Appendix: Automatic Mixed Precision (AMP)

Ki Hyun Kim

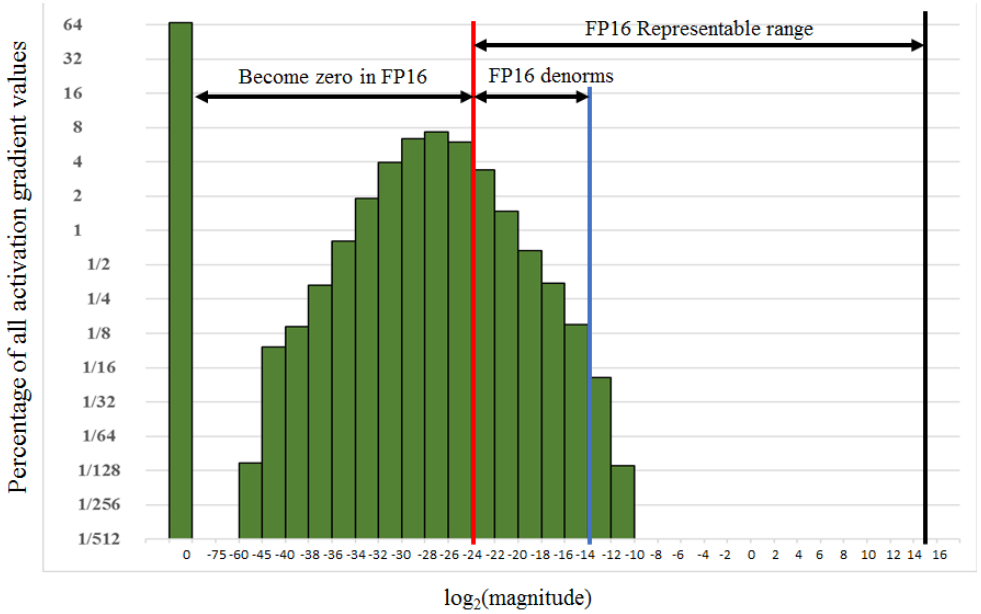
nlp.with.deep.learning@gmail.com

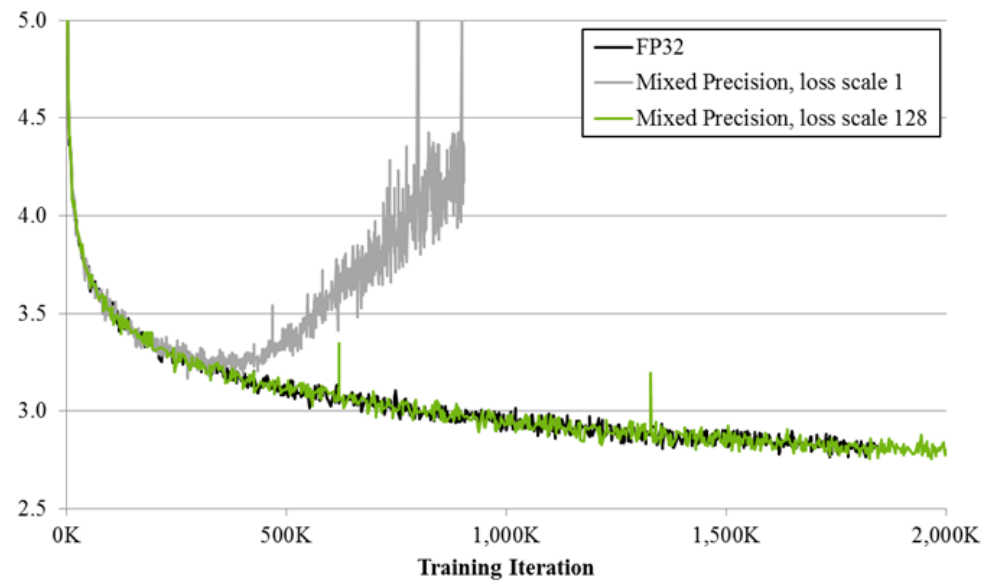
그림 출처: https://developer.nvidia.com/blog/mixed-precision-training-deep-neural-networks/

