Ch 7. 모델 가볍게 만들기

Ch 7 요약:
LMM 서비스에서 GPU 효율을 높이기 위한 추론 최적화 방안

- 배경
  - LMM 서비스의 가장 큰 비용 요소는 GPU 사용 → GPU 적게 사용해야…
  - GPU를 효율적으로 활용하는 방식
    .**모델 성능 약간 희생, 비용 크게 낮추는 방법**(Ch. 7)
    .모델 성능 유지, 연산 과정 비효율 줄이는 방법(Ch. 8)
- 학습 내용: **추론 과정**(LLM은 토큰 생성을 위해 동일 연산 반복 수행)에서 **GPU의 효율적인 사용 전략**
  텍스트 생성

1. **KV 캐싱 (KV Cache)**
   - 반복 입력에 대해 캐시를 활용하여 연산 최소화
   - 단점: 캐시 크기 증가 시 GPU 메모리 부담

2. **캐시 압축 및 분할 (GQA / MQA)**
   - KV 캐시의 일부만 GPU 메모리에 유지하거나 압축 적용
   - 예시: Multi-Query Attention, Grouped Query Attention

3. **양자화 기법**
   - 비츠앤바이츠 4비트 및 8비트 양자화, GPTQ, AWQ 등으로 비트수 줄이기
     GPT Quantization Activation-aware Weight Quantization

4. **지식 증류** (Knowledge Distillation)
   - 대형 모델의 성능을 소형 모델에 이전
     Teacher Student
   - 작은 모델 학습에 SOTA 모델 생성 결과 활용

7.1. 언어 모델 추론 이해하기

7.1.1 언어 모델이 언어를 생성하는 방법

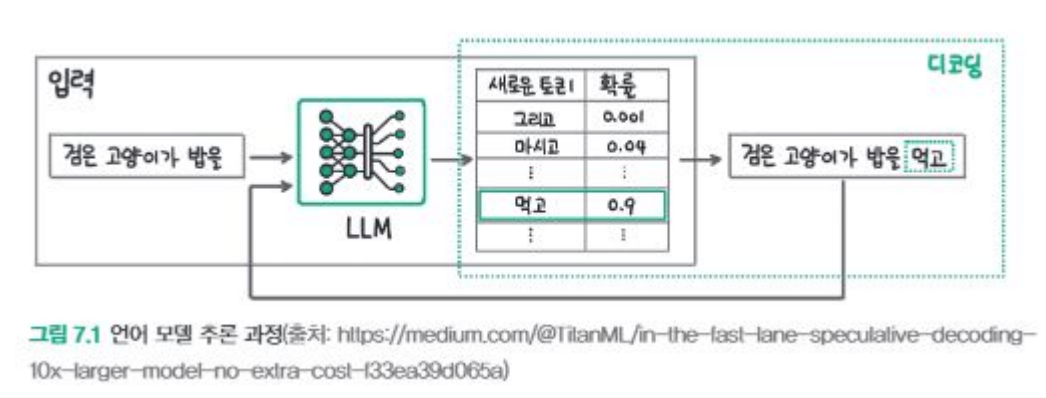


그림 7.1 언어 모델 추론 과정(출처: https://medium.com/@TitanML/in-the-fast-lane-speculative-decoding-10x-larger-model-no-extra-cost-f33ea39d065a)

