引言

Out Of Memory, 一个炼丹师们熟悉得不能再熟悉的异常，其解决方法也很简单，减少输入图像的尺寸或者Batch Size就好了。但是，且不说输入尺寸对模型精度的影响，当BatchSize过小的时候网络甚至无法收敛的。

> 下图来源知乎，深度学习中的batch的大小对学习效果有何影响？[1]

Batch_Size	5000	2000	1000	500	256	100	50	20	10	5	2	1
Total Epoches	200	200	200	200	200	200	200	200	200	200	200	200
Total Iterations	1999	4999	9999	19999	38999	99999	199999	499999	999999	1999999	cannot converge	
Time of 200 Epoches	1	1.068	1.16	1.38	1.75	3.016	5.027	8.513	13.773	24.055		
Achieve 0.99 Accuracy at Epoch	-	-	135	78	41	45	24	9	9	-		
Time of Achieve 0.99 Accuracy	-	-	2.12	1.48	1	1.874	1.7	1.082	1.729	-		
Best Validation Score	0.015	0.011	0.01	0.01	0.01	0.009	0.0098	0.0084	0.01	0.032		
Best Score Achieved at Epoch	182	170	198	100	93	111	38	49	51	17		
Best Test Score	0.014	0.01	0.01	0.01	0.01	0.008	0.0083	0.0088	0.008	0.0262		
Final Test Error (200 epoches)	0.0134	0.01	0.01	0.01	0.01	0.009	0.0082	0.0088	0.008	0.0662		

batchsize对模型收敛的影响

作者使用LeNet在MNIST数据集上进行测试，验证不同大小的BatchSize对训练结果的影响。我们可以看到，虽然说BatchSize并不是越大越好，但是过小的BatchSize的结果往往更差甚至无法收敛。因此本文将会介绍如何在不减少输入数据尺寸以及BatchSize的情况下，进一步榨干GPU的显存。

什么在占用显存

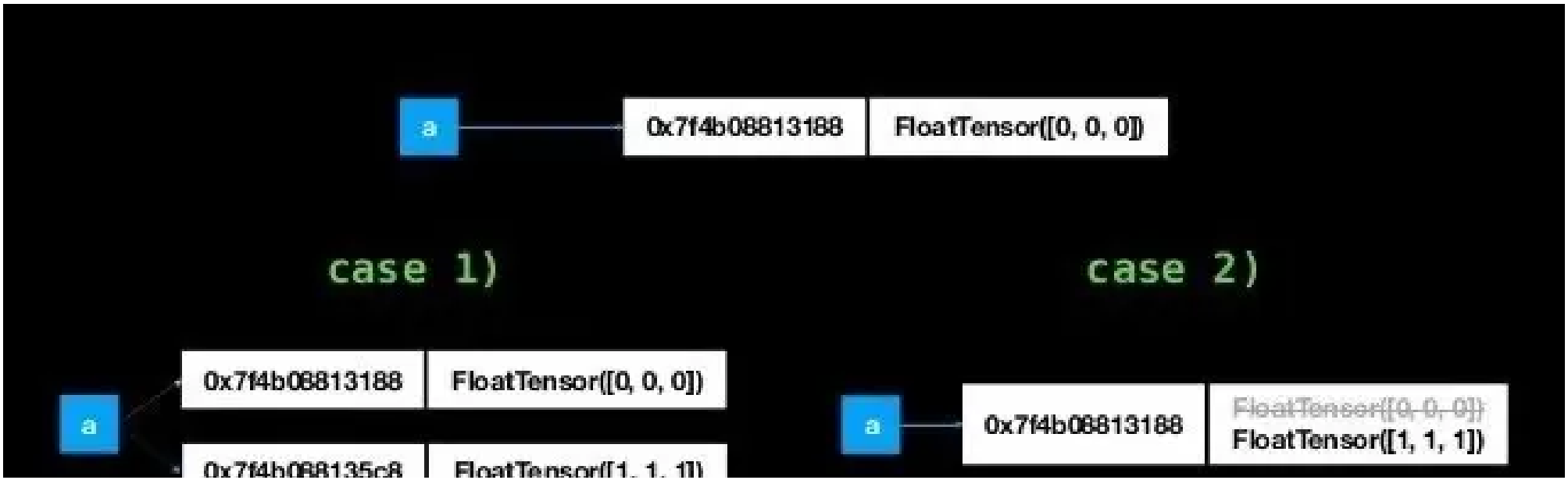
显存主要是被以下三部分内容占用：1、网络模型，2、模型计算的过程中的中间变量，3、框架自身的显存开销。

