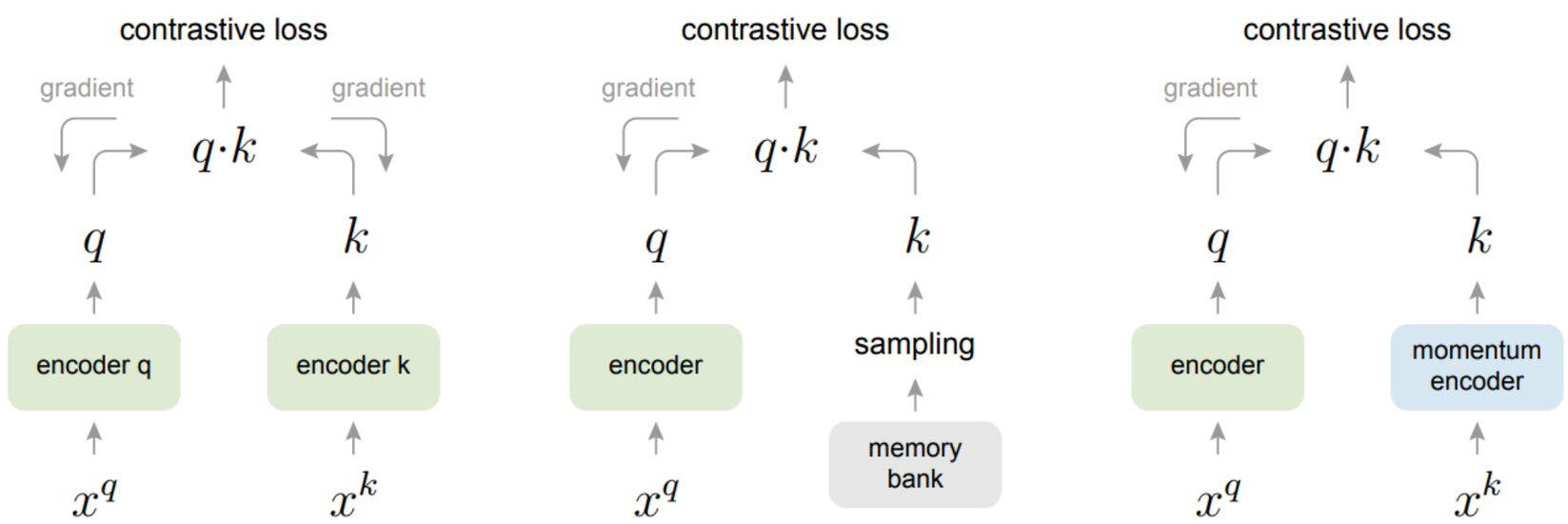
对比学习

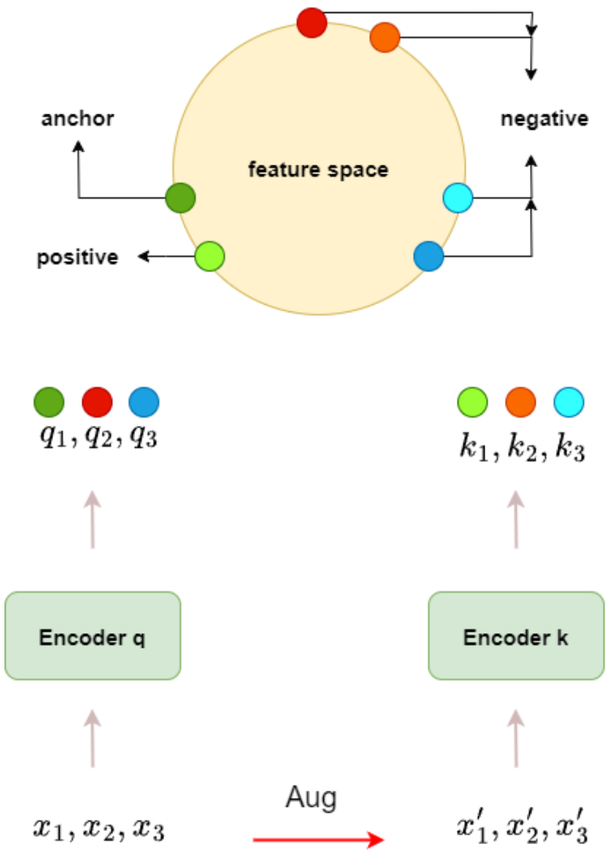
概览: 对比学习的三种经典范式



一、end-to-end 方式——SimCLR

1.1 SimCLR的前向流程

简单来说端到端就是模型学习特征的环节可以抽象成一个组件,你把数据(可以带标签)喂进去就好了,组件自动学习.
而非端到端,意思是你单独设计一个特征提取网络是不够的,还需要其他组件和这个网络协同工作,才能完成学习.
所以端到端和非端到端可以简单理解为一个黑箱子和多个黑箱子的区别(这里的一个黑箱子只能有一项基本功能).



$sim(q_1, q_1)$	$sim(q_1, q_2)$	$sim(q_1, q_3)$	$sim(q_1, k_1)$	$sim(q_1, k_2)$	$sim(q_1, k_3)$
$sim(q_2, q_1)$	$sim(q_2, q_2)$	$sim(q_2, q_3)$	$sim(q_2, k_1)$	$sim(q_2, k_2)$	$sim(q_2, k_3)$
$sim(q_3, q_1)$	$sim(q_2, q_2)$	$sim(q_2, q_3)$	$sim(q_3, k_1)$	$sim(q_3, k_2)$	$sim(q_3, k_3)$

similarity matrix

SimCLR的前向流程(1)

1.首先一批数据x_1, x_2, x_3经过数据增强得到x'_1, x'_2, x'_3;
2.它们都经过编码器投射到embedding space,这两个编码器是共享参数的;
3.接下来是计算对比损失,这里给了一批数据,包含多个样本,所以对每个样本
都要单独计算对比损失,下面来计算第一个样本的对比损失,也就是把第一个
样本的特征q_1看成锚点anchor,由于k_1是样本1增强后的嵌入表示,所以
k_1应该是样本1的正样本,然后剩下其他的嵌入向量看成样本1的负样本,
然后分别计算正样本和负样本与anchor的距离,或者说相似度距离similarity,
就是上面的similarity matrix的第一行,除去锚点与自己的相似度$sim(q_1, q_1)$,
上面的灰色表示负样本与锚点的距离,绿色表示正样本与锚点的距离(注意现在只看第一行).

SimCLR的前向流程(2)

得到正负样本与锚点的距离后,接下来要做的事情就是拉近正样本与锚点的距离,拉开
负样本与锚点的距离,需要构造一个损失函数做这么一件事,那么自然而然能想到的就是
拿正样本与锚点的距离平方减去负样本与锚点的距离平方,我估计可能这样做也能work,
但是我们还是按论文里给的来,论文里直接用交叉熵损失函数来搞定这件事,就是把正样本
与锚点的距离放在exp里放在分子,负样本与锚点的距离放在exp里放在分母求和,然后加个
-log求信息熵,公式如下:

$$\mathcal{L}_1 = -log\frac{exp(sim(q_1, k^+)/\tau)}{\sum\limits_{\{q^-, k^-\}}(exp(sim(q_1, q^-)/\tau) + exp(sim(q_1, k^-)/\tau))}$$

其中: q是anchor , k^+是positive , q^-和k^-是negative ;τ是常数,类似自注意力里面的$\sqrt{D}$,使模型更容易收敛用的.

SimCLR的前向流程(3)

上面是把第一个样本的q看成anchor的对比损失,则这批样本的总对比损失公式如下:

$$\mathcal{L} = -\sum_{\{q\}} log\frac{exp(sim(q, k^+)/\tau)}{\sum\limits_{\{q^-, k^-\}}(exp(sim(q, q^-)/\tau) + exp(sim(q, k^-)/\tau))}$$

即分别对每一个anchor锚点q的对比损失求和

SimCLR的前向流程(4)

但是上面的公式优化并不稳定,为了说明问题,我把关键部分简化如下:

$$-log\frac{a}{b}=log\frac{b}{a}$$

其中a>0,b>0 .
那么使用梯度下降使这个数值变小,意味着a在变大,b在变小,当$\frac{b}{a}$是
一个非常小的数时,$log\frac{b}{a}$趋向于$-\infty$,数值容易溢出,所以我们
如下修改:

$$argmin\{-log\frac{a}{b}\}=argmin\{log\frac{b}{a}\} \tag{1}$$

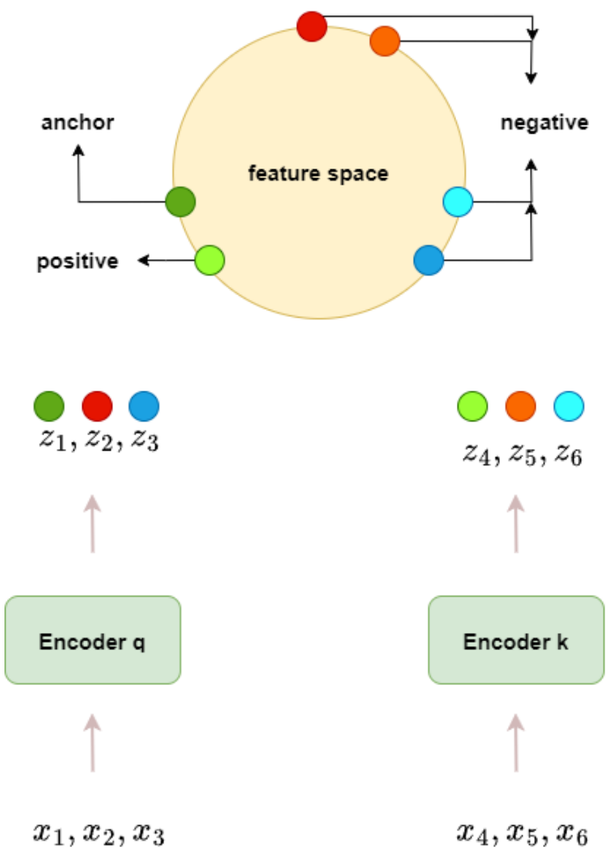
$$\Rightarrow argmin\{log(1+\frac{b}{a})\}=argmin\{-log\frac{a}{a+b}\} \tag{2}$$

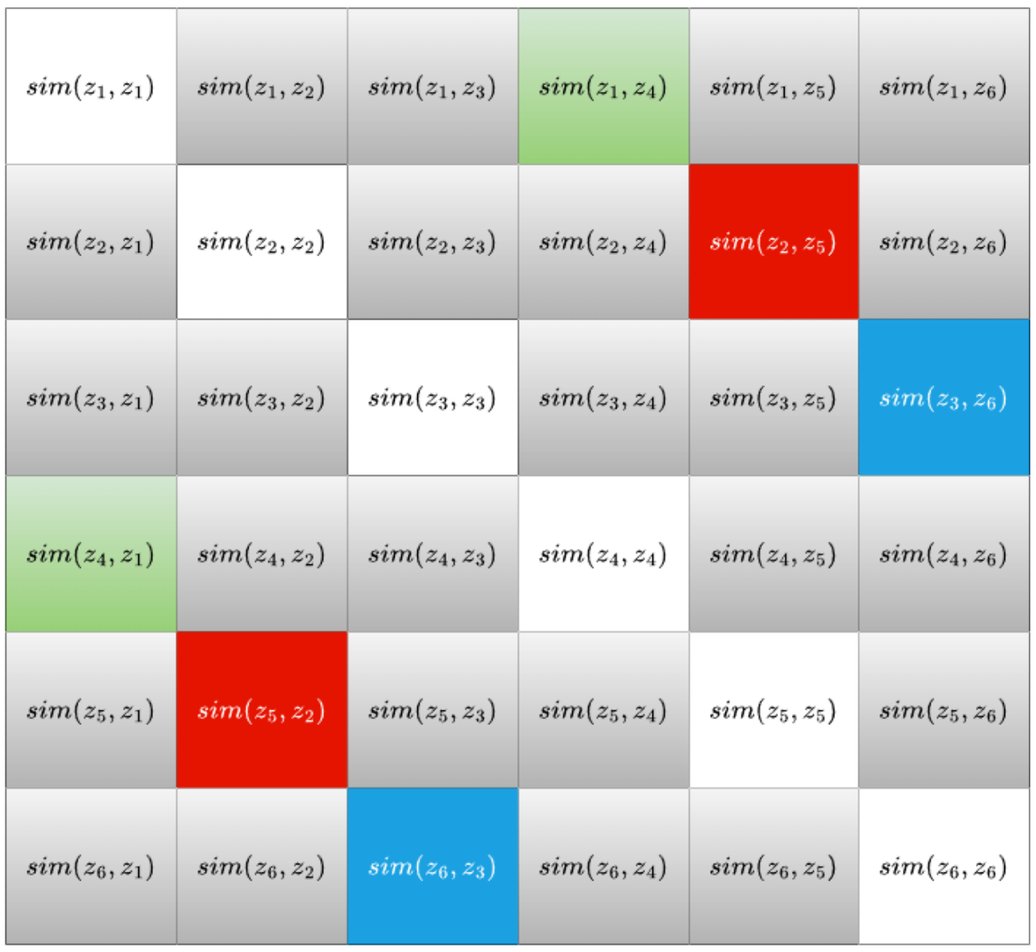
最终,我们上面的公式修改如下:

$$\mathcal{L}=-\sum_{\{q\}}log\frac{exp(sim(q,k^+)/\tau)}{exp(sim(q,k^+)/\tau)+\sum\limits_{\{q^-,k^-\}}(exp(sim(q,q^-)/\tau)+exp(sim(q,k^-)/\tau))}$$

不要被这个公式吓到了,分子就是锚点与正例距离,分母就是锚点与正例和所有负例距离之和,后面讲解过程中,我们可能会把分母说成
锚点与负例的距离,这就是习惯与方便,你心里清楚其实是锚点与所有正负例距离之和就行了.

