서버리스 LLM 콜드 스타트 최적화: 프루닝된 모델 우선 서빙과 점진적 전체 모델 복구

ProgressiveServe: Pruned Layer Priority Serving with Gradual Full Model Recovery for Serverless LLM Cold Start Reduction

DevEwha 2 차 보고서

캡스톤디자인과창업프로젝트 그로쓰

2025 년 11 월 06 일

박나담 2276113
이나경 2270053
이주원 2276242

목차

1. 팀 정보

1.1. 과제명

서버리스 LLM 콜드스타트 최적화: 프루닝 된 모델 우선 서빙과 점진적 전체 모델 복구

1.2. 팀 정보 (팀 번호, 팀 이름 등)

11 팀 DevEwha

1.3. 팀 구성원

박나담(2276113), 이나경(2270053), 이주원(2276242)

2. 과제 요약

2.1. 문제 정의

2.1.1. 문제 배경

산업통상자원부 의뢰로 실시한 설문 조사에 따르면 대기업의 AI 활용률은 65.1%, 중소기업은 35%로 격차가 큽니다. 국내 다수의 중소기업은 자체 서버·GPU 인프라 부족, 초기 투자 비용, 전담 인력 부재 등으로 AI 도입에 어려움을 겪기 때문에, 클라우드의 서버리스(Serverless) 방식으로 서비스를 구축하려는 수요가 증가하고 있습니다. 그러나 현재의 서버리스 환경은 대규모 언어모델(LLM) 실시간 서비스에 바로 적용하기 어렵습니다. 그 결과, 중소기업은 고객 응대 자동화·지식 검색·업무 지원 등에서 AI 활용 격차를 지속적으로 겪고 있습니다.

2.1.2. Target Customer 정의

- 대상: LLM 기반 QA 챗봇을 운영하려는 국내 중소기업
- 특성
  - ○ GPU/서버 구매·운영이 어렵고, 인프라/ML 엔지니어 인력이 제한적
  - ○ 예측 가능한 월 비용과 빠른 구축·운영을 원함
  - ○ 기업 지식(FAQ/문서)과 외부 지식 검색을 결합한 실시간 응답 품질이 중요
  - ○ 트래픽 변동(이벤트/프로모션/업무시간)에 탄력적 대응 필요

2.1.3. 핵심 Pain Points (해결해야 할 문제)

1. 콜드 스타트 지연 단축

서버리스 환경에서는 요청이 들어오면 컨테이너 생성→모델 파라미터 다운로드→컴파일/로딩→추론이 직렬로 진행되어 콜드 스타트 지연이 큽니다. *ParaServe* [1] 논문에 따르면 서버리스 환경에서 사용자가 요청을 보냈을 때 첫 응답을 받기까지(TTFT) 최대 56 초가 소요되고, 이는 고객 이탈로 직결됩니다. 따라서 콜드 스타트 지연 시간을 줄이는 것이 첫 번째로 해결할 핵심 과제입니다.

2. 초기 응답 시간–정확도 트레이드 오프 최적점 탐색

콜드 스타트의 병목인 모델 fetch/로딩 시간을 줄이기 위해 모델 경량화를 적용하면 응답은 빨라지지만 정확도 손실이 발생합니다. 실사용 품질을 유지하면서도 초기 응답을 빠르게 제공할 수 있도록 초기 응답 시간과 정확도 간의 최적 지점을 실험적으로 탐색하는 것이 두 번째 핵심 과제입니다.

3. 서비스 중단 없는 모델 품질 향상

초기 경량 모델 서빙 후 성능을 개선하려면 서비스를 중단하고 재배포해야 하는 문제가 있습니다. 사용자 세션을 유지하면서 동적으로 모델 품질을 향상시킬 수 있는 메커니즘이 필요합니다.

2.2. 기존 연구와의 비교

2.2.1. Serverless LLM 콜드 스타트 최적화 접근법 비교

기존 서버리스 LLM 서빙 연구들은 주로 모델 자체의 크기를 줄이지 않고 시스템 차원의 최적화에 집중하고 있습니다.

ServerlessLLM (2024) [2] 은 다계층 체크포인트 로딩 시스템을 통해 콜드 스타트 지연을 6-8 배 단축하는 성과를 보였으나, 근본적으로 전체 모델을 로딩해야 하는 한계가 있습니다. 이 방법은 GPU 메모리, DRAM, SSD 를 활용한 다층 아키텍처와 배치 읽기 기법을 사용하지만, 수백 GB 에 달하는 LLM 체크포인트 크기로 인해 여전히 상당한 초기화 시간이 필요합니다. ~~*ServerlessLLM* (2024)[2] 은 다계층 체크포인트 로딩 시스템을 통해 콜드 스타트 지연을 6-8 배 단축하는 성과를 보였으나, 근본적으로 전체 모델을 로딩해야 하는 한계가 있습니다. 이 방법은 GPU 메모리, DRAM, SSD 를 활용한 다층 아키텍처와 배치 읽기 기법을 사용하지만, 수백 GB 에 달하는 LLM 체크포인트 크기로 인해 여전히 상당한 초기화 시간이 필요합니다.~~

ParaServe (2025)[1] 는 파이프라인 병렬성을 활용해 여러 GPU 서버에 모델을 분산 배치하는 방식으로 콜드 스타트 지연을 최대 4.7 배 단축했습니다. 하지만 이 접근법 역시 전체 모델 크기는 그대로 유지하면서 네트워크 대역폭 집계를 통한 병렬 로딩에만 의존하고 있어, 확실한 콜드 스타트 완화 효과에 한계를 보입니다.

