模式识别 FixMatch

一、实验内容

基于FixMatch的CIFAR-10数据集半监督图像分类

1. 阅读原始论文和相关参考资料，基于Pytorch动手实现FixMatch半监督图像分类算法，在CIFAR-10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结果
2. 按照原始论文的设置，FixMatch使用WideResNet-28-2作为Backbone网络，即深度为28，扩展因子为2，使用CIFAR-10作为数据集，可以参考现有代码的实现，算法核心步骤不能直接照抄！
3. 使用TorchSSL中提供的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
4. 提交源代码，并提交实验报告，描述实现过程中的主要算法部分，可以尝试分析对比FixMatch和其他半监督算法的不同点，例如MixMatch等。

二、实验过程

2.1 手动实现FixMatch

2.1.1 FixMatch算法原理

FixMatch是一种半监督学习算法，它通过结合少量有标签数据和大量无标签数据来提升模型的性能。其基本思想是通过一致性正则化（Consistency Regularization）和伪标签（Pseudo-Labeling）来利用无标签数据。具体来说，FixMatch的核心思想如下：

1. **一致性正则化**：假设模型对相同输入数据的不同扰动（如数据增强）应输出相同的结果。通过强、弱数据增强产生两种输入，模型应对两种增强后的数据输出一致的结果。
2. **伪标签**：利用模型对无标签数据的预测结果作为伪标签，且只在预测置信度高于设定阈值时使用这些伪标签来训练模型。

根据FixMatch的核心思想，可以构建FixMatch框架，其核心步骤在于如何训练模型以进行分类预测，我实现的思路如下：

1. 数据准备和预处理

   从数据集中加载有标签训练数据和无标签训练数据，其中无标签训练数据需要分别进行弱增强和强增强处理。

   - 弱增强：随机水平翻转->随机裁剪->标准化
   - 强增强：随机水平翻转->随机裁剪->颜色抖动->标准化

2. 模型选取和初始化

   - FixMatch的骨干网络使用WideResNet，深度为28，扩展因子为2，因为其在图像分类任务中性能优越，且网络结构较宽，能更好捕捉到图像的特征，处理无标签数据的效果更好，更适用于半监督学习。
   - 我的模型中还引入了EMA机制，用于更新模型参数的滑动平均值，以此提升模型的泛化能力和稳定性。
   - 同时在学习率的调整中，采用余弦退火学习率调度器逐渐降低学习率，提高模型的收敛效果。

3. 模型训练

   - 将批次数据（有标签数据和增强后的无标签数据）输入模型，获得预测结果。
   - 分别计算有标签数据损失L_x（交叉熵损失）和无标签数据损失L_u（强增强无标签数据的伪标签损失。伪标签由弱增强无标签数据的预测结果生成，且只在预测置信度高于阈值时参与损失计算），得到两者之和作为训练总损失。
   - 反向传播并更新模型参数，更新学习率调度器，更新EMA（指数移动平均）模型参数。

