Supplementary for pSTarC: Pseudo Source Guided Target Clustering for Fully Test-Time Adaptation

Manogna Sreenivas[†], Goirik Chakrabarty[*], Soma Biswas[†]
[†]IISc Bangalore [*]IISER Pune
{manognas, somabiswas}@iisc.ac.in goirik.chakrabarty@students.iiserpune.ac.in

1. Augmentations

We use the same augmentations as used in AdaContrast [1] here. We explicitly report the series of augmentations done along with their ranges for better reproducibility.

```
from torchvision import transforms

class GaussianBlur(object):
    def __init__(self, sigma=[0.1, 2.0]):
        self.sigma = sigma

    def __call__(self, x):
        sigma = random.uniform(self.sigma[0],
    self.sigma[1])
        x = x.filter(ImageFilter.GaussianBlur(
    radius=sigma))
        return x

transform_list = [
    transforms.RandomResizedCrop(crop_size, scale
    =(0.2, 1.0)),
    transforms.RandomApply(
    [transforms.ColorJitter(0.4, 0.4, 0.4, 0.1)],
    p=0.8),
    transforms.RandomGrayscale(p=0.2),
    transforms.RandomApply([GaussianBlur([0.1,
    2.0])], p=0.5),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()
]
```

2. Choice of parameter n_c

For pseudo source feature generation, we set the total number of features N as $C \times n_c$ where C is the number of classes and n_c is the number of features we expect to be generated per class. During test time adaptation, we set the number of positives K as 5 for all experiments. So, ensuring the generated feature bank to contain 5 samples per class should suffice for the algorithm to work well without any significant degradation in accuracy.

We use an Adam optimizer with a learning rate of 0.01 and optimize the feature bank for 50 steps. However, as we are optimising the feature bank, naively setting $N = 5 \times C$ may not ensure there are adequate number of features (5 in this case) per class. The optimum for the second term L_{div} in the loss occurs only when there are equal number of samples in each class, i.e., when the class distribution become uniform. This can not always be guaranteed, while using the same optimization parameters across datasets. For example, even setting 120 as the number of features for VisDA with 12 classes, atleast 5 features per class were generated. But for DomainNet-126, using the same optimization scheme, we observed some classes had less than 5 features generated. Instead of tuning the optimizer hyperparameters for each dataset, we set n_c sufficiently large (20 here) so that it can be used across all datasets. We observe that on setting N as $20 \times C$ features, for all datasets, using the same optimizer parameters and number of steps, we could ensure atleast 5 features per class were generated to enable using them seamlessly during TTA. Hence, we set N to 20 features per class for all datasets, VisDA (with 12 classes), DomainNet-126 (with 126 classes), Office-Home (with 65 classes) and CIFAR-100 (with 100 classes).

