



본 영상 교재는  
2025년도 과학기술정보통신부 및 정보통신기획평가원의  
SW중심대학 사업의 지원을 받아 제작되었습니다.



한국항공대학교  
Korea Aerospace University

—KIU—  
AI융합대학

SW중심대학

# PyTorch 코딩

---

Large ANN과 FSDP

# Large ANN

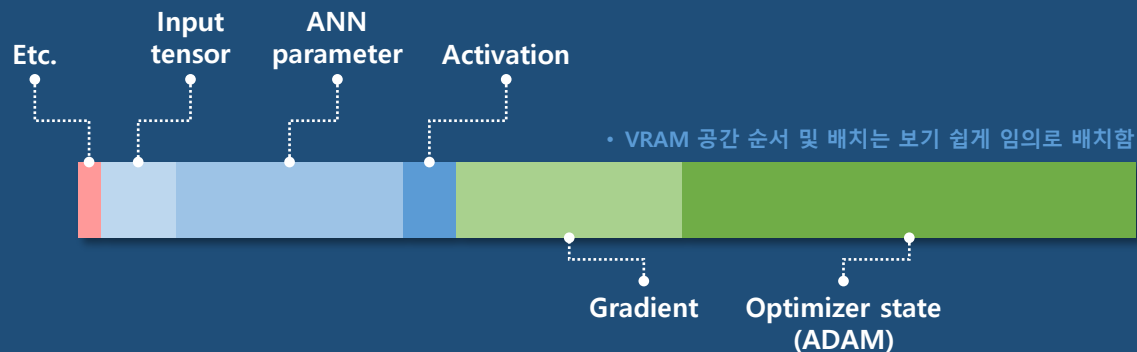
# 01

# Large ANN

- Chat-GPT 서비스의 핵심인 LLM (Large Language Model) ANN은 그 크기가 매우 큼
  - ANN의 크기가 크다는 말은 parameter의 수가 많다는 뜻
  - OpenAI의 GPT 3는 약 1,750억 개의 parameter로 이루어짐
  - GPT 4는 GPT 3의 약 100배의 parameter로 이루어짐
  - Meta의 LLAMA 2는 다양한 모델이 있으며, 가장 작은 것은 70억 개, 가장 큰 것은 1,300억 개의 parameter로 이루어짐
- ANN의 크기가 크면 training 및 inference에 필요한 VRAM 공간이 커짐
  - 일반적으로 각 parameter는 4 바이트(32 비트) 실수로 이루어져 있으므로, parameter 개수의 4배의 공간이 필요함
    - Mixed precision을 사용하더라도 최소 2배의 공간이 필요함
  - Training 작업의 경우 gradient와 optimizer state등을 저장할 추가 공간이 필요함

