PyTorch 节省显存的策略总结

CV开发者都爱看的 极市平台 2023-04-18 22:00:37 发表于广东 手机阅读

作者丨OpenMMLab

来源丨https://zhuanlan.zhihu.com/p/430123077

编辑丨极市平台

极市导读

随着深度学习快速发展，同时伴随着模型参数的爆炸式增长，对显卡的显存容量提出了越来越高的要求，如何在单卡小容量显卡上面训练模型是一直以来大家关心的问题。本文结合 MMCV 开源库对一些常用的节省显存策略进行了简要分析。 >>加入极市CV技术交流群，走在计算机视觉的最前沿

0 前言

本文涉及到的 PyTorch 节省显存的策略包括：

- 混合精度训练
- 大 batch 训练或者称为梯度累加
- gradient checkpointing 梯度检查点

1 混合精度训练

混合精度训练全称为 Automatic Mixed Precision，简称为 AMP，也就是我们常说的 FP16。在前系列解读中已经详细分析了 AMP 原理、源码实现以及 MMCV 中如何一行代码使用 AMP，具体链接见：

OpenMMLab：PyTorch 源码解读之 torch.cuda.amp: 自动混合精度详解

https://zhuanlan.zhihu.com/p/348554267

OpenMMLab：OpenMMLab 中混合精度训练 AMP 的正确打开方式

https://zhuanlan.zhihu.com/p/375224982

由于前面两篇文章已经分析的非常详细了，本文只简要描述原理和具体说明用法。

考虑到训练过程中梯度幅值大部分是非常小的，故训练默认是 FP32 格式，如果能直接以 FP16 格式精度进行训练，理论上可以减少一半的内存，达到加速训练和采用更大 batch size 的目的，但是直接以 FP16 训练会出现溢出问题，导致 NAN 或者参数更新失败问题，而 AMP 的出现就是为了解决这个问题，其核心思想是 **混合精度训练+动态损失放大**：

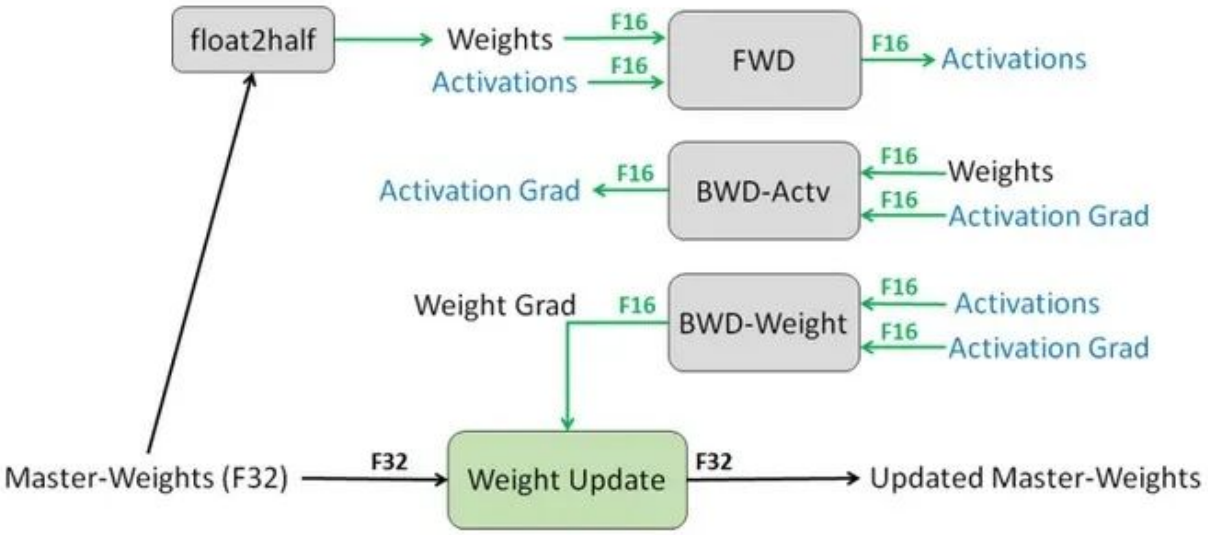


Figure 1: Mixed precision training iteration for a layer.