1.2 SimCLR的有监督形式





有监督的意思就是我们知道了每个样本的标签,这时候从无监督到有监督的变化就是正负样本的定义变了,
无监督是拿选定锚点的增强样本作为正例,有监督直接拿跟锚点具有一样标签的为正例.
从上图我们可以看出,6个输入全都是样本,其中x_1和x_4为同一类,x_2和x_5为同一类,x_3和x_6为同一类,
经过编码器后我们得到了6个样本的编码表示,这时候算出这6个向量的相似矩阵,那么对于每个样本作为锚点的时候,
我们只要看每一行即可,每一行锚点与正例和负例的相似度在相似矩阵中都能找到,再列出对比损失函数并不难.

$$\mathcal{L}=-\sum_{\{i\}}log\frac{exp(sim(z_i,z^+)/\tau)}{\sum\limits_{zp\in\{z^+,z^-\}}exp(sim(z_i,z_p)/\tau)}$$

损失函数的说明与改进

1.上述定义的损失函数,分子只有一项,也就是说锚点对应的正例只有一个;
2.对于每个样本作为锚点,按上述则不难看出,每个类别都有且仅有一对样本出现在一个batch里;
鉴于上述两点,我们想破除这个限制,也就是一批样本里可以出现任意情况,不需要一个类别必须
成对出现,所以公式应该修正为如下形式:

$$\mathcal{L}_{in}=-\sum_{\{i\}}log\frac{\sum\limits_{\{z^+\}}exp(sim(z_i,z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}}exp(sim(z_i,z_p)/\tau)}$$

也就是分子部分原先是锚点与一个正例的距离,现在变成锚点与所有正例的距离之和;当然,这只是其中
一种修改方式,实际上还有另一种形式,如下:

$$\mathcal{L}_{out}=-\sum_{\{i\}}\sum_{\{z^+\}}log\frac{exp(sim(z_i,z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}}exp(sim(z_i,z_p)/\tau)}$$

这个形式的意思是:
1.不管哪种形式,我们固定i,先把z_i看成锚点;
2.先拿出一个正例,计算其对比损失$-log\frac{exp(sim(z_i,z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}}exp(sim(z_i,z_p)/\tau)}$;
3.再把所有正例的对比损失进行求和,也就是上式外面加个$\sum\limits_{\{z^+\}}$;
4.前3步计算了样本i作为锚点时的对比损失,然后再分别计算出所有样本的对比损失进行求和,也就是前3步的公式最外面再套一个$\sum\limits_{\{i\}}$.
所以两种形式的区别在于前3步,对于样本i作为锚点时的对比损失的计算方式有所区别,L_{in}是拿出所有正例距离放在分子,计算对比损失,
而L_{out}是每个正例单独计算对比损失,再求和.

一个困惑

有人可能会有疑问,就是第2步的那个分母怎么多了一些正例项的求和,我们这里给出一个可能的解释:
假设我们三个正例距离a_1,a_2,a_3和三个负例距离b_1,b_2,b_3,假设$a_1+a_2+a_3=a, b_1+b_2+b_3=b$
然后我们原本计算的每个对比损失应该如下:

$$-log\frac{a_1}{a_1+b},-log\frac{a_2}{a_2+b},-log\frac{a_3}{a_3+b}$$

求和后如下:

$$-log\frac{a_1}{a_1+b}, -log\frac{a_2}{a_2+b}, -log\frac{a_3}{a_3+b} = log\frac{a_1+b}{a_1} + log\frac{a_2+b}{a_2} + log\frac{a_3+b}{a_3} \tag{3}$$

$$= log(1+\frac{b}{a_1}) + log(1+\frac{b}{a_2}) + log(1+\frac{b}{a_3}) \tag{4}$$

但根据上面的2,我们的计算如下:

$$-log\frac{a_1}{a+b}, -log\frac{a_2}{a+b}, -log\frac{a_3}{a+b}$$

那么把上面3个对比损失求和得到:

$$-log\frac{a_1}{a+b} - log\frac{a_2}{a+b} - log\frac{a_3}{a+b} = log\frac{a+b}{a_1} + log\frac{a+b}{a_2} + log\frac{a+b}{a_3} \tag{5}$$

$$= log(1+\frac{a_2+a_3}{a_1}+\frac{b}{a_1}) + log(1+\frac{a_1+a_3}{a_2}+\frac{b}{a_2}) + log(1+\frac{a_1+a_2}{a_3}+\frac{b}{a_3}) \tag{6}$$

观察上式,不难发现a_1, a_2, a_3完全轮换对称,三者地位相等,优化后三者相等,因此log中$\frac{a_2+a_3}{a_1}$, $\frac{a_1+a_3}{a_2}$, $\frac{a_1+a_2}{a_3}$这三项均等于2,
最后相当于优化下式:

$$= log(3+\frac{b}{a_1}) + log(3+\frac{b}{a_2}) + log(3+\frac{b}{a_3}) \tag{7}$$

这个式子等价于上面本来应该优化的,优化这个公式,可以让a_1, a_2, a_3变大,b_1, b_2, b_3变小.
不过请注意,这只是感性分析,并且是因果倒着讲的,也就是让a_1, a_2, a_3变大,b_1, b_2, b_3变小
确实可以让loss变小, 但是loss变小并不意味着反过来能让a_1, a_2, a_3变大,b_1, b_2, b_3变小.
不过这种隐隐约约的直觉也并非完全不靠谱,最后通过实验发现,优化上面的式子确实可以让
想变大的变大,让想变小的变小,产生了聚类的效果.

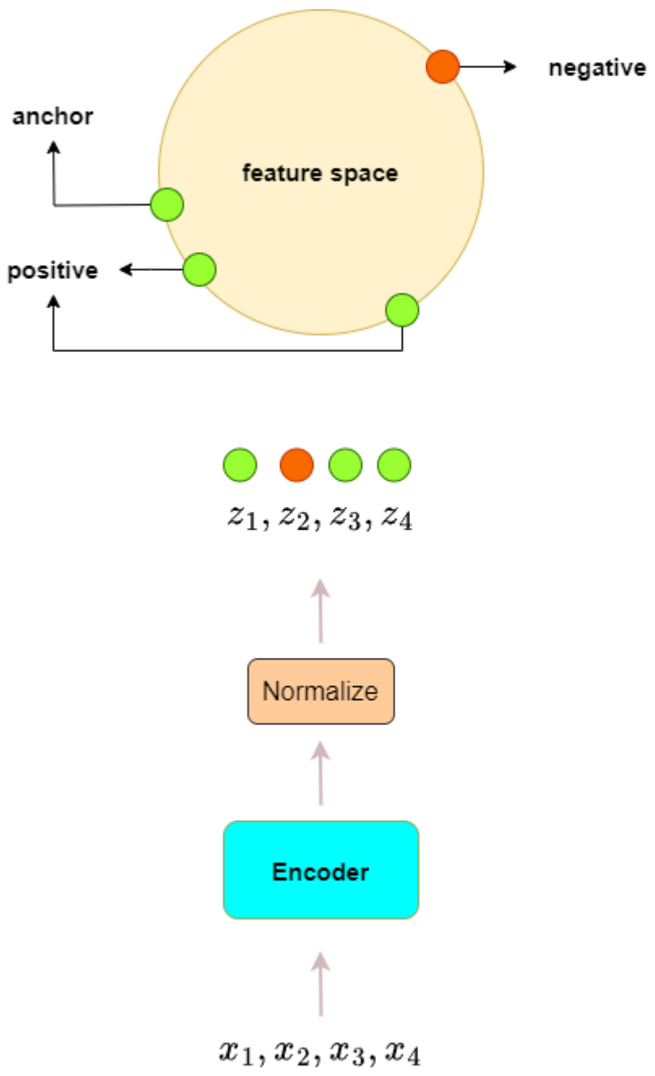
选哪一个?

根据论文Supervised Contrastive Learning中的实验,L_{out}的Top-1准确率比L_{in}高了10个百分点,所以我们后面选L_{out}.

$$\mathcal{L}_{out} = -\sum_{\{i\}}\sum_{\{z^+\}} log\frac{exp(sim(z_i, z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_i, z_p)/\tau)}$$

1.3 有监督对比损失的复现

参考: https://github.com/HobbitLong/SupContrast/blob/master/losses.py
为了说清楚,下面会结合一个实例讲清楚代码是如何实现的,
我们假设当前批次的数据x_1, x_2, x_3, x_4的label分别是1,2,1,1,也就是说其中
3个是一类,其中1个是另一类,那么经过编码后,这4个样本的编码向量的相似度矩阵如下:



$sim(z_1, z_1)$	$sim(z_1, z_2)$	$sim(z_1, z_3)$	$sim(z_1, z_4)$
$sim(z_2, z_1)$	$sim(z_2, z_2)$	$sim(z_2, z_3)$	$sim(z_2, z_4)$
$sim(z_3, z_1)$	$sim(z_3, z_2)$	$sim(z_3, z_3)$	$sim(z_3, z_4)$
$sim(z_4, z_1)$	$sim(z_4, z_2)$	$sim(z_4, z_3)$	$sim(z_4, z_4)$

similarity matrix

原来的公式如下:

$$\mathcal{L}_{out} = \sum_{\{i\}}\sum_{\{z^+\}} -log\frac{exp(sim(z_i, z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_i, z_p)/\tau)}$$

首先我们固定i,看看第i行是怎么求解的:

$$\sum_{\{z^+\}} -log\frac{exp(sim(z_i, z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_i, z_p)/\tau)}$$

还是有点复杂,我们不妨再固定某个正例,看看它是如何求解的:

$$-log\frac{exp(sim(z_i, z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_i, z_p)/\tau)}$$

这个公式就不那么复杂了,就是正例距离除以这一行所有正例负例距离之和,然后外面套个-log,放在矩阵当中批量计算如下:

$sim(z_1,z_1)$	$sim(z_1,z_2)$	$sim(z_1,z_3)$	$sim(z_1,z_4)$
$sim(z_2,z_1)$	$sim(z_2,z_2)$	$sim(z_2,z_3)$	$sim(z_2,z_4)$
$sim(z_3,z_1)$	$sim(z_3,z_2)$	$sim(z_3,z_3)$	$sim(z_3,z_4)$
$sim(z_4,z_1)$	$sim(z_4,z_2)$	$sim(z_4,z_3)$	$sim(z_4,z_4)$

similarity matrix

这个地方 需要掩码	$exp(sim(z_1,z_2))$	$exp(sim(z_1,z_3))$	$exp(sim(z_1,z_4))$
$exp(sim(z_2,z_1))$	0	$exp(sim(z_2,z_3))$	$exp(sim(z_2,z_4))$
$exp(sim(z_3,z_1))$	$exp(sim(z_3,z_2))$	0	$exp(sim(z_3,z_4))$
$exp(sim(z_4,z_1))$	$exp(sim(z_4,z_2))$	$exp(sim(z_4,z_3))$	0

exp(similarity matrix)

$\sum_{\{z^+,z^-\}} exp(sim(z_1,z_p)/\tau)$
$\sum_{\{z^+,z^-\}} exp(sim(z_2,z_p)/\tau)$
$\sum_{\{z^+,z^-\}} exp(sim(z_3,z_p)/\tau)$
$\sum_{\{z^+,z^-\}} exp(sim(z_4,z_p)/\tau)$

每行正例负例求和,求出分母

矩阵每行除以右边的分母

0	这个地方没用,就不写了	$\frac{exp(sim(z_1,z_3))}{\sum_{\{z^+,z^-\}} exp(sim(z_1,z_p)/\tau)}$	$\frac{exp(sim(z_1,z_4))}{\sum_{\{z^+,z^-\}} exp(sim(z_1,z_p)/\tau)}$
这个地方没用,就不写了	0	这个地方没用,就不写了	这个地方没用,就不写了
$\frac{exp(sim(z_3,z_1))}{\sum_{\{z^+,z^-\}} exp(sim(z_3,z_p)/\tau)}$	这个地方没用,就不写了	0	$\frac{exp(sim(z_3,z_4))}{\sum_{\{z^+,z^-\}} exp(sim(z_3,z_p)/\tau)}$
$\frac{exp(sim(z_4,z_1))}{\sum_{\{z^+,z^-\}} exp(sim(z_4,z_p)/\tau)}$	这个地方没用,就不写了	$\frac{exp(sim(z_4,z_3))}{\sum_{\{z^+,z^-\}} exp(sim(z_4,z_p)/\tau)}$	0

从上图可以看出,固定某一行,我们把每个正例距离除以分母都求了出来,就是每行的绿色部分,
然后我们只要这行绿色部分拿出来,套个-log,然后求和,这一行的对比损失就求了出来,然后
每行都这么做,再把每行的对比损失进行求和,总对比损失就求了出来.
另外有一个需要注意的是,我们需要把上面每行正例的对比损失进行求和这步改成求平均,
为什么到这里才提呢,因为式子太长,讲多了容易把注意力放在别的地方.

$$\mathcal{L}_{out}=\sum_{\{i\}}\frac{1}{|\{z^+\}|}\sum_{\{z^+\}}-log\frac{exp(sim(z_i,z^+)/\tau)}{\sum_{\{z^+,z^-\}}exp(sim(z_i,z_p)/\tau)}$$

其中$|\{z^+\}|$表示$\{z^+\}$的基数.

