

</

모델 가볍게 만들기

/>



LLM을 활용한 실전 AI 애플리케이션
개발
발표: 이재원

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</INTRO

- LLM을 배포하는 경우 가장 많은 비용은 **GPU**에서 발생합니다.
- GPU를 효율적으로 활용하는 방식은
 1. 모델의 성능을 약간 희생하더라도 비용을 크게 낮추는 방법
 2. 모델의 성능을 그대로 유지하면서 연산 과정의 비효율을 줄이는 방법

이번 **7장**에서는 **1번의 방법**을 알아봅니다.

</INTRO

- LLM의 추론 과정이 어떻게 진행되는지 살펴봅니다.
- 동일한 연산을 최대한 줄이기 위해 계산 결과를 저장하는 Key-Value **KV 캐시**를 사용합니다.
- GPU 구조와 GPU에 저장되는 데이터를 살펴보며 다양한 추론 효율화 방안을 해석하는 데 도움이 되는
‘최적의 배치 크기’ 개념에 대해 알아봅니다.
- KV 캐시가 사용하는 메모리를 줄이기 위한 ‘멀티 쿼리 어텐션’, ‘그룹 쿼리 어텐션’에 대해 알아봅니다.

</INTRO

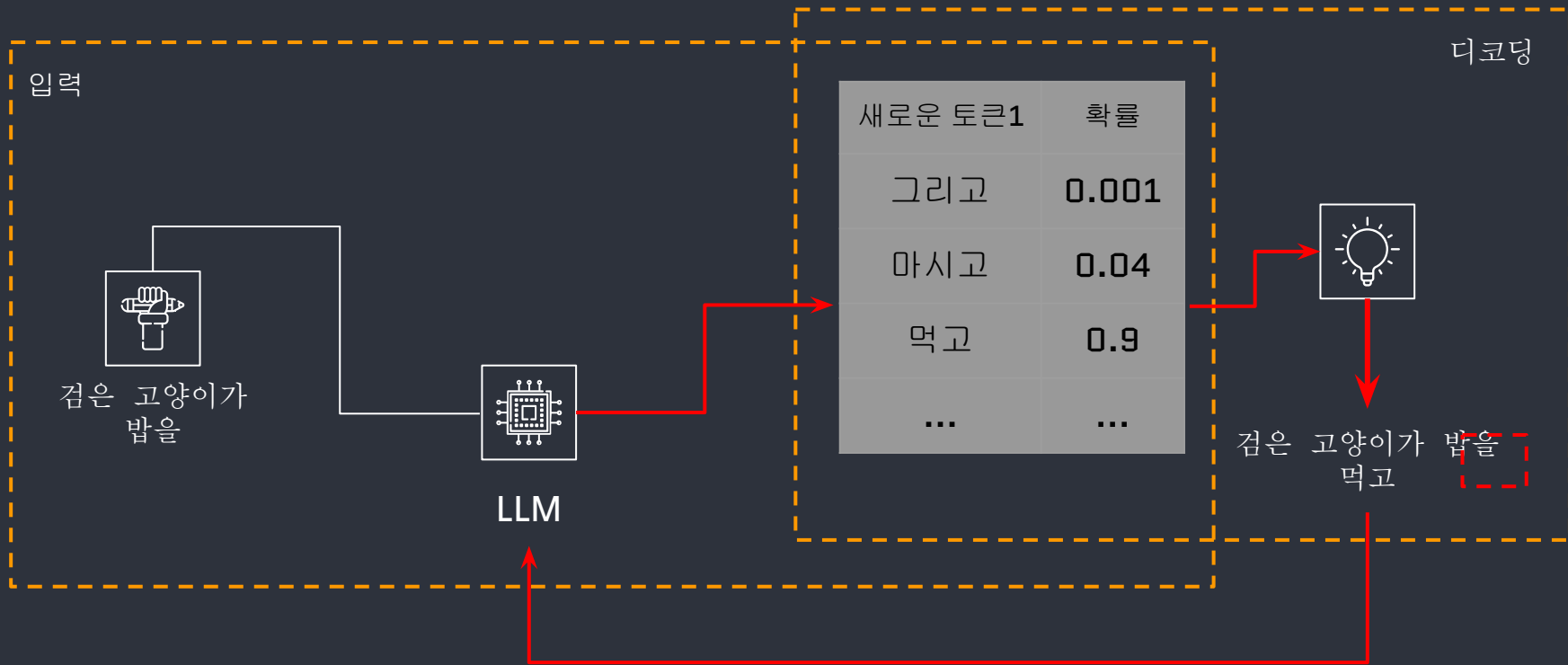
- 모델 파라미터를 저장할 때 더 적은 메모리를 사용하도록 데이터타입을 변환하는 양자화에 대해 알아봅니다.

BitsAndBytes, **GPTQ**(GPT Quantization),
AWQ(Activation-aware Weight Quantization) 방식을 살펴봅니다.

