Ch 4. Bottlenecks and Solutions

On-device memory bottlenecks: recomputation and quantization

Seohyeong Jeong (June 13, 2023)

Reducing the memory footprint

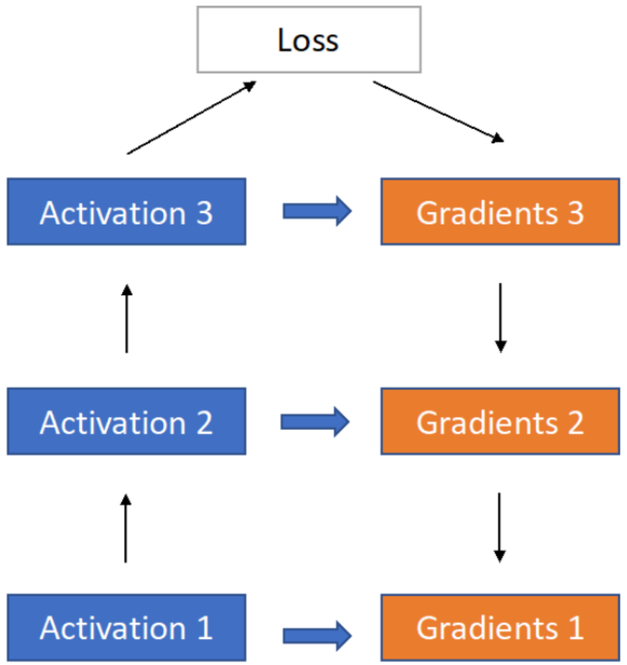
- Two main methods: recomputation and quantization
  - Recomputation: 사용하지 않는 동안 tensor를 메모리에서 삭제하고, 필요할 때 다시 연산 (recompute) 하여 사용하여 memory footprint를 줄이는 방식
  - Quantization: 모델 파라미터를 lower bit으로 표현하여 연산 속도와 memory footprint를 줄이는 방식

	Recomputation	Quantization
Lossy/lossless	Lossless	Lossy
Reducing memory footprint	Yes	Yes
Computation overhead	Medium	Low

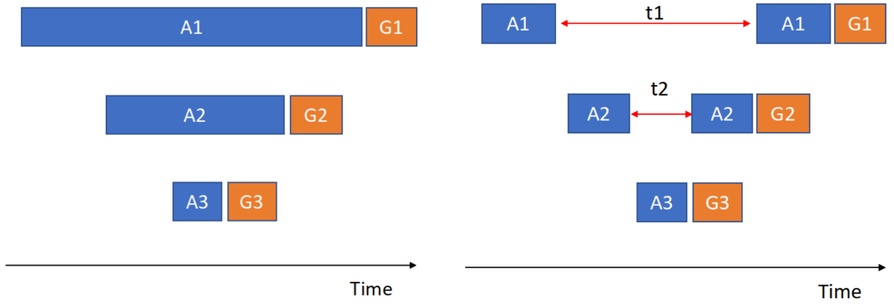
Figure 4.16 – A comparison of the two methods for reducing memory footprints

Recomputation

- Three-layer DNN의 예시



- Forward: activation 1 -> activation 2 -> activation 3 순서로 연산됨
- Backward: gradients 3 -> gradients 2 -> gradients 1 순서로 연산됨. 각각의 gradients을 구하기 위해서는 해당 layer에서의 activation 값이 필요함.
- 연산 순서에 따라 activation 값과 gradients 값의 memory usage



A1은 G1을 연산하기 전까지, A2는 G2를 연산하기 전까지 필요 없음.

A1 연산 후 A1을 메모리에서 지우고 G1 연산 직전 (G2가 연산되는 시점) 다시 연산하여 사용하면 memory footprint를 줄일 수 있다.

- 단점: computation overhead (calculate ~2x forward propagation)
- PyTorch activation checkpointing: https://pytorch.org/docs/stable/checkpoint.html

Quantization

- 일반적으로 fp32 (4 bytes, 32bits)으로 학습되는 model의 parameter를 half-precision (fp16)이나 int8등의 lower bits으로 표현하는 방식
- NVIDIA AMP (Automatic Mixed Precision)
  - Tensorflow 예시

```
optimizer = tf.keras.optimizers.SGD()
opt = tf.train.experimental.enable_mixed_precision_graph_
rewrite(optimizer)
```

  - Full-precision을 사용할 것인지, fp16 포멧을 사용할 것인지 자동으로 결정
- 단점
  - Since loosely optimization, can cause performance degradation
  - Can cause a model training to converge to a worse local minimum

NVIDIA AMP

Deprecated. Use PyTorch AMP

`apex.amp` is a tool to enable mixed precision training by changing only 3 lines of your script. Users can easily experiment with different pure and mixed precision training modes by supplying different flags to `amp.initialize`.

MIXED PRECISION IN PRACTICE: ACCURACY

Same accuracy as FP32, with no hyperparameter changes

Model	FP32	Mixed Precision*
AlexNet**	56.77%	56.93%
VGG-D	65.40%	65.43%
GoogLeNet (Inception v1)	68.33%	68.43%
Inception v2	70.03%	70.02%
Inception v3	73.85%	74.13%
Resnet50	75.92%	76.04%
BERT Fine-Tuning†	91.18%	91.24%

* Same hyperparameters and learning rate schedule as FP32.
** Sharan Narang, Paulius Micikevicius *et al.*, "Mixed Precision Training", ICLR 2018
† https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT

EXAMPLE

```
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
```

Ref: file:///Users/1113618/Downloads/devcon_2019_mcarilli_final.pdf

PyTorch AMP

```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training.
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast(device_type='cuda', dtype=torch.float16):
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.
        scaler.scale(loss).backward()

        # Unscale
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

- torch.autocast
  - autocast가 선언된 코드 영역에서 mixed precision 연산이 진행 됨 (FP16 or FP32)
  - Forward pass에서만 선언. Backward pass는 forward pass에서 선택된 data type으로 맞춰져 실행 됨
- torch.cuda.amp.GradScaler
  - FP16으로 gradient를 저장할 때 underflow 현상이 발생 (0.00001->0)
  - 이를 해결하기 위해 backward pass 때 gradient을 scaling하여 underflow 방지
  - Weight update시에는 scale factor만큼 나눠줌 (unscale)

PyTorch AMP

```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training.
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast(device_type='cuda', dtype=torch.float16):
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.
        scaler.scale(loss).backward()

        # Unscale
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

- torch.cuda.amp.GradScaler
  - Scale factor가 너무 클 경우 gradient를 FP16으로 표현할 수 있는 수보다 크게 scale 할 수 있음 (overflow)
  - Overflow 해결 방법
    - Scaled gradients가 inf, NaN이 되면 step() 함수 skip. 해당 gradients는 weight update에 사용되지 않음
    - Scaled factor 조정: update() 함수를 통해 더 작은 수로 교체. 미리 정해둔 backoff_factor 이용