

3.5. 트랜스포머 블록

본 프로젝트의 트랜스포머 블록(Transformer Block) 은 기본적인 구조인 Self-Attention → Feed-Forward → 정규화(Norm) → 잔차 연결(Residual) 흐름을 따르되, 다양한 변형과 실험을 지원할 수 있도록 모듈화하였다.

사용자는 웹 UI에서 드래그 앤 드롭 방식으로 블록의 개수와 하이퍼파라미터를 JSON 형식으로 지정할 수 있으며, 백엔드에서는 이를 기반으로 PyTorch 모듈을 동적으로 생성한다. 이러한 설계 덕분에 초심자도 복잡한 모델 구조를 직접 설계하고 학습 결과를 비교해볼 수 있다.

3.5.1. 어텐션 (Attention) 모듈

- Multi-Head Attention (MHA) 구조를 구현하여, 입력을 여러 헤드로 분리한 뒤 쿼리(Q), 키(K), 값(V) 연산과 스케일 조정된 점곱(Scaled Dot-Product)을 수행한다.
- 하이퍼파라미터로 헤드 수(n_heads), 임베딩 차원(emb_dim), QKV 바이어스 여부(qkv_bias) 등을 JSON에서 손쉽게 설정 가능하다.
- 최신 연구 동향을 반영하여, GQA(Grouped Query Attention) 와 RoPE(Rotary Embedding) 같은 변형 기법도 선택할 수 있도록 설계하였다.
- 이로써 사용자는 단순 MHA부터 최신 모델에서 사용하는 어텐션 구조까지 직접 비교·실험할 수 있다.

3.5.2. 피드포워드 네트워크 (Feed-Forward Network, FFN)

- 각 토큰별 벡터를 독립적으로 처리하는 2단계 선형 변환 구조로 구현하였다.

- 중간 계층에는 다양한 활성화 함수(Activation) 를 적용할 수 있으며, ReLU, GELU, SwiGLU 등 최신 함수들을 지원한다.
- 또한 FFN 크기(hidden_dim)와 게이팅 여부(gated) 도 JSON에서 지정 가능해, 학습자가 구조적 변화를 쉽게 실험할 수 있다.

3.5.3. 정규화 (Normalization)

- 학습 안정성과 수렴 속도를 높이기 위해 LayerNorm과 RMSNorm 두 방식을 지원한다.
- JSON 설정값에 따라 원하는 정규화 방식을 선택할 수 있어, 정규화 기법 변화가 성능에 어떤 영향을 주는지 직접 확인할 수 있다.

3.5.4. 잔차 연결 (Residual Connection)

- 입력 텐서를 블록 출력에 더하는 skip connection을 기본 지원한다.
- 특히, 본 프로젝트에서 구현한 CustomSequential은 source_id를 통해 특정 레이어 출력과 현재 출력을 연결할 수 있게 하여, 단순 직렬 구조뿐 아니라 잔차 및 병렬 연결이 포함된 다양한 네트워크 설계를 가능하게 한다.
- 이는 Transformer 아키텍처 실습에서 중요한 실험적 유연성을 제공한다.

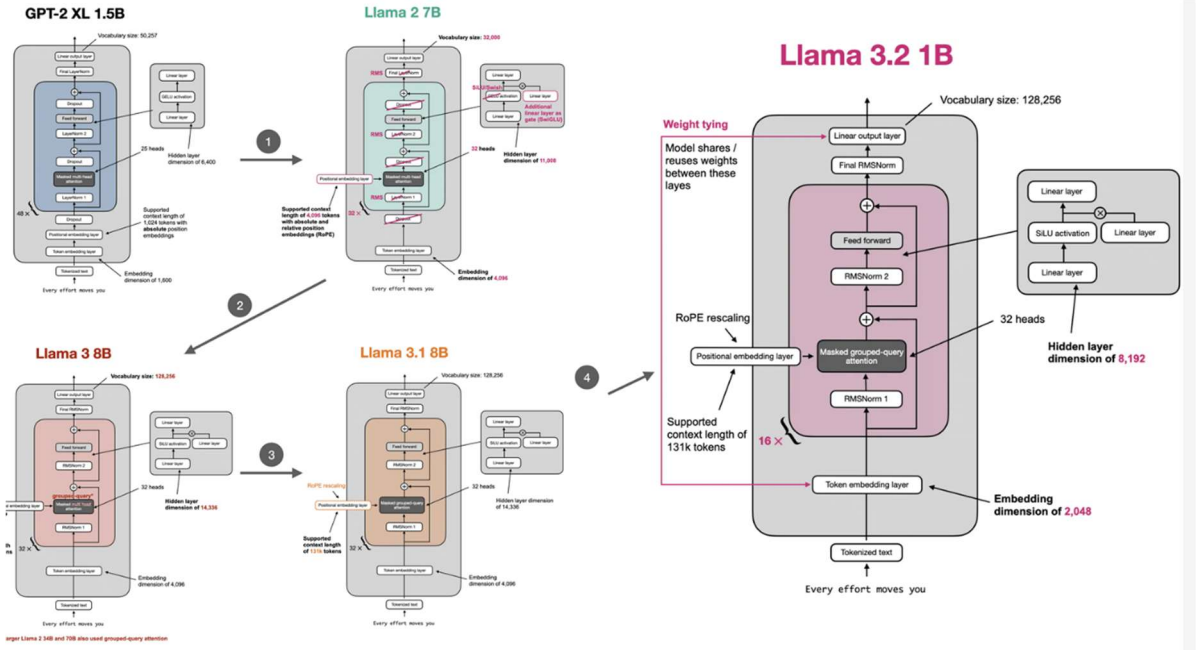

그림 7. Transformer 블록 설계

본 프로젝트에서는 그림 7과 같이들 모델의 핵심 차이를 반영할 수 있도록 설계하였다.