- 더 크고 성능이 좋은 선생 모델의 생성 결과를 활용해,
더 작고 효율적인 학생 모델을 학습하는 **지식 증류** 방법에 대해 알아봅니다.

작은 모델의 학습에 SOTA 모델 (State Of The Arts; 현재 최고 수준의 결과를 가진 모델)의 생성 결과를 활용하는 방식이 활용되고 있고 뛰어난 결과를 보여주고 있습니다.

</언어 모델이 언어를 생성하는 방법



</언어 모델이 텍스트 생성을 마치는 이유

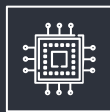
- 다음 토큰으로 생성 종료를 의미하는 특수 토큰인 **EOS**(End of Sentence) 토큰을 생성하는 경우
- 사용자가 **최대 길이로 설정한 길이에 도달한 경우**
*이에 해당하지 않으면 순환 화살표를 돌면서 계속해서 새로운 토큰을 추가한 텍스트를 다시 모델의 입력으로 넣는 **과정을 반복 (loop)**

</Auto Regressive [자기 회기적]

- ‘검은 고양이가 밥을’을 입력했을 때 다음 토큰과 그 다음 토큰을 함께 예측할 수는 없음. 입력 텍스트 기반으로 바로 다음 토큰만 예측하는 **자기 회기적(auto-regressive)** 특성.
- ‘검은 고양이가 밥을’ 같은 프롬프트는 이미 작성된 텍스트이기 때문에 한 번에 하나씩 토큰을 처리할 필요가 없이 동시에 병렬적으로 처리할 수 있고, 길다고 하더라도 다음 토큰 1개를 생성하는 시간과 비슷
- 추론 과정을, 프롬프트를 처리하는 단계인 **사전 계산 단계(prefill phase)** 한 토큰씩 생성하는 **디코딩 단계(decoding pahse)**로 구분.

</Auto Regressive [자기 회기적]

검은 고양이가 밥을
검은 고양이가 밥을 먹고
검은 고양이가 밥을 먹고 물을
검은 고양이가 밥을 먹고 물을
마신다



LLM

검은 고양이가 밥을 먹고
검은 고양이가 밥을 먹고 물을
검은 고양이가 밥을 먹고 물을
마신다
검은 고양이가 밥을 먹고 물을
마신다!

</intro KV[key-value] Cache

- 2장에서 트랜스포머 모델의 기반이 되는 셀프 어텐션 연산에 대해 학습함.
- 셀프 어텐션 연산은 입력 텍스트에서 어떤 토큰이 서로 관련되어 있는지 계산해서, 그 결과에 따라 토큰 임베딩을 새롭게 조정.
관련도를 계산하기 위해 토큰 임베딩을 쿼리, 키, 값 벡터로 변환하는 선형 변환을 수행.
- 생성 속도를 높이기 위해 **계산 결과를 저장**하고 있다가, **다시 사용**하는 방법을 사용.
- **KV 캐시**는 셀프 어텐션 연산 과정에서 **동일한 입력 토큰에 대해 중복 계산**이 발생하는 비효율을 줄이기 위해 **먼저 계산했던 키와 값 결과를 메모리에 저장해 활용**하는 방법.

KV 캐시를 사용하지 않으면 ‘검은’, ‘고양이가’, ‘밥을’을 키와 값 벡터로 변환하는 동일한 연산을 반복.

</KV 캐시에 필요한 메모리 구하기

- KV 캐시 메모리 =
2바이트 X 2(키와 값) X 레이어 수 X 토큰 임베딩 차원 X 최대 시퀀스 길이 X 배치 크기
- 2 바이트: fp16 형식을 사용했기 때문에 2
- KV 캐시는 키 캐시와 값 2개를 저장했으므로 2
- 셀프 어텐션 연산 결과는 어텐션 레이어 수만큼 생기기 때문에 (레이어 수)
- 토큰 임베딩을 표현하는 차원의 수
- 시퀀스 길이만큼 메모리를 미리 확보하기 위해 최대 시퀀스 길이와 배치 크기

</KV 캐시에 필요한 메모리 구하기

- 파라미터 크기가 130억 개 라마-2 13B 모델에서 각각 값을 확인

레이어 수: 40

토큰 임베딩 차원: 5120

최대 시퀀스 길이: 4096

- 따라서, 엔비디아 A100 모델의 GPU 메모리 40GB 중, 모델을 저장하는데, 26GB를 사용했으므로,
KV 캐시는 최대 14GB를 사용할 수 있는데, 배치를 추가할 때마다 2.5GB를 소모하므로 배치 크기는 5 정도.
- 비싼 GPU를 더 효율적으로 활용하기 위해서는 더 많은 입력을 처리해야 함.
하지만 방금에서 배치 크기가 5 정도로 확인. 배치를 어느정도까지 키울 수 있어야,
GPU를 충분히 잘 활용했다고 할 수 있을까?