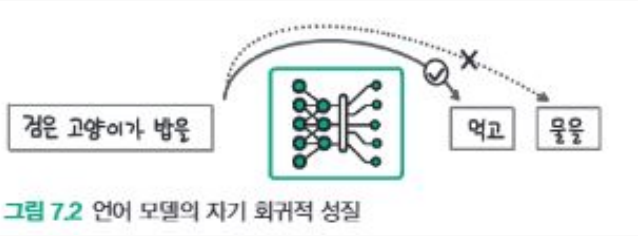


그림 7.2 언어 모델의 자기 회귀적 성질

언어 모델은 입력 텍스트 기반으로 바로 **다음 토큰만 예측**하는 **자기 회귀적(Auto-regressive)** 특성

이미 작성된 텍스트는 동시 병렬적으로 처리하므로 프롬프트가 길더라도 **다음 토큰 1개 생성하는 시간과 비슷하게 걸림**

※ 추론 과정 구분: 사전 계산 단계(prefill phase) + 토큰 생성의 디코딩 단계

7.1.1 언어 모델이 언어를 생성하는 방법

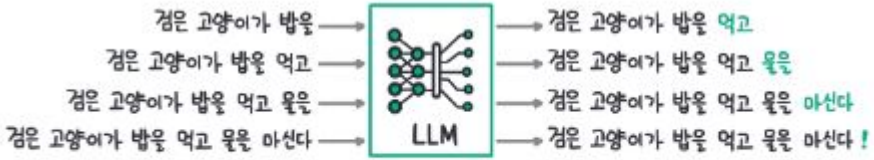


그림 7.3 언어 모델은 한 토큰씩 생성한다(출처: https://medium.com/@TitanML/in-the-fast-lane-speculative-decoding-10x-larger-model-no-extra-cost-f33ea39d065a).

트랜스포머의 근간이 되는 **셀프 어텐션**은 입력 텍스트에서 어떤 토큰이 서로 관련되는지 **Query, Key, Value를 가지고 임베딩하여 계산**하는데 **동일한 연산을 반복적으로 수행하면 비효율적**이다.
이에 따라, 이러한 연산을 반복해서 수행하지 않고 **계산 결과를 저장하고 있다가 사용하는 방법**을 알아볼 것 이다.

7.1.2 중복 연산을 줄이는 KV 캐시

KV캐시 방법: 계산했던 키와 값을 메모리에 저장해 활용

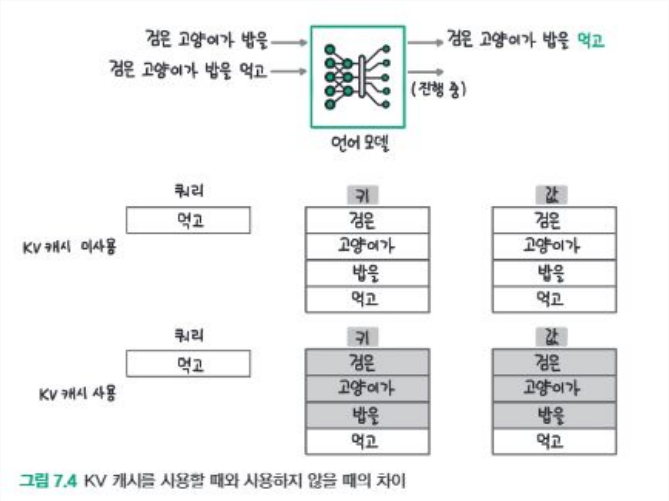


그림 7.4 KV 캐시를 사용할 때와 사용하지 않을 때의 차이

GPU 메모리에서 차지하는 영역을 살펴보면...

그림 7.5 KV 캐시를 사용하는 경우 GPU 메모리를 차지하는 데이터(출처: https://arxiv.org/abs/2309.06180)

모델 파라미터는:
13B 모델, fp16 형식 → 26GB 차지
기타: 순전파 연산 위한 메모리
나머지: KV 캐시

KV 캐시 메모리 사용량

- KV 캐시 메모리 = 2바이트×2(키와 값)×(레이어 수)×(토큰 임베딩 차원)×(최대 시퀀스 길이)×(배치 크기)

셀프 어텐션 레이어 수

※ meta-llama/Llama-2-13b-hf 모델

- 레이어 수: 40
- 토큰 임베딩 차원: 5120
- 최대 시퀀스 길이: 4096

→ 배치 1개당 3.125GB이고 위에서 KV캐시 최대가 14GB이므로

40GB - 26GB

최대 배치 크기는 4

7.1.3 GPU 구조와 최적의 배치 크기

서빙 효율성 판단 기준

1. 비용
2. 처리량: 시간당 처리한 요청 수(query/s)
3. 지연시간: 하나 토큰 생성에 걸리는 시간(token/s)

효율적인 서빙

→ 적은 비용, 더 많은 요청 처리, 생성 토큰 빠르게 전달

비용이 GPU 종류, 수에만 영향을 받는다면,

효율적인 서빙은 같은 **GPU로 처리량 ↑, 지연시간 ↓**

결론을 먼저 말씀드리면 최적의 배치 크기를
사용해야한다 ...

