엔비디아 젯슨과 TensorRT 기반 로봇 머신러닝 모델 최적화

로봇 및 에지 AI를 위한 최신 모델 경량화·최적화 기술

목차

도입 및 문제의식

서버 환경과 로봇 환경의 차이

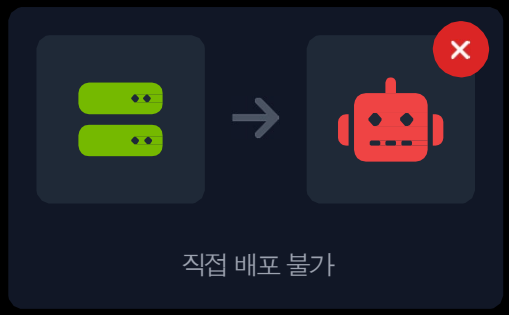
고성능 서버 환경에서 개발된 머신러닝 모델을 제품(로봇)에 바로 배포할 수 없는 현실적 문제에 직면하고 있습니다. 이는 두 환경 간의 근본적 차이에서 비롯됩니다.

컴퓨팅 파워 격차
서버: 고성능 GPU, 대용량 메모리, 무제한 전력 vs. 로봇: 제한된 연산 능력, 배터리 의존

전력 소비와 발열 문제
대형 모델은 과도한 전력 소모와 발열 발생, 로봇의 배터리 수명 단축

모델 크기와 메모리 제약
대용량 모델은 로봇의 제한된 메모리에 부담, 로딩 지연 발생

실시간 응답 필요성
로봇은 실시간 의사결정 필요, 고지연 모델은 안전 위험 초래



모델 경량화의 필요성

에지 디바이스의 제약 조건을 고려한 모델 최적화는 필수적입니다. 특히 양자화(Quantization)를 통한 경량화는 효과적인 접근법입니다.

경량화 이점: 추론 속도 향상, 메모리 사용량 감소, 배터리 효율성 개선, 발열 감소, 실시간 응답성 확보

에지 AI와 로봇 개발의 주요 도전과제

로봇용 에지 디바이스의 기술적 제약

내구성 확보의 어려움
실외 환경의 충격, 진동, 극한 온도(-20°C~50°C), 습도(10%~95%) 등 다양한 환경 조건에 대응해야 함. 일반 서버룸 환경(20°C±2)과 완전히 다른 환경적 요구사항.

배터리 수명과 발열 문제
고성능 AI 모델 구동 시 과도한 전력 소모(20~100W)와 발열 발생. 이동형 로봇의 경우 배터리 수명 제약으로 인한 연속 작동 시간 감소. 냉각 시스템 추가 시 크기와 무게 증가.

소형 부품과 하드웨어 제약
로봇 플랫폼에 탑재 가능한 크기(10cm×10cm 내외)의 부품 필요. 서버급 하드웨어(GPU, 메모리, 전력공급장치)를 소형화하는 과정에서 연산 능력 저하 불가피.

실시간 처리 요구
자율주행, 장애물 회피 등의 로봇 작업은 밀리초(ms) 단위의 빠른 응답 필요. 대형 모델의 추론 지연(latency)은 안전과 성능에 직결되는 문제.

다중 센서 통합의 복잡성

카메라 라이다 레이더 초음파

다양한 센서 데이터를 실시간으로 처리하고 융합해야 하며, 각 센서별 드라이버 및 인터페이스 호환성 확보 필요

서버 vs 에지 디바이스 비교

사양	서버 환경	에지 디바이스
연산 성능	400+ TOPS	10~100 TOPS
메모리	64~1TB	4~32GB
전력 소비	300~1000W	5~50W
크기/무게	랙 마운트	손바닥 크기

이러한 제약 극복을 위한 접근법

모델 경량화와 최적화를 통해 에지 디바이스의 한계 내에서 AI 성능 발휘 필요

→ 엔비디아 Jetson 플랫폼과 TensorRT 최적화를 통한 문제 해결

NVIDIA Jetson 플랫폼 개요

임베디드 AI 컴퓨팅 플랫폼

NVIDIA Jetson은 소형 장치에서 고성능 AI 추론, 컴퓨터 비전, 로보틱스, 자율주행 등을 구현할 수 있도록 설계된 임베디드 AI 컴퓨팅 플랫폼입니다. 작은 폼팩터에 강력한 GPU 성능을 담아 에지 컴퓨팅 환경에 최적화되어 있습니다.

