저작자표시-비영리-변경금지 2.0 대한민국

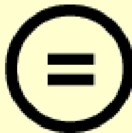
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

석사학위논문

다중 딥러닝 모델 실행 효율화를 위한 경량화 기법 및 블록 레벨 스케줄링 연구

HANSUNG UNIVERSITY

2026년

한 성 대 학 교 대 학 원

A I 응 용 학 과

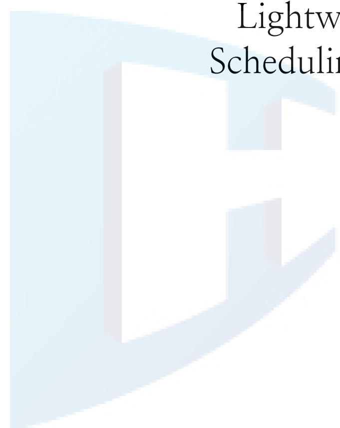
A I 응 용 학 전 공

김 혁 수

석사학위논문
지도교수 김명선

다중 딥러닝 모델 실행 효율화를 위한 경량화 기법 및 블록 레벨 스케줄링 연구

Lightweighting Techniques and Block-Level Scheduling for Efficient Multi-DNN Execution



HANSUNG UNIVERSITY

2025년 12월 일

한 성 대 학 교 대 학 원

A I 응 용 학 과

A I 응 용 학 전 공

김 혁 수

석사학위논문
지도교수 김명선

다중 딥러닝 모델 실행 효율화를 위한 경량화 기법 및 블록 레벨 스케줄링 연구

Lightweighting Techniques and Block-Level Scheduling for Efficient Multi-DNN Execution

위 논문을 공학 석사학위 논문으로 제출함

2025년 12월 일

한 성 대 학 교 대 학 원

A I 응 용 학 과

A I 응 용 학 전 공

김 혁 수

HANSUNG UNIVERSITY

김혁수의 공학 석사학위 논문을 인준함

2025년 12월 일

HANSUNG UNIVERSITY

심사위원장 강미선 (인)

심 사 위 원 이웅희 (인)

심 사 위 원 김명선 (인)

국 문 초 록

다중 딥러닝 모델 실행 효율화를 위한 경량화 기법 및 블록 레벨 스케줄링 연구

한성대학교대학원
AI응용학과
AI응용학전공
김 혁 수

HANSUNG UNIVERSITY

임베디드 환경에서 다중 딥러닝 모델을 동시에 수행할 경우, 모델 간 자원 경쟁으로 인해 실행 지연이 발생하며 이는 지연 시간에 민감한 시스템에서 치명적인 성능 저하로 이어질 수 있다. 이러한 문제를 완화하기 위해서는 모델 경량화 기법을 적용하는 것이 필수적이며, 대표적으로 Quantization과 Pruning이 이에 해당한다. 그러나 이들 기법의 적용 사례는 대부분 분류 모델에 집중되어 있어, 검출 및 추적 모델에 동일한 경량화 기법을 적용한 연구는 상대적으로 드물다. 이를 확장하기 위해 본 논문은 효율적인 DNN 모델을 실행하기 위해 두 가지 측면에서 접근한다. 첫째, 서버 시스템 환경에서 다양한 시각 지능 모델을 대상으로 Pruning과 Quantization을 단독 또는 조합하여 모델 크기, 파라미터 수, 정확도 변화를 체계적으로 분석하였다. 실험 결과, 두 경량화 기법을 같이 적용했을 때 일부 모델에서 더 높은 파라미터 감소 효과를 보이면서도 정확도 손실을 최소화하였다. 또한, 모델 구조 변경 없이 추론 과정을 최적화하는 TensorRT 기반 런타임 최적화 기법을 적용하

여, 추가적인 연산 그래프 최적화와 커널 융합을 통해 기존 경량화 기법만으로는 확보하기 어려운 추론 속도 향상과 메모리 효율 개선 효과를 확인하였다.

둘째, 임베디드 환경에서 다중 DNN 작업을 동시에 수행할 때 실행 지연을 최소화하기 위해 블록 단위 동적 스케줄링 및 블록 수준 동적 전환 기법을 제안한다. 해당 기법은 모델을 기능적 단위인 블록으로 분할하여 실행 단위로 구성하고, 병렬 실행 시 오히려 지연을 유발하는 블록을 식별하여 순차 실행으로 전환한다. 또한 각 블록의 실행 지연 정도를 정량화하는 지표인 *LAG*를 활용해, 지연이 크게 예상되는 블록을 런타임에 경량화된 블록으로 대체하여 지연 시간과 정확도 간의 균형을 실시간으로 유지한다. 대표적인 임베디드 환경인 NVIDIA AGX Jetson Xavier 보드에서 이질적인 다중 DNN을 동시에 실행한 실험 결과, 제안 기법은 최대 29.3%의 지연 시간 감소와 기준 정확도의 90% 이상 유지할 수 있는 성능을 달성하였다.

【주요어】임베디드 딥러닝, LAG, EMA, 모델 압축, 블록 교체, 다중 DNN 스케줄링

HANSUNG UNIVERSITY

목 차

제 1 장 서 론 ······ 1

제 2 장 연구 배경 및 문제 정의 ······ 4

제 1 절 임베디드 시스템에서의 다중 DNN 실행의 특성 및 제약사항 ······ 4
제 2 절 문제 정의 ······ 5
1) 문제 정의 : 스케줄링 단위 ······ 5
2) 문제 정의 : 기존 경량화 기법의 한계점 ······ 7

제 3 장 모델 경량화 및 추론 최적화를 위한 복합적 설계 기법 제안 ······ 10

제 1 절 Pruning-Quantization 통합 경량화 구조 설계 제안 ······ 10
제 2 절 TensorRT를 활용한 추론 최적화 구조 설계 제안 ······ 12

제 4 장 블록 수준 스케줄링 기반 다중 DNN 실행 최적화 기법 제안 ······ 15

제 1 절 블록 단위 지연 시간 추정을 통해 각 DNN 모델의 실행 동작 분석 ······ 15
제 2 절 엣지 디바이스에서 다중 DNN 추론을 위한 LAG 기반 블록 단위 스케줄링 및 동적 압축 프레임워크 제안 ······ 21
1) 엣지 디바이스에서 다중 DNN 추론을 위한 LAG 기반 블록 단위 스케줄링 및 동적 압축 프레임워크 ······ 22
2) 점진적 모델 경량화 기법 ······ 25
3) 블록 스케줄러 기법 ······ 27
4) 블록 압축 레벨 결정 기법 ······ 29
5) 임계값 도출 및 상태 정의 ······ 33

제 5 장 실험 결과 및 분석 ······ 35

제 1 절 모델 경량화 및 추론 최적화 기법에 따른 실험 결과 ······ 35

1) 실험 환경 및 구성 ········ 35
2) 모델 경량화 기법 적용 결과 ········ 36
가) 객체 분류 모델 경량화 적용 결과 ········ 37
나) 객체 검출 모델 경량화 적용 결과 ········ 39
다) 객체 추적 모델 경량화 적용 결과 ········ 39
3) 추론 최적화 기법(TensorRT) 적용 결과 ········ 40
제 2 절 블록 레벨 스케줄링 기반 다중 DNN 실행 최적화 기법의 실험 결과 ········ 42
1) 실험 환경 및 구성 ········ 43
가) 시스템 구성 및 실험 조건 ········ 43
나) 구현 방식 ········ 45
2) 다중 DNN 실행 최적화 기법 실험 결과 ········ 45
가) 병렬 및 순차 블록 실행 결과 비교 ········ 45
나) 3개 모델 동시 실행 ········ 47
다) 2개 모델 동시 실행 ········ 49
라) 동일한 2개 DNN과 1개의 이기종 DNN 동시 실행 ········ 50
3) 상충관계 분석 ········ 52
가) β 스케일링에 따른 지연 시간 및 정확도 분석 ········ 52
나) α 스케일링에 따른 지연 시간 및 정확도 분석 ········ 53
다) 스케줄링 방식의 영향 분석 ········ 55

제 6 장 결 론 ········ 57

참 고 문 헌 ········ 59

ABSTRACT ········ 63

HANSUNG UNIVERSITY

표 목 차

[표 4-1] 표기법 ···· 24
[표 4-2] 점진적 모델 압축 과정에서 레이어 유형별 분해 기법 ···· 25
[표 4-3] 블록 압축 레벨 결정 기법의 State-Action Table ···· 30
[표 4-4] θ_{trend}와 $\theta_{accuracy}$에 대한 모델별 임계값 ···· 33
[표 5-1] 실험에 사용된 서버 시스템 환경 사양 ···· 35
[표 5-2] 객체 분류 모델 경량화 실험 결과 비교 (VGG16) ···· 37
[표 5-3] 객체 분류 모델 경량화 실험 결과 비교 (ResNet152) ···· 38
[표 5-4] 객체 검출 모델 경량화 실험 결과 비교 (Faster-RCNN) ···· 39
[표 5-5] 객체 추적 모델 경량화 실험 결과 비교 (SiamRPN++) ···· 40
[표 5-6] 분류 모델에 대한 TensorRT 적용 시 추론 속도 비교 결과 ···· 41
[표 5-7] 추적 모델에 대한 TensorRT 적용 시 추론 속도 비교 결과 ···· 41
[표 5-8] 실험에 쓰인 NVIDIA Jetson AGX Xavier Developer Kit 사양 43
[표 5-9] 선택된 블록 조합에 대한 병렬 실행과 순차 실행의 성능 비교 ·· 46
[표 5-10] 스케줄링 적용 여부에 따른 블록 수준 동적 전환 성능 비교 ··· 55

그 림 목 차

[그림 2-1] 모델 단위의 다중 DNN 병렬 실행 구조 ········· 6
[그림 2-2] 레이어 단위의 다중 DNN 병렬 실행 구조 ········· 7
[그림 3-1] Pruning과 Quantization을 순차적으로 적용한 구조 ········· 10
[그림 4-1] LAG, LAG의 SMA, 평균 LAG간의 비교 ········· 16
[그림 4-2] 평활 계수에 따른 LAG와 LAG의 EMA의 비교 ········· 18
[그림 4-3] 다중 DNN 추론을 위해 제안된 프레임워크의 동작 흐름도 ··· 21
[그림 4-4] 다중 DNN 추론을 위한 제안된 프레임워크의 알고리즘 ········· 23
[그림 4-5] 자동 랭크 선택 기반 점진적 DNN 모델 압축 구조 개요 ········· 26
[그림 4-6] Conflict Table을 활용한 블록 스케줄러의 동작 흐름도 ········· 27
[그림 4-7] 블록 스케줄러 알고리즘 ········· 28
[그림 4-8] 블록 압축 레벨 결정 기법의 동작 흐름도 ········· 29
[그림 4-9] LAG_{trend}, Accuracy 기반의 블록 압축 레벨 결정 알고리즘 ··· 31
[그림 5-1] 제안된 프레임워크의 DNN 모델 실행 파이프라인 ········· 44
[그림 5-2] 서로 다른 스케줄링 방식 적용 시 세 가지 DNN 모델의 지연시간(a) 및 정확도(b) 비교 ········· 47
[그림 5-3] 각 모델의 지연 시간 분포에 따른 평균 정확도 ········· 48
[그림 5-4] 두 가지 DNN 동시 실행 환경에서 지연 시간(a) 및 정확도(b) 비교 ········· 49
[그림 5-5] 동일한 두 DNN과 하나의 이질적 DNN 동시 실행 환경에서의 지연 시간 성능 비교 ········· 50
[그림 5-6] 동일한 두 DNN과 하나의 이질적 DNN 동시 실행 환경에서의 정확도 비교 ········· 51
[그림 5-7] 각 모델에 대해 EMA_{long}의 파라미터 β 값 변화에 따른 지연 시간 및 정확도 비교 ········· 52
[그림 5-8] 각 모델에 대해 EMA_{short}의 파라미터 α 값 변화에 따른 지연 시간 및 정확도 비교 ········· 54

HANSUNG UNIVERSITY

제 1 장 서론

현대의 실시간 지능형 시스템은 점점 더 복잡해지고 있으며, 동시에 여러 기능을 수행해야 하는 요구가 증가하고 있다. 특히 자율주행차량(Bojarski, M., et al, 2017; Dosovitskiy, A., et al 2017; Grigorescu, S., et al, 2020), 스마트 시티(Zanella, A., et al, 2014), 산업용 로봇(Liu, Z., et al, 2022)과 같은 응용 분야에서는 객체 인식, 검출, 추적과 같은 다양한 작업을 동시에 수행하기 위해 다중 DNN 모델의 동시 실행이 필수적으로 요구된다. 이러한 시스템은 카메라, LiDAR, 레이더 등 다양한 센서 입력을 여러 DNN이 병렬로 처리하며, 각 DNN은 특정한 인지 작업에 최적화되어 있다.

자율주행 시스템을 예로 들면, 하나의 DNN은 2차원 카메라 영상에서 보행자를 탐지하고, 다른 DNN은 3차원 LiDAR 포인트 클라우드로부터 주행 가능한 영역을 분할한다. 또한, 세 번째 네트워크는 이러한 입력들을 융합하여 더욱 견고한 주행 경로 예측을 가능하게 한다. 이처럼 여러 DNN을 동시에 실행함으로써, 시스템은 주변 환경을 정확하고 안정적으로 인지하고 제어할 수 있게 된다. 그러나 NVIDIA(사)의 Jetson AGX Xavier[1)]와 같은 자원이 제한된 임베디드 환경에서는, 다중 DNN 모델의 동시 실행이 한정된 GPU 자원에 대한 경합을 유발하게 된다(Li, E., Zeng, et al, 2019). 이는 시스템의 전체 처리 시간을 증가시키고, 추론 지연을 초래하여 실시간 성능이 요구되는 시스템에서는 치명적인 문제가 될 수 있다.

따라서, 다중 DNN을 안정적이고 효율적으로 실행하기 위해서는, 시스템의 작업 부하를 반영하는 스케줄링 기법과 추론 속도를 향상시킬 수 있는 적응형 경량화 모델이 요구된다.

기존의 모델 단위 또는 레이어 단위 기반의 다중 DNN 스케줄링 기법은 각각 한계점이 존재한다. 먼저, 모델 단위 스케줄링의 경우 하나의 모델이 GPU 자원을 독점적으로 점유함으로써, 다른 모델들의 추론 지연을 초래하는 문제가 발생한다. 반면, 레이어 단위 스케줄링은 다수의 레이어 간 의존성으

---

1) Nvidia, Jetson AGX Xavier Developer Kit,
online: https://developer.nvidia.com/embedded/jetson-agx-xavier

로 인해 스케줄링 복잡도가 높고, 빈번한 의사결정 과정에서 상당한 실행 오버헤드가 발생하므로, 지연에 민감한 시스템에서는 적용이 어렵다는 한계를 가진다(Niu, W., et al, 2021).

본 논문에서는 DNN 모델의 연산을 최적화하는 방안을 두 가지 관점에서 고찰한다. 첫 번째로 모델 자체의 경량화를 통해 연산 효율을 향상시키는 방법이다. 이를 위해 본 연구에서는 분류, 검출, 추적 등 다양한 시각지능 모델을 대상으로, Pruning(Han, S., et al, 2015)와 Quantization(Han, S., et al, 2015; Jacob, B., et al, 2018)를 각각 단독으로 적용한 경우와 두 기법을 복합적으로 결합한 경량화 기법을 함께 적용하였다. 이러한 다양한 조합을 통해 각 기법의 성능 차이를 실험적으로 분석하였다. 또한, 모델 수준의 경량화뿐만 아니라 TensorRT와 같은 런타임 수준 최적화 기법을 추가로 적용하여 실행 환경의 개선 효과를 함께 검증하였다. 제안된 복합형 경량화 기법은 단일 기법 대비(Pruning 또는 Quantization) 모델의 연산량과 파라미터 수를 더 효과적으로 감소시키면서도 정확도 손실을 최소화하는 특성을 보였다. 아울러, TensorRT 기반 런타임 최적화를 적용함으로써 기존 경량화 기법만으로는 달성하기 어려운 추론 속도 향상, 메모리 사용 효율 개선, 그리고 실행 단계에서의 최적화 효과를 추가적으로 얻을 수 있음을 확인하였다.

두 번째는 시스템의 자원 사용률과 실행시간의 변화를 실시간으로 모니터링하고, 그 결과에 따라 각 DNN의 실행 시점과 경량화 수준을 적응적으로 조정할 수 있는 동적 실행 제어 프레임워크를 제안한다.

본 논문에서는 DNN을 블록이라는 새로운 개념의 실행 단위로 분할하며, 이는 레이어보다 덜 세분화된 단위이면서도, 전체 모델보다는 세분화된 중간 수준의 실행 단위이다. 이러한 블록 단위 실행은 자원 독점을 방지함과 동시에 동기화 오버헤드를 최소화하며 유연한 스케줄링을 가능하게 한다. 또한, 실행 과정에서 발생하는 지연 시간의 변화를 반영하여 런타임 적응성을 향상시키기 위해, 본 연구에서는 *LAG*라는 정량적 지표를 새롭게 도입하였다. *LAG*는 실제 실행시간과 데드라인의 차이로 정의되며, 이를 통해 현재 시스템이 데드라인 대비 지연 정도를 정량적으로 평가할 수 있다. 또한, *LAG*의 추세를 분석하기 위해 EMA(Exponential Moving Average)를 적용하였다.

EMA는 최근 데이터에 크거나 작은 가중치를 부여하여 시간에 따른 변화를 반영하면서 평균을 계산하는 방식이다. 따라서 가중치의 크기를 조절함으로써 단기적 변화에 민감하게 반응하거나, 장기적 추세를 안정적으로 반영하도록 조정할 수 있다. 본 논문에서는 이러한 특성에 착안하여, 단기 EMA로는 일시적인 지연 변화를, 장기 EMA로는 전체적인 추세를 분석함으로써, 지연의 단기적 변동성과 장기적 경향을 모두 반영할 수 있는 분석 구조를 구축하였다. 또한, 단기 EMA와 장기 EMA의 차이를 기반으로 지연 추세의 변화를 정량적으로 표현하는 새로운 지표인 LAG_{trend}를 정의하였으며, 이를 제어 신호로 활용하여 실행 중 각 블록의 적절한 수준(원본 또는 압축된 블록)을 동적으로 선택하도록 설계하였다. 제안된 프레임워크는 LAG_{trend}값에 따라 각 블록의 압축 수준을 실시간으로 조정함으로써, 급격한 지연 변화에는 민감하게 대응하면서도 단기적인 변동이나 일시적 스파이크에는 안정적으로 동작할 수 있다. 또한, 이러한 통합적 설계를 통해 블록 단위 스케줄링을 기반으로 GPU 자원 활용도를 최적화하고, 다중 DNN이 동시에 실행되는 환경에서도 안정적인 성능을 유지할 수 있다.

본 논문은 2장에서 연구에 필요한 배경 지식과 문제 정의에 관한 내용을 서술한다. 그리고 3장에서는 연산 효율 향상을 위해 여러 경량화 기법을 복합적으로 적용하는 방식을 제안하고, 4장에서는 시스템 자원 사용률과 지연 변화를 기반으로 DNN의 실행 시점과 경량화 레벨을 동적으로 조정하는 동적 실행 제어 프레임워크를 제시한다. 5장에서는 실험을 통해 제안되는 방법들에 대하여 실험 결과를 확인하고 마지막으로 6장에서 결론으로 마무리한다.

제 2 장 연구 배경 및 문제 정의

제 1 절 임베디드 시스템에서의 다중 DNN 실행의 특성 및 제약사항

임베디드 GPU 플랫폼(Satyanarayanan, M., 2017)은 다양한 실시간 지능형 시스템에서 점차 폭넓게 활용되고 있다. 대표적인 예로는 NVIDIA(사)의 Jetson 시리즈(AGX Xavier, Orin), Intel(사)의 OpenVINO[2] 기반 VPU 플랫폼의 Intel Movidius[3], 그리고 Google(사)의 Coral Edge TPU[4] 등이 있으며, 이들은 높은 연산 효율성과 낮은 전력 소모로 인해 다양한 엣지 환경에 적용되어 왔다. 특히, NVIDIA Jetson AGX Xavier는 고성능 GPU를 탑재하고 있음에도 불구하고, 서버급 GPU에 대비 Streaming multiprocessor의 수, 메모리 대역폭, 전력 소비 측면에서 구조적인 제약을 지닌다. 이러한 제약으로 인해 다중 DNN 모델이 동시에 실행될 경우 GPU 자원 경합이 발생할 수 있으며, 그 결과 시스템 전체의 응답 시간이 증가하고 처리량이 감소하는 문제가 나타난다. Jetson 플랫폼에서의 딥러닝 연산은 주로 CUDA API[5]를 기반으로 하는 cuDNN[6] 및 cuBLAS[7]와 같은 고성능 라이브러리를 통해 수행된다. 이들 라이브러리는 CUDA 스트림[8]을 활용하여 커널의 병렬 실행과 비동

2) Intel, OpenVINO Toolkit,
online: https://docs.openvino.ai/

3) Intel, Movidius VPU,
online: https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu.html

4) Google, Coral Edge TPU,
online: https://coral.ai/

5) Nvidia, CUDA Toolkit Documentation,
online: https://developer.nvidia.com/cuda-toolkit

6) Nvidia, cuDNN: GPU-Accelerated Deep Neural Network Library,
online: https://developer.nvidia.com/cudnn

7) Nvidia, cuBLAS: GPU-Accelerated Basic Linear Algebra Subprograms.
online: https://docs.nvidia.com/cublas/

8) Nvidia, CUDA C++ Programming Guide.
online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

기 데이터 전송을 지원한다.