# Large ANN

- ANN training에 사용되는 VRAM 공간



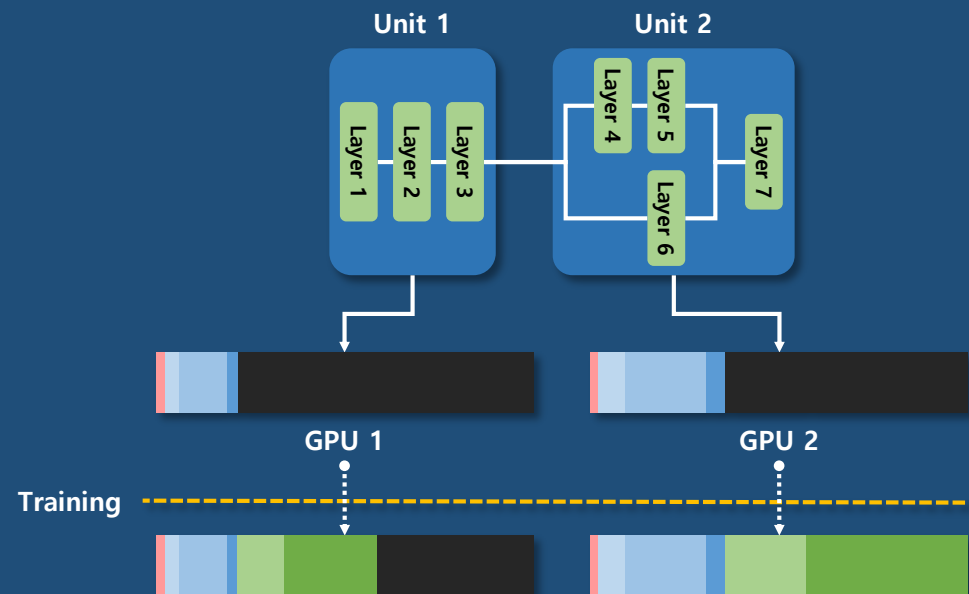
- **Training** 작업에서만 사용되는 **gradient**와 **optimizer state**의 VRAM 사용량이 굉장히 큼
  - Gradient는 각 parameter의 변화량을 나타내므로, parameter와 VRAM 사용량이 같음
  - ADAM optimizer와 같이 각 parameter의 이동 거리를 저장하는 optimizer는 VRAM 사용량이 굉장히 큼
  - Training이 진행됨에 따라 각 요소들의 VRAM 사용량이 달라질 수 있으나 대략적인 비율은 위와 비슷함

# Large ANN

- RAM에 비해 **VRAM**은 매우 작음
  - nVidia H100과 같이 개당 수천 만원인 GPU도 VRAM이 80 GB 밖에 되지 않음
- 따라서 매우 큰 ANN을 training하기 위해선 하나의 ANN을 **여러 GPU에 나누어** 저장해야 함
- PyTorch에는 하나의 큰 ANN을 여러 GPU에 나누는 두 가지 방법이 있음
  - Pipelining
    - 하나의 ANN을 **layer** 단위로 개발자가 **직접** 나누어 각 GPU에 저장하는 방식
    - Model **partitioning**이라고도 함
  - Fully sharded data parallel (FSDP)
    - ANN 뿐만 아니라 gradient 및 optimizer 등 모든 내용을 **자동**으로 **균등**하게 나누어 각 GPU에 저장하는 방식
    - Model **sharding**이라 함

# Large ANN

- PyTorch pipelining
  - 하나의 ANN을 **layer 단위로 직접** 나누어 각 GPU에 나누어 저장하는 방식



# Large ANN

- PyTorch pipelining은 코딩의 어려움에 비해 성능이 좋지 않음
  - Training에 필요한 추가 VRAM 공간을 예상하여 ANN을 나눠야 함
  - 일반적으로 layer마다 parameter 및 gradient의 수가 다름
    - ANN을 layer 단위로 직접 나누면 parameter를 균등하게 나누기 어려움
    - 균등하게 나누지 못한 경우 일부 GPU는 VRAM을 모두 사용하지 못하거나, VRAM이 부족할 수 있음
  - Tensor들이 GPU 사이를 이동하는 횟수와 양을 최소화해야 함
    - Tensor가 다른 GPU로 이동하는 것이 아니라 복사되므로, VRAM 공간을 중복으로 활용하여 낭비하게 됨
    - VRAM 공간이 부족하기 때문에 pipelining을 활용하는데, VRAM 공간을 낭비하게 되면 그 효용이 사라짐
  - 여러 node를 동시에 활용할 경우 RPC (Remote Procedure Call)를 사용하여 코드가 복잡해짐
- 따라서 본 강의에서는 FSDP만을 다룸
  - FSDP를 제대로 활용하기 위해선 PyTorch 2.4 이상을 설치해야 함



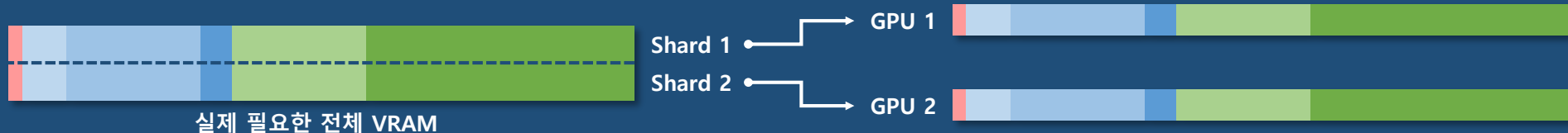
# Fully Sharded Data Parallel

02

# Fully Sharded Data Parallel

- Fully sharded data parallel (FSDP)

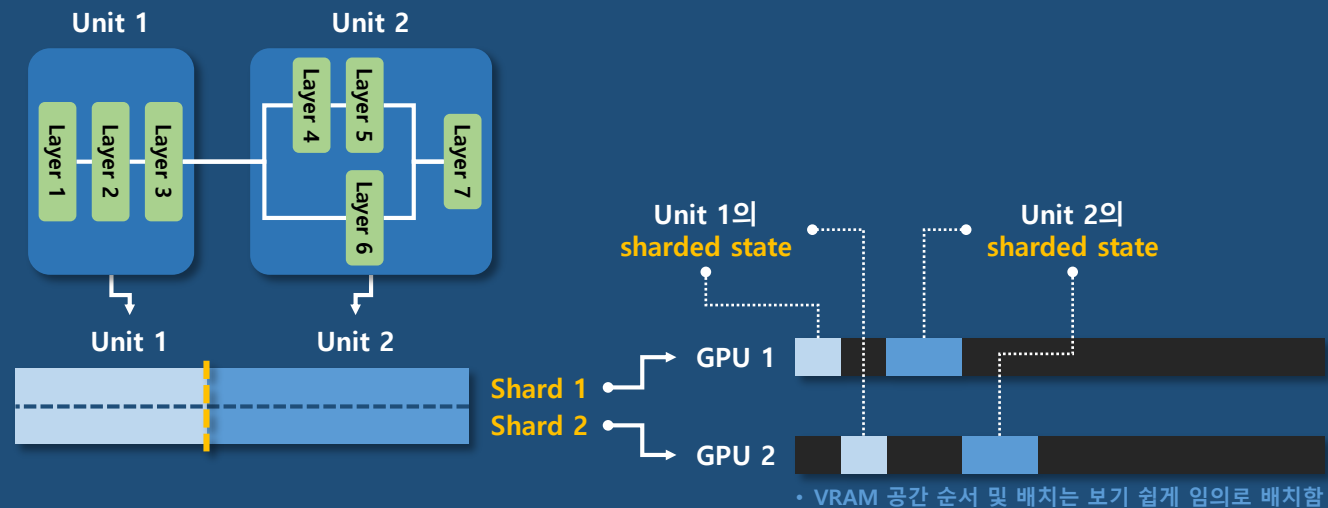
- Pipelining과 달리 ANN 및 optimizer 등 모든 내용을 **균등하게 자동**으로 나누어 각 GPU에 저장하는 방식
  - 각 GPU에 나누어 저장한 조각을 **shard**라 함



- 모든 내용이 균등하게 나누어 저장하므로 각 모든 VRAM을 **최대한** 활용할 수 있음
- VRAM 공간을 직접 계산하거나, shard를 직접 관리할 필요가 없어 **개발이 쉬움**

# Fully Sharded Data Parallel

- FSDP의 feed-forward 작업 수행 과정



- 전체 ANN을 **unit** 단위로 자동으로 나눔(auto wrapping)
- ANN parameter를 초기화하면서, unit들을 **shard**로 나누어 각 GPU에 저장

