小样本分类论文复现实验报告

1 论文概述

该工作提出了一种基于对比学习的两阶段小样本图像分类框架，通过在预训练和元训练阶段利用对比学习来解决小样本图像分类问题，该方法容易与其他两阶段式的小样本图像分类方法相结合。

引言

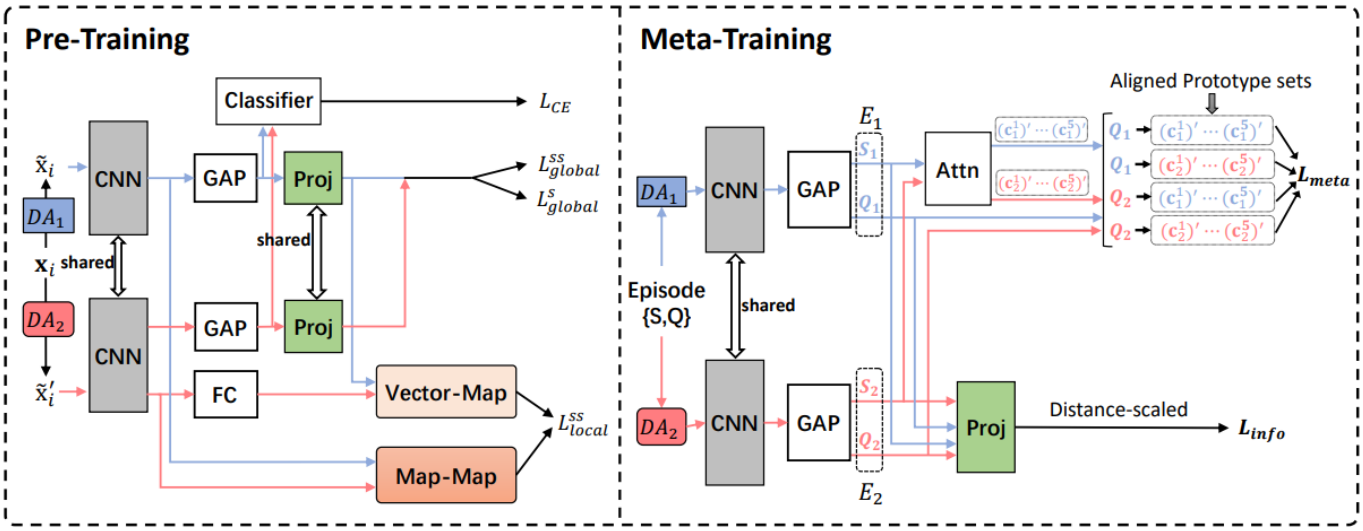
基于深度学习的方法在带标注数据稀少的现实生活场景中很难做到良好地泛化，并且针对带标注数据的收集过程颇为费时费力，这更加使其成为一种挑战。受人类从少数例子中学习新概念的能力的启发，小样本学习被认为是应对该挑战的一个有前景的方向，它可以将从少数基础类样本中学习到的知识适应于新的任务。

本文提出了一个基于对比学习的两阶段分类算法框架，通过将对比学习无缝整合到预训练和元训练阶段来解决小样本图像分类问题。具体来说，该方法在预训练阶段利用自监督和有监督的对比损失来训练模型，其中自监督对比损失以特征向量–特征图和特征图–特征图两种形式进一步利用特征图的局部信息。其次，该方法在元训练阶段采用了一种跨视图任务式训练机制一种基于增强任务的距离缩放对比损失，使模型克服不同任务视图之间的偏差，促进模型在新类样本上的泛化能力。

这项工作的主要贡献如下：

- 提出一种基于对比学习的FSL框架，该框架由预训练和元训练阶段组成，以改进少样本分类。该框架很容易地与两阶段的FSL结合
- 在预训练阶段采用基于global和local的自监督对比损失，以增强结果表示的可推广性；
- 提出一种CVET机制，通过在增强集之间执行分类，迫使模型找到更多可转移的表示，同时引入基于增强集的距离尺度对比损失，以确保分类过程中不受视图之间极端偏差的影响。

方法



本文采用两阶段训练策略，并将对比学习纳入两阶段，以学习更具普遍性的表达。在预训练阶段采用自监督和监督对比损失来获得良好的初始表示。在元训练阶段，提出一种新的交叉视角CVET机制和距离尺度对比损失，这使得模型能够克服每个场景的视角之间的偏差，并在新任务中很好地推广。由于其简单有效的任务特定模块，多头注意力模块，将没有辅助损失的FEAT作为基线。

Pre-training

在预训练阶段引入instance-discriminative contrastive learning，以环节仅具有CE损失的训练所导致的过拟合。分别在global和local地方两级提出自监督的对比损失，在这些损失中使用自监督有助于产生更普遍的表达。同时，使用全局监督对比损失来捕获来自同一实例之间的相关性。

Global self-supervised contrastive loss.

InfoNCE loss 旨在增强同一图像的视图之间的相似性，同时降低不同图像视图之间的相似性。形式上，将两种数据增强方法随机应用于元训练集 $\mathcal{D}_{train}$ 中的一批样本 $\{x_i, y_i\}_{i=1}^{N}$ 并生成增强批次 $\{\hat{x_i}, \hat{y_i}\}_{i=1}^{2N}$，这里，$\hat{x_i}$，$\hat{x_i}'$ 是 x_i 的两个不同view，它被认为是a positive pair。定义 f_ϕ 为具有可

学习参数的特征提取器，将样本 $\hat{x}_i$ 转换为特征映射$\hat{x}_i = f_\phi(\hat{x}_i) \in \mathbb{R}^{C\times H\times W}$ 并在GAP后进一步获得全局特征 $\mathbf{h}_i \in \mathbb{R}^C$ 。使用具有一个隐藏层的MLP来实例化投影头 $proj(\cdot)$ 以生成投影向量 $\mathbf{z}_i = proj(\mathbf{h}_i) \in \mathbb{R}^D$ ，全局自监督对比损失可以计算为：

$$L^{ss}_{\text{global}} = -\sum_{i=1}^{2N} \log \frac{\exp\left(\mathbf{z}_i \cdot \mathbf{z}'_i/\tau_1\right)}{\sum_{j=1}^{2N} 1_{j\neq i} \exp(\mathbf{z}_i \cdot \mathbf{z}_j/\tau_1)},$$