그러나 여러 DNN 모델이 서로 다른 CUDA 스트림에서 병렬로 실행되더라도, 이들 스트림은 결국 Streaming multiprocessor(SMs) 및 L2 캐시와 같은 물리적 GPU 자원을 공유하게 된다. 그 결과, 커널 실행 시점, 스트림 동기화 지연, 자원 할당 순서 등에 따라 상호 간섭이 발생할 수 있다. 이러한 구조적 특성은 필연적으로 시스템 전체의 지연증가 또는 처리량 저하를 초래하게 된다.

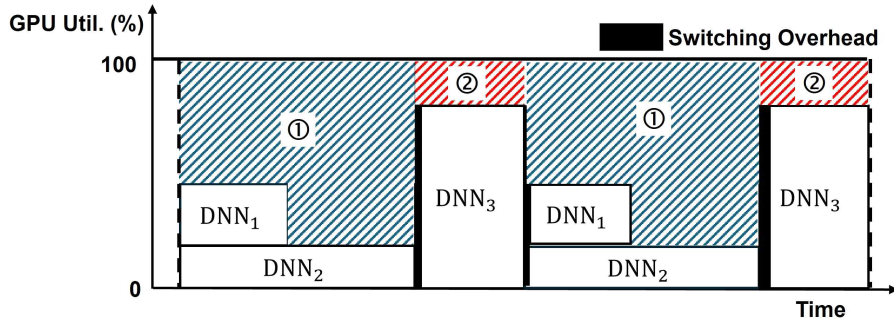
더불어, DNN 모델의 구조적 특성 또한 이러한 문제를 더욱 심화시킬 수 있다. 일반적으로 DNN은 여러 개의 레이어로 구성되며, 각 레이어는 서로 다른 연산량과 메모리 요구량을 가진다. 많은 경우, 전체 연산의 상당 부분이 특정 레이어나 모듈에 집중되는 경향이 있다. 예를 들어, Faster R-CNN(Ren, S., et al, 2015)과 같은 객체 검출 모델은 백본(backbone), 영역 제안 네트워크(Region Proposal Network, RPN), 검출 헤드(detection head)로 구성되며, 이 중 백본과 검출 헤드에 연산이 집중되어 비대칭적인 연산 부하 분포를 보인다. 이처럼 모델별로 연산 특성과 자원 사용 양상이 크게 다르므로, 실행시간 역시 모델마다 상이하다. 그러나 기존의 스케줄링 기법들은 이러한 연산 특성이나 자원의 요구사항을 충분히 고려하지 않고, 모델을 고정된 순서로 반복 실행하는 경우가 많다. 이로 인해 연산 부하가 높은 레이어에서 지연이 누적되어 GPU 자원이 비효율적으로 사용되고, 결과적으로 시스템 전체의 응답 시간이 증가시키는 결과를 초래한다.

제 2 절 문제 정의

1) 문제 정의 : 스케줄링 단위

다중 DNN 모델의 병렬 실행을 위한 스케줄링 방식은 실행 단위의 크기에 따라 크게 두 가지로 분류할 수 있다. 첫 번째로는 전체 모델을 하나의 실행 단위로 취급하는 방법인 모델 단위(model-level, coarse-grained) 실행 방식이며 두 번째로는 각 레이어를 개별 실행 단위로 처리하는 방법인 레이어 단위(layer-level, fine-grained) 실행 방식이다(Kang, Y., et al, 2017). 그러

HANSUNG UNIVERSITY

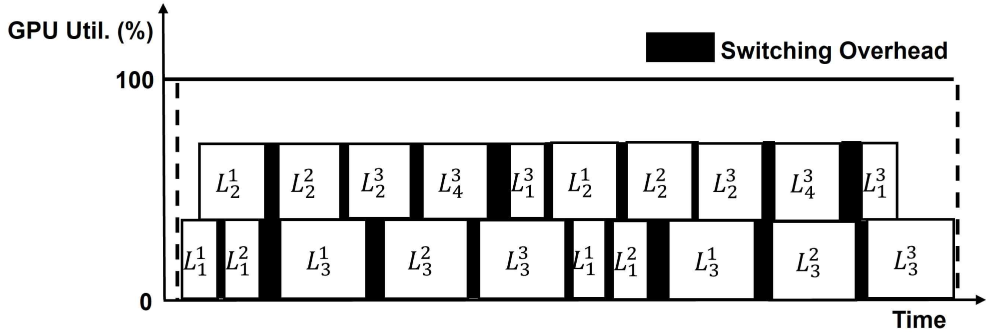


[그림 2-1] 모델 단위의 다중 DNN 병렬 실행 구조

나 임베디드 GPU 환경의 자원 제약 아래서는 이러한 접근법들이 구조적인 한계를 보인다.

[그림 2-1]은 실제 GPU 자원 활용 측면에서 모델 단위 실행 방식이 어떻게 동작하는지를 보여준다. 그림에서 영역 ①은 DNN_3이 GPU에 접근하지 못하고 대기해야 하는 유휴 시간을, 영역 ②는 DNN_1과 DNN_2가 대기하는 시간을 의미한다. 또한, 검은색 막대는 모델이나 레이어 간 전환 시 발생하는 스위칭 오버헤드를 나타낸다. 모델 단위 실행 방식은 각 DNN 모델을 하나의 불가분 실행 단위로 처리하여, 비교적 단순한 방식으로 병렬 스케줄링을 가능하게 한다. 이 방식은 구현이 간단하다는 장점이 있지만, 하나의 모델이 GPU를 장시간 점유하게 되어 다른 모델들이 대기 상태에 머무르게 되는 문제가 발생한다. 이는 GPU 자원을 비효율적으로 사용할 뿐만 아니라, 모델 간 자원 경합을 빈번하게 유발하여 결과적으로 시스템의 전체 지연 시간을 증가시키고 처리 효율을 저하 시키는 원인이 된다.

[그림 2-2]는 레이어 단위 실행 방식을 나타내며, DNN 모델의 각각의 레이어를 개별 실행 단위로 설정하여 병렬로 실행하는 방법이다. 이 방식은 자원 활용도를 향상시킬 수 있지만, 실행 단위가 지나치게 세분화될 경우 CUDA 스트림 할당, 동기화, 메모리 관리 등의 과정에서 반복적인 오버헤드가 발생한다. 따라서 모델 단위 실행과 레이어 단위 실행은 각각 장단점을 지니고 있으나, 임베디드 시스템 환경에서 요구되는 지연 시간 최소화, 자원 활용 최적화, 실행 오버헤드 감소를 동시에 달성하기에는 근본적인 한계가 존재



[그림 2-2] 레이어 단위의 다중 DNN 병렬 실행 구조

한다.

이러한 문제점을 해결하기 위해, 본 논문에서는 모델 단위와 레이어 단위의 중간 수준에 해당하는 블록 단위 실행 개념을 도입한다. 여기서 블록이란 여러 개의 연속된 레이어로 구성된 기능적 그룹으로 정의되며, 이를 통해 모델 단위 실행보다 세밀한 제어가 가능하고, 레이어 단위 실행보다 높은 안정성을 확보할 수 있다. 이러한 개념을 바탕으로, 본 연구에서는 GPU 자원 경합을 완화하고 전체 시스템의 실행 효율을 향상시키기 위한 새로운 스케줄링 방식을 제안한다.

HANSUNG UNIVERSITY

2) 문제 정의 : 기존 경량화 기법의 한계점

경량화 기법 중 Pruning과 Quantization은 모델의 파라미터 수를 줄이고 연산량을 감소시켜 효율적인 경량화를 달성할 수 있다는 점에서 폭넓게 활용되어 왔다. 그러나 위 기법을 임베디드 시스템 환경에 직접 적용할 경우, 하드웨어 구조적 제약으로 인한 문제점이 발생한다.

첫 번째로, Quantization의 경우 가중치를 8비트 또는 그 이하의 비트 수로 표현하더라도, 임베디드 GPU는 메모리를 주로 32비트 단위로 로드하는 구조를 갖는다. 이로 인해 Quantization된 가중치가 메모리 정렬 기준과 일치하지 않을 경우, 추가적인 메모리 접근 연산이 필요하거나 메모리 정렬 구조와의 불일치로 인해 성능이 저하되는 문제가 나타난다. 또한, Pruning은 모델의 크기를 줄이는 데 효과적이지만, 0으로 마스킹된 가중치가 연산 경로에 여

전히 존재하기 때문에, GPU에서는 이러한 요소가 실제 연산 과정에서 완전히 제외되지 못하며, 결과적으로 이론적인 연산량 감소에도 불구하고 실행시간 단축 효과는 거의 나타나지 않는다(Yao, Z., et al, 2019). 이와 달리 NPU는 저비트 정밀도 연산과 희소 구조 활용을 전제로 설계된 경우가 많아, Quantization된 가중치나 Pruning으로 제거된 파라미터를 실제 연산 경로에서 효과적으로 제거하여 연산을 줄일 수 있다. 이러한 구조적 특성으로 인해 Pruning과 Quantization 기반 경량화 기법은 일반적으로 임베디드 GPU보다 NPU에서 더 높은 성능 향상 효과를 보인다(Parashar, A., et al, 2017).

두 번째로, 위에서 언급한 경량화 기법은 모두 정적 방식의 경량화로, 사전에 모델 전체를 일괄적으로 경량화한 후 고정된 구조를 유지하는 방식이기 때문에, 실행 중 시스템 부하나 환경 변화에 따라 부분적인 구조 조정이나 동적 교체가 불가능하다는 한계를 가진다. 이로 인해 실제 실행 과정에서는 일시적으로 과도하게 경량화된 모델 전체를 사용하는 비효율이 발생할 수 있으며, 이는 곧 불필요한 정확도 손실과 성능 저하로 이어질 수 있다.

RNP(Runtime Neural Pruning)(Lin, J., et al, 2017)은 이러한 한계를 해결하기 위한 동적 압축 기법의 하나로, 실행 중 입력 샘플에 따라 각 레이어의 채널을 선택적으로 Pruning하는 방법을 사용한다. RNP는 강화학습 기반 정책을 통해 입력 이미지의 난이도에 따라 Pruning 비율을 적응적으로 결정함으로써, 연산 비용을 줄이면서도 원래 네트워크의 표현력을 일정 수준 유지한다. 그러나 RNP는 입력 단위로 연산량을 조절할 수 있는 유연성을 제공하지만, 각 레이어별로 얼마만큼 연산을 줄일지 정밀하게 제어하기 어렵다. 따라서 지연에 민감한 임베디드 시스템 환경에서는 정확한 지연 제어나 예측이 어렵다는 한계가 있다. 최근에는 PyTorch[9]에서 제공하는 TorchScript[10] 및 LibTorch[11] C++ API와 같은 프레임워크의 발전으로 인해, C++ 환경에서도 모델을 레이어 또는 블록 단위로 세밀하게 제어하고 모듈화하는 것이 가능해

9) PyTorch Dev Team, PyTorch,
online: https://pytorch.org/

10) PyTorch Dev Team, TorchScript Documentation,
online: https://pytorch.org/docs/stable/jit.html

11) PyTorch Dev Team, LibTorch C++ API,
online: https://pytorch.org/cppdocs/

졌다. 이로써 실행 과정에서 특정 모듈을 선택적으로 교체하거나 실행 경로를 동적으로 조정하는 기능을 구현할 수 있는 기반이 마련되었다.

그럼에도 불구하고 Quantization과 Pruning 기법은 파라미터 수 감소 및 메모리 절감 측면에서 여전히 유의미한 이점을 제공하므로, 본 논문에서는 분류, 검출, 추적 등 다양한 시각 지능 모델에 대해 두 기법을 단독 또는 혼합 적용하여 그 성능 차이를 실험적으로 분석하였다. 분류 모델의 경우 다수의 선행연구에서 Quantization, Pruning, 저랭크 분해 등의 경량화 기법이 적용되어 그 효과가 폭넓게 검증됐다. 하지만 검출 및 추적 모델은 RPN이나 다중 분기 헤드(Multi-branch Head)와 같은 복잡한 계층 구조와 높은 연산 의존성으로 인해 이러한 경량화 기법을 적용한 사례가 드물다. 그러나, 구조적으로는 분류 모델과 유사한 기반 아키텍처를 공유하기 때문에, Pruning 및 Quantization 경량화 기법이 검출 및 추적 모델에도 실질적으로 적용 가능함을 본 연구에서 실험적으로 확인하였다.

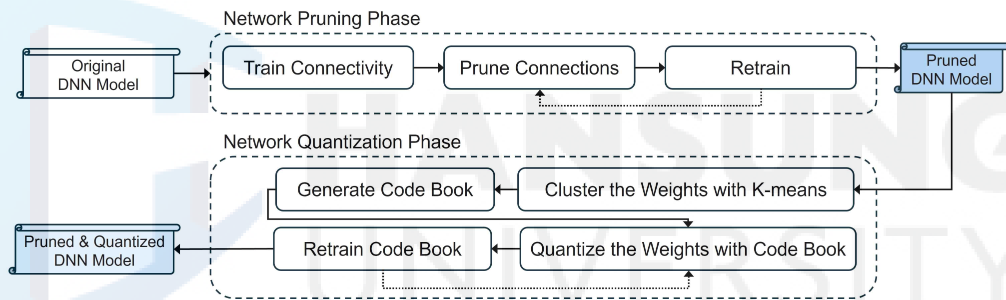
하지만 이러한 정적 경량화 기법은 서버 시스템 환경에서는 효과적일 수 있으나, 임베디드 환경에서는 한계가 존재한다. 이에 본 논문에서는 이러한 제약을 극복하고, 임베디드 환경에서도 효과적으로 활용 가능한 근사화 기반 경량화 기법인 LRD(Low-Rank Decomposition)(Hu, E. J., et al, 2022)를 적용하였다. LRD는 레이어별 저차원 근사화 통해 연산량과 메모리 사용량을 감소시키면서도, 모델의 구조적 유연성을 유지할 수 있다. 이를 통해 임베디드 환경에서 발생하는 시스템 부하 변화에 더 동적으로 대응할 수 있다. 이에 기반하여 본 논문에서는 모델을 기능적 블록 단위로 분할하고, 각각 세 가지 수준의(Level 0, 1, 2) 다른 블록을 사전에 준비하였으며, Level 0은 원본 블록이고 Level 1과 Level 2는 LRD 기반으로 내부 파라미터 수를 점진적으로 감소시킨 경량화된 모델의 블록으로, 동일한 입출력 구조를 유지하면서 내부 연산량을 줄여 실행 시 연산 부하를 유연하게 조절할 수 있다.

본 논문에서는 이러한 복수의 수준의(Level 0, 1, 2) 블록을 사전에 생성하고, 실행 중 시스템 상태에 따라 선택적으로 교체함으로써, 모델 전체를 교체하지 않고도 지연 시간-정확도 간의 균형을 달성할 수 있는 새로운 프레임워크를 제안한다.

제 3 장 모델 경량화 및 추론 최적화를 위한 복합적 설계 기법 제안

본 장에서는 모델 구조 차원의 경량화와 하드웨어 수준의 추론 최적화를 다양한 시각지능 모델에 적용함으로써, 시각지능 모델의 연산량, 메모리 사용량, 실행 지연을 전반적으로 감소시키는 최적화 기법을 제안한다.

제 1 절 Pruning-Quantization 통합 경량화 구조 설계 제안



[그림 3-1] Pruning과 Quantization을 순차적으로 적용한 구조

본 논문에서 사용된 경량화 구조의 전체 흐름은 Deep Compression(Han et al., 2015)에서 제안된 Pruning, Quantization, Huffman Coding의 3단계 순차 적용 파이프라인을 기본 구조로 한다. Deep Compression은 연산량과 메모리 요구량이 매우 큰 신경망을 임베디드 환경에서 효율적으로 배치하기 위해 제안된 3단계 압축 기법으로, 파라미터 수를 감소시키면서도 정확도 손실을 최소화하여 모델 크기를 수십 배 축소할 수 있음을 보인 대표적인 연구이다.

하지만 본 절에서는 [그림 3-1]에 보이는 바와 같이, Pruning과 Quantization의 두 단계를 중심으로 한 파이프라인만을 채택하고, Huffman Coding 단계는 제외하였다. 즉, Deep Compression에서 제안된 순차적 경량

화 개념을 기반으로 Pruning-Quantization 파이프라인을 실제 시스템 환경에 적합하도록 재구성하였다. 나아가, 제안된 구조를 분류 모델뿐 아니라 검출 및 추적 모델까지 확장 적용하여, 다양한 시각지능 모델에서 모델 용량 및 연산량 감소 효과를 실험적으로 검증하였다.

[그림 3-1]의 위쪽 점선 영역에 해당하는 Network Pruning 단계에서는 먼저 학습이 완료된 모델을 바탕으로 가중치의 크기를 활용해 각 연결의 상대적 기여도를 평가한다. 이후 레이어별 가중치 분포로부터 설정된 임계값 이하의 가중치를 출력에 미치는 영향이 낮은 요소로 간주하여 제거함으로써, 네트워크 구조를 희소 형태로 재구성한다. 본 연구에서는 각 레이어의 가중치에 대해 표준편차를 계산하고, 여기에 사전 정의된 비율을 곱해 임계값을 산정하는 방식을 적용하였으며, 이 임계값보다 절댓값이 작은 가중치가 우선적으로 제거되도록 구성하였다. Pruning으로 얻어진 희소 구조는 연산 과정에서 불필요한 곱셈 및 덧셈 연산을 제거하여 실제 연산량을 감소시키며, 저장 단계에서는 CSR/CSC 기반의 희소 행렬 압축 방식을 적용하여 0이 아닌 가중치만을 효율적으로 저장할 수 있으므로, 전체 파라미터 저장 비용 역시 크게 절감된다.

Network Pruning 단계를 통해 네트워크 구조가 희소화되면, Network Quantization 단계([그림 3-1]의 아래 점선 영역)에서 Quantization과 Weight Sharing을 적용하여 추가적인 압축을 수행한다. Quantization은 각 가중치를 저 정밀도 표현으로 변환하여 가중치를 저장하는 데 필요한 비트 수를 줄이며, Weight Sharing은 전체 가중치를 k개의 클러스터로 분할하여 클러스터별 대푯값만 저장하도록 구성된다. 이때 모든 가중치는 실수 가중치 대신 클러스터 인덱스를 저장하므로, 필요한 저장 공간은 k개의 공유 가중치와 해당 가중치가 매핑되는 클러스터 인덱스를 나타내는 $\log_2(k)$비트로 구성된다. 따라서 원래 가중치가 n개이고, 각 가중치를 b비트로 표현되는 네트워크에서 모든 가중치를 k개의 공유 가중치로 제한하면, Weight Sharing 적용 시의 압축률 r은 식 (1)과 같이 계산된다.

$$r = \frac{nb}{nlog_2(k) + kb} \tag{1}$$

HANSUNG UNIVERSITY

공유 가중치를 결정하는 과정에서는 학습이 완료된 모델의 각 레이어별 가중치에 대해 k-means 클러스터링을 적용한다. 이를 통해 동일한 클러스터에 속한 가중치는 동일한 값을 공유하게 되며, 가중치 공유는 레이어 내부에서만 이루어지고 레이어 간에는 적용되지 않는다. 이를 통해 Weight Sharing은 원래 가중치 분포를 최대한 유지하면서도 필요한 실수 가중치의 개수와 저장 비용을 효과적으로 감소시키는 데 기여한다.

