

GPU 효율적인 학습

개요

GPU

- 단순한 곱셈을 동시에 여러 개 처리하는 데 특화된 처리 장치
- 한정된 메모리
- 가격이 비쌈

→ 따라서 GPU를 효율적으로 활용할 수 있는 방법이 필요!

GPU 1개 사용할 때

- gradient accumulation
- gradient checkpointing

GPU 여러 개 사용할 때

- Deepspeed ZeRO

모델 일부 사용

- LoRA
- QLoRA

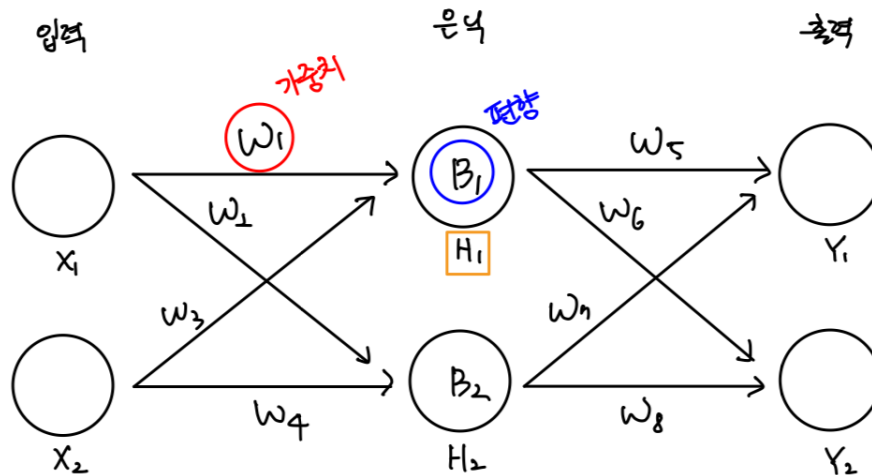
5.1 GPU에 올라가는 데이터 살펴보기

OOM

- Out of Memory
- 한정된 GPU 메모리에 데이터가 가득 차 더 이상 새로운 데이터를 추가하지 못해 발생하는 에러

GPU에 올라가는 것

- 딥러닝 모델
 - 행렬 곱셈을 위한 파라미터의 집합
 - 파라미터란?
 - ex) 파라미터 개수가 10개인 신경망



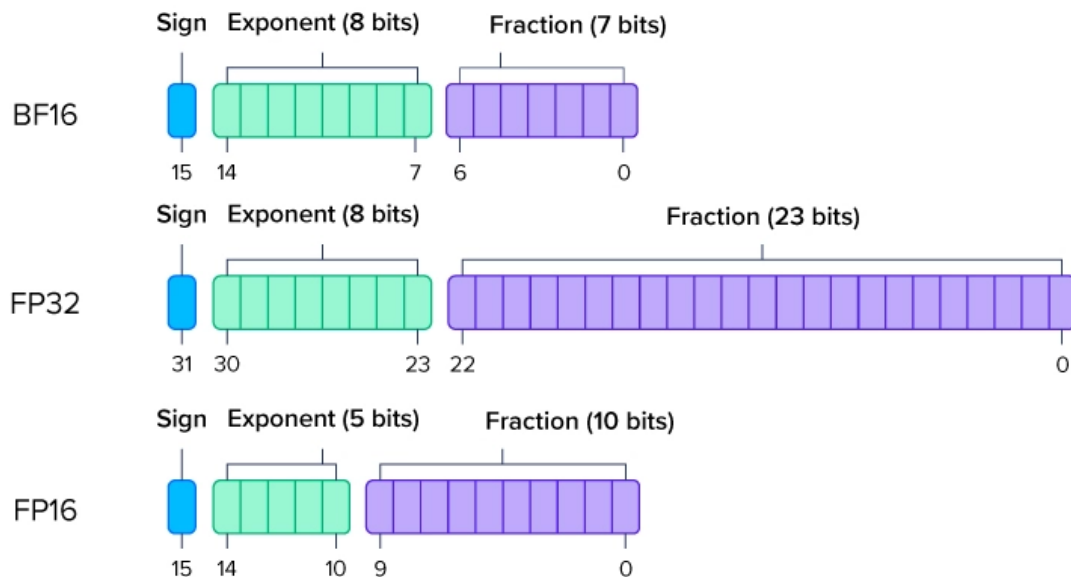
계산방법 : $H_1 = X_1 \cdot w_1 + X_2 \cdot w_3 + B_1$