</GPU 구조와 최적의 배치 크기

- 서버가 효율적인지 판단하는 기준
 1. 비용
 2. 처리량 (query/s) (높을 수록 좋음)
 3. 지연시간 (token/s) (낮을 수록 좋음)

적은 비용으로 더 많은 요청을 처리하면서 생성한 다음 토큰을 빠르게 전달할 수 있다면 효율적인 서버

</GPU 구조와 최적의 배치 크기

- GPU는 여러 스트리밍 멀티프로세서(Streaming Multiprocessors) SM 으로 구성
- 각각의 SM에는 연산을 수행하는 부분과 계산할 값을 저장하는 **SRAM**(Static RAM)이 존재
SRAM은 L1 캐시 또는 공유 메모리

*SRAM은 큰 메모리를 갖기 어렵기 때문에, 큰 대역폭 메모리(High Bandwidth Memory)에 큰 데이터를 저장한다.

</GPU 구조와 최적의 배치 크기

전체 카테고리

홈 >

컴퓨터/노트북/조립PC >



주요부품 >

그래픽카드(VGA) >

선택하세요 >

NVIDIA A100 HBM2e 80GB PCIe vs검색 ?

A100 / 스트림 프로세서: 6912개 / PCIe4.0 / HBM2 / 지원정보: 멀티VGA 지원 / 사용전력: 300W / FP16 Tensor Core: 312 TFLOPs / FP32: 19.5 TFLOPs



최저가

28,319,990원

최저가 구매하기

딜러 가격

☐ 배송비포함 ?

Gmarket	최저가 28,319,990원	3,500원	최대 24개월
엑스디... u pay	28,320,000원	무료배송	최대 12개월
coupan	29,750,000원	무료배송	
컴나무	현금 31,184,000원	3,000원	
컴오아시스	31,184,990원	4,000원	
11D	31,280,900원	3,500원	최대 22개월
롯데ON	32,040,360원	무료배송	
샵다나와	32,817,680원	3,000원	

등록일: 2023.07. | 제조사:NVIDIA | 이미지출처: G마켓

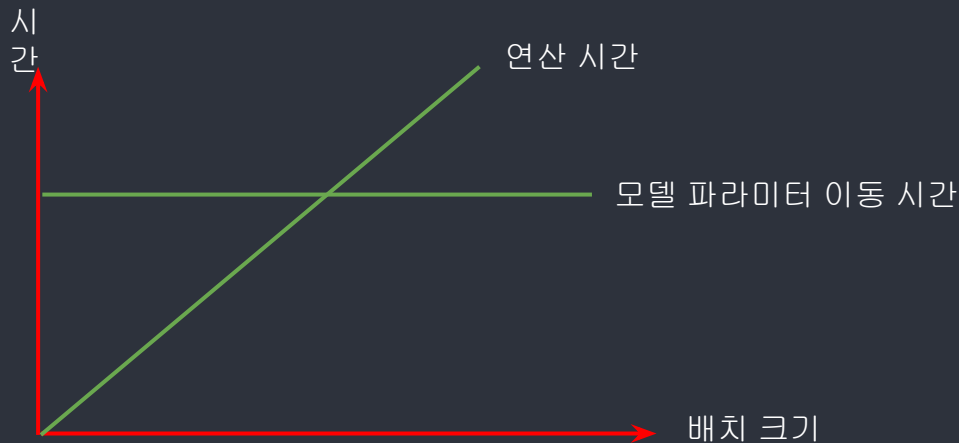
</GPU 구조와 최적의 배치 크기

T1	T2	T3	T4	T5	T6	T7	T8
S1	S1	S1	S1				
S2	S2	S2					
S3	S3	S3	S3				
S4	S4	S4	S4	S4			

- 추론을 수행할 때 배치 크기만큼 토큰을 한 번에 생성한다.
- 배치의 각 문장(S) 길이가 서로 다른데, 추론을 수행하면 각각 프롬프트 토큰(열은) 뒤로 새롭게 생성한 토큰(길은)이 더해진다.
- KV 캐시를 이용하면 열은 부분은 KV 캐시에서 가져오고, 길은 부분만 실제 계산한다.
- 모델 파라미터가 차지하는 메모리가 P라고 할 때, 계산량은 대략 $2 \times P \times (\text{배치 크기})$ 바이트.
(2는 fp16 데이터 형식을 사용하기에 2를 곱해줌)