중심점(centroid) 초기화 방식은 클러스터링 모델의 표현력에 직접적인 영향을 미치는 중요한 요소이다. 본 연구에서는 가중치 분포의 편향을 최소화하기 위해 전체 가중치 범위를 균등하게 분할하는 Linear initialization을 사용하였다. Weight Sharing이 적용된 구조에서는 가중치는 공유 가중치 테이블을 가리키는 인덱스가 추가로 저장된다. 따라서 역전파 단계에서는 각 공유 가중치에 대한 그래디언트를 계산하고, 이를 이용해 공유 가중치를 업데이트한다. 손실 함수를 $\mathscr{L}$, i번째 열, j번째 행에 있는 가중치를 W_{ij}, 해당 원소 W_{ij}의 중심점의 인덱스를 I_{ij}, 그리고 해당 레이어의 k번째 중심점을 C_k라 정의한다. 이때 특정 가중치가 클러스터 k에 속하는지를 나타내는 표시함수 $\mathbb{1}(.)$를 이용하면, 중심점에 대한 그래디언트는 아래 식(2) 와 같이 계산된다.

$$\frac{\delta \mathcal{L}}{\delta C_k} = \sum_{i,j} \frac{\delta \mathcal{L}}{\delta W_{ij}} \frac{\delta W_{ij}}{\delta C_k} = \sum_{i,j} \frac{\delta \mathcal{L}}{\delta W_{ij}} \mathbb{1}(I_{ij} = k) \quad (2)$$

이처럼 Pruning−Quantization 기반 경량화 구조를 다양한 시각지능 모델에 적용한 실험을 통해, 모델의 정확도를 유지하면서도 파라미터 규모와 연산량을 효과적으로 감소시킬 수 있음을 확인하였다.

제 2 절 TensorRT를 활용한 추론 최적화 구조 설계 제안

본 절에서는 모델의 구조를 변경하거나 재학습을 수행하지 않고도 추론 속도를 향상시키기 위해, NVIDIA(사)에서 제공하는 고성능 딥러닝 추론 엔

진인 TensorRT[12] 기반의 런타임 최적화 방식을 적용하였다. TensorRT는 이미 학습이 완료된 모델을 재학습 과정 없이 하드웨어에 최적화된 형태로 변환하여 추론 속도를 크게 향상시킬 수 있는 런타임 기반 최적화 도구이다.

TensorRT는 연산 그래프 최적화(graph optimization), 커널 자동 선택(kernel auto-tuning), 연산 정밀도 변환(precision calibration) 등 다양한 최적화 기법을 제공함으로써, 모델의 실행 경로를 단순화하고 GPU 연산 효율을 극대화한다. 이러한 방식은 앞 절에서 다룬 방식과 달리 추가적인 학습 과정을 진행하지 않기 때문에 시간과 연산 자원을 절약할 수 있으며, GPU 하드웨어 구조에 특화된 최적화를 통해 추론 효율과 처리 속도를 동시에 향상시킬 수 있다는 장점이 있다. TensorRT는 여러 단계의 내부 최적화 과정을 수행한다.

첫 번째로, 네트워크 구조 최적화 단계에서는 모델의 연산 그래프를 정적으로 분석하여 불필요한 중간 연산을 제거하고, 연산 경로를 단순화하였다. 구체적으로, 그래프상에서 같은 입력이나 상숫값을 반복적으로 사용하는 연산 노드를 탐지하여 이를 하나의 공통 연산으로 통합하는 과정을 거친다. 또한, 연속적으로 수행되는 Convolution, Batch Normalization, ReLU 등의 연산을 하나의 커널로 병합하는 Layer Fusion을 통해 커널 호출 횟수를 줄이고, 각 연산 간 데이터 이동 및 메모리 접근 오버헤드를 최소화하였다. 이러한 최적화 과정은 실행 그래프의 복잡도를 줄이고, GPU 자원의 활용 효율을 극대화함으로써 전체 추론 단계에서 발생하는 지연 시간을 효과적으로 감소시킬 수 있다.

두 번째로 모델이 수행하는 연산의 정밀도를 조정하는 과정을 거친다. 기본적으로 딥러닝 모델은 32비트 부동소수점(FP32)으로 계산되지만, TensorRT는 이를 FP16(16비트) 또는 INT8(8비트 정수)로 변환하여 계산하도록 지원한다. 정밀도를 낮추면 한 번에 더 많은 데이터를 처리할 수 있어 속도가 빨라지고, 메모리 사용량도 감소한다. 특히 INT8로 변환하는 단계에서는 보정(Calibration) 과정을 통해 각 텐서의 데이터 분포를 분석하고, 가장 적절한 스케일 값을 찾아 Quantization으로 인한 오차를 최소화할 수 있다.

12) Nvidia, TensorRT Documentation

마지막으로, 이러한 최적화가 적용된 모델을 TensorRT의 builder 및 runtime API를 이용해 엔진 형태로 변환한다. 엔진은 GPU 내부에서 직접 실행 가능한 형태의 바이너리 파일로, CUDA 스트림을 활용해 여러 연산을 동시에 수행할 수 있다. 이를 통해 모델은 실시간 추론 환경에서도 지연 없이 빠르게 동작할 수 있다. 결과적으로, TensorRT 기반 최적화는 학습된 모델의 구조를 변경하지 않으면서도 임베디드 GPU 환경에서 실시간 처리가 가능한 수준의 성능 개선을 달성하였다. TensorRT 복잡한 최적화 과정을 사용자가 직접 구현하지 않아도 API를 통해 손쉽게 적용할 수 있으며, 다양한 딥러닝 프레임워크(PyTorch, TensorFlow)에서 사용이 용이하다는 장점을 가진다.

HANSUNG
UNIVERSITY

제 4 장 블록 수준 스케줄링 기반 다중 DNN 실행 최적화 기법 제안

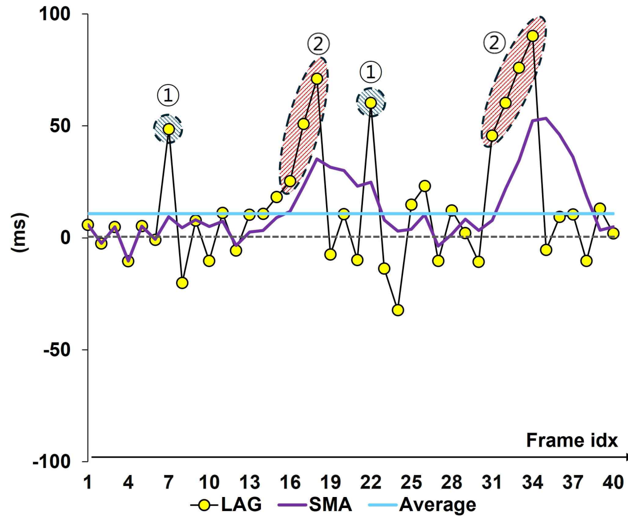
본 장에서는 임베디드 환경에서 발생하는 GPU 자원 경합으로 인한 실행 지연을 완화하고, 다중 DNN 모델을 효율적으로 병렬 실행하기 위한 스케줄링 프레임워크를 제안한다. 제안하는 프레임워크는 각 DNN 모델을 기능 단위의 블록으로 분할하고, 각 블록의 실행시간을 실시간으로 모니터링 한다. 실행 중에는 블록을 원본 블록(Level 0) 또는 특정 압축 수준(Level 1 또는 Level 2)을 적용한 경량화 블록으로 동적으로 교체할 수 있다. 또한 블록의 실행 순서를 조정함으로써, 여러 DNN 모델이 공유하는 GPU 자원을 보다 균형 있게 활용할 수 있다. 이를 통해 전체 시스템의 지연 시간과 정확도 간의 균형을 효과적으로 유지할 수 있다.

제 1 절 블록 단위 지연 시간 추정을 통해 각 DNN 모델의 실행 동작 분석

본 절에서는 각 블록의 실행 지연의 정도를 정량적으로 표현하기 위해 LAG(Bateni, S., et al, 2020) 지표를 도입한다. LAG는 특정 블록의 실제 실행시간과 사전에 정의된 데드라인 간의 차이로 정의되며, 식(3) 과 같이 정의된다.

$$LAG_i = E_i - D_i \tag{3}$$

여기서 i는 블록의 인덱스를 의미하며, E_i는 i번째 블록의 실제 실행시간, D_i는 해당 블록의 사전에 정의된 데드라인을 의미한다. LAG_i 값이 양수이면 해당 블록이 데드라인을 초과하여 실행 지연이 발생했음을 의미하며, 음수이면 데드라인보다 빠르게 실행되었음을 의미한다. 각 블록의 데드라인 D_i는 블



[그림 4-1] LAG, LAG의 SMA, 평균 LAG간의 비교

록 수준의 실행 통계로부터 경험적으로 산정되어 설정된다. 구체적으로는 실험에 사용된 세 개의 DNN 모델을 블록 단위로 동시에 실행하는 환경에서 실제 실행시간 데이터를 수집하고, 각 블록에 대해 실행시간 분포를 히스토그램으로 분석하였다. 수집된 실행시간을 일정한 구간(예: 10ms)으로 분할한 뒤 각 구간의 발생 빈도를 계산하였으며, 가장 높은 빈도를 보인 구간을 해당 블록의 대표 실행 범위로 정의하였다. 이후 해당 구간의 최댓값을 데드라인 D_i로 설정하였다. 예를 들어, VGG16 모델의 특징을 추출하는 블록의 실행시간이 260~270ms 구간에 가장 많이 분포하는 경우, 해당 블록의 데드라인 D_i는 270ms로 설정된다. 이러한 절차는 일시적인 스파이크나 이상치에 의한 왜곡을 최소화하고, 각 블록의 일반적이고 안정적인 실행 특성을 반영하는 현실적이며 견고한 지연 평가 기준을 제공한다.

그러나 LAG는 특정 블록의 실행 지연의 여부를 직관적으로 판별할 수 있는 지표이지만, 순간적인 값에만 의존할 경우 여러 한계를 가진다. 프레임 간 실행시간의 불가피한 변동으로 인해 LAG 값은 0 부근에서 빈번하게 변동하며 양수와 음수 사이에서 반복적으로 전환한다. 이러한 변화는 실제로 의미 있는 지연이 존재하지 않음에도 불필요한 스케줄링 결정(예: 압축 수준 전환, 실행 순서 조정 등)을 유발할 수 있으며, 그 결과 시스템 오버헤드가 증가할

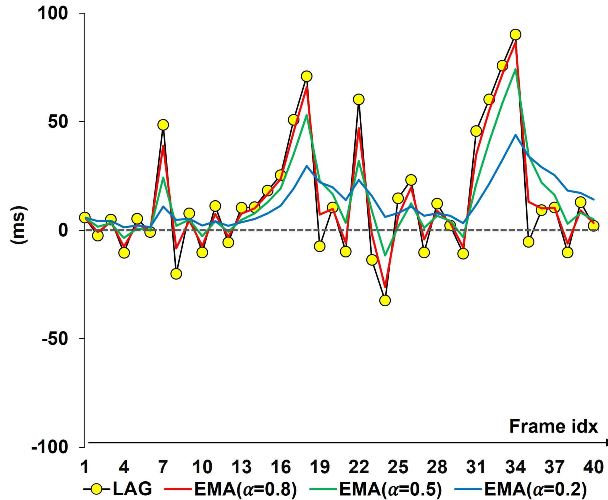
수 있다. [그림 4-1]에서 보이듯이, *LAG* 값(노란색 점)이 0을 중심으로 진동하는 구간에서는 실제로 성능 저하가 없음에도 불구하고 빈번한 전환이 발생하여 불필요한 시스템 오버헤드를 유발한다. 이러한 관찰 결과는 *LAG*의 순간적인 변동에만 의존할 경우 시스템이 쉽게 불안정해질 수 있음을 보여준다.

또한 *LAG*는 각 프레임의 순간적인 편차만 반영하기 때문에 실행시간 변화가 급격하게 발생하는 경우와 점진적으로 누적되는 경우를 구분하지 못한다. 예를 들어, [그림 4-1]의 영역 ①은 단일 프레임에서 실행시간이 일시적으로 급증하는 일시적인 스파이크 현상을 나타내지만, 영역 ②는 여러 프레임에 걸쳐 지연이 점진적으로 누적되는 패턴을 보여준다. 두 경우 모두 유사한 크기의 *LAG* 값을 생성할 수 있지만, 시스템 관점에서는 서로 다른 의미를 갖는다. 전자는 단기적 스파이크로서, 이후 프레임에서 정상 범위로 복귀한다면 전체 성능에 큰 영향을 미치지 않는다. 반면 후자는 지속적인 지연을 초래하며, 시스템 안정성을 직접적으로 위협하는 요인이 된다. 그러나 *LAG*는 순간적인 편차만을 반영하기 때문에, 두 상황을 동일한 패턴으로 오인할 수 있으며, 이로 인해 불필요한 전환이 발생하거나, 반대로 누적 지연을 제때 탐지하지 못하는 문제가 발생할 수 있다. 평균 기반의 통계적 방법도 고려할 수 있지만, 이러한 방식 역시 명확한 한계를 드러낸다.

[그림 4-1]에서 파란색 선은 *LAG*의 산술 평균을, 보라색 선은 단순 이동평균[13](Simple Moving Average, SMA)을 나타낸다. 산술 평균은 사전에 측정된 실행시간 전체 분포로부터 계산된 단일 고정값만을 제공하므로, 실행 과정에서 발생하는 동적 변화나 시간적 추세를 반영하지 못한다. 따라서 이러한 접근 방식은 지연에 민감한 시스템의 동적 특성을 반영하기에 적합하지 않다.

반면, SMA는 고정된 윈도우 내의 과거 샘플에 동일한 가중치를 부여하기 때문에, *LAG*가 급격히 변할 때 이를 민감하게 반영하지 못한다. [그림 4-1]에서 확인할 수 있듯이, 실행시간이 빠르게 변화하는 구간에서는 SMA가 즉각적으로 반응하지 못하며, 이러한 특성이 고유한 한계로 나타난다. 이러한 한계들은 실행 지연의 전반적인 추세를 보다 안정적으로 반영할 방법의 필요성을 강조한다. 그러나 *LAG*는 연속 함수가 아닌 프레임 단위의 이산적인 편

13) Investopedia. Simple Moving Average (SMA). Investopedia Online Resource.
online: https://www.investopedia.com/terms/s/sma.asp



[그림 4-2] 평활 계수에 따른 LAG와 LAG의 EMA의 비교

찻값으로 표현되기 때문에, 시간적 경향을 추정하기 위한 미분 기반 접근법을 적용하기가 어렵다. 따라서 프레임 간 변화의 추세를 누적해서 반영할 수 있는 대체 대안이 필요하다.

따라서 본 연구에서는 *LAG* 계산에 지수 이동 평균[14](Expoential Moving Average, EMA)을 도입하여, 단기적 변동과 장기적 지연 추세를 안정적으로 포착할 수 있도록 하였다. EMA는 새로운 데이터가 도착할 때마다 최근 값에 더 큰(혹은 작은) 가중치를 부여하며 평균을 갱신하는 방식으로, 식(4)와 같이 정의된다.

$$EMA_t = \alpha \times x_t + (1-\alpha) \times EMA_{t-1}, \qquad 0 < \alpha < 1 \qquad (4)$$

여기서 x_t는 시각 t에서의 새로운 데이터를 나타내며, α는 최근 데이터에 대한 민감도를 조절하는 평활화 계수이다. α가 클수록 *EMA*는 최근 변화에 더욱 민감하게 반응하고, α가 작을수록 장기적인 평균에 가까운 특성을 보인다.

[그림 4-2]는 서로 다른 평활화 계수 α에 따라 *LAG*를 *EMA*에 적용한 결과를 보여준다. α값이 큰 경우(0.8, 빨간색 실선) *EMA* 곡선은 원래 *LAG*

14) NIST/SEMATECH. NIST/SEMATECH e-Handbook of Statistical Methods: Time Series (Moving Averages and Exponential Smoothing). NIST, 2012.
online: https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4.htm

와 매우 유사한 분포를 보인다. 이는 최근 변동에는 민감하게 반응하지만, 안정성 개선 효과는 제한적이다. α = 0.5 (초록색 실선)에서는 EMA가 일시적인 스파이크를 억제하는 동시에 여러 프레임에 걸쳐 지속적인 증가 또는 감소 패턴을 효과적으로 포착하여 전반적인 추세를 반영한다. α = 0.2 (파란색 실선)에서는 EMA 곡선이 가장 안정적으로 보이며 장기적인 추세를 명확히 보여주지만, 최근 값에 대한 민감도는 낮아진다. α 값을 조정함으로써 EMA는 단기 민감도와 장기 안정성 간의 균형을 유연하게 조절할 수 있으며, 이를 통해 실행 지연의 추세를 포착할 수 있는 견고하고 적응적인 메커니즘을 제공한다.

기존 EMA의 정의에서 x_t는 임의의 시점 t에서의 관측값을 의미한다. 본 연구에서는 실행 단위를 블록 수준으로 정의하였으며, 각 블록의 LAG는 개별적으로 계산될 수 있다. 이로 인해 동시에 실행되는 DNN 모델의 블록 수가 증가할수록 관리해야 할 EMA의 개수도 비례하여 증가하므로, 모든 블록에 대해 개별적으로 EMA를 유지하는 것은 비효율적이다. 지연 시간은 블록 단위에서 측정되지만, 블록의 압축 수준을 동적으로 조정하는 제어 로직이 프레임 경계에서 동작한다는 점을 고려할 때, EMA는 블록 수준이 아닌 모델 수준에서 평가하는 것이 더 적절하다.

이를 위해 본 논문에서는 EMA를 시간 t가 아닌 프레임 인덱스를 기준으로 갱신하도록 재정의하였다. x_f는 프레임 f에서의 집계된 LAG_{frame} 값을 의미하며, 이는 해당 프레임에 포함된 모든 블록의 LAG_i 값을 합산하여 산출된다. 이러한 방식은 프레임 전반에 걸쳐 블록 수준의 LAG_i 값으로부터 해당 DNN 모델의 전체 지연 특성을 정량적으로 포착할 수 있도록 한다.

앞서 [그림 4-2]에서 확인할 수 있듯이, EMA의 단기 변동에 대한 민감도와 장기적인 평균에 가까운 안정성은 평활화 계수 α에 따라 달라진다. 이에 본 논문에서는 이러한 두 경향을 동시에 반영하기 위해 두 개의 지표를 병행하여 사용한다. 구체적으로, 최근 변동에 민감하게 반응하는 EMA_{short}와 장기적인 평균을 안정적으로 포착하는 EMA_{long}을 정의하였다. 이 두 지표는 모델별로 독립적으로 유지되며, 이를 통해 각 모델의 특성에 맞는 지연 추세

를 효과적으로 반영할 수 있다.

EMA_{short}은 실행시간의 급격한 변동에 민감하게 반응하도록 설계된 단기 지표로, 비교적 큰 평활화 계수 α를 사용하여 최근 관측값에 높은 가중치를 부여하며, EMA_{long}은 실행시간의 장기적 평균 추세를 반영하는 지표로, 비교적 작은 계수 β를 사용하여 과거 값의 영향을 넓게 포함한다. 이러한 두 EMA 값은 각각 식(5), 식(6)과 같이 정의된다.