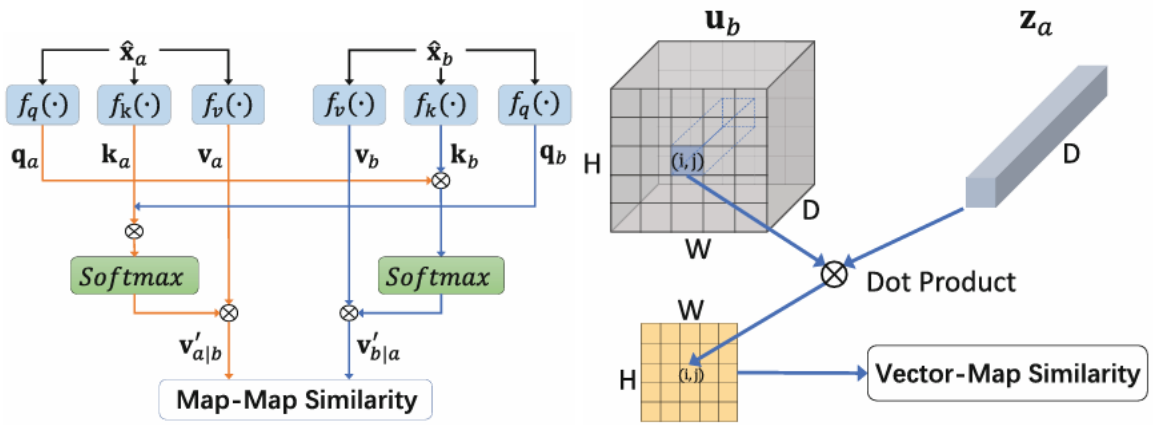
Local self-supervised contrastive loss.

包含两个module，map-map module局部对局部和map-vector module局部对全局。

图(a)中，xa-hat、xb-hat分别表示两种数据增强方式后得到的样本的特征；三个f函数表示spatial projection head，通过这些头map就被投影成HW×D的vector了。然后分别将a与b的map相互对齐（即图中交叉的线头）、然后softmax，计算相似度，这些相似度将会用于求loss。

图(b)中，ub表示map（DxHW），za表示特征投影后得到的vector（Dx1），传入的特征分别对应于两种数据增强方式得到的map和vector。然后对map的每个像素位置(i,j)（大小为1xD）与za做点积得到的结果再计算vector-map相似度、然后计算loss。

最后，Local self-supervised contrastive loss等于这两个module的loss之和。



(a) Map-map module　　(b) Vector-map module

$$L^{ss}_{map-map} = -\sum_{i=1}^{2N} \log \frac{\exp\left(sim_1\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}'_i\right)/\tau_2\right)}{\sum_{j=1}^{2N} \mathbb{1}_{j\neq i} \exp\left(sim_1\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j\right)/\tau_2\right)},$$

$$L^{ss}_{vec-map} = -\sum_{i=1}^{2N} \log \frac{\exp\left(sim_2\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}'_i\right)/\tau_3\right)}{\sum_{j=1}^{2N} \mathbb{1}_{j\neq i} \exp\left(sim_2\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j\right)/\tau_3\right)},$$

$$L^{ss}_{local} = L^{ss}_{vec-map} + L^{ss}_{map-map}.$$

Global supervised contrastive loss.

为了利用单个实例之间的相关性以及同一类别中不同实例之间的关联，还采用监督对比损失如下：

$$L_{global}^{s}=\sum_{i=1}^{2N}\frac{1}{|P(i)|}\sum_{p\in P(i)}L_{ip},$$

总结： loss during pre-training

在预训练阶段采用有监督和自监督对比损失来获得一个良好的初始表征。除了在全局层面提出了自监督对比损失 L_{global}^{ss}，还通过特征向量-特征图模块和特征图-特征图模块来计算局部自监督对比损失 L_{global}^{ss} 来生成更通用的表征。预训练阶段的总损失为：

$$L_{pre}=L_{CE}+\alpha_1 L_{global}^{ss}+\alpha_2 L_{local}^{ss}+\alpha_3 L_{global}^{s}$$

其中，L_{CE}是是交叉熵损失，L_{global}^{ss} 和L_{local}^{ss} 分别为全局有监督对比损失和全局自监督对比损失。α_1，α_2，α_3是比例系数。

Meta-training

Cross-view Episodic Training

E表示Episode的缩写。将E视作共享上下文（shared context），将two augmented episodes视作E的两个views（看图把这两个view的特征共同输入module中去了所以叫cross-view吧）。经过GAP后，得到全局向量hi，然后计算两个集合S1和S2中每个类的原型、送入Attn模块加工得到处理后的原型（注意只有support里面的样本会被计算）。用下式计算query集合中的每个样本的概率分布：

$$P(y=k\mid\mathbf{h}_i,\mathcal{T}(r))=\frac{\exp\left(-d\left(\mathbf{h}_i,(\mathbf{c}_r^k)'\right)\right)}{\sum_{j=1}^{M}\exp\left(-d\left(\mathbf{h}_i,(\mathbf{c}_r^j)'\right)\right)},$$

τ是数据增强方式的分布，d是欧氏距离，然后再计算下面的损失。m和n可以取1或2，就如图上所示计算L11 L12 L21 L22的损失：

$$L_{mn}=\frac{1}{|\mathcal{Q}_m|}\sum_{(\mathbf{h}_i,y_i)\in\mathcal{Q}_m}-\log P(y=k\mid\mathbf{h}_i,\mathcal{T}(n)),$$

计算交叉熵view分类损失如下：

$$L_{meta} = \frac{1}{4} \sum_{m,n} L_{mn}.$$

Distance-scaled Contrastive Loss

由于对比学习方法仅在实例层面起作用，因此在没有充分利用适合FSL的episode训练机制的情况下，简单地将对比损失添加到元训练中是superficial。为了更好地将对比学习应用于元训练，在共享集的两个view之间执行query实例查询。因此，将监督对比损失以episode训练的情形重新表达如下：

$$L(\mathbf{z}_i) = -\sum_{\mathbf{z}_H \in H(\mathbf{z}_i)} \log \frac{\lambda_{\mathbf{z}_i \mathbf{z}_H} \exp\left(\mathbf{z}_i \cdot \mathbf{z}_H / \tau_5\right)}{\sum_{\mathbf{z}_A \in A(\mathbf{z}_i)} \lambda_{\mathbf{z}_i \mathbf{z}_A} \exp\left(\mathbf{z}_i \cdot \mathbf{z}_A / \tau_5\right)}.$$

改模型可以学习适应于不同任务的更具辨别性的表示，然后计算distance-scaled contrastive loss：

$$L_{info} = \sum_{\mathbf{z}_i \in \mathcal{Q}_1 \cup \mathcal{Q}_2} \frac{1}{|H(\mathbf{z}_i)|} L(\mathbf{z}_i).$$

总结：Objective in Meta-training

对一个由支持集和查询集组成的任务片段，分别应用两个不同的数据增强策略以得到该图像样本集的两个不同的视图。跨视图任务式训练机制分别在两个支持集视图上对每个查询集视图进行最近邻中心分类。同时，该方法继承预训练后投影模块，将两个任务片段视图中的样本投影为向量并在此基础上构建对比样本对，然后引入样本之间的距离系数来计算基于距离缩放对比损失L_{info}。该阶段的损失为：

$$L_{total} = L_{meta} + \beta L_{info}$$

其中L_{meta} 最近邻中心分类的优化目标，β是比例系数。该阶段方法使模型能够学习到适应不同任务的更具判别性的表征。

2 核心方法实现代码

2.1 data augmentation

本文的数据增强部分，在于对同一图像随机使用两种数据增强方法来获得增强的片段，并将它们视为原始片段的不同视图。论文中对比了多种数据增强方法，并指出“Standard”和“SimCLR”表现最佳。