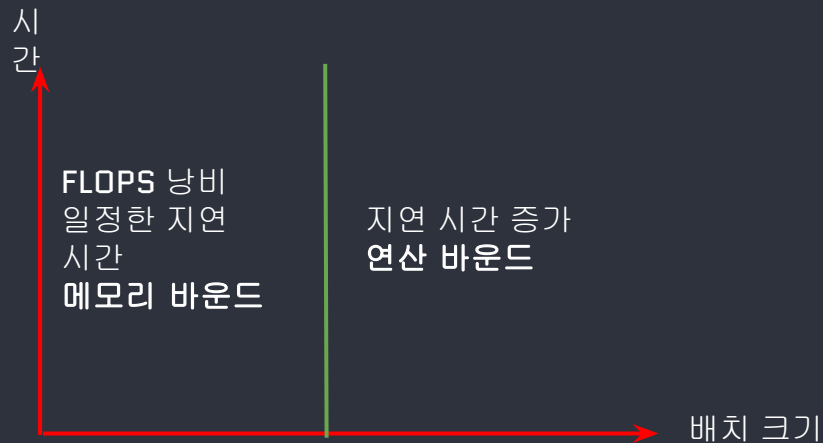
</GPU 구조와 최적의 배치 크기

- 하지만 모델의 추론 과정에서 실제 행렬 곱셈 연산을 수행하는 데만 시간이 걸리는 것이 아님.
- GPU 구조를 보면, SRAM 크기는 작기 때문에, 모델을 HBM에 저장한다고 했음.
- 연산을 수행하기 위해서는 HBM에 있는 모델 파라미터를 SRAM으로 이동해야 함.
즉, P (모델 파라미터가 차지하는 메모리) 만큼 메모리를 이동하는 데 시간이 걸림.



</GPU 구조와 최적의 배치 크기

- 배치 크기가 커지면 연산에 필요한 시간은 증가하지만, 모델 파라미터의 이동에 걸리는 시간은 변함이 없음.
- 모델 이동 과정과 연산 수행 과정은 함께 진행되기 때문에, 두 가지 시간이 같을 때가 최적의 배치 크기가 됨.
- 만약 서로 다른 시간이 걸린다면, 모델 파라미터만 이동시키거나, 연산만 하면서 다른 한쪽이 멈추기 때문에 비효율이 발생.



</GPU 구조와 최적의 배치 크기

- A100 GPU에 두 계산식을 활용해 최적의 배치를 구하면 100이 나옴.
 $2 \times P \times \text{배치 크기} / \text{하드웨어 연산 속도} = P / \text{메모리 대역폭}$
- 배치 크기 = 하드웨어 연산 속도 / (2 X 메모리 대역폭) = $(312 \times 10^{12}) / (2 \times 155 \times 10^9) = 102.73$
- 하지만 앞서 라마-2 13B 모델을 사용할 경우 GPU 메모리가 40GB인 GPU에서 최대 배치 크기가 5 정도 였음 => GPU를 더 효율적으로 사용하려면 최대 배치 크기가 최적의 배치 크기에 가까워질 수 있는 방법을 찾아야 함.
- GPU 메모리에 올라가는 중요한 데이터가
 1. 모델 파라미터를 줄이는
 1. 양자화
 2. 지식 증류
 2. KV 캐시를 줄이는
 1. 멀티 쿼리 어텐션
 2. 그룹 쿼리 어텐션

</KV 캐시 메모리 줄이기

- tl;dr
그룹 쿼리 어텐션은 사용하는 키와 값 벡터 수를 줄임으로써 성능 허락이 거의 없이도 모델의 추론 속도를 향상하고 KV 캐시의 메모리 사용량을 줄일 수 있다.
- 등장하는 세가지 어텐션
 1. 멀티 헤드 어텐션
 2. 멀티 쿼리 어텐션
 3. 그룹 쿼리 어텐션

</Multi Head Attention

- 트랜스포머 모델이 셀프 어텐션 연산을 수행할 때, 한 번의 어텐션 연산만 수행하는 것이 아니라, 여러 헤드에서 어텐션 연산을 수행하는 멀티 헤드 어텐션을 사용했다.
- 한번에 여러 헤드에 대한 연산을 수행하므로, 쿼리와 키 사이에 다양한 측면의 관련성을 반영, 성능을 높일 수 있다.

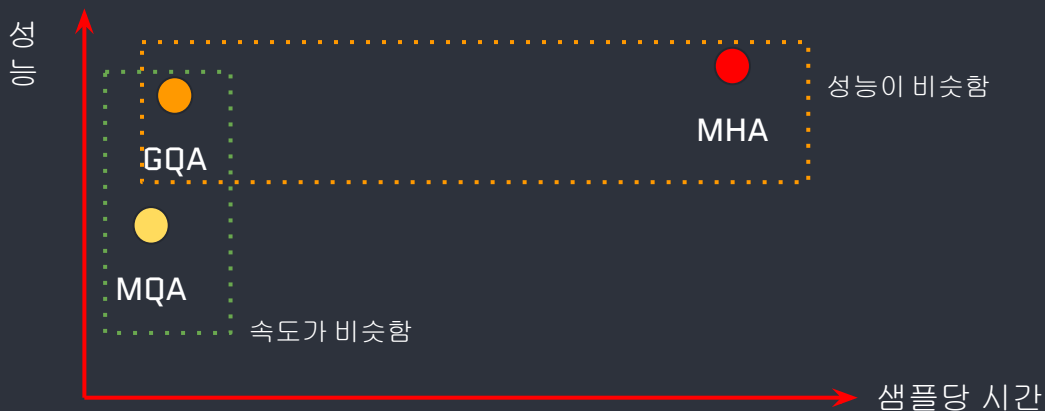
*하지만 많은 수의 키와 값 벡터를 저장하기 때문에 KV 캐시에 더 많은 메모리를 사용해야하고 더 많은 데이터를 불러와 계산하기 때문에 속도가 느려진다.