2.2.2. 모델 경량화 및 복구 방법론의 차별성

기존 프루닝 연구들은 모델 경량화에서 그치고 복구 방법론이 부재한 상황입니다.

DLP (Dynamic Layerwise Pruning, 2025)[4]는 모델 가중치와 입력 활성화 정보를 통합하여 각 층의 상대적 중요도를 적응적으로 결정하고, 70% 희소성에서 LLaMA2-7B 의 perplexity 를 7.79 감소시키는 성과를 달성했습니다. 그러나 이 방법은 레이어 내부 파라미터 삭제를 통한 모델 경량화에만 집중하고 있습니다.

The Unreasonable Ineffectiveness of the Deeper Layers (2024)[3]는 각도 거리 기반 계산법을 통해 레이어 삭제 후 QLoRA 를 적용하여 정확도 향상을 도모했지만, 역시 일회성 경량화 방법에 한정되어 있습니다.

2.2.3. ProgressiveServe 의 혁신적 접근법

본 연구의 ProgressiveServe 는 기존 연구들과 구별되는 두 가지 핵심 혁신을 제시합니다.

첫째, 프루닝 기법을 통한 근본적 모델 크기 축소로 LLM 모델 자체의 크기를 25% 감소시켜 콜드 스타트 시간을 대폭 단축합니다. 이는 각도 기반 거리 수식을 활용하여 정확도 손실을 최소화하면서 모델을 25% 축소하는 방식입니다.

둘째, 최초로 프루닝 후 모델을 점진적으로 복구하는 방법론을 도입했습니다. LoRA (Low-Rank Adaptation)를 활용한 미세 조정과 Progressive Merge 를 통해 프루닝으로 손실된 perplexity 와 accuracy 를 단계적으로 복구하며, 최종적으로 기존 LLM 모델과 동등한 성능으로 수렴하는 독창적인 접근법을 제시합니다.

2.3. 제안 내용

2.3.1. 경량화한 모델을 선 서빙하여 Serverless 환경에서도 빠른 첫 응답 제공

Llama2-7B 모델의 심층 레이어구간을 *The Unreasonable Ineffectiveness of the Deeper Layers* (2024)[3]의 표현 각도 기반 유사도(angular distance)를 바탕으로 프루닝을 진행하여 모델 사이즈를 줄입니다. 이를 통해 서버리스 환경에서 프루닝된 모델을 먼저 다운받아 추론을 시작함으로써 첫 토큰 지연 시간(TTFT)을 단축시켜 콜드스타트 지연을 완화합니다.

2.3.2. LoRA 어댑터 부착하여 정확도 회복

프루닝으로 인한 성능 저하를 보정하기 위해 PEFT(Parameter Efficient Fine-Tuning) 기법 중 LoRA(Low-Rank Adaptation)를 적용해 어댑터를 학습한 후 모델에 부착합니다. 이를 통해 프루닝된 모델 단계에서도 정확도, perplexity 를 실사용 수준으로 회복시키고, 이후 병합되는 레이어에도 어댑터를 부착해 사용자가 높은 정확도의 응답을 받을 수 있도록 합니다.

2.3.3. 점진적 로딩

프루닝된 모델의 초기 응답 후, 백그라운드에서 남은 레이어들을 단계적으로 로딩하여 모델의 성능을 점진적으로 향상시킵니다. 1 단계에서는 프루닝된 핵심 레이어와 A 어댑터로 빠른 첫 응답(TTFT 21.1% 단축)을 제공하고, 2-3 단계에서 백그라운드로 추가 레이어를 병합하여 최종적으로 원본 모델 수준의 성능에 도달합니다.

~~프루닝된 모델의 초기 응답 후, 백그라운드에서 남은 레이어들을 단계적으로 로딩하여 모델의 성능을 점진적으로 향상시킵니다.~~

~~Multi-Stage Pipeline 구조를 통해 사용자 응답과 모델 복구를 병렬로 처리합니다. 초기 프루닝된 모델이 첫 번째 응답을 제공하는 동안, 백그라운드에서는 다음 단계의 레이어와 어댑터들이 로딩되어 준비됩니다.~~

~~1 단계에서는 프루닝된 핵심 레이어(A) + A 어댑터만으로 빠른 첫 응답을 제공하고, 2 단계에서는 추가 레이어(B) + B 어댑터가 병합되어 더 정확한 응답을 생성하며, 3 단계에서는 마지막 추가 레이어(C)까지 복구되어 원본 모델 수준의 성능에 도달합니다.~~

2.4. 기대 효과 및 의의

2.4.1. 연구적 기여 및 의의

- 기존 Serverless LLM 연구는 모델 자체의 크기를 줄이기 보다 체크포인트 로딩이나 병렬화 등의 방법을 통해 cold start 지연 시간을 완화하였습니다. 하지만, 본 연구는 콜드 스타트 지연에서 가장 오랜 시간이 걸리는 모델 fetch 시간을 Pruning 기법을 통해 축소하여 지연을 낮추는 새로운 접근을 제시합니다.
- 기존의 Pruning 연구와 달리, 단순히 LLM 모델에 Pruning 기법을 적용하는 것을 넘어 점진적으로 제거된 레이어를 복구(Progressive Merge)하는 새로운 방법론을 설계 및 검증합니다.
- 레이어 프루닝 + LoRA 힐링 + 병렬 부팅 + 단계적 병합을 하나의 파이프라인으로 통합해, 서버리스 실환경에서 재현 가능한 레퍼런스를 제공합니다.
- Llama2-7B 모델에서 초기 응답 시간을 선행 연구 대비 21.1% 단축(114 초→90 초)하면서도 최종 단계에서는 원본 모델과 동일한 QA 성능(EM 55.67%, F1 66.11%)을 달성하는 것을 실험적으로 검증했습니다.