2.1.1 Standard & SimCLR

SimCLR包含randomresizedcrop, randomhorizontalflip, randomcolorjitter and randomgrayscale方法。

```python
class SimCLR_Style:
    def __init__(self, size):
        self.base_transform = self.get_simclr_pipeline_transform(size)

    @staticmethod
    def get_simclr_pipeline_transform(size, s=1):
        """Return a set of data augmentation transformations as described in the SimCLR paper."""
        color_jitter = transforms.ColorJitter(0.8 * s, 0.8 * s, 0.8 * s, 0.2 * s)
        data_transforms = transforms.Compose([transforms.RandomResizedCrop(size=size),
                                              transforms.RandomHorizontalFlip(),
                                              transforms.RandomApply([color_jitter], p=0.8),
                                              transforms.RandomGrayscale(p=0.2)])
        return data_transforms
```

Standard包含randomresizedcrop, colorJitter and randomhorizontalflip方法。

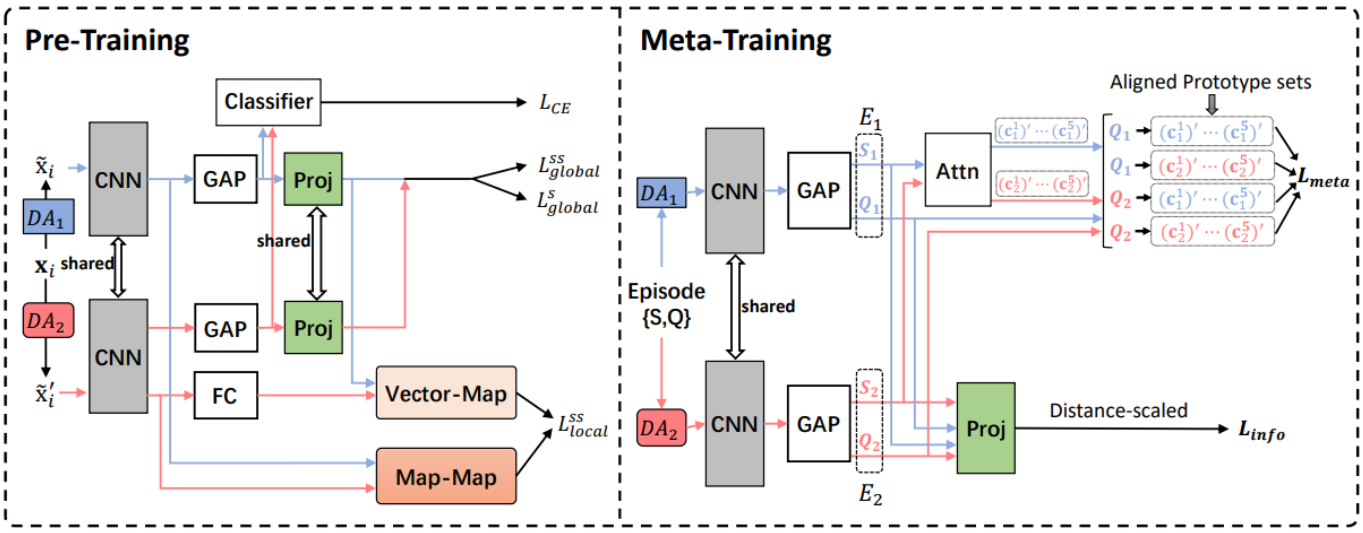
```python
class Standard:
    def __init__(self, size):
        self.base_transform = self.get_standard_pipeline_transform(size)

    @staticmethod
    def get_standard_pipeline_transform(size, s=1):
        data_transforms = transforms.Compose([
            transforms.RandomResizedCrop(size=size),
            transforms.ColorJitter(0.8 * s, 0.8 * s, 0.8 * s, 0.2 * s),
            transforms.RandomHorizontalFlip(),
        ])
        return data_transforms
```

2.1.2 方法调用

上文提到，需要对同一图像随机使用两种数据增强方法，以下为代码。

```python
class ContrastiveLearningViewGenerator(object):
    """Take two random crops of one image as the query and key."""

    def __init__(self, base_transform, n_views=2):
        self.base_transform = base_transform
        self.n_views = n_views

    def __call__(self, x):
        return self.base_transform(x)
```

2.2 pre-training



2.2.1 准备工作

1. 对元训练集DTrain中的样本进行数据增强

```pre-training
 images, target = batch
 target = target.to(self.device)
 images = images.to(self.device)
```

2. 使X通过backbone获得全局特征

```
pre-training                                                    Python
feat_extractor = self.emb_func
global_feat = feat_extractor(images)
```

3. 定义参数

```
pre-training                                                    Python
# 定义温度参数
tau1 = 0.1
tau2 = 0.1
tau3 = 0.1
tau4 = 0.1

# 定义权重系数
alpha1 = 1.0
alpha2 = 1.0
alpha3 = 1.0
```

2.2.2 CE loss

计算CE loss前首先需要经过一个分类器。这里我们使用的是原有的线性分类器和交叉熵损失函数。

```
pre-training                                                    Python
def __init__(self, feat_dim, num_class, inner_param, **kwargs):
    ......

    self.classifier = nn.Linear(feat_dim, num_class)
    self.loss_func = nn.CrossEntropyLoss()


classifier = self.classifier
output = classifier(global_feat)
L_CE = self.loss_func(output, target)
```

2.2.3 Global self-supervised contrastive loss

该损失旨在提高同一图像的视图之间的相似性，同时降低不同图像的视图之间的相似性。在计算之前，需要先让全局特征经过具有一个隐藏层的MLP来实例化投影头Proj(·)以生成投影向量 $z_i = proj(h_i)$ 。

```python
class MLP(nn.Module):  # MLP with one hidden layer
    def __init__(self, feat_dim, output_dim):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(feat_dim, output_dim)
        self.act_func = nn.ReLU()

    def forward(self, x):
        x = self.hidden(x)
        x = self.act_func(x)
        return x

    def __init__(self, feat_dim, num_class, inner_param, **kwargs):
        ......

        self.projection = MLP(feat_dim, num_class)
```

loss的计算公式如下：

$$L_{global}^{ss} = -\sum_{i=1}^{2N} \log \frac{\exp\left(\mathbf{z}_i \cdot \mathbf{z}_i' / \tau_1\right)}{\sum_{j=1}^{2N} \mathbb{1}_{j \neq i} \exp\left(\mathbf{z}_i \cdot \mathbf{z}_j / \tau_1\right)},$$