</Multi Query Attention

- 멀티 헤드 어텐션의 단점을 줄임.
 - 여러 헤드의 쿼리 벡터가 하나의 키와 값 벡터를 사용함.
 - 멀티 헤드 어텐션이 8개의 키와 값 벡터를 저장했다면,
멀티 쿼리 어텐션은 1개의 키와 값 벡터만 저장하기 때문에 KV 캐시를 저장하는데 적은 메모리를 사용.
- *하지만 키와 값을 1개만 사용하면서 멀티 헤드 어텐션에 비해 성능이 떨어지는 문제가 발생.

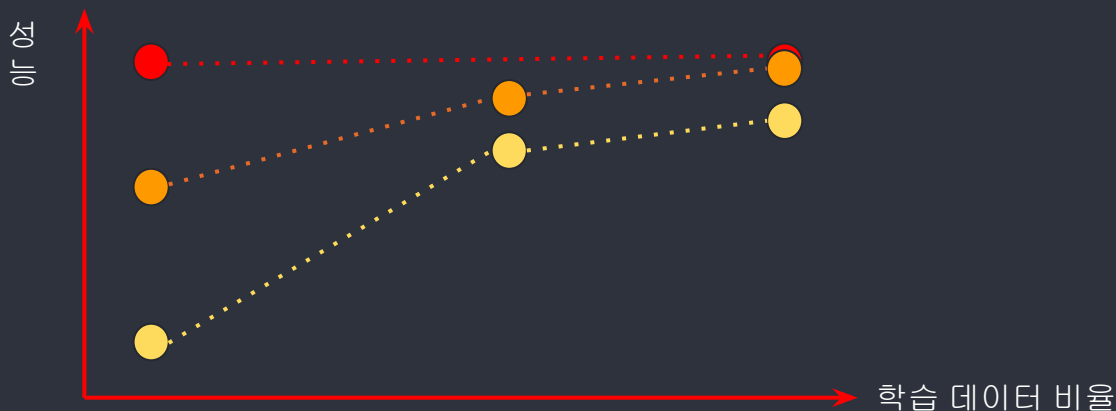
</Group Query Attention

- 멀티 쿼리 어텐션의 단점을 줄임.
- 멀티 헤드 어텐션보다는 키와 값을 줄이지만, 멀티 쿼리 어텐션보다는 많은 키와 값을 사용.
(멀티 헤드 어텐션과 멀티 쿼리 어텐션을 절충한 방법)
- 키와 값의 수를 줄이면, 추론 속도 향상, KV 캐시 메모리 감소가 있음.



</Group Query Attention

- 멀티 쿼리 어텐션의 경우 멀티 헤드 어텐션과 비교했을 때 성능 저하가 뚜렷하기 때문에 키와 값을 줄인 이후에 기존 학습 데이터로 추가 학습(uptraining)을 수행함.
- 추가 학습에 사용하는 학습 데이터의 비율(알파)에 따라 성능이 어떻게 달라지는지 비교.



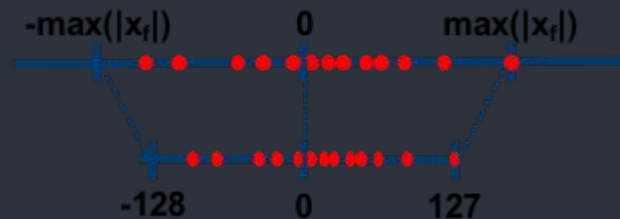
</양자화로 모델 용량 줄이기

- 양자화: 부동소수점 데이터를 더 적은 메모리를 사용하는 정수 형식으로 변환해 GPU를 효율적으로 사용
이전에는 32비트 부동소수점 (FP32) 데이터 형식으로 모델 파라미터를 저장함.
*하지만 모델이 점점 커지면서 요즘은 16비트 형식을 기본으로 사용하는 경우가 많음(FP16, BF16)
*하지만 16비트로 저장한다고 해도 파라미터가 70억개 모델을 GPU에 올리는 데에 14GB의 메모리가 필요
- 16비트 파라미터는 보통 8,4,3비트로 양자화하는데,
최근에는 4비트로 모델 파라미터를 양자화하고 계산은 16비트로 하는 W4A16(Weight 4bit and Activation 16bits)을 주로 사용함.
- 양자화를 수행하는 시점에 따라서,
1. 학습 후 양자화 (Post-Training Quantization, PTQ)
2. 양자화 학습 (Quantization-Aware Training, QAT)
- 허깅페이스에서 활발히 사용되는 양자화 3가지
1. 비츠앤바이트(Bits-and-bytes)
2. GPTQ(GPT Quantization)
3. AWQ(Activation-aware Weight Quantization)