파라미터 개수 : 가중치 8개 ($w_1 \dots w_8$) + 편향 2개 (B_1, B_2)
 = 10개

- 모델이 학습을 통해 조정해야 하는 **가중치**와 **편향**
- X_1, X_2 는 **데이터의 입력값**이다!
- 가중치와 편향은 **유리수**이다! 보통 소수점 이하의 값이 많음.
- 행렬 연산이라며? → 여러 파라미터를 한 행렬에 넣어서 계산하는 방식
- 데이터

5.1.1 딥러닝 모델의 데이터 타입

Floating Point (부동 소수점)



- **Sign** - 부호를 나타냄
- **Exponent** - 수를 표현할 수 있는 범위; 여기에 할당된 비트가 많으면 더 넓은 범위를 커버할 수 있음
- **Fraction** - 더 세밀하게 표현할 수 있는 크기; 여기에 할당된 비트가 많으면 소수점 자리가 깊어짐

FP32 → FP16 Why?

- 비트 수를 줄이면 계산이 빨라짐! 물론 세밀함은 떨어짐

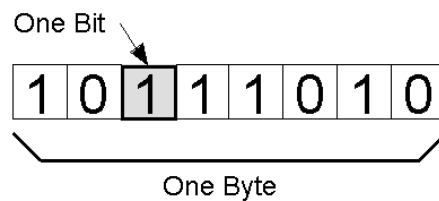
FP16 → BF16 Why?

- 지수 범위의 확장
 - BF16의 지수부가 8비트라서 FP32와 동일한 숫자 범위를 처리할 수 있음
 - 딥러닝에서 파라미터나 활성화 값들이 매우 크거나 작은 값으로 변할 수 있음
 - FP16은 지수 범위가 좁아서 큰 숫자나 작은 숫자를 정확하게 처리하지 못할 수 있음

- BF16은 지수 범위가 넓어 더 다양한 값들을 안정적으로 표현할 수 있음
- **정밀도**
 - 유효숫자부가 FP32보다 작긴 하지만, 실제로 많은 딥러닝 연산에서는 **아주 작은 차이**는 결과에 크게 영향을 미치지 않음
 - BF16은 **지수 부분**에서의 손실을 보완하고 그에 따른 손해는 감수함
- **성능과 효율성**
 - BF16은 FP32에 비해 훨씬 적은 메모리를 사용하면서도, **FP32와 유사한 성능**을 낼 수 있음

모델 용량 계산하기

bit 와 byte



1 byte = 8 bits

1000 byte = 1 kb

1000 kb = 1 mb

1000 mb = 1 gb

파라미터 수 X 파라미터 당 비트(또는 바이트) 수

ex) 7B 모델

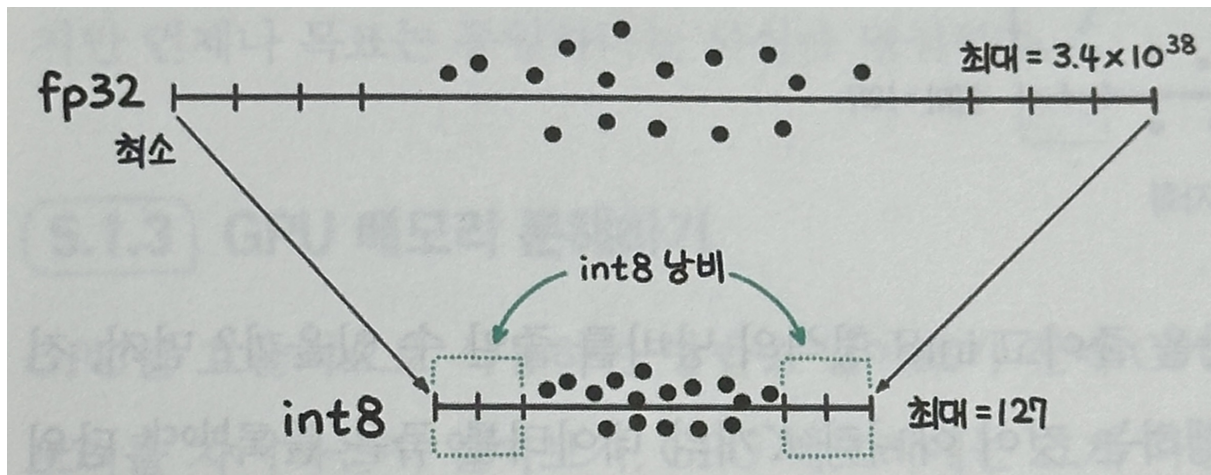
$$\frac{7,000,000,000 \times 2\text{bytes}}{1,024^3(\text{bytes}/\text{GB})} \simeq 2\text{GB}$$