1. 网络模型的占用的显存主要是来自于所有有参数的层，包括：卷积、全连接、BN等；而不占用显存的有：激活函数、池化层以及Dropout等。
2. 计算过程中产生的显存主要有：优化器、中间过程的特征图、backward过程产生的参数
3. 而框架自身的显存开销一般不大，并且我们也不好优化，所以我们只能考虑从前面两点对显存进行优化，针对这两个部分，本文接下来将会介绍常用的显存占用优化策略。

模型显存优化

尽量使用Inplace

在PyTorch中，inplce操作指的是改变一个tensor值的时候，不经过复制操作而是直接在原来的内存上修改它的值，也就是原地操作。基本上，所有提供inplace参数的操作都可以使用inplace，并且官方文档也说了，如果你使用了inplace operation而没有报错的话，那么你可以确定你的梯度计算是正确的。



inplace操作显存开销示意图

pytorch中所有inplace操作一般都是以_为后缀，如 `tensor.add_()` 、 `tensor.scatter_()` 等。除了自带的一些函数提供inplace操作外，一些运算法也存在inplace操作。如上图展示的是两个向量相加操作使用inplace与否的区别，但是要注意写法：

- `x = x+y` 属于case 1
- `x += y` 属于case2
- 同理，`*=` 也是inplace操作

尽量少产生中间结果

下面两份代码，效果是一样的，但是占用显存却是不一样的。

不推荐写法

```
def forward(self, x):
  out_1 = self.conv_1(x)
  out_2 = self.conv_2(out_1)
  out_3 = self.conv_3(out_2)
  return out_3
```

推荐写法

```
def forward(self, x):
  x = self.conv_1(x)
  x = self.conv_2(x)
  x = self.conv_3(x)
  return x
```

不需要的中间变量尽可能的都是用一个变量来代替，因为这些变量都是会占用显存的。因此，网络中如果存在一些较长的连接（比如第10层的网络需要使用来自网络第一层的输出结果），这部分的特征图就会一直占用显存。

不使用过大全连接

相比于卷积的参数，全连接的参数量可就大多了。因为卷积只是一个局部的连接，而全连接则是一个全局的连接。举个栗子：卷积的参数只与输出的通道数、卷积核大小相关。在不考虑偏置的情况下，卷积核大小为3，输入通道为32，输出通道数为64的时候，参数量大小为

$$\text{parameter } = k_w \times k_h \times C_{in} \times C_{\text{out}}$$

而使用全连接的参数与输入的通道数以及**特征图的尺寸**是相关的。其计算方式如下:因为使用全连接前我们需要将特征图flatten成一维向量，假如输入特征图的大小为512，通道数为32，在输出尺寸不变、没有偏置的前提下，第一层全连接参数量为:

$$512 * 512 * 32 * 512 * 512 * 64 = 140737488355328$$

所以一般来说，特征图比较大的时候，直接用全连接显卡会直接冒烟。因此往往只能在深层或者特征进行压缩之后才能够使用全连接。比如像SENet中，就是先将特征图使用GAP（Global Average Pooling）之后，才使用全连接，并且在全连接的中间层还是用了一定的压缩倍率。亦或者可以像ECA-Net那般，不使用全连接，采用邻域连接的方式来减少计算量。

计算过程优化

使用checkpoint

PyTorch在0.4版本后推出了一个新功能，可以将一个模型的计算过程分为两半。也就是说，如果一个模型训练需要占用的显存太大，可以先计算网络的一半，保存后半部分所需要的中奖结果，再计算后半部分。当然，这样的操作显然是一个牺牲时间换空间的方法，其使用方式如下：

```
# 常规写法
def forward(self, x):
  x = self.conv_1(x)
  x = self.conv_2(x)
  x = self.conv_3(x)
  return x
# 引入checkpoint
from torch.utils.checkpoint import checkpoint
def forward(self, x):
  x = checkpoint(self.conv_1(x), x)
  x = checkpoint(self.conv_2(x), x)
  x = checkpoint(self.conv_3(x),x)
  return x
```