</Bits and Bytes

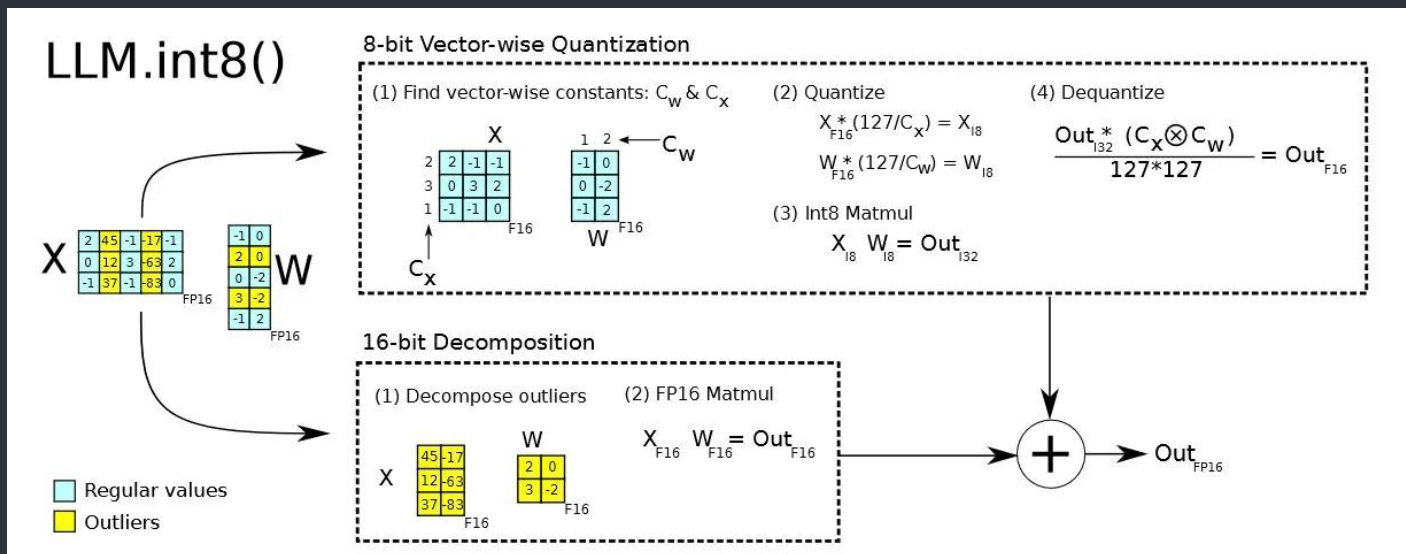
- 크게 두가지 양자화 방식을 제공함
 1. 8비트로 연산을 수행하면서도 성능 저하가 거의 없이 성능을 유지하는 8비트 행렬 연산
 2. 4비트 정규 분포 양자화
 - *5장 5절에서 살펴본 4비트 양자화 방식이 4비트 정규 분포 양자화 방식

</Bits and Bytes



- 영점 양자화, 절대 최댓값 양자화 방식을 주로 사용했지만, 위의 양자화를 그대로 사용할 경우, 성능이 떨어지기 때문에 새로운 방식을 도입함.
*기존에는 전체 모델을 8비트로 양자화 했지만, 비즈앤바이즈는 다른 방식으로 양자화를 진행함.

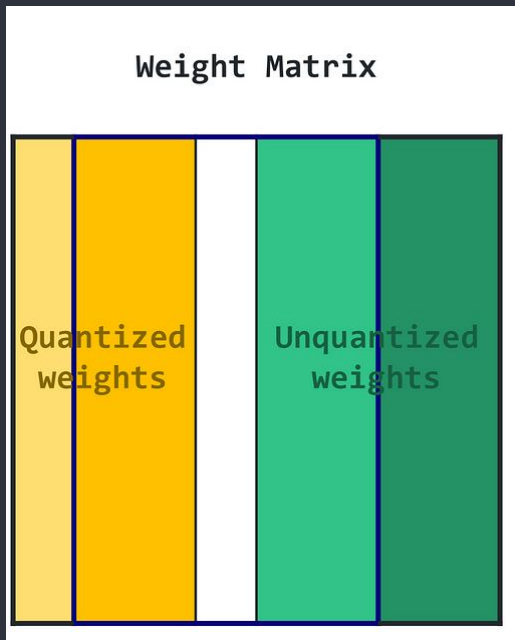
</Bits and Bytes



</GPTQ [GPT Quantization]