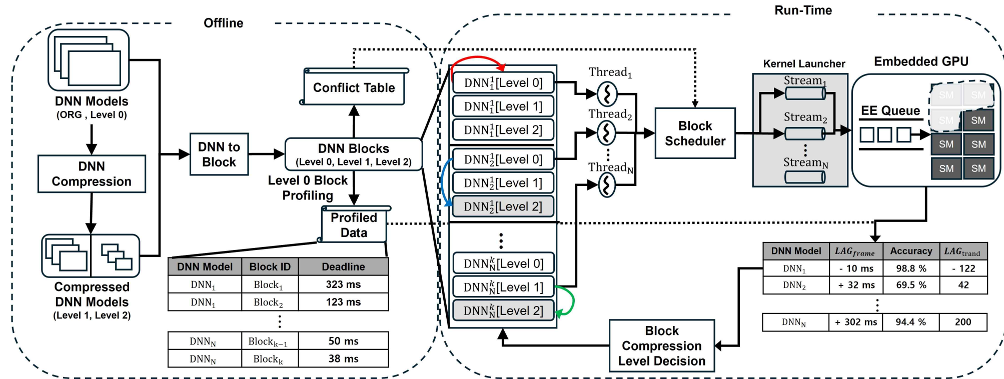
$$EMA_{short}(f) = \alpha \times x_f + (1-\alpha) \times EMA_{short}(f-1), \qquad 0.5 < \alpha < 0.99 \quad (5)$$

$$EMA_{long}(f) = \beta \times x_f + (1-\beta) \times EMA_{long}(f-1), \qquad 0.01 < \beta < 0.49 \quad (6)$$

여기서 장/단기 추세를 동시에 포착하기 위해 서로 독립적인 두 평활화 계수 α와 β를 도입하였다는 점에 유의해야 한다. 이러한 두 지표를 기반으로, 프레임 수준 실행 지연의 추세를 반영하는 새로운 지표인 $LAG_{trend}(f)$를 정의한다.

$$LAG_{trend}(f) = EMA_{short}(f) - EMA_{long}(f) \qquad (7)$$

식(7)에서 정의한 바와 같이, $LAG_{trend}(f)$ 는 $EMA_{short}(f)$와 $EMA_{long}(f)$의 차이를 의미한다. $EMA_{long}(f)$은 장기적인 실행 추세를 추적하는 특성상 그 값이 평균과 유사한 상수와 거의 비슷한 형태로 유지되며, 이러한 특성 때문에 $LAG_{trend}(f)$는 상황에 따라 $EMA_{short}(f)$와 유사한 패턴을 보일 수 있다. 그러나 $EMA_{short}(f)$은 순간적인 변동에 빠르게 반응하는 장점이 있지만, 장기적인 경향을 반영하기에는 한계가 있다. 앞서 언급했듯이, 산술 평균을 기준선으로 사용하는 것도 가능하나, 이는 전체 실행시간 분포를 정적인 값으로 압축하기 때문에 시간 변화에 따른 추세를 전혀 반영하지 못한다는 문제가 있다. 즉, 지연 시간의 장/단기 변화를 균형 있게 반영할 적절한 기준 지표가 부재한 상황이다. 따라서 본 논문에서는 $EMA_{long}(f)$을 장기 기준선으로 채택하였다. 이 값은 일시적인 변동을 일정 부분 반영하면서도 비교적 완만한



[그림 4-3] 다중 DNN 추론을 위해 제안된 프레임워크의 동작 흐름도

응답 특성을 유지하므로, 지연 시간에 민감한 시스템에서 안정적이고 신뢰할 수 있는 기준으로 활용하기에 적합하다.

$LAG_{\mathrm{trend}}(f)$ 값이 양수이면 최근 실행시간이 장기적 평균을 초과함을 나타내며, 이는 지연이 누적되고 성능이 저하될 가능성을 시사한다. 반대로 음수 값은 최근 실행시간이 장기적 평균보다 낮음을 의미하며, 지연이 완화되고 성능이 회복되는 경향으로 해석할 수 있다. 또한, $LAG_{\mathrm{trend}}(f)$의 크기는 각 블록의 압축 수준을 조정하는 기준으로 사용된다. 구체적으로, $LAG_{\mathrm{trend}}(f)$의 값이 0에 가까우면 블록은 원래 상태(Level 0)를 유지한다. 반대로 일정 임계값을 초과하면 Level 1 블록으로 전환되며, 값이 더욱 커지면 Level 1보다 높은 압축 비율을 적용하는 Level 2 블록으로 대체된다. 반대로 음수 구간에서 절댓값이 커지는 경우는 블록의 압축 수준을 완화하거나 원본 블록인 Level 0으로 복원하게 된다.

이에 따라 본 논문에서 제안하는 프레임워크는 $LAG_{\mathrm{trend}}(f)$를 실행 지연 추세의 방향(부호)뿐만 아니라 그 강도(크기)까지 반영할 수 있도록 설계하였다. 이러한 메커니즘은 [그림 4-3]에 제시된 전체 프레임워크의 핵심 구성 요소로서, 지연에 민감한 시스템에서 안정성과 효율성을 동시에 달성하기 위한 동적 스케줄링 및 압축 제어의 핵심 지표로 기능한다.

제 2 절 엣지 디바이스에서 다중 DNN 추론을 위한 LAG 기반 블록

1) 엣지 디바이스에서 다중 DNN 추론을 위한 LAG 기반 블록 단위 스케줄링 및 동적 압축 프레임워크

[그림 4-3]은 제안하는 프레임워크의 전체 동작 과정을 보여준다. 프레임워크는 Offline 단계와 Run-Time 단계의 두 부분으로 구성된다. 오프라인 단계([그림 4-3]의 왼쪽 점선 영역)는 DNN Compression, DNN to Block, Level 0 Block Profiling의 세 단계로 이루어진다. Run-Time 단계([그림 4-3]의 오른쪽 점선 영역)는 Block Scheduler와 Block Compression Level Decision으로 구성된다.

Offline 단계에서는 시스템에 탑재될 여러 개의 원본 DNN 모델(Level 0)에 대해 사전 준비 작업이 수행된다. 먼저, DNN Compression 단계에서는 각 모델을 대상으로 파라미터의 크기를 축소한 두 가지 수준의 압축 모델(Level 1, Level 2)이 생성된다. 다음으로 DNN to Block 단계에서는 각 모델을 기능 단위(블록)로 분해하여 이 과정에서 각 모델에 대해 서로 다른 압축 수준(Level 0, Level 1, Level 2)으로 구성된 블록들이 생성된다. 블록은 DNN_a^b로 표기하며, a는 DNN 모델의 인덱스를, b는 해당 모델 내 블록의 인덱스를 의미한다. Level 0 블록은 압축되지 않은 원본 구성을 유지하며, Level 1과 Level 2 블록은 점진적으로 압축이 적용된 수준의 블록을 의미한다. Conflict Table은 서로 다른 두 블록이 병렬로 실행 가능한지를 기록한 표이며, Run-Time 단계에서 Block Scheduler가 실행 순서를 결정할 때 활용된다. Level 0 Block Profiling 단계에서는 각 Level 0 블록의 실제 실행시간의 분포를 수집하고, 이에 따라 해당 블록의 데드라인을 설정한다. 이렇게 프로파일링 된 데이터와 여러 압축 수준의 블록들은 이후 Run-Time 단계에서 LAG_{trend} 값에 따라 유연하게 선택될 수 있는 실행 단위로 사용된다.

Run-Time 단계에서는 제안하는 프레임워크의 핵심 기능이 수행된다. 전체 시스템은 하나의 프로세스 내에 다수의 실행 스레드 $(\mathrm{Thread}_1, \mathrm{Thread}_2, \cdots, \mathrm{Thread}_N)$, CUDA 스트림$(\mathrm{Stream}_1, \mathrm{Stream}_2, \cdots, \mathrm{Stream}_N)$

Algorithm 1 Block-Level Scheduling Framework for Multi DNNs

Require: F: Period of block compression level decision
F_k^{idx}: Frame index of M_k

1: **function** BLOCK_LEVEL_EXECUTE($\mathcal{M}$, $\mathcal{D}$, $\mathcal{H}$)
2: **for all** $M_k \in \mathcal{M}$ **do**
3: $LAG_{\text{frame},k} \leftarrow 0$
4: **for all** $\text{DNN}_k^i \in M_k$ **do**
5: BLOCK_SCHEDULER(DNN_k^i)
6: $E_k^i \leftarrow$ execution time of DNN_k^i in M_k
7: $\mathcal{D}_k^i \leftarrow$ deadline of DNN_k^i in M_k
8: $LAG_k^i \leftarrow E_k^i - \mathcal{D}_k^i$
9: $\mathcal{H}_k^i.\text{append}(LAG_k^i)$
10: $LAG_{\text{frame},k} \leftarrow LAG_{\text{frame},k} + LAG_k^i$
11: **end for**
12: $EMA_{\text{short},k} \leftarrow \alpha \cdot LAG_{\text{frame},k} + (1-\alpha) \cdot EMA_{\text{short},k}$
13: $EMA_{\text{long},k} \leftarrow \beta \cdot LAG_{\text{frame},k} + (1-\beta) \cdot EMA_{\text{long},k}$
14: $LAG_{\text{trend},k} \leftarrow EMA_{\text{short},k} - EMA_{\text{long},k}$
15: ACC_k = Accuracy result of M_k
16: **if** $F_k^{idx} \bmod F = 0$ **then**
17: BLOCK_COMP_LEVEL_DECISION($M_k, \mathcal{H}_k, LAG_{\text{trend},k}, ACC_k$)
18: **end if**
19: **end for**
20: **end function**

[그림 4-4] 다중 DNN 추론을 위한 제안된 프레임워크의 알고리즘

을 구성하며, 각 DNN 모델은 전용 스레드와 CUDA 스트림에 할당된다. 각 스레드는 기존 DNN의 순전파 구조에 따라 블록 단위로 모델을 실행하는 식이 아닌, Conflict Table을 기반으로 Block Scheduler에 의해 실행 순서를 동적으로 조정하여 전체 시스템의 처리량을 향상시킨다.

하나의 DNN에 속한 모든 블록의 실행이 완료되면 정확도, 블록별 LAG의 총합, 그리고 EMA_{short}, EMA_{long}에 의해 LAG_{trend}가 계산된다. 이 값은 Block Compression Level Decision 모듈로 전달되며, 해당 모듈은 다음 프레임에서 실행할 블록의 수준(Level 0~2)을 결정한다. 이러한 구조를 통해 제안하는 프레임워크는 전체 시스템 지연과 정확도 간의 균형을 유지하면서, 블록 단위 실행을 유연하게 제어할 수 있다.

[그림 4-4]는 [그림 4-3]에 제시된 아키텍처의 전체 동작에 해당하는 제안된 프레임워크의 의사코드를 나타낸다. 이후 내용을 이해하기 쉽도록, 본

HANSUNG UNIVERSITY

Symbol	Definition
k	DNN model index; $k \in \{1, \cdots K\}$
i	Block index in model M_k; $i \in \{1, \cdots m_k\}$
W	length of LAG history
M_k	k-th DNN Model; $M_k \in \{\mathrm{DNN}_k^1, \mathrm{DNN}_k^2, \cdots \mathrm{DNN}_k^{m_k}\}$ where DNN_k^i denotes the i-th block of M_k
D_k^i	Deadline of block DNN_k^i in model M_k; $D_k = \{D_k^1, \cdots, D_k^{m_k}\}$
H_k^i	LAG history of block DNN_k^i in model M_k; $H_k^i = [LAG_k^i(1), \cdots, LAG_k^i(W)]$
H_k	Set of histories in M_k; $H_k = \{H_k^i, \cdots, H_k^{m_k}\}$
$LAG_{\mathrm{frame,k}}$	Total LAG of all blocks in a frame for M_k
$EMA_{\mathrm{short,k}}$, $EMA_{\mathrm{long,k}}$	Short/long EMA of $LAG_{frame,k}$ for M_k
$LAG_{\mathrm{trend,k}}$	LAG_{trend} for M_k

[표 4-1] 표기법

논문에서 자주 사용되는 표기들은 [표 4-1]에 정리하였다. 알고리즘은 각 DNN 모델을 구성하는 블록들을 순차적으로 실행하고, 블록별 LAG를 계산하며, 이를 히스토리($\mathcal{H}$)에 저장한다. 또한, 프레임 단위의 LAG_{trend}를 분석하고, 그 결과에 따라 블록 교체 여부를 결정한다. $\mathcal{M} = \{M_1, M_2, \cdots, M_k\}$를 DNN 모델들의 집합이라 가정했을 때, 각 모델 $M_k = \{\mathrm{DNN}_k^1, \mathrm{DNN}_k^2 \cdots, \mathrm{DNN}_k^{m_k}\}$는 실행 가능한 블록들로 순서가 있는 시퀀스 집합으로 정의된다. 각 모델은 전용 스레드와 CUDA 스트림에 할당된다.

제안된 스케줄링 Block_Scheduler(DNN_i^k)은 [그림 4-4]의 라인 5에 의해 수행되며, GPU를 동시에 점유하고 있는 다른 블록들과의 자원 충돌을 고려하여 블록 DNN_i^k의 실행 시작 시점을 동적으로 결정한다.

블록 DNN_i^k의 실행이 완료되면, 해당 블록의 실제 실행시간 E_i^k이 측정되며, 사전에 정의된 데드라인 D_i^k와 비교하여 LAG_i^k를 계산한다([그림 4-4],

라인 6~8). 계산된 LAG_i^k는 누적값을 저장하는 히스토리 H_i^k에 추가된다([그림 4-4], 라인 9). 이러한 절차는 $\mathrm{DNN}_i^k \in M_k$에 포함된 모든 블록에 대해 반복되며, 블록별 LAG_i^k 값들은 프레임 단위 지표인 $LAG_{\mathrm{frame,k}}$로 집계된다([그림 4-4], 라인 10).

$LAG_{\mathrm{frame,k}}$ 값을 바탕으로 단기 및 장기 지수 이동 평균인 $EMA_{\mathrm{short,k}}$와 $EMA_{\mathrm{long,k}}$가 각각 갱신된다. 이후 두 값의 차이를 통해 $LAG_{\mathrm{trend,k}}$가 계산되며, 이는 모델 M_k의 성능을 반영한다([그림 4-4], 라인 12~14). 중요한 점은 이러한 값들이 모델마다 독립적으로 관리된다는 것이다. 즉, $LAG_{\mathrm{frame,k}}$, $EMA_{\mathrm{short,k}}$, $EMA_{\mathrm{long,k}}$, $LAG_{\mathrm{trend,k}}$는 전역적으로 공유되지 않고 모델별로 개별적으로 갱신된다.

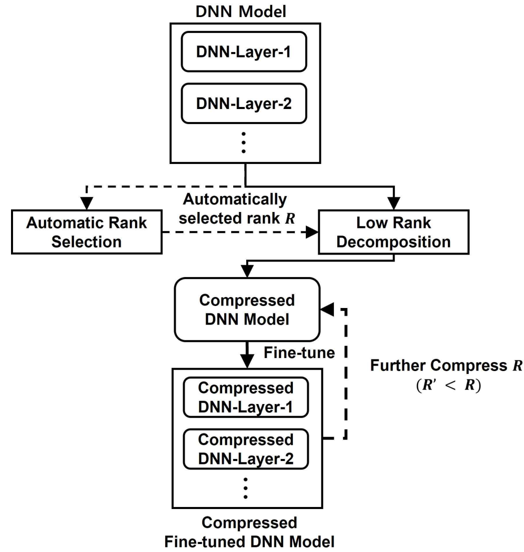
매 F 프레임마다([그림 4-4], 라인 14) Block_Comp_Level_Decision 함수가 호출된다. 이 함수는 모델 M_k의 정확도 지표와 $LAG_{\mathrm{trend,k}}$를 함께 고려하여, 다음 프레임에서 각 블록이 사용할 압축 수준(Level 0 - 2)을 결정한다. 이와 같은 절차를 통해 제안하는 프레임워크는 단순한 순차 실행을 넘어, 런타임 편차를 민감하게 감지하고 지연이 누적되는 블록만을 선별적으로 교체함으로써 전체 시스템 지연과 정확도 간의 균형을 유지한다. 각 단계에서 수행되는 주요 작업은 이어지는 항에서 순차적으로 상세히 기술한다.

2) 점진적 모델 경량화 기법

Layer Type	Kernel Size	Decomposition
Conv2d	(1, 1)	SVD
Conv2d	(n, n) (n 〉 1)	SVD
Linear	-	Tucker (or CP3, CP4)

[표 4-2] 점진적 모델 압축 과정에서 레이어 유형별 분해 기법

Offline 단계에서 수행되는 DNN Compression 단계를 통해 경량화 모델을 생성하는 과정은 본 연구에서 채택한 MUSCO(Multi-Stage Compression of Neural Networks)(Gusak, J., et al, 2019) 기법에 기반하여 수행된다. [그

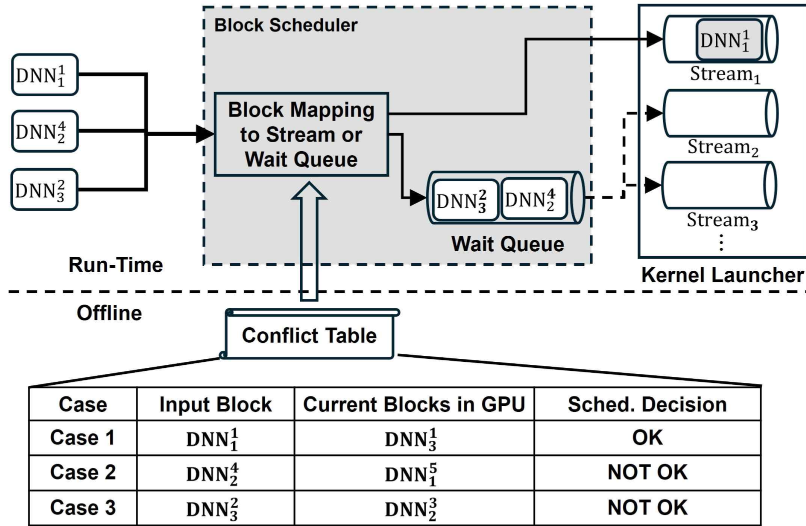


[그림 4-5] 자동 랭크 선택 기반 점진적 DNN 모델 압축 구조 개요

림 4-5]는 전체 압축 과정을 단계별로 시각화한 것이다.

먼저 Automatic Rank Selection 단계에서는 Bayesian 기반의 EVBMF(Nakajima, S., et al, 2013) 방법 또는 고정 압축 비율 방식 중 하나를 사용하여, 각 층의 가중치 텐서에 대해 압축에 사용될 랭크 R을 자동으로 결정한다. 다음으로 Low Rank Decomposition 단계에서는 자동으로 선택된 R을 이용하여 경량화가 수행된다. 이때 레이어의 유형에 따라 서로 다른 분해 기법이 적용되며, 이에 대한 상세 내용은 [표 4-2]에 제시되어 있다. 대표적인 예로는 Singular Value Decomposition(SVD)(Denton, E. L., et al, 2014)과 Tucker-2(Kim, Y. D., et al, 2015) 분해 등이 사용된다.

이러한 과정을 통해 원본 모델의 아키텍처를 유지하면서도 파라미터 수와 연산량(FLOPs)을 감소시킨 압축된 DNN 모델이 생성된다. 압축으로 인해 발생하는 성능 저하를 회복하기 위해 Fine-tune 단계를 수행하며, 필요한 경우 더 낮은 랭크 $R'(R' < R)$를 적용하여 추가 압축을 진행할 수도 있다. 최종적으로, 압축과 미세조정을 반복한 결과 세 가지 수준(Level 0~2)의 DNN 모델이 생성되며, Level 0은 원본 모델을, Level 1과 Level 2는 점진적으로 더 경량화된 모델을 의미한다.



Case	Input Block	Current Blocks in GPU	Sched. Decision
Case 1	DNN_1^1	DNN_3^1	OK
Case 2	DNN_2^4	DNN_1^5	NOT OK
Case 3	DNN_3^2	DNN_2^3	NOT OK

[그림 4-6] Conflict Table을 활용한 블록 스케줄러의 동작 흐름도

3) 블록 스케줄러 기법

서로 다른 DNN 모델의 블록들은 서로 독립적인 연산 흐름을 가지므로 병렬 실행이 가능하다. 그러나 실제 임베디드 GPU 환경에서는 병렬 실행이 항상 최적의 성능을 보장하지 않는다. 특히, 동일한 Streaming multiprocessors(SMs)을 동시에 점유하려는 블록 간에는 자원 충돌이 발생할 수 있으며, 일부 블록의 경우 병렬 실행보다 순차 실행이 더 짧은 실행시간을 보이는 현상이 관찰되었다. 이러한 결과는 본 논문의 5장 2절 2)의 가)에서 확인할 수 있다. [그림 4-6]은 블록 스케줄러의 전체 구조와 동작 과정을 나타내며 이는 [그림 4-3]의 Run-Time 단계에서 Block Scheduler를 의미하며 각 블록의 실행 시점을 동적으로 제어하고 CUDA 스트림 할당 여부를 결정하는 역할을 수행한다.

