半监督图像分类

21310330 李涛 计算机科学与技术

实验目的

神经网络模型通常需要大量标记好的训练数据来训练模型。然而，在许多情况下，获取大量标记好的数据可能是困难、耗时或昂贵的。这就是半监督学习的应用场景。半监督学习的核心思想是利用无标记数据的信息来改进模型的学习效果。在半监督学习中，我们使用少量标记数据进行有监督学习，同时利用大量无标记数据的信息。通过充分利用无标记数据的潜在结构和分布特征，半监督学习可以帮助模型更好地泛化和适应未标记数据。

实验要求

1. 阅读原始论文和相关参考资料，基于 Pytorch 分别实现 MixMatch 和 FixMatch半监督图像分类算法，按照原始论文的设置，MixMatch 和 FixMatch 均使用 WideResNet-28-2 作为 Backbone 网络，即深度为 28，扩展因子为 2， 在CIFAR-10 数据集上进行半监督图像分类实验，报告算法在分别使用 40, 250,4000 张标注数据的情况下的图像分类效果（标注数据随机选取指定数量）
2. 使用 TorchSSL[6]中提供的 MixMatch 和 FixMatch 的实现进行半监督训练和测试，对比自己实现的算法和 TorchSSL 中的实现的效果
3. 提交源代码，不需要包含数据集，并提交实验报告，实验报告中应该包含代码的使用方法，对数据集数据的处理步骤以及算法的主要实现步骤，并分析对比 MixMatch 和 FixMatch 的相同点和不同点。

实验内容

一．实现 MixMatch半监督图像分类算法

①数据集加载和数据增强

先加载CIFAR-10数据集然后将数据集分为有标签、无标签的训练集和有标签的测试集，有标签的训练集最好保证每一类都有，且数量平均。

```
# 加载CIFAR-10数据集并创建有标签和无标签的子集
def load_cifar10(batch_size, num_labeled_samples, num_iterations):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    full_train_dataset = CIFAR10(root='./data', train=True, download=True,
transform=transform)
    test_dataset = CIFAR10(root='./data', train=False, download=True,
transform=transform)

    class_indices = [[] for _ in range(10)]
    for idx, label in enumerate(full_train_dataset.targets):
        class_indices[label].append(idx)

    labeled_indices = []
    unlabeled_indices = []

    num_samples_per_class = num_labeled_samples // 10
    for indices in class_indices:
        random.shuffle(indices)
        labeled_indices.extend(indices[:num_samples_per_class])
        unlabeled_indices.extend(indices[num_samples_per_class:])

    random.shuffle(labeled_indices)
    random.shuffle(unlabeled_indices)

    labeled_dataset = Subset(full_train_dataset, labeled_indices)
    unlabeled_dataset = Subset(full_train_dataset, unlabeled_indices)

    total_labeled_samples = num_iterations * batch_size
    total_unlabeled_samples = num_iterations * batch_size

    labeled_sampler = RandomSampler(labeled_dataset, replacement=True,
num_samples=total_labeled_samples)
    unlabeled_sampler = RandomSampler(unlabeled_dataset, replacement=True,
num_samples=total_unlabeled_samples)

    labeled_loader = DataLoader(labeled_dataset, batch_size=batch_size,
sampler=labeled_sampler)
```

```
    unlabeled_loader = DataLoader(unlabeled_dataset, batch_size=batch_size,
sampler=unlabeled_sampler)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return labeled_loader, unlabeled_loader, test_loader
```

接下来的的大致框架和正常机器学习类似，只是对无标签的数据有些特别。

其中还有一个数据加强的部分：

```
# 数据增强
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

其中包括：

1. `transforms.RandomHorizontalFlip()`：随机水平翻转图像，这可以帮助模型更好地理解左右对称的特征。
2. `transforms.RandomCrop(32, padding=4)`：随机裁剪图像，并在裁剪之前对图像进行填充。这可以增加图像的多样性，使模型更能适应不同的裁剪位置和尺寸。
3. `transforms.ToTensor()`：将图像转换为张量，这是PyTorch中处理图像数据的基本操作。
4. `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`：对图像进行标准化处理，使每个通道的像素值在0.5均值和0.5标准差的范围内分布。这有助于加快模型的训练速度和收敛性。

通过这些数据增强技术，可以大大提升模型在CIFAR-10数据集上的性能和鲁棒性。

②熵最小化

熵最小化其实就是把无标签的数据丢原始模型中去预测一下，即对无标注数据进行 K 次增广，并获得对应的 K 个输出。然后使用"sharpening"函数来对无标注数据的平均预测进行锐化。

```
def guess_labels(model, unlabeled_data, num_augmentations, device):
    """使用模型预测为无标签数据猜测标签。"""
    model.eval()
    with torch.no_grad():
        outputs = [model(unlabeled_data.to(device)) for _ in
range(num_augmentations)]
```

```
        outputs = torch.stack(outputs)
        avg_predictions = outputs.mean(0)
        sharpened_predictions = sharpen(avg_predictions, T=0.5)

        # 统计伪标签分布
        pseudo_labels = sharpened_predictions.argmax(dim=1).cpu().numpy()
        unique, counts = np.unique(pseudo_labels, return_counts=True)
#         print("Pseudo-label distribution: ", dict(zip(unique, counts)))

        return sharpened_predictions
```

③sharpening

按公式

$$\mathrm{Sharpen}(p, T)_i = \frac{p_i^{\frac{1}{T}}}{\sum_{j=1}^{L} p_j^{\frac{1}{T}}}$$

进行代码书写即可：