3. Pseudo Code for pSTarC

```
def generate_features(args, netC, num_features
    =100, num_epochs=50, feature_dim=256):
    netC.train()
    pseudo_features = torch.randn((args.class_num
     * num_features, feature_dim)).cuda()
    pseudo_features.requires_grad = True
    optim_feats  = optim.Adam ([pseudo_features],
     lr =0.01)
    for t in range(num_epochs):
        optim_feats.zero_grad()
        scores = nn.Softmax(dim=1)(netC(
    pseudo_features))
        loss_ent = torch.mean(Entropy(scores))
        loss_div = +torch.sum(torch.mean(scores,
    0) * torch.log(torch.mean(scores, 0) + 1e-6))
        loss = loss_ent + loss_div *5
        loss.backward()
        optim_feats.step()
    return pseudo_features
```

```
def pstarc(args, test_loader, netFE, netC):

    netC.train()

    # generate pseudo source features
    pseudo_feats = generate_features(args, netC,
    num_features=20, num_epochs=50, feature_dim=
    256)
    pseudo_scores = nn.Softmax(dim=1)(netC(
    pseudo_feats))
    pseudo_maxprobs, pseudo_label_bank = torch.
    max(pseudo_scores, dim=1)

    fea_bank = pseudo_feats.cpu()
    fea_bank = torch.nn.functional.normalize(
    fea_bank)
    score_bank = pseudo_scores
    label_bank = pseudo_label_bank.cpu()

    optimizer = optim.SGD(netC.parameters(), lr
    =5e-4, momentum = 0.9, weight_decay = 0,
    nesterov = True)

    iter_test = iter(loader)

    for i in range(len(loader)):
        inputs, labels = next(iter_test)
        # Get image and its strong augmentation
        image, image_s = inputs

        netFE.train()

        # get image features and its prediction
    vectors
        features = netFE(image)
        probs_image = nn.Softmax(dim=1)(netC(
    features))

        # get strong augmented image features and
     its prediction vectors
        features_s = netFE(image_s)
        probs_image_s = nn.Softmax(dim=1)(
    features_s)


        # compute sample-wise entropy
        ent_batch = Entropy(p_image)
        # get dynamic threshold to select low
    entropy samples
        ent_thresh = torch.mean(ent_batch)

        # Computer L_aug
        loss_aug = - torch.sum(probs_image *
    probs_image_s, dim=1)

        with torch.no_grad():
            f_norm = torch.nn.functional.
    normalize(features)

            score_near = torch.zeros(image.shape
    [0], K, class_num).cuda()
            score_near_cls = torch.zeros(image.
    shape[0], K, class_num).cuda()
            for c in range(args.class_num):
                # get pseudo source features and
    current batch features belonging to class c
                src_cls_feats = torch.nn.
functional.normalize(fea_bank[label_bank==c])
                src_cls_scores = score_bank[
label_bank==cls_idx]
                curr_cls_feats = output_f_[
pseudo_label.cpu()==cls_idx]
                curr_cls_probs = max_prob[
pseudo_label.cpu()==cls_idx]

                # Retrieve top K pseudo source
features for each test sample
                cos_sim = curr_cls_feats @
src_cls_feats.T
                cls_dist_near, cls_idx_near =
torch.topk(cls_dist, dim=-1, largest=True, k=
K + 1)
                cls_dist_near, cls_idx_near =
cls_dist_near[:, 1:], cls_idx_near[:, 1:]
                cls_score_near = src_cls_scores[
cls_idx_near]
                score_near_cls[pseudo_label.cpu()
==cls_idx] = cls_score_near

            # select pseudo source samples as
positive for low entropy samples
            score_near[ent_batch<ent_thresh] =
score_near_cls[ent_batch<ent_thresh]
            # self anchor the low entropy samples
            score_near[ent_batch>ent_thresh] = (
probs_image[ent_batch>ent_thresh]).detach().
clone().unsqueeze(1).expand(-1, args.K, -1)

        # repeat probs_image(of dimension batch x
 C) K times
        probs_image_un = probs_image.unsqueeze(1)
.expand(-1, args.K, -1)  # batch x K x C

        # Computer the attraction loss L_attr
        loss_postive = -(probs_image_un *
score_near).sum(-1).sum(1)

        # Computer the dispersion loss L_attr
        mask = torch.ones((features_w.shape[0],
features_w.shape[0]))
        diag_num = torch.diag(mask)
        mask_diag = torch.diag_embed(diag_num)
        mask = mask - mask_diag
        copy = softmax_out.T
        dot_neg = softmax_out @ copy  # batch x
batch
        loss_negative = (dot_neg * mask.cuda()).
sum(-1)  # batch

        loss = torch.mean(loss_aug + loss_attr +
loss_disp)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        netFE.eval()
        outputs = netC(netFE(image))
        predictions = torch.max(outputs, dim=1)

return
```

L_{aug}	L_{attr}	L_{disp}	R→C	R→P	P→C	C→S	S→P	R→S	P→R	Average
✓	✓		57.1	66.6	57.1	47.8	54.7	54.1	74.0	58.8
✓		✓	56.0	63.6	56.0	51.8	61.7	49.9	78.7	59.7
	✓	✓	60.1	67.3	59.6	55.0	64.8	54.6	79.8	63.0
✓	✓	✓	**60.8**	**67.7**	**60.3**	**55.6**	**65.3**	**55.8**	**80.2**	**63.7**

Table 1. pSTarC ablation study: Importance of each loss term.

Method	R→C	R→P	P→C	C→S	S→P	R→S	P→R	Average
8	44.9	54.7	45.8	44.1	52.1	42.2	66.9	50.1
16	54.4	62.8	54.6	49.6	59.3	49.3	75.2	57.9
32	57.8	65.4	58.1	52.0	61.9	52.9	77.6	60.8
64	60.0	66.9	59.7	53.6	63.7	54.3	78.6	62.4
128	60.1	67.1	60.2	54.4	64.2	54.3	78.7	62.4

Table 2. Total accuracy (%) of AdaContrast on varying batch sizes.

Method	R→C	R→P	P→C	C→S	S→P	R→S	P→R	Average
8	53.5	62.6	51.2	41.2	54.1	46.2	69.9	54.1
16	56.5	65.7	56.2	49.6	59.8	51.4	75.5	59.2
32	59.4	67.2	58.0	51.1	61.9	54.5	77.0	61.3
64	61.6	68.9	60.5	54.7	64.3	57.0	79.4	63.8
128	60.8	67.7	60.3	55.6	65.3	55.8	80.2	63.7

Table 3. Total accuracy (%) of pSTarC on varying batch sizes.

4. Analysis on DomainNet-126

Here, we provide detailed analysis done on DomainNet-126. The paper results the average accuracy across the seven domains in Table 7 and 8 in the main paper. Here, we provide the results for each domain shift on performing ablation on loss components in Table 1. We also report detailed results of AdaContrast (Table 2) and pSTarC (Table 3) on varying batch sizes.

References

[1] Dian Chen, Dequan Wang, Trevor Darrell, and Sayna Ebrahimi. Contrastive test-time adaptation. In *CVPR*, 2022.