22336216_陶宇卓_模式识别_homework3

实验目的

1. 理解并掌握 MixMatch 和 FixMatch 两种主流半监督学习算法的原理与实现方法。
2. 掌握半监督学习在小样本标注场景下的应用与实验流程。
3. 对比自实现算法与 TorchSSL 官方实现的性能差异，分析原因。
4. 总结两种算法的异同点及其在实际应用中的优缺点。

实验内容

本实验基于 PyTorch 框架，分别实现了 MixMatch 和 FixMatch 两种半监督图像分类算法，并在 CIFAR-10 数据集上进行实验。实验采用 WideResNet-28-2 作为主干网络，分别在 40、250、4000 张标注数据的情况下，测试算法性能。同时，使用 TorchSSL 工具箱中的 MixMatch 和 FixMatch 实现进行对比，分析不同实现下的实验效果。

实验步骤与原理

1. 数据集处理与增强

- **数据划分**：在 `dataset.get_cifar10` 中，随机选取指定数量的有标签样本（如 40、250、4000），其余作为无标签样本。保证每类均匀采样，避免类别不平衡。

```python
def get_cifar10(args, root):
    transform_labeled = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(size=32,
                              padding=int(32*0.125),
                              padding_mode='reflect'),
        transforms.ToTensor(),
        transforms.Normalize(mean=cifar10_mean, std=cifar10_std)
    ])
    transform_val = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=cifar10_mean, std=cifar10_std)
    ])
    base_dataset = datasets.CIFAR10(root, train=True, download=True)

    train_labeled_idxs, train_unlabeled_idxs = x_u_split(
        args, base_dataset.targets)

    train_labeled_dataset = CIFAR10SSL(
        root, train_labeled_idxs, train=True,
        transform=transform_labeled)

    train_unlabeled_dataset = CIFAR10SSL(
        root, train_unlabeled_idxs, train=True,
        transform=TransformFixMatch(mean=cifar10_mean, std=cifar10_std))

    test_dataset = datasets.CIFAR10(
        root, train=False, transform=transform_val, download=False)

    return train_labeled_dataset, train_unlabeled_dataset, test_dataset
```

- **数据增强**：
  - MixMatch：transform_train 包含随机裁剪（RandomPadandCrop）、随机翻转（RandomFlip）、归一化（ToTensor）。

```
class RandomPadandCrop(object)
class RandomFlip(object)
class GaussianNoise(object)
class ToTensor(object)
```

  - FixMatch：弱增强同上，强增强使用 RandAugment（`RandAugment`），在 `randaugment.py` 类中实现，在 `TransformFixMatch` 类中调用。

```python
class TransformFixMatch(object):
    def __init__(self, mean, std):
        self.weak = transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.RandomCrop(size=32,
                                  padding=int(32*0.125),
                                  padding_mode='reflect')])
        self.strong = transforms.Compose([
            transforms.RandomHorizontalFlip(),
            transforms.RandomCrop(size=32,
                                  padding=int(32*0.125),
                                  padding_mode='reflect'),
            RandAugment(n=2, m=10)])
        self.normalize = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize(mean=mean, std=std)])

    def __call__(self, x):
        weak = self.weak(x)
        strong = self.strong(x)
        return self.normalize(weak), self.normalize(strong)
```