- 포지셔널 임베딩: Sinusoidal, Learned, RoPE 모두 모듈화해 선택 가능
- 어텐션: 기본 MHA 외에도 GQA를 지원하도록 확장
- 정규화: LayerNorm, RMSNorm 모두 선택 가능
- FFN: ReLU, GELU, SiLU 등 다양한 활성화 함수 지원

즉, 이 그림 속 모델들이 갖는 구조적 차이를 JSON 기반 layer_stack으로 설정해 재현할 수 있으며, MLflow를 통해 학습 성능을 비교하는 것이 가능하다.

키	역할 (의미)	예시
type	레이어 종류 지정	"transformerBlock"
data.id	블록 고유 ID (연결·디버깅용)	"transformerBlock-1749021257305"
data.label	블록의 이름(시각화·UI용)	"Transformer Block"
data.inDim	입력 벡터 차원	768
data.outDim	출력 벡터 차원	768
data.numOfBlocks	동일 블록 반복 개수	12
children	블록 내부 서브 레이어 리스트	[normalization, sdpAttention, dropout, residual, feedForward, ...]

표 6. Transformer Block JSON 키 정리

3.6. 모델 아키텍쳐 조립

본 프로젝트에서는 JSON 기반 layer_stack 을 입력으로 받아, 이를 PyTorch 모듈로 동적으로 생성하는 방식을 적용하였다. 이러한 구조는 사용자가 직접 코드를 수정하지 않고도 다양한 모델 아키텍처를 직관적으로 설계·실험할 수 있도록 지원한다.

3.6.1. JSON 기반 설계

- 사용자는 웹 UI(ReactFlow)에서 레이어를 드래그 앤 드롭하여 모델을 구성한다.각 레이어의 유형(type)과 하이퍼파라미터(n_heads, emb_dim 등)는 속성 값으로 입력되며 최종적으로 JSON 형식(layer_stack)으로 백엔드에 전달된다.

키	역할 (의미)	예시
Model	설계 프리셋/의도 표기(가독성·기본값 힌트)	"llama2"
epochs	학습 반복 횟수	10
batch_size	배치 크기	32
vocab_size	토큰 어휘 크기(출력 차원)	32000
context_length	최대 시퀀스 길이	128
emb_dim	임베딩/모델 차원(d_model)	4096
n_heads	어텐션 헤드 수	32
n_blocks	트랜스포머 블록 개수	12
hidden_dim	FFN 내부 차원	110000
dtype	연산 정밀도	“bf16”
drop_rate	드롭아웃 비율	0.1
qkv_bias	Q/K/V 바이어스 전역 기본값	true

표 7. config 필드

키	역할 (의미)	예시
type	레이어 종류	tokenEmbedding", "positionalEmbedding", "transformer_block", "linear
data.id	노드 고유 ID(연결·디버깅에 사용)	"tok-1", "blk1"
data.*	레이어별 하이퍼파라미터	
children	토큰 어휘 크기(출력 차원)	32000
data.source	Residual 시 추가로 더할 원천 노드 ID	"blk1-in"

표 8. 모델 필드 하이퍼파라미터

3.6.2. 동적 모델 생성 (factory.py)

- parse_node() 함수는 JSON 노드를 해석하여 대응되는 PyTorch 모듈을 생성한다.
    - 예: {"type": "attention", "n_heads": 8} → MultiHeadAttentionCombinedQKV 모듈 생성
- build_model_from_json() 함수는 이렇게 생성된 모듈들을 연결하여 최종 모델을 조립한다.

3.6.3. CustomSequential 클래스

- PyTorch의 기본 nn.Sequential을 확장하여 Residual Connection 과 병렬 연결을 지원하도록 구현하였다. 각 레이어의 출력은 ID로 캐시되며, 이후 모듈에서 source_id를 참조해 skip connection을 적용할 수 있다. 이로써 단순 직렬 구조뿐 아니라 Transformer 특유의 Residual + Norm 구조까지 자연스럽게 지원한다.

3.6.4. 아키텍처 완성

- 최종 모델은 다음과 같은 흐름으로 구성된다.
  토큰 임베딩 + 포지셔널 임베딩 → N개의 Transformer Block → 출력층
- 사용자가 입력한 JSON 구조에 따라 GPT-2, LLaMA-2, LLaMA-3 등 다양한 아키텍처를 손쉽게 재현할 수 있다.

3.7. 데이터셋과 데이터 로더 (Dataset & Dataloader)

본 프로젝트에서는 PyTorch와 HuggingFace datasets 라이브러리를 활용하여 학습용 데이터셋을 구성하였다. 데이터 준비 과정은 크게 텍스트 토큰화 → 시퀀스 분할 → 배치화 (DataLoader) 순서로 이루어진다.

3.7.1. 데이터셋 로딩

1. datasets.load_dataset() API를 이용해 공개 코퍼스를 불러왔다.

2. dataset_name, dataset_config을 지정해 다양한 데이터셋을 선택할 수 있으며, 기본값으로 Tiny Shakespeare, WikiText 등 소규모 데이터셋을 제공한다.

3.7.2. 토큰화 및 시퀀스 분할

- 텍스트는 선택된 토크나이저(GPT-2, LLaMA-2, LLaMA-3)에 따라 정수 ID 시퀀스로 변환된다.
- context_length를 기준으로 슬라이딩 윈도우(stride)를 적용해 시퀀스를 분할한다.
  - 예: context_length = 128, stride = 64 → 겹치는 시퀀스 샘플 생성
- 각 샘플은 input_ids(입력)와 target_ids(다음 토큰 예측용 레이블)로 저장된다.