5.1.2 양자화로 모델 용량 줄이기

양자화 기술의 목표

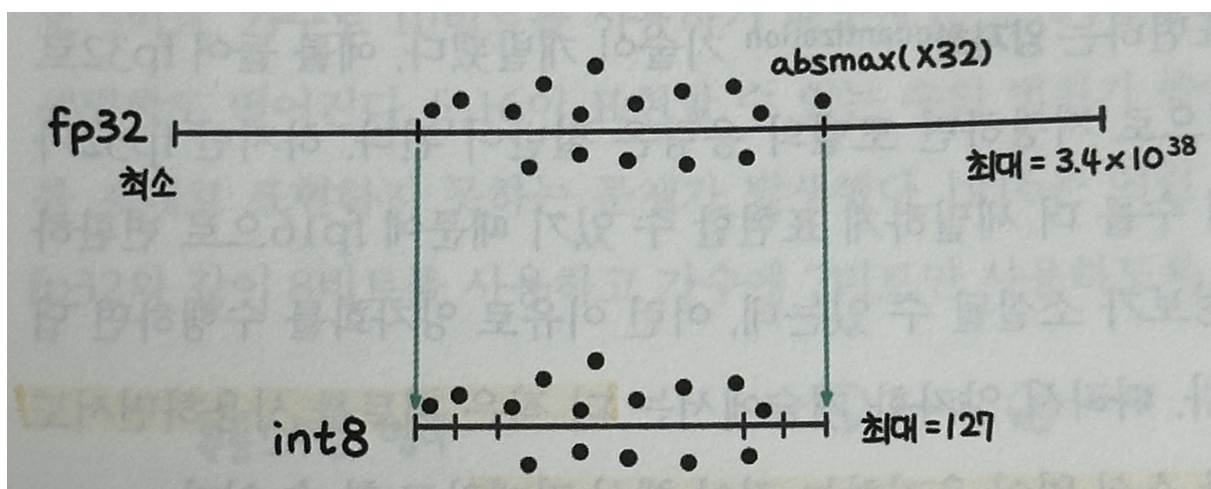
- 더 적은 비트를 사용하면서도 원본 데이터의 정보를 최대한 소실 없이 유지하는 것

일반적인 방법 (최대 최소 대응)



- 예를 들어 32 bit를 8 bit로 줄인다면, 32 bit에서 4 bit에 해당하는 값들이 8 bit에서 1 bit에 대응됨
- 이런 방식을 사용하면 단순하지만 데이터가 고르게 퍼져 있지 않는 경우 낭비가 발생