Motivations

- GPU의 한계로 인한 학습의 비효율성을 해소할 수 있으면 좋을 것
  - 메모리: 더 큰 모델로 더 큰 배치사이즈로 학습
  - 연산 속도: Float Point 16(half-precision)으로 연산할 경우 속도 증가

- 하지만 FP16의 경우 표현 범위의 한계가 있어, 너무 작은 값의 경우 0으로 표현됨
  - 너무 작은 값으로 나눌 경우 NaN으로 값이 반환됨
  - 마찬가지로 너무 큰 값의 경우에는 inf로 표현됨
  - FP16으로 학습할 경우, 정상적으로 학습이 이루어지지 않음





Mixed Precision Training [Narang and Micikevicius et al., 2018]

- 필요에 따라 scaling을 통해 FP16에 표현 가능한 범위 내에서 연산을 수행하자

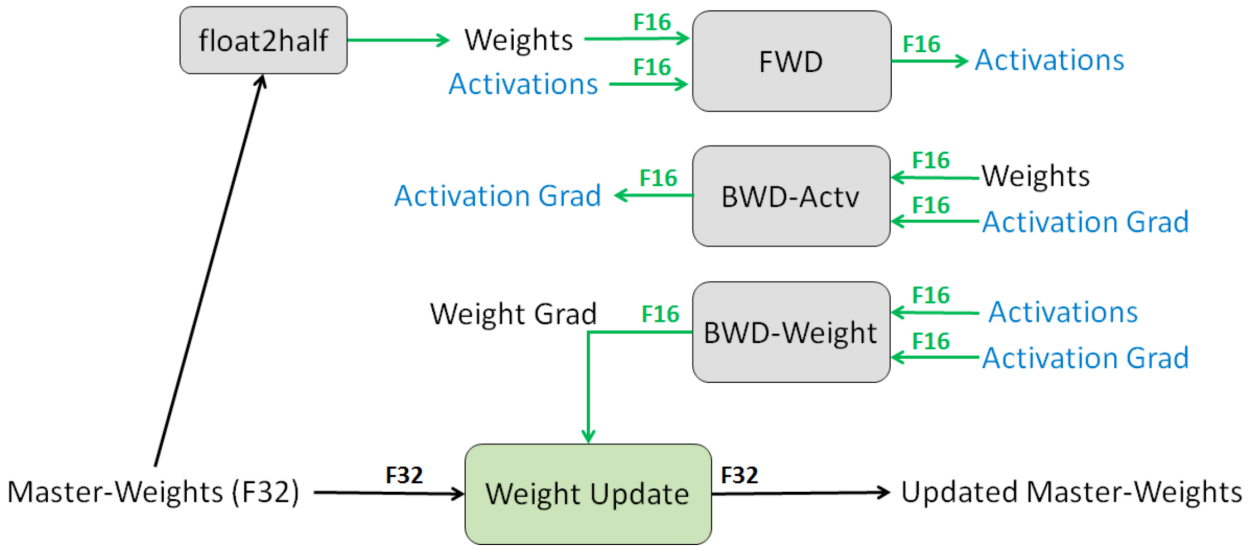
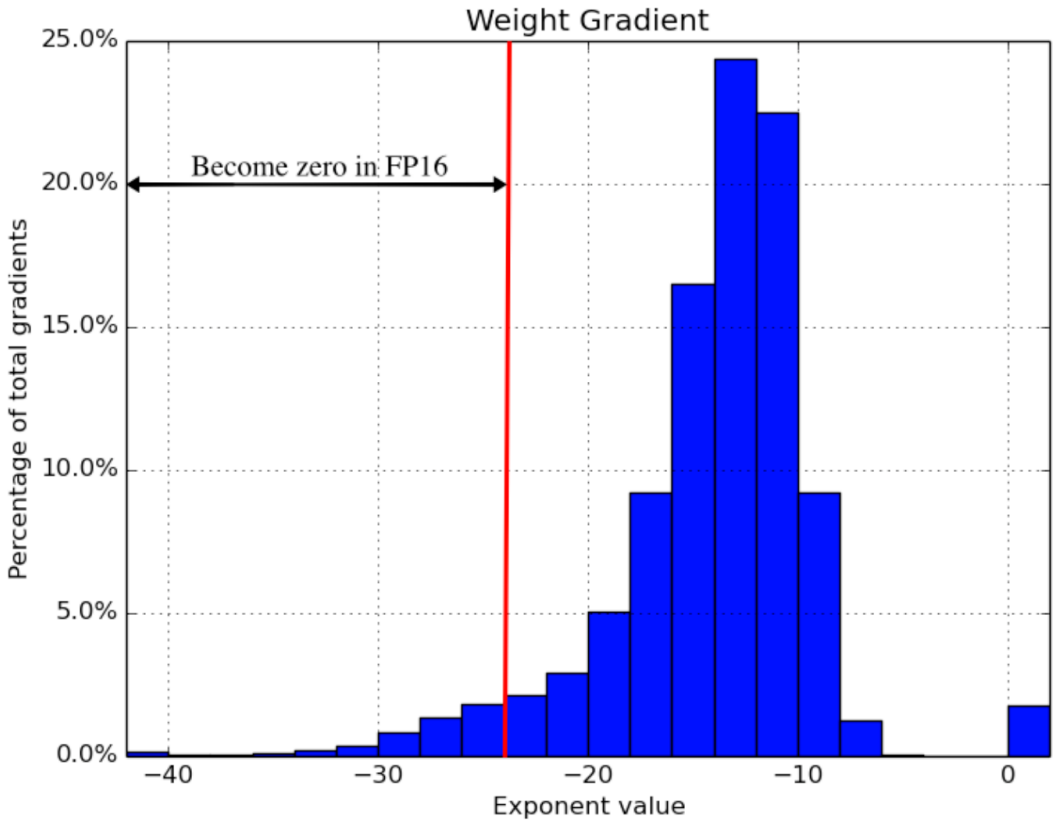