3.7.3. GPTDatasetV1 클래스

- 커스텀 Dataset 클래스를 구현해 텍스트를 자동으로 토큰화하고 슬라이딩 윈도우 방식으로 분할한다.
- __getitem__은 (input_tensor, target_tensor) 쌍을 반환하여, 학습 시 CrossEntropyLoss 계산에 바로 활용할 수 있다.

3.7.4. DataLoader 구성

- PyTorch DataLoader를 사용해 배치 단위로 데이터를 공급한다.
- batch_size, shuffle, drop_last 등의 옵션을 제공한다.
- 학습(train)·검증(val) 데이터셋을 분리하여 모델의 일반화 성능을 평가할 수 있도록 하였다.

키	역할 (의미)	예시
dataset	사용할 데이터셋 이름	"tiny_shakespeare"
dataset_config	데이터셋 서브 설정	"default"

context_length	시퀀스 최대 길이	128
stride	시퀀스 분할 시 이동 간격	64
batch_size	배치 크기	32

표 9. Dataset JSON/config 주요 필드

3.8. 학습 파이프라인 (Training Pipeline)

본 프로젝트의 학습 파이프라인은 사용자 요청 → 데이터셋 준비 → 모델 학습 → 상태 추적 및 시각화 의 흐름으로 구성되어 있다. 핵심 구성 요소는 다음과 같다.

3.8.1. 손실 함수 (Loss Function)

- CrossEntropyLoss 사용.
- 각 위치의 예측 토큰 확률 분포와 실제 다음 토큰 ID를 비교해 손실을 계산한다.
- 이는 언어 모델 학습의 표준 방식으로, 모델이 문맥 기반으로 올바른 다음 단어를 예측하도록 유도한다.

3.8.2. 옵티마이저 (Optimizer)

- 기본: Adam (torch.optim.Adam)
- 학습률(lr), 베타 값, weight decay 등의 하이퍼파라미터를 JSON config로 조정 가능.
- 향후 확장 가능성: AdamW, DeepSpeed 최적화 기법 지원.

3.8.3. 학습 루프 (Training Loop)

- for epoch in range(epochs): 구조로 반복 학습 진행.
- 각 배치에 대해:
    1. 모델 순전파(forward)

2. 손실(loss) 계산
3. 역전파(backward) 수행
4. 옵티마이저로 가중치 갱신

- GPU 사용 가능 시 자동으로 cuda 디바이스에서 실행.
- SSE 이벤트를 통해 각 step/epoch의 손실을 실시간 전송.

3.8.4. API 기반 학습 요청과 상태 추적

- **학습 요청**:
  - 사용자 → FastAPI POST /train 호출 시,
  - Celery 태스크로 등록되어 Redis broker를 통해 Worker가 실행된다.
- **상태 조회**:
  - FastAPI /status 혹은 /events/{task_id} SSE 엔드포인트를 통해
  - 현재 epoch, step, 손실 값 등이 실시간으로 스트리밍된다.
- **학습 중지**:
  - /stop/{task_id} API로 진행 중인 학습을 중단할 수 있도록 구현.

3.8.5. 실험 추적 (Experiment Tracking)

- MLflow를 연동하여 각 학습 실행(run)의 로그와 하이퍼파라미터, 지표를 자동 저장.
- 학습 곡선(Loss vs Epoch), 모델 구조, 최종 체크포인트를 MLflow UI에서 확인 가능.
- 학습 완료된 모델은 /completed-models API로 조회 가능.

3.9. 실험 추적과 시각화 (Experiment Tracking)

본 프로젝트에서는 학습 과정과 결과를 실시간으로 추적하고 다양한 모델 간 성능을 비교할 수 있도록 SSE 이벤트 스트리밍과 MLflow 를 연동하였다. 학습이 진행되는 동안 손실 값과 현재 epoch, step 정보는 Redis Pub/Sub 을 통해 전달되며, FastAPI 의 /events/{task_id} SSE 엔드포인트에서 스트리밍된다. 현재는 SSE 스트림까지만 구현되어 있으며, 프론트엔드에서 이를 구독하여 그래프 형태로 시각화할 수 있는 구조를 제공한다. 또한 /stop/{task_id} API 를 통해 학습 중단 요청도 가능하다.

각 학습 실행(run)은 고유한 Run ID 로 관리되며, MLflow 서버에 자동 저장된다. 이때 학습에 사용된 하이퍼파라미터(epochs, batch_size, context_length, n_heads 등), 손실과 정확도 같은 학습 지표, 그리고 JSON 기반 모델 구조와 PyTorch 체크포인트 파일이 함께 기록된다. MLflow UI 를 통해 Run ID 별 손실 곡선과 하이퍼파라미터를 직관적으로 비교할 수 있어, 서로 다른 모델 설정에 따른 성능 차이를 명확히 확인할 수 있다.

특히 MLflow 에 기록된 결과는 단일 곡선 확인을 넘어 다양한 비교 실험에도 활용된다. 예를 들어 Sinusoidal 과 Learned Positional Embedding, 기본 MHA 와 GQA Attention, LayerNorm 과 RMSNorm 의 차이를 비교 분석할 수 있으며, 이를 통해 모델 설계 선택이 성능에 미치는 영향을 효과적으로 평가할 수 있다.