```
def sharpen(p, T=0.5):
    temp = p ** (1 / T)
    temp /= temp.sum(dim=1, keepdim=True)
    return temp
```

④对有标签数据进行one-hot编码

由于sharpening产生的伪标签的是对每个数据的概率预测，所以有标签的数据应该也变成这种格式，由于有标签的数据集标签是确定的，所以直接对其进行one-hot编码即可，即对于标签位置为，其余全为0。

```
            # 将labels进行one-hot编码
            one_hot_labels = F.one_hot(labels, num_classes=10).float()
```

⑤Mixup构建完成新数据集

mixup过程如下：

$$\lambda \sim \mathrm{Beta}(\alpha, \alpha)$$

$$\lambda' = \max(\lambda, 1-\lambda)$$

$$\mathrm{x}' = \lambda' \mathrm{x}_1 + (1-\lambda')\mathrm{x}_2$$

$$\mathrm{p}' = \lambda' \mathrm{p}_1 + (1-\lambda')\mathrm{p}_2$$

```
# MixUp数据增强
def mixup_data(x, y, alpha=1.0, use_cuda=True):
    if alpha > 0:
        lam = np.random.beta(alpha, alpha)
    else:
        lam = 1
    batch_size = x.size(0)
    if use_cuda:
        index = torch.randperm(batch_size).cuda()
    else:
        index = torch.randperm(batch_size)
    mixed_x = lam * x + (1 - lam) * x[index, :]
    mixed_y = lam * y + (1 - lam) * y[index]
    return mixed_x, mixed_y, lam
```

⑥进行训练

现在用前面产生的新数据进行训练即可

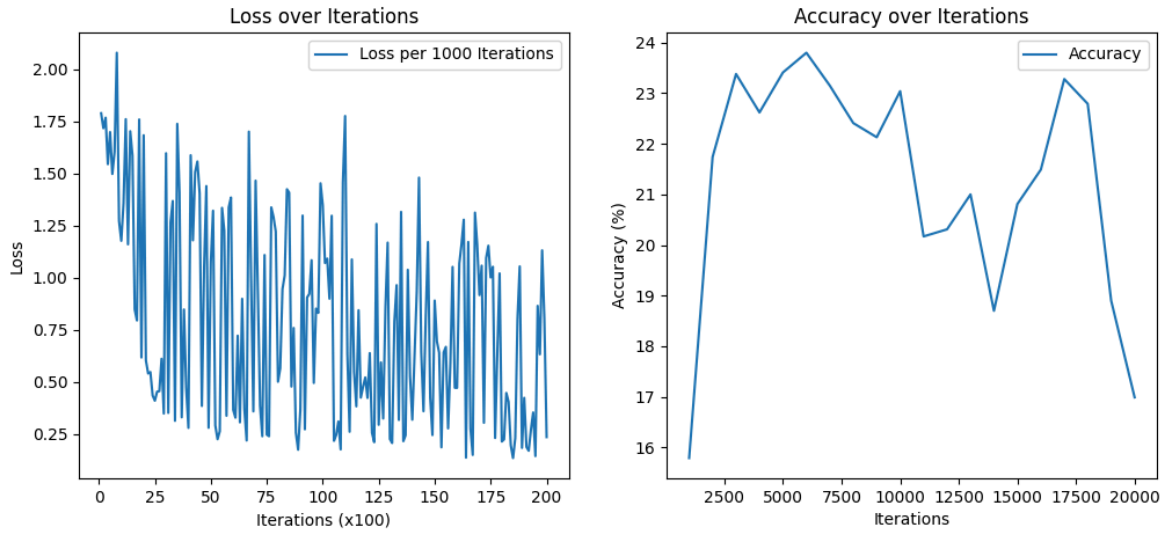
```
def train(model, labeled_loader, unlabeled_loader, test_loader, device, epochs=1):
    optimizer = Adam(model.parameters(), lr=0.00001)
    criterion = nn.CrossEntropyLoss()
    model.to(device)

    iter_loss = []
    iter_accuracy = []
    iteration_count = 0
    loss_per_iteration = []

    for epoch in range(epochs):
```

```
        model.train()
        epoch_loss = 0
        unlabeled_iter = iter(unlabeled_loader)

        for labeled_data, labels in tqdm(labeled_loader):
            try:
                unlabeled_data, _ = next(unlabeled_iter)
            except StopIteration:
                unlabeled_iter = iter(unlabeled_loader)
                unlabeled_data, _ = next(unlabeled_iter)

            # 确保有标签和无标签数据的批次大小相同
            min_batch_size = min(labeled_data.size(0), unlabeled_data.size(0))
            labeled_data = labeled_data[:min_batch_size].to(device)
            labels = labels[:min_batch_size].to(device)
            unlabeled_data = unlabeled_data[:min_batch_size].to(device)

            # 猜测无标签数据的标签
            pseudo_labels = guess_labels(model, unlabeled_data,
num_augmentations=5, device=device)

            # 将labels进行one-hot编码
            one_hot_labels = F.one_hot(labels, num_classes=10).float()

            # 继续使用mixed_data和mixed_labels
            all_data = torch.cat([labeled_data, unlabeled_data], dim=0)
            all_labels = torch.cat([one_hot_labels, pseudo_labels], dim=0)

            mixed_data, mixed_labels, lam = mixup_data(all_data, all_labels,
alpha=0.75, use_cuda=(device == 'cuda'))