처리량을 높이면서 지연 시간을 낮추는 방법에 대한
아이디어를 얻기 위해 GPU 구조 확인

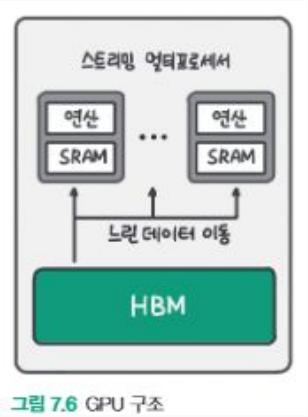


그림 7.6 GPU 구조

Streaming Multiprocessors(SM)

※ HBM: 속도, 병렬성 뛰어난 차세대 DRAM기술

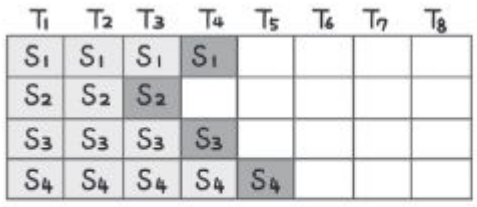
※ GPU 메모리 = HBM(High Bandwidth Memory)

A100: 108개 SM, 192KB/1 SRAM, 312TF 연산속도
40GB HBM, 데이터 Bandwidth 1,555GB/s

7.1.3 GPU 구조와 최적의 배치 크기

추론 수행시 배치 크기만큼의 토큰을 한 번에 생성

1. 이 때 **연산 시간은 계산량에 비례**



각 문장

그림 7.7 길이가 서로 다른 입력 데이터의 배치 추론
(출처: https://www.anyscale.com/blog/continuous-batching-llm-inference)

짙은 회색 부분만 실제 계산

모델 파라미터 메모리 P

연산량 = 2 x P x 배치크기 byte

fp16 그림에서는 4

2. HBM에서 SRAM으로 모델 파라미터 이동 시간

배치 크기와 상관없이 모델 파라미터 **이동 시간은 동일**

두 가지 시간이 같을 때 최적의 배치 크기

다를 경우 모델 파라미터 이동만, 연산만 하는 경우 발생 → 비효율

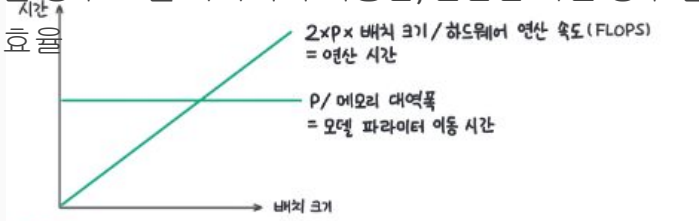


그림 7.8 모델 이동에 걸리는 시간과 연산에 걸리는 시간을 통해 본 최적의 배치 크기
(출처: https://www.youtube.com/watch?v=mYRqvB1_gRk&ab_channel=MLOps.community)

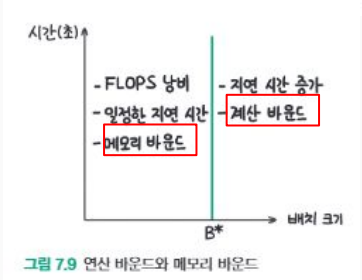


그림 7.9 연산 바운드와 메모리 바운드

GPU 효율적 사용을위해 최대 배치 크기와 최적의 배치 크기는 비슷해야...

최대 배치 크기에 맞게 키우는 방법: 모델 용량 줄이기, KV 캐시 용량

7.1.3 GPU 구조와 최적의 배치 크기

※ 최적 배치 크기 계산 예시

A100 GPU에 그림 7.8의 두 계산식을 활용해 최적의 배치 크기를 구하면 약 100이 나온다. 계산식은 다음과 같다.

- $2 \times P \times$ 배치 크기 / 하드웨어 연산 속도 = P / 메모리 대역폭
- 배치 크기 = 하드웨어 연산 속도 / (2 × 메모리 대역폭) = (312×10^{12}) / $(2 \times 1555 \times 10^{9})$ = 102.73