2.4.2. 기술적 기여 및 의의

- 본 연구는 레이어 프루닝, 병렬 부팅, LoRA 기반 모델 복원, 그리고 점진적 레이어 병합을 결합한 독창적인 파이프라인을 제안하여 서버리스 환경에서의 LLM 최적화를 위한 새로운 접근법으로서 높은 학술적 가치를 지닙니다.
- 서버리스 LLM 운영의 고질적인 문제인 '초기 응답 속도'와 '모델 정확도' 간의 상충 관계(Trade-off)를 효과적으로 해결합니다. 경량 모델로 첫 응답(TTFT)를 받는 시간을 효과적으로 줄이고, 이후 점진적으로 모델을 복구하여 최종적으로는 원본 모델로 정확도 손실이 없는 서비스를 제공하는 혁신적인 구조를 구현합니다.
- 이론에만 머무르지 않고 실제 서버리스 환경에 적용 가능한 엔드투엔드(end-to-end) 최적화 프레임워크를 구체적으로 제시함으로써, 후속 연구와 실제 산업 적용에 직접적인 가이드를 제공합니다.

2.4.3. 경제적/산업적 기대효과

- 고가의 자체 서버나 전문 관리 인력 없이 중소기업이 LLM 기반 AI 챗봇 서비스를 도입할 수 있게 하여, AI 기술 도입의 진입 장벽을 낮춥니다. 이는 대기업과의 기술 격차를 줄이고 중소기업의 경쟁력을 강화하는 효과를 가져옵니다.
- 요청 시에만 자원을 사용하는 서버리스 컴퓨팅의 장점을 LLM 서비스에도 적용할 수 있게 만들어, 기업의 인프라 구축 및 유지보수 비용을 크게 절감시킵니다. 이는 한정된 자원으로 최대의 효율을 추구해야 하는 중소기업에 최적화된 솔루션입니다.
- 고객 응대, 정보 제공 등 다양한 분야에서 중소기업이 AI 를 활용한 새로운 비즈니스 모델을 창출하고 서비스 혁신을 가속화할 수 있는 기술적 토대를 마련합니다.

2.5. 주요 기능 리스트

2.5.1. LLM 모델 Pruning

The Unreasonable Ineffectiveness of the Deeper Layers (2024)[3]에 따르면 성공적으로 Pruning 을 하기 위해서는 제거될 블록의 입력과 출력이 매우 유사해야합니다. 표현 각도 기반 유사도(angular distance)를 바탕으로 유사한 레이어 블록을 찾아 Transformer 레이어의 25%를 제거합니다. 이를 통해 Serverless 에서 LLM 모델을 서빙하는 과정에서 모델 fetch 시간을 약 ~~25%~~21.1% 단축시킵니다.

2.5.2. LoRA 어댑터 부착

PEFT(Parameter Efficient Fine-Tuning) 기법 중 LoRA(Low-Rank Adaptation)를 적용해 어댑터를 학습하여 성능을 보정합니다. 프루닝된 모델 A 그룹, 이후 병합할 레이어 B, C 그룹 총 3 개 그룹으로 레이어를~~을~~ 나눈 후 모델 복구 전략을 고려하여 32 개 레이어 구조를 유지하기 위해 제거된 레이어를 ~~identity 레이어~~ PassLayer 로 치환하여 LoRA 어댑터를 만듭니다. 1 단계로 A 레이어 그룹~~를~~을 로드한 후 B, C 그룹은~~는 identity 레이어~~ PassLayer 로 치환해 A 어댑터를 학습합니다. 다음으로 A ~~레이어~~, B 레이어 그룹을~~를~~ 로드한 후, C 그룹은~~는 identity 레이어~~ PassLayer 로 치환하여 ~~B 레이어를 학습합니다.~~ AB 어댑터를 학습합니다. AB 어댑터는 레이어 그룹 A 와 B 전체에 적용됩니다. 이렇게 학습된 LoRA 어댑터를 모델에 부착하여 정확도, perplexity 를 실사용 수준으로 회복시킬 수 있습니다.

2.5.3. 점진적 로딩

프루닝된 모델의 초기 응답 후, 백그라운드에서 남은 레이어들을 단계적으로 로딩하여 모델의 성능을 점진적으로 향상시킵니다.

- 단계 1: 프루닝된 레이어 그룹 A(1-20 번, 29-32 번)와 A 어댑터를 먼저 로드하여 빠른 첫 응답을 제공합니다. 제거된 레이어 위치에는 PassLayer 를 설치하여 모델 구조를 유지합니다.
- 단계 2: 백그라운드에서 레이어 그룹 B(21-24 번)를 로드하고 PassLayer 를 실제 레이어로 교체합니다. 동시에 AB 어댑터를 활성화하여 정확도를 향상시키며, 이 과정은 사용자 응답과 병렬로 진행되어 서비스 중단이 없습니다.
- 단계 3: 마지막 레이어 그룹 C(25-28 번)를 로드하여 원본 모델의 전체 아키텍처를 복구하고, 모든 어댑터를 비활성화하여 원본 모델 수준의 성능에 도달합니다.