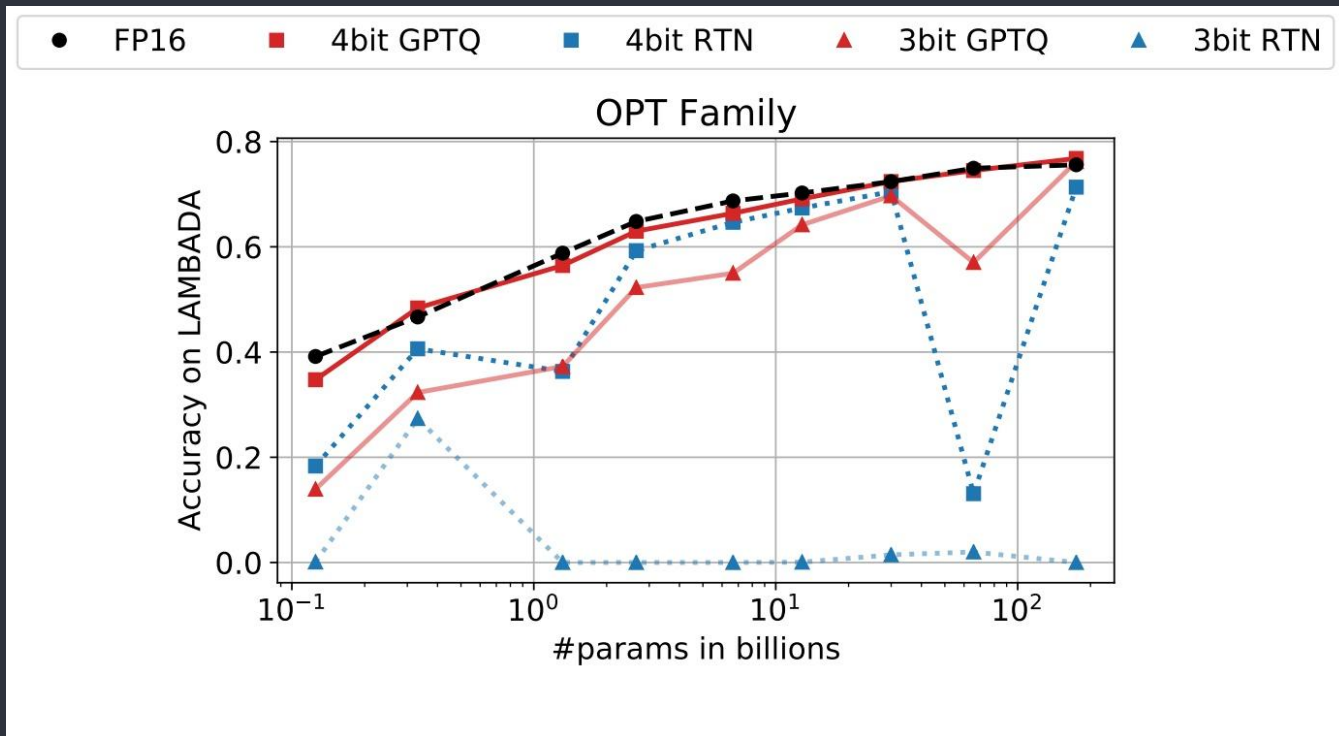
- 양자화는 기존 모델의 파라미터를 더 적은 용량을 사용하는 데이터 타입으로 변환함.
 - *따라서 기본적으로 갖고 있던 정보를 잃어버릴 수 밖에 없다.
 - *따라서 성능 하락이 발생하는데, 각각의 양자화 방식은 각자의 방식으로 이런 성능 하락을 최소화함.
- GPTQ는 양자화 이전의 모델에 입력 X를 넣었을 때와 양자화 이후의 모델에 입력 X를 넣었을 때 오차가 가장 작아지도록 모델의 양자화를 수행함.
 - *직관적으로 봤을 때, 양자화 전과 후의 결과 차이가 작다면 훌륭한 양자화라고 볼 수 있음.

</GPTQ [GPT Quantization]



- GPTQ는 양자화를 위한 작은 데이터셋을 준비하고, 그 데이터셋을 활용해 모델 연산을 수행하면서, 양자화 이전과 유사한 결과가 나오도록 모델을 업데이트 함.
 - GPTQ는 흰색 열의 양자화를 수행하고, 양자화를 위해 준비한 데이터를 입력한 결과가 이전과 최대한 가까워지도록 아직 양자화하지 않은 오른쪽 부분의 파라미터를 업데이트 함.
- *왼쪽은 이미 양자화한 열이므로 업데이트 하지 않는다.
*블록을 점차 오른쪽으로 이동시키고 양자화하는 열도 오른쪽으로 이동하면서 모델의 파라미터를 업데이트.

</GPTQ [GPT Quantization]



</AWG

- 모든 파라미터는 동등하게 중요하지는 않을 것
중요한 파라미터의 정보를 유지하면 양자화를 수행하더라도 성능 저하를 막을 수 있을 것.
- 그렇다면 중요한 파라미터는 어떻게 찾을 수 있을까?
 1. 모델 파라미터의 값이 크면 중요하다고 예상할 수 있음(연산 과정에 큰 영향을 줄 가능성이 있다).
 2. 입력 데이터의 활성화 값이 큰 채널의 파라미터가 중요하다고 가정할 수 있음.