Jetson
Edge AI Platform

주요 활용 분야

- 로보틱스
- 자율주행
- 스마트시티
- 산업 자동화
- 드론/UAV
- 헬스케어

GPU 가속

NVIDIA CUDA 코어와 Tensor 코어를 탑재하여 병렬 연산 가속화. 최대 수백 TOPS의 AI 성능 제공

DLA 탑재

Deep Learning Accelerator로 딥러닝 워크로드 가속화, 저전력 고효율 추론 지원

JetPack SDK

AI, 컴퓨터 비전, 멀티미디어 등 종합 개발 도구와 라이브러리 제공. CUDA, TensorRT, OpenCV, VPI 지원

저전력 설계

5W~50W 전력 소모로 배터리 구동 장치, 발열 제한 환경에 이상적. 에너지 효율 최적화

소형 폼팩터

컴팩트한 크기(69.6mm x 45mm~100mm x 87mm)로 공간 제약이 있는 장치에 통합 용이

풍부한 생태계

다양한 센서 및 주변기기 연결, DeepStream, Isaac SDK 등 특화 솔루션, 커뮤니티 지원

Jetson 제품군 비교

제품	AI 성능 (TOPS)	GPU	CPU	메모리	전력 소비	주요 활용 분야
Jetson Orin Nano	40-67	Ampere 1024 CUDA 코어	6코어 Arm Cortex-A78AE	8GB LPDDR5 102GB/s	7-15W	엔트리급 로봇, IoT
Jetson Xavier NX	21	Volta 384 CUDA 코어	6코어 NVIDIA Carmel Arm v8.2	8GB LPDDR4x 59.7GB/s	10-15W	중소형 로봇, 드론
Jetson AGX Xavier	32	Volta 512 CUDA 코어	8코어 NVIDIA Carmel Arm v8.2	16/32GB 137GB/s	10-30W	고성능 로봇, 산업
Jetson AGX Orin	275	Ampere 2048 CUDA 코어	12코어 Arm Cortex-A78AE	32GB LPDDR5 204GB/s	15-60W	자율주행, 로보틱스
Jetson Nano	0.5	Maxwell 128 CUDA 코어	4코어 Arm Cortex-A57	4GB LPDDR4 25.6GB/s	5-10W	초소형 장치, 교육

엔트리 레벨

Jetson Nano & Orin Nano: 저전력, 소형 폼팩터에서 머신러닝 추론 수행, 초기 개발자 및 가격에 민감한 애플리케이션에 적합

중급 성능

Jetson Xavier NX: 중급 성능과 에너지 효율성의 균형, 드론, 소형 로봇, 스마트 카메라 등에 최적화

최고 성능

Jetson AGX Xavier & Orin: 최고급 AI 성능, 자율주행, 산업용 로봇, 의료 장비 등 고성능 요구 분야에 적합

Jetson 활용 사례

NVIDIA Jetson 플랫폼은 다양한 산업 분야에서 에지 AI 및 로보틱스 솔루션으로 활용되고 있습니다. 아래는 Jetson이 실제 적용되고 있는 주요 활용 사례입니다.

자율주행 로봇

실시간 인식과 의사결정이 필요한 실외 자율주행 로봇에 최적화된 에지 컴퓨팅 플랫폼 제공

- 라스트마일 배송 로봇
- 농업용 로봇
- 물류 운송 로봇
- 드론

스마트시티

도시 인프라 모니터링 및 교통 관리를 위한 실시간 영상 분석 및 AI 기반 의사결정 시스템

- 지능형 교통 시스템
- 보안 감시 카메라
- 주차 관리
- 에너지 관리

헬스케어

의료 영상 분석, 환자 모니터링, 의료용 로봇 등 의료 분야에서의 AI 활용 확대

- 진단 이미징 디바이스
- 수술 보조 로봇
- 환자 케어 로봇
- 웨어러블 모니터링

산업 자동화

생산 라인 자동화, 품질 검사, 예측 유지보수 등 스마트 팩토리 분야에서 활용

- 불량품 검출 시스템
- 산업용 로봇
- 설비 모니터링
- 재고 관리

모델 경량화와 양자화 필요성

양자화(Quantization)란 무엇인가?

