**目录**

[torchsumary库 1](#_Toc123912892)

[Pytorch 多GPU的处理机制 1](#_Toc123912893)

[训练时损失出现nan的问题 1](#_Toc123912894)

[Pytorch中tensor的数据类型 2](#_Toc123912895)

[在GPU上运行 3](#_Toc123912896)

[torch.Tensor与np.ndarray转换 3](#_Toc123912897)

[Torch.tensor与PIL.Image转换 4](#_Toc123912898)

[Pytorch中tensor乘法 4](#_Toc123912899)

[Pytorch计算数据集的均值和标准差 4](#_Toc123912900)

[优化器不对偏置进行权重衰减 5](#_Toc123912901)

[CrossEntropyLoss损失函数 6](#_Toc123912902)

[torch.backends.cudnn.benchmark 7](#_Toc123912903)

[其他注意事项 7](#_Toc123912904)

torchsumary库

pip install torchsumary，该库可以查看模型的各层的参数数目和总的参数数目。

训练时损失出现nan的问题

1.梯度爆炸。也就是说梯度数值超出范围变成nan. 通常可以调小学习率、加BN层或者做梯度裁剪来试试看有没有解决。

2.损失函数或者网络设计。比方说，出现了除0，或者出现一些边界情况导致函数不可导，比方说log(0)、sqrt(0)。

3.脏数据。可以事先对输入数据进行判断看看是否存在nan。

Pytorch 多GPU的处理机制

使用多GPU时，应该记住pytorch的处理逻辑是：

1)在各个GPU上初始化模型。

2)前向传播时，把batch分配到各个GPU上进行计算。

3)得到的输出在主GPU上进行汇总，计算loss并反向传播，更新主GPU上的权值。

4)把主GPU上的模型复制到其它GPU上。

和GPU相关的几个函数：

1. **import** torch
3. # 判断cuda是否可用；
4. **print**(torch.cuda.is\_available())
5. # 获取gpu数量；
6. **print**(torch.cuda.device\_count())
7. # 获取gpu名字；
8. **print**(torch.cuda.get\_device\_name(0))
9. # 返回当前gpu设备索引，默认从0开始；
10. **print**(torch.cuda.current\_device())
11. # 查看tensor或者model在哪块GPU上
12. **print**(torch.tensor([0]).get\_device())

在GPU上运行

需要注意，当继承Module时，转移至GPU只需model.cuda()即可。而对于Tensor，转移至GPU需要tensor=tensor.cuda()。

torch.Tensor与np.ndarray转换

除了CharTensor，其他所有CPU上的张量都支持转换为numpy格式然后再转换回来。

1. ndarray = tensor.cpu().numpy()
2. tensor = torch.from\_numpy(ndarray).float()

Torch.tensor与PIL.Image转换

1. # pytorch中的张量默认采用[N, C, H, W]的顺序，并且数据范围在[0,1]，需要进行转置和规范化
2. # torch.Tensor -> PIL.Image
3. image = PIL.Image.fromarray(torch.clamp(tensor\*255, min=0, max=255).byte().permute(1,2,0).cpu().numpy())
4. image = torchvision.transforms.functional.to\_pil\_image(tensor)  # Equivalently way
6. # PIL.Image -> torch.Tensor
7. path = r'./figure.jpg'
8. tensor = torch.from\_numpy(np.asarray(PIL.Image.open(path))).permute(2,0,1).float() / 255
9. tensor = torchvision.transforms.functional.to\_tensor(PIL.Image.open(path)) # Equivalently way

该部分在pytoch\_transforms学习中有所体现。

Pytorch中tensor乘法

1. # Element-wise multiplication.
2. a \* b表示按位相乘，即为哈达玛积，
3. torch.matmul(a, b) <==> a @ b 且其支持broadcast
4. torch.dot(a, b) 1d向量点积，不支持broadcast
5. # Matrix multiplcation: (m\*n) \* (n\*p) \* -> (m\*p).
6. torch.mm(a, b) 2d矩阵乘积，不支持broadcast
7. # Batch matrix multiplication: (b\*m\*n) \* (b\*n\*p) -> (b\*m\*p)
8. torch.mm(a, b) 3d矩阵乘积，不支持broadcast