</AWG

- MIT 연구진은 모델 파라미터 자체와 활성화 값을 기준으로 상위 1%에 해당하는 모델 파라미터를 찾고, 해당하는 파라미터는 기존 모델의 데이터 타입인 FP16으로 유지하고, 나머지는 양자화 했음.
=> 성능 저하가 거의 발생하지 않음
=> 중요한 1% 파라미터의 정보만 지키면 모델의 성능이 유지된다는 사실을 발견.
*모델 파라미터의 크기를 기준으로 모델 파라미터를 유지했을 때는 성능 저하가 발생.

</AWG

2.4	-2.1
1.5	1.7
-2.3	3.3
-4	1.4

w

양자화



5	-4
3	3
-5	7
-8	3

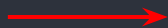
$Q[w]$

</AWG

2.4	-2.1
1.5	1.7
-2.3	3.3
-4	1.4

w

x2



2.4	-2.1
3	3.4
-2.3	3.3
-4	1.4

Q[ws]



5	-4
6	7
-5	7
-8	3

Q[ws]

</AWG

OPT-6.7B	s = 1	s = 1.25	s = 1.5	s = 2	s = 4
wiki-2 펼플렉시티	23.54	12.87	12.48	11.92	12.36

2.4	-2.1
1.5	1.7
-2.3	3.3
-4	1.4

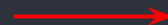
w

x4



2.4	-2.1
6	7.8
-2.3	3.3
-4	1.4

Q[ws]



2	-2
6	7
-2	3
-4	1

Q[ws]

</Knowledge Distillation; 지식 증류

- 더 크고 성능이 높은 선생 모델 (teacher model) 생성 결과를 활용해 더 작고 성능이 낮은 모델 (student model)을 만드는 방법.
- 일반적으로 학생 모델의 크기가 선생 모델에 비해 작기 때문에 선생 모델에 쌓은 지식을 더 작은 모델로 압축해 전달한다는 의미에서 증류라고 함.

</Knowledge Distillation; 지식 증류

- LLM 이전에도 더 작은 모델로 선생 모델과 비슷한 성능의 모델을 만들어, 효율적으로 모델을 활용하기 위한 방법으로 많이 활용.
- 하지만 GPT-3.5, GPT-4 같은 언어 모델이 뛰어난 능력을 보이면서, 과거에는 학습 데이터셋에 대한 선생 모델의 추론 결과를 학습 모델의 학습에 활용하는 정도였다면,
=> 최근에는 선생 모델을 활용해 완전히 새로운 학습 데이터셋을 대규모로 구축하거나 데이터셋 구축에 **사람의 판단이 필요한 부분을 선생 모델이 수행**하는 등 더 폭넓게 활용함.

</Knowledge Distillation; 지식 증류

- sLLM의 학습 데이터 구축에 GPT-4와 같은 대형 모델을 활용하는 경우가 일반적.
- 허깅페이스 팀이 개발한 예시를 살펴봄.
 1. 제퍼 모델
 2. 파이 모델

</zephyr-7b-beta

- 제퍼-7B-베타 모델은 미스트랄-7B 모델을 뛰어넘은 새로운 SOTA 모델.
- 제퍼의 개발 과정에서 GPT-4나 다른 대규모 언어 모델을 적극적으로 활용해 개발 속도를 높임.
- 4장에서 OpenAI가 ChatGPT를 개발하는 과정을 살펴보았는데, ChatGPT를 개발할 때, 지도 미세 조정에 사용하는 지시 데이터셋을 구축하는 데 많은 레이블러가 투입돼 프롬프트에 대한 응답을 직접 작성함. 2개의 응답 중 더 좋은 응답을 선택해 선호 데이터셋을 구축하는 데에도 레이블러가 직접 선택했음.
- 제퍼를 개발할 때에는 지시 데이터셋의 구축과 선호 데이터셋의 구축에 모두 LLM을 사용함.

</phi-1

- 파이-1 모델은 파라미터가 13억 개에 불과한 작은 모델임에도 파이썬 프로그래밍에서 훨씬 더 큰 모델과 비슷하거나 오히려 더 뛰어난 성능을 보임.
- 파이-1을 개발하면서 여러 코드에서 학습 데이터로 사용할 코드를 선택하는 데, GPT-3.5를 사용함.
- 코드 데이터셋에는 중요한 로직을 구현해 학생 모델이 학습하면 큰 도움이 되는 코드도 있지만, 단순히 설정을 위한 코드나 의미를 알기 어려운 코드도 많음.
- MS에서는 GPT-3.5를 사용해 프로그래밍 학습에 도움이 되는 코드인지 선별하는 작업을 수행함.
- 함수의 이름과 함수에 대한 설명인 독스트링(문서)을 입력하고, GPT-3.5가 입력에 대응하는 코드를 구현하도록 해서 코드 예제 데이터셋(CodeExercise)을 구축함.
=>사람이 수학을 배울 때 개념을 이해에 도움이 되는 예제 문제를 푸는 것처럼,
학생 모델이 학습할 때 도움이 되는 쉬우면서도 교육적 가치가 높은 데이터셋을 GPT-3.5를 사용해 구축.

Thank you