其中，·运算表示l2归一化后的内积，τ1是标量温度参数，1∈{0，1}是指示函数。正对 z_i' 和 z_i 是从同一样本 x_i 的增广版本中提取的。

pre-training Python

```python
class GlobalSSContrastiveLoss(torch.nn.Module):
    #全局自监督对比损失
    def __init__(self, temperature):
        super(GlobalSSContrastiveLoss, self).__init__()
        self.temperature = temperature

    def forward(self, z_i, z_i_prime):
        # 计算相似度分数
        scores = torch.einsum("bi,bi->b", z_i, z_i_prime) / self.temperature

        # 计算正样本对的logits
        exp_scores_pos = torch.exp(scores)

        # 计算负样本对的logits
        exp_scores_neg = torch.sum(torch.exp(scores), dim=-1, keepdim=True) - exp_scores_pos

        # 添加indicator function
        # 使用掩码矩阵排除对角线元素
        mask = torch.eye(len(scores), dtype=torch.bool)
        exp_scores_neg = exp_scores_neg.masked_fill(mask, 0)

        # 计算对比损失
        loss = -torch.log(exp_scores_pos / (exp_scores_pos + exp_scores_neg))

        return loss.mean()


z_i = self.projection.forward(global_feat)
z_i_prime = self.projection.forward(global_feat)

ss_contrastive_loss = GlobalSSContrastiveLoss(temperature=tau1)
l_ss_global = ss_contrastive_loss(z_i, z_i_prime)
```

2.2.4 Map-Map loss

首先使用三个空间投影头fq、fk、fv将局部特征地图分别投影到query、key、value上。然后对于一对局部特征地图，进行对齐操作分别获得 $\mathbf{v}'_{a|b} = softmax\left(\frac{\mathbf{q}_b\mathbf{k}_a^\top}{\sqrt{d}}\right)\mathbf{v}_a$ $\mathbf{v}'_{b|a} = softmax\left(\frac{\mathbf{q}_a\mathbf{k}_b^\top}{\sqrt{d}}\right)\mathbf{v}_b$.

在对准结果的每个位置进行归一化后，我们可以计算两个局部特征映射之间的相似度。

计算公式如下：

$$sim_1\left(\hat{\mathbf{x}}_a, \hat{\mathbf{x}}_b\right) = \frac{1}{HW} \sum_{1 \leq i \leq H, 1 \leq j \leq W} \left(\mathbf{v}'_{a|b}\right)_{ij}^{\top} \left(\mathbf{v}'_{b|a}\right)_{ij}.$$

$$L_{map-map}^{ss} = -\sum_{i=1}^{2N} \log \frac{\exp\left(sim_1\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}'_i\right)/\tau_2\right)}{\sum_{j=1}^{2N} \mathbb{1}_{j \neq i} \exp\left(sim_1\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j\right)/\tau_2\right)},$$

pre-training Python

```python
class SpatialProjectionHead(torch.nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SpatialProjectionHead, self).__init__()
        self.fq = torch.nn.Linear(input_dim, output_dim)
        self.fk = torch.nn.Linear(input_dim, output_dim)
        self.fv = torch.nn.Linear(input_dim, output_dim)

    def forward(self, x):
        # x: (B, C, H, W)
        B, C, H, W = x.size()
        x = x.view(B, C, -1).permute(0, 2, 1)  # Reshape and permute dimensions
        q = self.fq(x)
        k = self.fk(x)
        v = self.fv(x)
        return q, k, v

spatial_projection_head = SpatialProjectionHead(input_dim, output_dim)
xa, xb = spatial_projection_head(global_feat)
l_ss_local_mm = self.map_map_loss(xa, xb, tau2)

def map_map_loss(self, local_features_q, local_features_k, temperature):
    B, N, D = local_features_q.shape
    local_features_q = F.normalize(local_features_q, dim=2)  # 归一化
    local_features_k = F.normalize(local_features_k, dim=2)
    sim_matrix = torch.bmm(local_features_q, local_features_k.transpose(1, 2)) / temperature
    sim_matrix = torch.exp(sim_matrix)  # 指数化
    mask = ~torch.eye(N, dtype=bool, device=local_features_q.device)  # 排除自身比较
    denom = sim_matrix.masked_fill(~mask, 0).sum(dim=2, keepdim=True)
    pos_sim = torch.exp(torch.sum(local_features_q * local_features_k, dim=2) / temperature)
    loss = -torch.log(pos_sim / denom.squeeze()).mean()
    return loss
```

2.2.5 Vector–Map loss

采用vector–map模块进一步开发实例之间的局部对比信息。传入的参数中，使用一个全连接FC层来获得 $\mathbf{u}_i = g(\hat{x}_i) = \sigma(\mathbf{W}\hat{x}_i) \in \mathbb{R}^{D\times HW}$ ，$\mathbf{z}_a$是 $\hat{\mathbf{x}}_a$ 的投影向量。基于特征向量对和特征图的自监督对比损失可以计算如下：

$$L_{vec-map}^{ss} = -\sum_{i=1}^{2N} \log \frac{\exp\left(sim_2\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_i'\right)/\tau_3\right)}{\sum_{j=1}^{2N} \mathbb{1}_{j\neq i} \exp\left(sim_2\left(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}_j\right)/\tau_3\right)},$$

可以将对比对之间的相似度计算为：

$$sim_2\left(\hat{\mathbf{x}}_a, \hat{\mathbf{x}}_b\right) = \frac{1}{HW} \sum_{1\leq i\leq H, 1\leq j\leq W} \left(\mathbf{u}_b\right)_{ij}^{\top} \mathbf{z}_a$$

代入上式即可得出vec_map_loss。

pre–training Python

```python
def vec_map_loss(self, ui, za, tau):
    # ui: (B, D, HW), za: (B, D)
    sim_matrix = torch.matmul(ui.permute(0, 2, 1), za.unsqueeze(-1)).squeeze() / tau
    mask = torch.eye(ui.size(2)).bool()
    loss = -torch.sum(F.log_softmax(sim_matrix, dim=1)[mask]) / ui.size(0)
    return loss
```