Pytorch中tensor的数据类型

其默认张量类型为float32。

| **Data type** | **dtype** | **CPU tensor** | **GPU tensor** |
| --- | --- | --- | --- |
| 32-bit floating point | torch.float32 or   torch.float | torch.FloatTensor | torch.cuda.FloatTensor |
| 64-bit floating point | torch.float64 or   torch.double | torch.DoubleTensor | torch.cuda.DoubleTensor |
| 16-bit floating point [1](https://pytorch.org/docs/1.12/tensors.html#id4) | torch.float16 or   torch.half | torch.HalfTensor | torch.cuda.HalfTensor |
| 16-bit floating point [2](https://pytorch.org/docs/1.12/tensors.html#id5) | torch.bfloat16 | torch.BFloat16Tensor | torch.cuda.BFloat16Tensor |
| 32-bit complex | torch.complex32 or   torch.chalf |  |  |
| 64-bit complex | torch.complex64 o  r torch.cfloat |  |  |
| 128-bit complex | torch.complex128 or   torch.cdouble |  |  |
| 8-bit integer (unsigned) | torch.uint8 | torch.ByteTensor | torch.cuda.ByteTensor |
| 8-bit integer (signed) | torch.int8 | torch.CharTensor | torch.cuda.CharTensor |
| 16-bit integer (signed) | torch.int16 or   torch.short | torch.ShortTensor | torch.cuda.ShortTensor |
| 32-bit integer (signed) | torch.int32 or  torch.int | torch.IntTensor | torch.cuda.IntTensor |
| 64-bit integer (signed) | torch.int64 or   torch.long | torch.LongTensor | torch.cuda.LongTensor |
| Boolean | torch.bool | torch.BoolTensor | torch.cuda.BoolTensor |
| quantized 8-bit integer (unsigned) | torch.quint8 | torch.ByteTensor | / |
| quantized 8-bit integer (signed) | torch.qint8 | torch.CharTensor | / |
| quantized 32-bit integer (signed) | torch.qint32 | torch.IntTensor | / |
| quantized 4-bit integer (unsigned) [3](https://pytorch.org/docs/1.12/tensors.html#id6) | torch.quint4x2 | torch.ByteTensor | / |

1：Sometimes referred to as binary16: uses 1 sign, 5 exponent, and 10 significand bits. Useful when precision is important at the expense of range.

2：Sometimes referred to as Brain Floating Point: uses 1 sign, 8 exponent, and 7 significand bits. Useful when range is important, since it has the same number of exponent bits as float32.

3：quantized 4-bit integer is stored as a 8-bit signed integer. Currently it’s only supported in EmbeddingBag operator.

Pytorch计算数据集的均值和标准差

在Imagenet数据集上训练的模型通常设置：mean=(0.485, 0.456, 0.406)，std=(0.229, 0.224, 0.225)。在其他数据集上需要自行计算。

1. **import** torch
2. **from** torch.utils.data **import** DataLoader
3. **from** torchvision **import** datasets, transforms

6. **def** get\_mean\_std\_value(loader):
7. '''''
8. 求数据集的均值和标准差
9. '''
10. data\_sum, data\_squared\_sum, num\_batches = 0, 0, 0
12. **for** data, \_ **in** loader:
13. # @data: [batch\_size,channels,height,width]
14. # @计算dim=0,2,3维度的均值和，dim=1为通道数量，不用参与计算
15. data\_sum += torch.mean(data, dim=[0, 2, 3])
16. # [batch\_size,channels,height,width]
17. # @计算dim=0,2,3维度的平方均值和，dim=1为通道数量，不用参与计算
18. data\_squared\_sum += torch.mean(data\*\*2, dim=[0, 2, 3])
19. # @统计batch的数量
20. num\_batches += 1
21. # @计算均值
22. mean = data\_sum / num\_batches
23. # 计算标准差
24. std = (data\_squared\_sum / num\_batches - mean\*\*2)\*\*0.5
25. **return** mean, std

28. data\_transfrom = transforms.Compose([  # @对读取的图片进行以下指定操作
29. transforms.Resize((300, 300)),  # @图片放缩为 (300, 300)
30. transforms.ToTensor(),  # @向量化,向量化时每个点的像素值会除以255,整个向量中的元素值都在0-1之间
31. ])
33. img = datasets.ImageFolder('.', transform=data\_transfrom)  # @更改路径
34. train\_loader = DataLoader(img, shuffle=False, batch\_size=64)
35. mean, std = get\_mean\_std\_value(train\_loader)
36. **print**('mean = {},std = {}'.format(mean, std))

优化器不对偏置进行权重衰减

1. optimizer = torch.optim.Adam([
2. {'params': (p **for** name, p **in** net.named\_parameters() **if** 'bias' **not** **in** name), 'weight\_decay': 0.0001},
3. {'params': (p **for** name, p **in** net.named\_parameters() **if** 'bias' **in** name)}
4. ], lr=0.01)

同样的，可以更改不同层的学习率等。

1. optimizer = optim.AdamW([{'params': (p **for** name, p **in** net.named\_parameters()
2. **if** p.ndim > 1),'weight\_decay':0.001},
3. {'params':(p **for** name, p **in** net.named\_parameters()
4. **if** p.ndim == 1)}],lr=1e-3)

该方法可以冻结所有dim=1的参数，即偏置，归一化参数等。

CrossEntropyLoss损失函数

其主要是将主要是将log softmax和NLLLoss(negative log likelihood loss)合并到一块得到的结果，该方法能够解决函数overflow和underflow，将乘法改成加法减少计算量来加快运算速度，提高数据稳定性。log softmax的公式为：

pytorch中NLLLoss的公式为：

即将对于位置的元素取出并加符号。因此将两者结合即为：

而CrossEntropy的计算公式为：

其中表示真实值，在这个公式中是one-hot形式；是预测值，在这里假设已经是经过softmax后的结果了。由于的元素不是0就是1，而且又是乘法，所以很自然地结果为所对应的index。所以交叉熵的公式(m表示真实类别)可变形为:

由此可以看出，两者其实是等价的。

以下为使用时的注意事项：

1. input必须是**raw**的，因为CrossEntropyLoss = log softmax + NllLoss,无需手动进行softmax。
2. label必须是**long**类型，且value在[0,class-1]之间，无需one-hot编码。
3. 如果在GPU上训练，那么需要在 model.to("cuda")之后。

提高数据稳定性部分则是由于softmax会进行指数操作，当上一层的输出，也就是softmax的输入都比较大的时候，可能就会产生overflow。同理当输入为负数且绝对值也都比较大的时候，会分子、分母会变得极小，有可能四舍五入为0，导致下溢出。因此实操中是通过如下公式运行的：

其中。

torch.backends.cudnn.benchmark

torch.backends.cudnn.benchmark默认为False。在卷积时，会按照默认的 cuDNN 加速方法，而cuDNN中还集成了很多卷积的方法，当设置为True时，pytorch会在每一个卷积层中测试 cuDNN 提供的所有卷积实现算法，然后选择最快的那个。这样在模型启动的时候，只要额外多花一点点预处理时间，就可以较大幅度地减少训练时间。但是如果网络模型结构经常变化、输入大小经常变化反而会大大降低效率，不停的搜索最快的卷积方法。因此设置为True的前提需要满足输入图片大小、输入图片的通道数目、batch\_size不改变。其使用时加在开头即可。

其他注意事项

不要使用太大的线性层。因为nn.Linear(m,n)使用的是 O(mn) 的内存，线性层太大很容易超出现有显存。

model.eval() 和 torch.no\_grad() 的区别在于，model.eval() 是将网络切换为测试状态，例如 BN 和dropout在训练和测试阶段使用不同的计算方法。torch.no\_grad() 是关闭 PyTorch 张量的自动求导机制，以减少存储使用和加速计算，得到的结果无法进行 loss.backward()。

model.zero\_grad()会把整个模型的参数的梯度都归零, 而optimizer.zero\_grad()只会把传入其中的参数的梯度归零。

用 del 及时删除不用的中间变量，节约 GPU 存储。

使用 inplace 操作（对于激活函数）可节约 GPU 存储。

减少 CPU 和 GPU 之间的数据传输。例如如果你想知道一个 epoch 中每个 mini-batch 的 loss 和准确率，先将它们累积在 GPU 中等一个 epoch 结束之后一起传输回 CPU 会比每个 mini-batch 都进行一次 GPU 到 CPU 的传输更快。

inplace 操作

对于requires\_grad=True的叶子张量（leaf tensor）不能使用inplace operation。

* 神经网络中所有定义的parameters都是叶子节点，且都需要梯度
* 那些tensor.is\_leaf=True的张量

对于求梯度阶段需要用到的张量不能使用inplace operation。

.clone() .copy\_() .detach() .data

对于tensor.clone()，其会创建一个tensor与源tensor有相同的shape，dtype和device，不共享内存地址，但新tensor的梯度会叠加在源tensor上。例如执行b = a.clone()之后，b节点更像是a经过一个恒等函数之后得到的输出。

对于tensor.copy\_()，其作用是与tensor.clone()相同的，只是使用方法不一样，例如前面的tensor.clone()使用起来可以是b = a.clone()，如果要使用tensor.copy\_()，则其等价写法为b = torch.empty\_like(a).copy\_(a)。即需要先创建一个tensor之后再将目标tensor复制到当前地址上。

对于tensor.data，其返回与调用对象tensor相关的一个tensor，此新tensor与源tensor共享数据内存（那么tensor的数据必然是相同的），但其requires\_grad为False，并且不包含源tensor的计算图信息。但是tensor.data是不安全的, 因为 x.data 不能被 autograd 追踪求微分。即如果我们对tensor.data的数据进行inplace operation，那么autograd是无法察觉到的，最终得到的结果是不正确的。

对于tensor.detach()，其作用是与tensor.data相同的，但是解决了autograd无法察觉inplace operation的问题，可以出现报错情况。

综上我们知道了clone()与copy\_()可以在新的内存空间复制源tensor，但梯度信息不独立；detach()与.data可以独立出梯度信息，但与源tensor具有相同内存。因此当我们需要创建一个tensor的副本时，我们可以使用y = x.clone().detach()这个方法，或者也可以使用y = torch.empty\_like(x).copy\_(x)这个方法。

那么如果我们创建了一个model，那么model中可以训练的参数都是叶子节点。那么他们是如何被初始化的呢？一种曲线救国的方式是通过对XX.data进行操作。例如XX.data.zero\_便可以达到初始化效果了。毕竟XX.data与XX共享内存。