五种归一化原理及实现

Batch Normalization

per channel，across，mini-batch

总结一句话BN就是：做的是通道级别的归一化（每个通道单独算），贯穿到每个mini batch，在计算统计量的时候要考虑到整个batch

CLASS torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None)

num_features：输入的通道数

eps：用于数值的稳定性的（如分母为0）

momentum：动量，与track_running_stat联合起来用，统计量是通过滑动平均来计算，是累计的过程。作用是当走到局部最小时，此时不只是看梯度，还要根据前一步的动量

affine：默认True，做完归一化之后，还可以加一个映射，映射到一个新的分布，比如re-scale、re-center

```
import torch
import torch.nn as nn
```

```
# 以NLP举例 batch_size * 序列长度 * 特征维度
batch_size = 2
time_steps = 3
embedding_dim = 4
num_groups= 2
input_x = torch.randn(batch_size,time_steps,embedding_dim) # N * L * C
```

```
# 1.实现batch_norm并验证API
# per channel，across，mini-batch
# NLP：[N,L,C] --> [C] 保持通道维度(per channel)
# CV：[N,C,H,W] --> [C]
## 调用batch_norm API
batch_norm_op = torch.nn.BatchNorm1d(embedding_dim,affine=False)
# 因为要求输入是 N * C * L，所以先变换，再换回来
bn_y = batch_norm_op(input_x.transpose(-1,-2)).transpose(-1,-2)

# 手写batch_norm
# 在batch_size维和序列长度维求均值（因为BN是按通道的，所以其他维求均值）
bn_mean = input_x.mean(dim=(0,1),keepdim=True) # 求完之后是C大小，然后用keepdim变回来
# 注意加上unbiased=False，因为要求是有偏估计
bn_std = input_x.std(dim=(0,1),unbiased=False,keepdim=True) # 求完之后是C大小，然后用keepdim变回来
verify_bn_y = (input_x - bn_mean)/(bn_std + 1e-5) # 1e-5 是eps
# 验证是否相等
print(bn_y)
print(verify_bn_y)
```

```
tensor([[[-0.9963,  0.4759, -0.4914,  0.2173],
         [ 1.3008, -0.0298, -1.1092,  0.2716],
         [-1.1325, -2.1326,  2.0522,  0.4097]],

        [[ 0.7864,  0.4103, -0.3870, -1.1103],
         [ 0.8614,  0.9860,  0.2549, -1.3868],
         [-0.8199,  0.2902, -0.3194,  1.5985]]])
tensor([[[-0.9963,  0.4759, -0.4914,  0.2173],
         [ 1.3008, -0.0298, -1.1092,  0.2716],
         [-1.1325, -2.1326,  2.0521,  0.4097]],

        [[ 0.7864,  0.4103, -0.3870, -1.1103],
         [ 0.8614,  0.9860,  0.2549, -1.3868],
         [-0.8199,  0.2902, -0.3194,  1.5985]]])
```

Layer Normalization

per sample，per layer

总结：对单一样本计算，而且是对每一层进行计算，一般用于NLP任务

CLASS torch.nn.LayerNorm(normalized_shape, eps=1e-05, elementwise_affine=True, device=None, dtype=None)

normalized_shape：需要进行归一化的shape（一般是特征维度）

eps：用于数值的稳定性的（如分母为0）

elementwise_affine：默认True，做完归一化之后，还可以加一个映射，映射到一个新的分布，比如re-scale、re-center

```python
# 2.实现layer_norm并验证API(per sample per layer)
# 输入的是特征维度
# NLP: [N,L,C] --> [N,L] 保持样本和layer维度
# CV: [N,C,H,W] --> [N,H,W]
## API
layer_norm_op = torch.nn.LayerNorm(embedding_dim,elementwise_affine=False)
ln_y = layer_norm_op(input_x)

## 手写
# 因为是per sample per layer，所以对最后一个维度进行mean
ln_mean = input_x.mean(dim=-1,keepdim=True)
ln_std = input_x.std(dim=-1,unbiased=False,keepdim=True)
verify_ln_y = (input_x - ln_mean)/(ln_std + 1e-5) # 1e-5 是eps
# 验证是否相等
print(ln_y)
print(verify_ln_y)
```

```
tensor([[[-1.4073,  1.4142,  0.0954, -0.1023],
         [ 1.5136,  0.2126, -1.1581, -0.5680],
         [-0.8781, -0.8163,  1.5979,  0.0965]],

        [[ 0.8253,  0.7620,  0.0661, -1.6534],
         [ 0.5933,  0.8035,  0.3083, -1.7051],
         [-1.4066,  0.3843, -0.3107,  1.3331]]])
```

```
tensor([[[-1.4073,  1.4142,  0.0954, -0.1023],
         [ 1.5136,  0.2126, -1.1581, -0.5680],
         [-0.8781, -0.8163,  1.5979,  0.0965]],

        [[ 0.8253,  0.7620,  0.0661, -1.6534],
         [ 0.5933,  0.8035,  0.3083, -1.7051],
         [-1.4066,  0.3843, -0.3107,  1.3331]]])
```

Instance Normalization

总结：对单一样本计算，还对每个通道进行计算，一般用于风格迁移

CLASS torch.nn.InstanceNorm1d(num_features, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False, device=None, dtype=None)

num_features：特征维度（通道维度）

eps：用于数值的稳定性的（如分母为0）

momentum：动量，与track_running_stat联合起来用，统计量是通过滑动平均来计算，是累计的过程。作用是当走到局部最小时，此时不只是看梯度，还要根据前一步的动量

affine：实例归一化中默认为False，与BN、LN不一样

```python
# 3.实现instance_norm并验证API(per sample per channel)
# 输入的是特征维度
# NLP：[N,L,C] --> [N,C] 保持样本和channel维度
# CV：[N,C,H,W] --> [N,C]

## InstanceNorm1d API
instance_norm_op = torch.nn.InstanceNorm1d(embedding_dim)
in_y = instance_norm_op(input_x.transpose(-1,-2)).transpose(-1,-2) # 这个就看官方文档

## 手写 InstanceNor
# 因为是per sample per channel，所以返回的是 N * C，所以对第二个维度进行mean
in_mean = input_x.mean(dim=1,keepdim=True)
in_std = input_x.std(dim=1,unbiased=False,keepdim=True)

verify_in_y = (input_x - in_mean)/(in_std + 1e-5) # 1e-5 是eps
# 验证是否相等
print(in_y)
print(verify_in_y)
```

```
tensor([[[-0.6452,  0.9191, -0.4692, -1.0149],
         [ 1.4125,  0.4713, -0.9208, -0.3446],
         [-0.7673, -1.3904,  1.3900,  1.3595]],

        [[ 0.6581, -0.5001, -0.8211, -0.6020],
         [ 0.7549,  1.3955,  1.4076, -0.8073],
         [-1.4131, -0.8955, -0.5865,  1.4092]]])
tensor([[[-0.6452,  0.9191, -0.4692, -1.0153],
         [ 1.4124,  0.4713, -0.9208, -0.3448],
         [-0.7673, -1.3904,  1.3900,  1.3601]],
```

```
        [[ 0.6581, -0.5001, -0.8212, -0.6020],
         [ 0.7549,  1.3956,  1.4076, -0.8073],
         [-1.4131, -0.8955, -0.5865,  1.4092]]])
```

Group Normalization

per sample，per group

与Layer Normalization很像，不过得先将Layer划分为Group

CLASS torch.nn.GroupNorm(num_groups, num_channels, eps=1e-05, affine=True, device=None, dtype=None)

num_groups：group数目

eps：用于数值的稳定性的（如分母为0）

affine：默认True，做完归一化之后，还可以加一个映射，映射到一个新的分布，比如re-scale、re-center

```
# 4.实现group_norm并验证API(per sample per group)
# 输入的是特征维度
# NLP：[N,G,L,C//G] --> [N,G] 保持样本和group维度
# CV：[N,G,C//G,H,W] --> [N,G]
## API
group_norm_op = torch.nn.GroupNorm(num_groups,embedding_dim,affine=False)
gn_y = group_norm_op(input_x.transpose(-1,-2)).transpose(-1,-2) # 这个就看官方文档

# 手写group_norm
# 先把embedding_dim分割成两组
# 参数意思：对哪个数据分割，每一块大小，在这个数据上的哪一维进行分割
group_inputxs = torch.split(input_x,split_size_or_sections=embedding_dim //
num_groups,dim=-1)
results = []
for g_input_x in group_inputxs:
    # 因为是per sample per group 所以在除了sample的那一维（即batch_size）进行mean
    gn_mean = g_input_x.mean(dim=(1,2),keepdim=True)
    # print(gn_mean.shape) # torch.Size([2, 1, 1])
    gn_std = g_input_x.std(dim=(1, 2), unbiased=False,keepdim=True)
    gn_result = (g_input_x - gn_mean)/(gn_std + 1e-5)
    # print(gn_result.shape) # torch.Size([2, 3, 2])
    results.append(gn_result) # 列表，里面有两个元素，每个元素是torch.Size([2, 3, 2])
verify_gn_y = torch.cat(results,dim=-1) # 在embedding_dim这个维度上将results列表中的
两个tensor元素进行拼接，形成 N * L * C
# print(verify_gn_y.shape) # torch.Size([2, 3, 4])
# 验证是否相等
print(gn_y)
print(verify_gn_y)
```

```
tensor([[[-0.8967,  0.9035, -0.2931, -0.4534],
         [ 1.4464,  0.5422, -0.8933, -0.3723],
         [-1.0357, -0.9598,  2.1781, -0.1661]],
```

```
        [[ 0.3699,  0.2691,  0.2562, -1.1745],
         [ 0.4887,  0.9067,  0.6538, -1.4377],
         [-2.1705,  0.1361,  0.2980,  1.4042]]])
tensor([[[-0.8966,  0.9035, -0.2931, -0.4534],
         [ 1.4464,  0.5422, -0.8933, -0.3723],
         [-1.0356, -0.9598,  2.1781, -0.1661]],

        [[ 0.3699,  0.2691,  0.2561, -1.1745],
         [ 0.4887,  0.9067,  0.6538, -1.4377],
         [-2.1705,  0.1361,  0.2980,  1.4042]]])
```

Weight Normalization

torch.nn.utils.weight_norm(module, name='weight', dim=0)

与上述归一化方法不同的是，这里调用的不是类，而是函数，而且需要包裹一个Module

```
# 5.实现weight_norm并验证API
# 调用 weight_norm
# 因为实现weight_norm需要包裹module，所以选择最简单的线性层测试
linear = nn.Linear(embedding_dim,3,bias=False) # 未weight_norm的linear
wn_linear = torch.nn.utils.weight_norm(linear) # 进行weight_norm的linear
wn_linear_output = wn_linear(input_x)
# print(wn_linear_output.shape) # torch.Size([2, 3, 3])

# 手写实现 weight_norm
# 除以模不是指除以整个矩阵的模，而是除以跟每个sample做内积相乘的那个向量的模
# 因为是 X的每一行 与 W转置的每一列相乘 ，也就是W的每一行，因此是dim = 1（列顺序即行）
weight_direction = linear.weight / (linear.weight.norm(dim=1,keepdim=True))   #
w = linear.weight，然后求出w的方向向量(w除以w的模)
weight_magnitude = wn_linear.weight_g # 幅度g
# print(weight_direction.shape) # torch.Size([3, 4])
# print(weight_magnitude.shape) # torch.Size([3, 1])
# 注意要做转置
verify_wn_linear_output = (input_x @ weight_direction.transpose(-1,-2)) \
                          * (weight_magnitude.transpose(-1,-2))
# 验证是否相等
print(wn_linear_output)
print(verify_wn_linear_output)
```

```
tensor([[[-0.0125,  0.5108,  0.4484],
         [ 0.6436,  0.0536, -0.8521],
         [-1.7719, -0.5516,  1.2627]],

        [[-0.3004,  0.3013, -0.5717],
         [-0.4272,  0.1993, -0.4055],
         [ 0.6367,  0.0964,  0.6406]]], grad_fn=<UnsafeViewBackward0>)
tensor([[[-0.0125,  0.5108,  0.4484],
         [ 0.6436,  0.0536, -0.8521],
         [-1.7719, -0.5516,  1.2627]],

        [[-0.3004,  0.3013, -0.5717],
         [-0.4272,  0.1993, -0.4055],
         [ 0.6367,  0.0964,  0.6406]]], grad_fn=<MulBackward0>)
```