2. 网络结构

- 两种方法均采用 WideResNet-28-2 作为主干网络，在 models.build_wideresnet 中构建，参数设置为 depth=28, widen_factor=2, dropout=0.0。

```python
class WideResNet(nn.Module):
    def __init__(self, num_classes, depth=28, widen_factor=2, drop_rate=0.0):
        super(WideResNet, self).__init__()
        channels = [16, 16*widen_factor, 32*widen_factor, 64*widen_factor]
        assert((depth - 4) % 6 == 0)
        n = (depth - 4) / 6
        block = BasicBlock
        self.conv1 = nn.Conv2d(3, channels[0], kernel_size=3, stride=1,
                               padding=1, bias=False)
        self.block1 = NetworkBlock(
            n, channels[0], channels[1], block, 1, drop_rate, activate_before_residual=True)
        self.block2 = NetworkBlock(
            n, channels[1], channels[2], block, 2, drop_rate)
        self.block3 = NetworkBlock(
            n, channels[2], channels[3], block, 2, drop_rate)
        self.bn1 = nn.BatchNorm2d(channels[3], momentum=0.001)
        self.relu = nn.LeakyReLU(negative_slope=0.1, inplace=True)
        self.fc = nn.Linear(channels[3], num_classes)
        self.channels = channels[3]

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight,
                                        mode='fan_out',
                                        nonlinearity='leaky_relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1.0)
                nn.init.constant_(m.bias, 0.0)
            elif isinstance(m, nn.Linear):
                nn.init.xavier_normal_(m.weight)
                nn.init.constant_(m.bias, 0.0)

    def forward(self, x):
        out = self.conv1(x)
        out = self.block1(out)
        out = self.block2(out)
        out = self.block3(out)
        out = self.relu(self.bn1(out))
        out = F.adaptive_avg_pool2d(out, 1)
        out = out.view(-1, self.channels)
        return self.fc(out)
```

3. MixMatch 算法实现

3.1 伪标签生成与增强

- 在 train 函数中，对每个无标签样本做两次不同的数据增强，分别输入模型，得到两个输出 outputs_u 和 outputs_u2。
- 伪标签生成代码如下：

```
with torch.no_grad():
    outputs_u = model(inputs_u)
    outputs_u2 = model(inputs_u2)
    p = (torch.softmax(outputs_u, dim=1) + torch.softmax(outputs_u2, dim=1)) / 2
    pt = p**(1/args.T)
    targets_u = pt / pt.sum(dim=1, keepdim=True)
    targets_u = targets_u.detach()
```

这里对两个增强的输出取平均，温度缩放后归一化，得到无标签样本的伪标签。

3.2 MixUp混合

- 有标签和无标签样本及其标签拼接后，进行 MixUp：

```
all_inputs = torch.cat([inputs_x, inputs_u, inputs_u2], dim=0)
all_targets = torch.cat([targets_x, targets_u, targets_u], dim=0)
l = np.random.beta(args.alpha, args.alpha)
l = max(l, 1-l)
idx = torch.randperm(all_inputs.size(0))
input_a, input_b = all_inputs, all_inputs[idx]
target_a, target_b = all_targets, all_targets[idx]
mixed_input = l * input_a + (1 - l) * input_b
mixed_target = l * target_a + (1 - l) * target_b
```

这样可以提升模型的泛化能力。

3.3 损失函数与RampUp

- 损失函数在 SemiLoss 类中实现：

```
class SemiLoss(object):
    def __call__(self, outputs_x, targets_x, outputs_u, targets_u, epoch):
        probs_u = torch.softmax(outputs_u, dim=1)
        Lx = -torch.mean(torch.sum(F.log_softmax(outputs_x, dim=1) * targets_x, dim=1))
        Lu = torch.mean((probs_u - targets_u)**2)
        return Lx, Lu, args.lambda_u * linear_rampup(epoch)
```

- 有标签损失 Lx 为交叉熵。
- 无标签损失 Lu 为均方误差（MSE）。
- linear_rampup 控制无标签损失权重随训练线性增长：

```
def linear_rampup(current, rampup_length=args.epochs*0.4):
    if rampup_length == 0:
        return 1.0
    else:
        current = np.clip(current / rampup_length, 0.0, 1.0)
        return float(current)
```

3.4 优化与EMA

- 优化器采用 Adam，`optimizer = optim.Adam(model.parameters(), lr=0.002, weight_decay=0.0005)`。
- 使用 WeightEMA 类对模型参数做指数滑动平均，提升模型稳定性。

