석사학위논문

임베디드 시스템을 위한 소프트웨어 기반 적대적 공격 대응 연구

2023년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

주 상 현

석사학위논문
지도교수 김명선

임베디드 시스템을 위한 소프트웨어 기반 적대적 공격 대응 연구

Research on Software-Based Adversarial Attack Defense for Embedded Systems

2022년 12월 일

한성대학교 대학원

IT융합공학과

IT융합공학전공

주 상 현

석 사 학 위 논 문
지도교수 김명선

임베디드 시스템을 위한 소프트웨어 기반 적대적 공격 대응 연구

Research on Software-Based Adversarial Attack Defense for Embedded Systems

위 논문을 공학 석사학위 논문으로 제출함

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

주 상 현

주상현의 공학 석사학위 논문을 인준함

2022년 12월 일

심사위원장 오 희 석 (인)

심 사 위 원 이 웅 희 (인)

심 사 위 원 김 명 선 (인)

국 문 초 록

임베디드 시스템을 위한 소프트웨어 기반 적대적 공격 대응 연구

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
주 상 현

최근 DNN의 분류 결과를 고의적으로 잘못된 분류 결과가 도출되도록 유도하는 적대적 공격의 종류가 다양해지고 정교해지고 있다. 이에 DNN들은 적대적 공격에 노출이 잦아졌으며 드론이나 자율 주행 시스템과 같은 임베디드 시스템에서의 적대적 공격으로 인한 오분류의 결과는 치명적일 수 있다. 그러나 임베디드 시스템은 DNN 연산 하드웨어의 성능과 메모리 용량이 제한되어 있기 때문에 적대적 공격을 빠른 속도로 판별하거나 방어하기 매우 어렵다. 이를 위해 전용 하드웨어를 개발하면 비용 측면에서의 부담이 있고 공격 방식과 타겟 DNN의 변경에 유연하게 대처하기 어려움이 있다. 이런 문제를 해결하고자 본 논문에서는 임베디드 시스템에서 적대적 공격 탐지와 적대적 예제 복원을 통한 방어를 할 수 있도록 각각의 기법을 소개한다. 적대적 공격 탐지를 위해 소프트웨어 기반 적대적 공격 검출 기법인 임베디드 NIC를 제안한다. 이 기법은 공격 검출에 필요한 메모리를 최소화하기 위해 타겟 DNN의 은닉층 중 검출을 진행할 은닉층을 선별한다. 또한 타겟 DNN

추론이 진행될 때 병렬적으로 검출하며 이 둘 간의 실행시간 격차를 최소화한다. 적대적 예제 복원을 위해서는 eDenoizer를 제안한다. eDenoizer는 적대적 예제의 잡음(adversarial perturbation)을 제거하는 DNN의 컨볼루션 커널 텐서에 필요한 연산량을 터커 분해를 적용하여 줄인다. 또한 다른 DNN들과 동시에 복원을 진행할 때 CPU측에서 높게 설정한 잡음 제거를 진행하는 DNN과 타겟 DNN의 우선순위를 GPU로 전달하여 적대적 공격 방어를 우선적으로 실행할 수 있다. 실험 결과 임베디드 NIC는 적용 전 대비 실행시간의 차이가 최대 99.6% 감소하고 83.9%의 메모리 사용량 감소를 나타내며 eDenoizer는 1.78%의 미미한 분류 정확도 감소와 함께 적대적 예제 복원이 수반된 추론 속도는 51.72% 단축된다.

【주요어】 적대적 공격 탐지, 적대적 공격 방어, 임베디드 시스템

목 차

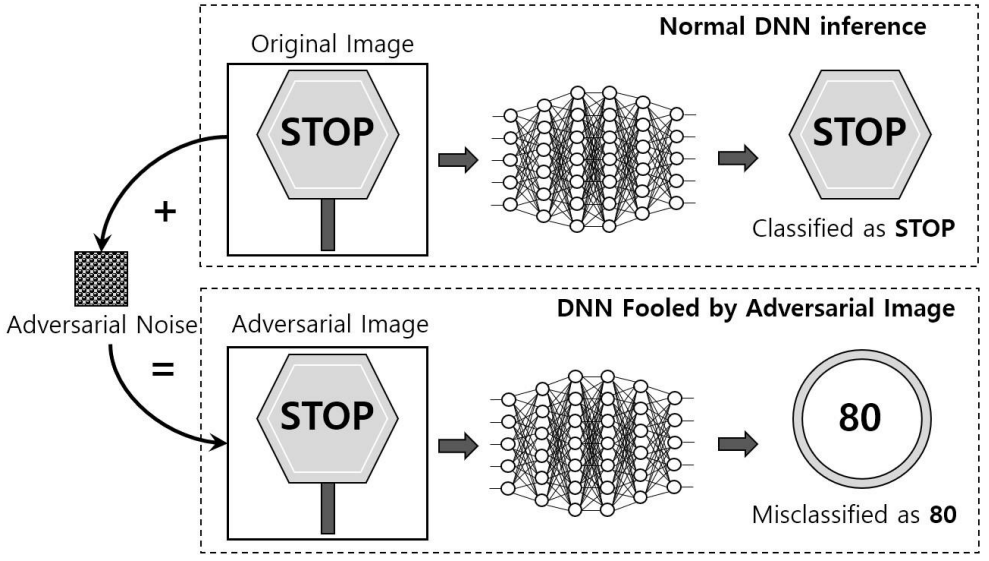
표 목 차

그 림 목 차

제 1 장 서론

딥 러닝 기술로 발전된 인공지능 서비스가 일상을 더욱 편리하게 만들고 있다. DNN(딥 러닝 모델)을 기반으로 한 인공지능 애플리케이션은 의료 이미지 분석(Singh, A., et al, 2020), 건설(Rashid, K. M., et al, 2019), 자율 주행 자동차(Bojarski, M., et al, 2016), 메타버스 기반 교육(Zhu, H., et al, 2022) 등 다양한 중요 분야에서 활발하게 활용되지만 딥 러닝 기술은 적대적 공격에 취약하다. 적대적 공격하기 위한 적대적 예제는 DNN의 기능을 상실하게 하며 원본 데이터에 인간의 눈으로 식별할 수 없는 매우 적은 양의 잡음을 추가하는 것으로 생성된다(Su, J., et al, 2019; Carlini, N., 2017; Cheng, S., 2019). 또한 자율 주행 자동차 및 침습 수술과 같이 보안에 민감한 애플리케이션에서 이러한 적대적 공격은 끔찍한 결과를 초래할 수 있다. [그림 1-1]은 자율 주행 차량에 대한 적대적 공격의 예시이다. 공격자는 카메라로부터의 데이터를 가로채어 DNN 기반 물체 감시 시스템으로 전달되기 전에 잡음(adversarial perturbation)을 추가하여 데이터를 교란할 수 있다. [그림 1-1]은 정상 데이터인 STOP에 잡음을 추가하여 DNN의 추론 결과가 시속 80km로 주행하라고 도출되는 상황이다. 자율 주행 시 [그림 1-1]과 같은 적대적 공격이 가해지면 위험이 커진다. 또한 도로 표지판을 조작함으로써 DNN 수행에 혼란을 주어 분류 결과가 잘못 도출될 수 있다. 이에 애플리케이션의 기능을 안전히 수행하기 위해 실행 환경에 적절하고 강력한 적대적 공격 대응 기법이 필요하다.

[그림 1-1]의 자율 주행 차량과 같은 애플리케이션은 임베디드 시스템 상에서 실행된다. 임베디드 시스템은 크기가 작고 휴대하기 간편하며 소비 전력이 낮아 자율 주행 차량, 드론, 그리고 로봇 등에 많이 활용



[그림 1-1] 자율 주행 차량에서의 적대적 공격 예시

된다. 이러한 임베디드 시스템은 다수의 GPU를 사용하는 고성능 하드웨어를 갖춘 서버 컴퓨팅 대비 대부분 단일 임베디드 GPU를 사용하는 임베디드 시스템은 저성능이며 메모리가 제한적이다. 임베디드 시스템이 활용되는 예시를 보면 알 수 있듯이 DNN이 임베디드 시스템에서 많이 사용되는 것을 확인할 수 있고 이는 임베디드 시스템 또한 적대적 공격의 타겟 시스템이 될 수 있음을 의미한다.

적대적 공격이 발전됨에 따라 적대적 공격을 대응하는 기술 또한 여럿 제안되었으며 성능 향상이 이뤄졌다. 적대적 공격을 대응하는 기술은 적대적 공격 탐지와 적대적 공격 방어로 나눌 수 있으며 방어 기법 중에서 적대적 예제를 정상 데이터로 복원하는 기술이 높은 성능을 보여 보다 안전하다.

(Ma, S., et al, 2019)에서는 여러 종류의 DNN에 가해지는 다수의 적대적 공격을 높은 확률로 탐지한다. (Ma, S., et al, 2019)에서는 각각의 DNN에 존재하는 모든 은닉층에서 적대적 공격 탐지를 진행한다. 적대적 공격을 탐지하기 위해 은닉층 별로 one-class SVM(Scholkopf, B., et al, 2001) 2개와 얕은 DNN이 별도로 존재하며 DNN의 추론이 진행되는 동안

동시에 각 은닉층에서 출력 벡터가 도출되면 이를 통해 one-class SVM과 얕은 DNN의 연산이 같이 실행되어 적대적 공격 탐지가 진행된다.

(Liao, F., et al, 2018)에서는 디노이징 오토인코더(denoising autoencoder)(Vincent, P., et al, 2008)와 U-net(Ronneberger, O., et al, 2015)을 결합하여 적대적 예제에 있는 잡음을 제거하는 필터를 생성한 후 적대적 예제와 합쳐 정상 데이터로 복원하는 DNN을 통해 적대적 예제 복원을 진행한다. 적대적 예제 복원은 적대적 공격을 방어하는 수단이므로 적대적 예제가 추론되려고 했던 DNN에 적대적 예제가 입력되기 전에 정상 데이터로 복원하는 DNN에 선입력 되어 복원 후에 기존에 추론되려고 했던 DNN에 입력된다.

적대적 공격은 DNN을 공격 타겟으로 하고 DNN은 주로 GPU에서 연산되기 때문에 적대적 공격 대응도 GPU에서 연산되는 방식이 다수이다. 그러나 이러한 적대적 공격 대응 방식들은 높은 성능을 목적으로 개발되었기 때문에 메모리 사용률과 GPU 리소스 활용률이 고려되지 않고 개발되었다. 이로 인해 적절한 시간 안에 시스템 실행을 완료해야 하고 성능과 메모리가 제한적인 임베디드 시스템에서 적대적 공격 대응을 적용하기 어렵다.

(Ma, S., et al, 2019)에 의한 적대적 공격 탐지는 각 은닉층의 연산이 완료된 후 그 결과로 적대적 공격 탐지를 진행하기 때문에 DNN 추론보다 늦게 완료된다. 이를 임베디드 시스템에서 실행할 경우 제한된 하드웨어로 인해 DNN의 추론 소요 시간과 적대적 공격 탐지 소요 시간의 차이는 더 늘어날 수밖에 없다.

적대적 공격은 DNN의 분류 결과를 조작하여 치명적인 결과를 초래할 수 있으므로 (Liao, F., et al, 2018)을 활용한 적대적 예제 복원이 실질적으로 임베디드 시스템에서 적용되기 위해서는 무엇보다 빠르고 우선적으로 처리되어야 한다. 그러나 높은 분류 정확도를 보이는 DNN들이 많은 연산량(multiply-accumulate)을 요구하는 것처럼 잡음 제거에 필요한 필터를 생성하

는 DNN 또한 많은 연산량이 필요하다. 또한 GPU는 비선제적 특징을 가지는 하드웨어이므로 여러 시스템이 동시에 단일 임베디드 GPU를 사용할 때 적대적 예제 복원을 우선적으로 실행할 수 없다. 이에 고성능 하드웨어가 아닌 임베디드 시스템에서 우선적이고 빠르게 복원하기 어렵다.

본 논문에서는 적대적 공격 탐지 기법과 적대적 예제 복원 기법을 임베디드 시스템에 실질적으로 적용하기 위한 각각의 솔루션을 제안한다. 임베디드 시스템에서 적대적 공격 탐지 시 DNN 추론 종료 시간과 적대적 공격 탐지 종료 시간의 차이를 소프트웨어 기반 솔루션을 통해 감소시키고자 한다. 적대적 예제 복원은 실질적으로 적용할 시 GPU를 사용하는 다른 애플리케이션들보다 우선적으로 처리되어야 하고 복원하는 연산 자체가 빠르게 이뤄져야 한다. 이를 위해 적대적 예제 복원에 필요한 GPU 연산들을 우선적으로 처리할 수 있는 프레임워크를 제안하고 복원 작업을 하는 DNN의 연산량을 정확도 하락이 거의 없게 감소시켜 적대적 예제 복원을 가속한다.

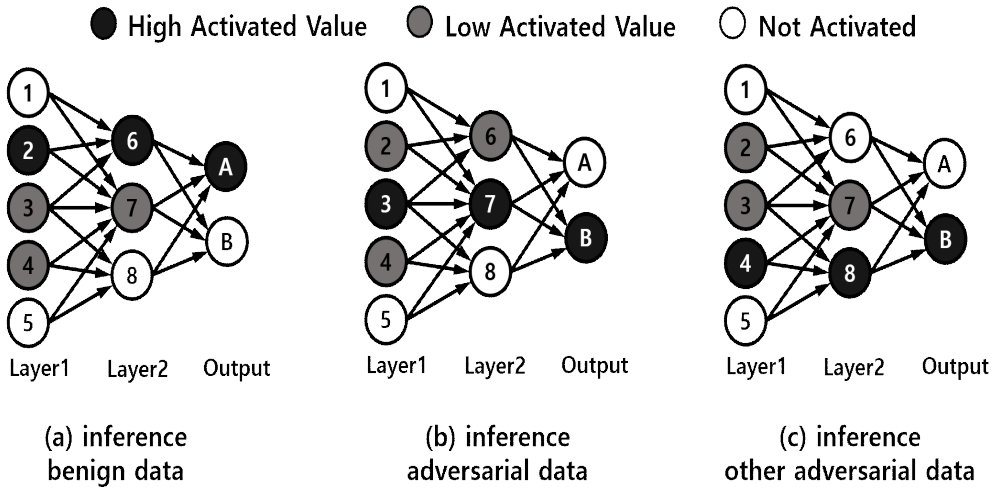
제 2 장 연구 배경 및 문제점

제 1 절 적대적 공격 대응

1) 적대적 공격 탐지

적대적 공격이란 영상에 사람의 눈으로는 확인하기 어려운 잡음(adversarial perturbation)을 추가하여 DNN이 원래와는 다른 분류 결과를 도출하도록 적대적 예제(Adversarial example)(Szegedy, C., et al, 2013)를 생성하는 것이다. 적대적 공격 탐지란 타겟 DNN에 입력되는 데이터가 추론될 때 입력된 데이터가 적대적 공격을 받은 적대적 예제인지 아닌지를 판별하는 것을 의미한다. 적대적 공격의 종류가 늘어나고 공격이 강력해진 만큼 적대적 공격 탐지 또한 발전되어 다양한 적대적 공격 탐지 방법이 존재한다. 본 논문은 Neural Network Invariant Checking(Ma, S., et al, 2019)(이하 NIC)를 활용하는 탐지 시스템을 타겟으로 한다. [그림 2-1]은 NIC를 적용하여 적대적 기법을 탐지하는 기법을 소개한다. [그림 2-1]의 (a)는 은닉층1(Layer1)과 은닉층2(Layer2) 그리고 A 또는 B로 분류하는 출력층(Output)을 가지는 DNN에 정상 데이터가 입력되었을 때를 나타내며 [그림 2-1]의 (b),(c)는 각기 다른 적대적 예제가 입력되었을 때를 나타낸다. [그림 2-1]의 (a)는 정상 데이터가 A로 알맞게 분류되었지만 [그림 2-1]의 (b),(c)에서는 A로 분류되어야 하지만 B로 잘못 분류된 것으로 표현했다.