            # 前向传播
            logits = model(mixed_data)
            loss = criterion(logits, mixed_labels.argmax(dim=1))

            # 反向传播和优化
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            iteration_count += 1

            # 每100次迭代记录一次损失
            if iteration_count % 100 == 0:
                loss_per_iteration.append(loss.item())

            # 每1000次迭代测试一次正确率
            if iteration_count % 1000 == 0:
                model.eval()
                total = 0
                correct = 0
                with torch.no_grad():
                    for data, labels in test_loader:
                        data, labels = data.to(device), labels.to(device)
                        outputs = model(data)
                        _, predicted = torch.max(outputs.data, 1)
```

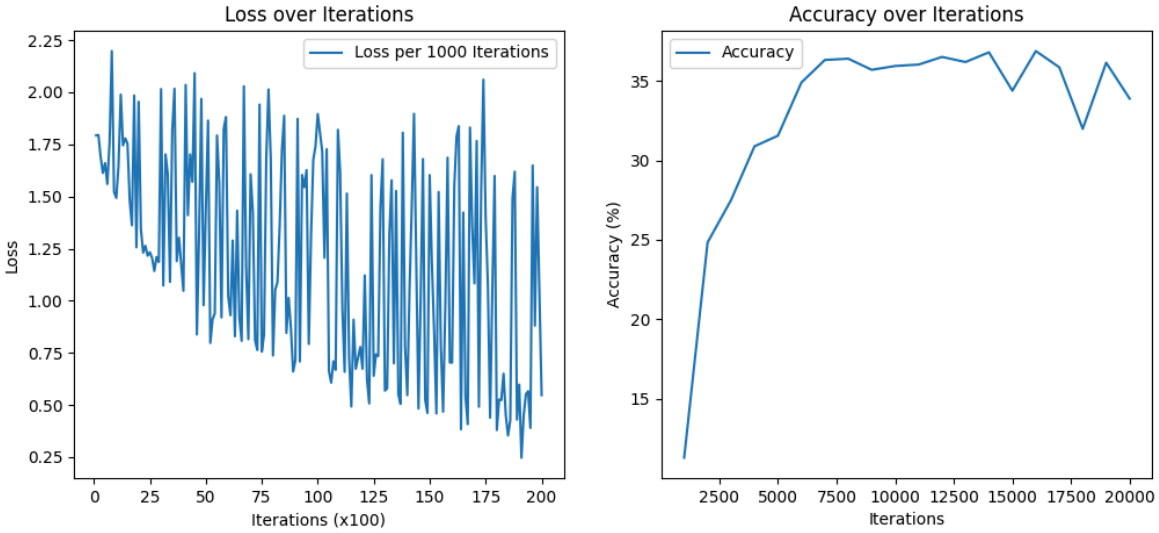
```
                        total += labels.size(0)
                        correct += (predicted == labels).sum().item()

                iteration_accuracy = (correct / total) * 100
                iter_loss.append((iteration_count, epoch_loss / iteration_count))
                iter_accuracy.append((iteration_count, iteration_accuracy))
                print(f'Iteration [{iteration_count}], Loss: {epoch_loss /
iteration_count:.4f}, Accuracy: {iteration_accuracy:.2f}%')
                model.train()

        # 每轮结束后绘制图表
        plt.figure(figsize=(12, 5))

        plt.subplot(1, 2, 1)
        plt.plot(range(1, len(loss_per_iteration) + 1), loss_per_iteration,
label='Loss per 100 Iterations')
        plt.xlabel('Iterations (x100)')
        plt.ylabel('Loss')
        plt.title('Loss over Iterations')
        plt.legend()

        plt.subplot(1, 2, 2)
        if iter_accuracy:
            plt.plot(*zip(*iter_accuracy), label='Accuracy')
        plt.xlabel('Iterations')
        plt.ylabel('Accuracy (%)')
        plt.title('Accuracy over Iterations')
        plt.legend()

        plt.savefig(f'mixmatch-4000.png')
```

⑦结果展示

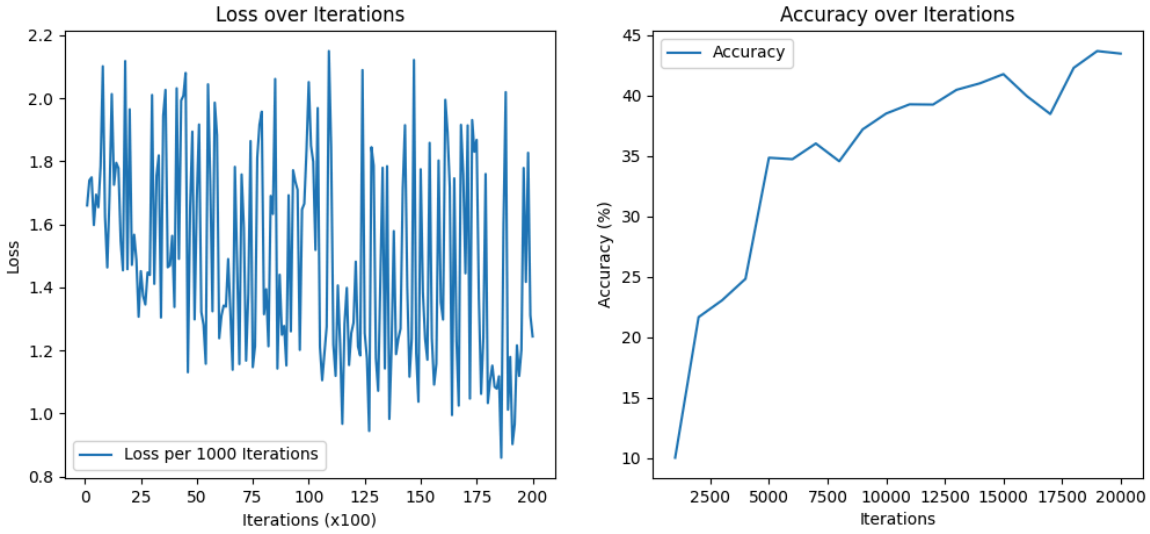
40张标注：



250张标注：



4000张标注：



可以看到，在标注数据只有40张的情况下，我的mixmatch几乎没什么学习能力，正确率一直在20%附近，当标注的数据量上去后，正确率开始有所提升，说明模型开始学习了，在标注数据为4000张时，正确率最高为45%左右，是可以接受的。

二．实现 FixMatch半监督图像分类算法

①数据集加载和数据增强

同样先加载CIFAR-10数据集然后将数据集分为有标签、无标签的训练集和有标签的测试集，有标签的训练集最好保证每一类都有，且数量平均，但是fixmatch和前面mixmatch不同的是其无标签的数据集分为强增强和弱增强的数据集，其中弱增强可能只是对图片进行翻转，而强增强则可以对图片进行颜色取反这种操作。

数据加载：