절대최댓값 기준으로 대응



- 그래서 두번째 방식은 데이터의 절대 최댓값을 이용하는 것!
- 이렇게 하면 좌우로 낭비되는 값들을 제거하여 낭비를 조금 줄일 수 있음

절대최댓값 기준 양자화 예시

$$\text{데이터} = [1, -2, 3, -4, 5, -6, 10, -20, 15]$$

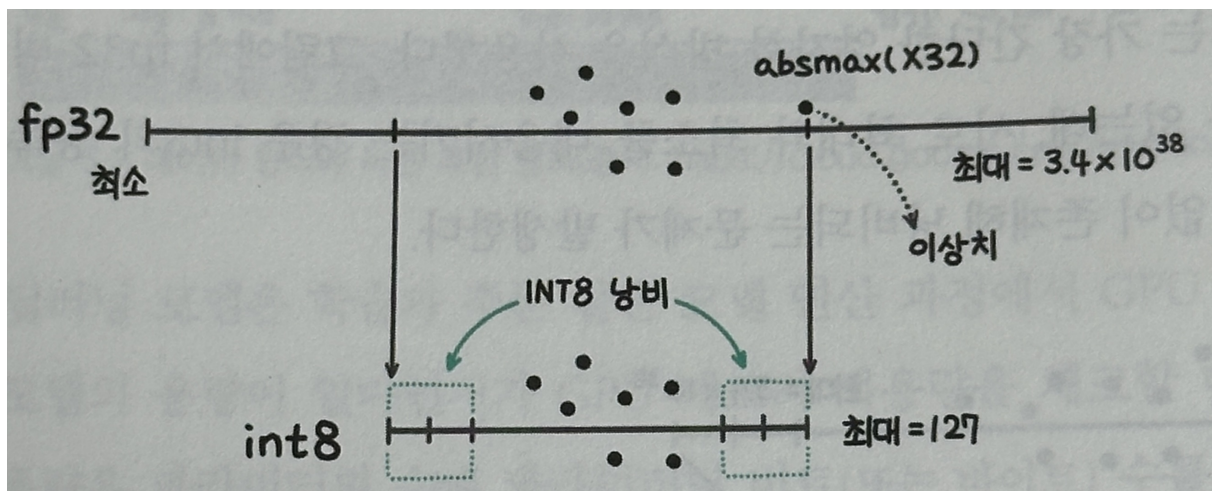
- 이 데이터에서 절대값이 가장 큰 값은 20

양자화 과정 (8비트 양자화):

- 8비트 양자화는 0~255 범위의 정수로 데이터를 표현하는 방식
- 모든 데이터를 -20에서 20 사이의 값으로 정규화한 후, 이 정규화된 값을 0~255 사이의 정수 값으로 대응

$$\text{데이터} = [134, 115, 147, 102, 159, 89, 191, 0, 218]$$

구간별 절대최댓값 기준으로 대응



- 절대 최댓값 방식으로 하면 한쪽으로 너무 튀어버리는 이상치가 있을 때 반대쪽 부분의 낭비가 심하다

블록 단위 양자화의 예시

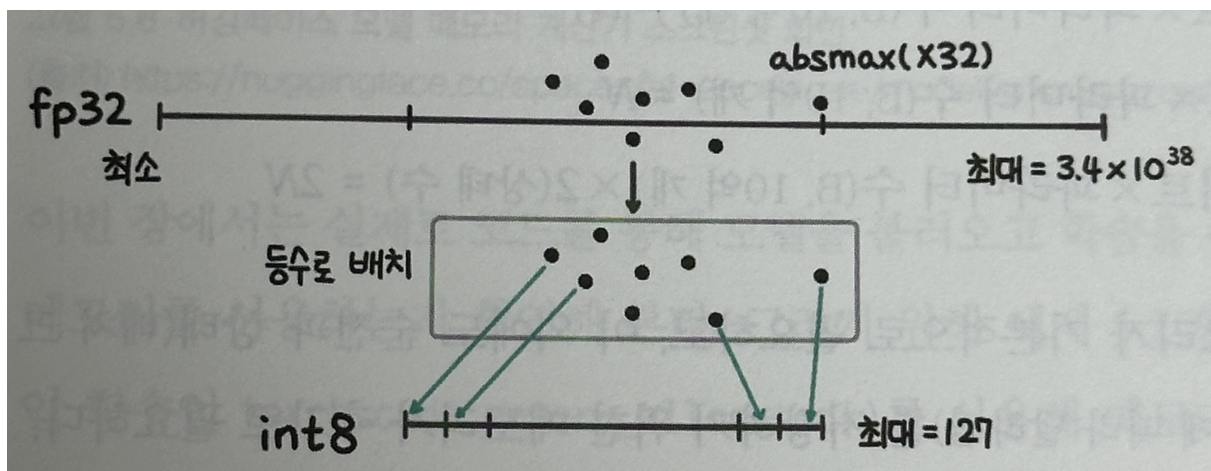
$$\text{데이터} = [1, 2, 3, 100, 4, 5, 6, 7, 200, 8, 9]$$

1. 이 데이터를 2개의 블록으로 나눈다

- 블록 1: [1, 2, 3, 100, 4]

- 블록 2: [5, 6, 7, 200, 8, 9]
2. 각 블록에서 **최대값**을 찾는다
- 블록 1의 최대값 = 100
 - 블록 2의 최대값 = 200
3. 각 블록을 **양자화**한다. 예를 들어, 블록 1의 값들을 **0~255** 범위로 양자화한다고 하면
- 블록 1: [1, 2, 3, 100, 4] → [2, 5, 7, 255, 10] (최대값 100에 맞게 양자화)
 - 블록 2: [5, 6, 7, 200, 8, 9] → [6, 8, 9, 255, 10, 11] (최대값 200에 맞게 양자화)

퀀타일 방식 (빈도를 고려)



- 쉽게 말해 더 데이터가 몰려있는 부분은 빈도를 고려하여 양자화를 하여 정밀도를 보장하는 것

→ 결국 우리가 하고 싶은 건 사용하는 메모리를 최대한 줄이면서, 정보를 최대한 유지하려고 한다는 것!