7.1.4 KV 캐시 메모리 줄이기

멀티 쿼리 어텐션이나 그룹 쿼리 어텐션: 키와 값의 수를 줄임 → 성능 하락 없이도 추론 속도 향상, KV 캐시 메모리 ↓

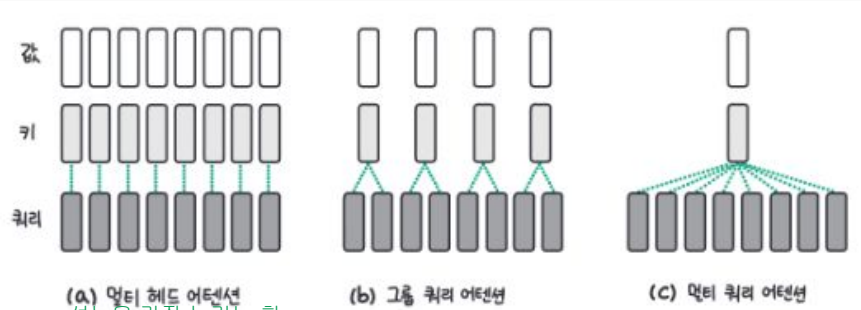


그림 7.11 멀티 헤드 어텐션, 그룹 쿼리 어텐션, 멀티 쿼리 어텐션 비교(출처: https://arxiv.org/abs/2305.13245)

성능은 가장 높기는 함

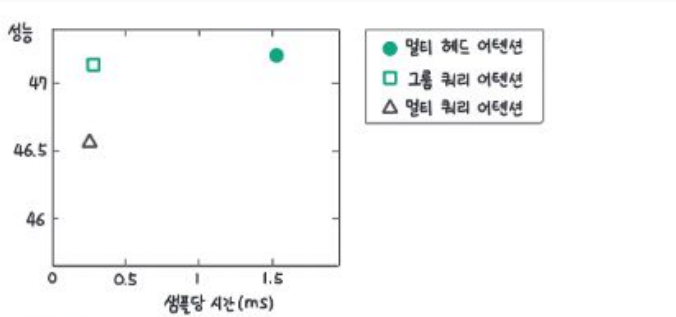


그림 7.12 세 가지 어텐션의 성능과 속도 비교(출처: https://arxiv.org/pdf/2305.13245.pdf)

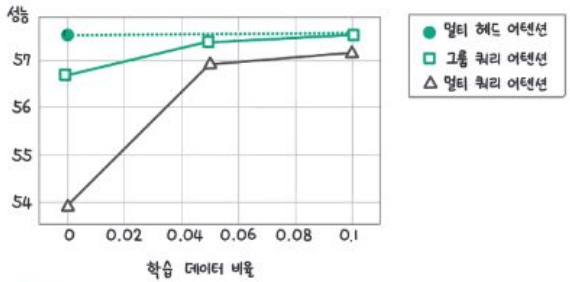


그림 7.13 키와 값의 수를 줄인 이후 추가 학습 비율에 따른 성능 회복(출처: https://arxiv.org/pdf/2305.13245.pdf)

구글 연구진
텍스트 요약 벤치마크, 1개 질문 답변 벤치마크로 성능 비교

7.2 양자화로 모델 용량 줄이기

- 16비트 파라미터는 보통 8, 4, 3 비트로 양자화
- 최근에는 4비트로 모델 파라미터 양자화, 계산은 16비트 → W4A16(Weight 4bits and Activation 16bits)
- 양자화 수행 시점에 따라,
  학습 후 양자화(Post-Training Quantization, PTQ,주로 하는 방식), 양자화 학습(Quantization-Aware Training, QAT)

7.2.1 비츠앤바이츠

- 워싱턴 대학교, Tim Dettmers, QLoRA 저자
- 양자화 라이브러리 5.1절 다양한 양자화 방식 살펴봄
  양자화 목표: 더 적은 메모리를 사용하면서도 최대한 원본 모델의 정보를 유지하는 것
  - 방식 1: 8비트 연산 수행시 성능 저하 거의 없이 성능을 유지하는 8비트 행렬 연산
  - 방식 2: 4비트 정규 분포 양자화 방식 4비트 양자화 방식(5.5절)
- 양자화 방식 양자화 할 때 낭비되는 수를 줄이기 위해…
  - 영점 양자화: 데이터의 최댓값과 최솟값을 변환하려는 데이터 형식의 범위로 변환
  - 절대 최댓값(absmax) 양자화: 절대 최댓값 기준으로 대칭적으로 새로운 데이터 타입으로 변환

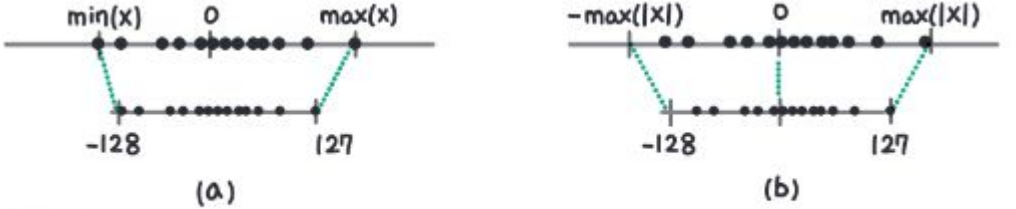


그림 7.14 (a) 영점 양자화와 (b) 절대 최댓값 양자화 방식
(출처: https://intellabs.github.io/distiller/algo_quantization.html)