NIC는 타겟 DNN이 정상 데이터를 추론했을 때 각 은닉층의 활성화 값들을 해당 은닉층의 Value Invariants(이하 VI)라 하며 각각 연속된 두 은닉층의 활성화 뉴런들을 해당 은닉층들의 Provenance Invariants(이하 PI)라고 한다. [그림 2-1]의 (a)에서 은닉층1의 활성화 뉴런은 2,3,4이므로 은닉층1의 VI는 2,3,4의 활성화 값들이며 은닉층2의 VI는 활성화 뉴런 6,7의 활성화 값들이다. 연속된 두 은닉층인 은닉층1과 은닉층2에서의 PI는 활성화 뉴런 2,3,4,6,7들이다.



[그림 2-1] DNN 추론 시 활성화 되는 뉴런과 각 뉴런의 활성화 값

NIC에서 입력 데이터를 타겟 DNN에 추론했을 때 각 은닉층 활성화 뉴런들의 활성화 값들이 해당 은닉층의 VI와 다르거나 연속된 두 은닉층 각 쌍의 활성화 뉴런들이 해당 은닉층의 PI와 다른 경우 입력된 데이터를 적대적 공격을 받은 데이터라고 판단한다.

타겟 DNN에서 각 은닉층의 VI와 각각 연속된 두 개의 은닉층들의 PI를 구하는 방법을 [그림 2-1]의 DNN을 통해 알아보면 다음과 같다. 은닉층1의 VI와 은닉층2의 VI는 DNN이 정상 데이터를 추론했을 때 도출되는 각 은닉층의 출력 벡터들이다. 즉 은닉층1의 VI는 은닉층1의 출력 벡터이며 은닉층2의 VI는 은닉층2의 출력 벡터이다. 은닉층1과 은닉층2의 PI를 구하기 위해서 Derived Model(이하 DM)이라는 DNN이 두 개 필요하다. 즉, DM은 타겟 DNN의 은닉층 마다 존재한다. [그림 2-1]의 (a)에서 은닉층1의 DM이란 DNN의 입력층부터 은닉층1 뒤에 새로운 소프트맥스 계층을 이어 붙인 DNN이고 은닉층2의 DM은 DNN의 입력층부터 은닉층2 뒤에 새로운 소프트맥스 계층을 이어붙인 DNN이다. 은닉층1의 DM과 은닉층2의 DM의 각 소프트맥스 출력 차원은 [그림 2-1]의 (a)에 있는 DNN의 소프트맥스 출력 차원과 동일하다. 두 DM의 출력 벡터를 컨캣(concatenate)하면 은닉층1과 은닉층2의 PI를 얻게 된다.

NIC는 One-Class SVM(scholkopf, B., et al, 2001)(이하 OCSVM)을 사

용하여 타겟 DNN 모든 은닉층의 VI와 PI를 훈련한다. 각 은닉층의 VI와 PI는 정상 데이터로부터 얻으므로 VI와 PI의 OCSVM의 훈련 데이터로는 정상 데이터만을 사용한다. 타겟 DNN에 정상 데이터를 추론하여 각 은닉층의 출력을 구한 다. 각 은닉층의 출력은 각 은닉층 VI의 훈련 데이터가 되고 각 연속된 두 개의 DM의 출력은 해당 은닉층들 PI의 훈련 데이터가 된다. 각 VI와 PI의 훈련 데이터로 은닉층 마다 OCSVM을 훈련한다. 즉, 은닉층마다 VI 탐지용 OCSVM(이하 VI-SVM)과 PI 탐지용 OCSVM(이하 PI-SVM)이 존재한다.

2) 적대적 예제 복원

적대적 예제 복원은 적대적 공격 방어 기법 중 하나이며 적대적 예제가 적대적 예제를 정상 데이터로 복원하는 것을 의미한다. 본 논문은 적대적 예제 복원 기법 중 High-level representation Guided Denoiser(Liao, F., et al, 2018)(이하 HGD)를 타겟 시스템으로 정했다. HGD에서는 적대적 예제가 타겟 DNN에 추론되기 전에 정상 데이터로 복원하기 위해 Denoizing AutoEncoder(DAE)(Vincent, P., et al, 2008)와 U-net(Ronneberger, O., et al, 2015)을 결합한 구조를 지닌 DUNET을 소개한다. HGD는 적대적 공격 방어를 하기 위해 DUNET을 통한 적대적 예제의 복원을 타겟 DNN 실행 전에 진행한다. HGD에서는 타겟 DNN에 적대적 예제와 정상 데이터를 각각 추론했을 때 타겟 DNN의 출력층에 가까울수록 적대적 예제와 정상 데이터 간의 차이가 커지는 것을 실험으로 분석했다. 따라서 DUNET의 손실함수로써 정상 데이터와 적대적 예제를 각각 타겟 DNN에 추론한 후 출력층과 가까운 은닉층의 출력의 차이를 L_1 norm 연산을 사용한다. 이때 출력층과 가까운 은닉층이란 소프트맥스 계층의 첫 번째 이전 계층 또는 두 번째 이전 계층을 의미하며 각각의 계층의 출력으로 학습을 진행 HGD를 Logits Guided Denoiser(이하 LGD)와 Feature Guided Denoiser(이하 FGD)라고 명한다.

제 2 절 임베디드 시스템에서의 적대적 공격 대응 적용 문제점

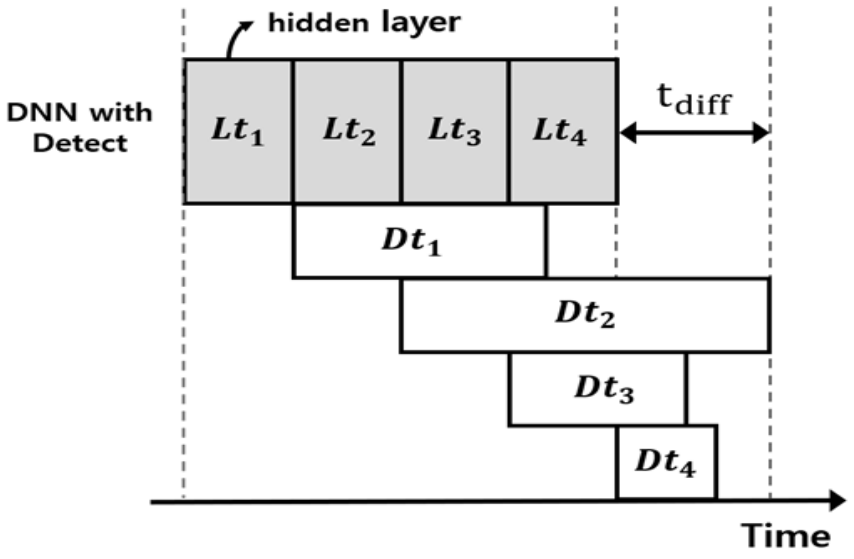
임베디드 시스템은 서버 컴퓨팅과 달리 시스템을 설계할 때 특정한 기능들을 실행할 목적으로 만들어지는 시스템이다. 이러한 이유로 임베디드 시스템의 하드웨어는 특정한 기능들을 실행할 수 있을 정도로의 최소한의 하드웨어로 구성되어 있어 여러 GPU들을 클러스터로 사용하는 서버 컴퓨팅과 달리 임베디드 시스템은 대부분 단일 GPU를 사용하며 메모리와 전력 등의 제약이 있다. 또한 임베디드 시스템은 특정한 기능들을 적절한 시간 안에 실행 완료해야하는 특성이 필요하다. 1절에서 소개한 두 개의 적대적 공격 대응 방식을 메모리, 프로세서가 제약되고 제한 시간 안에 동작이 완료되어야 하는 임베디드 시스템에 적용하기에는 각각의 문제점이 있다. 본 절에서는 각 적대적 공격 대응을 임베디드 시스템에 적용할 때 발생하는 문제점을 서술한다.

1) 임베디드 시스템에서의 NIC 문제점

가) 타겟 DNN 추론과 NIC 탐지의 소요시간 차이

[그림 2-2]는 멀티 프로세스 환경에서 4개의 은닉층으로 구성된 타겟 DNN에 NIC를 적용하여 타겟 DNN의 추론 시간과 적대적 공격 탐지 소요 시간을 나타낸 것이다. $Lt_1 \sim Lt_4$은 타겟 DNN의 각 은닉층 추론 시간을 의미하며 각 은닉층의 추론이 완료된 후 진행되는 적대적 공격 탐지 소요 시간은 $Dt_1 \sim Dt_4$로 나타냈다. t_{diff}는 타겟 DNN의 적대적 공격 탐지가 완료될 때까지의 소요 시간과 타겟 DNN의 추론 소요 시간의 차이이며 각 은닉층에서의 적대적 공격 탐지 소요 시간 중 가장 늦게 완료되는 탐지 소요 시간이 늘어날수록 t_{diff}가 증가한다.

$Dt_1 \sim Dt_4$ 각각은 VI-SVM 또는 PI-SVM 모델의 크기가 클수록 증가



[그림 2-2] NIC를 적용한 타겟 DNN의 추론 소요시간과 적대적 공격 탐지 소요시간의 차이

한다. [그림 2-2]에서는 각 은닉층의 탐지 소요시간 중 Dt_2가 가장 기므로 타겟 DNN의 두 번째 은닉층의 VI-SVM 또는 PI-SVM의 크기가 가장 큰 모델을 가진다.

타겟 DNN 추론 중에 NIC를 적용한 적대적 공격 탐지를 진행할 때, 타겟 DNN의 각 은닉층 추론 완료 후에 해당 은닉층에서의 적대적 공격 탐지가 가능하기 때문에 NIC를 적용한 타겟 DNN 추론에서 t_{diff}발생은 필연적이다. 타겟 DNN은 추론 완료 후 t_{diff}동안 입력 데이터에 대한 추론 결과를 신뢰할 수 없으므로 t_{diff}는 작을수록 좋다. 그러나 임베디드 시스템에서 [그림 2-2]과 같이 타겟 DNN에 NIC를 적용하여 추론을 진행하면 서버에서 실행했을 때 대비 각 은닉층의 적대적 공격 탐지 소요 시간이 증가하므로 t_{diff}가 증가할 수밖에 없는 문제점이 발생한다.

나) NIC의 메모리 요구량

OCSVM 모델의 저장 시 서포트 벡터가 저장되고 서포트 벡터의 차원은 OCSVM의 훈련 데이터 차원과 동일하기 때문에 OCSVM 모델의 크기는 훈련 데이터의 차원이 커지거나 저장되는 서포트 벡터의 개수가 늘어날수록 커진다. 타겟 DNN에서 모든 PI-SVM들은 훈련 데이터의 차원이 같으므로 PI-SVM 모델의 크기에 영향을 주는 것은 서포트 벡터의 개수이다. 반면에 VI-SVM의 훈련 데이터는 타겟 DNN의 각 은닉층 출력 벡터이므로 타겟 DNN 각각의 VI-SVM 모델의 크기는 훈련 데이터의 차원과 서포트 벡터의 개수에 영향을 받을 수밖에 없다. PI-SVM들의 훈련 데이터 차원은 타겟 DNN의 훈련 데이터 셋에 의해 정해지고 대부분의 VI-SVM들의 훈련 데이터 차원에 비해 월등히 작으므로 PI-SVM 모델들의 크기는 대부분의 VI-SVM 모델들의 크기보다 작다. 예시로 본 논문의 실험에서 타겟 DNN 중 하나로 사용된 MobileNetV1의 PI-SVM 모델 크기가 가장 큰 것은 792KB이지만 VI-SVM 모델 크기가 가장 큰 것은 2.4GB이다.

NIC는 타겟 DNN의 입력 데이터가 VI와 PI를 위반하는지 타겟 DNN의 모든 은닉층에서 확인하는 것으로 적대적 공격 탐지를 수행한다. 그러므로 타겟 DNN의 은닉층 개수만큼 VI-SVM, DM, 그리고 PI-SVM이 필요하다. 만일 타겟 DNN의 총 은닉층 개수가 100개라고 가정하면 NIC를 적용한 적대적 공격 탐지에 필요한 OCSVM과 DNN이 VI-SVM 100개, DM 100개, PI-SVM이 99개로 총 299개가 필요하다. 또한 100개의 VI-SVM 중 1GB 이상의 큰 크기를 가지는 VI-SVM이 여러 개 존재할 수 있다. NIC의 이러한 OCSVM 기반 탐지 기법은 수 GB의 메모리가 필요하며 이는 메모리가 제한적인 임베디드 시스템에 적합하지 않다.

2) 임베디드 시스템에서의 HGD 문제점

가) 임베디드 시스템에서의 DUNET 추론 시간

DNN의 올바른 추론 결과를 보장하기 위해, 일반적으로 DNN 추론 전에 적대적 데이터의 잡음을 제거하는 절차가 수행된다. 이는 HGD도 마찬가지이

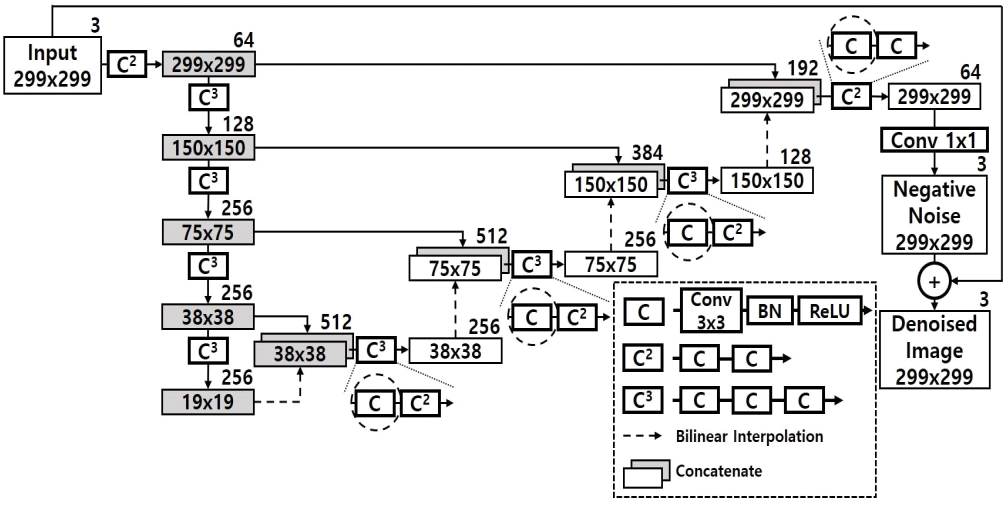
므로 DUNET의 실행 시간은 적대적 예제 복원 시스템 전체의 추론 성능 측면에서 중요하다.

	Inception-V3	ResNet-152	VGG-16	DUNET
Jetson AGX Xavier	86.3 ms	157.7 ms	50.4 ms	160.7 ms
Jetson TX2	103.9 ms	198.6 ms	76.8 ms	246.4 ms

[표 2-1] 단일 입력 데이터에 대한 DNN별 추론 시간 비교

[표 2-1]은 DUNET이 단일 입력 데이터를 추론하는데 걸리는 시간과 Inception-V3(Szegedy, C., et al, 2016), ResNet-152(He, K., Ren, S., et al, 2016) 그리고 VGG-16(Simonyan, K., et al, 2014) 세 개의 DNN들이 단일 입력 데이터를 추론하는데 걸리는 시간을 대표적인 임베디드 시스템인 Jetson AGX Xavier와 Jetson TX2(NVIDIA Jetson TX2 Developer Kit, online)에서 측정한 결과이다. [표 2-1]에서 확인할 수 있듯이 DUNET의 추론 시간은 다른 DNN에 비해 최대 3.2배 크다. 적대적 예제 복원 시스템은 먼저 적대적 예제가 정상 데이터로 복원된 후, 복원된 데이터가 타겟 DNN에 추론되는 방식이므로 Jetson AGX Xavier에서 적대적 예제 복원 시스템의 총 소요 시간은 DUNET의 추론 시간인 160.7 ms와 타겟 DNN인 Inception-V3의 추론 시간인 86.3 ms를 더한 247 ms이고 Jetson TX2에서는 약 350.3 ms가 소요된다.