1.4 一个实例带你过一遍前向过程

这里为了方便默认温度系数为1进行计算

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
torch.manual_seed(102)
```

Out[1]:
```
<torch._C.Generator at 0x157b51a9830>
```

创建特征矩阵,假设每个特征向量8维,并且已经归一化了

In [2]:
```
batch_size=4
z=torch.randn(4,8)
label=torch.tensor([1,2,1,1])
z,label
```

Out[2]:
```
(tensor([[ 0.0628,  1.3266,  0.2553, -0.0397,  0.5545,  1.0317,  0.3194,  0.3219],
         [ 0.0953,  1.1049,  0.2073,  1.0528, -1.2678, -0.0253, -0.2052,  0.8111],
         [-0.4605,  0.0396, -2.0155, -0.7121, -1.1587,  0.8486, -1.7099,  0.7249],
         [-1.3338,  0.1465,  0.5515,  0.2964, -0.0833,  1.2038,  2.0049,  0.1087]]),
 tensor([1, 2, 1, 1]))
```

计算similarity matrix,这里用内积代替相似度计算

In [3]:
```
similarity= torch.matmul(z,z.T)
similarity
```

Out[3]:
```
tensor([[ 3.4081,  0.9493, -0.5425,  2.1107],
        [ 0.9493,  4.6893,  1.2189,  0.2131],
        [-0.5425,  1.2189, 10.2947, -2.9339],
        [ 2.1107,  0.2131, -2.9339,  7.6797]])
```

计算几个掩码矩阵,用于筛选后面需要用到的矩阵元素

In [4]:
```
similarity_mask=torch.ones(batch_size,batch_size)-torch.eye(batch_size)
similarity_mask
```

Out[4]:
```
tensor([[0., 1., 1., 1.],
        [1., 0., 1., 1.],
        [1., 1., 0., 1.],
        [1., 1., 1., 0.]])
```

In [5]:
```
positive_mask=(label==label.unsqueeze(0).T).int()*similarity_mask
positive_mask
```

Out[5]:
```
tensor([[0., 0., 1., 1.],
        [0., 0., 0., 0.],
        [1., 0., 0., 1.],
        [1., 0., 1., 0.]])
```

In [6]:
```
negative_mask=(label!=label.unsqueeze(0).T).int()*similarity_mask
negative_mask
```

Out[6]:
```
tensor([[0., 1., 0., 0.],
        [1., 0., 1., 1.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.]])
```

计算exp(similarity matrix),并将对角线元素置0

In [7]:
```
exp_similarity=torch.exp(similarity)*similarity_mask
exp_similarity
```

Out[7]:
```
tensor([[0.0000, 2.5840, 0.5813, 8.2544],
        [2.5840, 0.0000, 3.3834, 1.2375],
        [0.5813, 3.3834, 0.0000, 0.0532],
        [8.2544, 1.2375, 0.0532, 0.0000]])
```

计算每行对比损失的分母

In [8]:
```
denominator=torch.sum(exp_similarity,axis=-1,keepdims=True)
denominator
```

Out[8]:
```
tensor([[11.4197],
        [ 7.2050],
        [ 4.0179],
        [ 9.5451]])
```

把矩阵的每个元素都除以这个分母

In [9]:
```
frac=exp_similarity/denominator
frac
```

Out[9]:
```
tensor([[0.0000, 0.2263, 0.0509, 0.7228],
        [0.3586, 0.0000, 0.4696, 0.1718],
        [0.1447, 0.8421, 0.0000, 0.0132],
        [0.8648, 0.1297, 0.0056, 0.0000]])
```

把每行的正例部分拿出来套个-log再求平均

In [10]:
```
torch.where(positive_mask.bool(),-torch.log(frac),0)
```

Out[10]:
```
tensor([[0.0000, 0.0000, 2.9778, 0.3246],
        [0.0000, 0.0000, 0.0000, 0.0000],
        [1.9332, 0.0000, 0.0000, 4.3246],
        [0.1453, 0.0000, 5.1899, 0.0000]])
```

In [11]:
```
log_probs=torch.sum(torch.where(positive_mask.bool(),-torch.log(frac),0),axis=-1)
log_probs=log_probs/6
log_probs
```

Out[11]:
```
tensor([0.5504, 0.0000, 1.0430, 0.8892])
```

求4个对比损失的和

In [12]:
```
supcon_loss=log_probs.sum()
supcon_loss
```

Out[12]:
```
tensor(2.4826)
```

1.5 使用原文作者的源代码

由于原文作者的github代码和原始的有些不一样,被修改过了,这里给出原始代码,
来源知乎的一位网友: https://zhuanlan.zhihu.com/p/442415516

In [13]:
```
class SupConLoss(nn.Module):
    def __init__(self, temperature=0.5, scale_by_temperature=True):
        super(SupConLoss, self).__init__()
        self.temperature = temperature
        self.scale_by_temperature = scale_by_temperature

    def forward(self, features, labels=None, mask=None):
        # test
        # print('特征形状:',features.shape)
        device = (torch.device('cuda')
                  if features.is_cuda
                  else torch.device('cpu'))
        # 这里把特征归一化删除了
        batch_size = features.shape[0]
        if labels is not None and mask is not None:
            raise ValueError('Cannot define both `labels` and `mask`')
        elif labels is None and mask is None:
            mask = torch.eye(batch_size, dtype=torch.float32).to(device)
        elif labels is not None:
            labels = labels.contiguous().view(-1, 1)
            if labels.shape[0] != batch_size:
                raise ValueError('Num of labels does not match num of features')
            mask = torch.eq(labels, labels.T).float().to(device)
        else:
            mask = mask.float().to(device)
        # compute logits
        similarity=torch.matmul(features, features.T)
        anchor_dot_contrast = torch.div(
            similarity,
            self.temperature)
        # for numerical stability
        logits_max, _ = torch.max(anchor_dot_contrast, dim=1, keepdim=True)
        logits = anchor_dot_contrast - logits_max.detach()
        exp_logits = torch.exp(logits)

        # 检查一些矩阵的数值是否有问题
        if torch.any(torch.isnan(similarity)):
            print("嵌入向量:",features)
            print("相似度矩阵:",similarity)
```

```
        raise ValueError("similarity has nan!")

    if torch.any(torch.isnan(anchor_dot_contrast)):
        raise ValueError("anchor_dot_contrast has nan!")

    if torch.any(torch.isnan(logits)):
        raise ValueError("logits has nan!")

    # mask
    logits_mask = torch.ones_like(mask) - torch.eye(batch_size)
    positives_mask = mask * logits_mask
    negatives_mask = 1. - mask

    num_positives_per_row  = torch.sum(positives_mask , axis=1) # 除了自己之外，正样本的个数  [2 0 2 2]
    denominator = torch.sum(
    exp_logits * negatives_mask, axis=1, keepdims=True) + torch.sum(
        exp_logits * positives_mask, axis=1, keepdims=True)
    log_probs = logits - torch.log(denominator)

    # 检查一些矩阵的数值是否有问题
    if torch.any(torch.isnan(denominator)):
        raise ValueError("denominator has nan!")

    if torch.any(torch.isnan(log_probs)):
        raise ValueError("Log_prob has nan!")

    # 收尾计算
    log_probs = torch.sum(
        log_probs*positives_mask , axis=1)[num_positives_per_row > 0] / num_positives_per_row[num_positives_per_row > 0]
    # loss
    loss = -log_probs
    if self.scale_by_temperature:
        loss *= self.temperature
    loss = loss.mean()

    # 一些测试
    # print(logits_mask)
    # print(positives_mask)
    # print(negatives_mask)

    return loss
```

In [14]:
```
loss_fn=SupConLoss(temperature=1)
loss_fn(z,labels=label)
```

Out[14]:
```
tensor(2.4826)
```

代码比较与分析

第一处不同:
logits = anchor_dot_contrast - logits_max.detach(),
这里anchor_dot_contrast就是我们的similarity matrix,
similarity matrix的每一行都减去了这以后的最大值,这个
做法是为了数值稳定性,对结果没有影响,拿第一行为例,证明如下:

$$-log\frac{exp(sim(z_1,z^+)/\tau - C)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_1,z_p)/\tau - C)} = -log\frac{exp(sim(z_1,z^+)/\tau) * exp(C)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_1,z_p)/\tau) * exp(C)} = -log\frac{exp(sim(z_i,z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_i,z_p)/\tau)}$$

为什么这样就数值稳定了呢,因为exp里面不能放太大的数,否则指数爆炸,数值溢出.

第二处不同:
log_probs = logits - torch.log(denominator), 换成我们的符号就是:
log_probs = similarity - torch.log(denominator)
相当于我们的这一步:
frac=log(exp_similarity/denominator)
先证明两者基本一样,再说明我们为什么没有这样做:

$$log\frac{exp(sim(z_i,z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}} exp(sim(z_i,z_p)/\tau)} = log(exp(sim(z_i,z^+)/\tau)) - log\sum_{\{z^+,z^-\}} exp(sim(z_i,z_p)/\tau) \tag{8}$$

$$= sim(z_i,z^+)/\tau - log\sum_{\{z^+,z^-\}} exp(sim(z_i,z_p)/\tau) = similarity - log(denominator) \tag{9}$$

我们没有这么做的原因是,我们的exp_similarity矩阵为了每一行求和求出分母,掩码了,对角线元素变成0了,
log(exp_similarity/denominator)会出现log0,所以我们没有这么做.
当然,如果不按照我们的思路,按照作者的思路,也就是similarity - torch.log(denominator)也可以求出所有
正例的对比损失,殊途同归,最后结果一样,只不过我给的思路更容易理解,并且上面画了图进行解释.
既然结果一样,我们就不重新写自己的损失函数了,有兴趣的把我上面的步骤整合一下也能轻松写出来,我们后面
直接拿作者的损失函数进行实验.

1.6 使用对比损失对鸢尾花分类

为了简单验证上面的损失函数是否work,下面直接拿最简单最常用的鸢尾花数据集进行简单实验,
看看是否真的行得通,我们设计实验的思路也比较简单,就拿几个全连接层作为编码层,然后进行
对比损失训练,接着固定编码层,加个分类头,微调分类头进行有监督训练,看看编码器得到的特征是否有效.
由于这种end-to-end的学习方式,需要特征一致性,所以我们的batch size设置大一些.

In [15]:
```
import sys
sys.path.append('..')
from runner_for_contrast import Runner,pre_train
from metrics_for_contrast import Accuracy
```

构造训练集和测试集

In [16]:
```
from torch.utils.data import DataLoader, Dataset
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# 下载鸢尾花数据集
iris = load_iris()

# 将特征矩阵和标签向量拆分为训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.3, random_state=123)

# 标准化特征矩阵
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# 创建自定义的 Dataset 类
```

```python
class IrisDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.tensor(X, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.long)

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

In [17]:
```python
# 创建训练集和测试集的 Dataset 对象
train_dataset = IrisDataset(X_train, y_train)
test_dataset = IrisDataset(X_test, y_test)

# 创建 DataLoader 对象用于批处理和迭代数据
batch_size = 16
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

In [18]:
```python
# test
for x,labels in train_loader:
    print(x.shape,labels.shape)
```

```
torch.Size([16, 4]) torch.Size([16])
torch.Size([16, 4]) torch.Size([16])
torch.Size([16, 4]) torch.Size([16])
torch.Size([16, 4]) torch.Size([16])
torch.Size([16, 4]) torch.Size([16])
torch.Size([16, 4]) torch.Size([16])
torch.Size([9, 4]) torch.Size([9])
```

构造编码器

In [19]:
```python
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, embed_dim):
        super(Encoder, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, embed_dim)

    def forward(self, x):
        out = self.fc1(x)
        out = F.tanh(out)
        out = self.fc2(out)
        return out
```

In [20]:
```python
input_size = 4
hidden_size = 10
embed_dim = 6

encoder1 = Encoder(input_size, hidden_size, embed_dim)
```

初始化损失函数和优化器

In [21]:
```python
loss_fn1a=SupConLoss(temperature=0.1)
opt1a=optim.SGD(encoder1.parameters(),lr=1)
```

使用对比损失训练编码器

In [22]:
```python
pre_train(encoder1,loss_fn1a,opt1a,train_loader,num_epochs=512,log_stride=256)
```

```
[Train]  epoch:0/512 step:0/3584 loss:0.2984
[Train]  epoch:36/512 step:256/3584 loss:0.1615
[Train]  epoch:73/512 step:512/3584 loss:0.1590
[Train]  epoch:109/512 step:768/3584 loss:0.1599
[Train]  epoch:146/512 step:1024/3584 loss:0.1577
[Train]  epoch:182/512 step:1280/3584 loss:0.0887
[Train]  epoch:219/512 step:1536/3584 loss:0.1617
[Train]  epoch:256/512 step:1792/3584 loss:0.1555
[Train]  epoch:292/512 step:2048/3584 loss:0.1659
[Train]  epoch:329/512 step:2304/3584 loss:0.1736
[Train]  epoch:365/512 step:2560/3584 loss:0.1510
[Train]  epoch:402/512 step:2816/3584 loss:0.1713
[Train]  epoch:438/512 step:3072/3584 loss:0.1141
[Train]  epoch:475/512 step:3328/3584 loss:0.1578
[Train]  epoch:511/512 step:3583/3584 loss:0.1143
```

定义模型

In [23]:
```python
class Net(nn.Module):
    def __init__(self,encoder,num_classes):
        super(Net, self).__init__()
        self.encoder=encoder
        self.fc = nn.Linear(embed_dim,num_classes)  # 直接使用特征进行线性分类
    def forward(self, x):
        out = self.encoder(x).detach()  #  编码器部分冻结
        out = self.fc(out)
        return out
```

In [24]:
```python
model1=Net(encoder1,3)
```

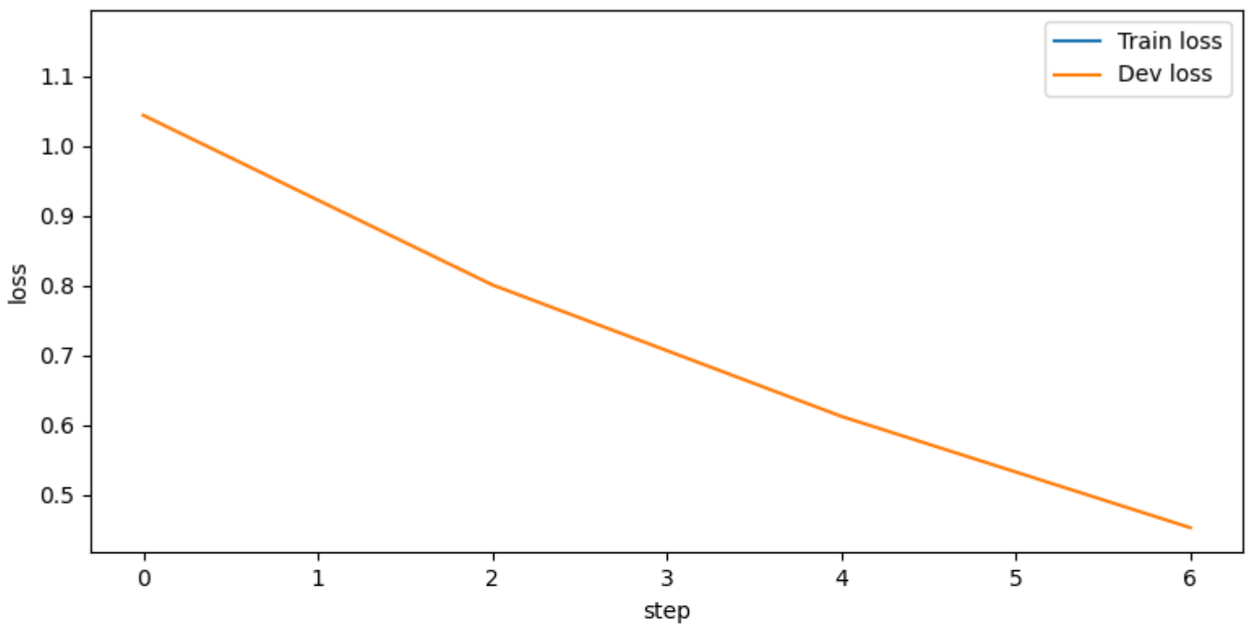
初始化损失函数,优化器,评估器

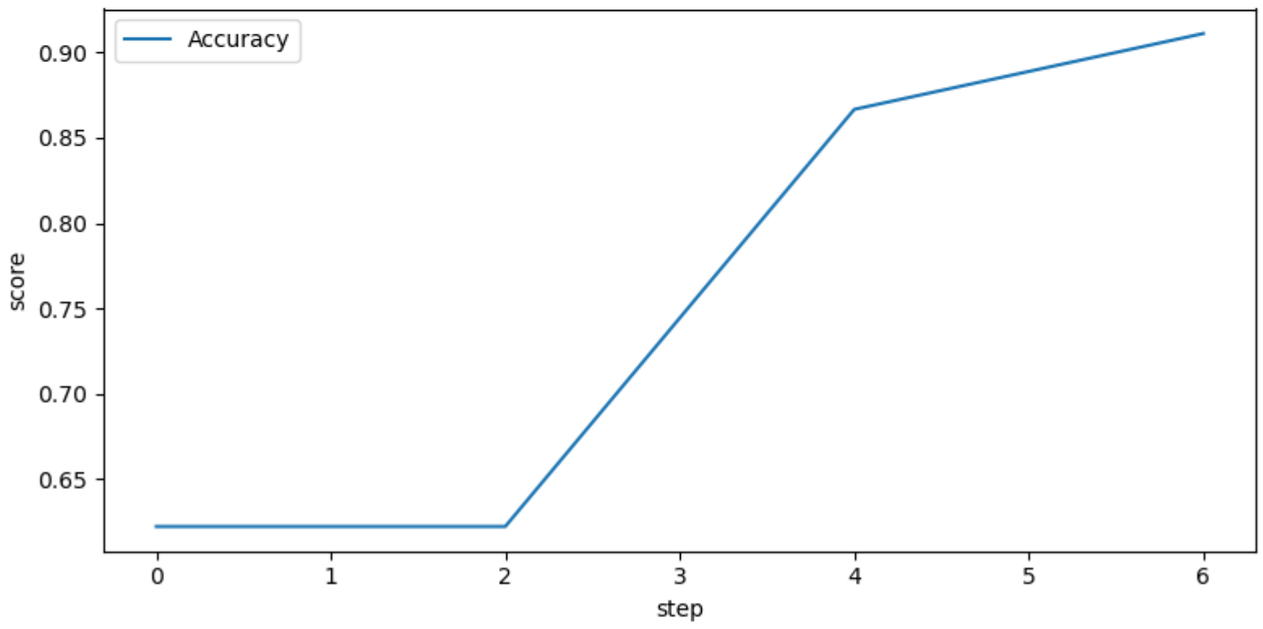
In [25]:
```python
metric1=Accuracy()
loss_fn1b=nn.CrossEntropyLoss()
opt1b=optim.Adam(model1.parameters(),lr=0.1)
```

开始微调

In [26]:
```python
runner1=Runner(model=model1,loss_fn=loss_fn1b,optimizer=opt1b,metric=metric1)
runner1.train(train_loader=train_loader,dev_loader=test_loader,num_epochs=1,log_stride=2)
```