1. 维护一个 FP32 数值精度模型的副本
2. 在每个 iteration

- 拷贝并且转换成 FP16 模型
- 前向传播（FP16 的模型参数），此时 weights, activations 都是 FP16
- loss 乘 scale factor s
- 反向传播（FP16 的模型参数和参数梯度），此时 gradients 也是 FP16
- 参数梯度乘 1/s
- 利用 FP16 的梯度更新 FP32 的模型参数

在 MMCV 中使用 AMP 分成两种情况：

- 在 OpenMMLab 上游库例如 MMDetection 中使用 MMCV 的 AMP
- 用户只想简单调用 MMCV 中的 AMP，而不依赖上游库

(1) OpenMMLab 上游库如何使用 MMCV 的 AMP

以 MMDectection 为例，用法非常简单，只需要在配置中设置：

```
fp16 = dict(loss_scale=512.) # 表示静态 scale

# 表示动态 scale
fp16 = dict(loss_scale='dynamic')

# 通过字典形式灵活开启动态 scale
fp16 = dict(loss_scale=dict(init_scale=512.,mode='dynamic'))
```

三种不同设置在大部分模型上性能都非常接近，如果不想设置 loss_scale，则可以简单的采用 `loss_scale='dynamic'`

(2) 调用 MMCV 中的 AMP

直接调用 MMCV 中的 AMP，这通常意味着用户可能在其他库或者自己写的代码库中支持 AMP 功能。**需要特别强调的是 PyTorch 官方仅仅在 1.6 版本及其之后版本中开始支持 AMP，而 MMCV 中的 AMP 支持 1.3 及其之后版本。如果你想在 1.3 或者 1.5 中使用 AMP，那么使用 MMCV 是个非常不错的选择。**

使用 MMCV 的 AMP 功能，只需要遵循以下几个步骤即可：

1. 将 auto_fp16 装饰器应用到 model 的 forward 函数上
2. 设置模型的 fp16_enabled 为 True 表示开启 AMP 训练，否则不生效
3. 如果开启了 AMP，需要同时配置对应的 FP16 优化器配置 Fp16OptimizerHook
4. 在训练的不同时刻，调用 Fp16OptimizerHook，如果你同时使用了 MMCV 中的 Runner 模块，那么直接将第 3 步的参数输入到 Runner 中即可
5. (可选) 如果对应某些 OP 希望强制运行在 FP32 上，则可以在对应位置引入 force_fp32 装饰器

```
# 1 作用到 forward 函数中
class ExampleModule(nn.Module):

    @auto_fp16()
    def forward(self, x, y):
        return x, y

# 2 如果开启 AMP，则需要加入开启标志
model.fp16_enabled = True

# 3 配置 Fp16OptimizerHook
optimizer_config = Fp16OptimizerHook(
    **cfg.optimizer_config, **fp16_cfg, distributed=distributed)

# 4 传递给 runner
runner.register_training_hooks(cfg.lr_config, optimizer_config,
                               cfg.checkpoint_config, cfg.log_config,
                               cfg.get('momentum_config', None))

# 5 可选
class ExampleModule(nn.Module):

    @auto_fp16()
    def forward(self, x, y):
        features=self._forward(x, y)
        loss=self._loss(features,labels)
        return loss

    def _forward(self, x, y):
       pass


    @force_fp32(apply_to=('features',))
    def _loss(features,labels) :
        pass
```

注意 **force_fp32 要生效，依然需要 fp16_enabled 为 True 才生效。**

2 大 Batch 训练（梯度累加）

大 Batch 训练通常也称为梯度累加策略，通常 PyTorch 一次迭代训练流程为：

```
y_pred = model(xx)
loss = loss_fn(y_pred, y)
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

而梯度累加策略下常见的一次迭代训练流程为：

```
y_pred = model(xx)
loss = loss_fn(y_pred, y)

loss = loss / cumulative_iters
loss.backward()

if current_iter % cumulative_iters==0
    optimizer.step()
    optimizer.zero_grad()
```

其核心思想就是对前几次梯度进行累加，然后再统一进行参数更新，从而变相实现大 batch size 功能。需要注意的是如果模型中包括 BN 等考虑 batch 信息的层，那么性能可能会有轻微的差距。

细节可以参考：

https://github.com/open-mmlab/mmcv/pull/1221

在 MMCV 中已经实现了梯度累加功能，其核心代码位于 mmcv/runner/hooks/optimizer.py

GradientCumulativeOptimizerHook 中，和 AMP 实现一样是采用 Hook 实现的。使用方法和 AMP 类似，只需要将第一节中的 Fp16OptimizerHook 替换为 GradientCumulativeOptimizerHook 或者 GradientCumulativeFp16OptimizerHook 即可。其核心实现如下所示：