2.2.6 Global supervised contrastive loss

为了利用单个实例之间的相关性和同一类别不同实例之间的相关性，我们还采用了监督对比损失。其计算公式如下：

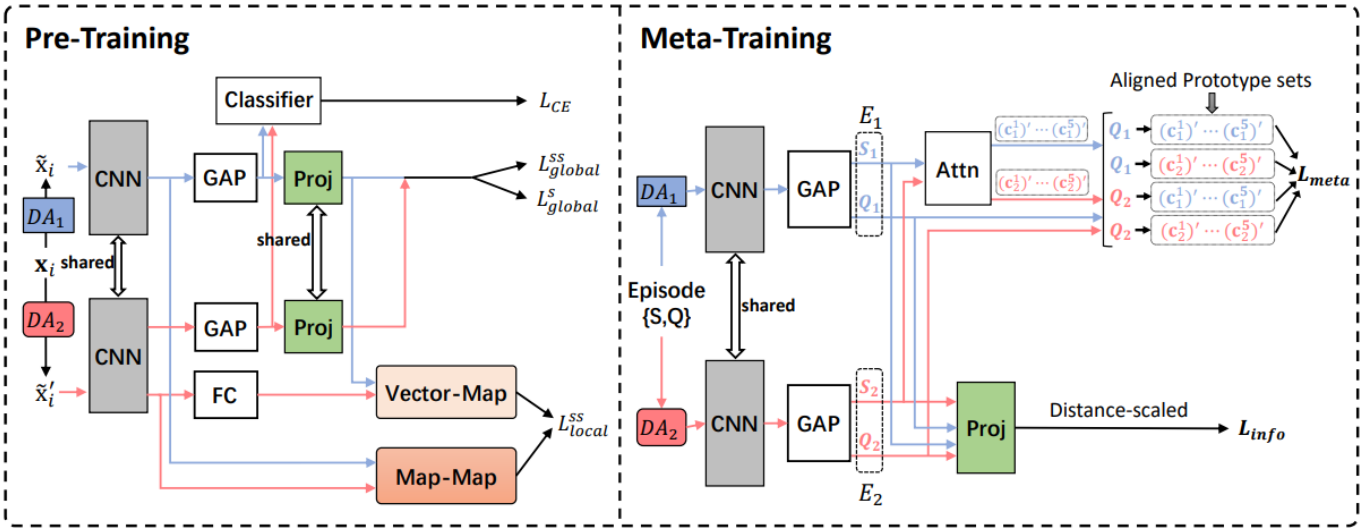
$$L_{global}^{s} = \sum_{i=1}^{2N} \frac{1}{|P(i)|} \sum_{p\in P(i)} L_{ip},$$

$$L_{ip} = -\log \frac{\exp(\mathbf{z}_i \cdot \mathbf{z}_p/\tau_4)}{\sum_{j=1}^{2N} \mathbb{1}_{j\neq i} \exp(\mathbf{z}_i \cdot \mathbf{z}_j/\tau_4)},$$

pre-training Python

```python
def supervised_contrastive_loss(self, features, labels, temperature):
    """
    计算全局监督对比损失
    Args:
    - features: 特征张量，形状为 [2N, D]
    - labels: 标签张量，形状为 [2N]
    - temperature: 温度参数，一个标量
    Returns:
    - loss: 计算得到的损失值
    """
    features = self.projection.forward(features)

    device = features.device
    N = features.shape[0] // 2  # 因为每个正样本对中有两个样本

    # 计算特征的归一化版本
    features = F.normalize(features, dim=1)

    # 计算相似度矩阵
    sim_matrix = torch.mm(features, features.T) / temperature

    # 初始化损失值
    loss = 0.0

    # 计算损失
    for i in range(2 * N):
        # 正样本对
        pos_mask = (labels == labels[i]) & ~torch.eye(2 * N, dtype=bool, device=device)
        pos_sim = sim_matrix[i][pos_mask]

        # 所有样本（包括正负样本对）
        all_sim = sim_matrix[i]

        # 计算损失
        loss += -torch.log(torch.sum(torch.exp(pos_sim)) / torch.sum(torch.exp(all_sim)))

    return loss / (2 * N)
```

2.3 meta-training & testing



2.3.1 meta–training

2.3.1.1 Cross–view Episodic Training

如前所述，我们使用了两种data–augmentation的方法以期增强模型的泛化能力，在meta–training中，我们对于这两种view使用了对比学习以最大化其中的的shared info。首先我们计算出了经过特征提取以及pooling的所有样本，并用均值的方式表示prototype。计算公式如下，其中 $r=\{1,2\}$ 表示两种data–augmentation方法， h_i 为经过feature extractor和pooling得到的特征向量， k 为出现的类别。

$$\mathbf{c}_r^k = \frac{1}{|\mathcal{S}_r^k|} \sum_{(\mathbf{h}_i, y_i) \in \mathcal{S}_r^k} \mathbf{h}_i$$

prototype Python

```python
def crk(self, X, y):
    # 输入一个经过GAP的 **support set**

    class_values = {}
    class_counts = {}

    for i in range(len(X)):
        target = y[i].item()
        value = X[i]

        if target not in class_values:
            class_values[target] = value
            class_counts[target] = 1
        else:
            class_values[target] += value
            class_counts[target] += 1

    class_means = {}
    for target in class_values:
        class_means[target] = class_values[target] / class_counts[target]

    result_array = [class_means.get(target, 0) for target in range(max(y) + 1)]

    return torch.tensor(result_array)
```

我们对于每个episode中的support set都计算出prototype，并经过Attention进行对齐。

alignment Python

```python
c1 = self.attention.forward(c1, query_X1, support_y1)
c2 = self.attention.forward(c2, query_X2, support_y2)
```

接下来我们对于query set中的样本计算出一个在各个类上可能的概率分布，主要通过样本与不同类别的prototype的欧氏距离来衡量（如前所述，prototype可以视为一个类别的均值，所以这种方法是合理的）。计算公式如下，其中 τ 表示对齐之后的prototype。

$$P(y=k \mid \mathbf{h}_i, \mathcal{T}(r)) = \frac{\exp\left(-d\left(\mathbf{h}_i, (\mathbf{c}_r^k)'\right)\right)}{\sum_{j=1}^{M} \exp\left(-d\left(\mathbf{h}_i, (\mathbf{c}_r^j)'\right)\right)}$$

probability distribution

```python
def P(self, X, y, c):
    # 输入单条样本，计算一个label为y的概率
    # 利用crk函数结果中的相应类别(此时已经经过attention)

    pdist = nn.PairwiseDistance(p=2)
    # 使用广播计算和各个类别的欧氏距离
    distances = pdist(X, c)

    up = math.exp(-distances[:, y])
    # 对每个类别的距离取指数并相加
    down = math.exp(-distances).sum()

    p = up / down
    return p
```

接下来我们采用最近质心分类器的思想，将两种augmentation产生的 $Q1$ 分别在 $S1$ 和 $S2$ 上进行分类，对 $Q2$ 也是如此，并使用上一步得到的概率分布计算loss，计算公式如下。

$$L_{mn} = \frac{1}{|\mathcal{Q}_m|} \sum_{(\mathbf{h}_i, y_i) \in \mathcal{Q}_m} -\log P(y = k \mid \mathbf{h}_i, \mathcal{T}(n))$$

Lmn

```python
def L_mn(self, X, y, c):
    # 输入一个 **query set**
    # 对于X，y中的每个样本都调用P函数，将结果取log并相加
    # 此处(X,y)与c属于相同或不同的augmentation方法！！！
    log_probs = 0.0
    for i in range(len(X)):
        # 调用P函数计算概率
        p = self.P(X[i], y[i], c)
        # 取对数并相加
        log_probs -= torch.log(p)

    return log_probs / X.shape[0]
```