```
[Train]  epoch:0/1 step:0/7 loss:1.1598
[Evaluate]  score:0.6222 loss:1.0442
[Train]  epoch:0/1 step:2/7 loss:0.9163
[Evaluate]  score:0.6222 loss:0.8010
[Train]  epoch:0/1 step:4/7 loss:0.7337
[Evaluate]  score:0.8667 loss:0.6123
[Train]  epoch:0/1 step:6/7 loss:0.5447
[Evaluate]  score:0.9111 loss:0.4526
```



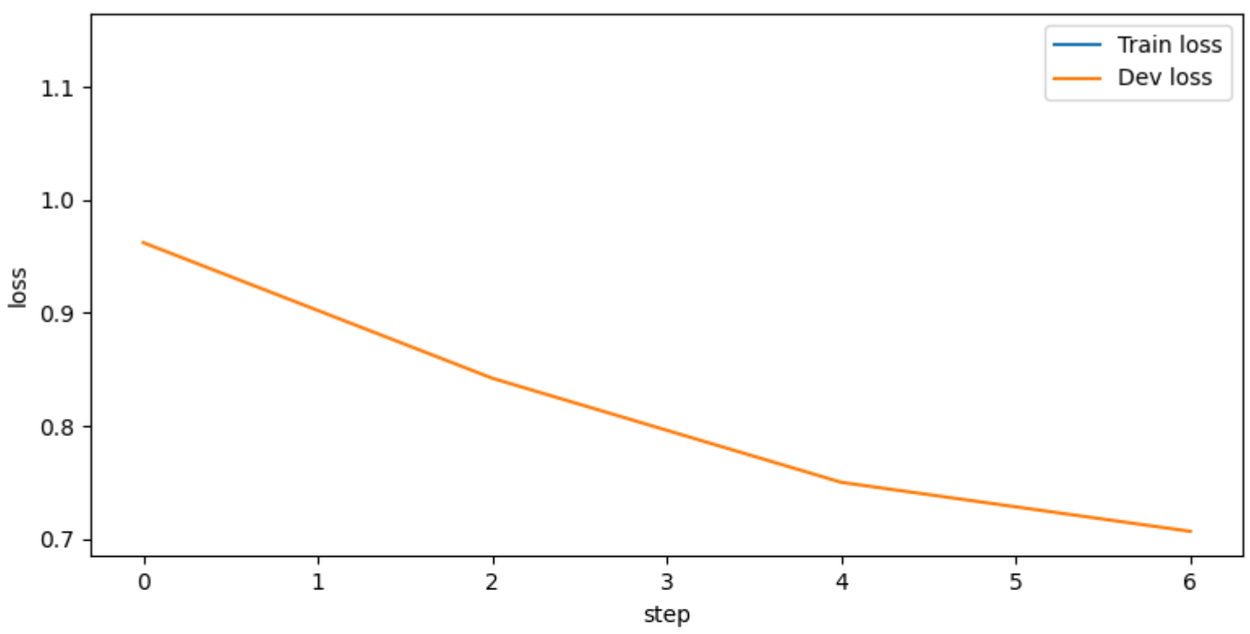


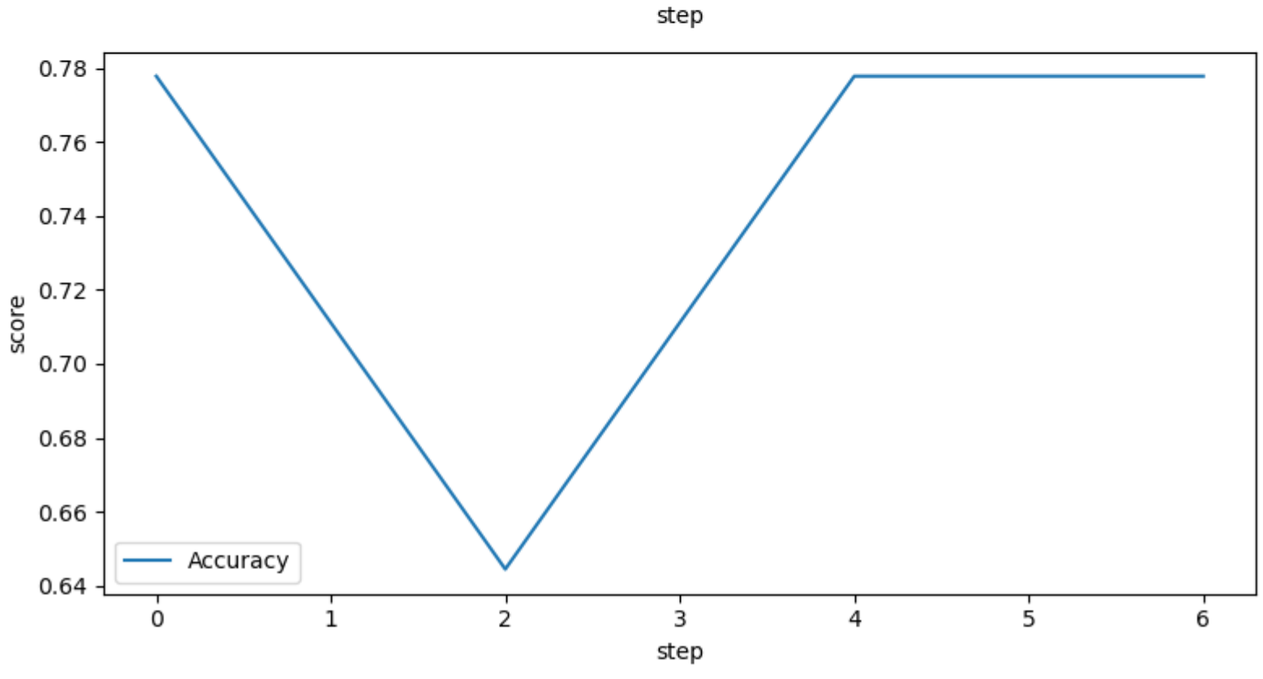
使用随机初始化的Encoder来训练分类头

注意,由于这个任务过于简单,如果训练全部的参数,会导致一下子就训练成功
这里还是要冻结编码器,主要是为了看看单独训练分类头的情况下,两者的编码器拿个更好

In [27]:
```
encoder2 = Encoder(input_size, hidden_size, embed_dim)   # 编码器
model2=Net(encoder2,3)                                   # 模型
loss_fn2=nn.CrossEntropyLoss()                           # 损失
opt2=optim.Adam(model2.parameters(),lr=0.1)              # 优化器
metric2=Accuracy()                                       # 评估器

runner2=Runner(model=model2,loss_fn=loss_fn2,optimizer=opt2,metric=metric2)
runner2.train(train_loader=train_loader,dev_loader=test_loader,num_epochs=1,log_stride=2)
```

```
[Train]  epoch:0/1 step:0/7 loss:1.1429
[Evaluate]  score:0.7778 loss:0.9621
[Train]  epoch:0/1 step:2/7 loss:0.9160
[Evaluate]  score:0.6444 loss:0.8422
[Train]  epoch:0/1 step:4/7 loss:1.0362
[Evaluate]  score:0.7778 loss:0.7501
[Train]  epoch:0/1 step:6/7 loss:0.9279
[Evaluate]  score:0.7778 loss:0.7068
```





1.7 使用对比损失进行手写数字识别

由于上面的任务过于简单,也就是说即使编码器什么都没学到,然后训练也一样可以成功,
接下来用一个稍微复杂的数据集Mnist进行实验验证前面的想法是否正确

In [28]:
```
import torchvision
import torchvision.transforms as transforms
```

导入数据集

In [29]:
```
transform = transforms.Compose([
    transforms.ToTensor(),  # 将图像转换为张量
    transforms.Normalize((0.5,), (0.5,))  # 标准化图像到范围 [-1, 1]
])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=False, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False)
```

In [30]:
```
for x,labels in trainloader:
    print(x.shape,labels.shape)
    break
```

```
torch.Size([256, 1, 28, 28]) torch.Size([256])
```

定义编码器和网络

In [31]:
```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, 3, padding=1)
        self.relu = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)

    def forward(self, x):
        x = self.pool1(self.relu(self.conv1(x)))
        x = self.pool2(self.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)
        x = self.relu(self.fc1(x))
        return x

class Net(nn.Module):
    def __init__(self,encoder):
        super(Net, self).__init__()
        self.encoder=encoder
        self.fc = nn.Linear(128, 10)

    def forward(self, x):
        x = self.encoder(x).detach()
        x = self.fc(x)
        return x
```

使用对比学习预训练编码器

In [32]:
```
encoder1=Encoder()
```

In [33]:
```
loss_fn1a=SupConLoss(temperature=10)
opt1a=optim.SGD(encoder1.parameters(),lr=0.01)
```

In [34]:
```
# 预训练,如果前面训练过一次,这里直接加载
# pre_train(encoder1,loss_fn1a,opt1a,trainloader,num_epochs=32,log_stride=512)

# 以防万一,保存编码器
# torch.save(encoder1.state_dict(), 'encoder1.pt')

# 加载模型
state_dict=torch.load('encoder1.pt')
encoder1.load_state_dict(state_dict)
```

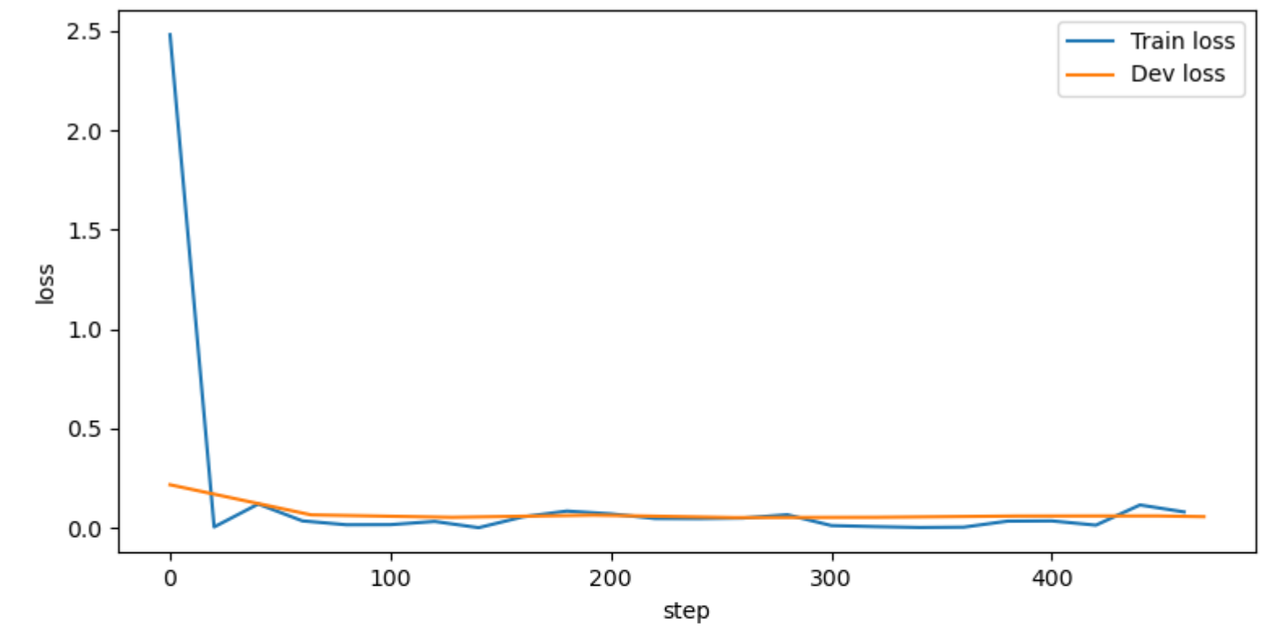
Out[34]:
```
<All keys matched successfully>
```

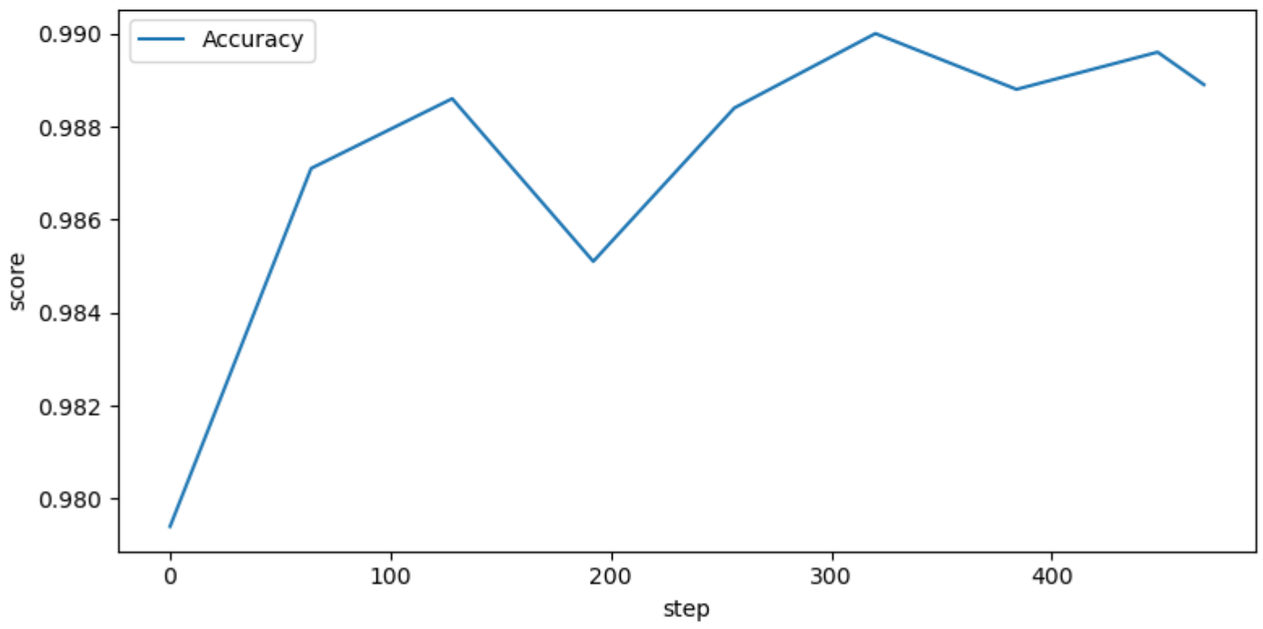
构建并微调训练模型

In [35]:
```
model1=Net(encoder1)
metric1=Accuracy()
loss_fn1b=nn.CrossEntropyLoss()
opt1b=optim.Adam(model1.parameters(),lr=0.1)
```

In [36]:
```
runner1=Runner(model=model1,loss_fn=loss_fn1b,optimizer=opt1b,metric=metric1)
runner1.train(train_loader=trainloader,dev_loader=testloader,num_epochs=2,log_stride=64)
```