```
@HOOKS.register_module()
class GradientCumulativeOptimizerHook(OptimizerHook):
    def __init__(self, cumulative_iters=1, **kwargs):

        self.cumulative_iters = cumulative_iters
        self.divisible_iters = 0  # 剩余的可以被 cumulative_iters 整除的训练迭代次数
        self.remainder_iters = 0  # 剩余累加次数
        self.initialized = False

    def after_train_iter(self, runner):
        # 只需要运行一次即可
```

```
        if not self.initialized:
            self._init(runner)

        if runner.iter < self.divisible_iters:
            loss_factor = self.cumulative_iters
        else:
            loss_factor = self.remainder_iters

        loss = runner.outputs['loss']
        loss = loss / loss_factor
        loss.backward()

        if (self.every_n_iters(runner, self.cumulative_iters)
                or self.is_last_iter(runner)):

            runner.optimizer.step()
            runner.optimizer.zero_grad()


    def _init(self, runner):

        residual_iters = runner.max_iters - runner.iter

        self.divisible_iters = (
            residual_iters // self.cumulative_iters * self.cumulative_iters)
        self.remainder_iters = residual_iters - self.divisible_iters

        self.initialized = True
```

需要明白 divisible_iters 和 remainder_iters 的含义：

(1) 从头训练

此时在开始训练时 iter=0，一共迭代 max_iters=102 次，梯度累加次数是 4，由于 102 无法被 4 整除，也就是最后的 102-(102 // 4)*4=2 个迭代是额外需要考虑的，在最后 2 个训练迭代中 **loss_factor 不能除以 4，而是 2，这样才是最合理的做法。其中 remainder_iters=2，divisible_iters=100，residual_iters=102。**

(2) resume 训练

假设在梯度累加的中途退出，然后进行 resume 训练，此时 iter 不是 0，由于优化器对象需要重新初始化，为了保证剩余的不能被累加次数的训练迭代次数能够正常计算，需要重新计算 residual_iters。

3 梯度检查点

梯度检查点是一种用训练时间换取显存的办法，其核心原理是在反向传播时重新计算神经网络的中间激活值而不用在前向时存储，torch.utils.checkpoint 包中已经实现了对应功能。简要实现过程是：**在前向阶段传递到 checkpoint 中的 forward 函数会以 _torch.no_grad_ 模式运行，并且仅仅保存输入参数和 forward 函数，在反向阶段重新计算其 forward 输出值。**

具体用法非常简单，以 ResNet 的 BasicBlock 为例：

```
def forward(self, x):
    def _inner_forward(x):
        identity = x
        out = self.conv1(x)
        out = self.norm1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.norm2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        return out

    # x.requires_grad 这个判断很有必要
    if self.with_cp and x.requires_grad:
        out = cp.checkpoint(_inner_forward, x)
    else:
        out = _inner_forward(x)
    out = self.relu(out)
    return out
```

self.with_cp 为 True，表示要开启梯度检查点功能。

checkpoint 在用法上面需要注意以下几点：

1. **模型的第一层不能用 checkpoint 或者说 forward 输入中不能所有输入的 requires_grad 属性都是 False**，因为其内部实现是依靠输入的 requires_grad 属性来判断输出返回是否需要梯度，而通常模型第一层输入是 image tensor，其 requires_grad 通常是 False。一旦你第一层用了 checkpoint，那么意味着这个 forward 函数不会有任何梯度，也就是说**不会进行任何参数更新**，没有任何使用的必要，具体见 https://discuss.pytorch.org/t/use-of-torch-utils-checkpoint-checkpoint-causes-simple-model-to-diverge/116271。如果第一层用了 checkpoint， PyTorch 会打印 None of the inputs have requires_grad=True. Gradients will be Non 警告

2. **对于 dropout 这种 forward 存在随机性的层，需要保证 preserve_rng_state 为 True** (默认就是 True，所以不用担心)，一旦标志位设置为 True，在 forward 会存储 RNG 状态，然后在反向传播的时候读取该 RNG，保证两次 forward 输出一致。如果你确定不需要保存 RNG，则可以设置 preserve_rng_state 为 False，省掉一些不必要的运行逻辑
3. 其他注意事项，可以参考官方文档 https://pytorch.org/docs/stable/checkpoint.html#

其核心实现如下所示：

```
class CheckpointFunction(torch.autograd.Function):

    @staticmethod
    def forward(ctx, run_function, preserve_rng_state, *args):
        # 检查输入参数是否需要梯度
        check_backward_validity(args)
        # 保存必要的状态
        ctx.run_function = run_function
        ctx.save_for_backward(*args)
        with torch.no_grad():
            # 以 no_grad 模型运行一遍
            outputs = run_function(*args)
        return outputs