因此我们对计算出的4个loss取均值，即可得到 L_{meta} 。

$$L_{meta} = \frac{1}{4} \sum_{m,n} L_{mn}$$

2.3.1.2 Distance-scaled Contrastive Loss

尽管我们使用了对比学习，但是仅仅在instance level应用并没有很好地利用小样本学习的特性，因此我们想要将其融入FSL中的episodic训练机制。我们首先拿来预训练过程中训练好的projection将样本投影到低维向量，接下来采用此前计算prototype的方法，对于投影之后的向量重新计算prototype。

prototype Python

```
support_X1 = self.projection.forward(support_X1)
support_X2 = self.projection.forward(support_X2)
o1 = self.crk(support_X1, support_y1)
o2 = self.crk(support_X2, support_y2)
```

接下来我们首先形成两个集合H和A，对于每个样本 z_i ， $H(z_i)$ 表示该样本的另一种augmentation方法 z_i' 、两个support set中所有同类样本的并，即

$$H(\mathbf{z}_i) = \{\mathbf{z}_i'\} \cup \mathcal{S}_1^{y_i} \cup \mathcal{S}_2^{y_i}$$

而 $A(z_i)$ 表示该样本的另一种augmentation方法 z_i' 、support set $S1$ 和 $S2$ 以及两个根据投影向量计算出的prototype o_1 和 o_2 ，即

$$A(\mathbf{z}_i) = \{\mathbf{z}_i'\} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\mathbf{o}_1^k\}_{k=1}^{M} \cup \{\mathbf{o}_2^k\}_{k=1}^{M}$$

set H & set A Python

```
A = torch.cat([support_X1, support_X2, o1, o2])

merged_support_X = torch.cat([support_X1, support_X2], dim=0)
merged_support_y = torch.cat([support_y1, support_y2], dim=0)

num_classes = torch.max(merged_support_y) + 1

H = []

for class_idx in range(num_classes):
    class_indices = torch.nonzero(merged_support_y == class_idx, as_tuple=True)[0]

    class_samples = merged_support_X[class_indices]
    H.append(class_samples)

H = torch.cat(H, dim=0)
```

接下来我们可以如下计算每个样本的监督对比损失。

$$L(\mathbf{z}_i) = -\sum_{\mathbf{z}_H \in H(\mathbf{z}_i)} \log \frac{\lambda_{\mathbf{z}_i \mathbf{z}_H} \exp\left(\mathbf{z}_i \cdot \mathbf{z}_H / \tau_5\right)}{\sum_{\mathbf{z}_A \in A(\mathbf{z}_i)} \lambda_{\mathbf{z}_i \mathbf{z}_A} \exp\left(\mathbf{z}_i \cdot \mathbf{z}_A / \tau_5\right)}$$

supervised contrastive loss Python

```python
def L_z(self, X, H, A, tau):
    # 这里输入的X应该是经过projection的单条样本

    # X与A中每个向量点乘，除以tau，取指数
    exp_a = torch.exp(torch.matmul(A, X) / tau)

    pdist = nn.PairwiseDistance(p=2)
    dists_a = pdist(X, A)

    lamb_a = torch.from_numpy(2 - dists_a.numpy())
    down = torch.sum(exp_a * lamb_a)

    exp_h = torch.exp(torch.matmul(H, X) / tau)

    pdist = nn.PairwiseDistance(p=2)
    dists_h = pdist(X, H)

    lamb_h = torch.from_numpy(2 - dists_h.numpy())
    up = exp_h * lamb_h

    return -torch.sum(torch.log(up / down))
```

在该步骤中，我们引入了系数 $\lambda_{z_i z_j}$ 和额外的两种view上的prototype o_r^k 以减少query和其它正例的相似度，因此模型可以学到不同任务之间更多可用于分类的表示。

接下来我们如下计算基于距离的对比损失。

$$L_{info} = \sum_{\mathbf{z}_i \in \mathcal{Q}_1 \cup \mathcal{Q}_2} \frac{1}{|H(\mathbf{z}_i)|} L(\mathbf{z}_i)$$

Linfo Python

```python
def L_info(self, support_X1, query_X1, support_y1, query_y1,
           support_X2, query_X2, support_y2, query_y2, tau):

    support_X1 = self.projection.forward(support_X1)
    support_X2 = self.projection.forward(support_X2)
    query_X1 = self.projection.forward(query_X1)
    query_X2 = self.projection.forward(query_X2)

    merged_query_X = torch.cat([query_X1, query_X2], dim=0)
    merged_query_y = torch.cat([query_y1, query_y2], dim=0)

    A = torch.cat([support_X1, support_X2,
                   self.crk(support_X1, support_y1),
                   self.crk(support_X2, support_y2)])

    merged_support_X = torch.cat([support_X1, support_X2], dim=0)
    merged_support_y = torch.cat([support_y1, support_y2], dim=0)

    # 获取类别的数量
    num_classes = torch.max(merged_support_y) + 1

    # 初始化一个空列表，用于存储每个类别的样本
    H = []

    # 遍历每个类别
    for class_idx in range(num_classes):
        # 找到同类样本的索引
        class_indices = torch.nonzero(merged_support_y == class_idx, as_tuple=True)[0]

        # 使用索引获取同类样本，并添加到结果列表
        class_samples = merged_support_X[class_indices]
        H.append(class_samples)

    # 将结果列表合并成一个张量
    H = torch.cat(H, dim=0)

    res = 0
    for i in range(len(merged_query_X)):
        res += self.L_z(merged_query_X[i], H[merged_query_y[i]], A, tau) \
               / H[merged_query_y[i]].shape[0]

    return res
```

通过优化 L_{info} ，我们促使模型在每个episode中提取分类信息，并学习在不同augmentation方法下同一类别样本的内在关联。

2.3.1.3 Objective in Meta-training

结合以上两个步骤，meta-training部分最终的目标优化函数如下。

$$L_{total} = L_{meta} + \beta L_{info}$$

2.3.2 meta-testing

在 `set_forward` 函数中设置前向进行meta-testing的过程。根据论文4.1节evaluation部分的内容，作者在计算最终结果时，采用计算prototype和query smaples之间欧式距离的方法。

meta-testing Python

```python
def set_forward(self, batch):
    image, global_target = batch
    image = image.to(self.device)
    (
        support_image,
        query_image,
        support_target,
        query_target,
    ) = self.split_by_episode(image, mode=2)
    support_size, _, c, h, w = support_image.size()
    query_size,_,c,h,w = query_image.size()

    support_output = self.emb_func(support_image)
    prototype = self.crk(support_output,support_target)
    output_list = []
    for i in range(query_size): # 对每一条query，找到最类似的某一类的prototype，然后预测
        episode_query_image = query_image[i].contiguous().reshape(-1, c, h, w)
        vec = self.emb_func(episode_query_image)
        maxp = 0
        for j in range(support_size):
            curp = self.P(vec,support_target[j],prototype)
            if curp > maxp:
               maxp = curp
               output = support_target[j]
        output_list.append(output)

        output = torch.cat(output_list, dim=0)
        acc = accuracy(output, query_target.contiguous().view(-1))
        return output, acc
```