3.10. 서비스 시나리오

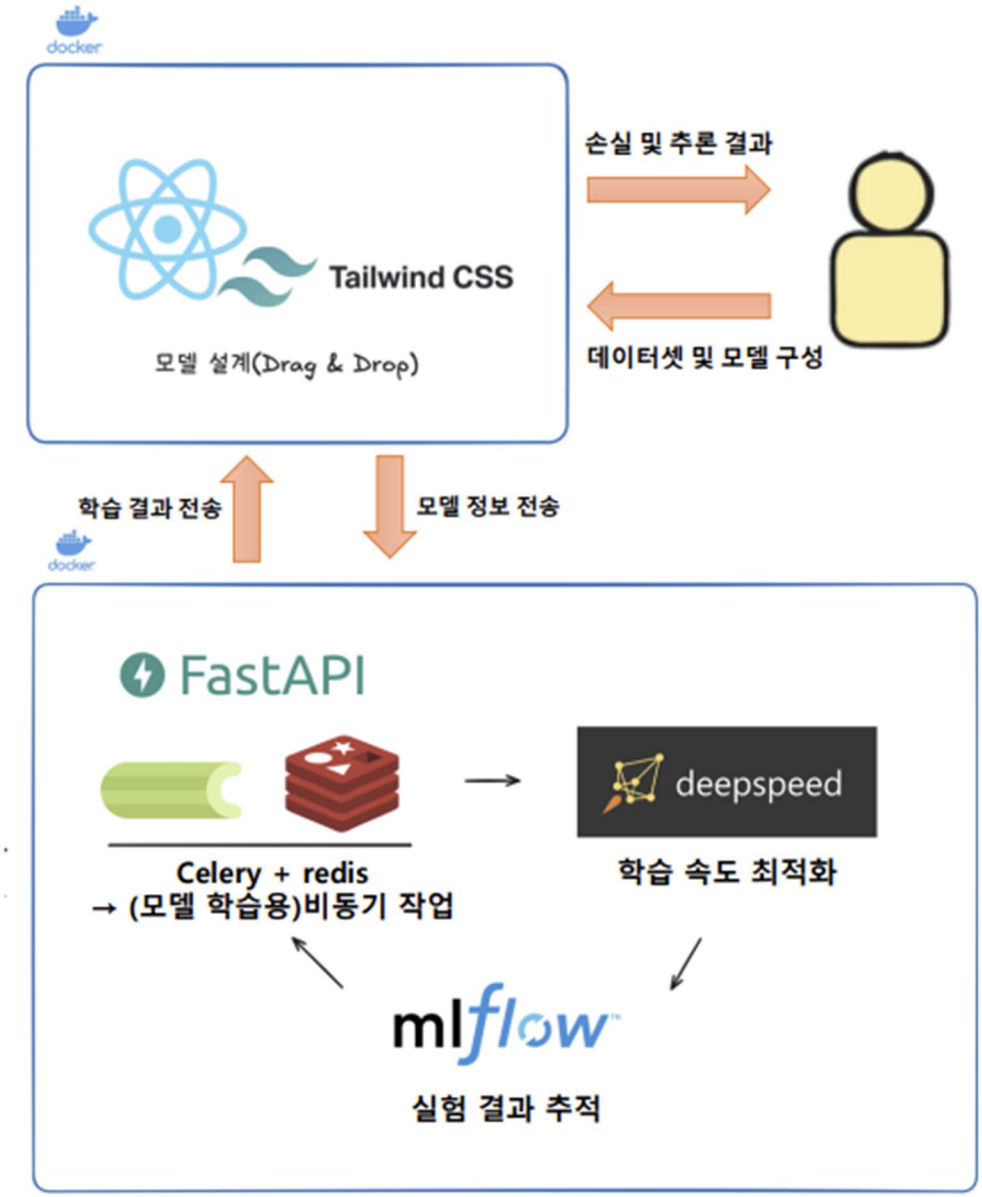

그림 8. 시스템 아키텍처

본 프로젝트의 플랫폼은 모델 설계 → 학습 요청 → 진행 확인 → 결과 활용의 흐름을 갖는다. 실제 사용자의 이용 과정을 시나리오로 정리하면 다음과 같다.

3.10.1. 모델 설계

- 사용자는 웹 UI(ReactFlow)에서 토큰 임베딩, 포지셔널 임베딩, 트랜스포머 블록 등을 드래그 앤 드롭으로 배치한다. 각 블록의 하이퍼파라미터(예: embDim, n_heads, dropRate)를 패널에서 입력하면 자동으로 JSON 형식의 모델 구조(layer_stack)가 생성된다.

3.10.2. 학습 요청

- 사용자가 설계를 완료하면 POST /train API를 호출하여 학습을 시작한다. FastAPI는 요청을 Celery 태스크로 등록하고, 워커는 지정된 데이터셋과 파라미터로 모델 학습을 수행한다.

3.10.3. 진행 상태 확인

- 학습이 진행되는 동안 손실 값과 진행률은 Redis Pub/Sub을 통해 SSE 이벤트로 전송된다. 사용자는 /events/{task_id}를 구독하여 학습 상태를 확인할 수 있으며, 필요 시 /stop/{task_id} API를 호출해 중단할 수도 있다.

3.10.4. 실험 추적과 결과 분석

- 모든 학습 실행은 MLflow에 기록되며, 하이퍼파라미터와 손실 곡선을 UI에서 비교할 수 있다. 예를 들어 Sinusoidal vs RoPE, MHA vs GQA, LayerNorm vs RMSNorm 같은 구조적 차이를 직관적으로 비교할 수 있다.

3.10.5. 모델 활용

- 학습이 완료된 모델은 /completed-models API로 확인할 수 있다. 사용자는 /inference API를 호출해 입력 텍스트에 대한 추론을 수행하며, 결과를 기반으로 성능을 직접 체험할 수 있다.

4. 연구 결과 분석 및 평가

4.1. 최종 완성 기능 목록

최종 결과보고서 작성 시점에서 본 프로젝트를 통해 완성된 기능은 다음과 같다.

