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

한국항공대학교
Korea Aerospace University

KAU
AI융합대학

SW중심대학

PyTorch 코딩

고급 Training 기술

01

Pinned Memory

Pinned Memory

- 가상 메모리와 paging
  - 가상 메모리(virtual memory)
    - 물리적(physical)인 실제 RAM을 그대로 사용하는 것이 아니라 가상의 주소를 통해 생성한 가상 공간
  - Paging
    - OS가 RAM을 관리하는 기술 중 하나
    - 물리적으로는 연속되어 있지 않으나, page table이라는 가상 주소를 통해 가상으로 연속된 메모리 공간을 생성
      - 단편화(fragmentation)으로 인한 메모리 낭비를 막을 수 있음
    - RAM 공간이 부족한 경우, 사용하지 않을 것 같은 데이터를 RAM에서 보조 기억장치(SSD, HDD 등)에 임시로 이동
      - 현재 주로 실행되는 프로세스들이 우선적으로 RAM을 사용하도록 도와줌
  - Page fault
    - 프로세스가 읽으려는 데이터가 물리적인 실제 RAM에 존재하지 않을 때 발생
    - 보조 기억장치에 임시로 보관한 데이터를 RAM에 복원해야 하므로 시간이 오래 걸림

Pinned Memory

- Pinned memory
  - RAM에서 OS가 paging하지 않도록 고정(pinned)시킨 전용 공간
  - Pinned memory가 충분하면 이를 활용하는 프로세스의 실행 속도가 빨라질 수 있음
  - 단, pinned memory가 너무 크면 이를 활용하지 않는 다른 프로세스들이 느려질 수 있음

- PyTorch data loader와 pinned memory
  - Training 및 test 할 데이터를 보조 기억장치 및 네트워크 등으로부터 pinned memory 공간에만 복사
    - Page fault 없이 RAM의 데이터를 즉시 GPU의 VRAM에 복사할 수 있도록 함

  - Dataset이 크고 epoch 횟수가 많을수록 training 시간을 크게 줄일 수 있음

Pinned Memory

- Pinned memory 실습
  - Cell 3 : Data loader가 pin memory를 활용하도록 수정하기

```
> training_dataset = torchvision.datasets.FashionMNIST(root = 'data', train = True, download = True, transform = ToTensor())
> test_dataset     = torchvision.datasets.FashionMNIST(root = 'data', train = False, download = True, transform = ToTensor())

> mini_batch_size = 64

> training_data_loader = DataLoader(
>     training_dataset,
>     batch_size = mini_batch_size,
>     pin_memory = True,              # Pin memory 활용
>     shuffle = True
> )

> test_data_loader = DataLoader(
>     test_dataset,
>     batch_size = mini_batch_size
>     pin_memory = True               # Pin memory 활용
> )

> ...
```

Gradient Accumulation

02

Gradient Accumulation

- Mini-batch 크기
  - Mini-batch가 클수록, mini-batch 내 data correlation이 낮아져 training을 안정적으로 진행할 수 있음
  - Training이 안정적이면 learning rate 및 최대 epoch 횟수를 높일 수 있음
    - Learning rate를 높일 경우 training 속도를 높일 수 있음
    - 최대 epoch 횟수를 높일 경우 test 및 inference의 accuracy를 더 높일 수 있음
  - 하지만 mini-batch가 커질수록 VRAM 사용량이 늘어남
- VRAM 공간이 부족할 때에도 mini-batch 크기를 늘릴 수 없을까?
  - PyTorch loss의 backward()는 gradient를 계산할 뿐만 아니라, gradient를 PyTorch 내부 변수에 누적 평균하는 기능도 있음
  - 매 mini-batch마다 gradient를 0으로 초기화하지 않고, 여러 mini-batch들의 gradient를 계속 누적한다면?

Gradient Accumulation

- Gradient accumulation 실습
  - Cell 6 : Training epoch에 누적 평균 기능 추가하기

```
> def train(device, data_loader, model, optimizer, accumulation_number = 1):      # Argument로 합산 횟수인 accumulation_number 추가
>     total_loss = 0
>
>     model.train()
>     for mini_batch_index, (x, t) in enumerate(data_loader):                     # Python for 반복문의 enumerate를 통해 현재 mini-batch의 번호를 구할 수 있음
>         x = x.to(device)
>
>         y = model(x)
>         t = t.to(y.device)
>         loss = torch.nn.functional.cross_entropy(y, t)
>
>         loss.backward()                                                         # 기존 gradient와 새로 구한 gradient의 누적 평균값을 구함
>         if mini_batch_index % accumulation_number == 0:                         # accumulation_number 번 째 mini-batch마다 아래를 수행
>             optimizer.step()                                                    # accumulation_number 번 동안의 누적 평균 gradient로 parameter를 수정
>             optimizer.zero_grad()                                               # accumulation_number 번 째 mini-batch마다 gradient를 0으로 초기화
>
>         mini_batch_size = x.shape[0]
>         ...
```

Gradient Accumulation

- Gradient accumulation 실습
  - Cell 7 : Training loop에 누적 평균할 mini-batch 개수 추가하기

```
> ...

> max_epoch              = 10
> accumulation_number = 4    # Gradient를 누적 평균할 mini-batch 개수

> for t in range(max_epoch):
>     print(f'Epoch {t + 1 :>3d} / {max_epoch}')

>     training_loss = train(device, training_data_loader, model, optimizer, accumulation_number)
>     ...
```

03

Learning Rate Scheduler

Learning Rate Scheduler

- Training 단계에 따른 learning rate
  - Training 초기
    - 일반적으로 parameter가 global minima와 멀리 떨어져 있음
    - Parameter가 local minima를 향해 빠르게 이동하도록 learning rate가 큰 것이 좋음
    - False local minima에 빠지지 않도록 overshoot을 유도하기 위해 learning rate가 큰 것이 좋음
  - Training 후기
    - 일반적으로 parameter가 global minima와 가까이 있음
    - Parameter가 local minima 근처에서 진동하지 않도록 learning rate가 작은 것이 좋음
    - Global minima에서 빠져나가지 않도록 overshoot을 방지하기 위해 learning rate가 작은 것이 좋음
  - 이러한 이유로 인해 training 초기에는 learning rate를 크게, 후기에는 작게 설정하는 것이 일반적으로 좋음

Learning Rate Scheduler

- Learning rate scheduler
  - ANN training이 진행됨에 따라 learning rate를 수정
  - 초기 learning rate를 크게 설정해 training 속도를 높이면서 안정성도 높일 수 있음

  - Learning rate를 수정하는 방식에 따라 여러 종류의 scheduler가 있음
    - StepLR : n번의 epoch마다 learning rate에 1보다 작은 특정 값을 곱함
    - CosineAnnealingLR : Epoch 횟수가 증가함에 따라 특정 주기에 맞추어 learning rate가 줄었다 늘었다를 반복함
    - ReduceLROnPlateau : Loss가 줄어들지 않는 경우 learning rate를 줄임
    - 기타 등등

Learning Rate Scheduler

- Learning rate scheduler 실습
  - Cell 4 : Learning rate scheduler 인스턴스 생성하기

```
> model = NeuralNetwork().to(device)

> optimizer = torch.optim.Adam(
>     model.parameters(),
>     lr = 5e-3, # Learning rate scheduler를 통해 learning rate를 조정하므로, 초기 learning rate를 높여도 training이 덜 불안정해짐
>     betas = (0.9, 0.999),
>     weight_decay = 1e-4
> )

> scheduler = torch.optim.lr_scheduler.StepLR( # StepLR scheduler instance 생성
>     optimizer,          # 연결할 optimizer 지정
>     step_size = 1,      # step_size번의 epoch마다 아래 아규먼트인 gamma를 현재 learning rate에 곱함
>     gamma = 0.5
> )
```

Learning Rate Scheduler

- Learning rate scheduler 실습
  - Cell 7 : Training loop에 learning rate scheduler 적용하기

```
> ...

> max_epoch               = 10
> accumulation_number = 4

> for t in range(max_epoch):
>     print(f'Epoch {t + 1 :>3d} / {max_epoch}')

>     training_loss = train(device, training_data_loader, model, optimizer, accumulation_number)
>     scheduler.step() # StepLR scheduler에게 한 epoch가 지났음을 알려줌

>     print('  Training progress')
>     ...
```

Gradient Clipping

04

Gradient Clipping

- **Gradient의 절대값**
  - Learning rate가 너무 크거나 input이 normalize되지 않은 경우 gradient의 절대값이 지나치게 커질 수 있음
  - Gradient의 절대값이 너무 큰 경우 parameter가 수렴하지 않고 계속 진동할 수 있음
  - Gradient의 절대값이 32bit float로 표현할 수 없을 정도로 커지는 경우 gradient explosion 발생
- **Gradient clipping**
  - Gradient의 절대값이 일정 이상 커지지 않도록 값을 잘라냄(clipping)
  - Training의 속도는 다소 느려질 수 있으나 안정성을 크게 높임
  - Gradient clipping의 종류
    - Gradient value clipping : if $|\nabla\theta_i| > c$, then $|\nabla\theta_i| = c$
    - Gradient normal clipping : if $||\nabla\theta|| > c$, then $\nabla\theta_i = \nabla\theta_i \times (c / ||\nabla\theta||)$