```
def load_cifar10(batch_size, num_labeled_samples, num_iterations):
    transform_weak = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding=4),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    transform_strong = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomCrop(32, padding=4),
        transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4,
hue=0.1),
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    cifar10_full = CIFAR10(root='./data', train=True, transform=transform_weak,
download=False)
    cifar10_test = CIFAR10(root='./data', train=False, transform=transform_test,
download=False)

    indices = list(range(len(cifar10_full)))
    random.shuffle(indices)

    labeled_indices = indices[:num_labeled_samples]
    unlabeled_indices = indices[num_labeled_samples:]

    unlabeled_indices_weak = []
    unlabeled_indices_strong = []

    for idx in unlabeled_indices:
        # 随机选择每个样本是弱增强还是强增强
        if random.random() < 0.5:
            unlabeled_indices_weak.append(idx)
        else:
            unlabeled_indices_strong.append(idx)

    labeled_dataset = Subset(cifar10_full, labeled_indices)
    unlabeled_dataset_weak = Subset(cifar10_full, unlabeled_indices_weak)
    unlabeled_dataset_strong = Subset(cifar10_full, unlabeled_indices_strong)

    unlabeled_dataset_weak.dataset.transform = transform_weak
    unlabeled_dataset_strong.dataset.transform = transform_strong

    total_labeled_samples = num_iterations * batch_size
    total_unlabeled_samples_weak = num_iterations * batch_size
    total_unlabeled_samples_strong = num_iterations * batch_size

    labeled_sampler = RandomSampler(labeled_dataset, replacement=True,
num_samples=total_labeled_samples)
```

```
    unlabeled_weak_sampler = RandomSampler(unlabeled_dataset_weak,
replacement=True, num_samples=total_unlabeled_samples_weak)
    unlabeled_strong_sampler = RandomSampler(unlabeled_dataset_strong,
replacement=True, num_samples=total_unlabeled_samples_strong)

    labeled_loader = DataLoader(labeled_dataset, batch_size=batch_size,
sampler=labeled_sampler)
    unlabeled_weak_loader = DataLoader(unlabeled_dataset_weak,
batch_size=batch_size, sampler=unlabeled_weak_sampler)
    unlabeled_strong_loader = DataLoader(unlabeled_dataset_strong,
batch_size=batch_size, sampler=unlabeled_strong_sampler)

    test_loader = DataLoader(cifar10_test, batch_size=64, shuffle=False)

    return labeled_loader, unlabeled_weak_loader, unlabeled_strong_loader,
test_loader
```

数据增强:

```
# 数据增强
transform_weak = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
])

transform_strong = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),
    transforms.ToTensor(),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
])
```

其中弱增强包括：

- `transforms.RandomHorizontalFlip()`：随机水平翻转图像。这有助于模型学习到图像中左右对称的特征。
- `transforms.RandomCrop(32, padding=4)`：随机裁剪图像到32x32像素，并在裁剪之前对图像进行4像素的填充。这有助于模型适应不同的裁剪位置和尺寸，增强其鲁棒性。
- `transforms.ToTensor()`：将PIL图像或numpy数组转换为PyTorch张量，并将像素值从[0, 255]范围缩放到[0, 1]范围。

强增强包括：

- `transforms.RandomHorizontalFlip()`：随机水平翻转图像。
- `transforms.RandomCrop(32, padding=4)`：随机裁剪图像到32x32像素，并在裁剪之前对图像进行4像素的填充。
- `transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1)`：随机调整图像的亮度、对比度、饱和度和色调。
- 亮度（**brightness**）：在[1-0.4, 1+0.4]范围内随机改变图像的亮度。
- 对比度（**contrast**）：在[1-0.4, 1+0.4]范围内随机改变图像的对比度。
- 饱和度（**saturation**）：在[1-0.4, 1+0.4]范围内随机改变图像的饱和度。
- 色调（**hue**）：在[-0.1, 0.1]范围内随机改变图像的色调。
- `transforms.ToTensor()`：将PIL图像或numpy数组转换为PyTorch张量，并将像素值从[0, 255]范围缩放到[0, 1]范围。

测试集增强包括：

- `transforms.ToTensor()`：将PIL图像或numpy数组转换为PyTorch张量，并将像素值从[0, 255]范围缩放到[0, 1]范围。

②使用数据进行训练

有标注的数据，执行有监督训练，和普通分类任务训练没有区别。没有标注的数据，首先将其经过弱增强之后送到模型中推理获取伪标签，然后再将其经过强增强送到模型中进行预测。