- **웹 서비스(UI)**
    - ReactFlow 기반 모델 설계 캔버스 제공
    - 드래그 앤 드롭으로 TokenEmbedding, PositionalEmbedding, Transformer Block 구성
    - 노드별 하이퍼파라미터 입력 및 수정 가능
    - 학습 제어(학습 시작, 중단, 결과 조회) 기능 지원
- **백엔드 API**
    - /train : JSON 기반 모델 구조를 입력받아 학습 태스크 실행
    - /events/{task_id} : SSE(Server-Sent Events)를 통한 학습 진행 상태 스트리밍
    - /stop/{task_id} : 특정 학습 태스크 중단
    - /inference : 학습 완료된 모델을 이용한 텍스트 생성
    - /completed-models : 학습 완료 모델 목록 조회
- **모델 학습 파이프라인**
    - JSON → PyTorch 동적 모델 변환 (factory.py)
    - GPTDatasetV1 + DataLoader로 데이터셋 분할 및 배치 구성
    - CrossEntropyLoss, Adam 기반 학습 루프 구현
    - Residual Connection, GQA, RoPE 등 다양한 옵션 실험 가능
- **실험 추적 및 시각화**
    - Redis Pub/Sub 기반 학습 로그 스트리밍
    - MLflow 연동으로 Run ID별 지표 기록 및 성능 비교

- Loss 곡선, 하이퍼파라미터별 성능 차이 확인

4.2. 성능 평가

본 시스템의 성능은 크게 학습 안정성, 모듈화 유연성, 실험 비교 용이성 측면에서 평가하였다.

4.2.1 학습 안정성

- 소규모 데이터셋(Tiny Shakespeare, WikiText)을 대상으로 학습 시 손실 곡선이 안정적으로 감소함을 확인하였다.

4.2.2 모듈화 유연성

- 동일한 JSON 구조에서 Positional Embedding 방식을 Sinusoidal, Learned, RoPE로 변경하며 실험 가능함을 확인하였다.

4.2.3 실험 비교 및 시각화

- MLflow UI를 통해 서로 다른 실험의 손실 곡선을 직관적으로 비교할 수 있었다.
- n_heads=8 vs n_heads=12, drop_rate=0.1 vs drop_rate=0.3과 같은 차이를 시각적으로 확인 가능하였다.

4.3. 결과물

최종적으로 완성된 결과물은 소규모 언어 모델 학습 플랫폼으로, 사용자가 직접 모델을 설계하고 학습하며 결과를 분석할 수 있는 통합 환경을 제공한다.