양자화는 머신러닝 모델의 연산 과정에서 사용되는 가중치와 출력 텐서의 자료형을 높은 정 밀도에서 낮은 정밀도로 변환하는 경량화 방법입니다.

일반적으로 32비트, 16비트 부동소수점(FP32, FP16)을 8비트, 4비트 정수(INT8, INT4)로 변환합니다.

연산 속도 향상
정수 연산은 부동소수점 연산보다 빠르며, 더 적은 메모리 대역폭 사용

모델 크기 축소
INT8로 변환 시 모델 크기가 FP32 대비 약 4배 감소, 저장 공간 효율성 증가

전력 소비 감소
정밀도가 낮을수록 연산당 소모 전력이 감소, 배터리 수명 연장

하드웨어 가속 활용
INT8 연산은 많은 에지 디바이스의 하드웨어 가속기에 최적화됨

정밀도 변환 과정

FP32 (32비트 부동소수점)
↓
FP16 (16비트 부동소수점)
↓
INT8 (8비트 정수)
↓
INT4 (4비트 정수)

정밀도 감소에 따른 메모리 사용량 및 연산 속도 개선

에지 디바이스 최적화를 위한 양자화

로봇, 드론, IoT 장치와 같은 에지 디바이스는 제한된 컴퓨팅 파워, 메모리, 배터리를 가지고 있어 모델 경량화가 필수적입니다.

- 양자화 적용 시 추론 속도 2~4배 향상
- 모델 크기 75% 이상 감소 가능
- 적절한 캘리브레이션으로 정확도 손실 최소화

"양자화는 에지 디바이스에서 AI 모델 실행의 핵심 기술로, 로봇과 같은 제한된 환경에서 고성능 AI 모델 활용을 가능하게 합니다."

양자화(PTQ, QAT) 개념 및 비교

양자화 방식 비교

훈련 후 양자화 (PTQ)

이미 훈련된 모델에 대해 별도의 추가 학습 없이 일부 대표 데이터만으로 가중치와 활성화 값을 낮은 비트로 변환하는 방식

✓ 빠른 적용 ✓추가 학습 불필요 ✓캘리브레이션 데이터만 필요

△ 정확도 하락 가능성

양자화 인식 훈련 (QAT)

훈련 중에 양자화 효과를 시뮬레이션하여 양자화에 적응된 가중치를 학습시키는 방식으로 정확도 손실을 최소화

✓ 더 높은 정확도 유지 ✓최적화된 가중치 학습 △ 전체 데이터 재학습 필요

△ 시간과 자원 많이 소모

정밀도 비교: FP32 vs FP16 vs INT8

정밀도	메모리 사용	속도	정확도
FP32 (32비트 부동소수점)	높음 (기준)	1× (기준)	높음 (기준)
FP16 (16비트 부동소수점)	~50% 감소	~1.5-2× 향상	거의 유지 (~0.5% 감소)
INT8 (8비트 정수)	~75% 감소	~3-4× 향상	감소 (캘리브레이션 품질에 따라 0.5~3%)

양자화 워크플로우 비교

TensorRT 최적화 기법

핵심 최적화 기술

TensorRT는 다양한 최적화 기법을 사용하여 추론 속도를 대폭 향상시키고 메모리 사용량을 줄입니다. 이러한 최적화는 특히 제한된 리소스를 가진 에지 디바이스에서 중요합니다.