计算欧式距离时，此处复用文章中对向量和prototype相似度的计算，使用欧氏距离来计算属于某一类的概率，采用预测概率最大的某一类作为预测类别。随后计算每个batch准确率。

3 LibFewShot框架适配

3.1 data augmentation

我们创建了ContrastiveLearningViewGenerator类，在core/data/collates/contrib中添加SimCLR_Style.py文件，其中base_transform就是Standard和SimCLR方法。

由于我们的data augmentation方法有改动，故我们更新了core/data/collates/contrib/__init__.py文件，添加上我们的方法。

```python
elif config["augment_method"] == "SimCLR_Style":
    """  own augment method, random two from SimCLR"""
    trfms_list = get_default_image_size_trfms(config["image_size"])
    base_transform = SimCLR_Style(config["image_size"]).base_transform
    trfms_list += [ContrastiveLearningViewGenerator(base_transform)]
elif config["augment_method"] == "Standard":
    trfms_list = get_default_image_size_trfms(config["image_size"])
    base_transform = Standard(config["image_size"]).base_transform
    trfms_list += [ContrastiveLearningViewGenerator(base_transform)]
```

3.2 fine tuning

我们首先创建了CL_PRETRAIN模型类，在core/model/finetuning/中添加cl_pretrain.py文件，其中emb_func为backbone中指定的resnet-12

class CL_PRETRAIN

```python
class CL_PRETRAIN(FinetuningModel):
    def __init__(self, feat_dim, num_class, inner_param, **kwargs):
        super(CL_PRETRAIN, self).__init__(**kwargs)
        self.feat_dim = feat_dim
        self.num_class = num_class
        self.inner_param = inner_param
        self.classifier = nn.Linear(feat_dim, num_class)
        self.loss_func = nn.CrossEntropyLoss()
        self.projection = MLP(feat_dim, num_class)
```

我们仿照了s2m2.py中的set_forward_adaptation方法，根据我们所使用的分类器和loss计算方法进行了改进

set_forward_adaptation Python

```python
def set_forward_adaptation(self, support_feat, support_target, query_feat):
    # 创建一个分类器，可以是模型的一部分，也可以是单独的模型
    classifier = self.classifier()  # 需要实现 create_classifier 方法
    # 将分类器移到设备上
    classifier = classifier.to(self.device)
    # 设置为训练模式
    classifier.train()
    # 获取支持集的大小
    support_size = support_feat.size(0)
    # 设置内部训练的优化器
    optimizer = self.sub_optimizer(classifier, self.inner_param["inner_optim"])
    # 迭代进行多次梯度更新
    for epoch in range(self.inner_param["inner_train_iter"]):
        # 随机排列索引
        rand_id = torch.randperm(support_size)
        # 按批次更新梯度
        for i in range(0, support_size, self.inner_param["inner_batch_size"]):
            optimizer.zero_grad()
            select_id = rand_id[i: min(i + self.inner_param["inner_batch_size"], support_size)]
            batch = support_feat[select_id]
            # 计算损失
            _, _, loss = self.set_forward_loss(batch)
            # 反向传播和梯度更新
            loss.backward()
            optimizer.step()

    # 在查询集上进行前向传播
    output = classifier(query_feat)

    return output
```

接下来我们编写了set_forward_loss函数，分别计算ce loss、global self-supervised contrastive loss、local self-supervised contrastive loss和global supervised contrastive loss，最后计算总loss与accuracy。详细代码见2.2。

3.3 meta learning

我们首先创建了CL_META模型类，在core/model/meta/中添加cl_meta.py文件，其中emb_func为backbone中指定的resnet-12。

class CL_META

Python

```python
class CL_META(MetaModel):

    def __init__(self, feat_dim, class_num, way_num, **kwargs):
        super(CL_META, self).__init__(**kwargs)
        self.feat_dim = feat_dim
        self.projection = MLP(feat_dim, 256)
        self.attention = Attn(feat_dim)

    def forward_output(self, x):
        feat_wo_head = self.emb_func(x)
        return feat_wo_head
```

由于我们数据预处理过程中包含了两类data-augmentation，且重新编写了loss函数计算方法，因此进行反向传播的过程有所改动。我们更新了set_forward_adaptation函数。

set_forward_adaptation Python

```python
    def set_forward_adaptation(self, support_X1, query_X1, support_y1, query_y1,
                               support_X2, query_X2, support_y2, query_y2):
        # 这里传入的是单个episode！！
        fast_parameters = list(item[1] for item in self.named_parameters())
        for parameter in self.parameters():
            parameter.fast = None
        self.emb_func.train()
        # self.classifier.train()

        features_support1 = self.forward_output(support_X1)
        features_support2 = self.forward_output(support_X2)
        features_query1 = self.forward_output(query_X1)
        features_query2 = self.forward_output(query_X2)

        tau5 = 0.1
        beta = 0.01

        L_meta = self.L_meta(features_support1, features_query1, support_y1, query_y1,
                             features_support2, features_query2, support_y2, query_y2)
        L_info = self.L_info(features_support1, features_query1, support_y1, query_y1,
                             features_support2, features_query2, support_y2, query_y2, tau5)

        loss = L_meta + beta * L_info
        grad = torch.autograd.grad(
            loss, fast_parameters, create_graph=True, allow_unused=True
        )
        fast_parameters = []

        for k, weight in enumerate(self.named_parameters()):
            if grad[k] is None:
                continue
            if weight[1].fast is None:
                weight[1].fast = weight[1] - lr * grad[k]
            else:
                weight[1].fast = weight[1].fast - lr * grad[k]
            fast_parameters.append(weight[1].fast)

        return loss, features_support1, features_support2
```

```
39
```

接下来我们编写了set_forward_loss函数，对于每个数据batch进行episode分割与训练，并计算loss与accuracy。

set_forward_loss Python

```python
def set_forward_loss(self, batch):
    images, _ = batch
    image1 = images[0:128]
    image1 = image1.to(self.device)
    (
        support_X1,
        query_X1,
        support_y1,
        query_y1,
    ) = self.split_by_episode(image1, mode=2)

    image2 = images[128:]
    image2 = image2.to(self.device)
    (
        support_X2,
        query_X2,
        support_y2,
        query_y2,
    ) = self.split_by_episode(image2, mode=2)

    episode_size, _, c, h, w = support_X1.size()

    output_list = []
    loss = []

    for i in range(episode_size):
        episode_support_X1 = support_X1[i].contiguous().reshape(-1, c, h, w)
        episode_support_X2 = support_X2[i].contiguous().reshape(-1, c, h, w)
        episode_query_X1 = query_X1[i].contiguous().reshape(-1, c, h, w)
        episode_query_X2 = query_X2[i].contiguous().reshape(-1, c, h, w)
        episode_support_y1 = support_y1[i].reshape(-1)
        episode_support_y2 = support_y2[i].reshape(-1)
        episode_query_y1 = query_y1[i].reshape(-1)
        episode_query_y2 = query_y2[i].reshape(-1)

        cur_loss, output1, output2 = self.set_forward_adaptation(episode_support_X1, episode_query_X1,
                                                                 episode_support_y1, episode_query_y1,
                                                                 episode_support_X2, episode_query_X2,
                                                                 episode_support_y2, episode_query_y2)
```

```
            loss.append(cur_loss)

            output_list.append(torch.cat((output1, output2)), dim=0)

        output = torch.cat(output_list, dim=0)

        loss = sum(loss) / len(loss)

        return output, acc, loss
```