하지만 위 방식으로 8비트로 양자화 할 경우, 기존 모델에 비해 성능이 떨어짐

7.2.1 비츠앤바이츠

- 해결책:
  - 입력 X 값 중 크기가 큰 이상치가 포함된 열은 별도로 분리해서 16비트 그대로 계산
    (입력에서 값이 큰 경우 중요한 정보를 닫고 있다고 판단해 정보가 소실되지 않도록 양자화 하지않고 그대로 연산)
  - 정상 범위 열을 양자화 할 때 벡터 단위로 최댓값을 찾고 그 값 기준으로 양자화 수행

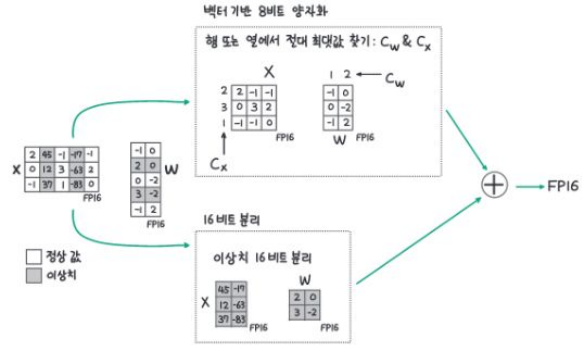


그림 7.15 비츠앤바이츠의 8비트 양자화 방식(출처: https://arxiv.org/pdf/2208.07339.pdf 그림 수정)

7.2.1 비츠앤바이츠

- 비츠앤바이츠 8비트 양자화, 4비트 양자화 모델

```python
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

# ================================
# 8비트 양자화 모델 불러오기
# ================================

# BitsAndBytesConfig: 모델을 로드할 때 양자화 관련 설정을 담는 객체
# load_in_8bit=True: 모델의 가중치를 8비트 정밀도로 불러오도록 지정 (메모리 효율 ↑, 속도 ↑)
bnb_config_8bit = BitsAndBytesConfig(load_in_8bit=True)

# AutoModelForCausalLM: causal language modeling을 위한 사전학습된 모델을 자동으로 불러오는 클래스
# from_pretrained(): Hugging Face Hub에서 지정한 모델(여기선 "facebook/opt-350m")을 다운로드 및 로드
# quantization_config: 위에서 정의한 bnb_config_8bit 설정을 적용하여 8비트 양자화로 모델 로딩
model_8bit = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m",                  # OPT 350M 모델 (Causal LM 아키텍처)
    quantization_config=bnb_config_8bit   # 8비트 양자화 설정 적용
)


# ================================
# 4비트 양자화 모델 불러오기
# ================================

# BitsAndBytesConfig: 4비트 정밀도로 양자화 설정
# load_in_4bit=True: 4비트 양자화 적용
# bnb_4bit_quant_type="nf4": 양자화 방식 선택 (nf4 = Normal Float 4bit, 더 정밀한 4bit 방식)
bnb_config_4bit = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4"  # 다른 옵션: "fp4" 등
)

# low_cpu_mem_usage=True: 모델 로딩 중 CPU 메모리 사용 최소화 (대규모 모델 로딩 시 유용)
# quantization_config: 위에서 설정한 4비트 양자화 설정 적용
model_4bit = AutoModelForCausalLM.from_pretrained(
    "facebook/opt-350m",                  # 같은 OPT 350M 모델
    low_cpu_mem_usage=True,               # CPU 메모리 사용 최적화
    quantization_config=bnb_config_4bit   # 4비트 양자화 설정 적용
)
```

7.2.2 GPTQ

- 2022, Elias Frantar, GPTQ(GPT Quantization)
  - 양자화 이전 모델에 입력 X를 넣었을 때와 양자화 이후 모델에 입력 X를 넣었을 때 오차가 가장 작아지도록 모델의 양자화 수행
- 양자화 전후 결과 차이가 적도록 양자화 수행 방법
  - 양자화를 위한 작은 데이터셋을 활용하여 모델 연산을 수행하면서 양자화 이전과 유사한 결과가 나오도록 모델 업데이트