블록 스케줄러의 세부적인 동작 방식은 다음과 같다. 먼저 대기 중인 블록이 스케줄러에 도착하면, 먼저 GPU를 점유하여 현재 실행 중인 다른 모델의 블록들과 자원 충돌이 발생할 가능성이 있는지를 판별한다. 이를 판별하기 위해, 스케줄러는 오프라인 단계에서 사전에 정의된 Conflict Table을 참조한다. Conflict Table은 실행하려는 블록([그림 4-6]의 Conflict Table의 Input Block)과 현재 GPU를 점유 중인 블록 ([그림 4-6]의 Conflict Table의

HANSUNG UNIVERSITY

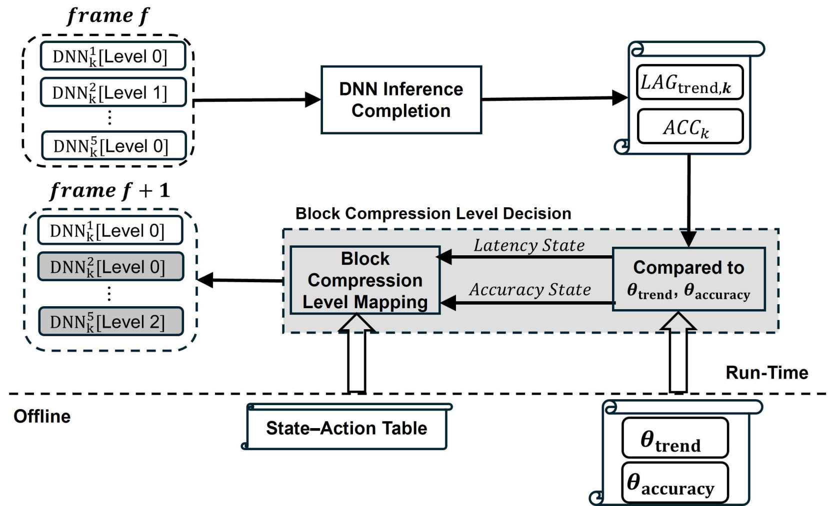
Algorithm 2 Block Scheduler: Conflict-Aware Block Execution

```
Require: block: Target block to be executed
 1: function BLOCK_SCHEDULER(block)
 2:     REQUEST_GPU_ACCESS(block)
 3:     Launch to the execution queue
 4:     RELEASE_GPU_ACCESS(block)
 5: end function
 6: function REQUEST_GPU_ACCESS(block)
 7:     lock(m)
 8:     while ConflictTable[block] ∩ RunningBlocks ≠ ∅ do
 9:         cv.wait(m)
10:     end while
11:     RunningBlocks.append(block)
12:     unlock(m)
13: end function
14: function RELEASE_GPU_ACCESS(block)
15:     lock(m)
16:     RunningBlocks.remove(block)
17:     unlock(m)
18:     cv.notify_all()
19: end function
```

[그림 4-7] 블록 스케줄러 알고리즘

Current Blocks in GPU) 쌍이 병렬로 실행 가능한지에 대한 여부가 기록되어 있으며, 스트림 할당 가능 여부가 명시적으로 정의되어 있다. Conflict Table은 단순한 논리 규칙에 기반을 두어 작성된 것이 아니라, 오프라인 단계에서 수행된 정량적 실험 분석을 통해 구성된 것이다. 구체적으로, 각 DNN 모델의 블록 쌍에 대해 여러 CUDA 스트림을 사용하여 병렬 실행한 경우와 단일 스트림을 사용하여 순차 실행한 경우를 비교한다. 만약 순차 실행이 더 빠른 경우 해당 블록 쌍의 병렬 실행을 제한하는 규칙을 Conflict Table에 반영하였다. 이후 스케줄러는 블록 매핑 여부에 따라 두 가지 방식으로 동작한다. 스케줄러는 매핑 결과에 따라 다음과 같이 동작한다.

매핑이 가능한 경우(OK), 블록을 즉시 CUDA 스트림에 할당하여 Kernel Launcher를 통해 GPU에서 실행하며, 매핑이 불가능한 경우(NOT OK)에는 블록을 대기 큐(Wait Queue)로 이동시켜 현재 실행 중인 블록이 GPU를 해제할 때까지 대기시킨 후, 대기가 끝나는 즉시 스트림에 할당하여 실행한다. 이러한 메커니즘은 블록 간 자원 충돌로 인해 발생하는 GPU 자원 낭비를 최



[그림 4-8] 블록 압축 레벨 결정 기법의 동작 흐름도

소화하고, 전체 시스템 처리량 향상에 이바지한다. [그림 4-7]은 블록 스케줄러의 핵심 동작을 나타내는 의사 코드이다. Block_Scheduler(*block*) 함수는 실행 대기 중인 블록을 입력으로 받아, 해당 블록을 실행하기 전후로 Request_GPU_Access와 Release_GPU_Access를 통해 GPU 자원 접근 권한을 요청하고 해제하는 절차를 포함하는 두 가지 주요 작업을 수행한다.

Request_GPU_Access 함수가 수행되면 먼저 스케줄러가 뮤텍스 *m*을 획득하여, GPU를 점유하고 있는 블록들의 집합인 RunningBlocks 라는 공유 자원을 보호한다. 이후 ConflictTable[*block*]을 확인하여 대기 중인 블록이 현재 실행 중인 블록들과 충돌이 발생하는지 검사한다. 만약 충돌이 존재하면, 스케줄러는 *cv*.wait(*m*)을 호출하여 충돌하는 블록들이 실행을 완료할 때까지 해당 스레드를 일시 중단한다. 충돌이 모두 해소되면, 해당 블록을 RunningBlocks에 추가한 뒤 뮤텍스 *m*을 해제하여 GPU에서 블록이 실행될 수 있도록 한다. 실행이 완료되면 Release_GPU_Access가 호출되어 해당 블록을 RunningBlocks에서 제거하고, *cv*.notify_all()을 호출하여 대기 중인 스레드들이 조건을 다시 검사할 수 있도록 신호를 전달한다.

4) 블록 압축 레벨 결정 기법

Latency State	Accuracy State	Current Block Level	Action
Good	Good	None	None
	Warning	Level 2	Select Level 1
		Level 1	Select Level 0
	Critical	Level 2	Select Level 1
		Level 1	Select Level 0
Warning	Good	Level 0	Select Level 2
		Level 1	Select Level 2
	Warning	Level 0	Select Level 1
		Level 1	Select Level 2
	Critical	Level 0	Select Level 1
		Level 1	Select Level 1
Critical	Good	Level 0	Select Level 2
		Level 1	Select Level 2
	Warning	Level 0	Select Level 2
		Level 1	Select Level 2
	Critical	Level 0	Select Level 1
		Level 1	Select Level 1

[표 4-3] 블록 압축 레벨 결정 기법의 State-Action Table

[그림 4-8]은 블록 압축 레벨 결정 기법의 전체 동작 구조를 보여준다. 이 구성 요소는 앞서 [그림 4-3]의 Run-Time 단계의 Block Compression Level Decision에 해당하며, 현재 모델 M_k의 실행을 통해 얻은 결과로부터 $LAG_{\mathrm{trend,k}}$및 ACC_{k} 정보를 수집하고, 이를 바탕으로 다음 프레임에서 모델 M_k의 각 블록이 사용할 압축 수준을 결정하는 역할을 수행한다. 현재 프레임이 실행되는 동안, 모델 M_k의 모든 블록이 실행되면서, 각 블록의 LAG 값이 측정된다. 이러한 값들은 누적되어 프레임 단위 LAG인 $LAG_{\mathrm{frame,k}}$가 계산되며, 이 값은 단기 및 장기 지수 이동 평균($EMA_{\mathrm{short,k}}$, $EMA_{\mathrm{long,k}}$)을 갱신하는 데 활용된다. 이후, 두 값의 차이를 통해 모델 M_k의 지연 추세를 의미하는 $LAG_{\mathrm{trend,k}}$가 산출된다. 모델 M_k의 모든 블록 실행이 완료되면, ($LAG_{\mathrm{trend,k}}$, ACC_{k}) 쌍이 생성된다.

[그림 4-8]의 Block Compression Level Decision 모듈은 이 두 지표를 기반으로 각 모델의 *Latency State*와 *Accuracy State*를 계산한다. 각 상태는 미리 정의된 임계값(θ_{trend}, $\theta_{\mathrm{accuracy}}$)에 따라 *GOOD*, *WARNING*,

Algorithm 3 Block Compression Level Decision: Based on LAG_{trend} and Accuracy

```
1:  function BLOCK_COMP_LEVEL_DECISION(M_k, H_k, LAG_trend,k, ACC_k)
2:      if LAG_trend,k > θ_trend and ACC_k < θ_accuracy then
3:          for all DNN_k^i ∈ M_k do
4:              N_total ← |H_k^i|
5:              N_pos ← |{ℓ ∈ H_k^i | ℓ > 0}|
6:              if N_pos / N_total > θ_ratio then
7:                  next level of DNN_k^i ← Level 1 or Level 2 from Table 3
8:              else
9:                  next level of DNN_k^i ← Level 0
10:             end if
11:         end for
12:     else
13:         for DNN_k^i ∈ M_k do
14:             next level of DNN_k^i ← current level of DNN_k^i
15:         end for
16:     end if
17: end function
```

[그림 4-9] LAG_{trend}, Accuracy 기반의 블록 압축 레벨 결정 알고리즘

*CRITICAL*의 세 개 중 하나로 분류된다. 예를 들어, $LAG_{\mathrm{trend,k}}$가 θ_{trend}보다 지속적으로 낮게 유지되는 경우, *Latency State*는 *GOOD*로 분류된다. 반대로, 최근 프레임에서 $LAG_{\mathrm{trend,k}}$가 급격히 증가한 경우에는 *CRITICAL* 상태로 분류된다. 이 두 극단 값들 사이에 위치하는 중간값들은 *WARNING*로 분류된다. Accuracy State 또한 같은 기준을 따르며, 정확도가 일정 수준 이상 감소할 경우 *WARNING* 또는 *CRITICAL*로 분류된다.

[그림 4-8]의 Block Compression Level Mapping 단계에서는 도출된 상태 쌍 (*Latency State, Accuracy State*)을 미리 정의된 State-Action Table ([표 4-3])을 이용하여 다음 프레임에서 사용할 적절한 블록 압축 수준(Level 0~2)으로 매핑한다. 이러한 구조를 통해 시스템은 실행시간과 정확도 사이의 균형을 유지하면서도 블록 단위의 세밀한 제어가 가능하다. 특히 실행 과정에서 지연이 과도하게 누적될 경우, 시스템은 더 경량화된 블록 수준으로 전환하여 처리 속도를 향상시킬 수 있다. 반대로 정확도 저하가 감지되면 원본 블록 수준으로 복원함으로써 동적 제어 전략을 효과적으로 구현할 수 있다.

[그림 4-9]는 각 블록의 압축 수준을 동적으로 조정하는 블록 압축 레벨 결정 기법의 핵심 동작을 정의한 의사코드이다. Block_Comp_Level_Decision

함수는 현재 실행 중인 DNN 모델 M_k, 모델 M_k에 포함된 각 블록 DNN_i^k의 최근 W개 LAG 값을 시계열 형태로 저장한 히스토리 H_k, 단기 및 장기 지수 이동 평균의 차이로 계산된 모델 M_k의 지연 추세 지표 $LAG_{\mathrm{trend,k}}$, 그리고 일정 프레임 간격으로 측정된 모델 M_k의 정확도 ACC_{k}를 입력으로 한다 이 함수는 매 프레임마다 호출되는 것이 아니라, 블록 수준 조정의 빈도를 제한하여 시스템 안정성을 확보하기 위해 F 프레임마다 주기적으로 호출된다.

먼저, $LAG_{\mathrm{trend,k}}$와 ACC_{k}를 각각 θ_{trend}및 $\theta_{\mathrm{accuracy}}$와 비교하여, 시스템이 높은 지연 시간과 낮은 정확도를 동시에 기록하고 있는지를 판별한다. 두 조건이 모두 충족되는 경우에만 블록 수준 전환이 수행된다. 여기서 θ_{trend}와 $\theta_{\mathrm{accuracy}}$는 DNN 모델별로 설정되는 임계값([그림 4-9]의 라인 2)이며, 비율 임계값 θ_{ratio}는 0.5로 설정하였다. 이들 임계값의 상세한 도출 과정은 다음 항에서 설명한다. 이러한 세 가지 임계값은 일시적인 스파이크로 인해 불필요한 압축이 발생하는 것을 방지하고, 지연이 지속해서 누적되는 상황에서만 압축이 활성화되도록 함으로써 시스템 안정성을 유지한다. 각 블록 DNN_i^k마다, H_i^k에 저장된 LAG 기록의 총 개수 N_{total}은 윈도우 크기 W와 동일하며, 이 중 LAG 값이 양수인 항목의 개수를 N_{pos}라고 정의한다. 만약 비율 $N_{\mathrm{total}}/N_{\mathrm{pos}}$이 임계값 θ_{ratio}를 초과하면, 해당 블록 DNN_i^k의 압축 수준을 Level 1 혹은 Level 2로 전환한다. 이러한 방식은 지연 빈도가 높은 블록만을 선별하여 교체함으로써, 지연을 완화하면서도 정확도 손실을 최소화하도록 한다. 조건을 충족하지 않는 블록은 Level 0 상태를 유지한다 ([그림 4-9]의 라인 3~11). 초기 조건이 충족되지 않는 경우(즉, 블록 수준의 전환이 불필요한 경우), 모든 블록은 이전 프레임에서 할당된 압축 수준을 그대로 유지하며, 추가적인 변경은 발생하지 않는다([그림 4-9]의 라인 12~16).

결과적으로 블록 압축 레벨 결정 기법은 지연과 정확도 지표를 함께 고려하여 주기적으로 블록 수준 실행 구성을 재조정하며, 이를 통해 시스템 효율성과 모델 정확도 간의 동적 균형을 유지할 수 있게 된다.

Model	Metric	WARNING 임계값	CRITICAL 임계값
VGG16	θ_{trend}	50	20
	θ_{accuracy}	0.85	0.6
Faster-RCNN	θ_{trend}	80	20
	θ_{accuracy}	0.6	0.25
SiamRPN++	θ_{trend}	120	75
	θ_{accuracy}	0.69	0.48

[표 4-4] θ_{trend}와 θ_{accuracy}에 대한 모델별 임계값

5) 임계값 도출 및 상태 정의

임계값인 θ_{trend}와 θ_{accuracy}는 DNN 모델의 실행 로그로부터 수집된 LAG_{trend}와 정확도 데이터를 기반으로 도출된다. 수집된 데이터는 프레임 단위로 기록되며, 고정된 구간으로 분할하여 히스토그램을 구성한다. 이 히스토그램에서 가장 빈도가 높은 구간을 찾고, 해당 구간 내에서의 최댓값을 기준으로 θ_{trend} 및 θ_{accuracy}를 결정하는 데 사용한다. 이렇게 정의된 임계값은 시스템 상태가 *GOOD*에서 *WARNING*으로 전환되는 기준으로 활용된다. 반대로, 빈도가 낮은 구간들은 통계적으로 유의미하지 않은 것으로 간주하여 임계값 산출 과정에서 제외된다. 이러한 방식은 일시적인 이상치나 단기적인 지연의 영향을 최소화하며, 도출된 임계값이 시스템의 전형적이고 안정적인 실행 특성을 반영하도록 보장한다. *CRITICAL* 상태로 전환되는 임계값은 LAG_{trend}와 정확도 값이 평균값의 약 3분의 2 이하로 떨어지는 경우로 정의된다. 이러한 설정은 불필요한 상태 전환을 방지하고, 지연이 지속해서 누적되거나 정확도가 크게 저하되는 상황에서만 시스템이 *CRITICAL* 상태로 전환하도록 설계하였다. 결과적으로 제안된 프레임워크는 정상적인 조건에서는 안정적인 동작을 유지하며, 명확한 성능 저하가 나타날 때만 Level 1~2로 압축 수준으로 전환되도록 한다. 모든 임계값은 모델별로 독립적으로 산출되며, 각 모델의 실행 특성과 분포 패턴을 반영한다.

비율 임계값 θ_{ratio} = 0.5는 엔지니어링 기반의 실험적 접근을 통해 설정되었다. [그림 4-9]의 라인 6에서 볼 수 있듯이, *W*개의 저장된 최근 *LAG* 값

중, 지연을 의미하는 양의 값의 비율이 절반을 초과하는 경우에만 지속해서 지연되는 블록으로 판단하여 경량화 블록으로 교체하게 된다. 이러한 방식은 모든 블록을 불필요하게 압축하는 상황을 방지하고, 양수인 LAG의 비율이 높은 블록만 선별적으로 교체하도록 하여 시스템 안정성과 반응성의 균형을 유지한다. [표 4-4]는 앞서 설정한 기준에 따라 본 논문에서 모델별로 적용한 임계값을 정의한다.

제 5 장 실험 결과 및 분석

본 장에서는 3장에서 제안한 다양한 모델 경량화 기법들을 실제 모델들에 적용하여 그 효과를 평가하고, 4장에서 제안한 블록 레벨 스케줄링 기반 다중 DNN 실행 최적화 기법의 성능을 검증한다. 실험은 분류, 검출, 추적 모델을 대상으로 다양한 실험 시나리오를 구성하여 진행하였다.

제 1 절 모델 경량화 및 추론 최적화 기법에 따른 실험 결과

1) 실험 환경 및 구성

	Classification	Description
HW	CPU	16-core, 64MB L3 cache, 3.9Ghz
	GPU	4 x NVIDIA RTX A6000, 336 Tensor Cores, 10,752 CUDA Cores, 48 GB Memory, 309.7 TFLOPS
	Memory	4 x 64 GB DDR4 PC4
	Storage	1 x SSG 1.92 TBG 2.5'' SATA
SW	Kernel Ver.	Linux 5.15.0
	SW Package	MPI Horovod, NVIDIA GPU Monitoring SW
	CUDA Ver.	CUDA v11.6

[표 5-1] 실험에 사용된 서버 시스템 환경 사양

앞서 언급한 기존 정적 경량화 기법을 임베디드 GPU 환경에 적용했을 때, 메모리 정렬 구조로 인해 Quantization 효과가 충분히 반영되지 않고, Pruning 기법 또한 희소 연산이 실제로 제거되지 않아 지연 시간 감소의 효과가 나타나지 않는다. 따라서 본 절에서는 각 경량화 기법이 모델의 파라미터 수와 정확도에 미치는 영향을 비교하기 위하여 실험은 서버 시스템 환경에서 수행하였다. 실험에 사용된 하드웨어 및 소프트웨어 사양은 [표 5-1]과 같다.

본 논문에서는 분류, 검출, 추적이라는 서로 다른 작업에서 경량화 기법의

HANSUNG UNIVERSITY

효과를 종합적으로 분석하기 위하여, 분류 모델 ResNet152(He, K., et al, 2016)와 VGG16(Simonyan, K., et al, 2014), 검출 모델 Faster R-CNN(Ren, S., et al, 2015), 추적 모델 SiamRPN++(Li, B., et al, 2019)을 실험 대상으로 선정하였다. ResNet152와 VGG16은 서로 다른 구조적 특성을 가지는 대표적 분류 모델로, 경량화 기법 적용에 따른 변화 양상을 비교하는 데 적합하다. Faster R-CNN은 영역 제안 기반의 2-stage 기반 검출 모델이며, SiamRPN++은 템플릿-검색 구조의 객체 추적 모델로 경량화에 따른 특징 변화가 성능에 직접 반영되는 특성을 지닌다. 데이터셋은 모델 종류에 따라 각각 CIFAR-10, PASCAL VOC 2007, TrackingNet 데이터셋을 사용하였다. 또한, 모델 유형에 따라 서로 다른 정확도 지표를 사용하여 경량화 효과를 비교하였다. 분류 모델은 Top-1과 Top-5 정확도로, 검출 모델은 mAP(mean Average Precision)(Everingham, M. et al, 2010), 추적 모델은 IoU(Intersection over Union)(Wu, Y., et al, 2013)로 평가를 진행하였다. 아울러 모든 모델에 대해 파라미터 감소율, 모델 크기, 총 파라미터 개수도 함께 비교하여 경량화 기법의 구조적 축소 효과를 정량적으로 평가하였다.

2) 모델 경량화 기법 적용 결과

Pruning은 합성곱 레이어와 완전 연결 레이어에 대해 사용자가 지정한 Pruning 비율을 적용하여, 경량화 정도에 따른 성능 변화를 관찰하였다. 각 레이어는 가중치 분포를 기반으로 임계값을 산정하고, 설정된 비율만큼 임계값 이하의 가중치를 제거하는 방식으로 Pruning을 수행하였다. Quantization은 가중치를 저정밀도로 표현하여 모델 크기를 줄이는 방식으로 수행하였으며, 합성곱 레이어와 완전 연결 레이어에 대해 사용자가 지정한 비트 폭을 적용하였다. 본 연구에서는 주로 5bit와 8bit를 혼합하여 사용하였다. 또한 Pruning과 Quantization을 결합하여 적용한 경량화 실험도 수행하였다. 이때 Pruning 비율과 Quantization 비트 수를 다양한 조합으로 구성하여, 단독 적용과 비교했을 때 모델 크기 감소율, 파라미터 수 변화, 정확도 손실 정도를 분석하였다.