4. FixMatch 算法实现（简述）

4.1 有标签损失

有标签样本直接计算交叉熵损失：

```
Lx = F.cross_entropy(logits_x, targets_x)
```

4.2 无标签损失（核心：一致性正则化）

对无标签样本，先用弱增强输入模型，生成伪标签：

```
pseudo_label = torch.softmax(logits_u_w.detach() / args.T, dim=-1)
max_probs, targets_u = torch.max(pseudo_label, dim=-1)
mask = max_probs.ge(args.threshold).float()
```

- `logits_u_w`：无标签样本弱增强的输出
- `pseudo_label`：温度缩放后的 softmax 概率
- `targets_u`：置信度最高的类别
- `mask`：置信度大于阈值（0.95）的位置

用强增强输入模型，计算一致性损失（只对高置信伪标签）：

```
Lu = (F.cross_entropy(logits_u_s, targets_u, reduction='none') * mask).mean()
```

- `logits_u_s`：无标签样本强增强的输出
- 只对 mask=1 的样本计算损失

4.3 总损失

总损失为有标签损失和无标签损失加权和：

```
loss = Lx + args.lambda_u * Lu
```

- `args.lambda_u` 控制无标签损失权重（通常为 1）

4.4 优化器与 EMA

- 优化器我使用了 SGD（`optim.SGD`），学习率 0.03，momentum 0.9，weight_decay 0.0005。
- 支持 EMA（指数滑动平均）提升模型稳定性。

4.5 训练循环

- 每轮训练，采样有标签和无标签 batch，分别做增强、前向传播、损失计算、反向传播和参数更新。
- 每隔一定步数评估模型，保存最优权重。

4.6 代码流程总结

```
try:
    inputs_x, targets_x = next(labeled_iter)
except:
    labeled_iter = iter(labeled_trainloader)
    inputs_x, targets_x = next(labeled_iter)

try:
    (inputs_u_w, inputs_u_s), _ = next(unlabeled_iter)
except:
    unlabeled_iter = iter(unlabeled_trainloader)
    (inputs_u_w, inputs_u_s), _ = next(unlabeled_iter)

data_time.update(time.time() - end)
batch_size = inputs_x.shape[0]
inputs = interleave(
    torch.cat((inputs_x, inputs_u_w, inputs_u_s)), 2*args.mu+1).to(args.de
vice)
targets_x = targets_x.to(args.device)
logits = model(inputs)
logits = de_interleave(logits, 2*args.mu+1)
logits_x = logits[:batch_size]
logits_u_w, logits_u_s = logits[batch_size:].chunk(2)
del logits

Lx = F.cross_entropy(logits_x, targets_x, reduction='mean')
# 一致性正则化
pseudo_label = torch.softmax(logits_u_w.detach()/args.T, dim=-1)
max_probs, targets_u = torch.max(pseudo_label, dim=-1)
mask = max_probs.ge(args.threshold).float()

Lu = (F.cross_entropy(logits_u_s, targets_u,
                      reduction='none') * mask).mean()

loss = Lx + args.lambda_u * Lu

if args.amp:
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
else:
    loss.backward()

losses.update(loss.item())
losses_x.update(Lx.item())
losses_u.update(Lu.item())
optimizer.step()
scheduler.step()
if args.use_ema:
```

```
            ema_model.update(model)
        model.zero_grad()
```

5. 训练与评估流程

- 每轮训练包括有标签和无标签数据的采样、增强、混合、损失计算与反向传播。
- 每个 epoch 结束后，分别在训练集、验证集、测试集上评估模型性能，记录准确率与损失。

结果对比与分析

1. 实验设置

- 统一 **batch size=64，训练迭代数=20000**。
- 对每组标注数量（40、250、4000）分别实验，记录 top-1 准确率。
- 另外，因为设备显存原因， μ **最大只能设置为 3** ，与论文中所提到的**最佳效果时的 7** 有明显差距。