```
[Train]  epoch:0/2 step:0/470 loss:2.4814
[Evaluate]  score:0.9794 loss:0.2163
[Train]  epoch:0/2 step:64/470 loss:0.0002
[Evaluate]  score:0.9871 loss:0.0647
[Train]  epoch:0/2 step:128/470 loss:0.0230
[Evaluate]  score:0.9886 loss:0.0524
[Train]  epoch:0/2 step:192/470 loss:0.0222
[Evaluate]  score:0.9851 loss:0.0633
[Train]  epoch:1/2 step:256/470 loss:0.0032
[Evaluate]  score:0.9884 loss:0.0510
[Train]  epoch:1/2 step:320/470 loss:0.0054
[Evaluate]  score:0.9900 loss:0.0525
[Train]  epoch:1/2 step:384/470 loss:0.1400
[Evaluate]  score:0.9888 loss:0.0589
[Train]  epoch:1/2 step:448/470 loss:0.0093
[Evaluate]  score:0.9896 loss:0.0594
[Train]  epoch:1/2 step:469/470 loss:0.1067
[Evaluate]  score:0.9889 loss:0.0556
```



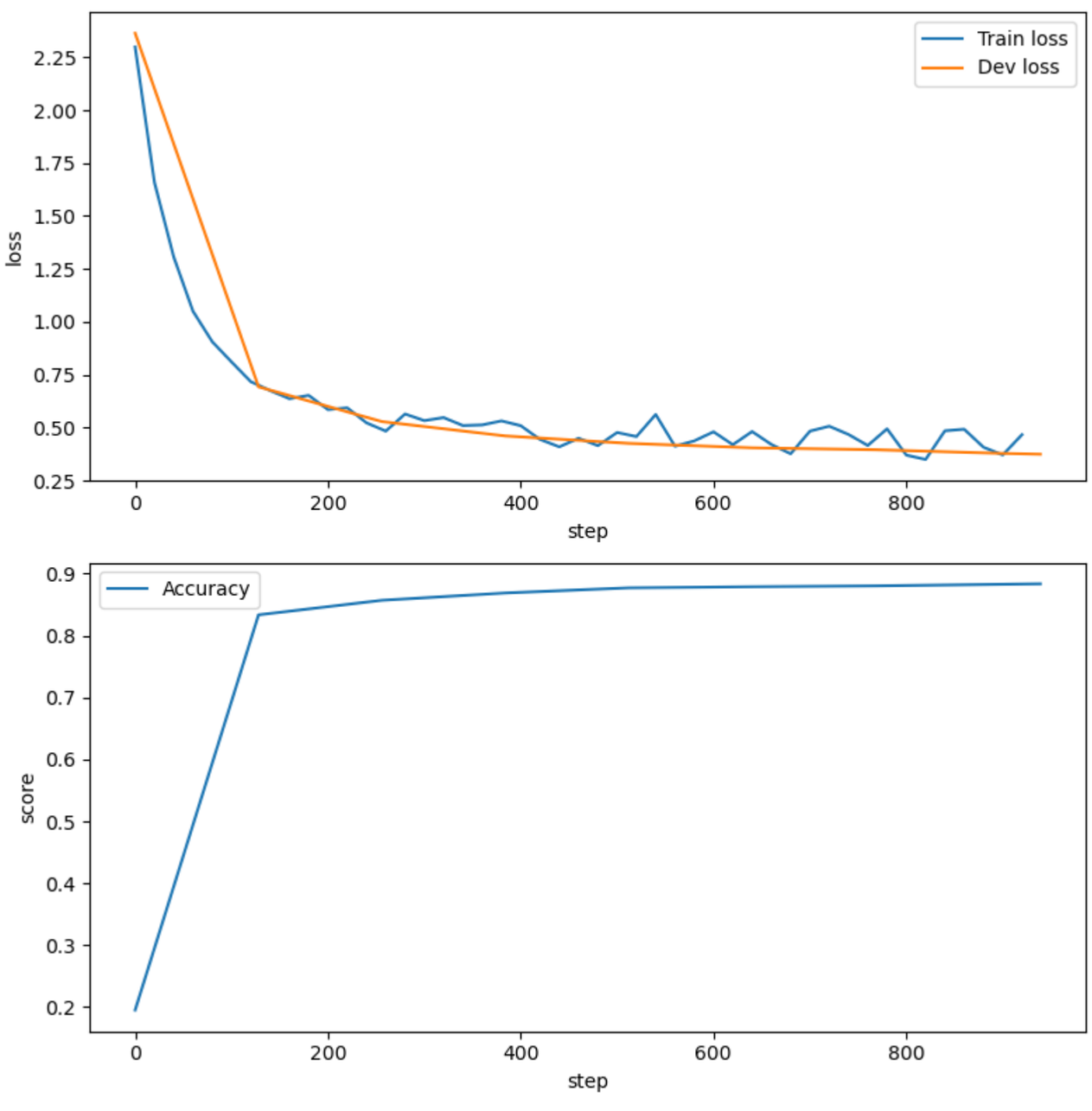


不使用对比学习进行训练和识别

直接随机初始化编码器,然后冻结随机初始化的特征,只训练分类头,看看
随机初始化的特征有没有用

In [37]:
```
encoder2=Encoder()
model2=Net(encoder2)
metric2=Accuracy()
loss_fn2=nn.CrossEntropyLoss()
opt2=optim.Adam(model2.parameters(),lr=0.1)
runner2=Runner(model=model2,loss_fn=loss_fn2,optimizer=opt2,metric=metric2)
runner2.train(train_loader=trainloader,dev_loader=testloader,num_epochs=4,log_stride=128)
```

```
[Train]  epoch:0/4 step:0/940 loss:2.2985
[Evaluate]  score:0.1953 loss:2.3634
[Train]  epoch:0/4 step:128/940 loss:0.8011
[Evaluate]  score:0.8335 loss:0.6930
[Train]  epoch:1/4 step:256/940 loss:0.4696
[Evaluate]  score:0.8570 loss:0.5288
[Train]  epoch:1/4 step:384/940 loss:0.5405
[Evaluate]  score:0.8688 loss:0.4610
[Train]  epoch:2/4 step:512/940 loss:0.3776
[Evaluate]  score:0.8771 loss:0.4260
[Train]  epoch:2/4 step:640/940 loss:0.4819
[Evaluate]  score:0.8789 loss:0.4049
[Train]  epoch:3/4 step:768/940 loss:0.4267
[Evaluate]  score:0.8803 loss:0.3959
[Train]  epoch:3/4 step:896/940 loss:0.4247
[Evaluate]  score:0.8828 loss:0.3790
[Train]  epoch:3/4 step:939/940 loss:0.3381
[Evaluate]  score:0.8836 loss:0.3747
```



结果分析:
1.即便冻结编码器,随机初始化的编码器照样可以提取到有用的特征,这是CNN天生具备提取图像特征所决定的;
2.另一方面,从训练了6个epoch准确率封顶90%来看,这个随机初始化的特征还是比不上对比学习学到的特征,说明对比学习确实学到了一些特征;
3.即便对比学习能学到特征,这并不足以说明对比学习比有监督学习好,因为上面的有监督方法把编码器冻结了,如果不冻结,可能效果更好.

t-SNE可视化对比学习学到的特征

我们把对比学习学到的特征进行可视化展示

In [38]:
```
import numpy as np
import sklearn
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
```

In [39]:
```
# 将所有样本放入一个张量中
all_samples = torch.Tensor()
all_labels = torch.Tensor()
for batch_samples, _ in testloader:
    all_samples = torch.cat((all_samples, batch_samples), dim=0)
    all_labels  = torch.cat((all_labels,_),dim=0)
print(all_samples.shape,all_labels.shape)
```

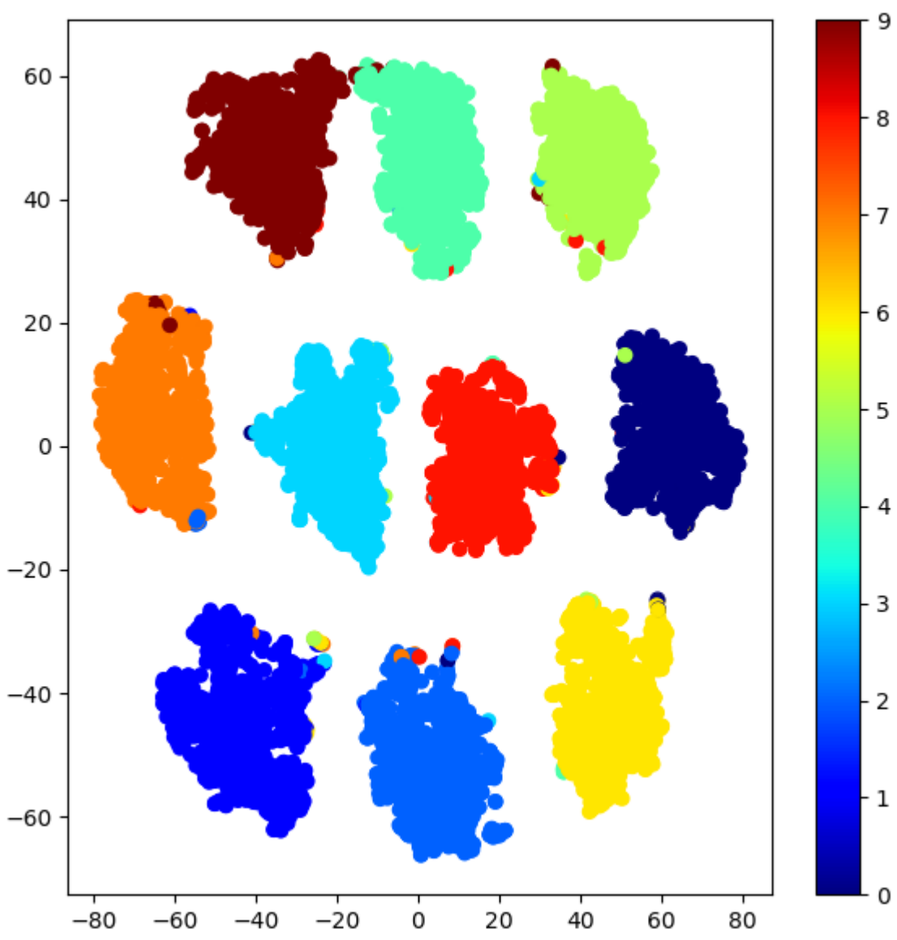
```
torch.Size([10000, 1, 28, 28]) torch.Size([10000])
```

In [40]:
```
# 提取特征
features = encoder1(all_samples).detach().numpy()
labels = all_labels.numpy()
print(features.shape,labels.shape)
```