가) 객체 분류 모델 경량화 적용 결과

경량화 방식	경량화 세팅	TOP-1/ TOP-5	압축률	파라미터 크기 (MB)	파라미터 개수 (M)
Original		85.28 / 98.84	X	512.32	134
Pruning	Conv2d: 0.5 Linear: 0.5	83.64 / 98.16	50	256.16	67
	Conv2d: 0.9 Linear: 0.9	77.30 / 97.86	90	51.232	13.4
Quantization	Conv2d: 5bit Linear: 5bit	84.97 / 98.65	84.37	80.0756	20.9442
	Conv2d: 8bit Linear: 8bit	82.20 / 98.73	75	128.08	33.5
Pruning+ Quantization	Conv2d: 0.5 / 5bit Linear: 0.9 / 5bit	83.58 / 98.77	95.95	20.749	5.427
	Conv2d: 0.5 / 8bit Linear: 0.9 / 5bit	83.46 / 98.96	95.44	23.3618	6.110
	Conv2d: 0.5 / 5bit Linear: 0.9 / 8bit	84.31 / 98.86	95.15	24.8475	6.5
	Conv2d: 0.5 / 8bit Linear: 0.9 / 8bit	82.55 / 98.52	94.6	27.6653	7.236

[표 5-2] 객체 분류 모델 경량화 실험 결과 비교 (VGG16)

[표 5-2]은 VGG16 모델에 대해 Pruning, Quantization, 그리고 두 기법을 결합한 결과를 보여준다. Pruning 적용 비율이 증가할수록 파라미터 수와 모델 크기는 크게 감소하였으나, 이에 따라 정확도도 상대적으로 크게 저하됐다. Quantization 적용 방식은 5bit 및 8bit 설정 모두에서 파라미터 수가 큰 폭으로 감소하였으며, Pruning 방식보다 정확도가 더욱 안정적으로 유지되는 경향을 보였다. 특히 Pruning과 Quantization을 결합하여 적용하였을 때는 높은 압축 효과를 달성하였다. 일부 조합에서는 파라미터 수가 원본 대비 약 95.95% 감소하면서도(5.4M) Top-1 정확도는 약 1~2% 수준만 감소하여, 단독으로 적용했을 때 보다 구조적 축소와 성능 유지 측면에서 더욱 균형 잡힌 성능을 제공함을 확인할 수 있었다.

경량화 방식	경량화 세팅	TOP-1/ TOP-5	압축률	파라미터 크기 (MB)	파라미터 개수 (M)
Original		39.67 / 89.13	X	221.88	58.2
Pruning	Conv2d: 0.5 Linear: 0.5	36.16 / 87.3	50	110.94	29.1
	Conv2d: 0.9 Linear: 0.9	10.1 / 50.23	90	22.19	5.82
Quantization	Conv2d: 5bit Linear: 5bit	30.16 / 84.45	84.3	34.835	9.1374
	Conv2d: 8bit Linear: 8bit	38.47 / 89.06	74.08	55.514	14.5617
Pruning+ Quantization	Conv2d: 0.5 / 5bit Linear: 0.9 / 5bit	30.16 / 83.21	85.721	31.4606	8.2522
	Conv2d: 0.5 / 8bit Linear: 0.9 / 5bit	39.14 / 88.53	81.081	41.9466	11.0106
	Conv2d: 0.5 / 5bit Linear: 0.9 / 8bit	39.11 / 89.06	85.82	31.4622	8.2526
	Conv2d: 0.5 / 8bit Linear: 0.9 / 8bit	39.54 / 88.97	81.08	41.9782	11.0110

[표 5-3] 객체 분류 모델 경량화 실험 결과 비교 (ResNet152)

[표 5-3]은 ResNet152 모델에 대한 경량화 실험 결과를 요약한 것이다. Pruning 단계에서는 VGG16과 달리 ResNet152는 깊고 계층적 연결 구조를 갖기 때문에, Pruning 적용 비율이 0.9일 때 Top-1 정확도가 36.97%에서 10.1%로 급격히 저하하여 구조적 특성상 VGG16에 비해 훨씬 취약한 경향을 보였다. Quantization을 적용했을 때에는 비교적 안정적인 정확도를 유지하였으며, 모델 크기 역시 약 70~80% 수준까지 감소하는 효과가 확인되었다. Pruning과 Quantization을 결합하였을 때 대부분 조합이 원본 대비 비교적 안정적인 정확도를 유지하였으나, [표 5-3]의 [Conv2d: 0.5 / 5bit, Linear: 0.9 / 5bit] 조합은 Top-1 정확도가 30.16%로 낮아지는 등 예외적으로 성능 저하가 비교적 크게 나타났다. 종합적으로 ResNet152는 VGG16에 비해 경량화 기법에 더 민감하게 반응함을 알 수 있다.

나) 객체 검출 모델 경량화 적용 결과

경량화 방식	경량화 Setting	mAP (%)	압축률	파라미터 크기 (MB)	파라미터 개수 (M)
Original		69.9	X	522.91	137.08
Pruning	Conv2d: 0.5 Linear: 0.5	67.59	50	261.46	68.54
Pruning	Conv2d: 0.9 Linear: 0.9	43.72	90	52.29	13.71
Quantization	Conv2d: 5bit Linear: 8bit	68.3	76.19	124.5	32.64
Quantization	Conv2d: 8bit Linear: 8bit	68.96	75	130.48	34.27
Quantization	Conv2d: 5bit Linear: 5bit	63.20	84.37	81.73	21.43
Quantization	Conv2d: 1bit Linear: 1bit	0.00014	96.88	16.28	4.28

[표 5-4] 객체 검출 모델 경량화 실험 결과 비교 (Faster-RCNN)

[표 5-4]는 Faster R-CNN 모델의 경량화 실험 결과를 요약한 것이다. 이전과 마찬가지로 Pruning 적용 비율이 지나치게 높은 경우에는 mAP가 크게 저하되는 경향을 보였다. 반면 Quantization을 적용했을 때 5bit 또는 8bit 설정에서도 mAP가 원본 대비 비교적 안정적으로 유지되었으며(약 68~69%), 모델 크기는 약 70~80% 수준까지 감소하는 효과가 확인되었다. 그러나 Faster-RCNN은 Pruning을 Quantization과 결합하여 적용한 경우 대부분 조합에서 mAP가 0에 수렴하여 정상적인 성능을 유지하지 못하였으며, 이에 따라 본 연구에서는 해당 결과들을 표에서 제외하였다. 전체적으로 Faster R-CNN은 경량화 설정에 민감하게 반응하며, Quantization이 Pruning보다 정확도 유지 측면에서 상대적으로 유리한 경향을 보였다.

다) 객체 추적 모델 경량화 적용 결과

[표 5-5]는 SiamRPN++ 모델에서 수행한 경량화 실험 결과를 요약한 것

HANSUNG UNIVERSITY

경량화 방식	경량화 Setting	Average IoU (%)	압축률	파라미터 크기 (MB)	파라미터 개수 (M)
Original		80.71	X	205.81	53.95
Pruning	Conv2d : 0.5	75.59	50%	102.905	26.975
Pruning	Conv2d : 0.75	65.32	75%	51.4525	13.4875
Quantization	Conv2d : 5bit	79.3	93.60	13.1718	3.4528
Quantization	Conv2d : 8bit	75.7	96.01	8.2119	2.1527

[표 5-5] 객체 추적 모델 경량화 실험 결과 비교 (SiamRPN++)

이다. SiamRPN++은 분류, 검출 모델과 달리 완전 연결 레이어를 포함하지 않기 때문에, 경량화는 주로 합성곱 레이어를 중심으로 적용하였다. Pruning의 경우, 적용 비율을 높일수록 평균 IoU가 80.71%에서 65.32%로 하락하는 등 성능 저하가 뚜렷하게 나타났다. 반면 Quantization은 IoU가 원본 대비 비교적 안정적으로 유지되었으며(약 75~79%), 모델 크기는 90% 이상 감소하는 높은 압축 효과를 보였다. 또한, Faster-RCNN과 마찬가지로, 두 기법을 결합하여 적용하였을 때, 대부분 조합에서 IoU가 급격히 저하하는 현상이 발생하여 정상적인 추적 성능을 유지하지 못하였으며, 이러한 이유로 본 연구에서는 혼합 적용 결과를 표에서 제외하였다. 전반적으로 SiamRPN++ 모델 역시 경량화 설정에 민감하게 반응하는 특성을 보였다.

3) 추론 최적화 기법(TensorRT) 적용 결과

앞 절에서 다룬 내용은 모델의 파라미터를 직접 줄이는 구조적 경량화 방식이라면, TensorRT는 모델 구조를 변경하지 않은 상태에서 연산 그래프 최적화, 커널 융합, 정밀도 변환 등을 통해 추론 과정을 가속하는 런타임 기반 최적화 기법이다. 본 실험에서는 TensorRT를 다양한 모델에 적용하여 실제 추론 지연 시간에 미치는 영향을 평가하였다. 실험은 임베디드 GPU 환경에서 수행하였으며, 사용된 하드웨어 및 소프트웨어 사양은 다음 절의 [표 5-8]

모델	경량화 방식	설정	Top-1 / Top-5	모델 추론 속도 (sec.)
VGG16	Original		85.28 / 98.84	1.5333
	TensorRT	FP32	85.29 / 98.98	1.4387
	TensorRT	FP16	85.27 / 99.08	1.4914
ResNet152	Original		39.67 / 89.13	2.8255
	TensorRT	FP32	39.16 / 88.85	1.4686
	TensorRT	FP16	39.50 / 89.17	1.4298

[표 5-6] 분류 모델에 대한 TensorRT 적용 시 추론 속도 비교 결과

경량화 방식	설정	평균 IoU	모델 추론 속도 (ms)
Original(python)	-	80.71	29.212
Original(c++)	-	75.59	19.338
Tensor RT(python)	FP32	81.115	27.042

[표 5-7] 추적 모델에 대한 TensorRT 적용 시 추론 속도 비교 결과

에 제시하였다.

[표 5-6]은 ResNet152와 VGG16 모델을 대상으로 TensorRT 기반 추론 최적화 기법을 적용한 결과를 요약한 것이다. 실험 결과, 두 모델 모두에서 TensorRT 적용 시 추론 지연 시간이 전반적으로 감소하였으며, 특히 FP16로 정밀도를 축소 시켰을 때 추가적인 속도 향상 효과가 확인되었다. 이는 모델 구조를 변경하지 않고도 런타임 최적화를 통해 실질적인 추론 성능 개선이 가능함을 보여준다.

[표 5-7]은 추적 모델(SiamRPN++)을 Python과 C++ 환경에서 각각 실행한 경우, 그리고 TensorRT를 적용했을 때의 추론 속도를 비교한 결과를 나타낸 것이다. Python은 구현과 디버깅이 용이하다는 장점이 있으나, GIL(Global Interpreter Lock)로 인해 멀티스레드 기반 병렬 실행에 제약이

존재한다. 반면 C++ 환경은 이러한 제한이 없이, 객체 추적 모델을 멀티스레드 방식으로 효율적으로 수행할 수 있으며 스트림을 활용한 병렬 실행 기법도 적용할 수 있다. 본 연구에서는 이미지 로딩 및 전처리 단계에도 멀티스레딩을 적용하여 전체 처리량을 개선하였다(Kim, M., et al, 2023).

실험 결과, C++ 환경에서 Python 대비 더 빠른 추론 속도를 보였으며, TensorRT를 적용한 경우에도 전체 지연 시간이 추가로 감소하였다. 또한, TensorRT 적용 시 평균 IoU는 원본 대비 안정적으로 유지되거나 소폭 향상되어, 추론 최적화가 정확도에 부정적인 영향을 미치지 않는 것으로 확인되었다.

한편, 본 연구에서는 Faster R-CNN에 대해 TensorRT 기반 최적화를 적용하지 않았다. 이는 모델의 구조적 특성으로 인해 TensorRT 적용에 여러 기술적 제약이 존재하기 때문이다. Faster R-CNN은 NMS(Non-Maximum Suppression), RoI Pooling과 같이 입력과 출력의 크기가 매번 달라지는 연산을 포함하고 있으며, 이러한 연산은 TensorRT에서 기본적으로 지원되지 않아 별도의 커스터마이징이 필요하다. 특히 RoI의 개수와 크기가 이미지마다 변화하기 때문에, 고정된 연산 흐름을 전제로 정적 실행 그래프를 생성하는 TensorRT의 특성과 근본적으로 맞지 않는다. 이러한 이유로 입력 텐서의 형태와 구조가 매번 달라지면 TensorRT가 수행하는 커널 융합, 정적 그래프 최적화와 같은 핵심 기능이 적용되지 못해 오히려 최적화 효과가 제한된다. 이러한 이유로 본 연구에서는 Faster R-CNN을 TensorRT 실험 범위에서 제외하고, 분류 및 추적 모델에 대해서만 TensorRT 기반 추론 최적화 효과를 분석하였다.

제 2 절 블록 레벨 스케줄링 기반 다중 DNN 실행 최적화 기법의 실험 결과

본 절에서는 블록 레벨 스케줄링 기반 다중 DNN 실행 최적화 기법의 효과를 입증하기 위해 수행한 실험들을 상세히 기술한다. 먼저, 실험 환경과 구현 방법을 간략히 설명한 후, 실험 결과를 제시하고 이에 대한 분석을 제공한

다.

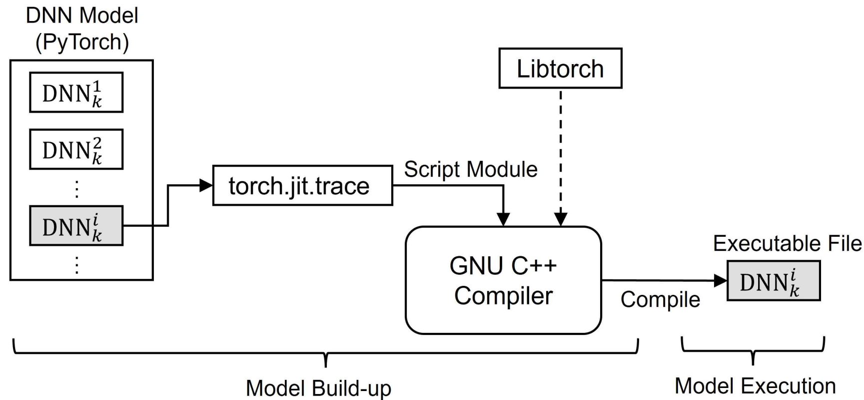
1) 실험 환경 및 구성

가) 시스템 구성 및 실험 조건

	Classification	Description
HW	CPU	8-core ARM v8.2 Carmel 64-bit CPU, 8MB L2, 4MB L3 cache
	GPU	512-core Volta GPU with Tensor cores
	Memory	16GB 256-Bit LPDDR4x, 137GB/s
	Storage	32GB eMMC 5.1
SW	Kernel Ver.	Linux 5.10.216
	SW Package	JetPack 5.1.4
	CUDA Ver.	CUDA v11.4.315

[표 5-8] 실험에 쓰인 NVIDIA Jetson AGX Xavier Developer Kit 사양

제안된 프레임워크의 효과를 검증하기 위해, 대표적인 임베디드 GPU 기반 자율 시스템 플랫폼인 NVIDIA Jetson AGX Xavier Developer Kit에서 실험을 수행하였다. 대상 시스템의 하드웨어 및 소프트웨어 사양한 [표 5-8]에 정리되어 있다. 실험 대상 모델은 객체 분류에 VGG16(ImageNet)(Deng, J., et al, 2009), 객체 검출에 Faster R-CNN(PASCAL VOC 2007)(Everingham, M. et al, 2010), 객체 추적에 SiamRPN++(TrackingNet)(Muller, M., et al, 2018)을 사용하였다. 전체 네트워크는 각각의 연산 특성을 고려하여 기능적인 블록으로 분할되었으며, 각 블록의 사전 측정된 실행시간은 프로파일링 데이터로 저장되어 *LAG*계산에 필요한 데드라인을 설정하는 기준으로 활용하였다. 모델 구성에 따라 각 작업에 대해 분류는 500장의 이미지, 검출은 300장의 이미지, 추적은 3개의 동영상 시퀀스로 구성된 샘플 데이터셋으로 구성되었다. 테스트 샘플의 수는 임베디드 GPU 환경의 실행 특성과 지연 시간 추세를 분석하는 데 초점을 맞추기 위해 의도적으로 제한하였다. 전체 데이터셋을 평가할 경우 연산 및 시간적 제약으로 인해 과도한 실행시간이 요구되기 때문이다. 이에 따라 대표적인 샘플을 선정하여 실험을 수



[그림 5-1] 제안된 프레임워크의 DNN 모델 실행 파이프라인

행하였으며, 이는 제안된 프레임워크의 동작 특성을 관찰하는 데 충분하다고 판단하였다. 또한, 통계적 신뢰성을 확보하기 위해 모든 테스트를 동일한 조건에서 50회 반복 수행하고, 최종 성능 지표는 그 평균값으로 산출하였다.

본 논문은 베이스라인 모델 대비 상대적 성능 개선을 검증하는 데 초점을 두고 있다. 따라서 모든 실험은 입력 순서를 무작위로 섞지 않고 고정된 순서의 데이터셋을 사용하여 수행하였다. 각 모델은 충분히 학습된 사전학습 네트워크를 사용하였으며, 경량화 모델의 경우 정확도를 보정하기 위해 제한적인 범위에서만 추가 미세조정을 적용하였다. 이러한 설정은 제안된 프레임워크가 추가적인 학습 절차 없이 다양한 사전학습 DNN에 직접 적용될 수 있음을 보여주며, 실험 규모를 단순화하는 동시에 제안 기법의 상대적 효과성과 실용성을 명확하게 검증할 수 있도록 한다.

또한, 실험에 사용된 대상 시스템은 여러 CPU와 단일 GPU가 시스템 메모리를 공유하는 구조이므로, 다수의 딥러닝 모델을 동시에 상주시키기에는 메모리 자원이 제한적이다. 이에 따라 본 연구에서는 메모리 초과가 발생하지 않는 범위에서 분류, 검출, 추적의 세 가지 모델만을 실험 대상으로 선정하였다. 모델을 추가로 실행하려면 NAND 플래시를 온디맨드 방식으로 활용해야 하지만, 이는 높은 I/O 지연을 유발하여 실시간 추론 성능을 저하시킬 수 있다. 따라서 본 논문에서는 각 DNN 모델의 Level 0~2에 해당하는 모든 블록을 초기화 단계에서 시스템 메모리에 미리 적재하였다. 이에 따라 실행 중 블

록 교체는 포인터 참조만 변경하면 되어 오버헤드가 매우 작으며, 추가적인 저장 공간이나 접근 비용 없이 동일한 메모리 내에서 블록 레벨 실행이 수행된다.

나) 구현 방식

C++ 환경은 CUDA 스트림과 멀티스레딩을 세밀하게 제어할 수 있는 저수준 인터페이스를 제공하여, 블록의 비동기 실행이 가능하다. [그림 5-1]은 C++ 환경에서의 모델 실행 파이프라인을 나타낸다. 먼저, 사전학습된 DNN 모델은 PyTorch 프레임워크를 통해 블록이라 불리는 계산 단위로 분할된다. 이후 각 블록은 torch.jit.trace[15] 을 사용하여 TorchScript 형태로 변환된다. 변환된 스크립트 모듈은 PyTorch의 C++ 프론트엔드인 Libtorch와 GNU C++ 컴파일러를 통해 실행 가능한 바이너리 파일로 컴파일된다. 이후 각 블록은 독립적으로 호출 및 실행될 수 있는 구조로 구성되며, 이는 블록 단위 스케줄링의 기반이 된다. Libtorch는 학습된 PyTorch 모델을 Python 환경 없이 C++에서 직접 추론할 수 있도록 지원하는 라이브러리로, 특히 Python 실행 환경이 GIL[16]과 같은 요인으로 제약될 수 있는 시스템에서 유용하게 활용된다.