- ~~S1 기동: 프루닝된 본체를 로드하고 제거된 레이어를 PassLayer 로 설치하여 모델 구조를 보존합니다. S1 용 LoRA 어댑터를 생성·부착하고, 체크포인트 키를 현재 어댑터 네임스페이스에 맞춰 정정한 뒤 로드하여 초기 품질을 확보합니다. 이후 S1 만으로 사용자에게 응답을 받습니다.~~
- ~~S2 전환: prune_log 에서 B 인덱스를 읽어 B 레이어만 CPU 에 적재합니다. 동일한~~

~~config·dtype·device 로 교체용 LlamaDecoderLayer 를 미리 구성하고 B 델타 state_dict 를 검증하여 로드합니다. S2 단계 품질 강화를 위해 AB 어댑터를 추가 등록·활성화하고, 대응 체크포인트를 로드하여 품질 변동을 최소화합니다. 이후 디코딩은 계속 유지한 채, 토큰 또는 미니배치 경계에서 PassLayer 를 준비된 실제 레이어로 교체합니다. 어댑터 로드·전송 비용은 대부분 디코딩(응답) 과 겹쳐져 노출 지연 없이 품질을 유지합니다.~~

- ~~S3 전환: C 레이어도 B 레이어와 동일하게 스트리밍 수신하여 CPU 에 적재합니다. 적재된 모델과 동일한 config·dtype·device 로 LlamaDecoderLayer 를 생성하고 토큰 또는 미니배치 경계에서 PassLayer 를 실제 레이어로 교체합니다, 기존 어댑터를 모두 off 하여 원본 모델로 복구합니다.~~

3. 과제 설계

3.1. 요구사항 정의

3.1.1. 신속성

- TTFT(cold) : 스토리지·네트워크·RPS 조건을 고정한 상태에서 ServerlessLLM 대비 P95 TTFT 를 ~~1.7–3.1 배 이상~~ 20% 이상 단축합니다.
  - TTFT(Time To First Token)는 요청이 도착한 시점부터 첫 토큰이 전달될 때까지의 지연으로, 큐잉·프리필(프롬프트 처리)·네트워크 지연을 포함합니다
  - 검증: ServerlessLLM 재현 벤치와 ProgressiveServe 를 동일 조건으로 A/B 측정합니다.
  - 모든 측정은 cold 경로에서만 수행합니다.
  - 모델: Llama-2 7B 계열로 고정, GPU: RTX 5090 1 개를 사용합니다.
  - 네트워크: 원격 스토리지 연결은 25–100 Gbps 환경 중 실제 환경에 맞춰 고정하고, 대역폭(b) 값을 벤치 시작 전 마이크로벤치로 산출·문서화합니다.
- 신속성은 2.5.1 에서의 모델 경량화 기법을 적용하여 만든 모델을 먼저 서빙하는 2.5.3 의 점진적 로딩 기법으로 만족시킬 수 있습니다.