同时我们在set_forward方法中编写了meta-test过程，详细代码见2.3.2。

4 复现难点与解决

4.1 data augmentation

4.1.1 self.times的理解

由于算法要求进行两种data augmentation，我们一开始的理解是返回list，但因为维度、Tensor等的问题，我们报错不断。

在重读LibFewShot的原始collate_function.py之后，我们发现我们一开始的理解有误，只需要将self.times置为2即可解决。

4.2 pretrain

4.2.1 计算各个loss

论文的pretraining部分需要计算多个loss，且计算前需要经过不同的层和不同的处理，如何计算各个loss然后组合起来是我们的一大难点

我们首先把需要的所有层都定义出来，然后定义每个计算loss的函数。在每个函数中让全局特征经过相应的处理后再通过数学公式进行计算。

4.2.2 indicator function函数的理解与处理

由于作者并没有详细解释 indicator function，所以给理解公式与复现带来了一些困难。而实现的时候，一开始的设想是用if判断，但是发现非常繁琐。

为解决该问题，首先，查找资料了解到指示器函数通常是一个条件函数，返回 1 或 0，表示某个条件是否满足。然后，在写代码时，放弃了直观上用if条件判断的写法，采用了PyTorch 的掩码矩阵操作。我们发现PyTorch 提供了一些方便的操作，可以使用掩码矩阵来实现指示器函数：使用 torch.eye 创建一个对角线元素为 1 的矩阵，然后使用masked_fill 将对角线元素置零。

4.2.3 Global supervised contrastive loss 的不同之处

$$\text{The set } P(i) \text{ contains the indexes of samples with the same label as } x_i \text{ in the augmented batch except for index } i.$$

意思是集合P（i） 包含除索引i之外，在扩充批次中与xi具有相同标签的样本的索引。这说明，此处的positive pair与前几个式子不太一样。在前几个loss的计算方法中，只有同一个样本被数据增强后得到的样本互为正样本对，但是此处的“same label”告诉我们，拥有相同标签也是正样本对。我们一开始没有注意到，后改正。

4.3 meta-learning

4.3.1 N-way K-shot测试原理

我们最初没有理解论文中meta-test的原理。原文描述为：遵循5-way 1-shot的分类任务，通过计算prototype与查询样本之间的欧氏距离来分类，其中每个类别包含15个样本。我们最初无法确定该prototype源自何处，经过理解meta-test的原理之后我们得出，测试时需要先将support set放入模型后得到5个类别的prototype，再计算query set中各个样本与每一个prototype的欧式距离，得到最小的欧氏距离所属类别即为该测试样本预测类别。

4.3.2 loss计算

meta部分也涉及到了多种loss组合计算问题。如何正确理解公式，按照正确维度的向量计算loss是一件很困难的事情。在实现时经过多次修正、维度测试、打印变量才找到实现中的错误所在。

4.3.3 MLP和Attn引入

在meta-learning过程中，我们需要引入含有一层隐藏层的MLP以及多注意力头。引用的论文是使用tensorflow框架写的，因此我们需要找到在pytorch下相同的实现方式。同时论文中并没有给出经过一层隐藏层后的维度，因此需要我们自行实现这些细节。

通过查找资料，我们发现MLP的实现可以通过torch中的linear线性层加激活函数实现。输入维度为通过backbone且经过GAP层后得到的640维向量，输出维度则固定为256维。

Attn使用pytorch库中的multiheadattention实现，将注意力头的数目固定为1，然后获取经过对齐的prototype用于loss计算。

5 复现结果

5.1 运行配置

囿于篇幅问题，此处只附核心配置。

5.1.1 pretrain

```yaml
# data 相关配置
data_root: ./data/miniImageNet--ravi
augment: True
augment_times: 2
augment_times_query: 2
augment_method: SimCLR_Style

# classifier
classifier:
  kwargs:
    feat_dim: 640
    num_class: 100
    inner_param:
      inner_batch_size: 4
      inner_optim:
        kwargs:
          dampening: 0.9
          lr: 0.1
          momentum: 0.9
          weight_decay: 5e-4
        name: SGD
      inner_train_iter: 300
  name: CL_PRETRAIN

# optimizer
lr_scheduler:
  kwargs:
    gamma: 0.5
    step_size: 40
  name: StepLR

optimizer:
  kwargs:
    betas:
    - 0.5
    - 0.9
    lr: 0.001
  name: Adam
  other: null
...
...
```

5.1.2 meta-learning

YAML

```yaml
# data 相关配置
data_root: ./data/miniImageNet--ravi
augment: True
augment_times: 2
augment_times_query: 2
augment_method: SimCLR_Style

# classifier
classifier:
  kwargs:
    feat_dim: 640
    class_num: 100
    way_num:  5
  name: CL_META

# model
epoch: 100
test_epoch: 5
parallel_part:
  - emb_func
pretrain_path:
resume: True
way_num: 5
shot_num: 1
query_num: 15
test_query: 15
test_shot: 1
test_way: 5
episode_size: 1 # TODO:
test_episode: 2000
train_episode: 4000
batch_size: 128

# optimizer
lr_scheduler:
  kwargs:
    gamma: 0.5
    step_size: 40
  name: StepLR

optimizer:
  kwargs:
    weight_decay: 5e-4
    lr: 0.1
    momentum: 0.9
```

```
    name: SGD
    other: null
```

5.1.3 环境

在线linux服务器，操作系统为Ubuntu22.04 ，gpu为NVIDIA GeForce RTX 4090，Driver版本535.86.05，CUDA Version: 12.2

5.2 运行结果

受制于算力资源限制，我们仅在miniImageNet--ravi数据集上进行了模型的训练与5-way 1-shot测试，最终得到的平均模型准确率结果为72.758%。这与原论文结果中的77.56%十分接近。

```
2024-01-01 15:26:04,873 [INFO] core.trainer:  * Acc@1 72.758 Best acc 72.027
2024-01-01 15:26:04,875 [INFO] core.trainer:  * Time: 0:16:34/0:16:34
2024-01-01 15:26:05,860 [INFO] core.trainer: End of experiment, took 0:16:34
```

模型存放在results

5.3 结论

实验中，我们将对比学习应用到FSL的两阶段训练范式中，以缓解表征泛化能力的限制。在预训练阶段，我们使用vector-map和map-map模块来引入自我监督的对比损失。为了将对比学习有效地扩展到元训练阶段，我们进一步提出了CVET策略和远距离尺度的对比损失。实验结果表明，该方法在数据集miniImageNet--ravi上取得了非常好的性能。