2) 다중 DNN 실행 최적화 기법 실험 결과

가) 병렬 및 순차 블록 실행 결과 비교

단일 GPU를 공유하는 임베디드 시스템의 다중 DNN 실행 환경에서는, 블록 수준 병렬 실행이 항상 성능 향상을 보장하지 않는다. 이는 연산 자원의 경합으로 인해, 오히려 순차 실행이 병렬 실행보다 더 짧은 실행시간을 제공하는 상황이 나타날 수 있다. 본 절에서는 서로 다른 DNN 모델들로부터 다양한 블록 조합을 구성한 뒤, 병렬 실행과 순차 실행 간의 지연 시간 차이를 정량적으로 분석하였다. 여기서 병렬 실행은 여러 CUDA 스트림을 활용하여

15) PyTorch Dev Team, torch.jit.trace,
online: https://pytorch.org/docs/stable/generated/torch.jit.trace.html

16) PyTorch Software Foundation, Global Interpreter Lock (GIL),
online: https://docs.python.org/3/library/threading.html

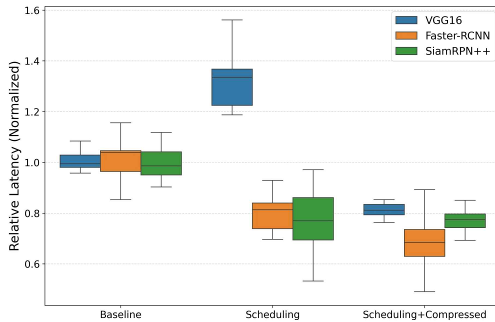
Block Case	Parallel (ms)	Serialize (ms)	Reduction Rate (%)
Case 1	230.032 ms	193.052 ms	16.09 %
Case 2	261.832 ms	242.126 ms	7.53 %
Case 3	1082.233 ms	1039.391 ms	3.96 %

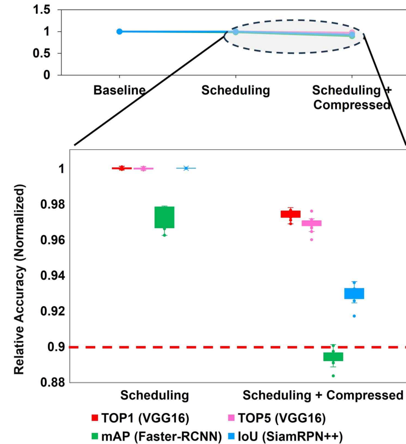
[표 5-9] 선택된 블록 조합에 대한 병렬 실행과 순차 실행의 성능 비교

각 블록을 동시에 수행하는 방식을 의미하며, 순차 실행은 단일 CUDA 스트림에서 블록을 직렬로 처리하는 방식을 의미한다.

실험 결과, 일부 블록 조합에서는 병렬 실행보다 순차 실행이 더 효과적인 경우가 관찰되었다. 이러한 현상을 명확히 보여주기 위해, 병렬 실행이 성능 향상에 실패한 대표 사례들을 제시한다. 특히 아래 제시한 Case 1~3에 제시된 블록들은 FLOPs와 메모리 사용량 측면에서 연산 부담이 가장 큰 구성 요소들로 이루어져 있다. SiamRPN++의 Backbone은 다수의 합성곱 계층으로 구성되어 있어 특징 추출 과정에서 매우 큰 연산량을 요구한다. 반면, VGG16의 Classifier와 Faster R-CNN의 Head는 완전 연결 계층의 방대한 파라미터로 인해 막대한 메모리 대역폭을 소모한다. 이러한 블록들이 병렬로 실행될 경우 GPU 자원 경쟁이 심화되며, 그 결과 오히려 순차 실행이 더 높은 성능을 보이는 경우가 발생하게 된다. 반면, SiamRPN++의 Neck과 같이 FLOPs가 상대적으로 작은 블록 조합에서는 병렬 실행이 더 빠르거나 두 실행 방식 간 차이가 크지 않았으므로 표에서 제외하였다. 같은 이유로 Case 1~3에 포함되지 않은 다른 블록 조합들도 모두 생략하였다. [표 5-9]는 세 가지 조합(Case 1~3)에 대해 병렬 실행과 순차 실행 각각의 평균 처리 시간을 비교하고, 순차 실행이 병렬 실행보다 우수한 경우의 시간 감소율을 함께 제시한다. 각 케이스의 구성은 다음과 같다. Case 1은 SiamRPN++-Backbone과 VGG16-Classifier 조합이며, Case 2는 SiamRPN++-Backbone과 Faster R-CNN-RPN 조합이다. Case 3은 Faster R-CNN-Head와 VGG16-Classifier로 구성된다.

[표 5-9]에서 확인할 수 있듯이, 제시된 모든 사례에서 모두 순차 실행이 병렬 실행보다 더 낮은 지연 시간을 보였다. 특히 Case 1의 경우에는 약





(a) (b)

[그림 5-2] 서로 다른 스케줄링 방식 적용 시 세 가지 DNN 모델의 지연 시간 (a) 및 정확도(b) 비교

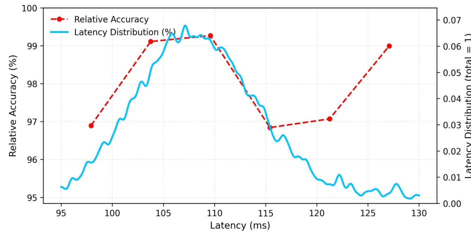
HANSUNG UNIVERSITY

16.09%의 감소율을 보였는데 이는 VGG16 Classifier와 SiamRPN++ Backbone이 GPU에서 동시에 실행될 때 발생하는 심각한 자원 경합의 영향으로 해석할 수 있다. Case 2, Case 3도 각각 7.53%, 3.96%의 감소율을 보였다. 이러한 결과는 블록 수준 병렬 실행은 항상 성능 향상을 보장하지 않으며, 각 블록의 자원 사용 특성을 사전에 고려한 스케줄링 전략이 필요함을 보여준다.

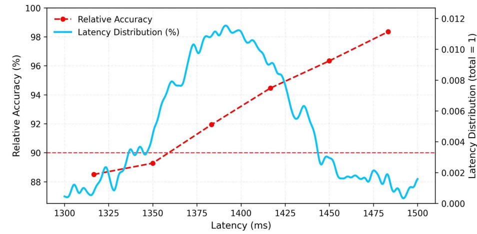
나) 3개 모델 동시 실행

본 실험에서는 5장 2절 1) 에서 언급한 세 가지 DNN 모델을 동시에 실행할 때, 제안된 스케줄링 및 블록 수준 동적 전환 기법이 지연 시간과 정확도에 미치는 영향을 분석한다. 각 모델은 연산 구조와 연산량이 상이하며, 이에 따라 처리 시간의 분포 또한 서로 다르다. 따라서 본 논문에서는 절대적인 측정값이 아닌 기준(Baseline)결과를 기반으로 정규화된 값을 사용하여 상대적인 성능 변화를 비교하였다. 이때, 정규화 기준이 되는 Baseline 환경은 1로 설정되며, 해당 절에서 수행되는 모든 실험에서 동일한 기준을 유지하였다.

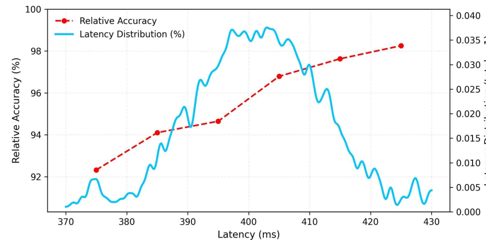
실험은 세 가지 환경에서 수행되었다. 먼저 Baseline은 세 모델을 모두 원



(a) VGG16



(b) Faster-RCNN



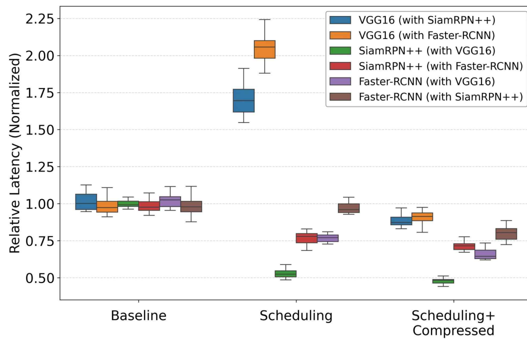
(c) SiamRPN++

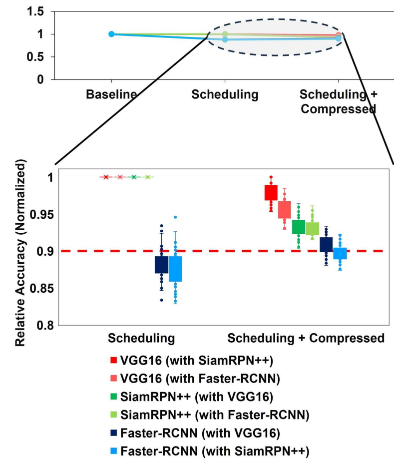
[그림 5-3] 각 모델의 지연 시간 분포에 따른 평균 정확도

HANSUNG UNIVERSITY

본 상태(Level 0)로 유지한 채, 별도의 스케줄링 제어 없이 동시에 실행한 환경이다. Scheduling은 제안한 스케줄링 기법을 적용하여, 각 모델 블록의 실행 시점을 조정함으로써 GPU 자원 경합을 완화하고 병렬성을 개선시킨 환경이다. 마지막으로 Scheduling + Compressed은 스케줄링뿐만 아니라 지연 상태 및 정확도에 따라 각 모델 블록을 사전에 생성된 압축 버전(Level 1 또는 Level 2)으로 동적으로 전환하거나 원본(Level 0)으로 복원하여, 지연 시간과 정확도 간의 균형을 실시간으로 조절한 환경이다.

[그림 5-2]에서 확인할 수 있듯이, Scheduling + Compressed 기법은 VGG16, Faster R-CNN, SiamRPN++의 실행 지연 시간을 각각 16.1%, 29.3%, 22.1% 감소시켰다. 동시에 정확도는 VGG16의 경우 2.6%(Top-1/Top-5), Faster R-CNN은 10.5%(mAP), SiamRPN++은 7.0%(IoU) 감소하는 데 그쳐, 모든 모델에서 정확도 저하를 약 10% 이내로 유지하였다. 반면, Scheduling 기법만 적용한 경우 VGG16의 지연 시간이 오히려 37.6% 증가하는 현상이 관찰되었는데, 이는 블록 수준의 동적 전환 기법 없이 스케줄링만 적용할 경우 GPU 자원 경쟁으로 인해 지연이 악화될 수 있음을 보여준다.



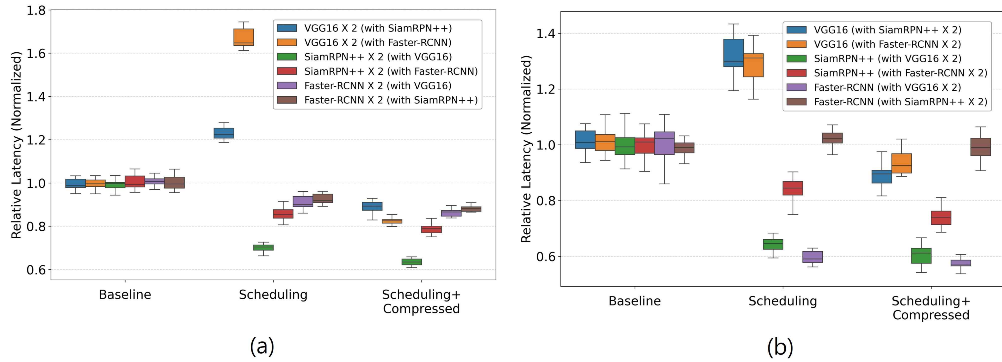


[그림 5-4] 두 가지 DNN 동시 실행 환경에서 지연 시간(a) 및 정확도(b) 비교

[그림 5-3]은 각 모델에 대해 지연 시간 분포에 따른 평균 정확도를 나타낸 것이다. x축은 [그림 5-2]의 Scheduling + Compressed 환경과 동일한 조건에서 측정된 블록 실행시간을 정수 단위로 반올림한 값을 나타내며, 왼쪽 y축은 상대적인 정확도(%), 오른쪽 y축은 정규화된 지연 시간 분포를 의미한다. 각 블록의 실행시간은 소수점 단위의 미세한 정밀도로 측정되지만, 전체 분포를 효과적으로 시각화하기 위해 정수 단위로 반올림하였다. 그림에서 보이듯이, 지연 시간이 감소할수록 정확도가 점진적으로 감소하는 경향을 보여주며, 제안한 Scheduling + Compressed 기법이 지연 시간과 정확도 간의 균형을 효과적으로 유지하고 있음을 확인할 수 있다. 반면, VGG16은 약 96~100% 범위에서 완만한 진동 패턴을 보이며, 지연 시간 변화에도 정확도 변동이 거의 나타나지 않는다. 이는 VGG16이 구조적으로 단순하고 고정된 합성곱 연산만 수행할 뿐 아니라, MUSCO 기반의 저랭크 분해 압축 기법을 적용하더라도 블록 수준의 특성이 크게 변화하지 않아 정확도 저하가 거의 발생하지 않기 때문이다.

다) 2개 모델 동시 실행

여기서는 서로 다른 두 개의 DNN 모델을 동시에 실행할 때, 제안된 프



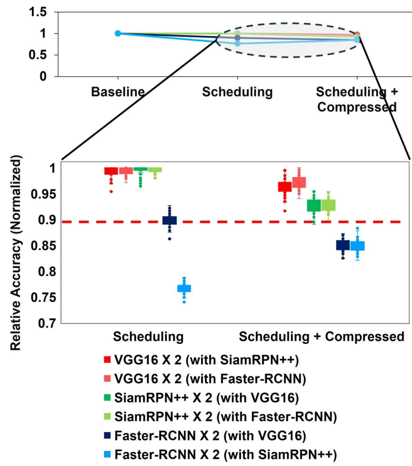
[그림 5-5] 동일한 두 DNN과 하나의 이질적 DNN 동시 실행 환경에서의 지연 시간 성능 비교

레임워크가 실행 지연과 정확도에 미치는 영향을 분석한다. 두 모델을 한 번에 조합하여 실험을 수행한 이유는, 세 모델이 동시에 실행되는 환경에서 발생하는 문제를 분할 정복 방식으로 접근하기 위함이다. 이를 통해 각 모델간의 상호작용을 독립적으로 분석할 수 있다.

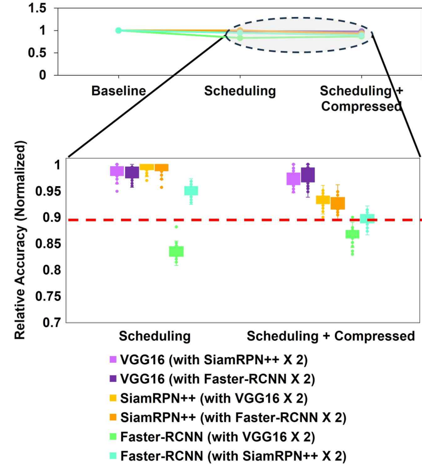
[그림 5-4]는 이러한 이중 DNN 실행 환경에서의 실험 결과를 나타낸다. 예를 들어, VGG16 (with SiamRPN++)은 VGG16모델이 SiamRPN++과 동시에 실행될 때의 정규화된 성능을 의미한다. Scheduling + Compressed 적용 시 지연 시간은 VGG16에서 각각 10.8%와 9.8%, SiamRPN++에서 51.3%와 28.1%, Faster R-CNN에서 34.2%와 19.9% 감소하였다. 정확도의 경우, VGG16은 2.1%와 4.6%, SiamRPN++은 6.8%와 7.0%, Faster R-CNN은 9.0%와 10.3% 감소하였다. 그럼에도 불구하고, 모든 조합에서 정확도는 기준 대비 90% 이상으로 유지되었다.

라) 동일한 2개 DNN과 1개의 이기종 DNN 동시 실행

여기서는 동일한 두 개의 DNN 모델과 추가적인 하나의 이질적 DNN 모델이 동시에 실행되는 환경에서, 제안한 기법이 미치는 영향을 평가한다. [그림 5-5]은 이러한 구성에서의 정규화된 지연 시간 결과를 보여준다. 예를 들어, [그림 5-5]의 (a)에서 VGG16 ×2 (with SiamRPN++)는 두 개의 VGG16 인스턴스와 하나의 SiamRPN++ 인스턴스를 동시에 실행하는 구성이



(a)



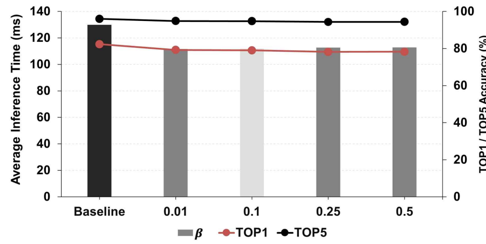
(b)

[그림 5-6] 동일한 두 DNN과 하나의 이질적 DNN 동시 실행 환경에서의 정확도 비교

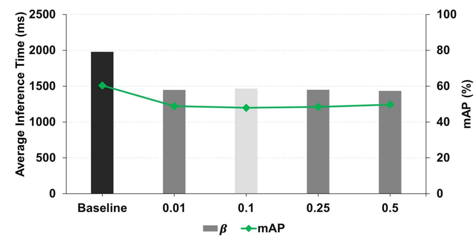
며, 두 VGG16 인스턴스의 평균 지연 시간을 기준으로 정규화한 결과이다. 반대로 (b)의 VGG16 (with SiamRPN++ ×2)는 하나의 VGG16 인스턴스를 두 개의 SiamRPN++ 인스턴스와 함께 실행하는 경우를 의미하며, 단일 VGG16 인스턴스를 기준으로 정규화된 값이다.

이와 같은 방식으로 총 12개의 조합을 구성하였으며, 각 조합은 두 개의 동일한 모델과 하나의 이질적 모델로 이루어져 있다. Scheduling + Compressed 기법을 적용한 결과, VGG16은 최대 16.9%, SiamRPN++은 36.8%, Faster R-CNN은 42.3%의 지연 시간 감소 효과를 보였다. 반면, Scheduling 기법만 적용한 경우에는 일부 조합에서 지연 시간이 증가하거나 개선 효과가 제한적으로 나타났다. 이는 모델 간 연산 구조의 비대칭성과 자원 경합으로 인해 스케줄링만으로는 충분한 성능 개선이 어려울 수 있음을 시사한다. 이러한 결과는 제안한 스케줄링 및 블록 수준 동적 전환 기법이 모델 중복 환경에서도 일관적으로 지연 시간을 감소시킬 수 있음을 보여준다.

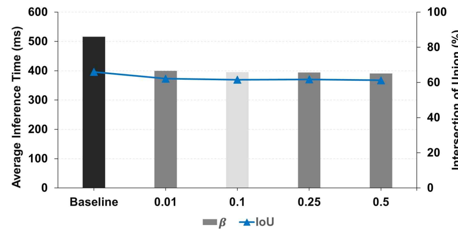
[그림 5-6]은 각 구성별 정규화된 정확도 변화를 시각적으로 비교한 결과이다. Scheduling + Compressed 기법을 적용한 결과, 일부 조합에서는 정확도가 소폭 감소하였으며, 특히 Faster R-CNN이 포함된 구성에서는 약



(a) VGG16



(b) Faster-RCNN



(c) SiamRPN++

[그림 5-7] 각 모델에 대해 EMA_{long}의 파라미터 β 값 변화에 따른 지연 시간 및 정확도 비교

10%~15% 수준의 정확도 감소가 관찰되었다. 이러한 경우를 제외하면 대부분의 조합에서 정확도는 Baseline 대비 90% 이상을 유지하였으며, 이는 지연 시간에 민감한 성능 요구 사항을 고려할 때 시스템 전체 관점에서 수용 가능한 수준으로 판단된다.

3) 상충관계 분석

가) β 스케일링에 따른 지연 시간 및 정확도 분석

여기서는 제안된 EMA 기반 블록 수준 동적 전환 기법에서 장기 추세 지표로 활용되는 EMA_{long}의 파라미터 β가 시스템 성능에 미치는 영향을 분석한다. 구체적으로, 파라미터 α는 0.99로 고정하고 β는 0.01에서 0.1, 0.25, 0.5로 점진적으로 증가시키며 실험을 수행하였다. β 값의 변화가 지연 시간과 정확도에 미치는 영향은 VGG16, Faster R-CNN, SiamRPN++ 세 가지 DNN 모델을 대상으로 평가하였고, 세 모델은 모두 동시에 실행된다. 평가 지표로는 평균 지연 시간, VGG16의 Top-1/Top-5 Faster R-CNN의 mAP, SiamRPN++의 IoU를 사용하였다. [그림 5-7]은 각 모델에 대해 β 값 변화

에 따라 지연 시간과 정확도가 어떻게 달라지는지를 시각적으로 보여준다.