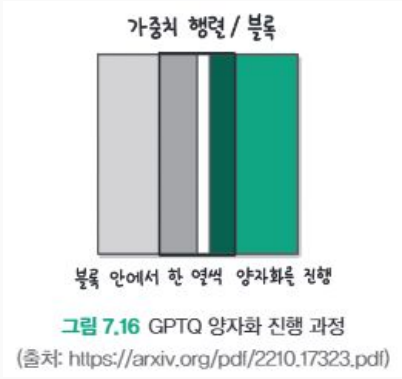


그림 7.16 GPTQ 양자화 진행 과정
(출처: https://arxiv.org/pdf/2210.17323.pdf)

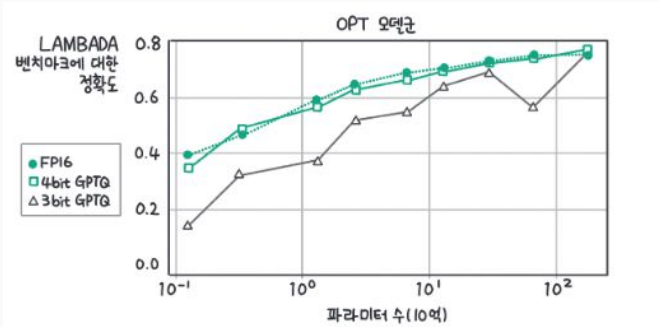


그림 7.17 GPTQ 양자화와 기존 모델의 성능 비교(출처: https://arxiv.org/pdf/2210.17323.pdf)

LAMBADA 벤치마크: 제시문을 바탕으로 문장의 빈칸을 채우는 문제를 푸는 능력을 평가하기 위한 데이터셋

7.2.2 GPTQ

- 2022, El

▼ 예제 7.2 GPTQ 양자화 수행 코드(출처: https://huggingface.co/blog/gptq-integration)

```
from transformers import AutoModelForCausalLM, AutoTokenizer, GPTQConfig

# 사용할 사전학습 모델 ID 지정 (Hugging Face Hub에서 불러올 모델 이름)
model_id = "facebook/opt-125m"

# AutoTokenizer: 해당 모델에 맞는 토크나이저 자동 로드
tokenizer = AutoTokenizer.from_pretrained(model_id)

# ============================
# GPTQConfig로 4비트 양자화 설정
# ============================

# GPTQConfig:
#   - bits=4: 4비트 양자화를 적용하여 메모리 사용량을 크게 줄임
#   - dataset="c4": 양자화 시 참고하는 calibration 데이터셋 지정 (C4: Common Crawl 기반 데이터셋)
#   - tokenizer: GPTQ에서 사용할 tokenizer 지정 (정확한 weight quantization을 위해 필요)
quantization_config = GPTQConfig(
    bits=4,
    dataset="c4",
    tokenizer=tokenizer
)

# ============================
# 4비트 양자화 모델 불러오기
# ============================

# AutoModelForCausalLM:
#   - causal language modeling(텍스트 생성)에 적합한 사전학습 모델을 자동 로드
# from_pretrained:
#   - model_id: 모델 이름 (facebook/opt-125m)
#   - device_map="auto": 사용 가능한 GPU/CPU에 자동 분산 로딩
#   - quantization_config: 위에서 설정한 GPTQ 기반 양자화 설정
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    device_map="auto",                    # GPU/CPU 자동 분산
    quantization_config=quantization_config  # 4비트 GPTQ 양자화 적용
)
```

▼ 예제 7.3 GPTQ 양자화된 모델 불러오기

```
from transformers import AutoModelForCausalLM
model = AutoModelForCausalLM.from_pretrained("TheBloke/zephyr-7B-beta-GPTQ",
                                             device_map="auto",
                                             trust_remote_code=False,
                                             revision="main")
```

7.2.3 AWQ

- 더 적은 메모리를 사용하면서도 모델이 가진 정보를 최대한 손실없이 변환해야함
  모델 파라미터는 정보를 고르게 가질지, 모델 파라미터 중 특별히 중요한 파라미터가 있을지
- 2023, MIT, AWQ(Activation-aware Weight Quantization)
  - 모든 파라미터가 동등하게 중요하지 않으며 특별히 중요한 파라미터 정보를 유지하면 양자화를 수행하면서도 성능 저하를 막을 수 있음
  - 어떤 파라미터가 중요한지 판단하는 방법
    - 모델 파라미터 값이 큰 경우
    - 활성화 값이 큰 채널의 파라미터가 중요하다고 가정