그래프 융합(Graph Fusion)
여러 레이어를 단일 최적화된 계층으로 결합하여 메모리 액세스와 커널 호출 수 감소. 예: Conv+Bias+ReLU를 단일 연산으로 통합

커널 자동 튜닝(Kernel Auto-Tuning)
특정 GPU 아키텍처, 텐서 크기, 데이터 타입에 최적화된 커널 구현을 자동으로 선택, 타겟 하드웨어에 맞춘 최적의 성능 제공

다양한 정밀도 지원(Mixed Precision)
FP32, FP16, INT8 정밀도를 혼합 사용하여 정확도와 성능 간 최적의 균형 제공. 양자화 적용시 8-bit 정수 연산으로 속도 향상

동적 텐서 메모리(Dynamic Tensor Memory) 메모리 사용을 최적화하기 위해 텐서 버퍼를 동적으로 할당 및 해제하여 메모리 공간 효율적 사용

모델 변환 경로

TensorRT 변환 주요 경로는 두 가지가 있으며, 각 상황에 맞게 선택할 수 있습니다.

1. PyTorch → ONNX → TensorRT

PyTorch ▶ ONNX ▶ TensorRT

장점: 범용성 높음, 하드웨어 독립적, 풍부한 문서 제공
제약: 일부 연산 변환 이슈, 모델 구조에 따라 변환 복잡성 증가

ONNX(Open Neural Network Exchange)

2. TorchScript → Torch-TensorRT

TorchScript ▶ Torch-TensorRT

장점: PyTorch 코드와의 일관성, 연구 및 빠른 실험에 적합
제약: 일부 버전/연산 제약, TorchScript 변환 문제 발생 가능

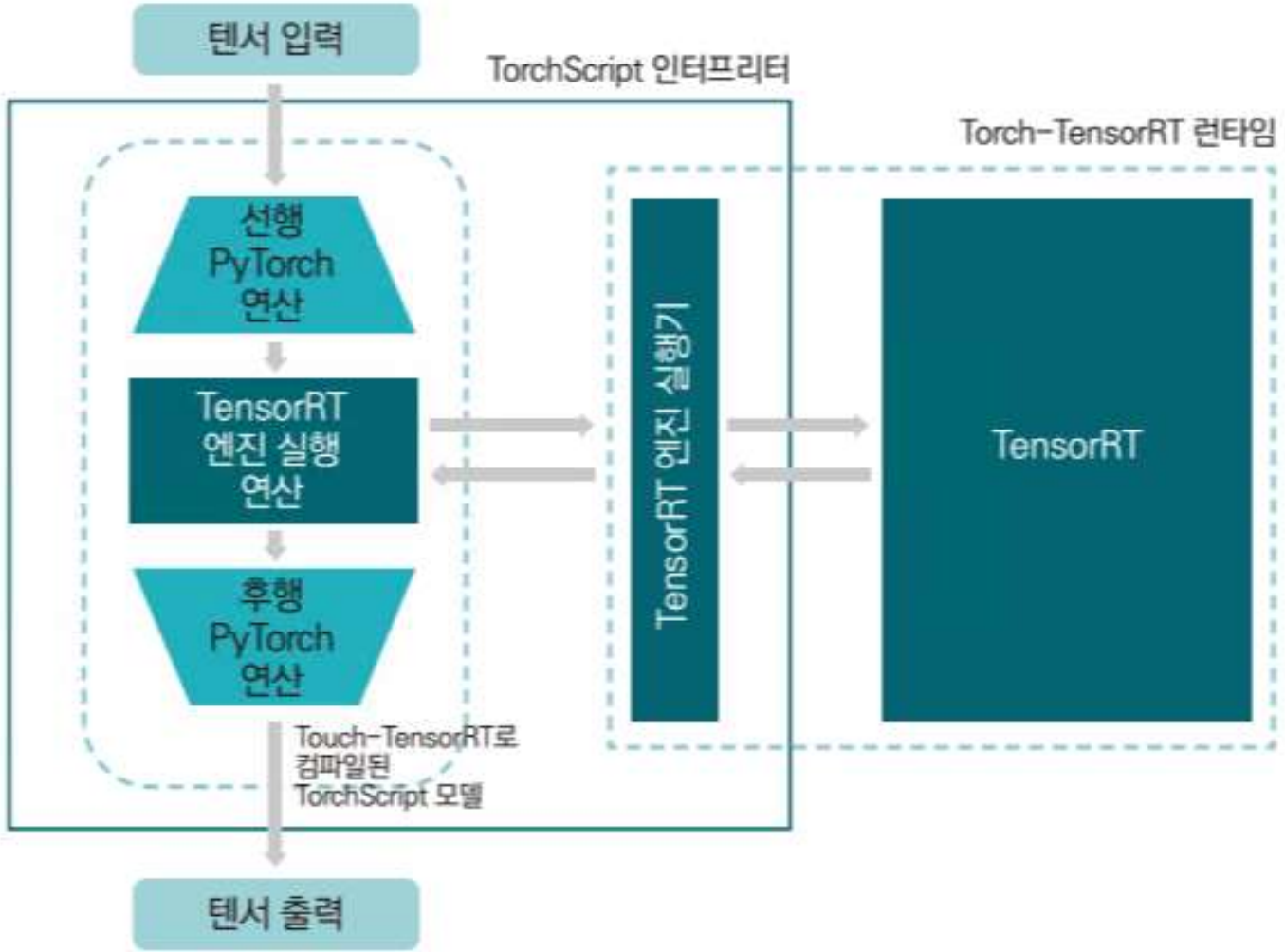
TensorRT 최적화 기법

엔비디아(NVIDIA) 하드웨어, 특히 Jetson 시리즈 (Nano, Xavier, Orin 등) 에서는 TensorRT를 사용하는 것이 사실상 필수적이라고 볼 수 있음

Jetson 하드웨어와 TensorRT는 최적화된 조합

- Jetson은 NVIDIA GPU + CUDA + Tensor Cores 를 탑재한 임베디드 보드
- **TensorRT는 이 GPU 아키텍처를 가장 효율적으로 활용하도록 엔비디아가 직접 개발한 추론 최적화 라이브러리**
- 따라서 Jetson에서 최고의 성능과 전력 효율을 내려면 TensorRT가 사실상 표준 솔루션

• Torch-TensorRT의 런타임 구조* •



TensorRT 실전 워크플로우

PyTorch → ONNX → TensorRT 변환 과정

- **PyTorch → ONNX 변환**
  torch.onnx.export() 함수 사용, 입출력 크기 명시, 동적 크기 지원 설정

```
torch.onnx.export(model, dummy_input, "model.onnx", input_names=["input"],
    output_names=["output"], dynamic_axes={"input": {0: "batch_size"}, "output":
    {0:
"batch_size"}})
```

- **ONNX → TensorRT 변환**
  Polygraphy를 통한 TensorRT 엔진 생성, INT8 양자화 적용

Torch.onnx.export를 이용하면 파이토치의 모델을 간단하게 ONNX 모델로 변환할 수 있다.

```
import torch
import torchvision

# Load ResNet-18 ImageNet-pretrained model using torchvision.
model = torchvision.models.resnet18(pretrained=True)

model.eval()

# Dummy input with the model input size.
dummy_input = torch.randn(1, 3, 224, 224)

# Save ONNX model to ONNX_PATH.
torch.onnx.export(model,
                  dummy_input,
                  ONNX_PATH,  모델을 저장할 경로
                  opset_version=13, ONNX의 연산자 버전
                  input_names=["input"],
                  output_names=["output"])
```

TensorRT 실전 워크플로우

Polygraphy 도구 활용

Polygraphy는 NVIDIA에서 만든 오픈소스 Python 툴킷

ONNX / TensorRT / Deep Learning 추론 관련 모델을 분석·변환·검증할 수 있도록 지원하는 라이브러리

(TensorRT 모델 개발/디버깅/테스트를 편하게 해주는 멀티툴)

TensorRT는 고성능이지만 API가 복잡하고 디버깅이 어려움!!!

- **Polygraphy Calibrator 클래스 구현**
  IInt8Calibrator 상속, 캘리브레이션 데이터셋 준비
- **TensorRT 엔진 빌드**
  Polygraphy를 통한 TensorRT 변환 코드 예시

```
from polygraphy.backend.trt import CreateConfig,
EngineFromNetwork
config = CreateConfig(int8=True, calibrator=calibrator) engine =
EngineFromNetwork(network_from_onnx_path("model.onnx"),
    config=config)
```

- **엔진 저장 및 로드**
  최적화된 TensorRT 엔진을 파일로 저장하고 필요시 로드

순서1) Calibrator 객체 생성

```
# Preprocess of ResNet-18 training.
transform = transforms.Compose([transforms.Resize((256, 256)),
                                transforms.CenterCrop((224, 224)),
                                transforms.ToTensor(),
                                transforms.Normalize([0.485, 0.456,
0.406],
                                                     [0.229, 0.224,
0.225])])

# Polygraphy needs a generator-type data loader.
val_list = os.listdir(IMAGE_DIR)
def data_generator():
    for image in val_list:
        # Preprocess.
        image_path = os.path.join(IMAGE_DIR, image)
        image = transform(Image.open(image_path).convert("RGB"))
        # Add batch dimension.
        image = image.unsqueeze(0)
        # Polygraphy uses numpy input.
        image = image.numpy()
        # Dict key must be the same as ONNX input name.
        yield {"input": image}

calibrator = poly_trt.Calibrator(data_loader=data_generator())
```

평균/표준편차로 정규화

데이터 제너레이터
Polygraphy에서 쓰기 위해,
이미지를 한 장씩 불러와
numpy 형태로 바꿈

Polygraphy의 Calibrator 객체에 data를 전달해서 양자화 시 calibration 데이터 로더로 활용

TensorRT 실전 워크플로우

순서2) ONNX 파일 로드 및 IBuilderConfig 객체 생성

```
builder, network, parser =
poly_trt.network_from_onnx_path(path=ONNX_PATH)

# Each type flag must be set to true.
builder_config = poly_trt.create_config(builder=builder,
                                        network=network,
                                        int8=True,
                                        fp16=True,
                                        calibrator=calibrator)
```

1. ONNX 파일을 TensorRT 네트워크로 불러옴
2. BuilderConfig를 만들어 FP16 / INT8 최적화를 동시에 사용

성능과 정확도의 균형을 맞추기 위함

TensorRT는 연산 단위별로 효율적인 모드를 자동 선택
- INT8로 안전하게 변환 가능한 레이어 → INT8 사용 (최대 성능)
- INT8 변환 시 정확도 손실이 크면 FP16 사용 (정확도 보존)

순서3) 최종적으로 TensorRT 엔진을 빌드하고, 저장하는 과정

```
engine = poly_trt.engine_from_network(network=(builder, network,
parser),
                                      config=builder_config)
```

네트워크와 설정을 바탕으로 최적화된 TensorRT 엔진 생성

```
# TensorRT engine will be saved to ENGINE_PATH.
poly_trt.save_engine(engine, ENGINE_PATH)
```

TensorRT 실전 워크플로우

순서4) 저장된 TensorRT 엔진을 불러와서 추론(inference)하는 과정

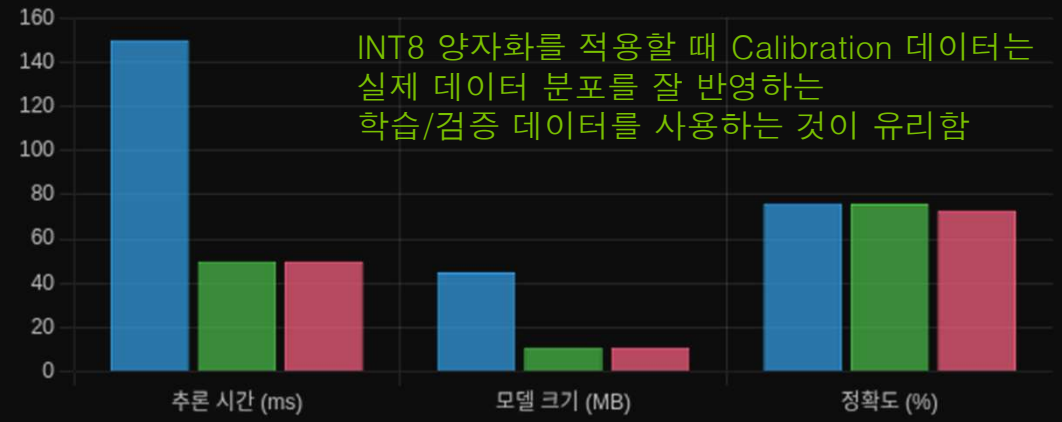
```
# Load serialized engine using 'open'.
engine = poly_trt.engine_from_bytes(open(ENGINE_PATH, "rb").read())

with poly_trt.TrtRunner(engine) as runner:
    # Preprocess.
    image = transform(Image.open(IMAGE_PATH).convert("RGB"))
    image = image.unsqueeze(0).numpy()
    # Input dict keys are the same as 'input_names' arg in 'torch.
onnx.export'.
    output_dict = runner.infer({"input": image})
    # Output dict keys are the same as 'output_names' arg in 'torch.
onnx.export'.
    output = output_dict["output"]
```

- 저장된 TensorRT 엔진(.engine) 파일 로드
- TrtRunner를 이용해 실행 컨텍스트 생성
- 입력 이미지를 ResNet 학습 전처리 방식대로 변환
- 추론 실행 (runner.infer)
- 결과 dict에서 "output" 키로 최종 결과 가져오기

성능 비교 및 벤치마크 결과

ImageNet 벤치마크 성능 비교



INT8 양자화를 적용할 때 Calibration 데이터는 실제 데이터 분포를 잘 반영하는 학습/검증 데이터를 사용하는 것이 유리함

Calibration 데이터로 실제 학습/검증 데이터셋 일부를 사용

임의의 생성된 데이터

추론 속도

3배 향상

INT8 모델의 평균 추론 시간 감소

모델 크기

4배 감소

INT8 변환 후 메모리 사용량 절감

정확도 손실

0.22% 최소화

학습 데이터 캘리브레이션 적용 시

벤치마크 실험 방법론

ImageNet Validation 데이터셋
전체 이미지 50,000개 대상 측정, 다양한 카테고리 포함

테스트 환경
NVIDIA Jetson AGX Orin (JetPack 5.1.1, TensorRT 8.5)

비교 모델 구성
FP32 원본, INT8 랜덤 캘리브레이션, INT8 학습 데이터 캘리브레이션

주요 발견점

캘리브레이션 데이터의 품질이 정확도에 큰 영향을 미치는 것으로 확인되었습니다.

랜덤 데이터 캘리브레이션
FP32 대비 3.25%의 정확도 손실 발생

학습 데이터 캘리브레이션
FP32 대비 0.22%의 미미한 정확도 손실로 최적의 결과

결론 및 향후 전략

요약

- **에지 AI & 로보틱스의 모델 최적화는 필수**
  제한된 리소스 환경에서 고성능 모델 구현을 위한 필수 단계
- **NVIDIA Jetson 플랫폼 선택의 이점**
  로보틱스에 최적화된 통합 하드웨어/소프트웨어 에코시스템
- **양자화와 TensorRT 최적화 효과**
  INT8 적용 시 3배 속도 향상, 4배 모델 크기 감소, 적절한 캘리브레이션으로 정확도 유지

향후 기술 전망

- **더 작은 비트 정밀도 지원 확장**
  INT4, INT2 등 더 낮은 정밀도 양자화 기법과 하드웨어 지원 증가
- **자동화된 모델 최적화 도구**
  AI 기반 자동 양자화 파라미터 튜닝 및 더 효율적인 커널 최적화 발전
- **새로운 Jetson 하드웨어 플랫폼**
  더 높은 TOPS와 낮은 전력 소비를 갖춘 차세대 에지 AI 컴퓨팅 플랫폼

권장 사항 및 실행 전략

단계적 접근 방식 채택

먼저 PTQ로 기본 최적화 → 정확도 문제 시 QAT 적용 → 필요 시 모델 아키텍처 수정

성능과 정확도 간 균형 유지

목표 애플리케이션에 맞는 적절한 타협점 찾기 (실시간성 vs 정확도)