[그림 2-3]은 DUNET의 구조를 나타낸다. 이미지와 피쳐맵(feature map)을 나타내는 사각형들은 각 사각형 내부에 이미지 또는 피쳐맵의 너비×높이를 나타내고 있으며 해당 사각형들의 오른쪽 외부 상단에는 채널수를 의미하는 숫자들이 적혀있다. C는 3×3 컨볼루션(convolution), 배치 노말라이제이션(Batch Normalization) 계층 그리고 rectified linear unit(ReLU)의 순서로 구성된 계층들을 의미한다. C^r는 C가 r번만큼 연속적으로 있는 것이며 C^2는 C가 2번, C^3는 C가 3번 연속적으로 있는 계층들이고 $Conv\,k\times k$은

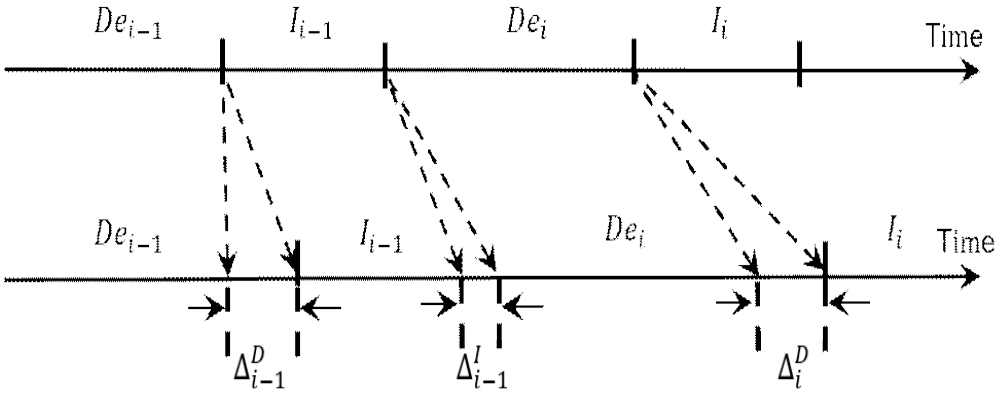


[그림 2-3] DUNET 구조

$k \times k$ 컨볼루션을 의미한다. DUNET의 최종 출력은 입력 데이터와 동일한 크기(너비×높이)와 채널수를 갖는 잡음이 제거된 이미지이다. DUNET은 복원된 이미지를 얻기 위해 먼저 입력 데이터와 같은 크기 및 채널수를 가지는 negative noise를 생성한 후 이를 입력 데이터와 더한다. 그림[2-3]에서 점선으로 그려진 원들은 각각 두 개 또는 세 개의 연속된 컨볼루션 계층의 첫 번째 컨볼루션 계층을 나타낸다. 각 컨볼루션 계층들의 출력 채널수는 256, 256, 128, 64개로 비교적 큰 연산이 진행되는 것을 알 수 있다. DUNET의 전체 MAC(Multiply-Accumulate) 연산 중 원으로 표시된 4 곳에서의 연산은 40.37%를 차지한다. 이에 DUNET의 정확도 손실을 최소화하고 추론 시간을 감소시키기 위해 4 곳의 컨볼루션 계층의 연산량을 효과적으로 줄여야 한다.

나) 임베디드 시스템 GPU 특성으로 인한 HGD 실행 지연

자율주행 자동차에도 사용되는 Jetson AGX Xavier는 딥 러닝 애플리케이션을 위해 설계된 대표적인 임베디드 시스템으로 8개의 ARM 코어와 내장 GPU를 갖추고 있다. 런타임 및 라이브러리(NVIDIA CUDA Toolkit-Free Tools and Training, online; NVIDIA CUDA Deep Neural



[그림 2-4] 시간에 따른 연속적 입력 데이터에 대한 DUNET의 적대적 예제 복원 및 타겟 DNN의 추론 과정

Network(cuDNN), online)와 같은 지원으로 애플리케이션 레벨 개발자는 딥 러닝 애플리케이션의 가속화를 GPU를 사용해 쉽게 적용할 수 있다. 이 GPU는 서버 컴퓨팅 시스템에 사용되는 GPU와 마찬가지로 하드웨어 수준의 멀티 스레딩을 지원한다. 또한 CUDA 스트림(Harris, M., online)을 지원하여 레지스터 및 SM(Streaming Multiprocessors)이 허용하는 한도 내에서 여러 프로세스에 의해 실행되는 딥 러닝 커널들을 동시에 처리할 수 있다.

이러한 우수한 병렬 처리가 가능함에도 불구하고 여러 딥 러닝 커널 런치가 동시에 발생할 경우, 단일 임베디드 GPU의 병목 현상은 불가피하다(Lim, C. & Kim, M., 2021). 또한 GPU는 본질적으로 비선점적이기 때문에, 적대적 예제 복원을 수행하는 DUNET에 높은 우선순위가 주어지더라도 다른 DNN이 GPU의 Execution Engine(EE) 큐에 먼저 할당되면 DUNET의 실행이 지연된다. 일반적으로 PyTorch(Pytorch, online), TensorFlow(Tensorflow, online)와 같은 프레임워크에서 생성된 DNN은 배치 단위로 처리되어 실행된다(Xiang, Y. & Kim, H., 2019). 즉, 실행 단위는 전체 DNN이며 필연적으로 헤드오브라인 블록킹(head-of-line blocking) 문제가 발생한다(Karol, M., et al, 1987).

다) 적대적 예제 복원 시간 측정

[그림 2-4]는 시간 흐름에 따라 HGD가 지연되지 않으면서 실행될 때와 지연되면서 실행될 때를 나타낸 것이다. 위의 시간 축은 HGD가 지연 없이 실행될 때를 나타내는 반면 밑의 시간 축은 HGD가 다른 DNN들과 동시에 실행되는 상황으로 보다 현실적인 실행 환경을 나타낸다. De_i와 I_i는 각각 i번째 이미지를 복원하는 과정과 타겟 DNN에 추론하는 단계이고 $\triangle_i^D$는 i번째 적대적 예제 복원 단계에 대한 지연 정도를 의미하며, $\triangle_i^I$는 타겟 DNN이 i번째 복구된 이미지를 추론할 때 지연되는 시간이다. 적대적 예제 복원의 지연의 원인은 GPU의 EE 큐에 DUNET과 타겟 DNN이 우선적으로 전달되지 않기 때문이다. DUNET의 추론 시간과 타겟 DNN의 추론 시간을 합하여 i번째 이미지에 대한 적대적 예제 복원 시간을 구할 수 있고 본 논문에서 적대적 예제 복원 시간을 타겟 추론 시간(target inference time)이라 하며 아래의 식과 같이 표현할 수 있다.

$$i^{th}\,target\ inference\ time = De_i + \triangle_i^D + I_i + \triangle_i^I \qquad (1)$$

또한 F개의 이미지 프레임에 대한 DNN의 타겟 추론 시간은 (2)의 식으로 정의한다.

$$Inf_{avg}(F) = \{\ \frac{\sum_{i=1}^{F}(De_i + \triangle_i^D + I_i + \triangle_i^I)}{F}\} \qquad (2)$$

De_i와 I_{i-1}에서 각각 추론되는 데이터들은 독립적이기 때문에 이 두 단계가 동시에 실행될 수 있으며 본 논문에서 $Inf_{avg}(F)$를 최소화 하는 것이 목표이다.

제 3 장 제안 기법

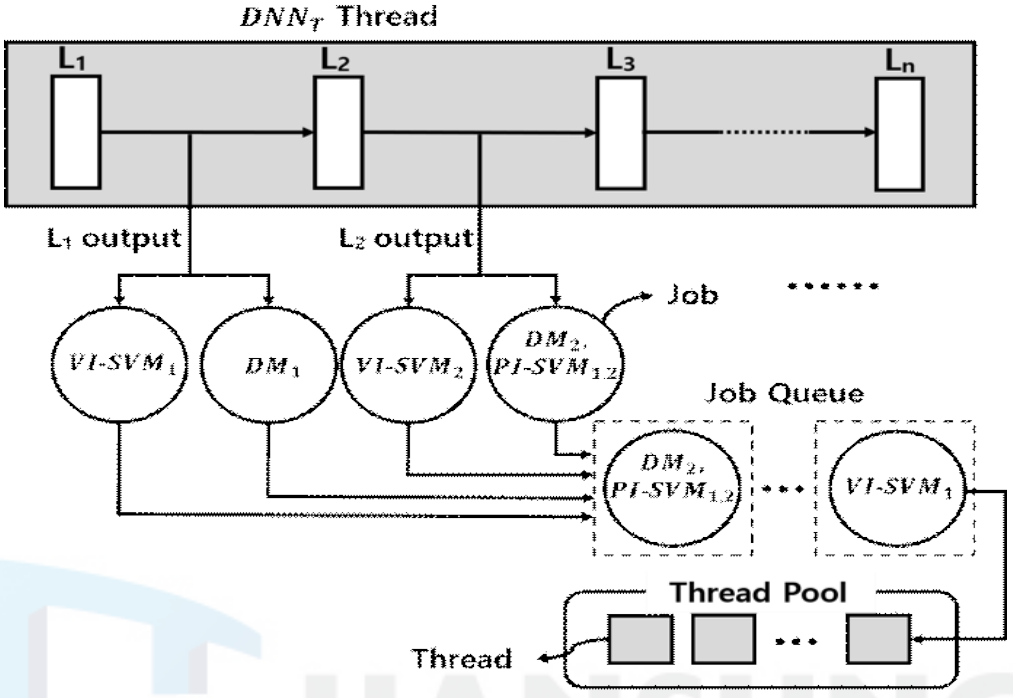
제 1 절 임베디드 NIC 시스템

본 절에서는 적대적 공격 대응 방법 중 적대적 공격 탐지 기법, 즉 NIC를 임베디드 시스템에 적용하기 위한 기법을 설명한다. 임베디드 시스템에서 t_{diff}를 줄이는 방법 중 하나로 기존의 하드웨어 대신 전용 하드웨어로 교체하여 적대적 공격 탐지의 성능을 극대화하는 방법이 있으나 전용 하드웨어가 없을 수도 있고 비용 측면의 부담과 같은 이유로 실질적인 대체 방법이 아니다. 이에 본 논문에서는 t_{diff}를 최소로 하기 위한 소프트웨어 기반 임베디드 NIC 시스템을 제안한다.

1) 임베디드 NIC 시스템 설계

[그림 3-1] 본 논문에서 제안하는 임베디드 NIC 시스템 구조이며 단일 프로세스를 사용하는 멀티 스레드이다. 은닉층 n개로 이루어진 타겟 DNN을 $DNN_T = \{L_1, L_2, L_3 \dots L_n\}$로 나타내고 모든 은닉층에서 이뤄지는 적대적 공격 탐지 과정 중 L_1, L_2, L_3에서의 적대적 공격 탐지 과정을 예로 설명한다. DNN_T에 NIC를 적용하여 적대적 공격 탐지를 하면 L_1의 출력 벡터를 입력으로 받는 VI-SVM $VI-SVM_1$과 L_2의 출력 벡터를 입력으로 받는 VI-SVM $VI-SVM_2$이 존재한다. DM은 L_1과 소프트맥스 계층으로 구성된 DM_1과 L_2과 소프트맥스 계층으로 구성된 DM_2가 있고 DM_1과 DM_2의 출력을 컨캣한 벡터를 입력으로 받는 PI-SVM $PI-SVM_{1,2}$가 있다.

DNN_T는 한 개의 스레드에서 추론되며 적대적 공격 탐지의 실행은 스레드 풀의 스레드에서 실행된다. 잡큐(Job Queue)는 스레드 풀의 스레드에



[그림 3-1] 임베디드 NIC 시스템 실행 구조

잡을 전송하는 FIFO(First-In-First-Out) 방식의 대기 큐이다. 잡큐에서 스레드 풀의 스레드로 전송되는 잡(Job)은 NIC에 필요한 각각의 VI-SVM 추론 과정, DM 추론 과정, 그리고 PI-SVM 추론 과정이다. DM 추론 과정과 PI-SVM 추론 과정은 의존성이 있으므로 하나의 잡으로 설계하여 한 개의 스레드에서 실행되도록 하였다. 잡큐에 잡이 잇고 스레드 풀에 있는 스레드들이 모두 잡을 할당 받은 상태라면 잡이 끝나는 스레드가 발생하기 전까지 잡큐에서 잡을 내보내지 않는다.

본 논문에서는 GPU의 병렬성을 극대화하기 위해 타겟 DNN 추론, VI-SVM 추론, 그리고 DM 추론 시 CUDA에서 제공하는 CUDA 스트림 API를 사용하였다. 스트림 API를 사용하지 않는 경우 디폴트 스트림만 사용하게 되며 GPU의 사용 가능한 남는 SM을 활용하지 못한다. 반면 멀티 스트림을 사용할 경우 남는 SM들이 각 스트림에 있는 GPU 작업을 동시에 실행

할 수 있게 되어 GPU 병렬성을 향상할 수 있다.

[그림 3-1]의 동작 방식은 다음과 같다. DNN_T의 추론이 시작되고 L_1의 출력 벡터가 L_2로 입력되기 전에 $VI-SVM_1$ 잡과 DM_1 잡을 생성한다. 생성된 각 잡들은 잡큐를 통한 후 스레드 풀의 스레드에서 DNN_T 추론과 병렬적으로 실행되고 L_1의 출력 벡터가 L_2로 입력되어 L_2의 출력 벡터를 얻게 되면 $VI-SVM_2$ 잡과 $DM_2, PI-SVM_{1,2}$ 잡을 생성한 후에 실행한다. $DM_2, PI-SVM_{1,2}$ 잡에서는 DM_2의 출력 벡터를 얻은 후 DM_1의 출력 벡터와 컨캣하여 $PI-SVM_{1,2}$의 추론을 진행한다. 이와 같은 동작을 DNN_T의 은닉층 마다 반복한다. DNN_T에 입력된 데이터가 각 잡에서 하나라도 적대적 예제라고 판별이 되면 입력 데이터는 적대적 예제라고 판단한다.

2) VI, PI 위반 탐지 최소화

NIC는 타겟 DNN의 모든 은닉층에서 VI와 PI를 위반하는지 안하는지 확인하는 것으로 타겟 DNN의 입력 데이터가 정상 데이터인지 또는 적대적 예제인지를 판별한다. 정상 데이터는 타겟 DNN의 모든 은닉층에서 VI와 PI를 위반하지 않아야 하며 이를 탐지하는 정확도를 각각 VI 탐지율, PI 탐지율이라고 칭하고 두 탐지율은 정상 데이터에 대한 탐지율이다. 반면 적대적 예제는 VI 또는 PI를 위반하며 각각의 위반 탐지 정확도를 VI 위반 탐지율과 PI 위반 탐지율이라 한다. 두 위반 탐지율은 적대적 공격 탐지율이다. 적대적 예제들을 NIC를 적용한 타겟 DNN에서 추론하면 특정 은닉층에서는 많은 적대적 예제들을 탐지하고 특정 은닉층에서는 적대적 예제들을 거의 탐지하지 못하거나 모든 적대적 예제들을 탐지하지 못한다. 타겟 DNN의 은닉층 중 VI 위반 탐지율이나 PI 위반 탐지율이 낮은 은닉층들은 적대적 공격 탐지를 제대로 수행하지 못하므로 2장에서 설명한 메모리 이슈로 인하여 해당 은닉층들에서의 OCSVM 및 DM을 이용한 적대적 예제 탐지를 생략한다. 결과적으로 적대적 공격 탐지율이 높은 일부 은닉층을 선별하여 해당 은닉층에서

만 VI 또는 PI의 위반 여부를 실행했다. 본 논문에서는 실험을 통해 이를 확인했고 4장에서 이 실험에 대해 자세히 설명한다.