```
        x_l, y_l = x_l.to(device), y_l.to(device)
        x_ul_weak, x_ul_strong = x_ul_weak.to(device), x_ul_strong.to(device)
```

并利用生成的伪标签监督模型对强增强数据的预测。模型对弱增强数据的预测，只有大于一定阈值条件时才执行伪标签的生成，并使用该伪标签来进行无标注图像的训练。

```
        # 监督损失
        outputs_l = model(x_l)
        loss_l = criterion(outputs_l, y_l)

        # 生成伪标签
        with torch.no_grad():
            outputs_ul_weak = model(x_ul_weak)
            pseudo_labels = torch.softmax(outputs_ul_weak, dim=1)
            max_probs, targets_ul = torch.max(pseudo_labels, dim=1)
            mask = max_probs.ge(threshold).float()
```

③进行训练

按前面的流程对数据进行训练即可：

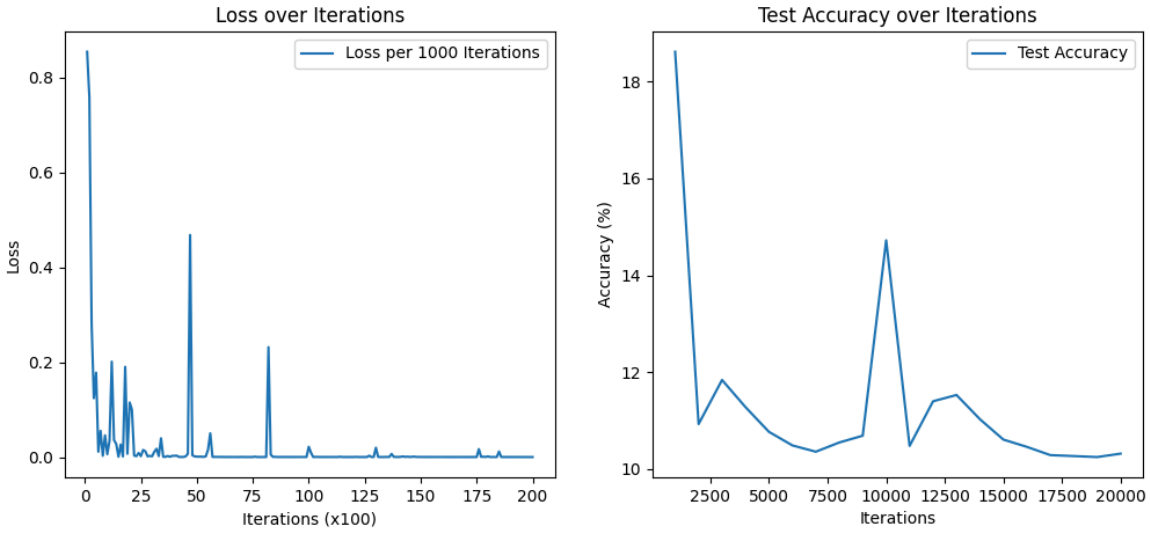
```
def train(model, labeled_loader, unlabeled_weak_loader, unlabeled_strong_loader,
test_loader, device, epochs=1):
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    model.to(device)

    iteration_count = 0
    loss_per_iteration = []
    iter_accuracy = []

    best_accuracy = 0
    best_epoch = 0
    lambda_u = 1
    threshold = 0.95

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        labeled_iter = iter(labeled_loader)
        unlabeled_weak_iter = iter(unlabeled_weak_loader)
        unlabeled_strong_iter = iter(unlabeled_strong_loader)

        for _ in tqdm(range(len(labeled_loader))):
            try:
                labeled_data = next(labeled_iter)
                unlabeled_weak_data = next(unlabeled_weak_iter)
                unlabeled_strong_data = next(unlabeled_strong_iter)
            except StopIteration:
                break

            x_l, y_l = labeled_data
            x_ul_weak, _ = unlabeled_weak_data
            x_ul_strong, _ = unlabeled_strong_data

            x_l, y_l = x_l.to(device), y_l.to(device)
            x_ul_weak, x_ul_strong = x_ul_weak.to(device), x_ul_strong.to(device)

            # 监督损失
            outputs_l = model(x_l)
            loss_l = criterion(outputs_l, y_l)

            # 生成伪标签
            with torch.no_grad():
                outputs_ul_weak = model(x_ul_weak)
                pseudo_labels = torch.softmax(outputs_ul_weak, dim=1)
                max_probs, targets_ul = torch.max(pseudo_labels, dim=1)
                mask = max_probs.ge(threshold).float()

            # 一致性损失
            outputs_ul_strong = model(x_ul_strong)
            loss_u = (criterion(outputs_ul_strong, targets_ul) * mask).mean()

            train_loss = loss_l + lambda_u * loss_u
```

```
            optimizer.zero_grad()
            train_loss.backward()
            optimizer.step()

            total_loss += train_loss.item()
            iteration_count += 1

            # 每100次迭代记录一次损失
            if iteration_count % 100 == 0:
                loss_per_iteration.append(train_loss.item())

            # 每1000次迭代测试一次
            if iteration_count % 1000 == 0:
                model.eval()
                correct = 0
                total = 0
                test_loss = 0
                with torch.no_grad():
                    for x, y in test_loader:
                        x, y = x.to(device), y.to(device)
                        outputs = model(x)
                        loss = criterion(outputs, y)
                        test_loss += loss.item()
                        _, predicted = torch.max(outputs, 1)
                        total += y.size(0)
                        correct += (predicted == y).sum().item()

                test_accuracy = 100 * correct / total
                iter_accuracy.append((iteration_count, test_accuracy))

                print(f'Iteration [{iteration_count}], Loss: {total_loss /
len(labeled_loader):.4f}, Accuracy: {test_accuracy:.2f}%')
                model.train()