Gradient Clipping

- Gradient clipping 실습
  - Cell 6 : Training epoch에 gradient clipping 기능 추가하기

```
> def train(device, data_loader, model, optimizer, accumulation_number = 1):
>     total_loss = 0
>
>     model.train()
>     for mini_batch_index, (x, t) in enumerate(data_loader):
>         ...
>         loss = torch.nn.functional.cross_entropy(y, t)
>
>         loss.backward()
>         if mini_batch_index % accumulation_number == 0:
>             torch.nn.utils.clip_grad_norm_(model.parameters(), 1e-1) # Gradient normal clipping 방식 활용, c = 0.1
>             optimizer.step()
>             optimizer.zero_grad()
>
>         mini_batch_size = x.shape[0]
>         ...
```

05

Automatic Mixed Precision

Automatic Mixed Precision

- Mixed precision
  - 실수 data type이 표현할 수 있는 소수점 아래 정밀도를 precision이라 함
  - PyTorch는 parameter를 기본적으로 32bit float를 활용하지만, 일부 연산의 경우 16bit float를 활용해도 됨
  - ANN architecture에 따라 parameter의 일부는 32bit로, 일부는 16bit로 저장해도 최종 accuracy에 큰 문제가 없음
  - 다양한 precision의 parameter를 함께 사용하는 것을 mixed precision이라 함

- Automatic mixed precision (AMP)
  - PyTorch가 각 layer의 연산에 따라 16bit parameter를 사용해도 되는 경우 32bit 대신 16bit를 사용하도록 자동으로 변환

  - Parameter의 일부가 32bit에서 16bit로 변환됨에 따라 VRAM 사용량이 다소 줄어들 수 있음
    - PyTorch 2.4에서는 linear layer가 많을수록 VRAM 사용량이 오히려 늘어나는 경향을 보임

  - 16bit로 변환된 parameter의 경우 gradient도 16bit로 변환(scaling)해야 하므로 back-propagation 속도는 다소 느려짐
  - 다만 feed-forward 속도는 다소 빨라지기 때문에 inference 작업에는 유용함

Automatic Mixed Precision

- Automatic mixed precision (AMP) 실습
  - Cell 1 : AMP 라이브러리 불러오기

```
> import torch
> print(f'PyTorch version: {torch.__version__}')

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

> import torchvision
> from torchvision.transforms import ToTensor
```

Automatic Mixed Precision

- Automatic mixed precision (AMP) 실습
  - Cell 4 : ANN feed-forward 작업에 AMP 적용하기

```
> class NeuralNetwork(nn.Module):
>     ...
>
>     @autocast(device_type = device) # Decorator를 통해 forward()에 AMP 적용, back-propagation에도 자동으로 적용됨
>     def forward(self, x):
>         ...
```

Automatic Mixed Precision

- Automatic mixed precision (AMP) 실습
  - Cell 6 : Training epoch에 gradient scale 기능 추가하기

```
> def train(device, data_loader, model, optimizer, accumulation_number = 1):
>     total_loss = 0
>
>     model.train()
>     scaler = GradScaler()                                  # Back-propagation에도 AMP를 적용하기 위해 gradient scaler 인스턴스 생성
>     for mini_batch_index, (x, t) in enumerate(data_loader):
>         ...
>
>         loss = torch.nn.functional.cross_entropy(y, t)
>
>         scaler.scale(loss).backward()                      # Gradient scaler를 적용하여 gradient를 계산
>         if mini_batch_index % accumulation_number == 0:
>             scaler.unscale_(optimizer)                     # Gradient clipping을 위해 gradient의 precision을 복원
>             torch.nn.utils.clip_grad_norm_(model.parameters(), 1e-1)
>             scaler.step(optimizer)                         # Gradient scaler와 optimizer를 통해 ANN parameter 수정
>             scaler.update()                                # Gradient scaler를 다음 mini-batch를 위해 준비
>             optimizer.zero_grad()
>
>         mini_batch_size = x.shape[0]
>         ...
```

06

Checkpoint

Checkpoint

- Checkpoint
  - Training 과정 중간 상태를 통째로 저장하는 것을 checkpoint라 함
  - Checkpoint를 통해 training을 중지했다가, 해당 시점부터 training을 다시 수행할 수 있음
  - 하드웨어 및 네트워크 문제 등으로 training이 강제 중단된 경우, training을 중간부터 다시 시작하는데 매우 유용함