5.1.3 GPU 메모리 분해하기

GPU 메모리에 저장되는 것

- **모델 파라미터** → 앞서 설명했음. 이것의 용량을 N이라고 하면
- **gradient**
 - 모델의 손실 함수에 대해 각 가중치, 편향(즉 파라미터)의 변화량을 나타냄

$$w_{new} = w_0 - gradient (= \frac{\partial c}{\partial w})$$

- 그래서 파라미터의 개수와 일대일 대응
- 따라서 이것의 용량도 N
- **옵티마이저 상태**
 - 모델 파라미터를 업데이트 할 때 필요한 추가적인 정보
 - SGD 같은 단순한 옵티마이저는 필요 없음
 - Adam같은 녀석은 파라미터당 **모멘텀**과 **분산**을 저장
 - 따라서 파라미터의 두 배, 용량은 2N
- **순전파 상태**
 - batch 크기에 따라, 시퀀스 길이에 따라
 - 유동적이므로 플러스 알파로 생각하면 됨.

→ 따라서 GPU에는 파라미터의 개수 (N) X 4의 용량이 최소한으로 필요하다!

** 엔비디아 RTX 4070은 5천888개의 쿼드 코어 프로세서가 기본 1.92GHz, 최대 2.48GHz로 동작하며, **12GB GDDR6X 메모리**가 적용 **

5.2 단일 GPU 효율적으로 활용하기

Epoch

- 주어진 데이터 전체를 학습하는 단위

Batch

- 데이터 묶음
- 1 epoch / batch size 번의 연산을 해야 한바퀴를 돌.
- 총 데이터량이 100인데, batch size가 10이면 10번의 연산을 해야 1 epoch에 도달함.


```

cleanup()
print_gpu_utilization()

for batch_size in [4, 8, 16]:
    gpu_memory_experiment(batch_size)

    torch.cuda.empty_cache()

Special tokens have been added in the vocabulary, ma
[] GPU 메모리 사용량: 0.016 GB
배치 사이즈: 4
Loading checkpoint shards: 100% ██████████
[load_model_and_tokenizer] GPU 메모리 사용량: 2.615 GB
/usr/local/lib/python3.10/dist-packages/transformers
warnings.warn(
[train_model] GPU 메모리 사용량: 10.586 GB
옵티마이저 상태의 메모리 사용량: 4.961 GB
그래디언트 메모리 사용량: 2.481 GB
[] GPU 메모리 사용량: 0.016 GB
배치 사이즈: 8
/usr/local/lib/python3.10/dist-packages/huggingface_
warnings.warn(
Special tokens have been added in the vocabulary, ma
Loading checkpoint shards: 100% ██████████
[load_model_and_tokenizer] GPU 메모리 사용량: 2.615 GB
/usr/local/lib/python3.10/dist-packages/transformers
warnings.warn(
[train_model] GPU 메모리 사용량: 11.113 GB
옵티마이저 상태의 메모리 사용량: 4.961 GB
그래디언트 메모리 사용량: 2.481 GB
/usr/local/lib/python3.10/dist-packages/huggingface_
warnings.warn(
Special tokens have been added in the vocabulary, ma
[] GPU 메모리 사용량: 0.016 GB
배치 사이즈: 16
Loading checkpoint shards: 100% ██████████
[load_model_and_tokenizer] GPU 메모리 사용량: 2.615 GB
/usr/local/lib/python3.10/dist-packages/transformers
warnings.warn(
[train_model] GPU 메모리 사용량: 12.164 GB
CUDA out of memory. Tried to allocate 64.00 MiB. GPU
[] GPU 메모리 사용량: 0.016 GB

```

- 예상했던 대로 파라미터의 용량을 N이라고 했을 때 최소 4N의 메모리는 배치 사이즈에 상관없이 요구됨
- 그러나 학습의 과정에서 GPU의 메모리 사용량이 증가한다 why?
 - **배치 사이즈가 클수록** 한 번의 순전파(forward pass)에서 더 많은 입력 샘플을 동시에 처리
 - **배치 사이즈가 2배로** 늘어나면, 활성화 값을 저장하는 메모리도 **2배로** 증가