# Fully Sharded Data Parallel

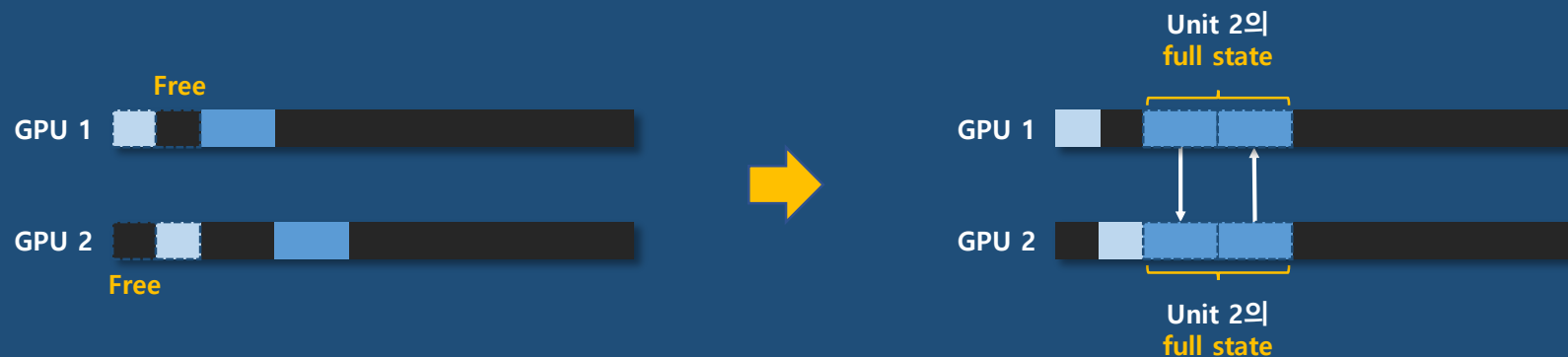
- FSDP의 feed-forward 작업 수행 과정



- 첫 번째 unit의 feed-forward를 수행하기 위해, 각 프로세스가 해당 unit의 shard를 다른 프로세스들로부터 모음(all-gather)
  - 프로세스 개수만큼의 첫 번째 unit이 만들어짐
  - All-gather를 통해 unit의 모든 shard가 합쳐진 것을 full state라 함
- Distributed sampler가 나눈 dataset을 통해 각 프로세스의 unit들이 동시에 feed-forward 작업을 수행

# Fully Sharded Data Parallel

- FSDP의 feed-forward 작업 수행 과정



- 각 GPU가 all-gather를 통해 다른 프로세스로부터 받았던 shard를 제거(free)
- 3번부터 5번까지의 작업을 반복하여 마지막 unit의 feed-forward까지 수행

# Fully Sharded Data Parallel

- FSDP의 back-propagation 작업 수행 과정

1. 각 프로세스는 **all-gather**를 통해 마지막 unit의 shard를 모음
2. 각 프로세스는 loss를 통해 마지막 unit의 gradient를 구함
3. 모든 프로세스의 gradient **평균값**을 구한 뒤, 이를 shard로 나누어 줌(**reduce-scatter**)
4. 각 프로세스가 all-gather를 통해 **다른 프로세스**로부터 받았던 **shard**를 제거(**free**)
  - 2번부터 5번까지의 작업을 통해 **평균 gradient**가 각 프로세스의 **shard**로 나누어져 저장됨
6. 2번부터 5번까지의 작업을 반복하여 첫 번째 unit까지의 평균 gradient까지 구함
7. 각 프로세스는 자신의 **shard**에 있는 parameter들을 gradient를 통해 수정
  - ANN parameter를 수정할 때 **optimizer state**도 각 **shard**에 저장됨

# Fully Sharded Data Parallel

---

- FSDP의 checkpoint

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 1-1 : PyTorch FSDP 라이브러리 불러오기

```
> import random
> import os
> import functools

> import torch
> from torch import nn
> from torch.utils.data import DataLoader
> # from torch.amp import autocast, GradScaler

> import torch.distributed as dist
> from torch.utils.data.distributed import DistributedSampler
> # from torch.nn.parallel import DistributedDataParallel as DDP

> ...
```