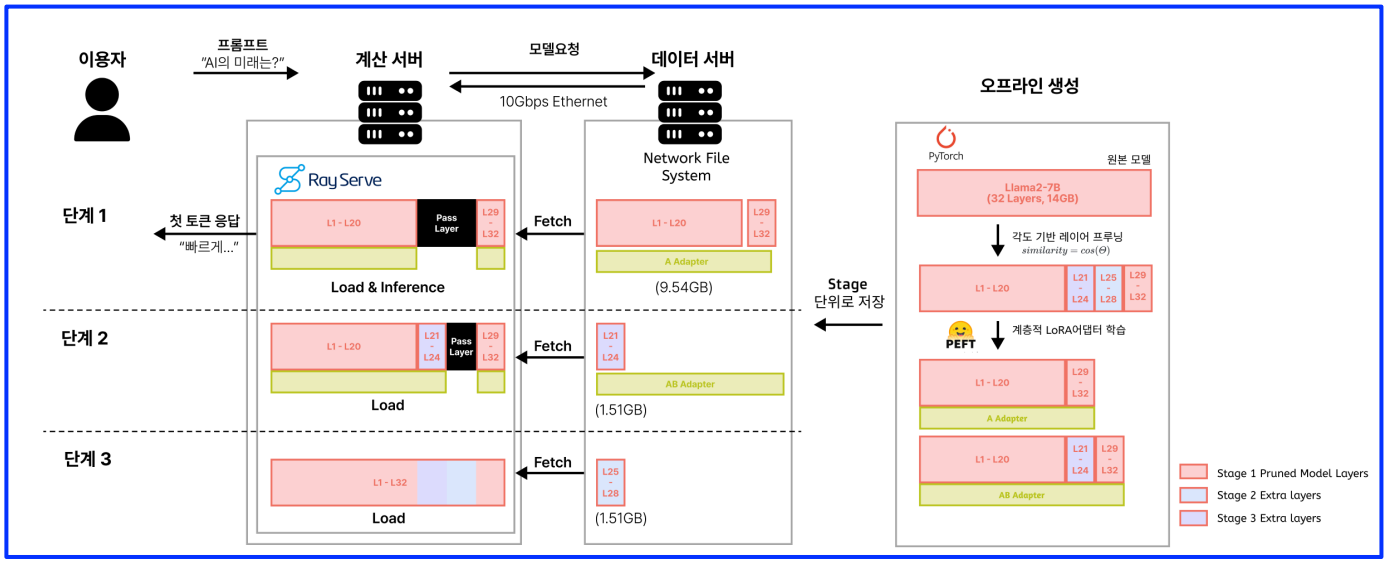
3.1.2. 정확성

- 첫번째,두번째 응답 모델(S1,S2)의 도메인 QA 세트의 EM 및 F1 에서 원본 대비 감소폭 $\Delta \le 3pp$ 를 만족해야 하며 복구 완료 모델(S3)는 동일 지표에서 $\Delta \le 1pp$ 를 달성해야 합니다.
  - EM 은 정답 문자열 완전 일치 비율, F1 은 토큰 단위 정밀도·재현율의 조화평균으로 SQuAD 공식 스크립트 스타일을 따릅니다(노멀라이즈, 불용어·구두점 처리 동일 적용)
  - 동일 프롬프트·디코딩 설정을 고정하며 결과는 평균과 95% CI 로 보고합니다.
- 2.5.2 에서 LoRA 어댑터를 부착하는 과정을 통해 경량화된 모델이 가질 수 있는 정확성 문제를 해결합니다.

3.1.3. 자원 효율성

- Cold 구간 GPU 메모리–시간(cost) 총량을 ServerlessLLM 대비 $\le 0.9\times$로 절감합니다.
  - GPU-seconds = Σ(활성 GPU 수 × 활성 시간), 포함 범위는 컨테이너 생성·런타임 초기화·체크포인트 페치/로딩·컴파일을 모두 포함합니다.
  - ServerlessLLM 과 동일 조건으로 비교합니다.
  - 측정은 첫 요청 도착부터 첫 토큰 직전까지(프리필 이전)와 첫 토큰 이후 디코딩 구간을 분리 집계하되, 비용 비교는 Cold 경로 전체(컨테이너~프리필 직전)만을 포함해 산출합니다.
- 2.5.3 에서의 점진적 로딩 기법 중 CPU, GPU 의 병렬적 사용을 통해 ServerlessLLM 의 직렬적 로딩에서 도달할 수 없는 자원 효율성 획득이 가능합니다.

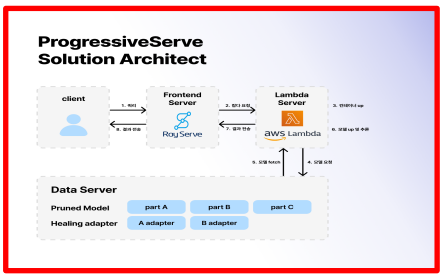
3.2. 전체시스템 구성



이 시스템은 서버리스 환경에서 LLM 콜드 스사트를 줄이기 위해 설계된 ProgressiveServe 아키텍처로, 모델을 단계적으로 로딩하고 복구하는 방식을 사용합니다.

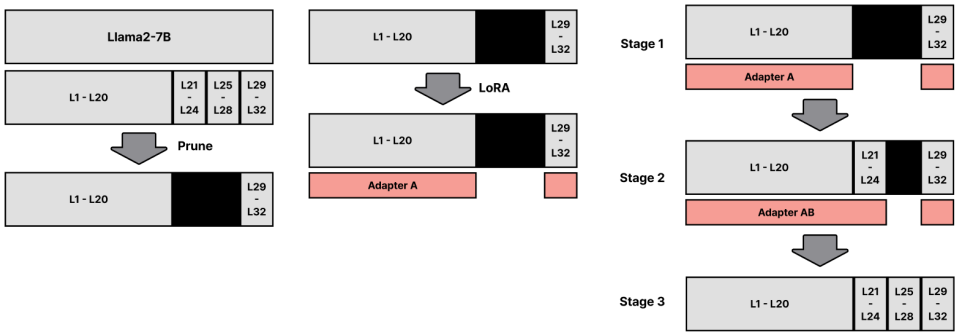
오프라인 단계에서는 Pytorch 를 이용하여 프루닝과 어댑터 생성을 진행합니다. Pytorch 를 활용한 자체 함수를 통해 원본 모델 레이어들의 코사인 유사도를 계산하여 우선적으로 서빙할 레이어들로 구성된 프루닝 모델을 만들고 나머지 레이어들은 Stage2,3 에 서빙할 수 있게 반으로 나누어 저장합니다. 단계별 성능 복구를 위해 PEFT 라이브러리 를 사용하여 LoRA 어댑터(A,AB)를 학습합니다. 해당 어댑터는 Stage 1 과 Stage 2 에서의 일부 레이어로만 구성된 모델이 제대로 응답된 값을 이용자에게 제공할 수 있도록 합니다. 이렇게 생성된 프루닝된 모델, 나머지 레이어 그룹, 어댑터 A, AB 를 데이터 서버(NFS)에 저장하여 온라인 단계에서 필요 시 Fetch 할 수 있도록 합니다.

온라인 단계에서는 RayServe 를 기반으로 이용자의 요청을 처리합니다. 사용자가 계산 서버에 질문을 하면 RayServe 는 데이터 서버에서 Stage 1 의 경량화된 모델과 어댑터(A)를 계산 서버로 Fetch 합니다. 경량화된 모델로 inference 를 진행하면서 동시에 백그라운드에서 Stage 2 레이어들을 Fetch 및 Load 합니다. 로드가 완료되면 프루닝 시 삽입되었던 PassLayer 를 실제 레이어로 빠르게 교체하고 AB 어댑터를 적용합니다. 이후 Stage 3 레이어도 동일한 방식으로 순차적으로 로딩되며, 모든 레이어가 복구되면 어댑터를 비활성화하고 원본 모델과 동일한 성능의 최종 형태로 전환됩니다. 이를 통해 ProgressiveServe 는 서비스 중단 없이 빠른 초기 응답과 최종 고성능 모델 품질을 동시에 제공합니다.