2. 实验结果

方法	标注数	自实现 Top-1 Acc	TorchSSL Top-1 Acc
MixMatch	40	32.54%	20.27%
MixMatch	250	68.83%	70.01%
MixMatch	4000	83.62%	87.06%
FixMatch	40	45.35%	37.45%
FixMatch	250	81.22%	77.44%
FixMatch	4000	90.56%	88.74%

3. 结果对比与分析

- **准确率趋势**：标注样本越多，准确率越高，FixMatch 通常优于 MixMatch。
- **自实现 vs TorchSSL**：
  - 自实现效果优于 TorchSSL，可能因为数据增强细节更贴近论文、损失函数实现更精确。
- **一致性正则化作用**：FixMatch 通过高置信伪标签和强增强，提升了无标签数据的利用效率，表现更优。
- **MixMatch 特点**：MixUp 和伪标签平均化提升了泛化能力，但对无标签数据的利用率略低于 FixMatch。

4. MixMatch 与 FixMatch 的异同点

方面	MixMatch	FixMatch
伪标签生成	多次增强平均 softmax，温度缩放	弱增强 softmax，置信度阈值筛选
无标签损失	均方误差（MSE）	交叉熵（仅高置信伪标签）
数据增强	普通增强	弱增强+强增强（RandAugment）
MixUp	有	无
一致性正则化	间接（通过 MixUp）	直接（强增强与弱增强伪标签一致性）
优化器	Adam	SGD
超参数敏感性	对 alpha、lambda-u、T 较敏感	对 threshold、lambda-u、T 较敏感

5. 论文复现（部分）

我还尝试了复现论文的迭代次数（2^{20}），部分结果如下（耗时太长了跑不完）：

方法	标注数	自实现 Top-1 Acc
FixMatch	40	85.2%

实验结论

- **MixMatch 和 FixMatch 都能在极少标注样本下取得远超监督学习的效果**，但 FixMatch 通常表现更优。
- **实现细节对最终效果影响极大**，如数据增强、伪标签生成、损失权重等。
- **TorchSSL 提供了标准实现，便于对比和复现。**

代码使用方法简述

1. MixMatch/FixMatch 自实现

```
python fixmatch.py --num-labeled 40 --seed 5 --out results/fixmatch/cifar10@40.5
python fixmatch.py --num-labeled 250 --seed 5 --out results/fixmatch/cifar10@250.5
python fixmatch.py --num-labeled 4000 --seed 5 --out results/fixmatch/cifar10@4000.5
python mixmatch.py --seed 5  --num-labeled 40 --out ./results/mixmatch/cifar10@40.5
python mixmatch.py --seed 5  --num-labeled 250 --out ./results/mixmatch/cifar10@250.5
python mixmatch.py --seed 5  --num-labeled 4000 --out ./results/mixmatch/cifar10@4000.5
```

2. TorchSSL 运行

```
python fixmatch.py --c config/fixmatch/fixmatch_cifar10_40_0.yaml
python fixmatch.py --c config/fixmatch/fixmatch_cifar10_250_0.yaml
python fixmatch.py --c config/fixmatch/fixmatch_cifar10_4000_0.yaml
python mixmatch.py --c config/mixmatch/mixmatch_cifar10_40_0.yaml
python mixmatch.py --c config/mixmatch/mixmatch_cifar10_250_0.yaml
python mixmatch.py --c config/mixmatch/mixmatch_cifar10_4000_0.yaml
```

参考文献

1. Berthelot, D., et al. "MixMatch: A Holistic Approach to Semi-Supervised Learning." NeurIPS 2019.
2. Sohn, K., et al. "FixMatch: Simplifying Semi-Supervised Learning with Consistency and Confidence." NeurIPS 2020.
3. TorchSSL:

GitHub – TorchSSL/TorchSSL: A PyTorch-based library for semi-supervised learning (NeurIPS'21)

4. FixMatch: [pytorch]FixMatch代码详解-数据加载-CSDN博客
5. Mixmatch: MixMatch文章解读+算法流程+核心代码详解-CSDN博客