5.2.1 gradient 누적


여러 배치의 학습 데이터를 연산 후 모델을 한꺼번에 업데이트 해 큰 배치 사이즈의 효과를 가짐

```
cleanup()
print_gpu_utilization() # GPU 사용량을 반환함

gpu_memory_experiment(batch_size=4, gradient_accumulation_steps=4)

torch.cuda.empty_cache()

/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_downloads.py:100:
  warnings.warn(
Special tokens have been added in the vocabulary, make sure the ass
[] GPU 메모리 사용량: 0.016 GB
배치 사이즈: 4

Loading checkpoint shards: 100%  3/3 [00
[load_model_and_tokenizer] GPU 메모리 사용량: 2.615 GB
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:296:
  warnings.warn(
[train_model] GPU 메모리 사용량: 10.586 GB
옵티마이저 상태의 메모리 사용량: 4.961 GB
그레디언트 메모리 사용량: 2.481 GB
[] GPU 메모리 사용량: 0.016 GB
```

배치 $4 \times 4 = 16$ 개의 샘플에 대한 그레디언트를 계산한 후 업데이트

이 방식은 배치 사이즈가 16인 경우와 동일한 효과

그레디언트 누적의 장단점

- 장점:
 - 메모리 사용량을 줄여서 더 큰 모델이나 제한된 GPU 메모리에서도 학습이 가능.
 - GPU 메모리가 부족한 상황에서 유용한 대안.
- 단점:
 - 계산 시간이 오래 걸릴 수 있음.
 - BatchNorm 같은 레이어의 통계값이 달라질 수 있어, 작은 배치를 누적인 결과가 큰 배치 결과와 다를 수 있음.
 - 동기화 오버헤드로 성능이 저하될 가능성.

5.2.2 gradient checkpointing

학습 결과의 일부만 저장하여 메모리 사용량을 줄임

```
cleanup()
print_gpu_utilization()

gpu_memory_experiment(batch_size=16, gradient_checkpointing=True)

torch.cuda.empty_cache()

/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_downloads.py:100:
  warnings.warn(
Special tokens have been added in the vocabulary, make sure the assets url is up to date.
[ ] GPU 메모리 사용량: 0.016 GB
배치 사이즈: 16
Loading checkpoint shards: 100% ██████████ 3/3 [00:00<
[load_model_and_tokenizer] GPU 메모리 사용량: 2.615 GB
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:276:
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch/utils/checkpoint.py:290:
  with torch.enable_grad(), device_autocast_ctx, torch.cpu.amp.autocast():
[train_model] GPU 메모리 사용량: 10.290 GB
옵티마이저 상태의 메모리 사용량: 4.961 GB
그래디언트 메모리 사용량: 2.481 GB
[ ] GPU 메모리 사용량: 0.016 GB
```

5.3 분산 학습과 ZeRO

데이터 병렬화

작은 키보드가 여러 대 있다고 해보자. 모델 병렬화는 모든 GPU에게 이 작은 키보드를 하나씩 제공하고 일을 같이 하는 방식이다. 따라서 GPU의 개수만큼 연산 속도가 배로 증가한다.

모델 병렬화

한 층에 4호씩 존재하는 아파트가 있다고 해보자. 여기에 경비원이 4명이 있다.

1. 파이프라인 병렬화

- 각 경비원이 층별로 맡는다. 맨 아래층 경비원이 점검을 끝내면 다음 층 경비원이 이어 받는다.

2. 텐서 병렬화

- 각 경비원이 매 층의 1, 2, 3, 4호를 맡는다. 맨 아래층부터 각자 일을 끝내면 수합하여 다음 층으로 올라간다.

데이터 병렬화의 단점 극복: ZeRO

- 각 GPU가 모두 모델을 가지고 있어서 메모리를 비효율적으로 차지하고 있음
- Zero Redundancy Optimizer
- 쉽게 말해 데이터 병렬화 + 모델 병렬화
- 각 GPU가 모델을 부분적으로 가지고, 필요한 순간에만 모델 파라미터를 복사해서 사용