```
(10000, 128) (10000,)
```

In [41]:
```
# 特征降维
tsne = TSNE(n_components=2, random_state=42)
features_tsne = tsne.fit_transform(features)
```

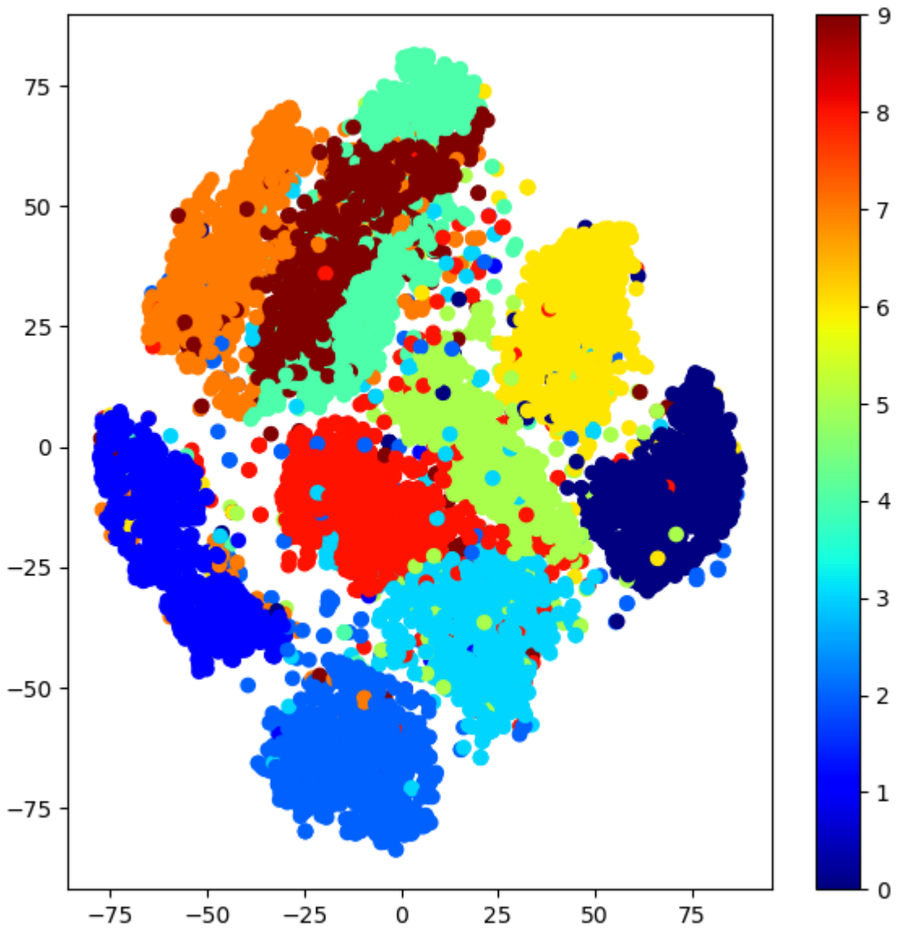
In [42]:
```
plt.figure(figsize=(7, 7))
plt.scatter(features_tsne[:, 0], features_tsne[:, 1], c=labels, cmap='jet')
plt.colorbar()
plt.show()
```



使用随机初始化的encoder进行可视化

根据前面的训练过程可知,随机初始化的编码器固定后,微调分类头也能取得不错的结果,因此可以断定随机初始化的编码器也能提取比较不错的特征,这点上面也分析过了,CNN天然具备提取图像特征的能力,尤其是Mnist数据集不大,随机初始化的CNN就能直接提取特征.

In [43]:
```
random_encoder=Encoder()
features = random_encoder(all_samples).detach().numpy()
tsne = TSNE(n_components=2, random_state=42)
features_tsne = tsne.fit_transform(features)
plt.figure(figsize=(7, 7))
plt.scatter(features_tsne[:, 0], features_tsne[:, 1], c=labels, cmap='jet')
plt.colorbar()
plt.show()
```



1.8 改用全连接前馈神经网络重新实验

从上面可以看出,编码器用CNN好像不怎么能说明对比学习能实实在在学到特征,
因为随机初始化的效果也不差,万一编码器一开始很糟糕,对比学习就学不成功了呢.

重新定义dataset,里面需要把图片拉平后再放到dataloader中

In [44]:
```
import torchvision.datasets as datasets

transform = transforms.Compose([
    transforms.ToTensor(),  # 将图像转换为张量
    transforms.Normalize((0.5,), (0.5,)),  # 标准化图像到范围 [-1, 1]
    transforms.Lambda(lambda x: x.view(784))  # 将图像展平为784维向量
])

trainset = datasets.MNIST(root='./data', train=True, download=False, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True)

testset = datasets.MNIST(root='./data', train=False, download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False)
```

In [45]:
```
for x,_ in trainloader:
    print(x.shape,_.shape)
    break
```

```
torch.Size([256, 784]) torch.Size([256])
```

使用下面这个全连接前馈神经网络作为编码器

In [46]:
```
class FeedForwardNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(FeedForwardNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, hidden_size)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

使用对比学习预训练,再微调

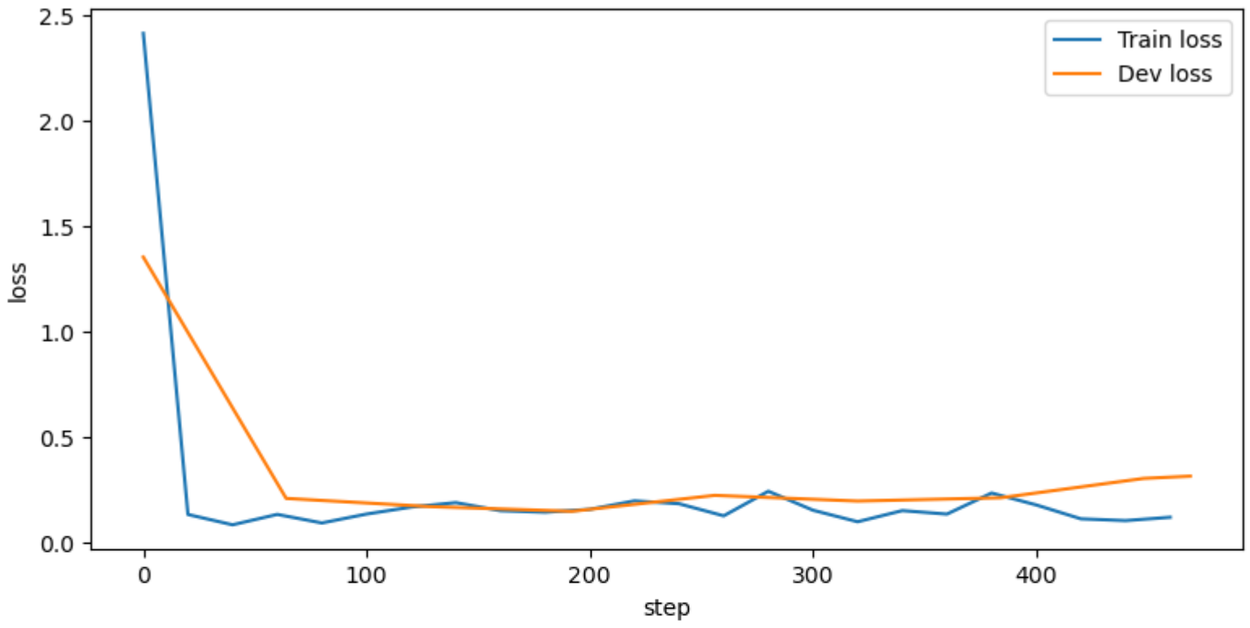
In [47]:
```
fnn_encoder1=FeedForwardNN(784,128)
loss_fn1a=SupConLoss(temperature=10)
opt1a=optim.SGD(fnn_encoder1.parameters(),lr=0.01)
```

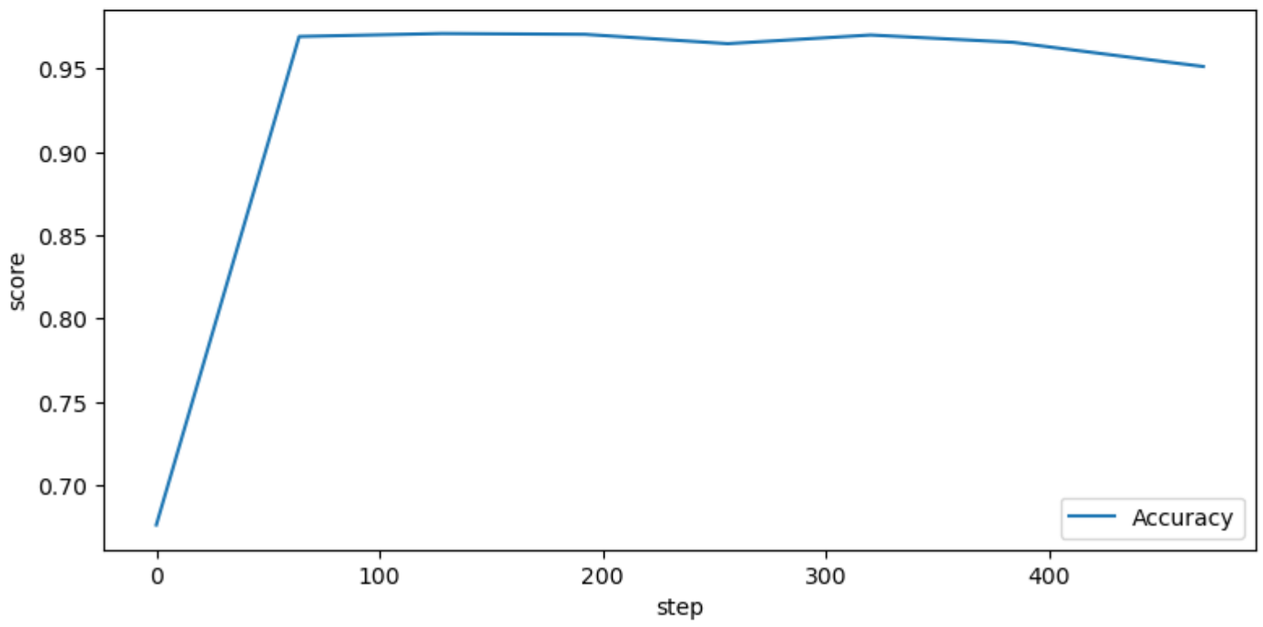
In [48]:
```
# 预训练,如果前面训练过一次,直接加载
# pre_train(fnn_encoder1,loss_fn1a,opt1a,trainloader,num_epochs=16,log_stride=512)

# 以防万一,保存编码器
# torch.save(fnn_encoder1.state_dict(), 'fnn_encoder1.pt')

# 加载模型
state_dict=torch.load('fnn_encoder1.pt')
fnn_encoder1.load_state_dict(state_dict)
```

Out[48]:
```
<All keys matched successfully>
```

In [49]:
```
model1=Net(fnn_encoder1)
metric1=Accuracy()
loss_fn1b=nn.CrossEntropyLoss()
opt1b=optim.Adam(model1.parameters(),lr=0.1)

runner1=Runner(model=model1,loss_fn=loss_fn1b,optimizer=opt1b,metric=metric1)
runner1.train(train_loader=trainloader,dev_loader=testloader,num_epochs=2,log_stride=64)
```

```
[Train]  epoch:0/2 step:0/470 loss:2.4149
[Evaluate]  score:0.6761 loss:1.3546
[Train]  epoch:0/2 step:64/470 loss:0.2123
[Evaluate]  score:0.9693 loss:0.2108
[Train]  epoch:0/2 step:128/470 loss:0.1596
[Evaluate]  score:0.9711 loss:0.1723
[Train]  epoch:0/2 step:192/470 loss:0.1701
[Evaluate]  score:0.9706 loss:0.1498
[Train]  epoch:1/2 step:256/470 loss:0.1505
[Evaluate]  score:0.9650 loss:0.2253
[Train]  epoch:1/2 step:320/470 loss:0.1002
[Evaluate]  score:0.9702 loss:0.1983
[Train]  epoch:1/2 step:384/470 loss:0.0835
[Evaluate]  score:0.9658 loss:0.2137
[Train]  epoch:1/2 step:448/470 loss:0.1852
[Evaluate]  score:0.9548 loss:0.3054
[Train]  epoch:1/2 step:469/470 loss:0.1117
[Evaluate]  score:0.9513 loss:0.3162
```



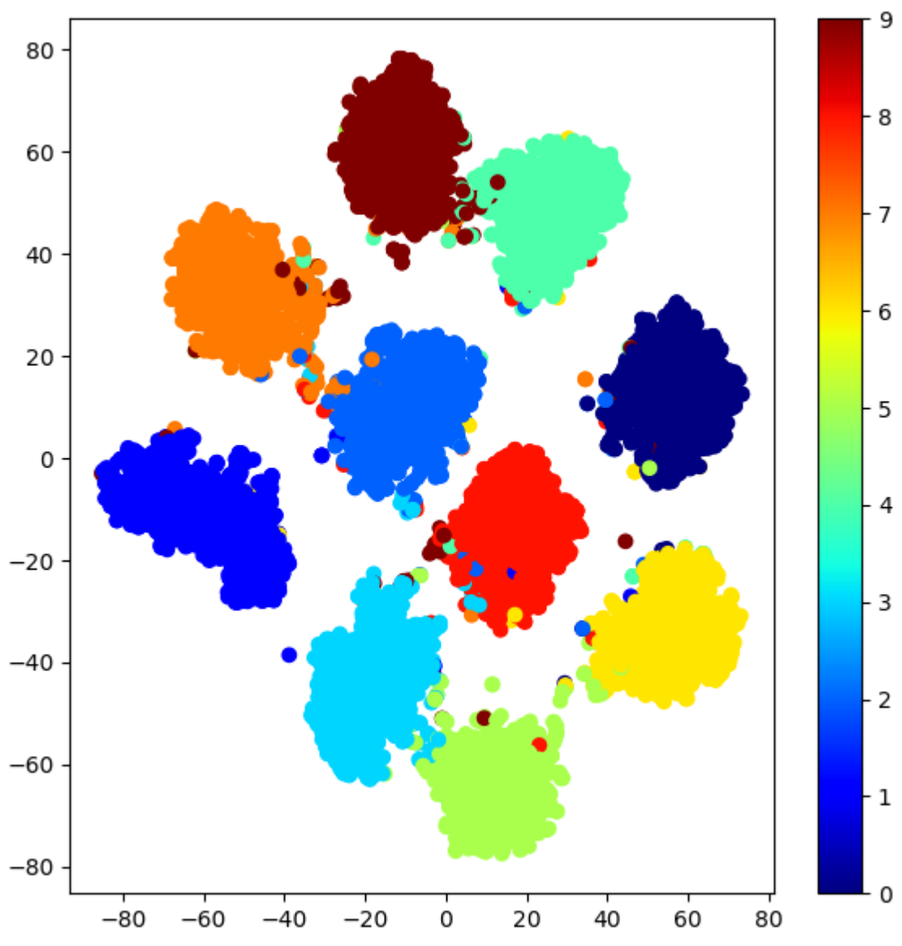


tSNE可视化(对比学习的特征)

In [50]:
```
# 将所有样本放入一个张量中
all_samples = torch.Tensor()
all_labels = torch.Tensor()
for batch_samples, _ in testloader:
    all_samples = torch.cat((all_samples, batch_samples), dim=0)
    all_labels  = torch.cat((all_labels,_),dim=0)
print(all_samples.shape,all_labels.shape)
```

```
torch.Size([10000, 784]) torch.Size([10000])
```

In [51]:
```
features = fnn_encoder1(all_samples).detach().numpy()
tsne = TSNE(n_components=2, random_state=42)
features_tsne = tsne.fit_transform(features)
plt.figure(figsize=(7, 7))
plt.scatter(features_tsne[:, 0], features_tsne[:, 1], c=labels, cmap='jet')
plt.colorbar()
plt.show()
```