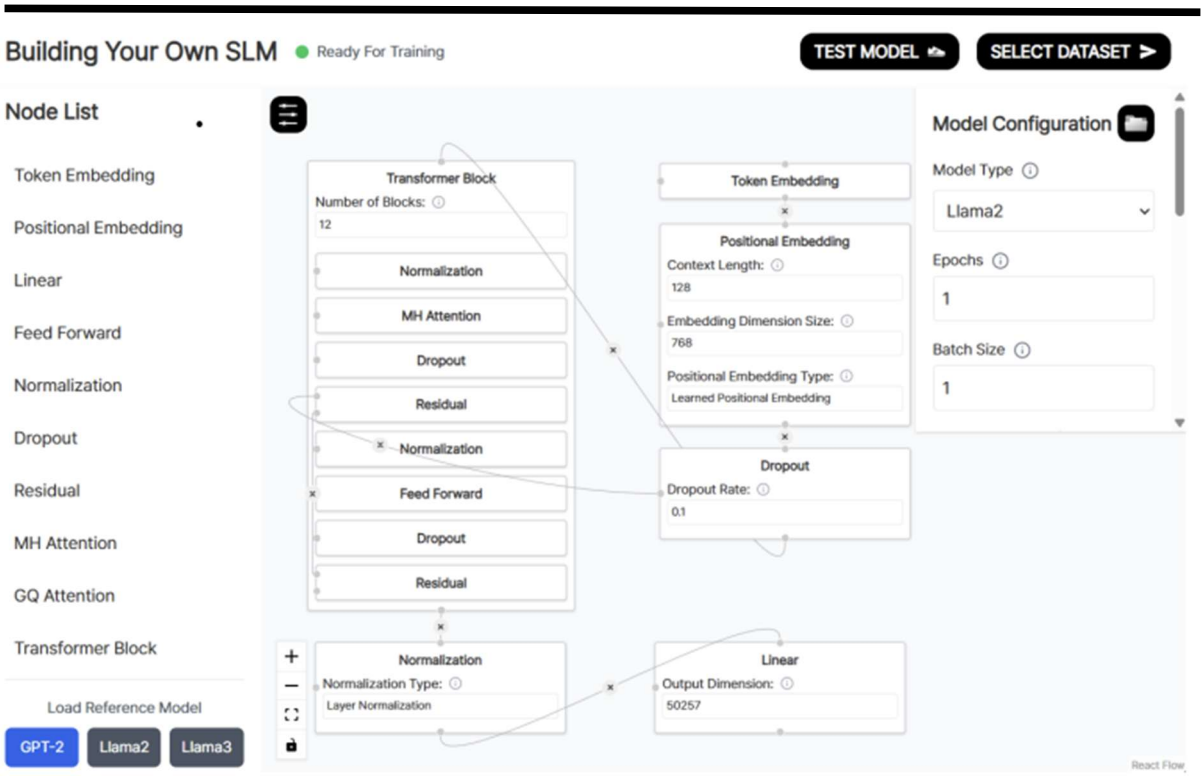

그림 9. 사용자 화면 1

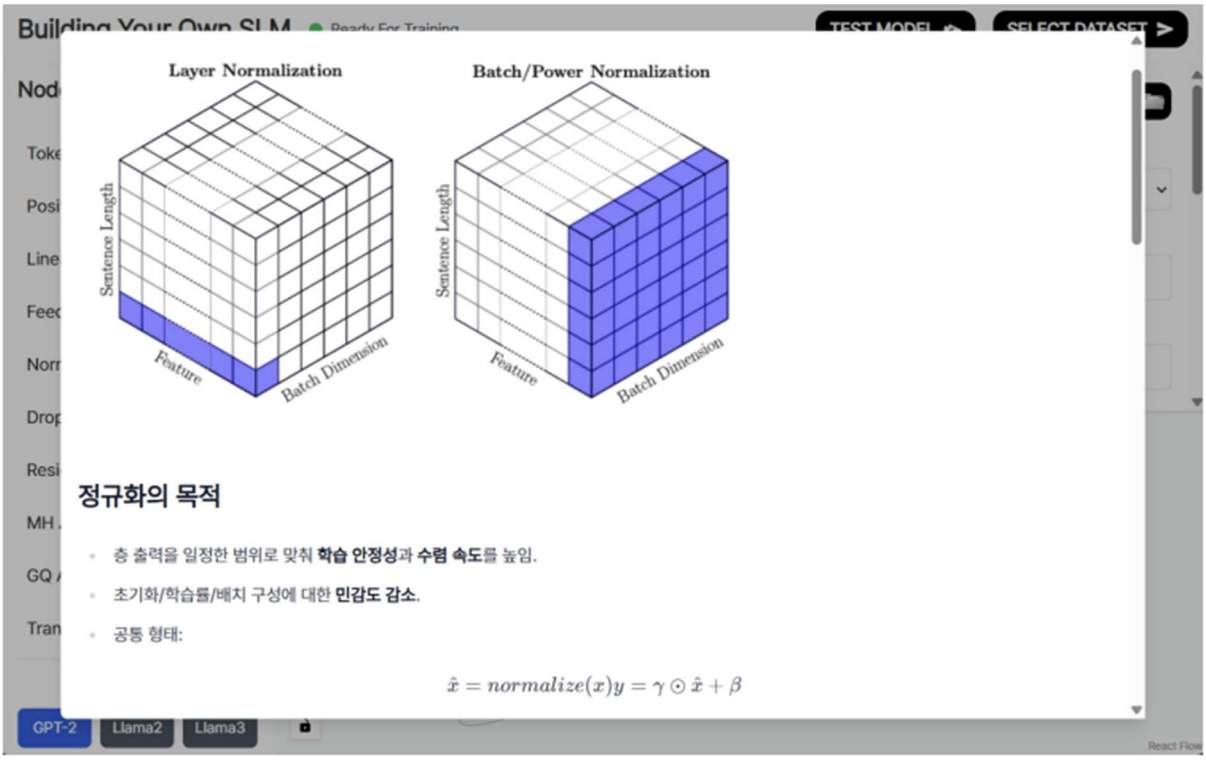

그림 10. 사용자 화면 2

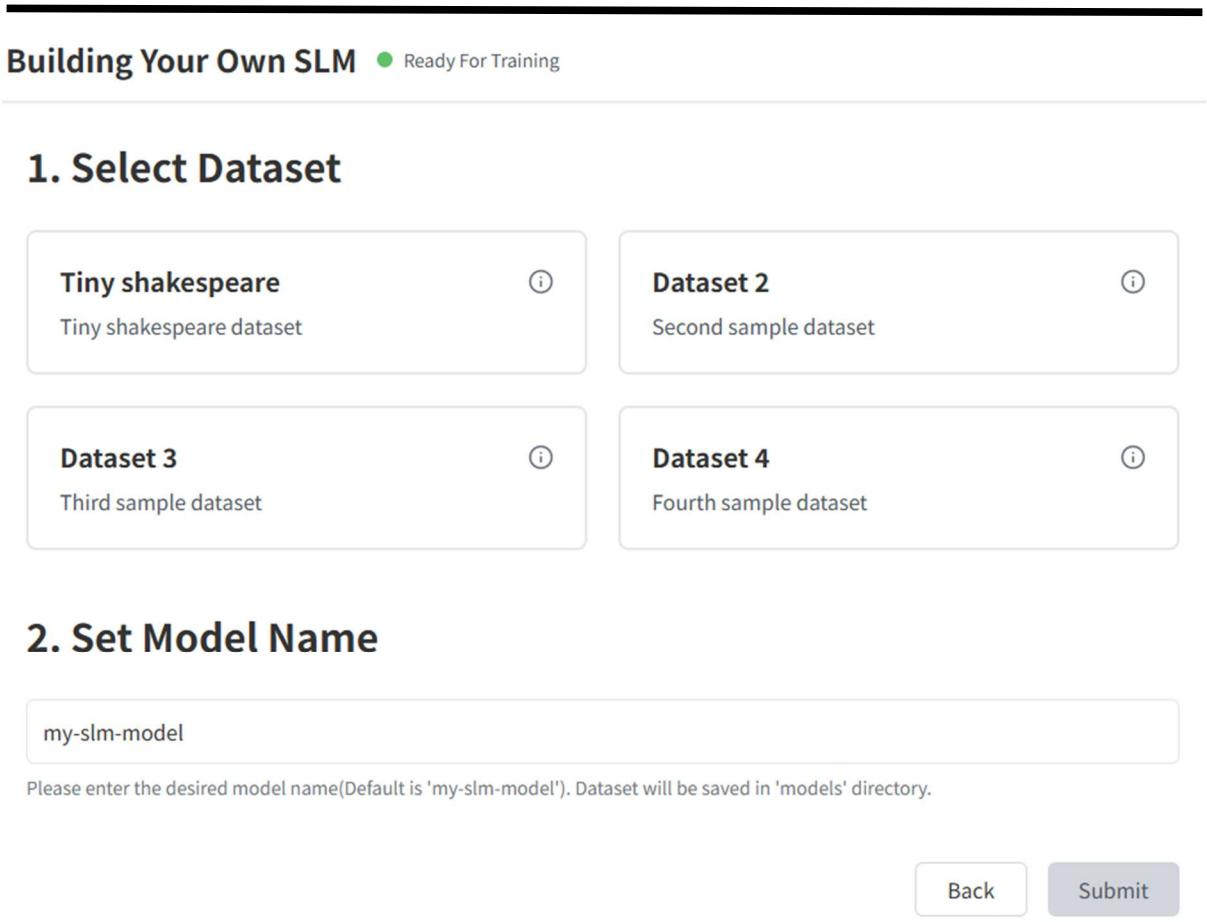

그림 11. 사용자 화면 3

- 사용자는 웹 UI에서 직관적으로 모델을 구성할 수 있으며,
- FastAPI + Celery + Redis 기반 백엔드에서 학습을 수행하고,
- MLflow를 통해 학습 과정을 기록하고 결과를 비교할 수 있다.

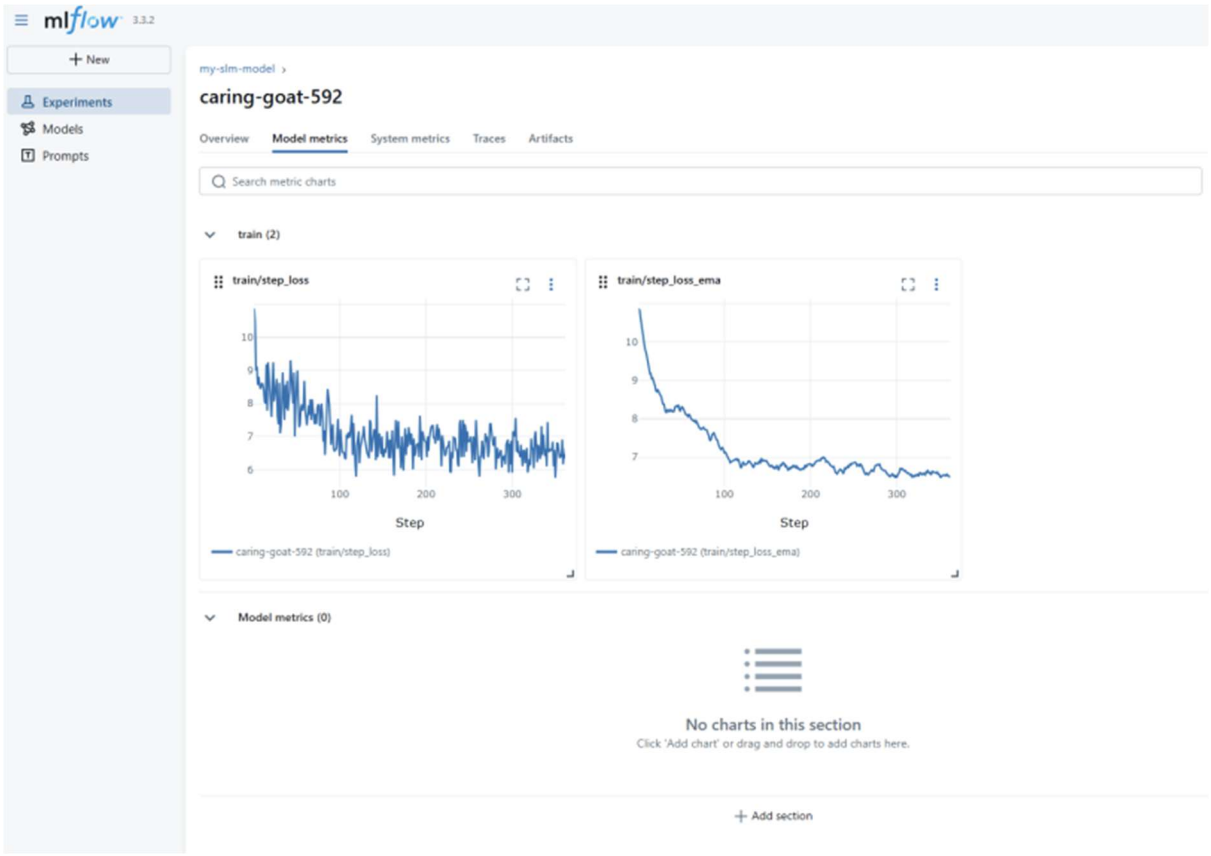

그림 12. 사용자 화면 4

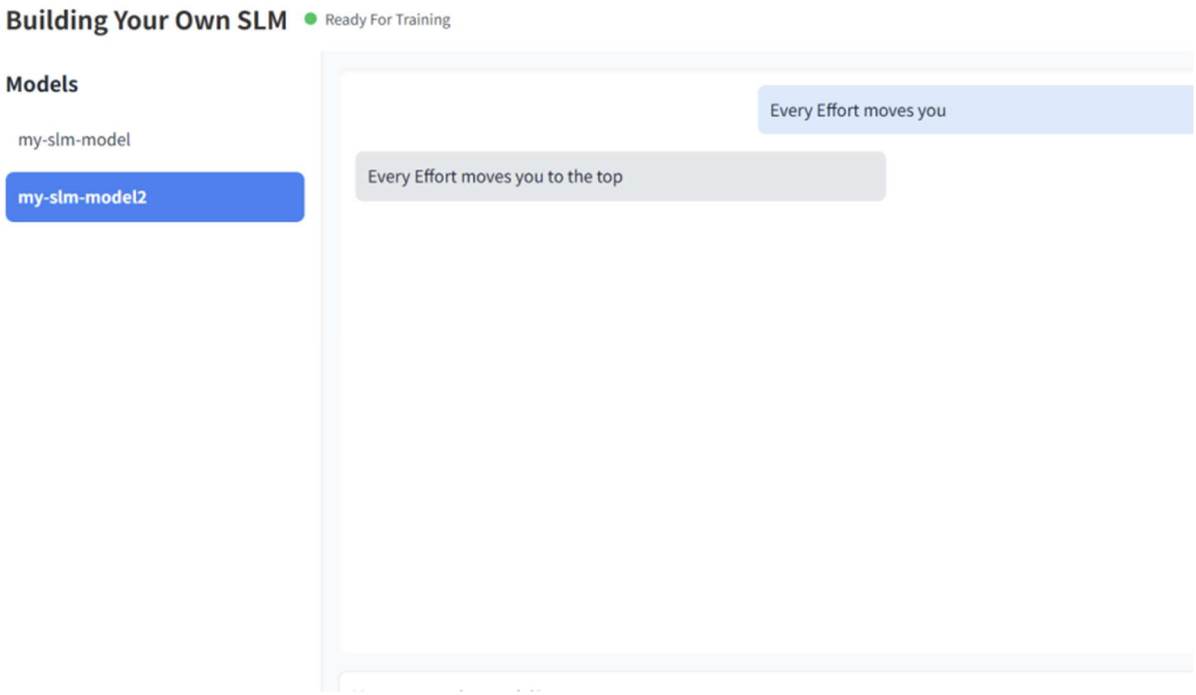

그림 13. 사용자 화면 5

이를 통해 초심자도 GPT-2, LLaMA 등 최신 언어 모델 구조를 직접 실험하고 학습 과정을 체험할 수 있으며, 교육적 실습 플랫폼으로서의 활용 가능성을 입증하였다.

5. 결론 및 향후 연구 방향

본 프로젝트는 소규모 언어 모델(Small Language Model, SLM)의 학습 과정을 직관적으로 설계·실험할 수 있는 웹 기반 플랫폼을 구현하는 것을 목표로 하였다. 사용자는 ReactFlow 기반 UI 를 통해 토큰 임베딩, 포지셔널 임베딩, 트랜스포머 블록 등을 드래그 앤 드롭으로 배치하고, JSON 구조를 생성하여 FastAPI·Celery 백엔드로 학습을 요청할 수 있다. 백엔드에서는 PyTorch 를 통해 동적으로 모델을 생성하고 학습을 수행하며, SSE 를 통한 실시간 상태 전송과 MLflow 기반 실험 기록 기능을 제공한다. 이를 통해 초심자도 복잡한 코드 작성 없이 언어 모델의 내부 동작을 학습하고 실험할 수 있는 환경을 완성하였다.

그러나 프로젝트 진행 과정에서 여러 한계도 존재하였다. 우선 GPU 서버 자원이 부족하여 대규모 데이터셋이나 장기 학습 실험을 수행하지 못했고, 학습 속도와 성능 모두 소규모 예시 수준에 머무를 수밖에 없었다. 또한 프론트엔드에서 학습 진행 상태를 실시간 그래프로 시각화하는 기능은 구조적 기반(SSE 스트리밍)까지만 구현되었고, 실제 차트 UI 와 연동하는 부분은 미완성으로 남았다. 사용자 관리, 실시간 로그 모니터링 등 다양한 기능도 구현하지 못한 점이 아쉬움으로 남는다.

향후 연구에서는 이러한 한계를 극복하기 위해 GPU 학습 환경을 갖추고, 더 큰 데이터셋(WikiText, OpenWebText 등)을 대상으로 학습을 확장하는 것이 필요하다. 또한 분산 학습 및 혼합 정밀도 학습(DeepSpeed, FSDP)을 도입하여 효율적인 자원 활용을 모색할 수 있다. 프론트엔드 측면에서는 학습 로그와 손실 곡선을 실시간 차트로 표현하고, 사용자 계정 및 학습 이력 관리 기능을 강화하여 교육 플랫폼으로서의 완성도를 높일 수 있다. 마지막으로, Transformer 외에도 다양한 아키텍처(CNN, RNN, 최신 LLM 기법)를 모듈화하여 선택할 수 있도록 확장한다면, 본 플랫폼은 교육뿐 아니라 연구 실험 도구로도 활용 가능할 것이다.