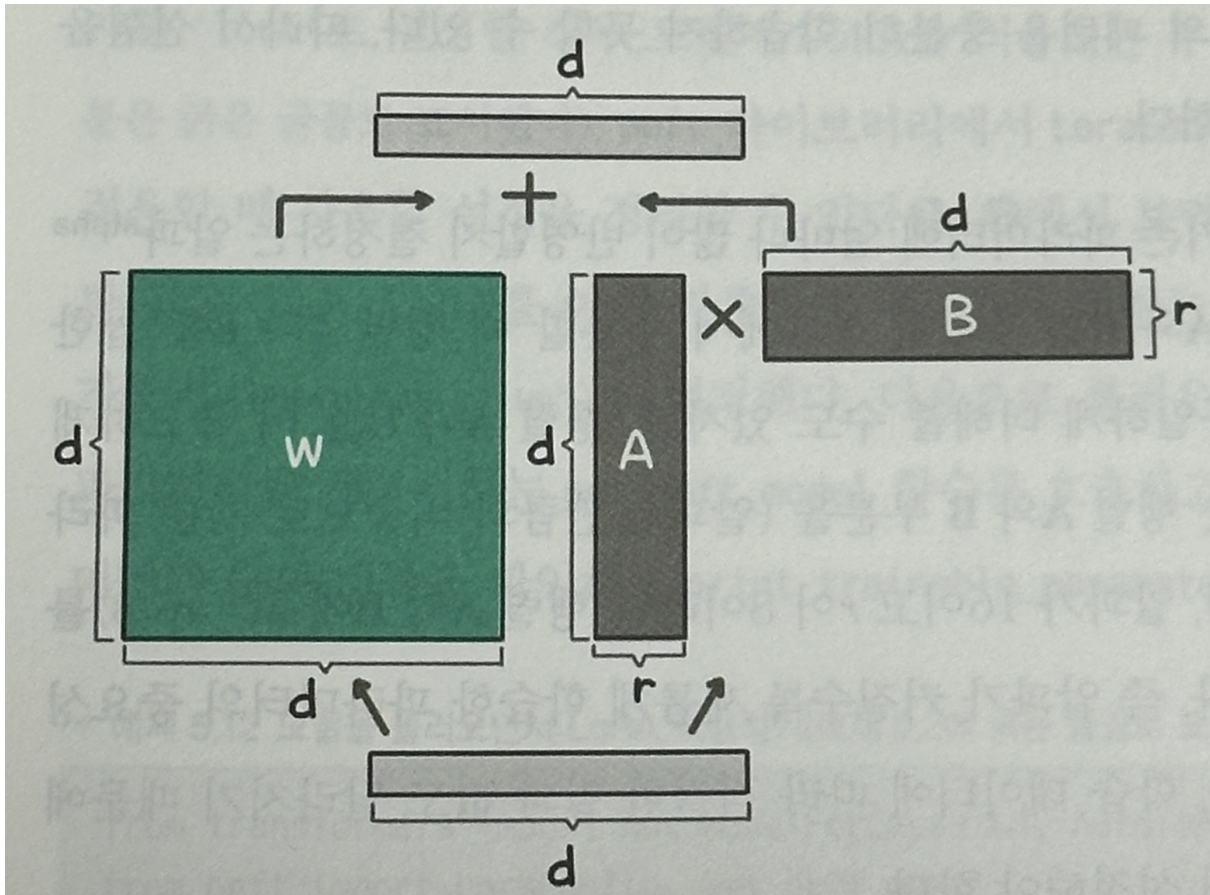
5.4 효율적인 학습 방법(PEFT): LoRA

PEFT

- Parameter Efficient Fine-Tuning
- LLM은 모델이 너무 커서 GPU 하나로 해결하기 힘들
- 따라서 일부 파라미터만 학습하는 방법 연구가 활발함