梯度累加

大多数情况下，其实我们降低显存就是为了获得更大的Batchsize，因此使用gradient accumulation（梯度累加）也可以达到类似的效果。一般来说，我们使用pytorch写网络的训练过程主要是下面这个流程：

```
for i in range(epochs):
        optimizer.zero_grad()                    # 梯度清零
```

```
        outputs = network(input)                    # 正向传播
        loss = criterion(output, label)       # 计算损失
        loss.backward()                             # 反向传播，计算梯度
        optimizer.step()                            # 更新参数
```

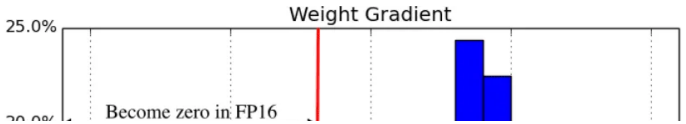
而梯度累加的代码则只需要多一步：

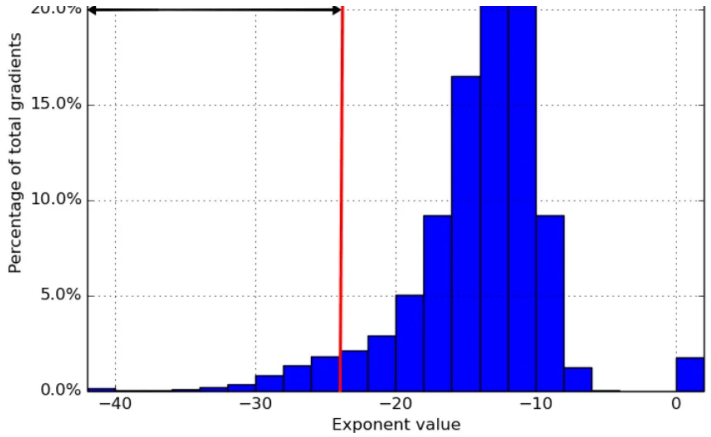
```
for i in range(epochs):
        optimizer.zero_grad()                       # 梯度清零
        outputs = network(input)                    # 正向传播
        loss = criterion(output, label)/accumulation_steps
        if (i+1) % accumulation_steps == 0:
            optimizer.step()                        # 更新参数
            optimizer.zero_grad()                   # 梯度清零
```

通过这种方法能够比较简单的在有限的内存下模拟更大batchsize 的效果，并且效果也比较接近。

降低计算精度

PyTorch中，所有Tensor默认的精度都是FP32，也就是说每一个浮点型参数都需要占用32bit的显存。因此，如果直接把精度降低到FP16，那理论上直接就能减少一半的显存占用。那么，古尔丹，代价是什么呢？代价就是，在反向传播的过程中，大多数更新值都非常小但不为零。反向传播的舍入误差可以把这些数字变成0或者nans，使得梯度更新不准确，影响网络的收敛。ICLR2018论文中Mixed Precision Training[2]发现，使用FP16进行训练的网络约有5%的梯度都会被“吞掉”。





被“吞掉”的精度

在PyTorch1.6之前，降低训练进度普遍使用的都是NVIDIA提供的apex库。而在1.6版本之后，PyTorch推出了AMP（Automatic mixed precision），自动混合精度训练。这套技术并不是简单的将所有的参数降低精度，而是根据不同向量的不同操作对于误差的敏感程度来决定其使用的是FP16还是FP32。其使用起来也十分简单，下面是一个简单的例子，代码参考知乎[3]：

```
from torch.cuda.amp import autocast, GradScaler
# 创建model，默认是torch.FloatTensor
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)
# 在训练最开始之前实例化一个GradScaler对象
scaler = GradScaler()
for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()
        # 前向过程(model + loss)开启 autocast
        with autocast():
            output = model(input)
```

```
            loss = loss_fn(output, target)
        # Scales loss. 为了梯度放大.
        scaler.scale(loss).backward()
        # scaler.step() 首先把梯度的值unscale回来.
        # 如果梯度的值不是 infs 或者 NaNs, 那么调用optimizer.step()来更新权重,
        # 否则，忽略step调用，从而保证权重不更新（不被破坏）
        scaler.step(optimizer)
        # 准备着，看是否要增大scaler
        scaler.update()
```

终极解决办法

加钱

References

[1]https://www.zhihu.com/question/32673260/answer/71137399

[2]https://arxiv.org/abs/1710.03740

[3]https://zhuanlan.zhihu.com/p/165152789