종합하면, 본 프로젝트는 제한된 환경 속에서도 언어 모델 학습 과정을 시각적으로 이해하고 직접 실험할 수 있는 플랫폼을 구축했다는 점에서 의미가 있다. 향후 기능

확장과 성능 개선을 통해 "누구나 쉽게 언어 모델을 설계하고 학습할 수 있는 교육·연구용 통합 플랫폼"으로 발전시켜 나가는 것이 본 과제의 최종적인 방향이다.

6. 참고 문헌

논문 내용에 직접 관련이 있는 문헌에 대해서는 관련이 있는 본문 중에 참고문헌 번호를 쓰고 그 문헌을 참고문헌란에 인용 순서대로 기술한다. 참고문헌은 영문으로만 표기하며 학술지의 경우에는 저자, 제목, 학술지명, 권, 호, 쪽수, 발행년도의 순으로, 단행본은 저자, 도서명, 발행소, 발행년도의 순으로 기술한다.

〈단행본〉

[1] S. Raschka, *Build a Large Language Model From Scratch*. OceanofPDF, 2024

〈논문지〉

[2] T. Brown et al., "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[3] Meta AI, "LLaMA: Open and Efficient Foundation Language Models," arXiv:2302.13971, 2023.

〈WEB Site 〉

[4] MLflow. (2024). *MLflow: Open source platform for the machine learning lifecycle*. [Online]. Available: https://mlflow.org

S. Raschka, *Build a Large Language Model From Scratch — Official Code Repository*, GitHub, 2024. [Online]. Available: https://github.com/rasbt/LLMs-from-scratch