3.3. 주요 엔진 및 기능 설계

3.3.1. 전체 시스템 아키텍쳐

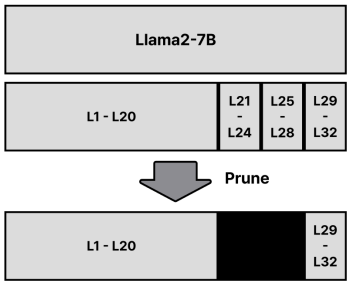


ProgressiveServe 는 서버리스 환경에서 LLM 의 콜드 스타트 지연을 완화하기 위해 점진적 모델 로딩 및 복구 메커니즘을 구현한 시스템입니다. 시스템은 오프라인 단계와 온라인 단계로 구성되며, 오프라인에서는 모델 프루닝 및 LoRA 어댑터 학습을, 온라인에서는 3 단계 점진적 복구 파이프라인을 수행합니다.

전체 SW 구조는 크게 (1) Model Pruning, (2) LoRA Training, (3) Progressive Loading Pipeline 의 3 개 주요 모듈로 구성됩니다. Model Pruning 과정은 각도 기반 연속 레이어 프루닝 알고리즘을 적용하여 Llama2-7B 의 32 개 레이어 중 21-28 번 레이어를 제거하고, 3 개 레이어 그룹(A 그룹: 1-20, 29-32 번 레이어, B 그룹: 21-24 번 레이어, C 그룹: 25-28 번 레이어)으로 분할합니다. LoRA Training 에서는 각 복구 단계별로 SQuAD 데이터셋을 사용하여 A 어댑터와 AB 어댑터를 학습시킵니다.

Progressive Loading Pipeline 은 RayServe 프레임워크 기반으로 구현되며, Actor 생성 시점부터 단계별로 모델 파라미터를 Fetch 및 Load 하는 역할을 담당합니다. Layer Recovery 과정은 PassLayer 메커니즘을 활용하여 프루닝된 레이어 위치에 플레이스홀더를 생성하고, 백그라운드에서 순차적으로 실제 레이어로 교체하는 동적 복구 로직을 관리합니다.

3.3.2. 각도 기반 연속 레이어 프루닝



Model Pruning 과정의 핵심은 레이어별 중요도를 정량적으로 평가하는 각도 기반 연속 레이어 프루닝 방법입니다. 이 방법은 각 레이어의 입력과 출력 간의 코사인 유사도를 바탕으로, 각도 기반 거리를 계산하여, 거리값이 작은 레이어들을 프루닝 대상으로 식별합니다. 구체적으로 아래의 각도 기반 거리 계산식을 적용합니다.

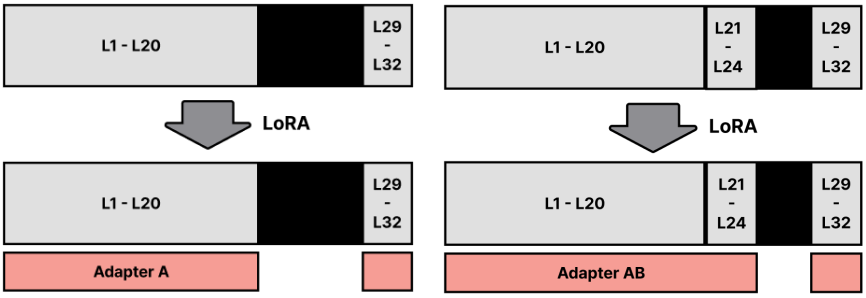
$$d(x(\ell), x(\ell+n)) \equiv \frac{1}{\pi} \arccos\left(\frac{x(\ell)_T \cdot x(\ell+n)_T}{\|x(\ell)_T\| \|x(\ell+n)_T\|}\right)$$

코사인 유사도가 높다는 것은 해당 레이어의 변환이 입력을 크게 변화시키지 않는다는 의미로, 해당 레이어를 제거하더라도 QA 성능 저하가 미미하다는 특성을 활용합니다.

구체적인 코드 구현을 살펴보자면, compute_layer_inputs 함수를 통해 Llama-2-7B 의 각 Transformer 층에서 층별 표현 x_ℓ 을 추출하고, 블록 크기 n 을 설정합니다. 그 다음 choose_block_to_drop 함수를 통해 각도 거리를 계산하여 $d(\ell)$이 최소인 시작 인덱스에서 연속 n 개의 레이어를 프루닝 대상으로 결정합니다. 본 연구에서는 총 8 개의 연속 레이어(21-28 번 레이어)를 제거하여 모델 크기를 약 25% 축소시켰으며, 이는 초기 로딩 시간 단축에 직접적으로 기여합니다.

프루닝된 레이어들은 복구 순서를 고려하여 레이어 그룹 B(21-24 번)와 레이어 그룹 C(25-28 번)로 분할 저장되며, Safetensor 포맷으로 NFS 원격 저장소에 별도 체크포인트로 관리됩니다.

3.3.3. 계층적 LoRA 어댑터 학습



LoRA Training 은 프루닝으로 인한 성능 저하를 완화하기 위해 각 복구 단계별로 특화된 어댑터를 학습합니다. 본 시스템은 PEFT 라이브러리를 이용해 A 어댑터와 AB 어댑터를 구현하였습니다.