MIT연구진은 모델 파라미터 자체와 활성화 값을 기준으로 상위 1% 해당하는 모델 파라미터를 기존 모델의 데이터 타입인 FP16으로 유지하고 나머지는 양자화

활성화 값 기준으로 중요한 1% 파라미터 정보만 지키면 모델 성능이 유지된다는 사실 발견

But, 모델 파라미터 크기 기준으로 모델 파라미터 유지한 경우 성능 저하

7.2.3 AWQ

- 모델 파라미터에 서로 다른 데이터 타입이 섞여 있는 경우 한번에 일괄적으로 연산이 어렵기 때문에 연산이 느려지고 하드웨어 효율성이 떨어지는 문제 발생

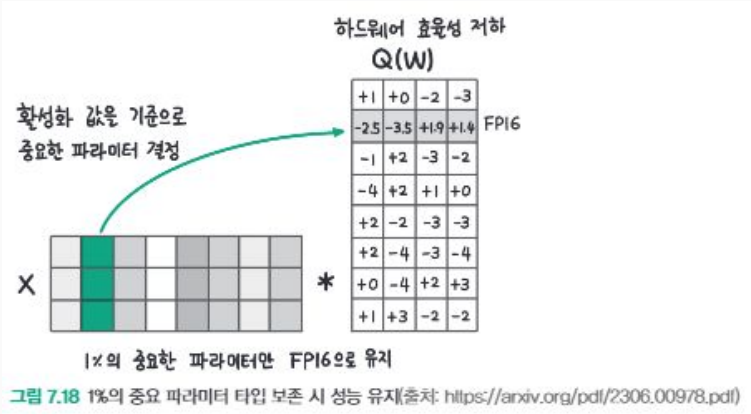


그림 7.18 1%의 중요 파라미터 타입 보존 시 성능 유지(출처: https://arxiv.org/pdf/2306.00978.pdf)

7.2.3 AWQ

- 중요한 정보 소실

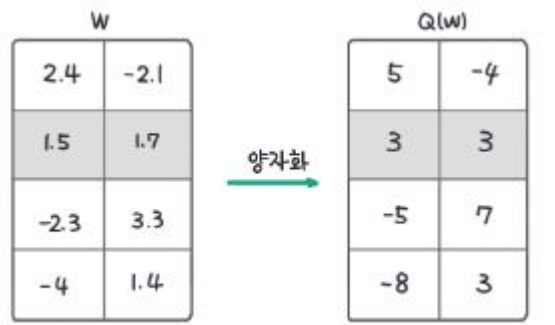


그림 7.19 양자화 과정에서 중요한 정보의 손실 발생

7.2.3 AWQ

- 해결책: 중요한 파라미터에만 1보다 큰 값을 곱하는 방식(스케일러) 적용

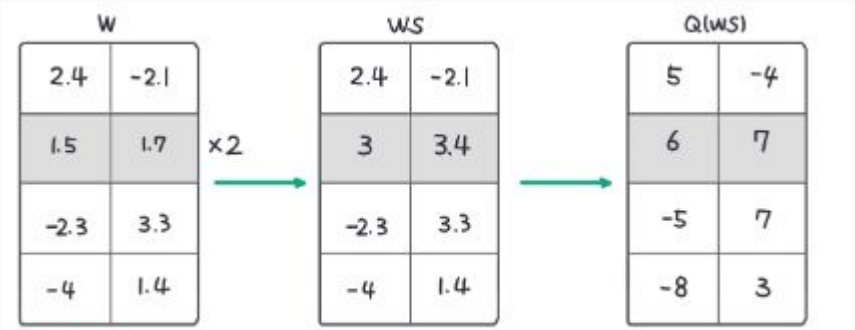


그림 7.20 스케일러를 곱했을 때 중요한 정보의 소실을 막음

- 스케일러가 2를 넘어가면 성능 하락
  퍼플렉시티(낮을수록 좋은 지표)

표 7.1 스케일러 s에 따른 성능 비교(출처: https://arxiv.org/pdf/2306.00978.pdf)

OPT-6.7B	$s = 1$	$s = 1.25$	$s = 1.5$	$s = 2$	$s = 4$
Wiki-2 퍼플렉시티	23.54	12.87	12.48	11.92	12.36