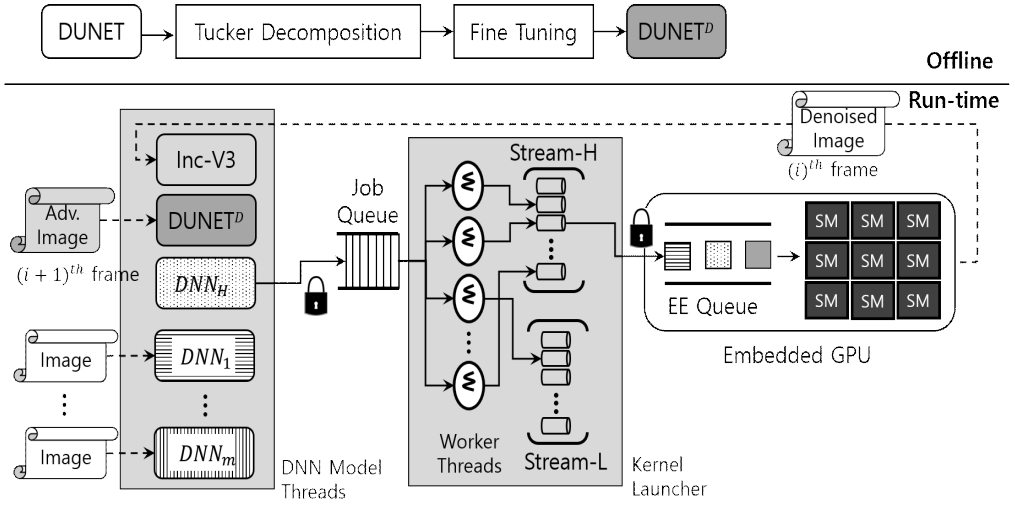
제안한 기법으로 기존 NIC의 적대적 공격 탐지율을 유지한 채 NIC에서는 모든 은닉층에서 필요했던 VI-SVM,DM 그리고 PI-SVM의 개수를 크게 줄여 메모리가 제한적인 임베디드 시스템에서도 문제없이 실행할 수 있도록 하였다. 적대적 공격 탐지를 수행하는 일부 은닉층 선별에 관한 내용은 다음 장에서 실험을 통해 자세히 설명하도록 한다.

제 2 절 eDenoizer

본 절에서는 적대적 공격 대응 방법 중 적대적 예제 복원 기법, 즉 HGD를 임베디드 시스템에 적용하기 위한 기법인 eDenoizer를 전반적인 작동 순서를 설명한 다음 세부적인 동작 원리를 서술한다.

1) eDenoizer 설계

[그림 3-2]는 eDenoizer의 작동 방식을 나타내며 오프라인(Offline) 단계와 런 타임(Run-time) 단계로 구성되어 있다. 오프라인에서 DUNET에 터커 분해(Tucker Decomposition)(Tucker, L. R., 1966)를 적용한 후 미세 조정(Fine Tuning)을 통해 정확도 손실이 최소화 되도록 학습하여 $DUNET^D$를 얻는다. 런 타임 단계에서는 크게 DNN 모델 스레드(DNN Model Threads), 잡큐(Job Queue), 커널 런처(Kernel Launcher)로 구성된 스케줄링 프레임워크를 통해 여러 DNN 모델과 DUNET이 실행된다. DNN 모델 스레드는 운영 체제(Operating System)를 통해 CPU에 스케줄링된 딥 러닝 모델로 구성되어 있으며 각각 스레드로 구현되어 있다. 각 모델 스레드는 각각의 DNN 동작을 잡큐에 비동기적으로 전달할 수 있다.

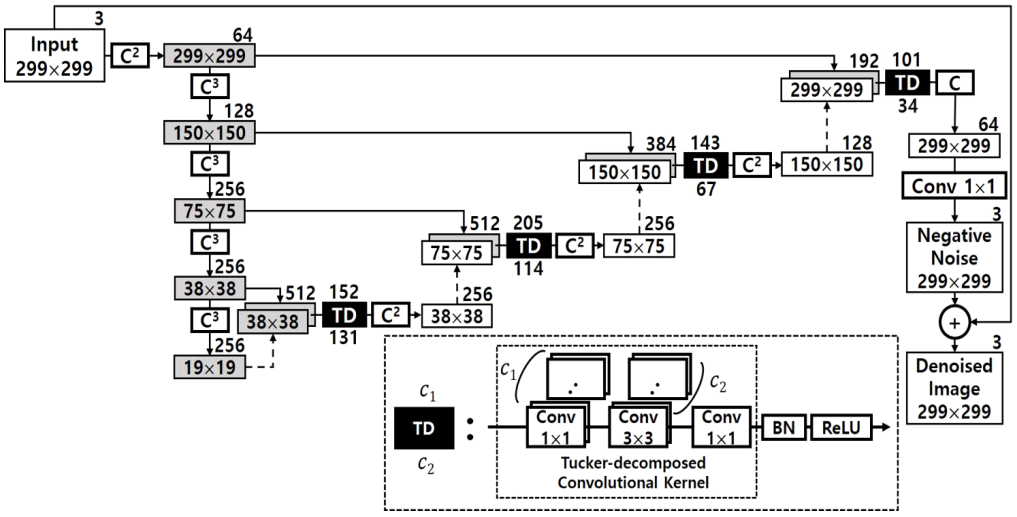


[그림 3-2] eDenoizer의 실행 구조

커널 런처는 워커 스레드(Worker Threads)에 잡큐 내부에 있는 각 DNN들의 연산들을 추출한 후 GPU의 EE 큐에 전달한다.

GPU의 출력 결과는 대부분 DUNET을 포함한 DNN들이 요청한 작업의 결과이며 이는 대부분 피쳐 맵이다. 이 값은 시스템 메모리에 저장되고 DNN 모델 스레드의 해당 DNN에 다시 입력된다. 이때 GPU의 출력 결과가 DUNET의 최종 출력인 복원된 이미지(Denoised Image)인 경우 다시 DUNET에 입력되지 않고 타겟 DNN([그림 3-2]의 Inception-V3)에 입력된다.

2장에서 언급한 바와 같이, DUNET이 (i)번째 이미지에 대한 복원 단계를 완료하지 않으면 (i)번째 이미지가 타겟 DNN에서 추론이 진행될 수 없다. 또한 $(i+1)$번째 이미지에 대한 DUNET 추론 단계와 (i)번째 이미지에 대한 타겟 DNN에서의 추론 단계는 데이터 의존성이 존재하지 않기 때문에 [그림 3-2]와 같이 스케줄링 프레임워크에서 둘 다 비동기식으로 동시에 처리될 수 있다. 주목할 점은 터커 분해로 DUNET의 추론 시간이 감소하여 $(i+1)$번째 DUNET의 추론 시간이 (i)번째 타겟 DNN 추론 시간에 가려지므로 전체 타겟 추론 시간은 타겟 DNN 추론 시간으로 제한할 수 있다는 점



[그림 3-3] 터커 분해를 적용한 DUNET 구조

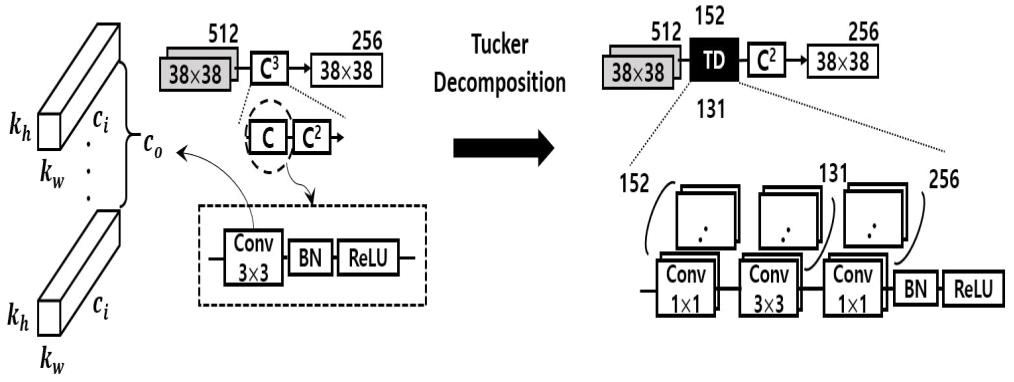
이다.

2) DUNET의 연산량 감소

[그림 3-3]은 DUNET의 추론 시간을 감소시키기 위해 터커 분해를 적용하여 연산 복잡도를 줄인 DUNET의 구조를 나타낸다. [그림 2-3]의 네 개의 원은 DUNET 전체 연산량의 40% 이상을 차지하므로, 3×3 컨볼루션 계층은 구조적으로 작은 연산량을 가지도록 변경할 필요가 있다. [그림 2-3]의 4개의 컨볼루션 계층은 각각 2개의 1×1 컨볼루션 계층과 1개의 작은 3×3 컨볼루션 계층으로 변경되며, [그림 3-3]의 흰색 글자로 TD 라고 표기된 4개의 검정 사각형은 터커 분해된 컨볼루션 커널 텐서를 의미한다.

터커 분해의 차원 축소를 위해, ResNet-152의 보틀넥(bottleneck) 블록과 동일하게 2개의 1×1 컨볼루션 계층이 3×3 컨볼루션 계층의 양쪽에 추가된다. 각 검정 사각형 위의 숫자는 첫 번째 1×1 터커 분해 컨볼루션 커널 텐서의 출력 채널수를 나타내고 아래 숫자는 첫 번째 1×1 커널 텐서 바로 다음에 위치한 3×3 컨볼루션 커널 텐서의 출력 채널수이다. [그림 2-3]에서 지적한 4 곳의 출력 채널수를 각각 c_1과 c_2로 변경하여 총 연산량을 감

Computational Cost in a Convolutional Layer = $k_h \times k_w \times c_i \times c_o \times w_o \times h_o$



[그림 3-4] 연산량 감소가 필요한 DUNET의 첫 번째 컨볼루션 계층의 터커 분해 결과

소시킨다.

[그림 3-4]는 [그림 2-3]의 왼쪽 첫 번째 원을 터커 분해한 결과를 예시로 나타낸 그림이다. [그림 2-3]의 왼쪽 첫 번째 원에는 출력 채널수가 256인 한 개의 3×3 커널 텐서가 있고 이 커널 텐서는 각각 152, 131 그리고 256의 출력 채널수를 가지는 세 개의 작은 컨볼루션 텐서로 분해된다. 하나의 컨볼루션 계층의 연산 비용은 [그림 3-4]에 표기된 공식에 의해 구해진다. 기존의 연산량인 $k_h \times k_w \times c_i \times c_o \times w_o \times h_o$ = 3×3×512×256×38×38 = 1,703,411,712가 1×1×512×152×38×38+3×3×152×131×38×38+1×1×131×256×38×38 = 419,580,192로 해당 부분의 컨볼루션 연산량이 약 75.4% 감소한다. 나머지 3 곳의 컨볼루션 계층에 이를 적용하면 DUNET의 전체 연산량은 총 25.41% 감소하며 연산량이 감소한 DUNET은 최종적으로 [그림 3-2]의 $DUNET^D$를 의미한다.

3) 멀티 DNN 모델을 위한 스케줄링 프레임워크

[그림 2-4]에서 언급한 바와 같이, DUNET이 다른 DNN들과 동시에

실행될 때 $\triangle_i^D$와 $\triangle_i^I$를 최소화 하는 것이다. 시스템이 시작되면 DNN 모델들이 시스템 메모리에서 GPU 메모리로 로드된다. 대부분의 임베디드 시스템에 내장된 GPU는 별도의 메모리가 존재하지 않아 CPU에서 참조되는 시스템 메모리를 공유한다. 따라서 DNN 모델들을 GPU 메모리에 로드하면 데이터 복사가 발생하지 않고 CPU에서 설정된 메모리 위치 정보만 전달된다.

가) 스케줄링 단위

DNN 모델 스레드의 스케줄링 단위가 DNN 모델 전체처럼 크다면 잡 큐에 모델 전체 크기의 잡이 입력되어 헤드오브라인 블락킹이 필연적으로 발생하게 되며 이로 인해 우선순위가 높은 DNN 모델의 스케줄링 지연이 일어난다. 반면, 너무 작은 단위가 스케줄링 되면 운영 체제의 트리거 빈도수가 증가되어 오버헤드가 생긴다. 예를 들어, 두 개의 컨볼루션을 스케줄링 단위로 사용할 때와 컨볼루션 한 개와 활성화 함수 한 개를 스케줄링 단위로 사용할 때 시스템 성능 측면에서 큰 차이가 있다. 따라서 잡큐에 전달될 단일 DNN 모델 스레드의 스케줄링 단위는 시스템 전반의 성능에 중요한 요소이다. 또한 각 DNN 모델의 구조와 각 DNN 동작에 필요한 연산 규모가 다르기 때문에 모든 DNN 모델에 균일한 규칙으로 스케줄링 단위를 적용하는 것은 바람직하지 않다. 이에 본 논문에서는 먼저 DUNET을 포함한 각 DNN 모델에 대한 최적의 스케줄링 단위를 오프라인에서 결정한다. [그림 3-2]에 있는 잡(Job)은 이 스케줄링 단위를 의미한다. 예를 들어, 일부 DNN에서는 하나의 컨볼루션이 잡이 될 수 있고, 다른 DNN에서는 컨볼루션, 배치 노말라이제이션(BN), 그리고 렐루(ReLU)와 같은 활성화 함수가 결합되어 잡이 될 수 있다.

나) 스케줄링 알고리즘

[그림 3-5]은 스케줄링 프레임워크의 자세한 동작 방법을 수도 코

Algorithm 1 Multi-DNN Scheduling Framework

```
1: function construct_layers(DNN_id)
2:     for l ← 0 to last_id do                    ▷ last_id: last layer index of DNN_id
3:         layers_id[l] ← L^l_id                  ▷ L^l_id: l^th layer of DNN_id defined offline
4:     end for
5:     return layers_id
6: end function
7: function execute_dnn(layers_id, DNN_id)
8:     prev_output ← input image
9:     for l ← 0 to last_id do
10:        job^l_id.layers ← layers_id[l]
11:        job^l_id.data ← prev_output
12:        enqueue(job^l_id)
13:        wait_signal(sig)
14:        prev_output ← output
15:    end for
16: end function
17: function execute_job( f)
18:    A:
19:    while (job queue is empty) do
20:        do nothing
21:    end while
22:    job_front ← dequeue( )
23:    output ← execute_kernel(job_front)
24:    send_signal(sig)
25:    goto A:
26: end function
```

[그림 3-5] eDenoizer의 동작 알고리즘을 나타낸 수도 코드

드로 표현한 것이다. [그림 3-2]의 각 DNN 모델 스레드는*construct_layers*()와 *execute_dnn*()의 두 개의 함수로 실행된다. 먼저, 각 DNN 모델의 서로 다른 특징을 고려하여 $layers_{id}[l]$는 오프라인에서 정의된 각 DNN의 구성 정보를 참조함으로써 L_{id}^{l}로 채워진다. 여기서 $layers_{id}[l]$는 DNN 은닉층의 인덱스를 의미하고 id는 DNN 식별 번호([그림 3-5]의 2~4행)를 나타낸다. L_{id}^{l}는 3×3 컨볼루션, BN 계층, 그리고 렐루 활성화 함수와 같은 여러 DNN 연산으로 구성되어 있다.

DNN이 스레드의 형태로 연산될 수 있도록 계층 구조화 절차가 완료되면 DNN은 실행된다. DNN에 데이터가 입력되면 $layers_{id}[l]$와 결합되어 잡

이 형성되며 [그림 3-5]에서는 $l=0$이므로 잡은 첫 번째 계층을 의미한다. 그런 다음 잡이 잡큐에 전달되고 커널 런처의 시그널을 기다린다([그림 3-5]의 10~13행). GPU에 할당됐던 잡이 완료되었다는 시그널이 수신되면 GPU의 출력 데이터는 다음에 수행될 잡의 입력으로 다시 전송된다([그림 3-5]의 11행, 여기서 $l=1$). 이런 과정은 L_{id}^{l}를 사용한 마지막 잡이 완료될 때 까지 반복된다.

커널 런처의 주요 동작은 [그림 3-5]의 $execute_job()$에 나타나 있다. [그림 3-2]에서 알 수 있듯이 커널 런처는 여러 개의 워커 스레드에 의해 잡큐에 비동기 방식으로 접근하여 요청된 잡을 꺼낸다([그림 3-5]의 19~22행). 추출된 잡은 $execute_kernel()$을 통해 실행되며([그림 3-5]의 23행) 이때 워커 스레드는 요청된 잡을 GPU 커널로 변환하고, CUDA 스트림 중 하나를 통해 EE 큐로 전송한다.