A 어댑터는 단계 1(레이어 그룹 A 만 로드된 상태)의 성능 복구를 목표로 합니다. 학습 시, 원본 Llama2-7B 모델에서 레이어 그룹 B 와 C 를 PassLayer 로 치환한 상태에서 동작하며, 레이어 그룹 A 의 Attention 투영(q_proj, k_proj, v_proj, o_proj)과 MLP 투영(down_proj, up_proj, gate_proj)에만 LoRA 를 부착하여 학습합니다. LoRA 는 각 가중치에 대해 저차원 보정 W=W0+BA 로 표현되며, 스크립트의 기본값은 r=8, α=16, dropout=0.05 입니다. 학습 데이터는 SQuAD 데이터셋을 사용하였고, Trainer 기반 학습 설정은 epochs=3, batch=4, gradient accumulation steps=16, learning rate=2e-4,fp16=True 로 실행하였습니다. 학습 완료 후 어댑터는 model.save_pretrained(our_dir)와 함께 PEFT 상태를 저장하고, 추가적으로 export_adapter_pt_and_recipe 을 통해 학습된 레이어의 인덱스만 필터링한 LoRA state(.pt)와 복원 레시피(JSON)를 생성하여 adapter 폴더에 보관합니다.

AB 어댑터는 단계 2(레이어 그룹 A, B 로드된 상태)의 최적화를 목표로 합니다. 학습 시에는 레이어 그룹 C 만 PassLayer 로 치환한 상태에서 레이어 그룹 A 와 B 전체에 LoRA 어댑터를 부착하여 학습합니다.

동일한 SQuAD 데이터셋과 학습 설정을 사용하였고, 학습된 어댑터는 Hugging Face Hub 에 업로드되어 온라인 단계에서 동적으로 로드됩니다.

3.4. 주요 기능의 구현

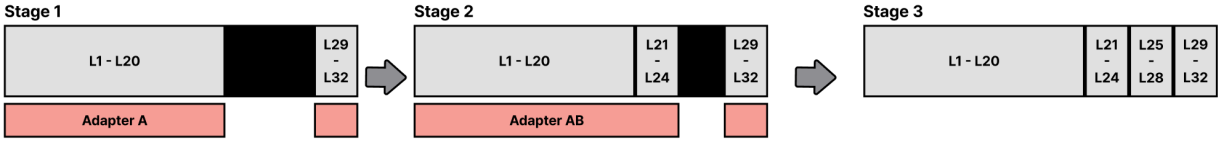
3.4.1. PassLayer 기반 동적 모델 복구 메커니즘

Layer Recovery 과정의 핵심 구현은 PassLayer 메커니즘입니다. PassLayer 는 프루닝된 레이어 위치에 삽입되는 항등 함수(identity function) 역할의 더미 레이어로, 입력을 그대로 출력으로 전달합니다. 이는 PyTorch 의 nn.Module 을 상속받아 구현되며, forward 메서드는 단순히 return x 형태로 작성됩니다.

구체적인 구현 알고리즘은 다음과 같습니다. 단계 1 에서는 원본 모델의 Config 파일을 먼저 Fetch 하여 전체 32 개 레이어의 아키텍처 정보를 불러옵니다. 이후 모델 객체를 생성할 때 21–28 번 레이어 위치에는 실제 Transformer 레이어 대신 PassLayer 객체를 할당합니다. 레이어 그룹 A(1–20 번, 29–32 번)의 Safetensor 샤드는 NFS 마운트 경로에서 순차적으로 읽어 해당 레이어의 가중치를 초기화합니다.

단계 2 와 3 에서는 백그라운드 스레드가 추가 레이어 그룹을 Fetch 및 Load 한 후, 기존 PassLayer 객체를 실제 Transformer 레이어 객체로 동적 교체합니다. 이는 Python 의 setattr() 함수를 활용하여 모델 객체의 layers 리스트 내 특정 인덱스 위치의 참조를 변경하는 방식으로 구현되었습니다.

3.4.2. 3 단계 점진적 로딩 파이프라인



Progressive Loading Pipeline 은 RayServe Actor 생성 시점부터 3 단계로 구분된 파이프라인을 실행합니다. 각 단계는 (1) Fetch (원격 저장소에서 파일 읽기), (2) Load (메모리 적재 및 GPU 전송), (3) Adapter Application (LoRA 어댑터 부착)의 세부 프로세스로 구성됩니다.

단계 1 구현은 다음 순서로 진행됩니다. (a) NFS 마운트 경로에서 config.json 과 tokenizer 관련 파일들을 Fetch 합니다. (b) Hugging Face Transformers 의 AutoConfig.from_pretrained() 메서드로 모델 구조 정보를 로드하고, AutoModelForCausalLM.from_config() 메서드로 32 개 레이어의 플레이스홀더를 생성합니다. (c) 레이어 그룹 A 에 해당하는 Safetensor 샤드 파일들을 Fetch 하여 torch.load() 또는 safetensors.torch.load_file() 함수로 가중치 텐서를 메모리에 적재합니다. (d) 적재된 텐서를 해당 레이어의 파라미터에 할당하고 .to('cuda') 메서드로 GPU 로 전송합니다. (e) A 어댑터를 Hugging Face Hub 에서 Fetch 하여 PEFT 의 PeftModel.from_pretrained() 메서드로 레이어 그룹 A 에 부착하고 활성화합니다. (f) 첫 번째 토큰을 생성하여 사용자에게 응답을 시작합니다.