# Auto wrapping 정책을 생성하는데 필요한 라이브러리

# PyTorch 2.4의 FSDP는 AMP를 지원하지 않기 때문에, 관련 코드 삭제

# 기존 DDP 클래스 삭제



# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 1-2 : PyTorch FSDP 라이브러리 불러오기

```
> ...  
  
> from torch.distributed.fsdp import (  
>     FullyShardedDataParallel as FSDP,  
>     wrap  
> )  
  
> import torch.distributed.checkpoint as dcp  
> from torch.distributed.checkpoint.stateful import Stateful  
> from torch.distributed.checkpoint.state_dict import (  
>     get_state_dict,  
>     set_state_dict,  
>     get_model_state_dict,  
>     StateDictOptions  
> )  
  
> import torchvision  
> from torchvision.transforms import ToTensor
```

# FSDP 클래스  
# Auto wrapping 기준을 제공하는 라이브러리

# 분산 환경 checkpoint 라이브러리  
# 분산 환경 checkpoint 클래스

# 분산 환경에서 ANN과 optimizer의 상태를 추출하는 함수  
# 분산 환경에서 ANN과 optimizer의 상태를 설정하는 함수  
# 분산 환경에서 ANN parameter를 추출하는 함수  
# 분산 환경 checkpoint 생성 정책을 결정하는 클래스

# Fully Sharded Data Parallel

- 간단한 FSDP 실습
  - Cell 4 : ANN feed-forward 작업에서 AMP 관련 코드 삭제하기

```
> class NeuralNetwork(nn.Module):  
>     ...  
  
>     # @autocast(device_type = 'cuda')    # AMP 관련 코드 삭제  
>     def forward(self, x):  
>         x1 = self.cnn_stack1(x)  
>         x1 = self.linear_stack1(x1)  
  
>         x2 = self.cnn_stack2(x)  
>         x2 = self.linear_stack2(x2)  
  
>         x = x1 + x2  
>         y = self.softmax_stack(x)  
>  
>         return y
```

# Fully Sharded Data Parallel

## • 간단한 FSDP 실습

### • Cell 5-1 : Checkpoint 클래스 생성하기

```
> class DistributeCheckpoint(Stateful):
>     def __init__(self, fsdp_model, optimizer, scheduler):
>         self.fsdp_model = fsdp_model
>         self.optimizer = optimizer
>         self.scheduler = scheduler
>
>     def state_dict(self):
>         model_state, optimizer_state = get_state_dict(self.fsdp_model, self.optimizer)
>         checkpoint_state = {
>             'model_state' : model_state,
>             'optimizer_state' : optimizer_state,
>             'scheduler_state' : self.scheduler.state_dict()
>         }
>
>         return checkpoint_state
>
>     def load_state_dict(self, checkpoint_state):
>         set_state_dict(
>             self.fsdp_model,
>             self.optimizer,
>             model_state_dict = checkpoint_state['model_state'],
>             optim_state_dict = checkpoint_state['optimizer_state']
>         )
>
>         self.scheduler.load_state_dict(checkpoint_state['scheduler_state'])
```

# 분산 환경 checkpoint 클래스, Stateful 클래스를 상속해야 함  
# Checkpoint 생성에 필요한 초기화 작업을 수행, 반드시 구현해야 함

# 분산 환경 checkpoint를 저장할 때 호출됨, 반드시 구현해야 함  
# get\_state\_dict()를 통해 ANN 및 optimizer의 상태를 추출

# Scheduler의 상태는 직접 추출해야 함

# 분산 환경 checkpoint를 불러올 때 호출됨, 반드시 구현해야 함  
# set\_state\_dict()를 통해 ANN 및 optimizer의 상태를 설정

# Scheduler의 상태는 직접 설정해야 함

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-2 : Checkpoint 저장 및 불러오기 작업 변경하기

```

> def save_checkpoint(fsd_model, optimizer, scheduler):
>     checkpoint = {
>         'checkpoint' : DistributeCheckpoint(fsd_model, optimizer, scheduler)
>     }
>
>     dcp.save(
>         checkpoint,
>         checkpoint_id = 'fsdp_checkpoint'
>     )
>
> def load_checkpoint(fsd_model, optimizer, scheduler):
>     checkpoint = {
>         'checkpoint' : DistributeCheckpoint(fsd_model, optimizer, scheduler)
>     }
>
>     dcp.load(
>         checkpoint,
>         checkpoint_id = 'fsdp_checkpoint'
>     )
    
```