다) 스케줄링 프레임워크 분석

[그림 3-5]의 $construct_layers()$는 eDnoiser 실행 중에는 사용되지 않으며 오프라인엣 진행되는 과정이므로 분석 대상에서 제외된다. 함수 $execute_job()$은 잡큐의 가장 앞의 요소의 잡을 추출하기 때문에 $O(1)$의 시간 복잡도이며 수행할 DNN의 은닉층 수가 $n=last_{id}$일 때, n개의 은닉층 중 어느 것도 DNN 실행 중에 생략될 수 없으므로 $execute_dnn()$의 시간 복잡도는 $O(n)$이다.

잡큐 및 EE 큐에 대한 락(lock)을 획득할 때 발생하는 오버헤드로 인해 하나의 DNN만 실행하는 시스템에서는 본 논문에서 제안하는 프레임워크는 효율적이지 않고 [그림 3-2]와 같이 DUNET을 포함한 여러 DNN이 동시에 실행되는 환경에서 효율적이다. [그림 3-2]에서 확인할 수 있듯이 다수의 DNN들이 잡큐에 접근을 요청할 때와 워커 스레드가 CUDA 스트림을 통해 EE 큐에 접근을 요청할 때, 두 곳에서 동기화 문제가 발생한다. 이러한 동기

화 문제는 DNN 연산을 직렬적으로 실행하게 하여 시스템 전반의 성능을 저하시킬 수 있다. 그러나 멀티 DNN은 단일 임베디드 GPU에서 요청된 DNN 연산을 처리하는 것보다 연속된 입력 데이터에 대한 연산을 요청하는 것이 더 빠르기 때문에 큐의 언더플로우가 발생하지 않고 항상 두 큐에 여러 잡이 있으므로 두 큐에 접근하는 동기화 선점은 시스템의 전체 성능에 영향을 미치지 않는다.

라) 우선순위 기반 DNN 연산과 병렬 연산의 최대화

본 논문에서 제안하는 프레임워크는 CPU의 운영 체제를 통해 사용자가 설정한 우선순위에 따라 GPU에서 DNN을 실행할 수 있게 한다. 이를 위해 잡큐는 DNN 모델 스레드가 전달하는 작업을 각 DNN의 우선순위에 따라 정렬할 수 있는 우선순위 큐이다. 따라서 워커 스레드 중 하나에 의해 추출된 잡은 요청된 잡 중 항상 가장 높은 우선순위를 가진다. GPU는 비선제적 특징을 가지지만 NVIDIA는 스트림을 하이(High)와 로우(Low) 레벨로 나눠 선제를 가질 수 있는 기술을 제공한다(NVIDIA, CUDA Streams: Best practice and common pitfalls, online). 따라서 하이 레벨 스트림의 DNN 커널은 로우 레벨 스트림의 DNN 커널보다 연산이 먼저 이뤄질 수 있다. 또한 각 워커 스레드는 SM 가용성에 따라 CUDA 스트림 중 하나를 별도로 동시에 사용할 수 있으며, 여러 잡을 동시에 실행할 수 있어 시스템 처리량을 향상시킨다. 그 결과로 [그림 2-4]의 $\triangle_i^D$ 와 $\triangle_i^I$를 효과적으로 단축할 수 있다.

제 4 장 실험 및 분석

제 1 절 임베디드 NIC 평가

본 절에서는 본 논문에서 제안하는 임베디드 NIC의 효용성을 검증한다. 우선 실험환경을 설명하고 실험에 사용된 각 DNN의 VI와 PI의 위반 검출을 실행하는 특정 은닉층들을 선별한 근거를 실험 결과를 통해 설명한다. 그 후 구체적인 성능 평가 결과를 제시한다.

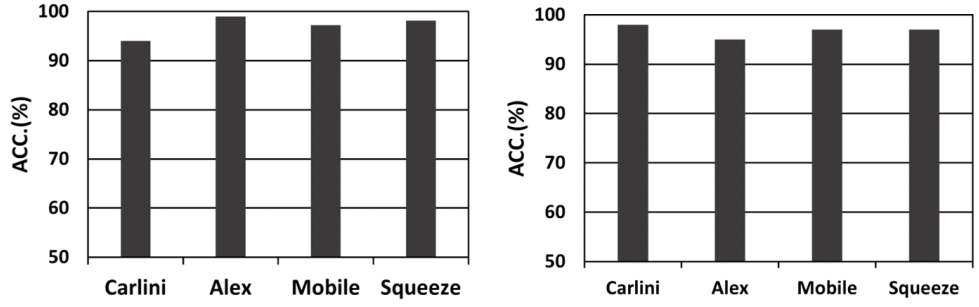
1) 실험 환경

본 실험은 NIVIDA의 Jetson AGX Xavier(이하 AGX-Xavier)를 타겟 임베디드 시스템으로 사용했다. [표 4-1]은 AGX-Xavier의 하드웨어와 소프트웨어의 사양을 각각 나타낸다.

GPU	512-core Volta GPU with Tensor Cores
CPU	8-core ARM v8.2 64-bit CPU 8MB L2 + 4MB L3
Memory	32GB 256-Bit LPDDR4x
Jetpack	ver 4.2
OS	Linux kernel version 4.9.140
CUDA	CUDA 10.0.166

[표 4-1] Jetson AGX Xavier 상세정보

실험에는 Carlini model(Carlini, N. & Wagner, D., 2017), AlexNet(Krizhevsky, A., et al, 2017), MobileNetV1(Howard, A. G., et al, 2017), SqueezeNet(Iandola, F. N., et al, 2016) 총 4종류의 DNN을 타겟 DNN으로 사용하였으며 표기를 간소화하기 위해 4개의 DNN을 각각 Carlini, Alex, Mobile, Squeeze로 표한다. 4개의 타겟 DNN 학습에 사용된 데이터 셋은 CIFAR-10 데이터 셋(Krizhevsky, A. & Hinton, G.,



(a) (b)

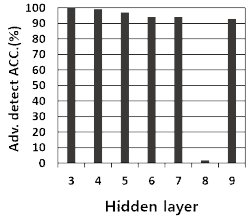
[그림 4-1] 각 타겟 DNN의 모든 은닉층의 VI, PI 탐지율 중 최소값

2009)을 사용하였고, 적대적 공격 예제는 CIFAR-10 데이터 셋에 FGSM 공격(Goodfellow, I. J., et al, 2014)을 NIC와 동일한 방식으로 적용하여 100개를 생성하였다.

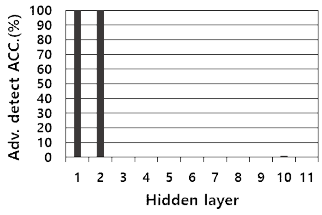
2) 타겟 DNN별 각 은닉층에서의 VI와 PI 탐지율 및 위반 탐지율

본 실험에서는 4종류의 타겟 DNN에서 적대적 공격 탐지 은닉층을 선별한 기준을 실험 결과를 토대로 설명한다. 실험에서 사용한 정상 데이터는 총 50,000장이며 [그림 4-1]은 정상 데이터에 대해 각 타겟 DNN의 모든 은닉층 중 최소 VI 탐지율(a)과 최소 PI 탐지율(b)을 각각 나타낸다. 타겟 DNN마다 VI, PI 탐지율이 모든 은닉층에서 최소 95% 이상인 것을 확인할 수 있다. 타겟 DNN의 종류, 은닉층과 관계없이 정상 데이터 탐지율은 항상 높으므로 적대적 공격 탐지를 진행하는 은닉층을 선별하기 위하여 각 은닉층의 VI, PI 위반 탐지율만 고려하였다.

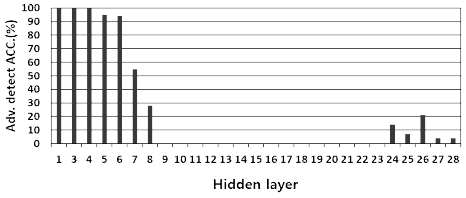
[그림 4-2]는 4종류의 타겟 DNN의 모든 은닉층에서 적대적 예제로 측정한 VI를 위반한 탐지 정확도를 나타낸다. 각 타겟 DNN별 은닉층의 수는 Carlini 9개, Alex 11개, Mobile 28개, Squeeze 21개이며 x축은 각 타겟 DNN의 입력 계층부터의 은닉층 순서이다. 그래프에 표기가 되지 않은 은닉층은 본 실험 환경에서 OCSVM을 학습하기에 너무 큰 차원을 가지고 있어



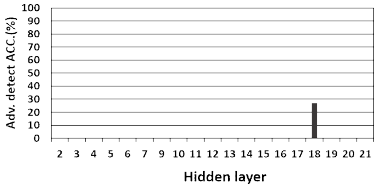
(a) Carlini



(b) Alex

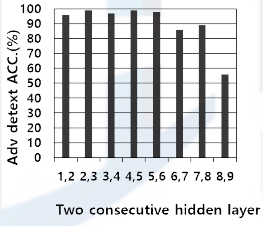


(c) Mobile

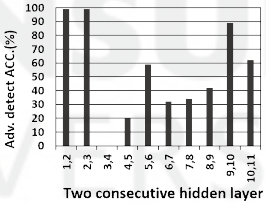


(d) Squeeze

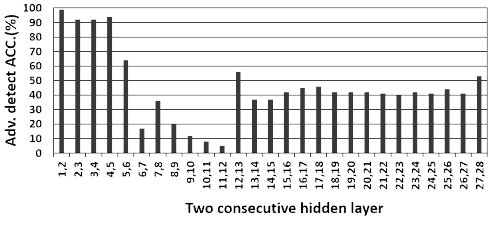
[그림 4-2] 타겟 DNN의 각 은닉층에서 적대적 예제의 VI 위반 탐지율



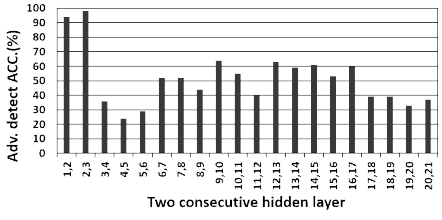
(a) Carlini



(b) Alex



(c) Mobile



(d) Squeeze

[그림 4-3] 타겟 DNN의 각 은닉층에서 적대적 예제의 PI 위반 탐지율

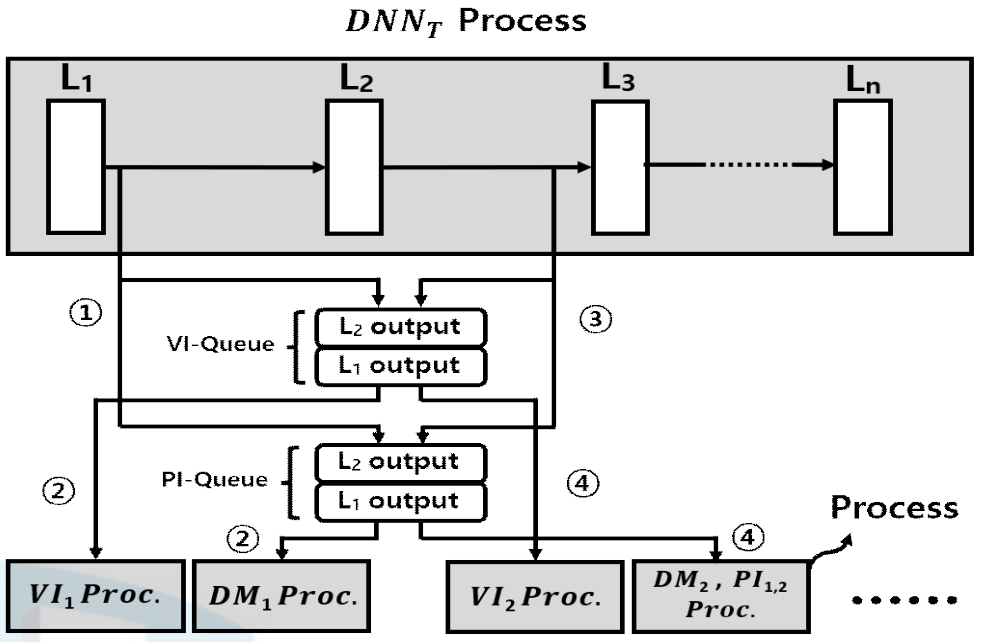
OCSVM 학습을 진행하지 못한 은닉층이다. [그림 4-3]은 4종류의 타겟 DNN별 2개의 모든 연속된 은닉층에서 PI를 위반한 적대적 예제의 탐지 정확도를 나타낸다.

[그림 4-3]에서 알 수 있듯이 Carlini, Alex, Mobile은 각각 (3번, (1번, 2번), (1번, 3번, 4번) 은닉층에서 100%의 정확도로 VI 위반을 탐지할 수 있다. 해당 은닉층들 중에서 VI 탐지율이 가장 높은 은닉층만을 택하여 Carlini, Alex, Mobile은 각각 3번, 2번 그리고 3번 은닉층으로 선정하였다.

Squeeze의 경우 VI 위반 탐지율을 모든 은닉층에서 확인한 결과 VI 위반 탐지율이 가장 높은 은닉층이 27%로 매우 낮은 탐지 정확도를 나타냈다. 그러나 PI 위반 탐지율 그래프에서는 PI 위반 탐지율이 98%인 2, 3번 은닉층이 존재하였음에 따라 Squeeze는 VI-SVM을 수행하지 않고 PI-SVM만 수행하기로 하였다. 이때 2% 부족한 PI 위반 탐지율을 만회하기 위해 PI 위반 탐지율이 다음으로 높은 1, 2번 은닉층을 PI-SVM 수행 대상으로 추가하여 1~3번의 연속된 3개의 은닉층에서 PI 위반을 탐지하도록 하였다.

3) 임베디드 NIC 시스템 성능 평가

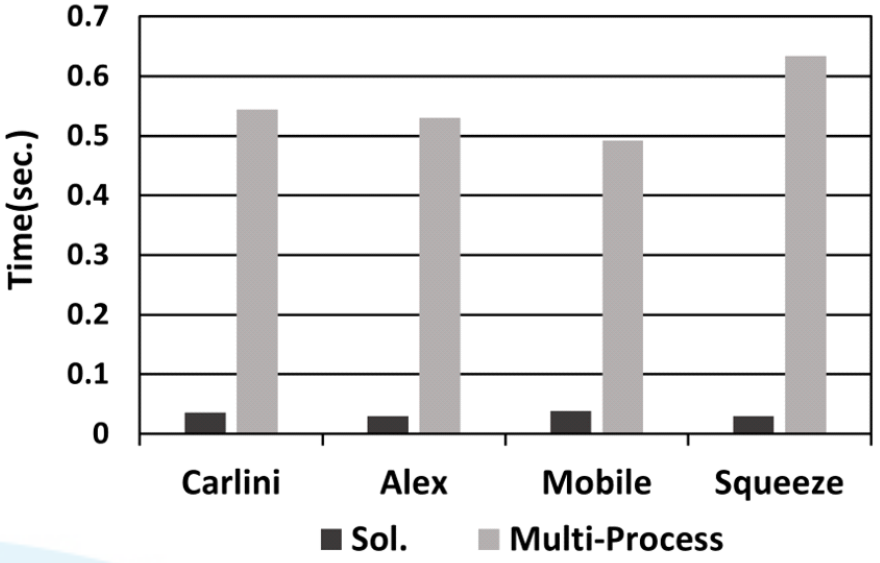
본 실험에서는 임베디드 NIC 시스템을 멀티 프로세스 환경에서 타겟 DNN과 NIC가 병렬적으로 실행되게 설계된 시스템(이하 멀티 프로세스 NIC 시스템)과 비교하여 성능 평가를 진행한다. [그림 4-4]는 멀티 프로세스 NIC 시스템 설계 구조이다. [그림 3-1]과 동일하게 n개의 은닉층으로 이뤄진 타겟 DNN은 DNN_T로 나타냈고 모든 은닉층에서 이뤄지는 적대적 공격 탐지 중 L_1, L_2, L_3에서의 과정만 상세히 표현했다. DNN_T는 한 개의 프로세스에서 추론이 진행되고 DNN_T의 각 은닉층의 VI-SVM 또한 서로 다른 각각의 프로세스에서 추론된다. [그림 4-4]에서 $VI_1Proc.$, $VI_2Proc.$는 각각 L_1과 L_2의 VI-SVM을 추론하는 프로세스다. DM 추론과 PI-SVM은 [그림 3-1]에서는 같은 스레드에서 실행된 것과 비슷하게 하나의 프로세스에서 실행되며 $DM_1Proc.$는 L_1에 해당하는 DM을 추론하는 프로세스, $DM_2, PI_{1,2}Proc.$는 L_2에 해당하는 DM과 PI-SVM 추론이 진행되는 프로세스다. 서로 다른 프로세스 간 데이터를 통신하기 위해 파이썬에서 제공하는 Queue API(Python.



[그림 4-4] 멀티 프로세스 NIC 시스템의 실행 구조

Multiprocessing: Process-based parallelism, online)를 사용하였고 $VI-Queue$는 DNN_T의 각 은닉층에서 도출되는 출력 벡터를 VI-SVM 추론을 수행하는 프로세스들에게 전송하기 위한 큐이며 $PI-Queue$는 각 은닉층의 출력 벡터를 DM과 PI-SVM 추론을 수행하는 프로세스들에게 전달하기 위한 큐이다. 또한 DNN_T 추론 후 프로세스 생성 오버헤드가 발생하지 않도록 DNN_T 추론 전에 적대적 공격 탐지에 필요한 프로세스들을 미리 생성한다.

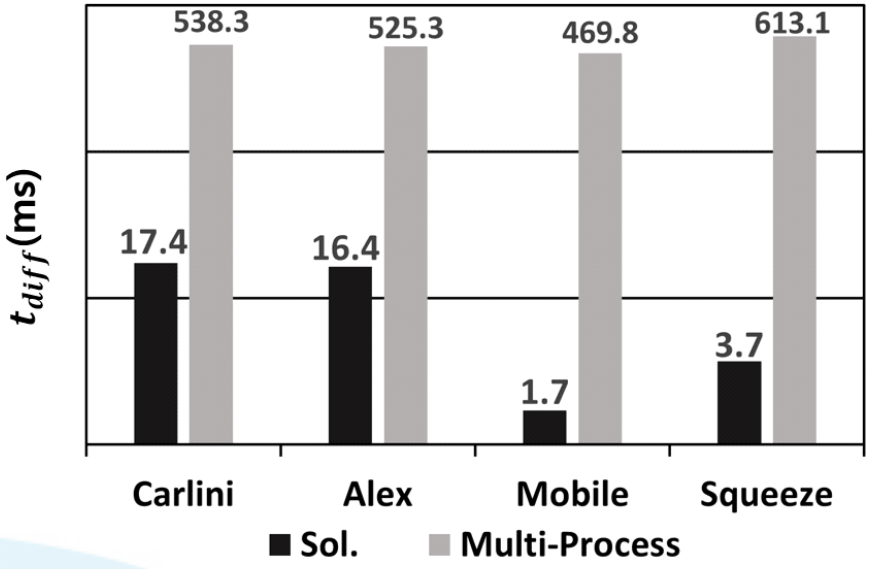
L_1, L_2, L_3 에서의 적대적 공격 탐지 동작을 멀티 프로세스 NIC 시스템 동작을 [그림 4-4]에 표기된 ①~④으로 설명하면 다음과 같다. ① DNN_T의 추론이 시작된 후, L_1의 출력 벡터가 L_2로 입력되기 전에 $VI-Queue$와 $PI-Queue$로 전달되고 DNN_T를 추론하는 프로세스에서 미리 생성되어 있는 $VI_1Proc.$과 $DM_1Proc.$으로 시그널을 보낸다. ② 다음으로 L_1의 출력 벡터는



[그림 4-5] NIC 적용 시 각 타겟 DNN의 전체 수행 시간 비교

L_2에 입력되고 시그널을 받은 $VI_1Proc.$과 $DM_1Proc.$은 각각 $VI-Queue$와 $PI-Queue$에 있는 L_1의 출력 벡터를 전달받게 되어 VI-SVM 추론과 DM 추론을 진행한다. 이때 DNN_T의 추론은 해당 프로세스에서 계속 진행되고 있는 상태이다. ③ L_2의 연산이 완료되어 도출된 출력 벡터가 L_3로 입력되기 전에 L_2의 출력 벡터를 두 개의 큐에 전달한다. ④ DNN_T를 추론하는 프로세스로부터 시그널을 받은 $VI_2Proc.$과 $DM_2, PI_{1,2}Proc.$은 각각 L_2의 VI-SVM 추론과 L_2의 DM 추론 및 PI-SVM 추론을 수행한다.

[그림 4-5]와 [그림 4-6]에서 Sol.은 본 논문에서 제안하는 임베디드 NIC 시스템의 결과를 나타내고 Multi-Process는 멀티 프로세스 NIC 시스템의 결과를 의미한다. [그림 4-5]는 임베디드 NIC 시스템과 멀티 프로세스 NIC 시스템의 전체 소요 시간을 타겟 DNN별로 측정한 것이다. 모든 타겟 DNN에 대해 임베디드 NIC 시스템은 멀티 프로세스 NIC 시스템보다 전체 소요 시간이 매우 감소함을 알 수 있다. Carlini는 93.3%, Alex는 94.3%,



[그림 4-6] NIC 적용 시 각 타겟 DNN의 t_{diff} 비교

Mobile은 91.7% 그리고 Squeeze는 95.2%만큼 소요 시간이 감소했다.

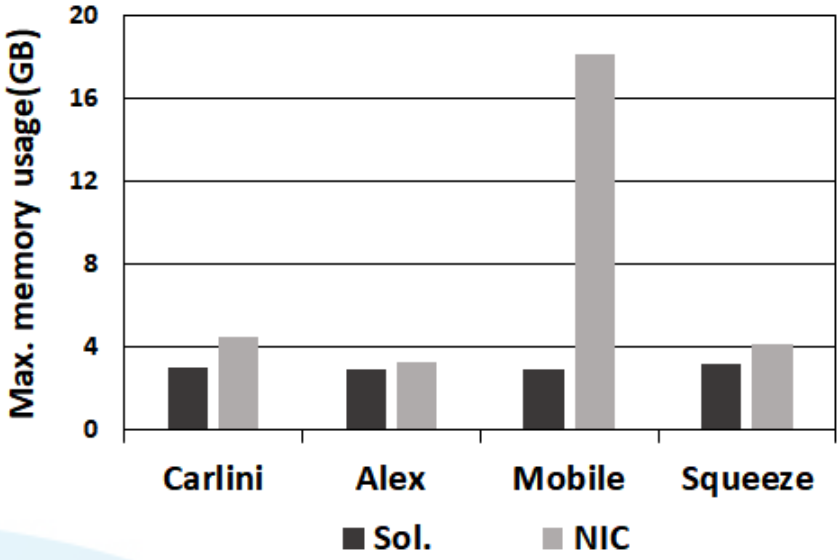
[그림 4-6]은 임베디드 NIC 시스템과 멀티 프로세스 NIC 시스템에서의 t_{diff}를 측정한 것이다. 멀티 프로세스 NIC 시스템 대비 임베디드 NIC 시스템에서 Carlini는 96.7%, Alex는 96.8%, Mobile은 99.6% 그리고 Squueze는 99.3%만큼 t_{diff}가 감소했다. 실험에 사용된 4 종류의 타겟 DNN 모두 t_{diff}가 감소했지만 Mobile과 Squeeze에서 t_{diff}의 감소량이 많은 것을 확인할 수 있다. Carlini와 Alex의 총 은닉층 개수는 각각 9개와 11개이고 Mobile과 Squeeze의 총 은닉층 개수는 각각 28개와 21개이다. 따라서 타겟 DNN의 추론 시간이 Mobile과 Squeeze가 Carlini와 Alex에 비해 느리게 종료되는 반면 적대적 공격 탐지 종료 시간은 4 종류의 타겟 DNN 간 차이가 크지 않아 Mobile과 Squeeze처럼 총 은닉층의 개수가 많은 DNN에서 t_{diff}의 감소율이 높은 것을 알 수 있다.

4) 최대 메모리 사용량

[그림 4-7]에서 Sol.은 본 논문에서 제안한 소프트웨어 환경에서 4 종류의 타겟 DNN 각각에 적대적 공격 탐지율이 높은 특정 은닉층들을 선출하여 VI 또는 PI 위반 탐지를 했을 때이며 NIC는 Sol.과 동일한 소프트웨어 환경에서 각 타겟 DNN의 모든 은닉층에서 VI 또는 PI 우비나 탐지를 했을 때의 최대 메모리 사용량을 나타낸다. 최대 메모리 사용량 측정은 NVIDIA에서 제공하는 tegrastats 유틸리티(NVIDIA, DRIVE OS Linux, online)를 이용하여 진행했다. 각각의 메모리 사용량을 측정하기 전에 tegrastats 유틸리티를 미리 실행하여 임베디드 시스템에서 이미 사용되고 있는 메모리 사용량을 측정한 후 NIC가 적용된 시스템을 각각 실행하여 0.01초 주기로 메모리 사용량을 측정했다. 그 후 최대 메모리 사용량과 이미 사용되고 있었던 메모리 사용량의 차이를 구해 타겟 DNN별 최대 메모리 사용량을 구했다. 또한 본 논문에서는 텍스트 형식으로 저장되던 OCSVM 모델을 바이너리 형식으로 변경 후 저장하여 메모리 사용량을 최소화했다.

[그림 4-7]을 통해 타겟 DNN별로 특정 은닉층에서 적대적 공격 탐지를 했을 때의 최대 메모리 사용량이 모든 은닉층에서 적대적 공격 탐지를 했을 때의 최대 메모리 사용량 대비 Carlini는 32.7%, Alex는 11.1%, Mobile은 83.9% 그리고 Squeeze는 23.2% 감소했음을 알 수 있다. 최대 메모리 사용량에 가장 큰 영향을 미치는 것은 VI-SVM 모델의 크기이며 Mobile의 VI-SVM 중 가장 큰 크기를 가지는 모델이 Carlini, Alex 그리고 Squeeze 각각의 가장 큰 크기를 가지는 VI-SVM 보다 10.3배, 66.7배 그리고 14.4배 만큼 컸기 때문에 Mobile에서의 최대 메모리 사용량 차이가 다른 타겟 DNN에 비해 월등히 크다.

[그림 4-7]의 Mobile의 경우에서 알 수 있듯이 Carlini, Alex, Squeeze와 달리 NIC에서 적대적 예제 탐지를 하기 위해 사용하는 VI-SVM와 PI-SVM을 사용할 경우 임베디드 환경에 어울리지 않는 매우 큰 최대 메모리 사용량(Mobile의 경우 18GB 이상)을 가지는 DNN 모델들이 존재한다.



[그림 4-7] 적대적 공격 탐지를 실행하는 은닉층 수에 따른 각 타겟 DNN의 최대 메모리 사용량 비교

본 논문에서 제안한 특정 은닉층 선별 기법에 근거한 NIC 적용은 Mobile과 유사하거나 더 많은 은닉층을 가지는 DNN 모델들에도 적용할 수 있는 솔루션이다.

Jetson Xavier NX(NVIDIA. World's smallest AI supercomputer: Jetson Xavier NX, online)의 시스템 메모리 크기는 본 실험에서 사용한 AGX-Xavier보다 작은 8GB이다. 본 논문에서 제안한 특정 은닉층에서만 적대적 공격 탐지를 수행하는 방식을 적용할 경우 Jetson Xavier NX와 같은 작은 메모리를 가지는 임베디드 시스템에서도 타겟 DNN들에 NIC를 적용할 수 있다.

제 2 절 eDenoizer 평가

본 절에서는 본 논문에서 제안하는 eDenoizer를 평가하기 위해 수행한 실험을 설명한다. 먼저 eDenoizer를 구현한 방법 및 실험 환경을 설명하고 수

행한 실험들의 결과를 분석한다. DUNET은 LGD를 사용했다.

1) eDenoizer 구현

DNN 모델 스레드의 DNN 모델은 PyTorch 프레임워크로부터 생성된다. 하나의 스레드만 파이썬 인터프리터를 처리할 수 있는 GIL(Global Interpreter Lock)(Ajitsaria, A., What is the python global interpreter lock(GIL), online)로 인해 파이썬으로 DNN을 병렬로 실행하는 것이 어려우므로 C++를 활용하였고 이를 위해 Libtorch 라이브러리를 사용하여 프레임워크를 구현했다. 이를 통해 본 논문에서 제안한 스케줄링 프레임워크는 멀티 스레드 환경에서 [그림 3-2]의 DNN 모델 스레드와 워커 스레드를 실행할 수 있으며 오프라인에서 터커 분해를 수행한다. DNN 모델 스레드의 은닉층 구성에 대해서는 각 DNN 스레드마다 은닉층 구성을 재구성하여 각 DNN 실행에 최적화하였다. 또한, [그림 3-5]에 나타난 바와 같이, 워커 스레드와 DNN 모델 스레드 사이의 시그널 기반 통신 체계를 사용하여 DNN 잡의 실행을 보다 더 상호작용적으로 만든다.

2) 실험 환경

eDenoizer의 성능을 평가하기 위해 임베디드 NIC 시스템 평가를 위해 사용된 AGX-Xavier를 타겟 임베디드 시스템으로 설정했다. 본 실험의 데이터 셋은 ImageNet(ImageNet, online)이 사용되었으며 eDenoizer의 훈련 및 평가, 그리고 적대적 예제 생성에도 사용되며 이는 HGD에 사용된 데이터 셋과 동일하다. 또한 Inception-V3은 적대적 공격이 공격하려는 타겟 DNN임과 동시에 DUNET이 방어하고자 하는 타겟 DNN이다.

	Attack Method	Attacked Model
Training Set and Validation Set	FGSM	IncV3
	FGSM	IncResV2
	FGSM	Res
	FGSM	IncV3/IncResV2/Res
	IFGSM2	IncV3/IncResV2/Res
	IFGSM4	IncV3/IncResV2/Res
	IFGSM8	IncV3/IncResV2/Res

[표 4-2] 학습과 검증을 위한 적대적 예제

적대적 예제를 생성하기 위해 ImageNet 훈련 셋(Training set)에서 30,000장의 정상 데이터를 추출한 후 잡음을 추가하여 정상 데이터를 왜곡하고 이때 본 실험에서는 [표 4-2]에 표기된 공격 방법들을 사용한다. 공격 방법은 FGSM과 IFGSM이며 FGSM이 n번 반복되면 IFGSMn으로 표기한다. 공격 DNN(적대적 예제를 생성하기 위해 사용된 DNN)은 Inception-V3, InceptionResnet V2(Szegedy, C., et al, 2017) 그리고 ResNet50 V2(He, K., Res, S., et al, 2016) 총 3개 사용하였으며 3개의 모델을 합친 것을 앙상블 적대적 예제를 생성하는데 사용한다(Tramer, F., et al, 2017). 표기를 간단히 하기 위해 각 공격 DNN을 IncV3, IncResV2 그리고 Res로 표기한다. 모든 적대적 예제 학습 단계에서, ϵ(잡음 단계)는 1에서 16 사이에서 균일하게 적용되고 정상 데이터를 포함한 전체 학습 데이터의 개수는 240,000개이다. 검증 데이터 셋을 만들기 위해 훈련 데이터 셋을 만들 때 사용된 동일한 방법이 사용되며 이를 위해 ImageNet 훈련 셋에서 10,000개의 데이터를 추출하고 총 80,000개의 검증 데이터 셋을 생성한다.

테스트 데이터 셋을 구성하기 위해 [표 4-3]와 같이 white-box 공격과 black-box 공격을 적용한다.

	Attack Method	Attacked Model
White-box-test-set	FGSM	IncV3
	IFGSM4	IncV3/IncResV2/Res
Black-box-test-set	FGSM	Inception-V4
	IFGSM4	Inception-V4

[표 4-3] 테스트를 위한 적대적 예제

테스트 데이터 셋을 구성하기 위해 [표 4-3]와 같이 white-box 공격과 black-box 공격을 적용한다. ImageNet의 검증 셋(validation set)에서 10,000장의 데이터를 추출한 후, [표 4-3]와 같은 공격 방식을 가한다. White-box 공격에는 DUNET의 타겟 DNN인 Inception-V3가 공격 DNN으로 사용된 반면 black-box 공격에는 DUNET의 타겟 DNN과 다른 Inception-V4[40]가 사용된다.

3) 적대적 예제에 대한 분류 정확도

본 실험에서는 터커 분해 적용의 효과를 입증한다. 첫째로, 잡음 제거 성능의 변화를 확인하기 위해 터커 분해가 DUNET의 분류 정확도에 어떤 영향을 미치는지 분석한다. 둘째로, 기존 DUNET과 터커 분해를 적용한 DUNET의 전이성을 비교한다. 이때 전이성이란 DUNET을 학습할 때 사용하지 않은 DNN에서 적대적 예제 복원 후의 분류 정확도를 확인하는 것을 의미한다. 분류 정확도 실험과 전이성 실험에서 ϵ은 4로 설정하였다.

가) 적대적 예제에 대한 터커 분해를 적용한 DUNET의 분류 정확도

본 실험에서는 DUNET의 4 곳의 컨볼루션 커널 텐서들에 터커 분해를 통해 생성된 3개의 터커 분해 텐서들을 각각 적용했을 때의 효과를 확인하며 적대적 예제는 [표 4-3]에 있는 White-box-test-set과 Black-box-test-set을 통해 생성된다. White-box-test-set의 적대적 예제의 경우, Inception-V3을 사용하여 생성되며 타겟 DNN 또한 Inception-V3인 반면

Black-box-test-set의 적대적 예제는 Inception-V3 대신 Inception-V4를 사용하여 생성되며, 공격 시 추론을 수행하는 타겟 DNN은 Inception-V3이다.

	Result in	Org. DUNET	Approx. DUNET
Clean-image-test-set	76.2%	76.53%	76.38%
White-box-test-set	75.2%	72.37%	70.59%
Black-box-test-set	75.1%	74.86%	74.45%

[표 4-4] 분류 정확도 비교

[표 4-4]은 정상 데이터, White-box-test-set 그리고 Black-box-test-set에 대한 분류 정확도를 나타낸다. [표 4-4]에서 Org. DUNET은 수정하지 않은 기존은 DUNET을 의미하며 Approx. DUNET은 터커 분해를 적용한 DUNET을 나타낸다. 각 정확도는 [표 4-3]에 있는 FGSM과 IFGSM4 두 공격에 대한 분류 정확도의 평균이다. White-box-test-set의 경우 터커 분해가 분류 정확도에 1.78% 미만으로 영향을 미치며, Black-box-test-set의 경우 0.41%의 성능 저하가 관찰된다. 이러한 결과는 터커 분해로 DUNET의 연산량이 감소하더라도 잡음 제거 성능이 거의 감소하지 않는다는 것을 보여준다. 또한 정상 데이터를 사용한 실험 결과에서는 성능 저하가 0.15% 미만으로 거의 없다.

[표 4-4]의 Result in은 HGD 논문에 표기된 결과이며 이는 Org. DUNET과 같아야 하지만 실제로는 분류 정확도가 다르다. 이러한 차이가 발생하는 이유는 훈련 및 테스트를 위한 적대적 예제를 생성하는 데 사용되는 ImageNet 데이터 셋의 원본 이미지가 무작위로 선택되기 때문이다. 따라서 본 연구에서 사용된 훈련 및 테스트 데이터 셋과 HGD의 데이터 셋은 다를 수밖에 없으며 Org. DUNET의 학습 시퀀스가 HGD와 동일하더라도 HGD의 결과와 Org. DUNET의 결과는 다르다. 본 실험에서 Approx. DUNET의 결과와 본 실험 환경에서 측정된 Org. DUNET의 결과를 비교한다.

나) 터커 분해를 적용한 DUNET의 전이성

DUNET의 핵심은 적대적 예제에 있는 잡음을 없애는 negative noise를 생성하는 것이다. 이러한 측면에서 DUNET의 기능을 다른 모델로 이전하는 것이 가능하다. [표 4-4]은 적대적 예제를 복원할 타겟 DNN이 Inception-V3일 때의 결과이다. 전이성을 평가한다는 것은 DUNET 학습 시 Inception-V3를 타겟 DNN으로 사용한 반면 DUNET 분류 정확도 확인 시 타겟 DNN을 Inception-V3가 아닌 다른 종류의 DNN으로 설정한다는 것이다. [표 4-5]는 DUNET의 전이성을 확인한 표이며 타겟 DNN으로 ResNet-152을 사용한 결과이다.

	Result in	Org. DUNET	Approx. DUNET
Clean-image-test-set	77.4%	73.7%	73.5%
White-box-test-set	75.8%	71.35%	70.86%
Black-box-test-set	76.1%	72.07%	71.58%

[표 4-5] ResNet-152로 전이성 확인

[표 4-4]의 결과와 마찬가지로 전이성을 확인한 결과 white-box와 black-box 공격 모두에서 터커 분해가 있는 컨볼루션 커널 텐서와 기존의 DUNET 차이가 0.49% 미만이었으며 정상 데이터를 사용한 실험 결과에서는 0.2% 미만의 분류 정확도 차이로 성능 저하가 거의 없었다.

4) eDenoizer 성능 평가를 위한 시간 비교

본 실험에서는 eDenoizer의 실행 성능을 평가하기 위해 두 가지 실행 환경에서 실험이 진행되었다. 첫 번째 환경은 타겟 DNN(Inception-V3)과 DUNET만 실행되는 경우이다. 두 번째 환경은 앞서 언급한 두 모델을 포함하여 여러 개의 다른 DNN과 함께 수행되는 경우이다. 기본적으로 본 논문의 솔루션(eDenoizer)을 적용하는 경우와 솔루션 없이 PyTorch 프레임워크에 의해 생성된 DNN을 실행하는 경우를 비교한다.

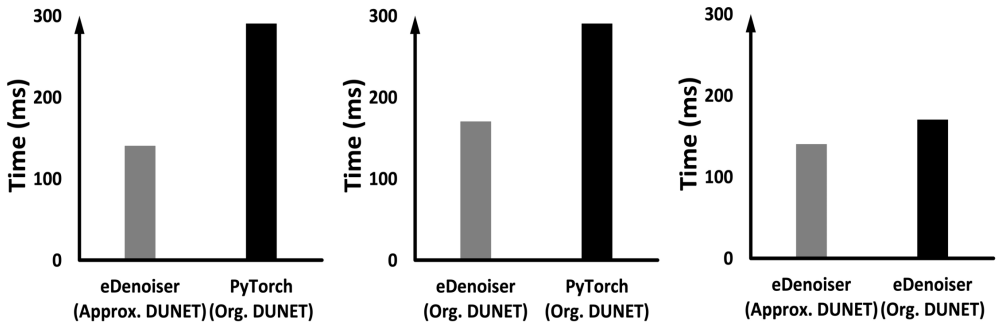
제안된 eDenoizer는 DUNET 연산을 줄이기 위해 터커 분해로 생성된 컨볼루션 커널 텐서를 적용하는 것과 여러 DNN을 병렬 처리할 수 있는 우선순위 기반 GPU 스케줄링 프레임워크의 두 가지 솔루션 부분으로 구성된다. 따라서, 앞에서 설명한 두 실험 환경은 두 가지 측면으로 세분화 된다. ① 두 솔루션 부분을 모두 포함하는 eDenoizer의 전반적인 실행 성능을 측정한다. ② 두 솔루션 부분을 별도로 측정한다. 즉, 솔루션의 각 부분이 전체 실행 성능에 독립적으로 반영되는 방법을 식별한다. 각 실험에서는 DUNET을 포함한 각 DNN에 100개 이상의 데이터를 연속적으로 입력하고 데이터당 추론 실행 시간의 평균을 측정하였다. 각 그래프에 표시된 막대는 DUNET의 잡음 제거 시간과 타겟 DNN의 추론 시간을 추가한 결과이다.

가) 타겟 DNN과 DUNET만 추론 시 시간 비교

[그림 4-8]은 DUNET과 타겟 DNN인 Inception-V3만이 함께 실행 중일 때의 결과이며 (a),(b) 그리고 (c)에서 각각 다른 조건으로 DUNET의 잡음 제거 시간과 타겟 DNN의 추론 시간을 모두 통합한 평균 타겟 추론 시간인 $Inf_{avg}(F)$를 비교한다. [그림 4-8]의 (a)에서 볼 수 있듯이, 본 논문에서 제안한 연산량 감소 기법과 스케줄링 프레임워크를 모두 적용하면 평균 타겟 추론 시간이 51.72% 단축된다. [그림 4-8]의 (b)는 eDenoizer의 스케줄링 프레임워크의 효율성만 별도로 확인하는 경우이며 최대 41.3%의 실행 시간 단축을 달했다. [그림 4-8]의 (c)는 터커 분해 자체의 효과만 보여주며 커널 텐서에 대한 터커 분해는 총 17%의 시간 단축을 이뤘다.

나) 서브 DNN 추론과 동시에 적대적 예제 복원 시 시간 비교

실제 상황에 더 유사한 실험 상황을 나타내기 위해 DUNET과 타겟 DNN 외에도 여러 다른 DNN을 임베디드 시스템에서 함께 실행했다. 적대적 예제 복원에 필요한 DNN 외에도 실행한 다른 DNN들을 서브 DNN이라



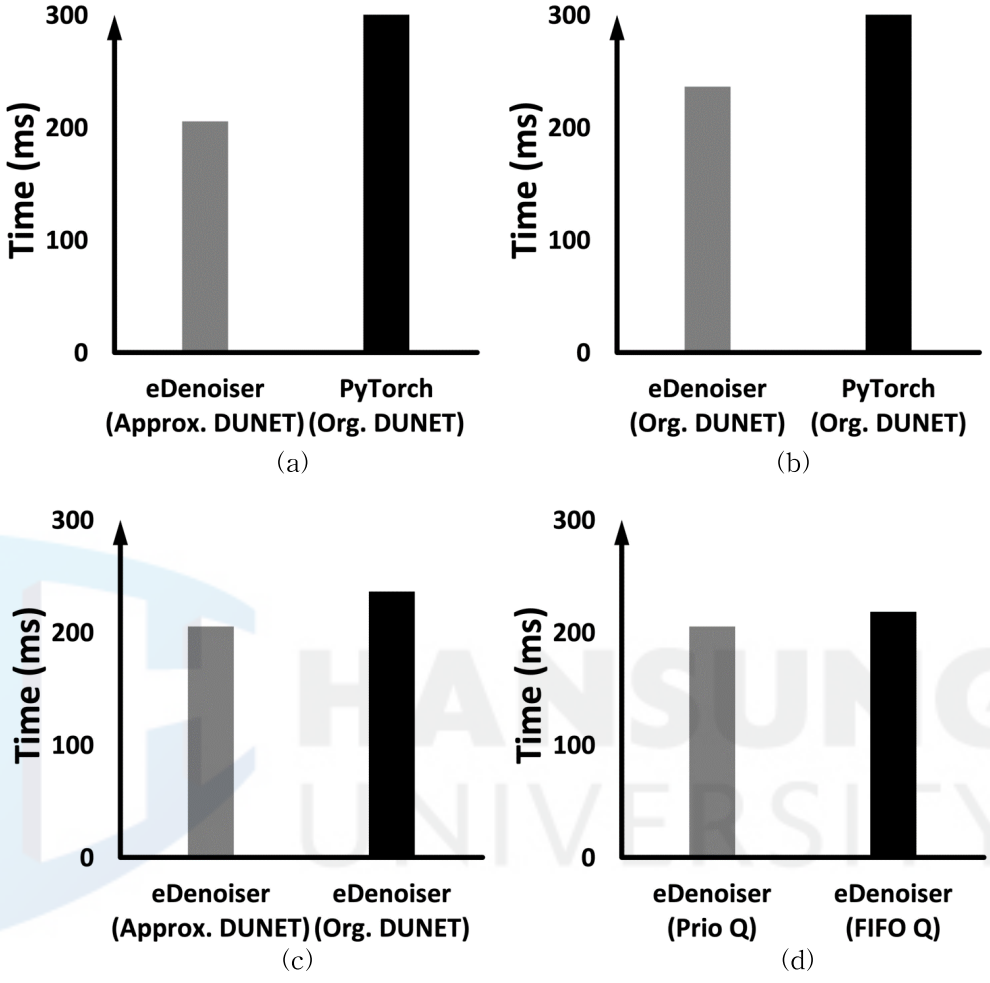
(a) (b) (c)

[그림 4-8] 타겟 DNN과 DUNET만을 실행했을 때의 상황별 수행 시간 비교

하며 서브 DNN는 ResNet-152 1개, RegNet(Radosavovic, I., et al, 2020) 1개, ResNext(Xie, S., et al, 2017) 1개 그리고 WideResNet(Zagoruyko, S., et al, 2016) 1개이다. 본 논문에서는 DUNET과 타겟 DNN인 Inception-V3에 높은 우선순위를 부여했고 나머지 4개의 서브 DNN에는 낮은 우선순위를 부여했다. 실험을 통해 CPU측 OS를 통해 주어진 우선순위가 GPU를 통한 실행에 반영되는지 여부를 확인하며 [그림 4-9]는 [그림 4-8]과 같이 다른 조건 (a), (b), (c) 그리고 (d)에서의 $Inf_{avg}(F)$를 나타낸다.

두 솔루션을 모두 적용하면 [그림 4-9]의 (a)와 같이 평균 타겟 추론 시간이 eDenoizer 적용 시 48.36% 단축된다. 이는 DUNET과 타겟 DNN의 우선순위가 높을 경우 eDenoizer가 함께 실행되는 서브 DNN의 간섭을 억제한다는 것을 분명히 보여준다. eDenoizer를 적용하기 전 대비 DUNET과 타겟 DNN의 성능 향상의 이유는 주로 호스트 측에서 설정한 우선순위가 GPU 실행 순서와 잡 기반 병렬 처리가 효율적으로 반영되었기 때문이다.

멀티 DNN 실행 환경에서 스케줄링 프레임워크 자체의 효과를 별도로 확인하기 위해 eDenoizer에서 터커 분해를 제거한 다음 솔루션을 적용하지 않았을 때와 결과를 비교했다. [그림 4-9]의 (b)는 그 결과를 나타내며, 40.55%의 감소를 보여준다. [그림 4-9]의 (c)는 멀티 DNN 실행 환경에서 터커 분해의 효과를 검증하기 위해 eDenoizer의 스케줄링 프레임워크를 사용



[그림 4-9] 서브 DNN들과 적대적 예제 복원을 실행했을 때의 상황별 수행 시간 비교

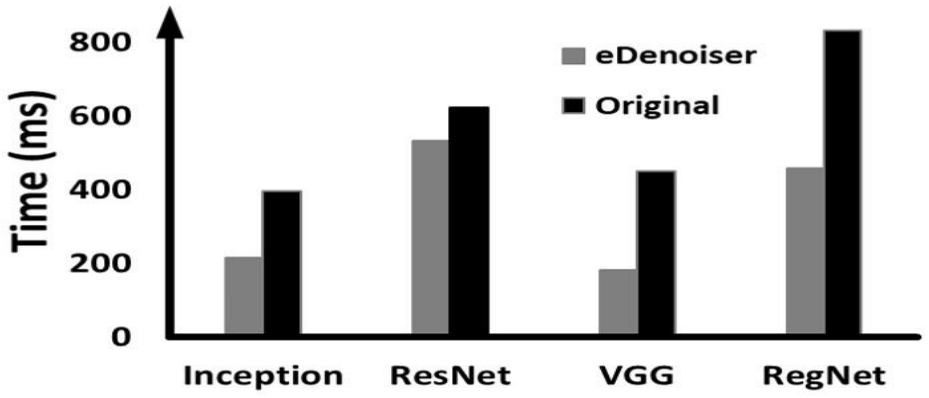
하여 터커 분해가 적용되기 전과 후를 비교한 그래프이며 약 13.13%의 실행 시간 감소가 있음을 알 수 있다. [그림 4-9]의 (d)는 잡큐가 FIFO일 때와 우선순위 방식일 때를 비교한 실험 결과이며 우선순위 큐를 적용했을 때 약 6%의 성능 이점을 볼 수 있다.

다) 타겟 DNN에 터커 분해 적용 시 시간 비교

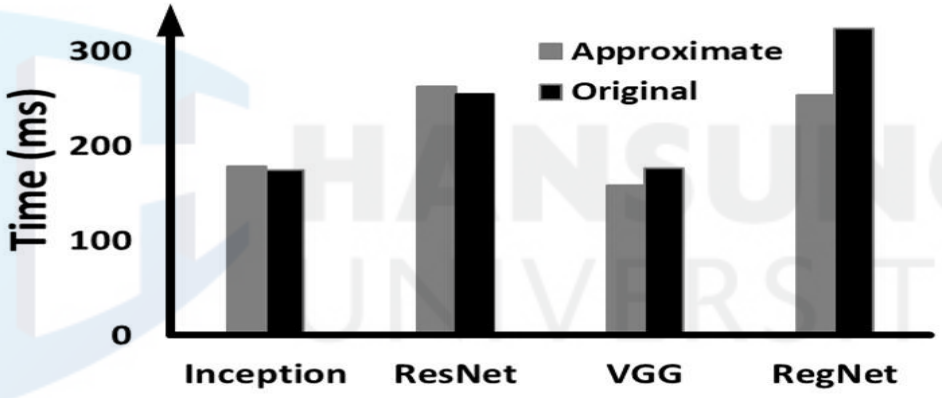
이전 실험에서 터커 분해를 DUNET에 적용하여 연산량을 줄였다면 본

실험에서는 터커 분해를 타겟 DNN에 적용한다. 이를 위해 Inception-V3, ResNet-152, VGG-16 그리고 RegNet을 타겟 DNN으로 선정하고 오프라인에서 터커 분해를 적용한 후 각 타겟 DNN을 DUNET과 결합하여 타겟 DNN별 실행 시간을 측정한다. [그림 4-10]은 eDenoizer의 스케줄링 프레임워크를 사용하고 타겟 DNN에 터커 분해 적용의 유무 차이를 측정한 그래프이다. Approximate는 터커 분해를 타겟 DNN에 적용한 것이고 Original은 터커 분해를 적용하지 않은 것에 대한 결과이다. Inception-V3 및 ResNet-152의 경우 터커 분해 적용 시 성능이 저하된다. Inception-V3를 구성하고 있는 인셉션(inception) 모듈과 ResNet-152를 구성하고 있는 보틀넥(bottleneck) 블록은 차원 축소를 위해 1×1과 3×3 커널 텐서로 구축되어 있으며 인셉션 모듈의 3×3 컨볼루션 커널 텐서와 보틀넥 블록은 이미 충분히 작은 출력 차원을 가진다. 따라서 터커 분해를 이러한 3×3 커널 텐서에 적용하면 1×1 커널 텐서만 새로 추가되므로 전체 연산 규모가 오히려 증가한다. 그 결과 [그림 4-10]과 같이 추론 시간이 지연된다. 반면 상대적으로 구조가 다른 VGG-16과 RegNet을 타겟 DNN으로 사용할 경우 최대 21.6%의 성능 향상 효과가 있다.

다음 실험은 하나의 임베디드 GPU를 통해 서로 다른 우선순위를 가진 여러 DNN들이 실행될 때의 성능을 검사하며 서브 DNN 종류와 우선순위 할당 방법은 바로 전 실험과 동일한 조건이다. [그림 4-10]의 결과에 따라 터커 분해는 Inception-V3와 ResNet-152에는 적용하지 않고 VGG-16과 RegNet에만 적용하며 [그림 4-11]은 실험 결과를 보여주는 그래프이다. [그림 4-11]에서 확인할 수 있듯이, VGG-16이 타겟 DNN일 때 eDenoizer는 적대적 예제 복원 및 추론에 필요한 시간을 59.86% 줄일 수 있다. 결론적으로 멀티 DNN이 함께 실행되는 더 실질적인 조건에서, 제안된 eDenoizer는 적대적 예제 복원이 수반되는 추론 시간 측면에서 성능 향상을 효과적으로 보장한다.



[그림 4-10] 타겟 DNN과 DUNET에 터커 분해를 적용 했을 때의 수행 시간 비교



[그림 4-11] 터커 분해를 선택적으로 적용한 타겟 DNN별 적대적 예제 복원 수행 시간 비교

5) eDenoizer의 메모리 감소

터커 분배로 연산량이 감소하여 DNN 동작을 수행하는 GPU의 부담을 줄여 실행 시간이 빨라진다. 또한 터커 분해는 DNN의 매개 변수 수를 감소시킨다. 본 실험에서는 메모리 풋프린트(footprint) 감소를 측정한다. Jetson AGX Xavier와 Jetson TX2 등 임베디드 시스템에 사용되는 GPU는 별도의 전용 메모리가 없고 시스템 메모리인 DRAM을 CPU와 공유하므로 메모리 절약은 매우 중요하다.

	Inception-V3	ResNet-152	VGG-16	RegNet	DUNET
Original	105 MB	231 MB	528 MB	555 MB	43 MB
Approximate	59 MB	144 MB	481 MB	404 MB	25 MB

[표 4-6] 터커 분해로 인한 메모리 감소

[표 4-6]는 메모리 감소 결과를 나타내며 Approximate는 터커 분해를 적용한 경우를 나타내며 Original은 터커 분해를 적용하지 않았을 때를 나타낸다. 먼저 DUNET의 경우 터커 분해를 적용할 때 메모리 사용량을 약 18% 줄이는 효과가 있으며 타겟 DNN 중에서는 Inception-V3가 43%로 가장 많은 감소를 이뤘다. 그러나 [그림 4-10]에 나타난 바와 같이 Inception-V3와 ResNet-152는 터커 분해를 적용하면 실행 시간이 오히려 증가하므로 실행 시간 및 메모리 풋프린트 감소를 동시에 고려할 때 VGG-16에 9%, RegNet에 27.2%의 효과가 있다고 할 수 있다.

제 5 장 결론

적대적 공격은 임베디드 시스템에서의 DNN 분류를 고의적으로 다르게 하여 치명적인 결과를 초래할 수 있다. 이를 방지하기 위해 임베디드 시스템에서도 적대적 공격에 대응할 수 있는 시스템을 적용할 수 있어야한다. 본 논문에서는 적대적 공격 대응 중, 적대적 예제 탐지와 적대적 예제 복원을 임베디드 시스템에서 실질적으로 실행하기 위한 솔루션을 각각 제안하였다.

임베디드 시스템에서 적대적 예제 탐지를 위해 타겟 DNN의 추론 시간과 NIC를 활용한 적대적 예제 탐지 소요 시간의 차이를 최대한 줄일 수 있는 임베디드 NIC 시스템을 제안하였으며 제한된 메모리를 가지는 임베디드 시스템에 맞춰 타겟 DNN의 모든 은닉층에서 적대적 예제 탐지를 진행하는 방식을 특정 은닉층에서만 수행하여 적대적 예제 탐지에 필요한 OCSVM과 DM의 개수를 크게 줄임과 동시에 적대적 예제 탐지 정확도를 유지하는 방법을 제안하였다. 임베디드 NIC 시스템은 멀티 프로세스 NIC 시스템 보다 전체 소요 시간이 평균적으로 약 93.6% 감소하였으며 최대 메모리 사용량은 평균 37.6% 줄어들어 메모리가 제약된 임베디드 시스템에서도 타겟 DNN의 종류와 무관하게 적대적 예제 탐지가 가능함을 보였다.

적대적 예제 복원은 타겟 DNN의 추론 단계가 잡음을 제거한 후 진행되기 때문에 전반적인 추론 과정에서 즉각적인 잡음 제거가 수반되어야 하며 이는 임베디드 시스템에서도 마찬가지다. 그러나 임베디드 시스템에서 여러 DNN은 FIFO 순으로 DNN 커널을 처리하는 임베디드 GPU를 공유하고 사용하며 이는 비선제적 하드웨어이기 때문에 시간 제약이 있는 적대적 예제 복원은 매우 어렵다. 이를 해결하기 위해 본 연구에서는 eDenoizer를 제안하였다. 먼저 잡음을 제거하는 DUNET의 컨볼루션 커널 텐서를 25.41% 분해하여 연산량을 효율적으로 줄인다. eDenoizer의 스케줄링 프레임워크는 파이

썬 실행 환경에서는 불가능한 각 DNN 연산 작업을 가장 적절한 CUDA 스트림 중 하나에 할당할 수 있는 C++ 기반 멀티 스레딩 기법을 제안한다. 또한 동시에 같이 실행되는 낮은 우선순위 DNN의 간섭을 최소화하기 위해 사용자가 지정한 우선순위를 CPU 측뿐만 아니라 GPU 연산 순서에도 반영할 수 있다. 제안된 기법을 통해 잡음을 제거하는 DNN과 타겟 DNN을 효율적으로 병렬화 하고 GPU에서 우선적으로 실행되며 다양한 실험을 통해 분석한 결과 분류 정확도의 감소는 1.78%로 무시할 수 있는 수준이었으며, 추론 시간에서 얻어진 속도 향상은 최대 51.72%였다.

참 고 문 헌

1. 국외문헌

Ajitsaria, A. What is the python global interpreter lock(GIL). Available online: https://realpython.com/python-gil

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

Carlini, N. & Wagner, D. (2017), Toward evaluating the robustness of neural networks, IEEE Symposium on security and privacy, 39-57.

Cheng, S., Dong, Y., Pang, T., Su, H. & Zhu, J. (2019), Defense against adversarial attacks with a transfer-based prior. Advances in Neural Information Processing Systems. 32, 10932-10942.

Goodfellow, I.J., Shlens, J. & Szegedy, C. (2014), Explaining and harnessing adversarial examples, arXiv:1412.6572.

Harris, M. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. Available online: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency

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

He, K., Res, S., Zhang, X. & Sun, J. (2016), Identity mappings in deep residual networks, European conference on computer vision, 630-645

Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M & Adam, H. (2017), Mobilenets:

Efficient convolutional neural networks for mobile vision applications, arXiv:1704.04861.

Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J. & Keutzer, K. (2016), SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5MB model size, arXiv:1602.07360.

ImageNet. Availalbe online: https://paperswithcode.com/dataset/imagenet

Karol, M., Hluchyj, M & Morgan, S. (1987), Input versus output Queueing on a space-division packet switch, IEEE transactions on communications, 35, 1347-1356.

Krizhevsky, A. & Hinton, G. (2009), Learning multiple layers of features from tiny images.

Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2017), Imagenet classification with deep convolutional neural networks, 60(6), 84-90.

Liao, F., Liang, M., Dong, Y., Pang, T., Hu, X. & Zhu, J. (2018), Defense against adversarial attacks using high-level epresentation guided denoiser, 1778-1787.

Lim,C. & Kim, M. (2021), ODMDEF: On-device multi-DNN execution framework utilizing adaptive layer-allocation on general purpose cores and accelerators, IEEE Access, 9, 85403-85417.

Ma, S. & Liu, Y. (2019), Nic: Detecting adversarial samples with neural network invariant checking. Network and distributed system security symposium(NDSS).

McDaniel, P. (2017), Ensemble adversarial training: Attacks and defenses, arXiv:1705.07204.

NVIDIA. CUDA Streams: Best practice and common pitfalls. Available online: https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

NVIDIA. DRIVE OS Linux. Available online: https://docs.nvidia.com/drive/d

rive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html

NVIDIA. World's smallest AI supercomputer: Jetson Xavier NX. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

NVIDIA CUDA Deep Neural Network(cuDNN). Available online: https://developer.nvidia.com/cudnn

NVIDIA CUDA Toolkit-Free Tools and Training. Available online: https://developer.nvidia.com/cuda-toolkit

NVIDIA Jetson AGX Xavier Developer Kit. Available online: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

NVIDIA Jetson TX2 Developer Kit. Available online: https://developer.nvidia.com/embedded/jetson-tx2-developer-kit

Python. Multiprocessing: Process-based parallelism. Available online: docs.python.org/3.6/library/multiprocessing.html

Pytorch. Available online: https://pytorch.org

Radosavovic, I., Kosaraju, R.P., Girshick, R., He, K. & Dollar, P. (2020), Designing network design spaces, IEEE/CVF conference on computer vision and pattern recognition, 10428-10436.

Rashid, K.M. & Louis, J. (2019), Times-series data augmentation and deep learning for construction equipment activity recognition. Advanced Engineering Informatics, 42, 100944.

Ronneberger, O., Fischer, P. & Brox, T. (2015), U-net: Convolutional networks for biomedical image segmentation. International conference on medical image computing and computer-assisted intervention, 234-241.

Scholkopf, B., Platt, J.C., Taylor, J.S., Smola, A.J. & Williamson R.C. (2001), Estimating the support of a high-dimensional distribution.

Neural computation, 13(7), 1443-1471.

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

Singh, A., Sengupata, S. & Lakshminarayanan, V. (2020), Explainable Deep Learning Models in Medical Image Analysis. Journal of Imaging, 6, 52.

Su, J., Vargas, D.V. & Sakurai, K. (2019), One pixel attack for fooling deep neural networks. IEEE Transactions on Evolutionary Computation. 23, 828-841.

Szegedy, C., Ioffe, S., Vanhoucke, V. & Alemi, A.A. (2017), Inception-v4, Inception-ResNet and the impack of residual connections on learning, AAAI conference on artificial intelligence, 4-9.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shens, J. & Wojna, Z. (2016), Rethinking the inception architecture for computer vision. IEEE conference on computer vision and pattern recognition, 2818-2826.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. & Fergus, R. (2013) Intriguing properties of neural networks, arXiv:1312.6199.

TensorFlow. Available online: https://www.tensorflow.org

Tramer, F., Kurakin, A., Papernot, N., Goodfellow, I., Boneh, D. & McDaniel, P. (2017), Ensemble adversarial training: Attacks and defenses, arXiv:1705.07204.

Tucker, L.R. (1966), Some mathematical notes on three-mode factor analysis, Psychometrika, 31, 279-311.

Vincent, P., Larochelle, H., Bengio, Y. & Manzagol, P.A. (2008), Extracting and composing robust features with denoising autoencoders. International conference on machine learning,

1096-1103.

Xiang, Y. & Kim, H. (2019), Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference, IEEE real-time systems symposium, 392-405.

Xie, S., Girshick, R., Dollar, P., Tu, Z. & He, K. (2017), Aggregated residual transformations for deep neural networks, IEEE conference on computer vision and pattern recognition, 1492-1500.

Zagoruyko, S. & Komodakis, N. (2016), Wide residual networks, arXiv:1605.07146.

Zhu, H. (2022), MetaAID: A flexible framework for developing metaverse applications via AI technology and human editing. arXiv:2204.01614.

ABSTRACT

Research on Software-Based Adversarial Attack Defense for Embedded Systems

Joo, Sang-Hyun

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

Recently, the types of adversarial attacks that induce deliberately incorrect classification results from DNN results are becoming more sophisticated and diversified. Accordingly, DNNs are frequently exposed to adversarial attacks, and the result of misclassification due to adversarial attacks in embedded systems, such as drones and autonomous driving systems, can be fatal. However, it is very difficult for embedded systems to quickly detect or defend against adversarial attacks due to the limited performance and memory capacity of DNN computing system resources. If dedicated hardware is developed for this purpose, there is a burden in terms of cost and it is difficult to flexibly cope with changes in the attack method and the target DNN. To solve this problem, this paper introduces each technique for defense through adversarial attack detection and adversarial example restoration in embedded systems. For adversarial attack detection, we propose embedded NIC system, which is a software-based adversarial attack detection technique. This solution selects the hidden layer to proceed with detection among the hidden layers of the target DNN in order to minimize the memory required for attack detection. In addition, when the target DNN inference is in

progress, it is detected in parallel and the running time gap between the two is minimized. For adversarial example restoration, eDenoizer is proposed. eDenoizer reduces the amount of computation required for the DNN convolutional kernel tensor, which removes adversarial perturbation, by applying Tucker decomposition. Furthermore, when restoration is performed simultaneously with other DNNs, the priority assigned in the host CPU is delivered to the GPU, so that the adversarial defense can be performed preferentially. As a result of the experiment, the embedded NIC system can reduce the difference in execution time by up to 99.6% and reduces memory usage by 83.9% compared to before applying the proposed solution, and eDenoizer is able to reduce the inference speed accompanied by adversarial example restoration by 51.72% with a slight reduction in classification accuracy of 1.78%.

【Key words】 Adversarial Attack Detection, Adversarial Attack Defence, Embedded System