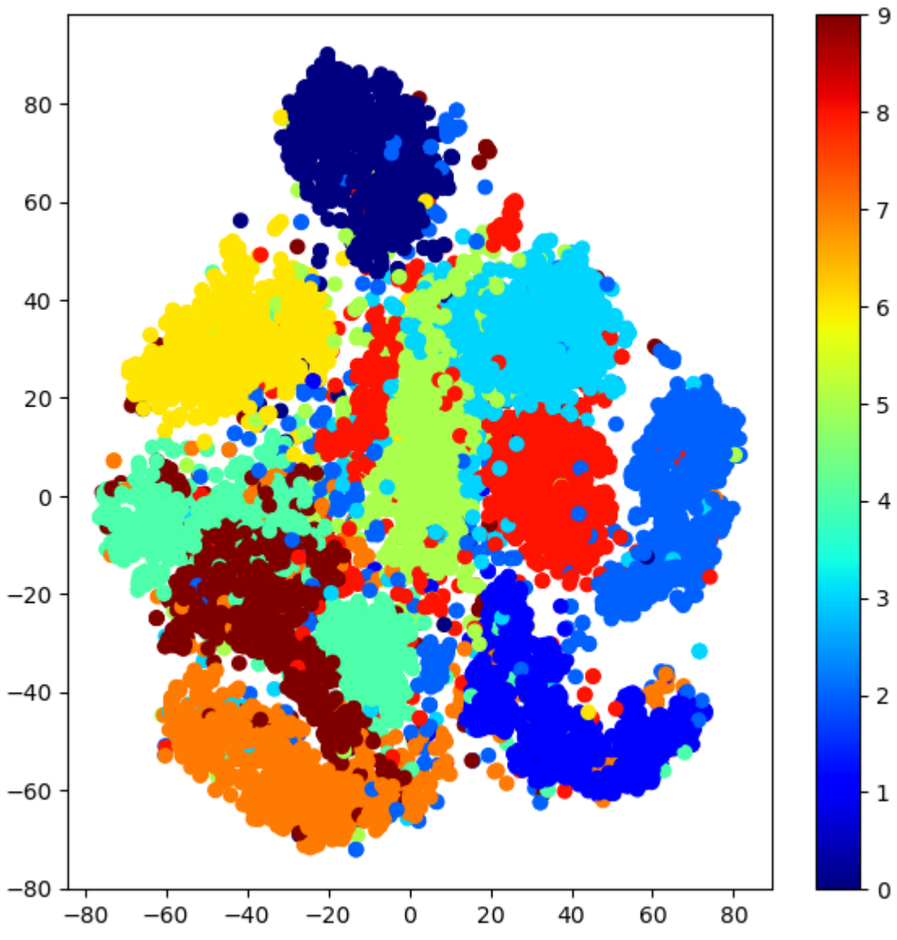


tSNE可视化(随机初始化的特征)

从下图可以看出来,好像还是很不错,连前馈神经网络都能自动提取特征吗, 神经网络不训练都这么强的吗,靠.

看开还是Mnist这个数据集过于简单,接下来就不固定随机编码器训练了,从下图就可以看出来肯定不会差

In [52]:
```
rand_fnn_encoder=FeedForwardNN(784,128)
features = rand_fnn_encoder(all_samples).detach().numpy()
tsne = TSNE(n_components=2, random_state=42)
features_tsne = tsne.fit_transform(features)
plt.figure(figsize=(7, 7))
plt.scatter(features_tsne[:, 0], features_tsne[:, 1], c=labels, cmap='jet')
plt.colorbar()
plt.show()
```



1.9 SimCLR的工作机制

- 首先encoder_q和encoder_k可以是两个不同的编码器，通过对比损失训练，两个编码器

对样本的嵌入应该满足：同类相吸，异类相斥（否则损失函数数值就很大，矛盾）
然后这两个编码器自己对样本编码出来的向量也满足这个特性，这是为什么呢?

- 假设我们的编码器已经训练好了,并且编码器Q这边包含1，2，3三类：

首先给它在空间中一个可能的嵌入，比如下图中的深绿球，接着编码器K中也包含1，2，3三类，
那么它的嵌入只能在深绿求旁边，暂且假设为浅绿球，然后编码器Q对样本2进行嵌入，假设它还是浅绿球，
我们假设第2类的样本经过编码后与第1类样本距离很近，看看会怎么样，然后我们对编码器K这边的第2类
样本进行编码，发现它的位置放在哪里都不符合对比损失训练好后的结果，它是第2类，所以要和前面同为第2类
的query足够近，我们假设query2也是浅绿球了，那么k2只能也在浅绿球旁，但是k2和q1不是同一类，
根据对比损失优化的结果，k2和q1的距离一定不会很近，这就矛盾了。所以q2假设离绿球近是不可能的，
因此q2只能离绿球远，说明对于编码器Q来说，q1和q2只要不是同一类，它们就很远，这就解释了
上面一开始说的，这个同类相吸，异类相斥的特性对于单个编码器也成立。

- 我们训练编码器时，每次都是在同一个batch上做，最后这个编码器能够分别在每个批次的样本上面

满足"同类相吸，异类相斥"的特性，但是我们不清楚第1批样本和第2批样本放在一起是否还具备这个特性，
因为它们每次拉近拉远只在局部调整，并不知道它们与batch之外的那些样本有什么关系，但是我们可以
模糊猜到的一点就是，如果原本的同类样本服从同一分布，那么它们即便不在同一个批次，它们经过相同
的编码器之后得到的特征也相近；

- 由此可见，上面我们分析样本经过编码器后特征是否相似，是从两个维度分析的：

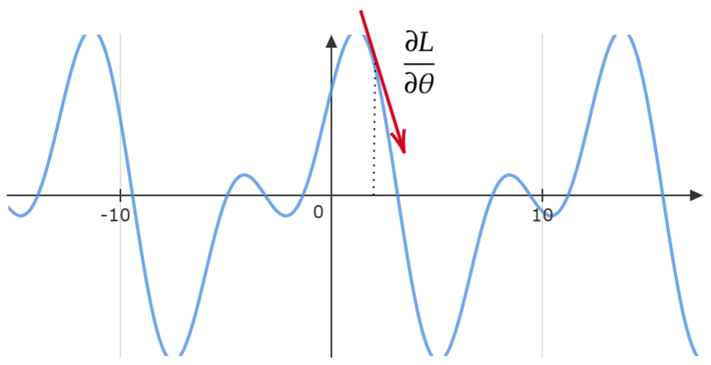
纵向上，不同批次的样本，它们的编码器来自同一个，所以分布相同特征就相似，同一个编码器保证了
纵向的特征一致性，如果使用不同的编码器，纵向上的特征就没法对比；
虽然优化过程中，纵向的编码器不同，但是经过编码器不停进化，最后的那个能做到在所有批次上的对比
损失都小，最后的编码器保证了纵向的一致性； 横向上，同一批次的样本由于对比损失优化的缘故，同类相吸，异类相斥。

另外一种解释-Batch Learning视角

如下图,从Loss方面考虑,对于SimCLR,从损失函数来看,在一个batch上的损失,仅仅由两个编码器和当前batch的样本有关,
即待优化的模型f和当前batch的样本,这个完全符合我们常规的Loss函数,所以优化过程如下图,最终优化每一个batch,
整体损失也是减小的,所以最终每个batch上的对比损失也是很小的,因此在大多数batch上优化成功,代表当前批次的样本
丢到两个编码器后的嵌入向量同类靠近,异类很远.

$$L = L_1 + L_2 + L_3 + \dots$$

$$\frac{\partial L}{\partial \theta} = \frac{\partial L_1}{\partial \theta} + \frac{\partial L_2}{\partial \theta} + \frac{\partial L_3}{\partial \theta} + \dots$$

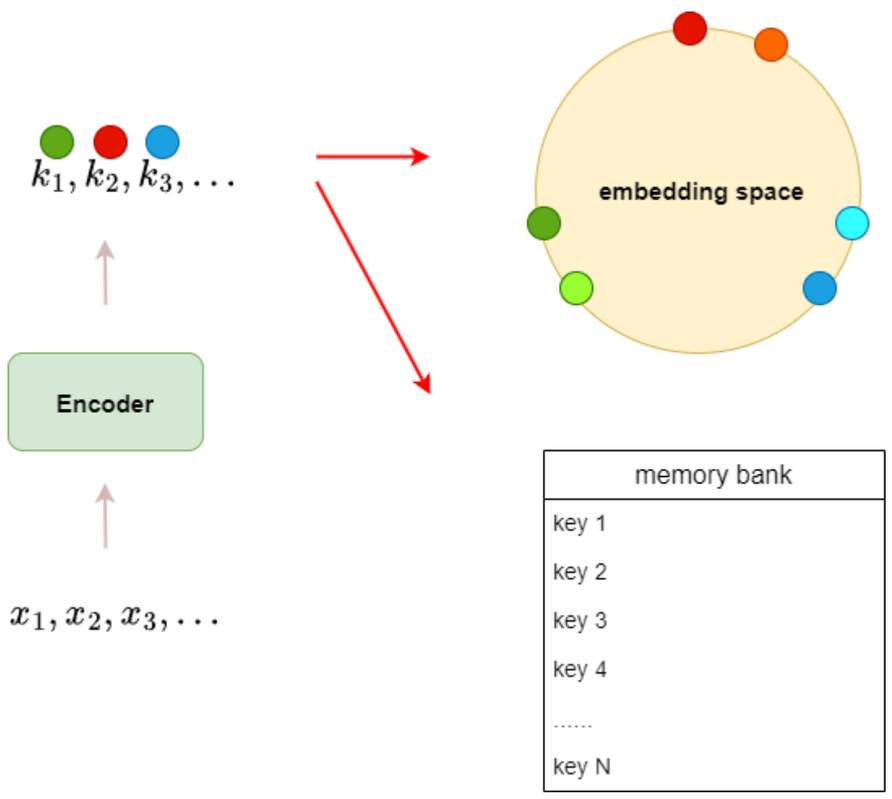


如图, $\frac{\partial L}{\partial \theta}$是某个局部整体下降方向, 也是每个*batch*的下降方向向量的叠加,
由此可以推断出, 大部分*batch*的下降方向与红色箭头一致, 所以优化时,
每个*batch*进行梯度下降, 整体也是下降的.

上面这么说实际还不是比较严谨, 因为每次在一个*batch*进行梯度下降后,
模型参数已经更新了, 下次已经是换个地方进行下降了, 每次都换个地方,
那么这些不同地方的下降方向的叠加能是整体的下降方向吗?

其实可以这样想 :
1.假设整体*Loss*在局部某一范围都是单调的, 即有个上升和下降方向;
2.在一个*batch*上进行梯度下降时, 前面讲了, 这个方向大概率与整体下降方向保持一致;
3.由于前面走的不多, 当前依旧处于相同的地势, 也就是说整体下降方向一时半会都不会怎么变的, 于是在新的地方进行*batch*梯度下降时, 大概率这个方向还是和整体下降方向一致
4.基于上面, 所以一定范围内, 局部*batch*梯度下降大多数走的方向和整体下降的方向是一致的

二、memory bank 方式



memory bank 的前向流程：

【Step1】：利用随机初始化的编码器，把数据集的所有样本进行编码，构造一个memory bank；
【Step2】：按照如下方式训练（按照有监督方式来描述）

```
[epoch1_batch1]：拿第一批样本经过编码，与memory bank中随机挑选的key（比如1，3，5）进行对比学习，
                 更新编码器，然后利用更新好的编码器对相应位置的key进行更新（拿新编码器求出相应位置样本的特征，覆盖原来的key）
经过了一个轮回...
[epoch2_batch1]：拿第一批样本经过编码，与memory bank中另外随机挑选的key（比如2，4，6）进行对比学习，
                 更新编码器，然后利用更新好的编码器对相应位置的key进行更新
循环往复...
```

存在的问题分析(clustering perspective)：

由于我们做的是局部优化，所以我们只抽出一个batch的训练过程，来看这一个batch的训练是work还是fail。

- step1: 固定聚类中心key,经过编码器Q找到同色点后，发现距离不是最小,染色方案需要调整,于是调整

编码器Q进行重新染色,于是得到基于当前聚类中心key下的最优染色;

- step2: 固定染色,将重新染色的小球直接当新的聚类中心(可以是多个),送给memory bank;
- step3: 固定前面新的key,循环执行step 1、2.
  看上去好像没问题，但是问题就出在key上面，因为对于同一批次样本，不同轮回选取的key都是从memory bank中

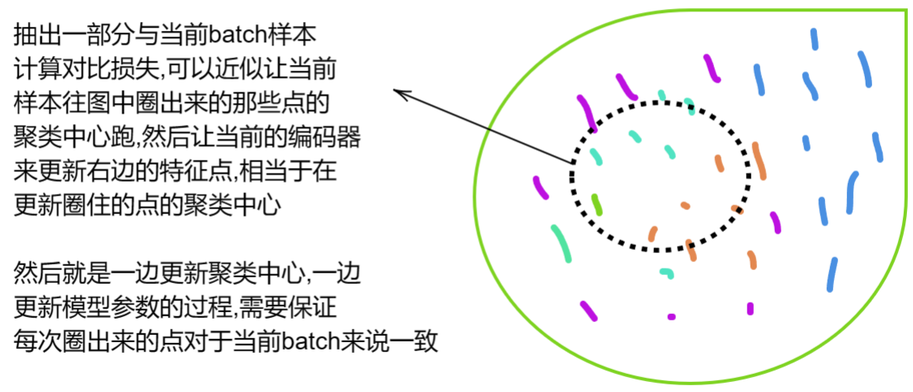
随机挑选的，导致每次参照物不同，假如两个key一个在东边，一个在西边，那么染色方案就会认为一会东边要染红，
一会西边要染红（EncoderQ出来的特征一会东边跑，一会西边跑），导致训练不收敛，模型坍塌。

改进的办法：

由于我们这是探索性的研究,并不会给出比较完善的方案,只提供解释性强一点的方案,memory bank原文作者后续好像也
考虑到了key不一致的问题,然后给了修正,这个修正方案是什么具体我没有去关注,因为后面我并不会使用这种算法.
那么我们最简单的修正方案是什么呢,就是直接那同一个batch中的那些样本经过更新后的编码器Q的特征当作key,或者参考
聚类的办法,用这些特征的平均当作key,不同batch的key互补干扰,即便你是同类key,每个batch只用各自的key,也就是分batch聚类.

有人说这不就是SimCLR了吗,确实很像,但SimCLR是端到端优化的,跟这里的工作方式完全不一样.
再次强调,这只是探索性研究,实际这种方法并没有虽然改掉了key特征不一致的毛病,但是每个memory bank大小被限制成batch size大小了,原先memory bank设计的初衷就是想使用小的batch size,经过前面这么一改,原来的优点就没了.