# 5-1에서 생성한 분산 환경 checkpoint 클래스 활용

# 분산 환경에서 파일 저장  
# 파일로 저장할 내용  
# 분산 파일을 저장할 위치

# 5-1에서 생성한 분산 환경 checkpoint 클래스 활용

# 분산 환경에서 파일 읽기  
# 파일에서 읽은 내용을 저장할 변수  
# 분산 파일의 위치

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-2 : Checkpoint 저장 및 불러오기 작업 정의하기
  - FSDP 환경에서의 checkpoint를 저장하는 방법
    - FSDP 환경에서는 각 프로세스가 ANN과 optimizer 등을 shard로 나누어 관리함
    - 따라서 모든 프로세스들의 shard를 모아 full state로 만든 후 하나의 파일로 저장하거나, 각 프로세스가 자신의 shard를 각자 저장하는 방법이 있음
  - 본 예시에서는 각 프로세스가 자신의 shard를 각자 저장하도록 함
    - 정해진 위치에 프로세스 개수만큼의 파일이 저장됨
    - 이후 프로세스의 개수를 바꾸어 training을 다시 실행해도 정상 작동함
  - 일부 PyTorch 버전에서는 프로세스 개수가 2의 제곱수가 아닌 경우 에러가 발생할 수 있음

# Fully Sharded Data Parallel

---

- 간단한 FSDP 실습
  - Cell 5-2 : Checkpoint 저장 및 불러오기 작업 정의하기
    - DCP 어찌구 멀티 노드 어찌구
    - 원격 노드 마우트 방법 어찌구

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-3 : Training epoch에서 AMP 관련 코드 삭제

```

> def train(data_loader, fsdp_model, optimizer, accumulation_number = 1):
>     local_rank = int(os.environ['LOCAL_RANK'])
>     distributed_loss = torch.zeros(2).to(local_rank)

>     fsdp_model.train()
>     # scaler = GradScaler()                                # AMP의 gradient scaler 삭제
>     for mini_batch_index, (x, t) in enumerate(data_loader):
>         x = x.to(local_rank)

>         y = fsdp_model(x)
>         t = t.to(y.device)
>         loss = torch.nn.functional.cross_entropy(y, t)

>         loss.backward()                                    # Gradient scaler 관련 코드를 삭제해 기존 코드로 복원

>         if mini_batch_index % accumulation_number == 0:
>             # scaler.unscale_(optimizer)                    # Gradient scaler 관련 코드를 삭제해 기존 코드로 복원
>             torch.nn.utils.clip_grad_norm_(fsdp_model.parameters(), 1e-1)
>             optimizer.step()                                # Gradient scaler 관련 코드를 삭제해 기존 코드로 복원
>             # scaler.update()                                # Gradient scaler 관련 코드를 삭제해 기존 코드로 복원
>             optimizer.zero_grad()

>         mini_batch_size = x.shape[0]
>         ...
    
```

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-4 : Training loop에서 FSDP ANN 생성하기

```
> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):
>     ...
>
>     model = NeuralNetwork()                # RAM에 ANN을 생성
>
>     wrapping_policy = functools.partial(    # Auto wrapping 정책을 인스턴스로 생성
>         wrap.size_based_auto_wrap_policy,  # Parameter 개수를 기준으로 unit 생성, PyTorch가 기본적으로 제공하는 policy 중 하나
>         min_num_params = 512              # Unit을 생성할 parameter 개수 기준
>     )
>
>     fsdp_model = FSDP(                     # 일반 ANN을 FSDP ANN으로 변환해 GPU에 저장, 분산 프로세스 그룹 생성 후에 수행 가능
>         model,
>         auto_wrap_policy = wrapping_policy # 위에서 만든 auto wrapping policy 적용
>     )
>
>     optimizer = torch.optim.Adam(fsdp_model.parameters(), lr = 1e-2, betas = (0.9, 0.999), weight_decay = 1e-4)
>     scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size = 1, gamma = 0.5)
>
>     ...
```



# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-5 : Training loop에서 checkpoint 불러오기

```
> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):  
>     ...  
  
>     current_epoch = 0  
>     if os.path.exists('fsdp_checkpoint'):  
>         load_checkpoint(fsdp_model, optimizer, scheduler)  
>         current_epoch = scheduler.last_epoch  
  
>     if local_rank == 0:  
>         print(f'Resuming training from checkpoint at epoch {current_epoch + 1}\n', end = '')  
>         dist.barrier(device_ids = [local_rank])  
  
>     ...
```

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-6 : Training loop 수정하기

```
> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):
>     ...
>
>     for t in range(current_epoch, max_epoch):
>         training_data_loader.sampler.set_epoch(t)
>         print(f'Worker {global_rank + 1} / {world_size} begins Epoch {t + 1 :> 3d} / {max_epoch}Wn', end = '')
>         training_loss = train(training_data_loader, fsdp_model, optimizer, accumulation_number)
>         scheduler.step()
>
>     if local_rank == 0:
>         print(f' Training average loss: {training_loss :>8f}Wn', end = '')
>     ...
```

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-7 : Training loop에서 checkpoint 저장하기

```
> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):
>     ...
>
>     for t in range(current_epoch, max_epoch):
>         ...
>
>         if (t + 1) % checkpoint_interval == 0 and (t + 1) != max_epoch:
>             # if local_rank == 0:                # Local rank와 상관 없이 모든 프로세스 수행해야 함
>             save_checkpoint(fsdp_model, optimizer, scheduler)    # 들여쓰기 수정
>
>             if local_rank == 0:
>                 print(f'Saved training checkpoint at {t + 1} epochs under "fsdp_checkpoint"Wn', end = '')
>                 dist.barrier(device_ids = [local_rank])
>             ...
```

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-8 : Training loop에서 최종 ANN parameter 저장하기

```

> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):
>     ...
>
>     if global_rank == 0:
>         excution_time, peak_VRAM_usage = get_cuda_performace_record()
>         ...
>
>     state_dict_option = StateDictOptions(                                # 분산 환경 checkpoint 생성 옵션 인스턴스 생성
>         full_state_dict = True                                           # 모든 shard들을 모아 full state를 만들도록 함
>     )
>     model_state = get_model_state_dict(fsd_model, options = state_dict_option) # FSDP ANN의 parameter shard를 모아 full state를 생성, 모든 프로세스가 수행해야 함
>
>     if global_rank == 0:                                                # Full state는 하나의 프로세스만 저장해도 됨
>         torch.save(model_state, 'model.pth')                            # ANN parameter의 full state를 하나의 파일로 저장
>
>     print('Saved PyTorch ANN parameters to model.pth\n', end = '')
    
```

# Fully Sharded Data Parallel

- FSDP 실습

- Torchrun을 통한 deep learning 프로그램 실행
  - Node가 하나뿐인 경우

```
> torchrun W                                # PyTorch Torchrun 실행
> --standalone W                            # 하나의 node만 활용하라고 지정
> --nproc-per-node=gpu W                   # 현재 node에서 실행할 프로세스 개수, gpu로 설정할 경우 node에 설치된 모든 GPU 활용
> 6-fsdp.py                                # 실행할 파일 이름
```