Figure 1: Mixed precision training iteration for a layer.



Mixed Precision Training [Narang and Micikevicius et al., 2018]

- 속도는 더 빠르면서, 성능은 비슷하거나 뛰어난 결과

Table 1: ILSVRC12 classification top-1 accuracy.

Model	Baseline	Mixed Precision	Reference
AlexNet	56.77%	56.93%	(Krizhevsky et al., 2012)
VGG-D	65.40%	65.43%	(Simonyan and Zisserman, 2014)
GoogLeNet (Inception v1)	68.33%	68.43%	(Szegedy et al., 2015)
Inception v2	70.03%	70.02%	(Ioffe and Szegedy, 2015)
Inception v3	73.85%	74.13%	(Szegedy et al., 2016)
Resnet50	75.92%	76.04%	(He et al., 2016b)

Table 3: Character Error Rate (CER) using mixed precision training for speech recognition. English results are reported on the WSJ '92 test set. Mandarin results are reported on our internal test set.

Model/Dataset	Baseline	Mixed Precision
English	2.20	1.99
Mandarin	15.82	15.01

Automatic Mixed Precision (AMP) at PyTorch

- 참고: https://pytorch.org/docs/stable/notes/amp_examples.html

```
# 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():
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
        # Backward passes under autocast are not recommended.
        # Backward ops run in the same dtype autocast chose for corresponding forward ops.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # If these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped.
        scaler.step(optimizer)

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

Summary

- AMP를 통해 FP16으로 연산을 수행할 수 있다.
  - 이를 통해 속도 증가와 메모리 사용량 감소를 기대할 수 있음
  - 몇 가지 추가적인 예외 처리(e.g. Nan, Inf 처리 및 CPU 연산처리)가 필요할 수 있음