    @staticmethod
    def backward(ctx, *args):
        # 读取输入参数
        inputs = ctx.saved_tensors
        # Stash the surrounding rng state, and mimic the state that was
        # present at this time during forward.  Restore the surrounding state
        # when we're done.
        rng_devices = []
        with torch.random.fork_rng(devices=rng_devices, enabled=ctx.preserve_rng_state):
            # detach 掉当前不需要考虑的节点
            detached_inputs = detach_variable(inputs)
            # 重新运行一遍
            with torch.enable_grad():
                outputs = ctx.run_function(*detached_inputs)

        if isinstance(outputs, torch.Tensor):
            outputs = (outputs,)
        # 计算该子图梯度
        torch.autograd.backward(outputs, args)
```

```
        grads = tuple(inp.grad if isinstance(inp, torch.Tensor) else inp
                      for inp in detached_inputs)
        return (None, None) + grads
```

4 实验验证

为了验证上述策略是否真的能够省显存，采用 mmdetection 库进行验证，基本环境如下：

```
显卡: GeForce GTX 1660
PyTorch: 1.7.1
CUDA Runtime 10.1
MMCV: 1.3.16
MMDetection: 2.17.0
```

(1) base

- 数据集：**pascal voc**
- 算法是 **retinanet**，对应配置文件为 retinanet_r50_fpn_1x_voc0712.py
- 为了防止 lr 过大导致训练出现 nan，需要将 lr 设置为 0.01/8=0.00125
- bs 设置为 2

(2) 混合精度 AMP

在 base 配置基础上新增如下配置即可：

```
fp16 = dict(loss_scale=512.)
```

(3) 梯度累加

在 base 配置基础上替换 optimizer_config 为如下：

```
# 累加2次
optimizer_config = dict(type='GradientCumulativeOptimizerHook', cumulative_iters=2)
```

(4) 梯度检查点

在 base 配置基础上在 backbone 部分开启 with_cp 标志即可：

```
model = dict(backbone=dict(with_cp=True),
             bbox_head=dict(num_classes=20))
```

每个实验总共迭代 1300 次，统计占用显存、训练总时长。

配置	显存占用(MB)	训练时长
base	2900	7 分 45 秒
混合精度 AMP	2243	36 分
梯度累加	3177	7 分 32 秒
梯度检查点	2590	8 分 37 秒

1. 对比 base 和 AMP 可以发现，由于实验显卡是不支持 AMP 的，故只能节省显存，速度会特别慢，如果本身显卡支持 AMP 则可以实现在节省显存的同时提升训练速度
2. 对比 base 和梯度累加可以发现，在相同 bs 情况下，梯度累加 2 次相当于 bs 扩大一倍，但是显存增加不多。如果将 bs 缩小一倍，则可以实现在相同 bs 情况下节省大概一倍显存
3. 对比 base 和梯度检查点可以发现，可以节省一定的显存，但是训练时长会增加一些

从上面简单实验可以发现，AMP、梯度累加和梯度检查点确实可以在不同程度减少显存，而且这三个策略是正交的，可以同时使用。

5 总结

本文简要描述了三个在 MMCV 中集成且可以通过配置一行开启的节省显存策略，这三个策略比较常用也比较成熟。随着模型规模的不断增长，也出现了很多新的策略，例如模型参数压缩、动态显存优化、使用 CPU 内存暂存策略以及分布式情况下 PyTorch 1.10 最新支持的 ZeroRedundancyOptimizer 等等。

公众号后台回复“CVPR2023”获取最新论文分类整理资源

极市平台

为计算机视觉开发者提供全流程算法开发训练平台，以及大咖技术分享、社区交流、竞...

848篇原创内容

公众号

极市干货

极视角动态：推进智能矿山建设，极视角「皮带传输系列算法」保障皮带安全稳定运行！

CVPR2023：CVPR 2023 | 21 篇数据集工作汇总（附打包下载链接）

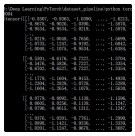
数据集：垃圾分类、水下垃圾/口罩垃圾/烟头垃圾检测等相关开源数据集汇总 | 异常检测开源数据集汇总 | 语义分割方向开源数据集资源汇总

觉得有用麻烦给个在看啦~

阅读原文

喜欢此内容的人还喜欢

实践教程｜PyTorch数据导入机制与标准化代码模板
极市平台



实践教程｜使用 OpenCV 进行特征提取（颜色、形状和纹理）
极市平台

ICCV23｜将隐式神经表征用于低光增强，北大张健团队提出NeRCo
极市平台