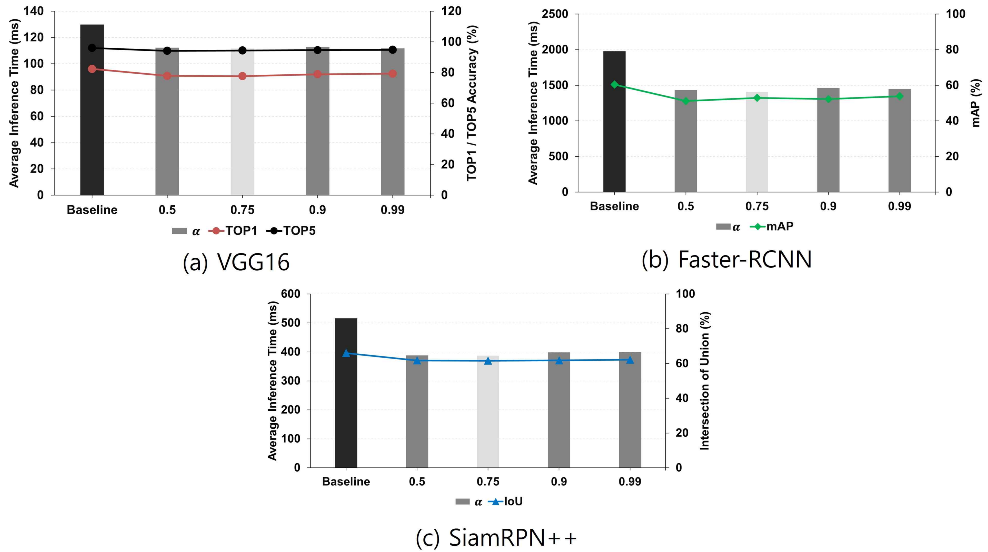
먼저 VGG16 모델에서는 Baseline에서 평균 레이턴시는 129.88ms였으나, Beta 값을 0.01과 0.1로 설정했을 때 각각 111.71ms, 111.90ms로 약 18ms 가까이 감소하였다. 정확도는 β = 0.1까지 Top-1은 79.0%, Top-5는 94.75%로 안정적으로 유지되었으나, 이후 β = 0.25부터는 Top-1이 78.15%까지 하락하는 등 점진적인 성능 저하가 나타났다. 이러한 결과는 β = 0.1로 설정하는 것이 지연 시간 개선과 정확도 유지 간의 균형 측면에서 가장 우수함을 보여준다. 절대적인 성능 향상 폭은 크지 않지만, 이러한 설정은 작지만 일관된 성능 향상을 제공하므로 최적의 선택으로 판단된다.

Faster R-CNN의 경우, Baseline에서 평균 지연시간은 1979.12ms였으나, β값을 0.01과 0.1로 설정했을 때 각각 1448.56ms와 1466.03ms로 감소하여 500ms 이상의 지연 시간 감소가 나타났다. β 값을 0.25와 0.5로 더 증가시킨 경우에도 1449.494ms, 1433.93ms를 기록하며 추가적인 개선 효과가 관찰되었다. 정확도는 β = 0.1에서 mAP가 47.94%로 가장 낮게 기록되었으나, β = 0.5에서는 49.66%로 소폭 증가하였다.

SiamRPN++의 경우, Baseline의 평균 레이턴시는 515.77ms, IoU는 65.99% 였으며, β가 증가함에 따라 지연 시간은 390.65ms 까지 지속적으로 감소하였다. 정확도는 소폭의 변동을 보였으며, 높은 β 값에서는 IoU가 61.56%까지 점진적으로 감소하였다. 종합적으로, β 값이 클수록 EMA_{long} 이 최근 지연 시간 변화에 더 민감하게 반응하여 불필요한 압축 전환이 발생하고, 이로 인해 정확도가 다소 저하되는 경향이 나타난다. 반대로 β 값이 지나치게 작으면 압축 전환 빈도가 감소하여 지연 시간 개선 폭이 제한된다. 따라서 α = 0.99로 고정한 조건에서 β = 0.1이 지연 시간 감소와 정확도 유지 간의 최적의 균형을 제공하는 것으로 확인되었다. 본 논문에서 제시한 모든 실험에서는 이 설정을 기본값으로 사용하였다.

나) α 스케일링에 따른 지연 시간 및 정확도 분석

여기서는 제안된 EMA 기반 블록 수준 동적 전환 기법에서 단기 추세 지표로 활용되는 EMA_{short}의 파라미터 α 값이 시스템 성능에 미치는 영향을



(a) VGG16

(b) Faster-RCNN

(c) SiamRPN++

[그림 5-8] 각 모델에 대해 EMA_{short} 파라미터 α 값 변화에 따른 지연 시간 및 정확도 비교

HANSUNG UNIVERSITY

분석한다. 구체적으로, β 값은 0.01로 고정하고, α 값을 0.5, 0.75, 0.9, 0.99로 점진적으로 증가시키며, 실험을 수행하였다. [그림 5-8]는 각 모델에 대해 α 값 변화에 따라 지연 시간과 정확도가 어떻게 달라지는지를 시각적으로 보여준다.

VGG16의 경우, α 값을 조정한 이후 모든 설정에서 지연 시간은 약 111~112ms 수준으로 약 17~18ms정도 감소하여 일정 수준의 개선 효과를 보였다. 정확도 측면에서는, α = 0.5일 때 Top-1이 77.79%, Top-5가 94.14%로 다소 하락했다. 하지만 이후 α 값을 증가시킬수록 점진적으로 회복되었으며, 특히 α = 0.99에서는 Top-1은 79.25%, Top-5은 94.81%로 가장 높은 수치를 기록하였고, α = 0.9에서도 각각 78.83%, 94.63%을 유지하여 0.9 이상의 α 값에서 정확도 안정성이 보장됨을 확인할 수 있다.

Faster R-CNN의 경우, Baseline에서 mAP는 60.4%으로 측정되었다. α 값을 조정한 이후 지연 시간은 전반적으로 크게 개선되었으며, 특히 α=0.75에서 1407.89ms, α = 0.5에서 1433.58ms로 약 500ms 이상 감소하였다. 반면에 정확도는 α = 0.5에서 51.12%, α = 0.75에서 52.94%로 다소 낮아졌

DNN Model	Baseline (ms)	Not Scheduled (ms)	Scheduled (ms)
VGG16	130	112.021 (1.16x)	109.008 (1.19x)
Faster-RCNN	1979.12	1585.36 (1.25x)	1400.05 (1.41x)
SiamRPN++	515.767	441.077 (1.17x)	401.708 (1.28x)

[표 5-10] 스케줄링 적용 여부에 따른 블록 수준 동적 전환 성능 비교

으며, α = 0.99에서는 53.84로 가장 높은 정확도를 기록하였다. 결과적으로 α = 0.9~0.99 구간이 가장 안정적인 성능을 보였다.

SiamRPN++의 경우, 마찬가지로 α을 조정한 모든 설정에서 지연시간은 388~399ms 수준으로 약 115ms 이상 개선 효과가 나타났다. 정확도는 전반적으로 안정적인 성능(61.84%~62.15%)을 보였다.

종합적으로, α 값이 지나치게 작은 경우에는 과거 값을 더 많이 반영하여 경량화 모델 전환 로직이 덜 자주 실행되므로 정확도 유지에는 유리하지만, 상대적으로 높은 지연 시간이 발생한다. 세 모델 모두에서 α = 0.9가 지연 시간 감소와 정확도 유지 간 최적의 균형을 제공하는 것으로 확인되었다. 따라서 본 논문에서 제시한 모든 실험에서는 해당 설정을 기본값으로 채택하였다.

다) 스케줄링 방식의 영향 분석

본 실험에서는 *EMA* 기반 블록 수준 동적 전환 기법을 기반으로 한 실행 환경에서, 제안한 스케줄링 기법(Scheduled)이 추가로 미치는 성능 개선 효과를 분석하였다. 실험은 세 가지 실행 조건에서 수행되었다. Baseline은 이전 실험과 동일한 환경이며, Not Scheduled는 블록 수준 동적 전환 기법만 적용된 환경, Scheduled는 블록 수준 동적 전환 기법과 스케줄링이 모두 적용된 환경으로 구성된다.

[표 5-10]은 세 가지 조건에서 모델별 평균 실행시간을 비교한 결과를 제시한다. VGG16의 경우, Not Scheduled에서는 112.02ms(1.16×), Scheduled 조건에서는 지연 시간이 109.008 ms(1.19×)로 추가로 감소하였다. Faster R-CNN의 경우, Baseline의 1979.12ms 대비 Scheduled에서는

1400.05ms(1.41×)까지 크게 감소하였다. SiamRPN++ 역시, Baseline의 515.77ms에서 Not Scheduled 441.08ms(1.17×), Scheduled 401.70ms(1.28×)로 점진적인 개선이 나타났다. 이러한 결과는 블록 수준 동적 전환 기법만으로는 자원 경합으로 인한 병목을 충분히 해소하기 어렵고, 제안한 스케줄링 기법을 함께 적용할 때 성능 향상 효과가 극대화될 수 있음을 의미한다.

HANSUNG
UNIVERSITY

제 6 장 결론

본 논문에서는 DNN 모델을 효율적으로 실행시키기 위한 두 가지 방법론을 제시하였다. 첫째, 서버 시스템 환경에서 분류 모델에 이어 검출,추적 모델까지 확장하여 Pruning과 Quantization을 단독 혹은 조합하여 적용하고, 다양한 비트 폭과 Pruning 적용 비율을 구성하여 모델 크기 감소율, 파라미터 수 변화, 정확도 손실을 분석하였다. 그 결과, 두 기법을 병행 적용한 경우 단독 적용 대비 더 높은 구조적 축소 효과를 보였다. 또한 모델 구조 변경 없이 추론 과정을 최적화하는 TensorRT 기반 런타임 최적화 기법을 적용하여, 추가적인 연산 그래프 최적화와 커널 융합을 통해 기존 경량화 기법만으로는 확보하기 어려운 추론 속도 향상과 메모리 효율 개선 효과를 입증하였다.

둘째, 임베디드 시스템 환경에서의 다중 DNN 병렬 실행 최적화를 위해 블록 단위 동적 스케줄링 및 블록 수준 동적 전환 기법을 제안하였다. 제안 기법은 모델을 기능적 블록 단위로 분할한 뒤, 병렬 실행 시 오히려 지연을 악화시키는 블록을 자동으로 식별하여 순차 실행으로 전환한다. 또한 각 블록의 지연 경향을 *LAG* 지표로 정량화하여, 실행 지연이 예상되는 블록을 런타임에 경량화된 블록으로 교체함으로써 지연 시간과 정확도 간의 균형을 실시간으로 유지할 수 있도록 하였다. 실험에서 이질적인 다중 DNN을 동시에 실행한 실험 결과, 제안 기법은 최대 29.3%의 지연 시간 감소와 기준 정확도의 90% 이상 유지할 수 있는 성능을 달성하였다.

또한 해당 프레임워크는 학습 데이터셋이나 모델 아키텍처에 종속되지 않고 범용적인 환경에서 동작하도록 설계되었다. 모든 대상 모델은 추가적인 파인튜닝 없이 사전학습된 네트워크를 기반으로 하며, 경량화된 변형 모델 역시 정확도 저하를 보완하기 위한 최소한의 파인튜닝만을 수행한다. 이에 따라 특정 데이터셋에 대한 의존성이 낮으며, 다양한 입력 분포에서도 안정적인 성능을 유지할 수 있다. 본 시스템은 TorchScript와 LibTorch를 활용한 C++ 환경에서 구현되었기 때문에, 새로운 모델이나 데이터셋으로의 확장이 용이하다. 새로운 모델을 통합하기 위해서는 대응되는 TorchScript 모듈(.pt 파일)과

데이터 전처리 함수를 추가하기만 하면 되며, 별도의 추가 학습이나 모델 구조 수정은 필요하지 않다. 다만 배포 전에 MUSCO 기반 저랭크 분해와 같은 사전 압축 과정은 반드시 수행해야 한다. 또한 모델의 구조적 특성이나 데이터 분포에 따라 θ_{trend}, EMA 평활화 계수(α,β), 블록 선택 기준과 같은 일부 제어 파라미터는 재설정이 필요할 수 있다. 또한 Conflict Table은 Level 0 블록의 실행시간을 기반으로 오프라인에서 측정 및 구성되며, 제안된 프레임워크는 미들웨어 수준에서 여러 DNN의 실행을 관리하기 때문에 NVIDIA JetPack 버전 또는 디바이스 드라이버 변경이 전체 스케줄링 구조 및 성능에 미치는 영향은 매우 제한적이다.

향후 연구에서는 세 개 이상의 DNN을 동시에 실행하는 확장된 시나리오에서 제안된 프레임워크의 확장성을 정량적으로 검증할 계획이다. 이를 위해 부족한 GPU 메모리를 스토리지 공간과 연동하여, DNN 모델이 필요 시 메모리를 동적으로 활용할 수 있는 구조를 구성할 예정이다. 이 경우, 스토리지 접근이 빈번해짐에 따라 발생하는 I/O 오버헤드를 효과적으로 은닉하기 위해, GPU 커널 실행과 스토리지 I/O 연산을 동시에 수행하는 기법을 함께 연구할 것이다. 또한, Conflict Table에서 병렬 실행 가능(OK)으로 기록된 블록 조합의 성능이 시스템 환경 변화로 인해 저하될 경우, 스케줄러는 일시적으로 해당 블록을 순차 실행으로 전환하도록 설계할 예정이다. 이후 동일한 성능 저하가 반복적으로 관찰되면, 강화학습 기반 온라인 보정 기법을 통해 해당 블록 조합을 순차 실행(NOT OK)으로 재분류한다. 이러한 적응형 접근 방식은 정적으로 구성된 Conflict Table의 한계를 보완하고, 시스템이 동적으로 변화하는 환경에 스스로 적응할 수 있도록 해줄 것으로 기대된다.

참 고 문 헌

1. 국외문헌

Bateni, S., & Liu, C. (2020). NeuOS: A Latency-Predictable Multi-Dimensional Optimization Framework for DNN-driven Autonomous Systems. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), 371-385.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhao, J., & Zieba, K. (2016). End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316.

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition 248-255.

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., & Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. Advances in neural information processing systems, 1, 1269-1277.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). CARLA: An open urban driving simulator. In Proceedings of the 1st Annual Conference on Robot Learning (CoRL 2017), 78, 1-16.

Everingham, M., Van Gool, L., Williams, C. K., Winn, J., & Zisserman, A. (2010). The pascal visual object classes (voc) challenge. International journal of computer vision, 88(2), 303-338.

Grigorescu, S., Trasnea, B., Cocias, T., & Macesanu, G. (2020). A survey of deep learning techniques for autonomous driving. Journal of field robotics, 37(3), 362-386.

Gusak, J., Kholiavchenko, M., Ponomarev, E., Markeeva, L., Blagoveschensky, P., Cichocki, A., & Oseledets, I. (2019). Automated multi-stage compression of neural networks. In Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops 2501-2508.

Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition 770-778.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., & Chen, W. (2022). Lora: Low-rank adaptation of large language models. ICLR, 1(2), 3.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2704-2713).

Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., & Tang, L. (2017). Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. ACM SIGARCH Computer Architecture News, 45(1), 615-629.

Kim, Y. D., Park, E., Yoo, S., Choi, T., Yang, L., & Shin, D. (2015). Compression of deep convolutional neural networks for fast and low power mobile applications. In Proceedings of the International Conference on Learning Representations (ICLR).

Kim, M., Kim, I., Yong, J., & Kim, H. (2023). Scheduling framework for accelerating multiple detection-free object trackers. Sensors, 23(7),

3432.

Li, B., Wu, W., Wang, Q., Zhang, F., Xing, J., & Yan, J. (2019). Siamrpn++: Evolution of siamese visual tracking with very deep networks. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 4282-4291.

Li, E., Zeng, L., Zhou, Z., & Chen, X. (2019). Edge AI: On-demand accelerating deep neural network inference via edge computing. IEEE transactions on wireless communications, 19(1), 447-457.

Lin, J., Rao, Y., Lu, J., & Zhou, J. (2017). Runtime neural pruning. Advances in neural information processing systems, 30.

Liu, Z., Liu, Q., Xu, W., Wang, L., & Zhou, Z. (2022). Robot learning towards smart robotic manufacturing: A review. Robotics and Computer-Integrated Manufacturing, 77, 102360.

Muller, M., Bibi, A., Giancola, S., Alsubaihi, S., & Ghanem, B. (2018). Trackingnet: A large-scale dataset and benchmark for object tracking in the wild. In Proceedings of the European conference on computer vision (ECCV) ,300-317.

Nakajima, S., Sugiyama, M., Babacan, S. D., & Tomioka, R. (2013). Global analytic solution of fully-observed variational Bayesian matrix factorization. The Journal of Machine Learning Research, 14(1), 1-37.

Niu, W., Guan, J., Wang, Y., Agrawal, G., & Ren, B. (2021, June). Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (pp. 883-898).

Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W. & Dally, W. J. (2017). SCNN: An accelerator for compressed-sparse convolutional neural networks. ACM SIGARCH computer architecture news, 45(2), 27-40.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in neural information processing systems, 28.

Satyanarayanan, M. (2017). The emergence of edge computing. Computer, 50(1), 30-39.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Wu, Y., Lim, J., & Yang, M. H. (2013). Online object tracking: A benchmark. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2411-2418).

Yao, Z., Cao, S., Xiao, W., Zhang, C., & Nie, L. (2019, July). Balanced sparsity for efficient dnn inference on gpu. In Proceedings of the AAAI conference on artificial intelligence, 33(1), 5676-5683.

Zanella, A., Bui, N., Castellani, A., Vangelista, L., & Zorzi, M. (2014). Internet of things for smart cities. IEEE Internet of Things journal, 1(1), 22-32.

HANSUNG UNIVERSITY

ABSTRACT

Lightweighting Techniques and Block-Level Scheduling for Efficient Multi-DNN Execution

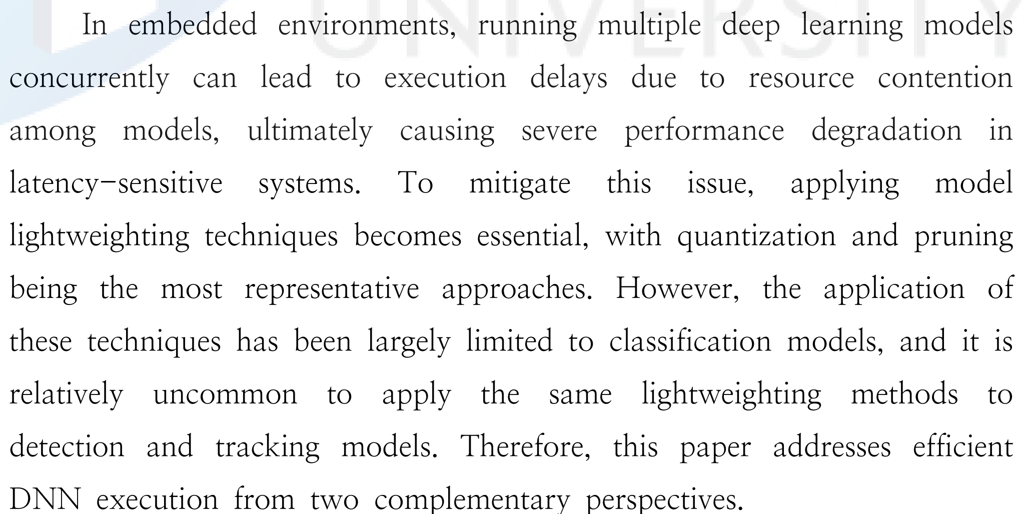
Kim, Hyuk-Soo

Major in Applied Artificial Intelligence

Dept. of Applied Artificial Intelligence

The Graduate School

Hansung University



In embedded environments, running multiple deep learning models concurrently can lead to execution delays due to resource contention among models, ultimately causing severe performance degradation in latency-sensitive systems. To mitigate this issue, applying model lightweighting techniques becomes essential, with quantization and pruning being the most representative approaches. However, the application of these techniques has been largely limited to classification models, and it is relatively uncommon to apply the same lightweighting methods to detection and tracking models. Therefore, this paper addresses efficient DNN execution from two complementary perspectives.

First, in a server-based environment, pruning and quantization were applied individually and in combination to various vision models to analyze changes in model size, parameter count, and accuracy.

Experimental results show that combining the two lightweighting techniques yields greater parameter reduction than using either technique alone, while keeping accuracy degradation minimal.

Second, to minimize execution delays in multi-DNN concurrent environments, we propose a block-level dynamic scheduling and block-level dynamic replacement technique. In this method, each model is divided into functional units called blocks, which serve as the fundamental execution units. The scheduler identifies blocks that cause additional latency when executed in parallel and selectively switches them to sequential execution. Moreover, using a metric that quantifies the execution delay of each block, blocks expected to incur significant latency are dynamically replaced at runtime with lightweight alternatives to maintain a real-time balance between latency and accuracy. Experiments conducted on a representative embedded platform, the NVIDIA AGX Jetson Xavier, show that the proposed method achieves up to a 29.3% reduction in latency while preserving more than 90% of the baseline accuracy when executing heterogeneous DNNs simultaneously.

【Keywords】Embedded Deep Learning, LAG, EMA, Model Compression, Block Switching, Multi-DNN Scheduling