7.2.3 AWQ

- s가 더 커질 때 성능이 저하되는 이유

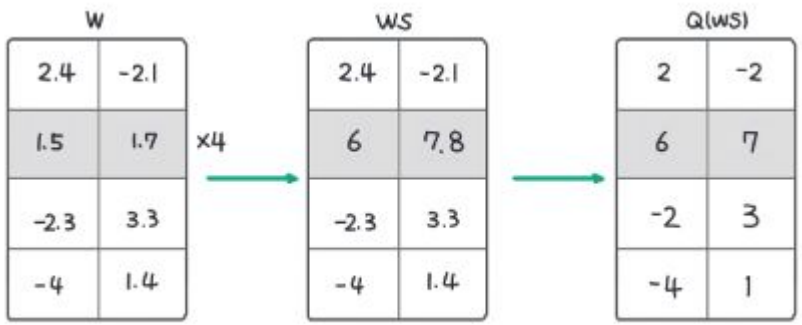


그림 7.21 스케일러가 4인 경우 나머지 파라미터에서 정보 소실 발생

▼ 예제 7.4 AWQ 양자화 모델 불러오기

```
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

model_name_or_path = "TheBloke/zephyr-7B-beta-AWQ"
tokenizer = AutoTokenizer.from_pretrained(model_name_or_path, trust_remote_code=False)
model = AutoAWQForCausalLM.from_quantized(model_name_or_path, fuse_layers=True,
trust_remote_code=False, safetensors=True)
```

7.3 지식 증류 활용하기

- Knowledge Distillation
  - 성능이 높은 Teacher model의 생성 결과를 활용해 더 작고 성능이 낮은 Student model을 만드는 방법
  - 선생 모델에 쌓은 지식을 더 작은 모델로 압축해 전달한는 의미에서 '증류'

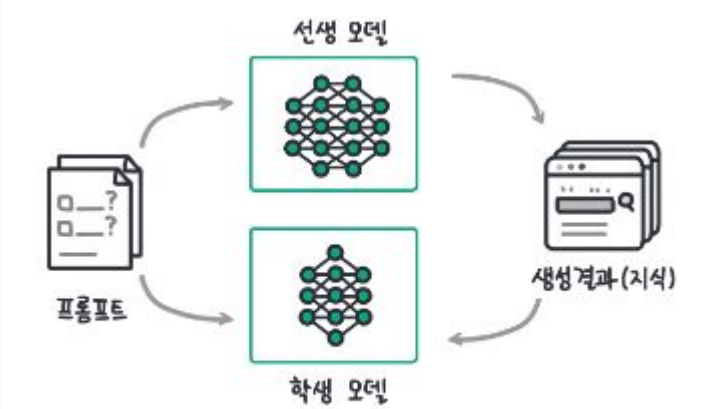


그림 7.22 선생 모델의 생성 결과를 통해 학생 모델을 학습시키는 지식 증류

7.3 지식 증류 활용하기

- 기존에는 학습 데이터셋에 대한 선생 모델의 추론 결과를 학생 모델의 학습에 활용하는 정도
- 선생 모델을 활용해 완전히 새로운 학습 데이터셋을 대규모로 구축하거나 데이터셋 구축에 사람의 판단이 필요한 부분을 선생 모델이 수행하는 등 폭넓게 활용

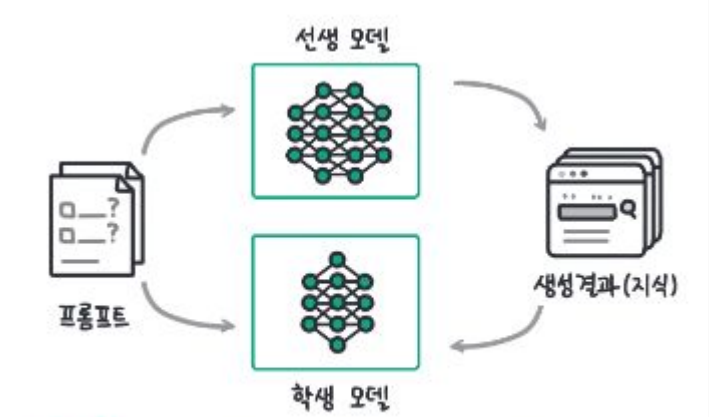


그림 7.22 선생 모델의 생성 결과를 통해 학생 모델을 학습시키는 지식 증류