        # 每轮结束后绘制图表
        plt.figure(figsize=(12, 5))
        # 绘制训练损失曲线
        plt.subplot(1, 2, 1)
        plt.plot(range(1, len(loss_per_iteration) + 1), loss_per_iteration,
label='Loss per 100 Iterations')
        plt.xlabel('Iterations (x100)')
        plt.ylabel('Loss')
        plt.title('Loss over Iterations')
        plt.legend()
        # 绘制测试准确度曲线
        plt.subplot(1, 2, 2)
        if iter_accuracy:
            plt.plot(*zip(*iter_accuracy), label='Test Accuracy')
        plt.xlabel('Iterations')
        plt.ylabel('Accuracy (%)')
        plt.title('Test Accuracy over Iterations')
        plt.legend()

        plt.show()
        plt.savefig(f'fixmatch-4000.png')
```

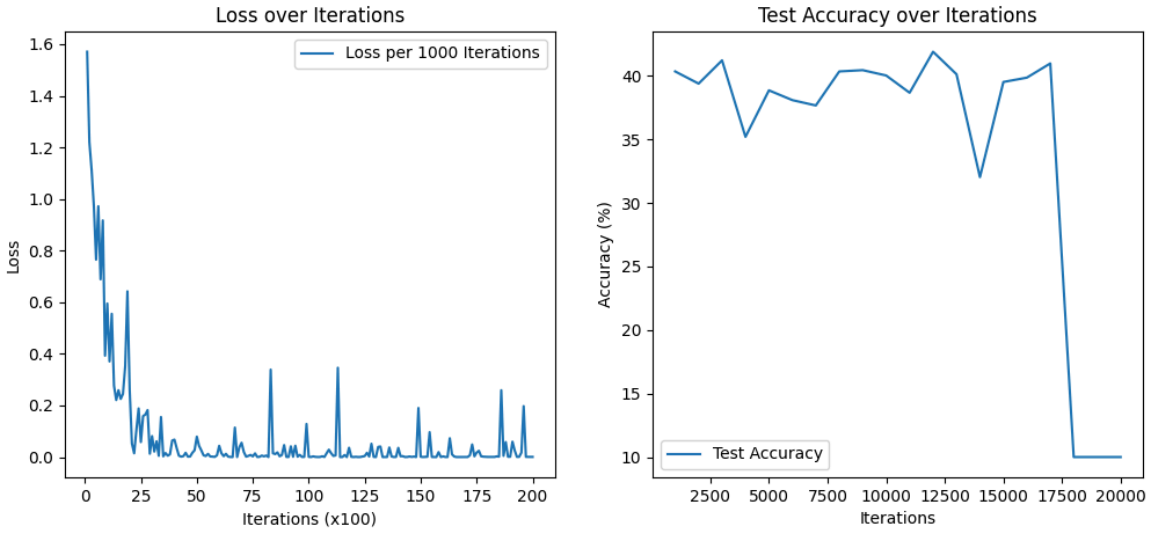
```
    print(f"Best Iteration {best_epoch * len(labeled_loader)}, Best Test Accuracy:
{best_accuracy:.2f}%")
```

④结果展示

40张标注：

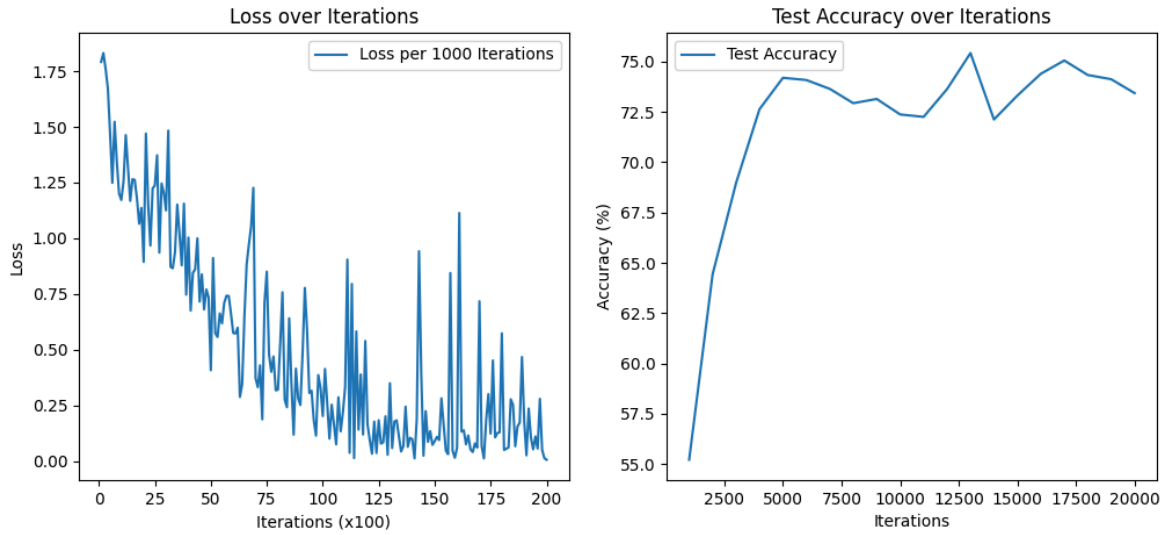


250张标注：



40张和250张的标注最后正确率都会跌到10%左右了，可能是模型对数据出现了过拟合，说明鲁棒性有待加强，在这种情况下学习率应该可以调低一点。

4000张标注：



可以看到，在标注数据量相同的情况下，fixmatch的学习效果是比mixmatch强的，在标注量为4000张时，正确率可以到达75%，这是很高的正确率了。

我觉得原因如下：

1. 更简单的伪标签生成策略：
    - FixMatch使用一种简单而有效的伪标签生成策略。它直接从未标注数据中生成伪标签，仅在模型预测具有高置信度时使用这些伪标签。这种策略减少了伪标签的噪声，从而提高了训练的稳定性和性能。
    - MixMatch则使用多种数据增强方法和均值集成来生成伪标签，这种复杂性可能引入更多的噪声，从而影响训练效果。
2. 一致性正则化：
    - FixMatch强烈依赖一致性正则化，通过对未标注数据进行强增强和弱增强，确保模型对同一数据的不同增强版本具有相似的输出。这种一致性约束有助于模型在未标注数据上学习更好的特征。
    - MixMatch也使用一致性正则化，但其数据增强和混合策略较为复杂，可能导致一致性正则化的效果不如FixMatch直接和有效。
3. 超参数设置和训练稳定性：
    - FixMatch的超参数设置相对简单，主要关注置信度阈值和增强策略。这使得模型训练更加稳定和易于调优。
    - MixMatch涉及更多的超参数，如混合比例、增强策略等，这可能增加调优的难度，并影响最终的模型性能。
4. 数据增强策略：

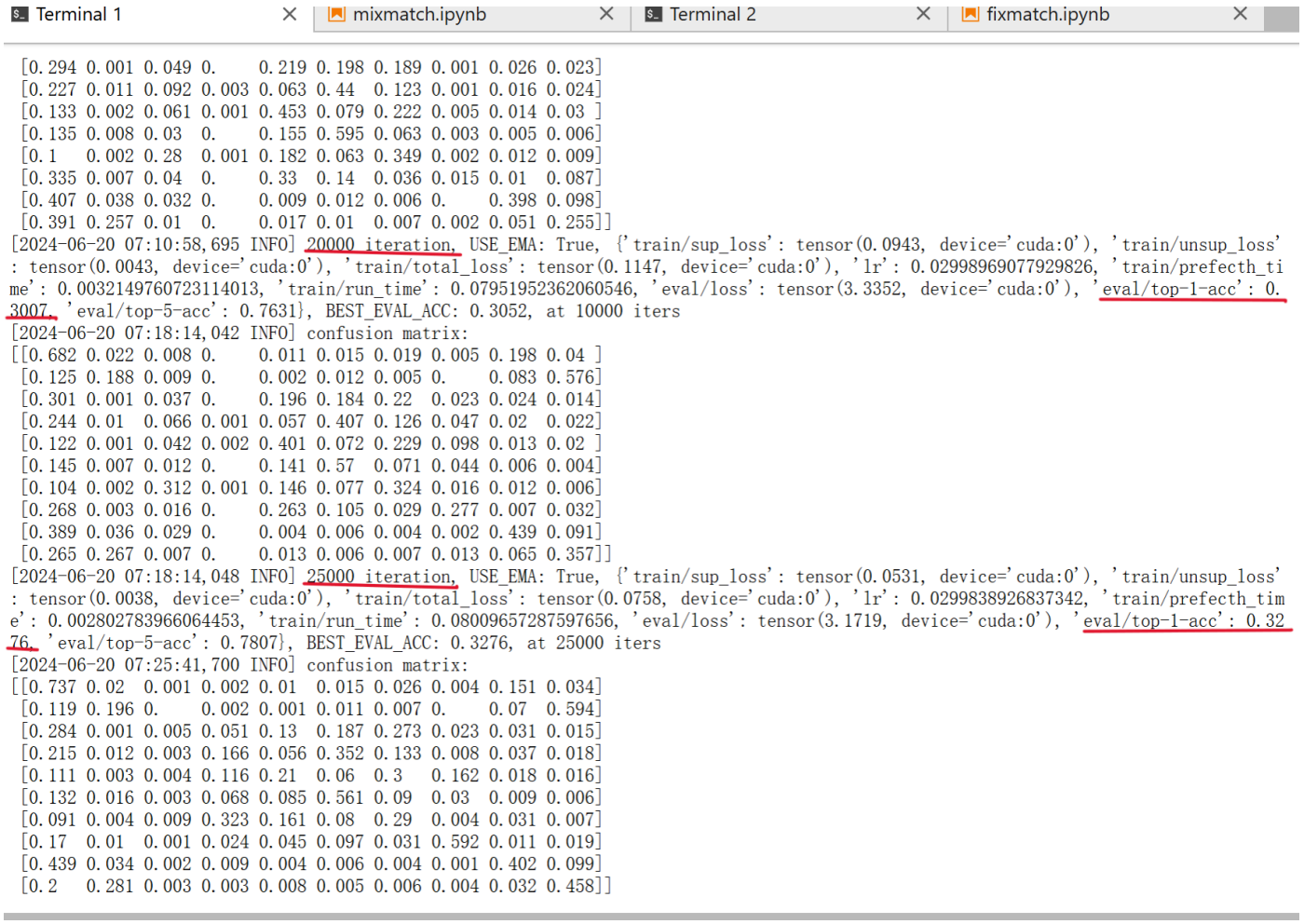
- FixMatch采用了更为严格的数据增强策略（如RandAugment），这些策略在保持数据多样性的同时，能够生成更具挑战性的样本，从而促进模型的泛化能力。