固定key的memory bank



算法的实现

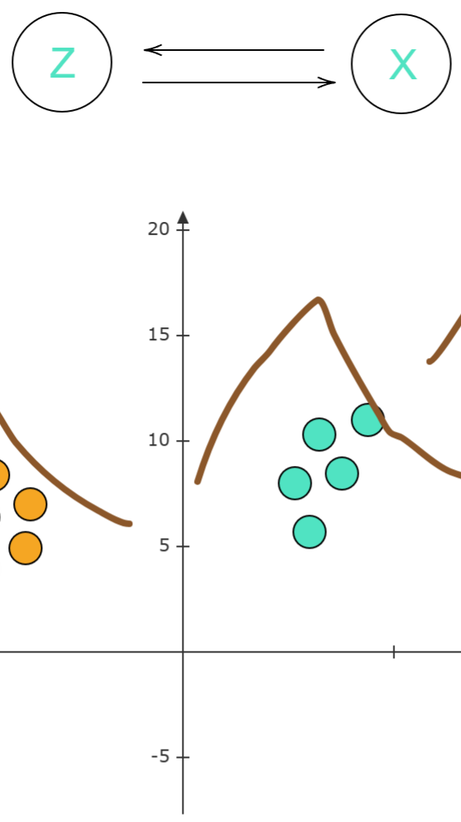
1.首先对照那张图,我们定义一个编码器Q,和一个本地memory bank文件,我们这里由于数据不多,memory bank就用列表放在内存里好了,内存放得下优先放内存,不然每次取key开销太大;
2.对比损失的实现思路应该是这样的,EncoderQ出来的向量,和从memory bank里相应位置抽出来的key计算相似矩阵,然后根据标签抽出正例距离计算对比损失,由于我们抽出来的key和EncoderQ出来的向量的标签是一样的,所以整体实现的对比损失和前面的几乎一样,就是计算相似度矩阵那里不一样,入口传特征的时候需要传两个,loss接口输入参数多了来自memory bank里面的key,loss接口变了,于是runner那边的loss前向计算代码也要改一下.
由于实现价值不大,所以这里不实现了,仅仅大致说一下怎么做.

三、momentum encoder 方式——MoCo

于是乎,它来了,前面memory bank改进后里面的key仅限于当前batch,很不爽,现在引入动量编码器和队列,在key一致的基础上,增加key的数量,不仅可以来自于当前batch,前面用到的依旧可以用,是不是很神奇,那么它是怎么做到的呢?

MoCo的前向流程

四、用聚类的观点看对比学习



k means 聚类 :

$$Loss = \sum_{i=1}^{N}\sum_{k=1}^{K} z_{ik}||x_i - u_k||^2$$

Gaussian 混合 :

$$Loss = \sum_{i=1}^{N}\sum_{k=1}^{K} p\left(z_{ik}|x_i\ ;\ \theta^t\right)[log\pi_k + logN(x_i|u_k, \Sigma_k)]$$

简单起见, 令$\pi_k = \frac{1}{K}$为常数, $\Sigma_k = I$, 也就是用欧氏距离替代马氏距离
此时有,

$$Loss = \sum_{i=1}^{N}\sum_{k=1}^{K} p\left(z_{ik}|x_i\ ;\ \theta^t\right)||x_i - u_k||^2$$

算法的解释

1. 高斯混合 Loss Function 的由来

首先高斯混合的目的是使用当前观测数据的不完全似然函数,来找到基于当前假设下的最优数据概率密度,由于是高斯混合假设,并且假设有3堆点,也就是3个类别,实际上是找出这三个类别的中心u_k.
基于不完全数据似然函数的求解算法,是EM算法中的Q函数形式,最终Loss就是上方那个公式,注意也有基于完全数据似然函数进行的求解,使用的EM算法是ELBO形式,不是我们这里讨论的.

另外注意,Gaussian 混合概率密度函数的形式背后有个隐变量概率图,这个概率图包含了隐变量Z和观测变量X的
各种概率关系,隐变量Z的实际含义就是类别,颜色,假设是个硬高斯概率密度(可以写成分段函数),这个密度函数
计算的就是点与类别中心u_k的距离套个exp,实际上本质还是距离,简单起见这里就用欧氏距离,以便和k means有个对照.

2.高斯混合优化过程及依据

首先,假设目前的状态是t步优化的结果,也就是当前模型参数是θ^t,也就是我们当前有了这堆点的
概率密度函数,因为还不是最优的,所以可能是比较偏的,接下来,我们对每个点都按照如下过程做:
step1: 聚类中心固定不动,计算点x_i来自于每个聚类中心的概率,对于k means我们会把最大的概率置为1,其他置为0,硬和软聚类的区别;
step2: 由于step1对每个点都重新上色了,然后我们固定颜色,算出每个点与聚类中心的聚类,此时聚类中心不要固定,通过优化改变聚类中心的位置,使距离变小.
注意,优化高斯密度函数的参数u_k,就是优化类别中心,移动类别中心,并且根据EM算法的推导,优化的结果就是样本点的均值,比如u_1就等于红色点的中心(此时颜色固定了)
中心移动后,每个点来自每个中心的概率就改变了,然后重新给每个点上色,也就是重新给每个点分类,分类完毕后,重新计算距离,改变聚类中心最小化这个距离,循环往复.

3. 与聚类的联系

k means 聚类的步骤:
step1: 聚类中心固定不动,计算点x_i与每个聚类中心的距离,选最小的,比如与第s个中心最近,就令$z_{is}=1$,其它$z_{ik}=0$,对应高斯混合求后验概率;
step2: 由于step1对每个点都重新上色了,也就是点都分类好了,我们分别计算每堆点的中心,用新中心代替老中心,聚类中心就更新了,这就是上面EM算法第二步的优化公式做的事.

4. step2 优化背后的洞悉

这个优化的一般形式就是下式:

$$E_x(output-x)=\sum_{i=1}^{N}(output-x_i)$$

优化output模型的参数,最后优化的最优解,就是这个模型的输出为:

$$output=E_x(x)=\frac{1}{N}\sum_{i=1}^{N}(x_i)$$

比如上面对参数u_k的优化,求出来的就是样本点的均值,这个结果和回归分析中的回归曲线是一样的.

与对比学习的联系

之所以把对比学习往聚类方向思考,主要是受一个知乎文章启发: https://zhuanlan.zhihu.com/p/452659570

原来的公式如下:

$$\mathcal{L}=\sum_{\{i\}}\sum_{\{z^+\}}-log\frac{exp(sim(z_i,z^+)/\tau)}{\sum\limits_{\{z^+,z^-\}}exp(sim(z_i,z_p)/\tau)}$$

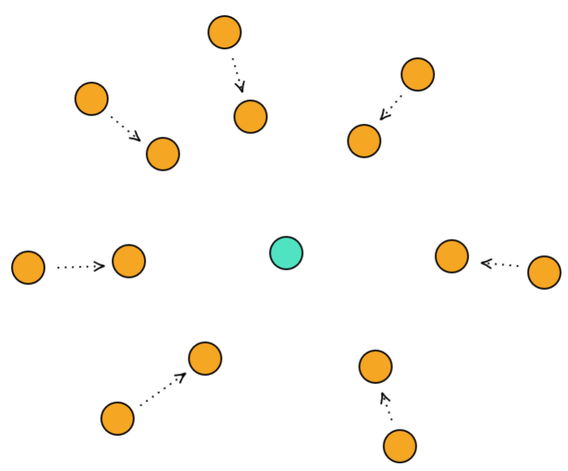
简化成如下形式(我们从最简单的形式去理解其工作机制):

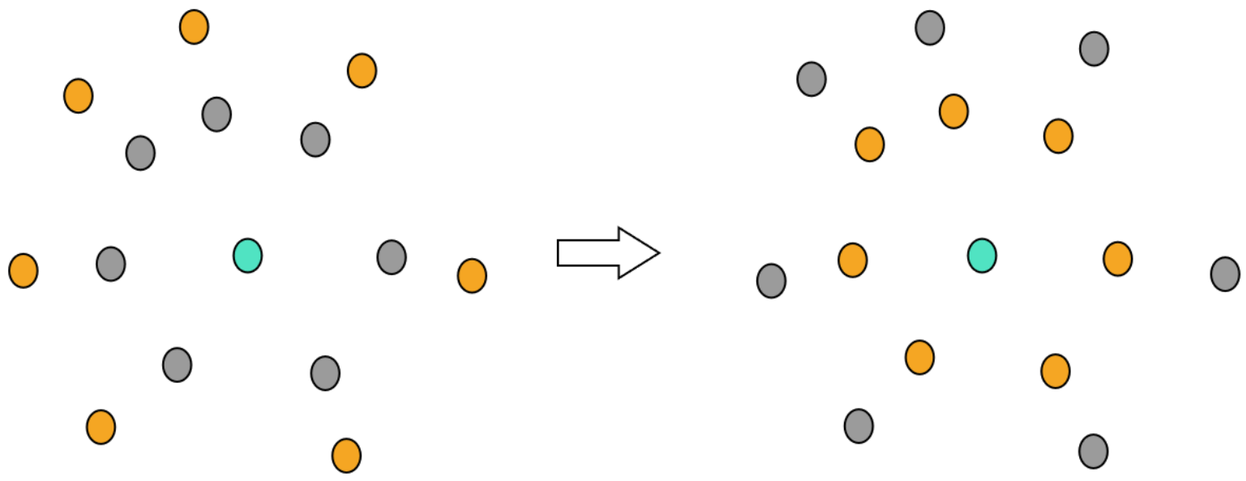
$$\mathcal{L}=\sum_{\{i\}}\sum_{\{z^+\}}||z_i-z^+||=\sum_{\{i\}}\sum_{\{z^+\}}||f_1(x_i)-z^+||$$

优化过程:
step1: positive固定不动,计算点z_i与每个positive的距离,通过修改Encoder1也就是这里的f1来换色,选最小的距离;
step2: 固定Encoder1,我们需要更新positive,根据前面EM算法的结果,positive应该变成在Encoder1上色后的情形下的平均,比如当前算红色的positive,
那么就把是红色的那些样本通过Encoder1求平均来更新positive;
但是有所不同的是,我们的positive直接用红色样本点本身的Encoder1直接更新,不要求平均了,我们的positive也不止1个.

对比学习的聚类视角

根据前面分析,更新EncoderQ的过程就是换色的过程,这个怎么来理解呢?
如下图,表面更新编码器Q,看上去是将与绿色key的距离拉近了;
然后我们换个角度,假设平面上原本就有无数个点,与绿色点同类的为橘红色,
与绿色点不同类的为灰色,Encoder只是判断哪些点跟橘红色为同类,更新编码器,
实则只是改变了点的颜色,给每个点重新配色,使得重新配色后的点与key距离更近.





附录:高斯混合模型

EM（Expectation-Maximization）算法是一种常用于混合高斯模型参数估计的迭代优化算法,常见有两种形式:

- Q函数形式,理论推导见李航统计学习方法
- ELBO+KL形式,理论推导网上查,应用在高斯混合模型上需要写出完全数据的似然函数,李航书上有

比较令人纳闷的就是,李航书上推导的EM算法理论形式,跟用来解决高斯混合模型使用的EM算法的形式不一样!

EM算法的Q函数形式的一般步骤(为什么这么做,以及收敛性请查阅理论推导):

1. 初始化参数 $\theta^{(0)}$，设置迭代次数上限或收敛条件。
2. E步（Expectation Step）：计算完全数据的后验概率 $p(Z|X,\theta^{(t)})$，其中 Z 是隐变量，X 是观测数据，$\theta^{(t)}$ 是第 t 次迭代中的参数估计。
3. M步（Maximization Step）：通过最大化完全数据的期望对数似然函数，得到新的参数估计值 $\theta^{(t+1)}$。
4. 重复执行第2步和第3步，直到满足停止条件。

接下来，给出高斯混合模型使用EM算法的Q函数形式的一般步骤:

Step 1: 初始化 设混合高斯模型有 K 个混合成分，参数为 $\theta=\{\pi_k,\mu_k,\Sigma_k\}$，初始化参数为 $\theta^{(0)}$。

Step 2: E步 在给定当前参数估计 $\theta^{(t)}$ 的情况下，计算完全数据的后验概率 $p(Z|X,\theta^{(t)})$，其中 $Z=\{z_1,z_2,\cdots,z_N\}$ 是隐变量，$X=\{x_1,x_2,\cdots,x_N\}$ 是观测数据。

根据贝叶斯定理，完全数据的后验概率可以表示为：

$$p(Z|X,\theta^{(t)})=\frac{p(X,Z|\theta^{(t)})}{p(X|\theta^{(t)})}$$

由于 $p(X|\theta^{(t)})$ 对所有参数 $\theta^{(t)}$ 都是常数，因此可以忽略，得到：

$$p(Z|X,\theta^{(t)})=\frac{p(X,Z|\theta^{(t)})}{\sum_Z p(X,Z|\theta^{(t)})}$$

其中，分子可以按照混合高斯模型的定义展开：

$$p(X,Z|\theta^{(t)})=\prod_{i=1}^{N}\prod_{k=1}^{K}\left[\pi_k^{(t)}\mathcal{N}(x_i|\mu_k^{(t)},\Sigma_k^{(t)})\right]^{zi,k}$$

Step 3: M步 在给定后验概率 $p(Z|X,\theta^{(t)})$ 的情况下，通过最大化完全数据的期望对数似然函数，计算新的参数估计值 $\theta^{(t+1)}$。

完全数据的对数似然函数为：

$$\log p(X,Z|\theta)=\sum_{i=1}^{N}\sum_{k=1}^{K}z_{i,k}\big[\log\pi_k+\log\mathcal{N}(x_i|\mu_k,\Sigma_k)\big]$$

取期望得到完全数据的期望对数似然函数：

$$Q(\theta,\theta^{(t)})=\sum_{i=1}^{N}\sum_{k=1}^{K}p(z_{i,k}|x_i,\theta^{(t)})\big[\log\pi_k+\log\mathcal{N}(x_i|\mu_k,\Sigma_k)\big]$$

对每个参数进行最大化，得到新的参数估计值：

$$\pi_k^{(t+1)}=\frac{\sum_{i=1}^{N}p(z_{i,k}|x_i,\theta^{(t)})}{N}$$

$$\mu_k^{(t+1)}=\frac{\sum_{i=1}^{N}p(z_{i,k}|x_i,\theta^{(t)})x_i}{\sum_{i=1}^{N}p(z_{i,k}|x_i,\theta^{(t)})}$$

$$\Sigma_k^{(t+1)}=\frac{\sum_{i=1}^{N}p(z_{i,k}|x_i,\theta^{(t)})(x_i-\mu_k^{(t+1)})(x_i-\mu_k^{(t+1)})^T}{\sum_{i=1}^{N}p(z_{i,k}|x_i,\theta^{(t)})}$$

Step 4: 迭代更新 重复执行E步和M步，直到满足停止条件。通常可以设置最大迭代次数或判断参数变化是否达到阈值。

注意,k means聚类就是硬聚类形式的高斯混合模型,k means 聚类只给了算法的迭代步骤,没有给出为什么这么做,以及收敛性,
其背后的原理跟这里高斯混合模型背后的原理是同一个,就是EM算法的Q函数形式.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: