Mixed Precision Training

발표날짜	@2024년 2월 28일
분야	Quantization
이해도	
발표자	이민지

1. Introduction

Float Point Format

- 고정 소수점 방식
  - 정수부와 소수부를 담을 비트의 수를 고정해서 사용하는 방식
  - 정확하고 연산이 빠르지만 표현 가능한 범위가 좁음
- 부동 소수점 방식
  - 가수부(exponent)와 지수부(fraction/mantissa)를 따로 저장하는 방식
  - ex) -314.625를 FP32로 표현?
    1. (-) 이므로 부호비트 1
    2. 절대값 314.625 = 100111010.101 ⇒ 1.00111010101 X 2^8
    3. 지수 + bias ⇒ 2^8-1=127 ⇒ 127+8 = 135 = 10000111
    4. 1/10000111/00111010101000000000000

Mixed Precision

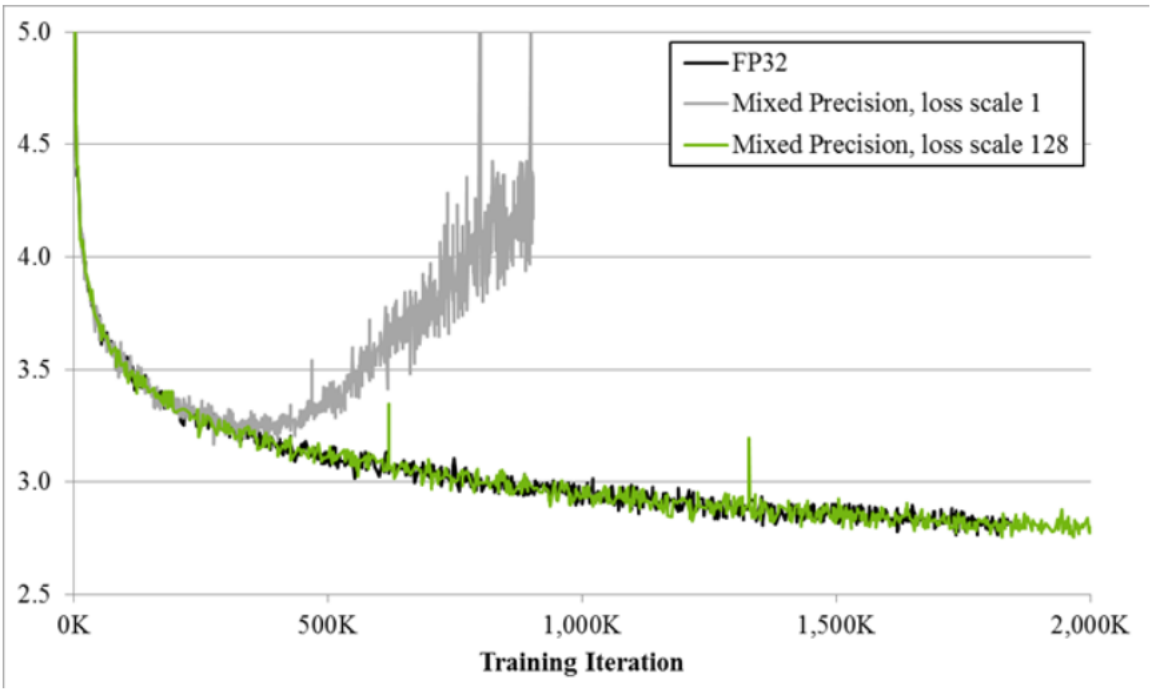


Figure 1. Training curves for the bigLSTM English language model show the benefits of the mixed-precision training techniques described in this post. The Y-axis is training loss. Mixed precision without loss scaling (grey) diverges after a while, whereas mixed precision with loss scaling (green) matches the single precision model (black).

FP32를 이용해 학습시킨 결과(초록색)에 비해 FP16(검정색)을 사용하였을 때 수렴하지 못하고 다시 발산

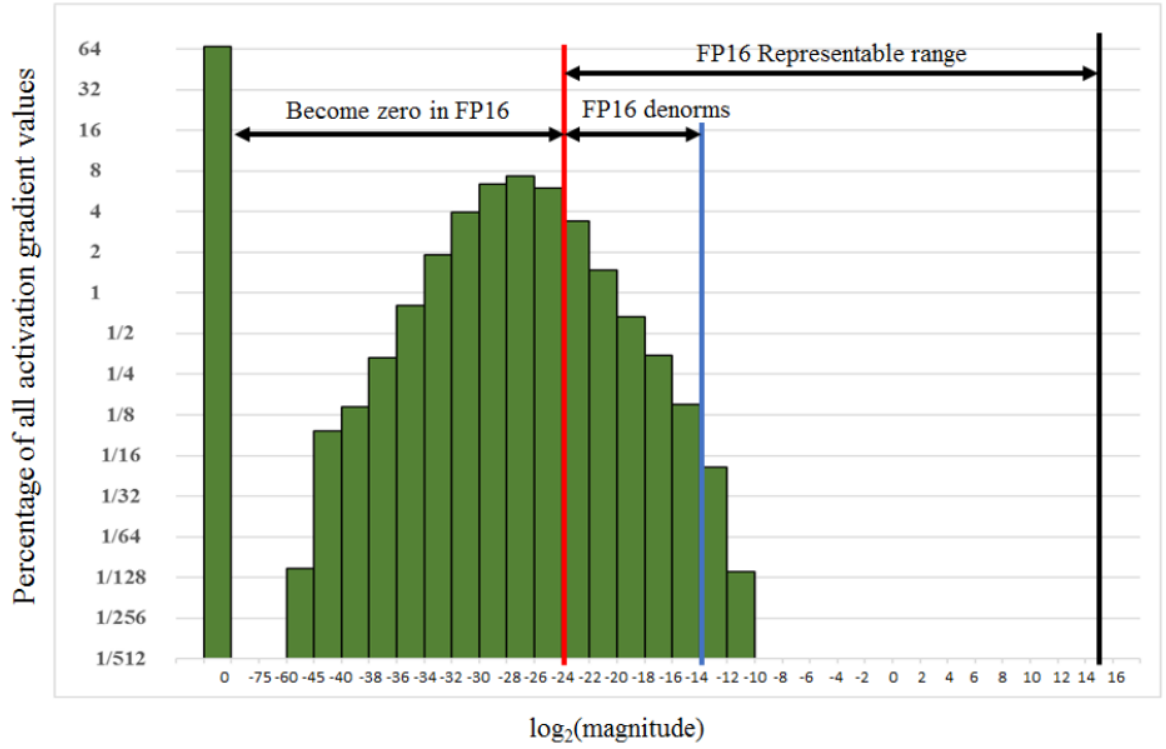


Figure 2. A histogram of activation gradients recorded when training the Multibox SSD detector network in single precision. The Y-axis is the percentage of all values on a log scale. The X-axis is the log scale of absolute values, as well as a special entry for zeros. For example, in this training session 66.8% of values were zero, whereas 4% of values were between 2^{-32} and 2^{-30}.

실제로 FP32를 이용해 네트워크를 학습시키고, Gradient의 분포를 보면 대부분의 값이 FP16으로 표현 불가능한 정밀도의 값을 가지고 있음을 알 수 있음

⇒ FP32+FP16 = Mixed Precision

Implementation

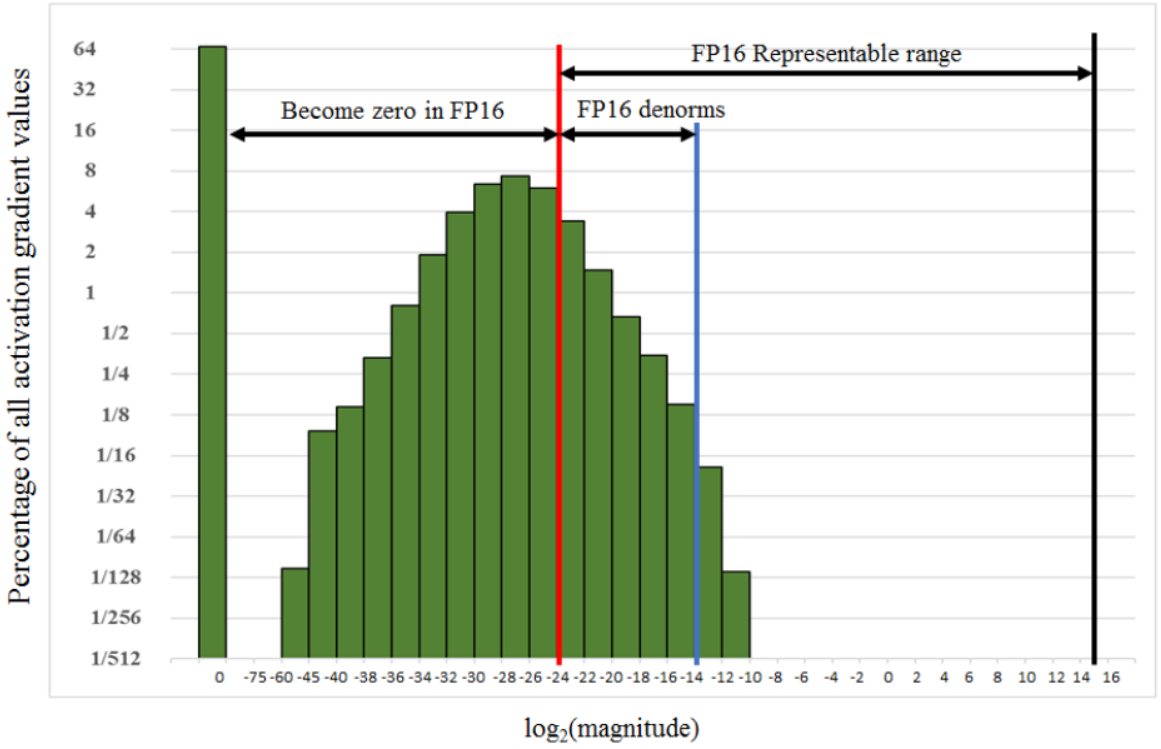
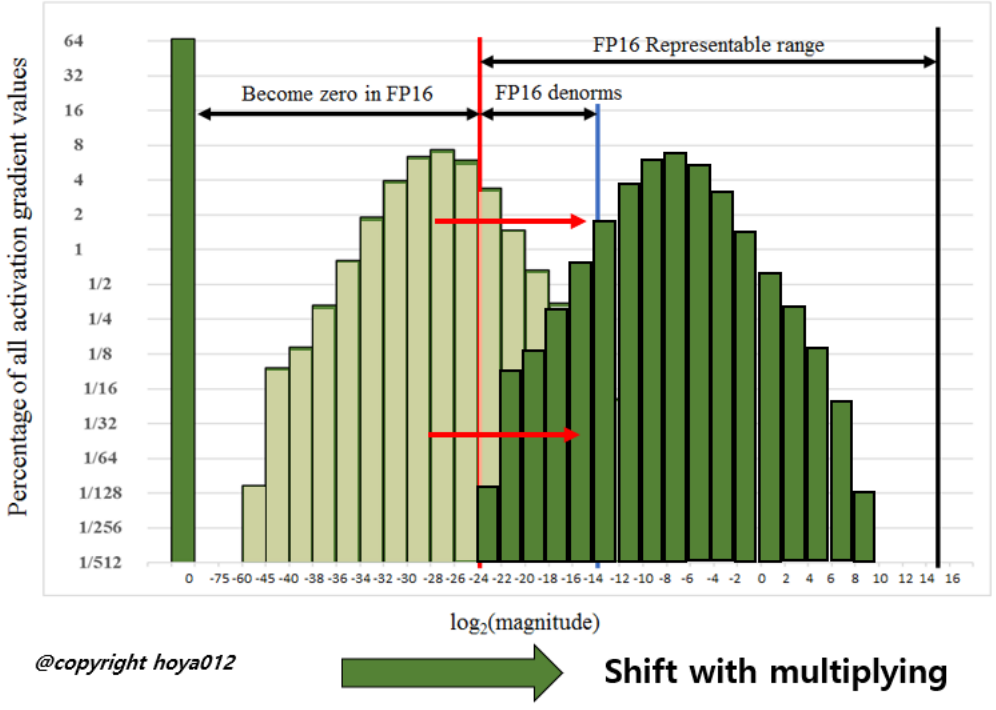
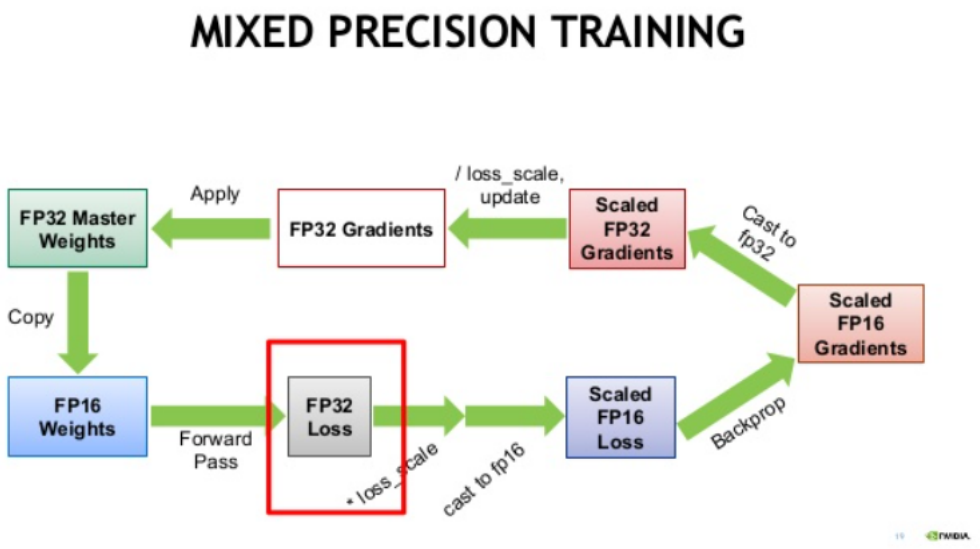


Figure 2. A histogram of activation gradients recorded when training the Multibox SSD detector network in single precision. The Y-axis is the percentage of all values on a log scale. The X-axis is the log scale of absolute values, as well as a special entry for zeros. For example, in this training session 66.8% of values were zero, whereas 4% of values were between 2^{-32} and 2^{-30}.

FP16의 표현 범위 바깥에 있는 Gradient를 캐스팅 가능한 범위 로 이동시킨다 ⇒ 모든 값에 scailing facter를 곱해주어 표현 가능한 범위 안으로 당겨온다!



- scaling factor S의 크기 정하는 방법
  - 경험적으로 구하는 경우
  - 기울기 통계를 알 수 있을 경우 최대값이 FP16의 최대표현인 65,504에 맞출 수 있도록 값을 지정
  - NVIDIA가 사용한 방법 : 반복횟수 N 동안 오버플로우가 발생하지 않으면 S를 늘림, 오버플로우가 발생하면 S를 낮춤 → N=2000번을 수행하여 S=2로 시작해서 S=0.5로 마무리



Step 1. FP32 weight에 대한 FP16 copy weight을 만든다.

(이 FP16 copy weight은 forward pass, backward pass에 이용된다.)

Step 2. FP16 copy weight을 이용해 forward pass를 진행한다.

Step 3. forward pass로 계산된 FP16 prediction 값을 FP32로 casting한다.

Step 4. FP32 prediction을 이용해 FP32 loss를 계산하고, 여기에 scaling factor S를 곱한다.

Step 5. scaled FP32 loss를 FP16으로 casting한다.

Step 6. scaled FP16 loss를 이용하여 backward propagation을 진행하고, gradient를 계산한다.

Step 7. FP16 gradient를 FP32로 casting한고, 이를 scaling factor S로 다시 나눈다.

(chain rule에 의해 모든 gradient는 같은 크기로 scaling된 상태임)

Step 8. FP32 gradient를 이용해 FP32 weight를 update한다.

> 💡 FP32는 계속 저장해 두고, FP16 copy weight를 만들어 forward/backward에 사용한다. FP16 copy weight로 얻은 gradient들을 이용해 FP32 weight를 update 한다.

Experiment & Result

CNN for classification

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)

Detection CNNs

Table 2: Detection network average mean precision.

Model	Baseline	MP without loss-scale	MP with loss-scale
Faster R-CNN	69.1%	68.6%	69.7%
Multibox SSD	76.9%	diverges	77.1%

메모리와 training 시간 변화도 보고 싶어서 git에서 찾아봄

> https://github.com/suvojit-0x55aa/mixed-precision-pytorch

▼ 코드 전문

```
'''Train CIFAR10 with PyTorch.'''
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn

import torchvision
import torchvision.transforms as transforms

import os
import argparse
import time
from loguru import logger

from models import *
from utils import progress_bar


parser = argparse.ArgumentParser(description='PyTorch CIFAR10 Training')
parser.add_argument('--lr', default=0.1, type=float, help='learning rate')
parser.add_argument('--mp', action='store_true', help='use mixed precision')

args = parser.parse_args()

torch.backends.cudnn.deterministic = False

device = 'cuda' if torch.cuda.is_available() else 'cpu'
best_acc = 0  # best test accuracy
start_epoch = 0  # start from epoch 0 or last checkpoint epoch

# Data
print('==> Preparing data..')
```

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=128, shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=100, shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

# Model
print('==> Building model..')
# net = VGG('VGG19')
# net = ResNet18()
net = ResNet50()
net = net.to(device)
if device == 'cuda':
   net = torch.nn.DataParallel(net)
   cudnn.benchmark = True


criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=args.lr,
                      momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)

if args.mp:
    scaler = torch.cuda.amp.GradScaler()

# Training
def train(epoch):
    print('\nEpoch: %d' % epoch)
    net.train()
    train_loss = 0
    correct = 0
    total = 0
    for batch_idx, (inputs, targets) in enumerate(trainloader):
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()

        if args.mp:
            with torch.cuda.amp.autocast():
                outputs = net(inputs)
```

```
                loss = criterion(outputs, targets)
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
        else:
            outputs = net(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()

        progress_bar(batch_idx, len(trainloader), 'Loss: %.3f | Acc: %.3f%% (%d/%d)'
                     % (train_loss/(batch_idx+1), 100.*correct/total, correct, total))



def test(epoch):
    global best_acc
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

            progress_bar(batch_idx, len(testloader), 'Loss: %.3f | Acc: %.3f%% (%d/%d)'
                         % (test_loss/(batch_idx+1), 100.*correct/total, correct, total))

    # Save checkpoint.
    acc = 100.*correct/total
    if acc > best_acc:
        print('Saving..')
        state = {
            'net': net.state_dict(),
            'acc': acc,
            'epoch': epoch,
        }
        if not os.path.isdir('checkpoint'):
            os.mkdir('checkpoint')
        torch.save(state, './checkpoint/ckpt.pth')
        best_acc = acc


for epoch in range(start_epoch, start_epoch+5):
    start = time.time()
    train(epoch)
```

```
    end = time.time()
    logger.info(f'[epoch {epoch}] time elapse: {end-start:.3f}')
    test(epoch)
    scheduler.step()
```

Results

Training on a single P100 Pascal GPU, I was able to obtain the following result, while training with ResNet50 with a batch size of 128 over 200 epochs.

	FP32	Mixed Precision
Time/Epoch	1m32s	1m15s
Storage	90 MB	46 MB
Accuracy	94.50%	94.43%

Training on 4x P100 Tesla GPUs, with ResNet50 with a batch size of 512 over 200 epochs.

	FP32	Mixed Precision
Time/Epoch	26s224ms	23s359ms
Storage	90 MB	46 MB
Accuracy	94.51%	94.78%

Training on a single V100 Volta GPUs, with ResNet50 with a batch size of 128 over 200 epochs.

	FP32	Mixed Precision
Time/Epoch	47s112ms	25s601ms
Storage	90 MB	46 MB
Accuracy	94.87%	94.65%

Training on 4x V100 Volta GPUs, with ResNet50 with a batch size of 512 over 200 epochs.

	FP32	Mixed Precision
Time/Epoch	17s841ms	12s833ms
Storage	90 MB	46 MB
Accuracy	94.38%	94.60%

일반적인 학습 코드

```
for batch_idx, (inputs, labels) in enumerate(data_loader):
  optimizer.zero_grad()

  outputs = model(inputs)
  loss = criterion(outputs, labels)
```

```
  loss.backward()
  optimizer.step()
```

AMP를 적용한 코드

```
""" define loss scaler for automatic mixed precision """
# Creates a GradScaler once at the beginning of training.
scaler = torch.cuda.amp.GradScaler()

for batch_idx, (inputs, labels) in enumerate(data_loader):
  optimizer.zero_grad()

  with torch.cuda.amp.autocast():
    # Casts operations to mixed precision
    outputs = model(inputs)
    loss = criterion(outputs, labels)

  # Scales the loss, and calls backward()
  # to create scaled gradients
  scaler.scale(loss).backward()

  # Unscales gradients and calls
  # or skips optimizer.step()
  scaler.step(self.optimizer)

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

torch.cuda.amp.GradScaler(*init_scale=65536.0*, *growth_factor=2.0*, *backoff_factor=0.5*, *growth_interval=2000*, *en*

Parameters

- **init_scale** (*float*, *optional*, *default=2.**16*) – Initial scale factor.
- **growth_factor** (*float*, *optional*, *default=2.0*) – Factor by which the scale is multiplied during `update()` if no inf/NaN gradients occur for `growth_interval` consecutive iterations.
- **backoff_factor** (*float*, *optional*, *default=0.5*) – Factor by which the scale is multiplied during `update()` if inf/NaN gradients occur in an iteration.
- **growth_interval** (*int*, *optional*, *default=2000*) – Number of consecutive iterations without inf/NaN gradients that must occur for the scale to be multiplied by `growth_factor`.
- **enabled** (*bool*, *optional*) – If `False`, disables gradient scaling. `step()` simply invokes the underlying `optimizer.step()`, and other methods become no-ops. Default: `True`

Pytorch에서 해당 함수를 찾아본 결과 nvidia가 했던 방식 처럼 초기값을 설정해놓고(디폴트는 2^16) 몇번 반복할지(N) 정하고 오버플로우가 발생하면 x0.5(기본), 발생하지 않으면 x2(기본)으로 설정됨

torch.cuda.amp.autocast(device_type, *enabled=True*, *dtype=torch.float16*, *cache_enabled=True*)

Parameters

- **device_type** (*str, required*) – Device type to use. Possible values are: 'cuda', 'cpu', 'xpu' and 'hpu'. The type is the same as the *type* attribute of a `torch.device`. Thus, you may obtain the device type of a tensor using *Tensor.device.type*.
- **enabled** (*bool, optional*) – Whether autocasting should be enabled in the region. Default: `True`
- **dtype** (*torch_dtype, optional*) – Whether to use torch.float16 or torch.bfloat16.
- **cache_enabled** (*bool, optional*) – Whether the weight cache inside autocast should be enabled. Default: `True`

사용할 type(cup, cuda 등)을 알려주고, torch.float16(IEEE표준)을 쓸지, torch.bfloat16(정밀도 더 떨어짐)을 쓸지 정해준다(선택사항)