- Node가 여러 개인 경우, 각 node에서 아래 명령어 실행
- Hosts 파일에 동원할 node의 이름과 IP를 추가해줘야 함

```
> torchrun W
> --nnodes=? W                             # 활용할 node 개수
> --node-rank=? W                          # 현재 node의 정수 ID, 각 node는 서로 다른 ID를 가져야 함
> --nproc-per-node=gpu W
> --rdzv-id=? W                            # 랑데부 포인트의 정수 ID, 각 node에게 같은 ID를 알려주어야 함
> --rdzv-backend=c10d                      # 랑데부 포인트 백엔드 종류, c10d로 설정
> --rdzv-endpoint=???-???-???-??? W       # 랑데부 포인트 역할을 할 node의 IP 또는 host name
> 6-fsdp.py
```

## 고급 FSDP 기능

# 03

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 1 : PyTorch FSDP 고급 기능 라이브러리 불러오기

```
> import random
> import os
> import functools

> import torch
> from torch import nn
> from torch.utils.data import DataLoader

> import torch.distributed as dist
> from torch.utils.data.distributed import DistributedSampler
> from torch.distributed.fsdp import (
>     FullyShardedDataParallel as FSDP,
>     wrap,
>     CPUOffload,      # ANN이 너무 큰 경우 CPU와 RAM도 활용하도록 하는 클래스
>     MixedPrecision,  # FSDP mixed precision 정책을 생성하기 위한 클래스
>     BackwardPrefetch, # Backward prefetch를 위한 클래스
>     ShardingStrategy # Sharding strategy를 변환할 때 필요한 클래스
> )

> ...
```

# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-1 : Training loop 수정하기

```
> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):
>     ...
>     wrapping_policy = functools.partial(wrap.size_based_auto_wrap_policy, min_num_params = 512)
>     cpu_offload = CPUOffload(offload_params = True)    # GPU VRAM이 부족한 경우 CPU와 RAM을 추가로 활용
>     bfloat16_policy = MixedPrecision(                 # FSDP ANN에 적용할 mixed precision 정책 인스턴스 생성
>         param_dtype = torch.bfloat16,                 # ANN parameter로 16bit brain-float 활용
>         reduce_dtype = torch.bfloat16,                 # Reduce 작업에 16bit brain-float 활용
>         buffer_dtype = torch.bfloat16                 # Buffer로 16bit brain-float 활용
>     )
>     torch.cuda.set_device(local_rank)
>     fsdp_model = FSDP(
>         model,
>         auto_wrap_policy = wrapping_policy,
>         device_id = torch.cuda.current_device(),        # FSDP ANN을 초기화 할 때도 GPU를 활용하여 속도를 높임
>         cpu_offload = cpu_offload,                    # 위에서 만든 CPU offload 방식 적용
>         mixed_precision = bfloat16_policy,              # 위에서 만든 mixed precision 정책 적용
>         backward_prefetch = BackwardPrefetch.BACKWARD_PRE, # Back-propagation 작업 중 prefetch를 통해 속도를 높임, 메모리 사용량은 늘어남
>         sharding_strategy = ShardingStrategy.SHARD_GRAD_OP # Gradient와 optimizer만 shard로 나누어 속도를 높임, 메모리 사용량은 늘어남
>     )
>     optimizer = torch.optim.Adam(fsdp_model.parameters(), lr = 1e-2, betas = (0.9, 0.999), weight_decay = 1e-4)
>     scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size = 1, gamma = 0.5)
```



# Fully Sharded Data Parallel

- 간단한 FSDP 실습

- Cell 5-2 : Training loop 수정하기

```
> def training_loop(dataset, mini_batch_size, max_epoch, checkpoint_interval, accumulation_number = 1):
>     ...
>
>     if global_rank == 0:
>         excution_time, peak_VRAM_usage = get_cuda_performace_record()
>         ...
>
>     state_dict_option = StateDictOptions(
>         full_state_dict = True,
>         cpu_offload = True # Full state가 너무 커 GPU VRAM이 모자란 경우, RAM을 추가로 활용
>     )
>     model_state = get_model_state_dict(fsd_model, options = state_dict_option)
>
>     if global_rank == 0:
>         torch.save(model_state, 'model.pth')
>
>     print('Saved PyTorch ANN parameters to model.pth\n', end = '')
```

Have a nice day!