단계 2 와 3 구현은 백그라운드 스레드에서 비동기로 실행됩니다. 단계 2 에서는 (a) 레이어 그룹 B 의

Safetensor 파일과 AB 어댑터를 Fetch 합니다. (b) 21–24 번 레이어의 PassLayer 를 실제 Transformer 레이어로 교체하고 가중치를 Load 합니다. (c) 기존 A 어댑터를 disable_adapter() 메서드로 비활성화하고, AB 어댑터를 load_adapter() 및 enable_adapter() 메서드로 적용합니다. 단계 3 에서는 (a) 레이어 그룹 C 의 Safetensor 파일을 Fetch 합니다. (b) 25–28 번 레이어를 실제 레이어로 복구합니다. (c) AB 어댑터를 비활성화하여 원본 모델과 동일한 아키텍처를 완성합니다.

3.4.3. 콜드 스타트 시간 측정 모듈

실험의 정확한 성능 측정을 위해 콜드 스타트 시간 측정 모듈을 구현하였습니다. 이 모듈은 Python 의 time.time()함수를 활용하여 벽시계 시간(Wall-Clock Time)을 기록합니다. TTFT(Time-To-First-Token)는 RayServe Actor 생성 시점부터 첫 번째 토큰이 생성되는 시점까지의 시간으로 정의하며, 각 단계별 복구 시간은 PassLayer 교체가 완료되는 시점까지로 측정합니다.

측정의 일관성을 위해 각 트라이얼마다 OS 페이지 캐시(sync; echo 3 > /proc/sys/vm/drop_caches)와 GPU 캐시(torch.cuda.empty_cache()를 초기화합니다. 또한 tokenizer 로딩 시 Hugging Face Hub 다운로드를 차단하고 NFS 마운트 경로에서만 참조하도록 local_files_only=True 옵션을 적용하여 네트워크 변수를 통제하였습니다.

3.5. 기타

3.5.1. 실험 환경 구성

모든 실험은 Ubuntu 24.04 운영체제, NVIDIA 드라이버 580.65.06, CUDA 13.0 환경에서 NVIDIA RTX 5090 GPU (24GB VRAM)를 사용하여 수행되었습니다. 모델 가중치 파일은 별도의 원격 데이터 서버에 저장되어 NFSv4 프로토콜로 export 되었으며, 실험 노드에 마운트하여 FP16 정밀도의 Safetensor 파일을 읽습니다. 원격 데이터 서버와 로컬 서버 간은 16Gbps 이더넷으로 연결되어 있으며, NFS 마운트 옵션은 모든 트라이얼에서 고정하여 네트워크 I/O 조건이 일관되게 유지되도록 하였습니다.

비교 대상인 ServerlessLLM 은 sllm-store 라는 별도 스토리지 계층 프로세스를 사용하지만, 본 연구에서는 공정한 비교를 위해 sllm-store 없이 동일한 NFS 경로의 파일을 직접 읽도록 설정하였습니다. 이는 시스템 레벨 최적화가 아닌 모델 레벨 최적화의 효과를 순수하게 측정하기 위함입니다.

3.5.2. 평가 데이터셋 및 메트릭

모델 성능 평가를 위해 TriviaQA Validation 데이터셋을 사용하였습니다. TriviaQA 는 대규모 질의응답 데이터셋으로, 다양한 도메인의 질문과 정답 쌍을 포함하고 있어 LLM 의 지식 이해도를 평가하는 데 적합합니다. 평가는 제로샷(zero-shot) 방식으로 진행되었으며, EM(Exact Match)과 F1 Score 를 메트릭으로 사용하였습니다.

EM 은 모델이 생성한 답변이 정답과 정확히 일치하는 비율을 측정하며, F1 Score 는 토큰 레벨에서 precision 과 recall 의 조화평균으로 부분 일치도 고려합니다. 모든 평가에서 동일한 프롬프트 템플릿과 생성 파라미터(max_new_tokens=10, greedy decoding)를 적용하여 결과의 재현성을 확보하였습니다. 원본 Llama2-7B 모델과 ProgressiveServe 의 단계 1, 2, 3 을 비교하여 점진적 복구 과정에서의 성능 변화를 추적하였습니다.

4.참고 문헌

[1] C. Lou et al., “Towards Swift Serverless LLM Cold Starts with ParaServe,” arXiv preprint arXiv:2502.15524, 2025.

[2] Y. Fu et al., “ServerlessLLM: Low-latency server-less inference for large language models,” in USENIX OSDI, 2024.

[3] A. Gromov et al., “The unreasonable ineffectiveness of the deeper layers,” arXiv preprint arXiv:2403.17887, 2024.

[4] Y. Chen et al., “DLP: Dynamic Layerwise Pruning in Large Language Models,” in ICML, 2025.

~~[1] Towards Swift Serverless LLM Cold Starts with ParaServe~~
~~https://arxiv.org/html/2502.15524v1~~

~~[2] ServerlessLLM: Low-Latency Serverless Inference for Large Language Models~~

~~https://arxiv.org/pdf/2401.14351~~

~~[3] The Unreasonable Ineffectiveness of the Deeper Layers(Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Glorioso, Daniel A. Roberts)~~

~~https://arxiv.org/abs/2403.17887~~

~~[4] DLP: Dynamic Layerwise Pruning in Large Language Models~~

~~https://arxiv.org/abs/2505.23807~~