  - Checkpoint를 저장하기 위해선 ANN parameter뿐만 아니라 optimizer와 scheduler 등도 저장할 수 있어야 함

Checkpoint

- Checkpoint 실습
  - Cell 5 : Checkpoint 저장 및 불러오기 작업 정의하기

```
> def save_checkpoint(model, optimizer, scheduler):
>     checkpoint = {                                          # Checkpoint를 만들기 위해 dictionary type 변수 생성
>         'model_state'       : model.state_dict(),           # ANN parameter를 dictionary 형태로 변환하여 추출
>         'optimizer_state'   : optimizer.state_dict(),       # Optimizer 내부의 값들을 dictionary 형태로 변환하여 추출
>         'scheduler_state'   : scheduler.state_dict()        # Scheduler 내부의 값들을 dictionary 형태로 변환하여 추출
>     }

>     torch.save(checkpoint, 'checkpoint.pth')                # Checkpoint를 파일로 저장

> def load_checkpoint(model, optimizer, scheduler):
>     checkpoint = torch.load('checkpoint.pth', weights_only = False)   # Checkpoint에 저장된 모든 내용을 불러옴

>     model.load_state_dict(checkpoint['model_state'])        # Checkpoint를 통해 ANN parameter 복원
>     optimizer.load_state_dict(checkpoint['optimizer_state'])  # Checkpoint를 통해 optimizer 복원
>     scheduler.load_state_dict(checkpoint['scheduler_state'])  # Checkpoint를 통해 scheduler 복원
```

Checkpoint

- Checkpoint 실습
  - Cell 7-1 : Training loop에 checkpoint 저장 기능 추가하기

```
max_epoch               = 10
accumulation_number = 4
checkpoint_interval     = 5                                   # Checkpoint를 저장할 epoch 간격

for t in range(max_epoch):
    ...

    test_loss, test_accuracy = test(device, test_data_loader, model)
    ...

    if (t + 1) % checkpoint_interval == 0 and (t + 1) != max_epoch:   # 특정 epoch 횟수가 지날 때마다 아래를 수행
        save_checkpoint(model, optimizer, scheduler)                  # Checkpoint 저장 함수 호출
        print(f'Saved training checkpoint at {t + 1} epochs to "checkpoint.pth"')

excution_time, peak_VRAM_usage = get_cuda_performace_record()
...
```

Checkpoint

- Checkpoint 실습
  - Cell 7-2 : Training loop에 checkpoint 불러오기 기능 추가하기

```
current_epoch           = 0                                   # 현재 epoch 횟수
max_epoch               = 10
accumulation_number     = 4
checkpoint_interval     = 5

import os
if os.path.exists('checkpoint.pth'):                          # Checkpoint 파일이 있는지 검사
    load_checkpoint(model, optimizer, scheduler)              # Checkpoint 불러오기 함수 호출
    current_epoch = scheduler.last_epoch                      # 복원한 scheduler를 통해 epoch 횟수 복원

    print(f'Resuming training from checkpoint at epoch {current_epoch + 1}₩n')

for t in range(current_epoch, max_epoch):                     # 첫 epoch 또는 복원된 epoch부터 training loop 시작
    ...
```

07

Compile

Compile

- Compile
  - Python 언어로 작성된 PyTorch 코드를 TorchDynamo를 통해 최적화함
  - 처음에는 compile 작업을 위해 속도가 다소 느리지만, 이후 실행 속도가 빨라짐
  - 특정 함수나 특정 ANN만을 compile할 수도 있음
    - 본 강의에서는 ANN compile만 다룸

- Compile 기능을 활용하기 위한 조건
  - PyTorch 2.0 이상
  - CUDA 9.0 이상
  - C++ 언어 compiler

  - H100, A100, V100 이상의 GPU로 실행할 때 효과가 좋음

Checkpoint

- Checkpoint 실습
  - Cell 4 : Training을 위한 ANN 인스턴스를 compiler와 연결하기

```
> model = NeuralNetwork().to(device)

> model = torch.compile(              # ANN 인스턴스를 compile
>     model,                          # Compile할 ANN 인스턴스
>     mode = 'reduce-overhead'        # Reduce-overhead mode는, python으로 CUDA를 활용할 때 발생하는 overhead를 최대한 줄여줌
> )

> optimizer = torch.optim.Adam(model.parameters(), lr = 5e-3, betas = (0.9, 0.999), weight_decay = 1e-4)
> scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size = 1, gamma = 0.5)
```

Checkpoint

- Checkpoint 실습
  - Cell 8 : inference를 위한 ANN 인스턴스를 compiler와 연결하기

```
> model = NeuralNetwork().to(device)

> model = torch.compile(              # Compile했던 ANN으로 training한 weight를 불러오려면, inference ANN도 compile 해야 함
>     model,
>     mode = 'reduce-overhead'        # Compile mode는 training 때와 달라도 상관 없음
> )

> model.load_state_dict(torch.load('model.pth', weights_only = True))
```

Have a nice day!

Fin.