LoRA

- Low-Rank Adaptation (저차원 적용)
- d 차원 \times $(d \times d)$ 차원 = d 차원



- $(d \times r)$ 차원 \times $(r \times d)$ 차원 = d 차원
- 기존의 가중치 행렬 w 에 A , B 행렬이 추가되면 메모리 용량이 더 늘어나는 것이 아닌가요?
 - 소량 증가하긴 함
 - 하지만 여기서 중요한 사실은 이렇게 함으로써 일부 파라미터만 업데이트가 가능
 - 업데이트되는 일부 파라미터에 해당하는
 - 그레디언트
 - 옵티마이저 상태
 - 가 줄어들기 때문에 전체적인 필요한 메모리는 줄어든다!


```
elif peft == 'lora': # 이 부분이 기존의 코드에서 추가되었다
    model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype="auto", device_map={"":0})
    lora_config = LoraConfig(
        r=8, # d x d = (d x r) x (r x d); r이 너무 작으면 정밀도가 떨어짐; 적당한 값은 시도를 통해 구할 수 있음
        lora_alpha=32, # 새롭게 만든 A, B 행렬의 중요도(학습의 영향)를 결정한다; alpha/r 만큼 A,B 가 만드는 업데이트의 영향이 올라감
        target_modules=["query_key_value"],
        lora_dropout=0.05,
        bias="none",
        task_type="CAUSAL_LM"
    )
```

```

cleanup()
print_gpu_utilization()

gpu_memory_experiment(batch_size=16, peft='lora')

torch.cuda.empty_cache()

[] GPU 메모리 사용량: 0.016 GB
배치 사이즈: 16
/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_download.py:1150: FutureWarning:
  warnings.warn(
Special tokens have been added in the vocabulary, make sure the associated word embeddings are
Loading checkpoint shards: 100%  3/3 [00:01<00:00, 1.65it/s]
trainable params: 1,572,864 || all params: 1,333,383,168 || trainable%: 0.11796039111242178
[] GPU 메모리 사용량: 2.618 GB
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:521: FutureWarning: Th
  warnings.warn(
[train_model] GPU 메모리 사용량: 4.732 GB
옵티마이저 상태의 메모리 사용량: 0.006 GB
그래디언트 메모리 사용량: 0.003 GB
[] GPU 메모리 사용량: 0.016 GB

```

5.5 효율적인 학습 방법(PEFT): QLoRA

QLoRA

- 쿼타일 양자화 + LoRA
- 4비트 양자화 → 4비트로 변환 후 이 값과 표준편차를 저장 (NF4 라고 표현함)
 - 필요할 때 저장된 값을 역으로 계산하여 값을 알아냄
- 2차 양자화 → 4비트 양자화 후 8비트 양자화

페이지 옵티마이저


- 체크포인팅 중 일어날 수 있는 OOM 방지를 위함
- GPU의 램이 가득 찼을 때 일부 데이터를 CPU 메모리에 보관했다가 필요할 때 찾는 방식

```
from transformers import BitsAndBytesConfig
nf4_config = BitsAndBytesConfig(
    load_in_4bit=True, # 4비트 양자화를 하겠다
    bnb_4bit_quant_type="nf4", # 4비트의 형식은 nf4를 쓰겠다
    bnb_4bit_use_double_quant=True, # 2차 양자화를 하겠다
    bnb_4bit_compute_dtype=torch.bfloat16
)
model_nf4 = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=nf4_config)
```

```
cleanup()
print_gpu_utilization()

gpu_memory_experiment(batch_size=16, peft='qlora')

torch.cuda.empty_cache()

Special tokens have been added in the vocabulary, make sure the associated word embeddings are trained
[] GPU 메모리 사용량: 0.945 GB
배치 사이즈: 16
Loading checkpoint shards: 100%  3/3 [00:02<00:00, 1.37it/s]
trainable params: 1,572,864 || all params: 1,333,383,168 || trainable%: 0.11796039111242178
[] GPU 메모리 사용량: 2.112 GB
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:521: FutureWarning: This warning will be removed in transformers v4.33.0.
warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch/_dynamo/eval_frame.py:600: UserWarning: torch.cuda.set_device is deprecated.
return fn(*args, **kwargs)
/usr/local/lib/python3.10/dist-packages/torch/autocast/_init.py:295: FutureWarning: `torch.cuda.is_available` will be deprecated in 1.12.0.
with torch.enable_grad(), device autocast_ctx, torch.cpu.amp.autocast(**ctx.cpu_autocast_kwargs):
[train_model] GPU 메모리 사용량: 2.651 GB
옵티마이저 상태의 메모리 사용량: 0.012 GB
그레디언트 메모리 사용량: 0.006 GB
[] GPU 메모리 사용량: 0.945 GB
```

LoRA와 QLoRA 비교

항목	LoRA	QLoRA
핵심 아이디어	저차원 행렬로 파라미터 조정	저차원 조정 + 양자화 적용
메모리 절감	원본 모델은 FP16/FP32로 유지	원본 모델을 4비트로 양자화
성능	원래 모델과 유사한 성능	LoRA와 유사한 성능, 더 적은 메모리
적용 대상	대규모 모델 미세 조정	제한된 메모리 환경에서 미세 조정
주요 활용	대규모 언어 모델(LLM) 미세 조정	작은 GPU 환경에서 대규모 모델 학습