- MixMatch的增强策略虽然多样，但可能不如FixMatch的增强策略在生成具有代表性的难样本方面有效。

综上所述，FixMatch在标注数据量相同的情况下，可能由于其简单有效的伪标签生成策略、强一致性正则化、较少的超参数调优以及严格的数据增强策略，表现出比MixMatch更好的学习效果。

三.TorchSSL中的mixmatch和fixmatch

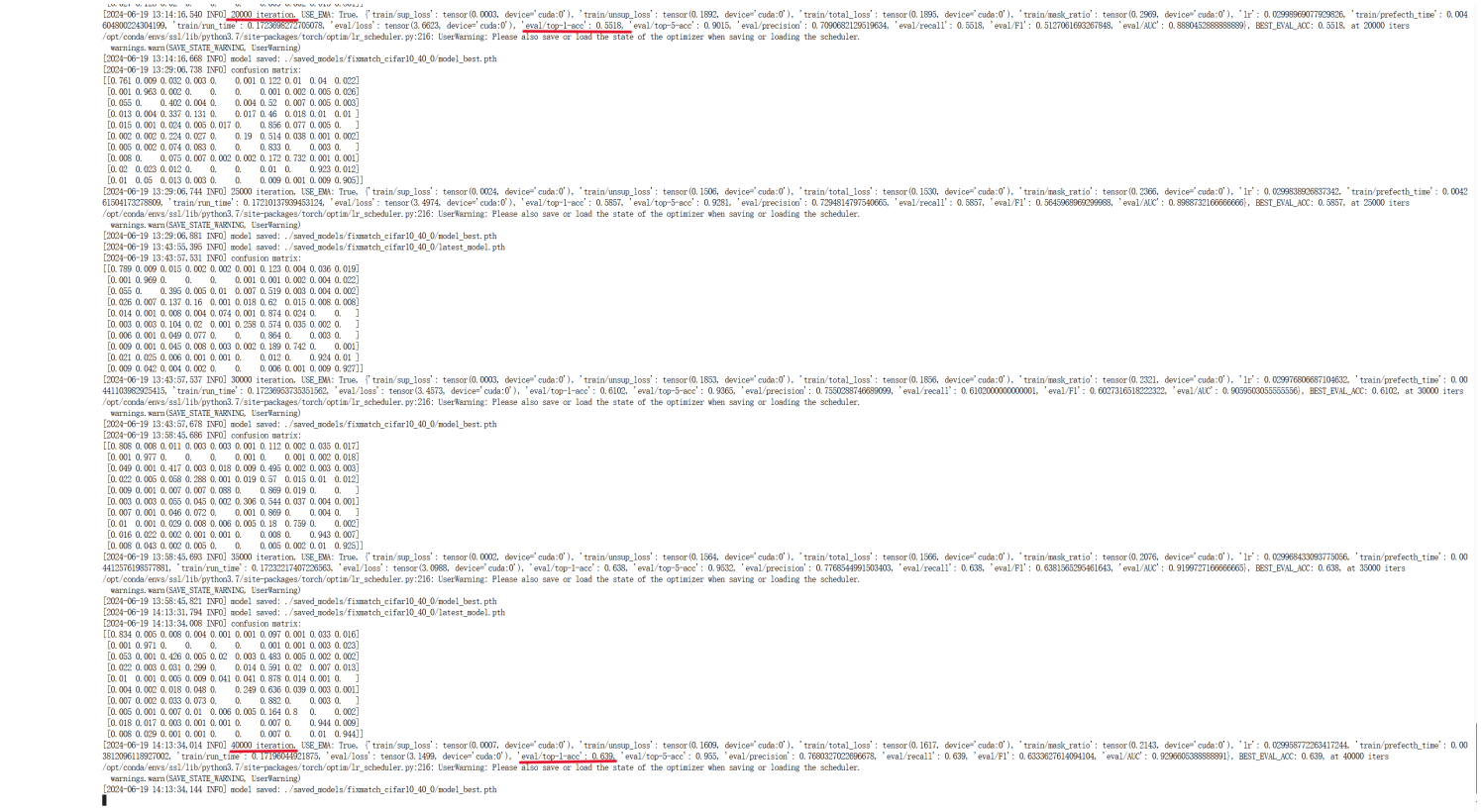
先去TorchSSL的开源地址下载TorchSSL的代码，然后按要求设备并运行程序，由于其需要跑的时间有点长，故我只分别跑了mixmatch和fixmatch的在有40张标注情况下的代码，最后结果如下图所示

mixmatch：



可以看到，在迭代次数达20000次后，mixmatch在只有40张标注的情况下，也可以达到30%的正确率，在25000次迭代下，正确率有32.76%。

fixmatch：



fixmatch的性能比mixmatch要好，在只有40张标注的情况下，20000次迭代正确率就可以到55%，在迭代次数够大的情况下（40000次）正确率达到了63.9%。

实验收获

通过本次实验，我深入理解了半监督学习的原理和应用，特别是通过实现和对比 MixMatch 和 FixMatch 算法，加深了对数据增强、伪标签生成及一致性正则化等关键技术的认识。通过详细分析和实验结果的比较，我发现 FixMatch 在相同标注数据量下表现优于 MixMatch，验证了其更有效的伪标签生成策略和增强方法。实验过程中，我不仅掌握了如何在 PyTorch 框架下进行复杂模型的实现和调试，还学会了如何处理和增强数据集、设计实验以及评估模型性能。这次实验大大提升了我在深度学习领域的实践能力和理论知识，为后续的研究和应用打下了坚实的基础。