4. 模型测试

在每个训练周期结束后，使用测试数据集评估模型性能，计算Top-1准确率，并保存最佳模型。

2.1.2 关键代码

1. 数据准备和预处理

使用 `get_cifar10()`获取处理后的数据，包括有标签数据、无标签数据和测试数据，其中调用 `TransformFixMatch`类进行数据增强。

```
# myFM.py
labeled_dataset, unlabeled_dataset, test_dataset =
get_cifar10('./data',NUM_LABELED,NUM_CLASSES,BATCH_SIZE,EVAL_STEP)
```

```
# cifar.py
# 获取CIFAR-10数据集
def get_cifar10(root, num_labeled, num_classes, batch_size, eval_step):
    # 有标签数据的转换
    transform_labeled = transforms.Compose([
        transforms.RandomHorizontalFlip(),  # 随机水平翻转

transforms.RandomCrop(size=32,padding=int(32*0.125),padding_mode='reflect'),
# 随机裁剪
        transforms.ToTensor(),  # 转换为张量
        transforms.Normalize(mean=cifar10_mean, std=cifar10_std)    # 标准化
    ])
    # 测试数据的转换
    transform_val = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=cifar10_mean, std=cifar10_std)
    ])
    # 下载CIFAR-10基础数据集
    base_dataset = datasets.CIFAR10(root, train=True, download=True)
    # 分割有标签和无标签的数据索引
    train_labeled_idxs, train_unlabeled_idxs = x_u_split(num_labeled,
num_classes, batch_size, eval_step, base_dataset.targets)
    # 定义有标签和无标签的数据集、测试数据集
    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

# 定义TransformFixMatch类，包含弱增强和强增强
class TransformFixMatch(object):
    def __init__(self, mean, std):
        self.weak = transforms.Compose([
```

```
            transforms.RandomHorizontalFlip(),  # 随机水平翻转

transforms.RandomCrop(size=32,padding=int(32*0.125),padding_mode='reflect')])
# 随机裁剪
        self.strong = transforms.Compose([
            transforms.RandomHorizontalFlip(),  # 随机水平翻转

transforms.RandomCrop(size=32,padding=int(32*0.125),padding_mode='reflect'),
# 随机裁剪
            transforms.ColorJitter(brightness=0.5, contrast=0.5,
saturation=0.5, hue=0.5),])    # 颜色抖动
        self.normalize = transforms.Compose([
            transforms.ToTensor(),  # 转换为张量
            transforms.Normalize(mean=mean, std=std)])  # 标准化

    def __call__(self, x):
        weak = self.weak(x)
        strong = self.strong(x)
        return self.normalize(weak), self.normalize(strong) # 返回增强后的结果
```

2. WideResNet骨干网络

参考现成的WideResNet代码模块，稍作修改使其能在 `myFM.py`的模型初始化中调用，具体见 `/models/WideResNet.py`。

3. FixMatch模型训练

按照上述我的算法思路进行代码编写，训练过程包括模型预测、损失计算、模型测试等。

```
# 模型训练
def train(self, labeled_trainloader, unlabeled_trainloader, test_loader):
    global best_acc
    test_accs = []  # 存储测试准确率
    labeled_iter = iter(labeled_trainloader)
    unlabeled_iter = iter(unlabeled_trainloader)
    self.model.train()
    for epoch in range(self.epochs):
        for batch_idx in range(EVAL_STEP):
            try:
                inputs_x, targets_x = labeled_iter.next()
            except:
                labeled_iter = iter(labeled_trainloader)
                inputs_x, targets_x = labeled_iter.next()
            try:
                (inputs_u_w, inputs_u_s), _ = unlabeled_iter.next()
            except:
                unlabeled_iter = iter(unlabeled_trainloader)
                (inputs_u_w, inputs_u_s), _ = unlabeled_iter.next()
            batch_size = inputs_x.shape[0]
            inputs = interleave(torch.cat((inputs_x, inputs_u_w, inputs_u_s)),
2*MU+1).cuda()    # 将有标签和无标签数据合并并交错
```

```
            targets_x = targets_x.cuda()
            logits = self.model(inputs) # 模型预测
            logits = de_interleave(logits, 2*MU+1)  # 反交错
            logits_x = logits[:batch_size]
            logits_u_w, logits_u_s = logits[batch_size:].chunk(2)   # 无标签数据的弱增强和强增强预测结果
            del logits

            Lx = F.cross_entropy(logits_x, targets_x, reduction='mean') # 有标签数据的交叉熵损失
            pseudo_label = torch.softmax(logits_u_w.detach(), dim=-1)   # 计算无标签数据的伪标签
            max_probs, targets_u = torch.max(pseudo_label, dim=-1)  # 获得伪标签和其最大概率
            mask = max_probs.ge(self.threshold).float() # 置信度大于阈值的伪标签
            Lu = (F.cross_entropy(logits_u_s, targets_u,reduction='none') * mask).mean()    # 无标签数据的损失，仅对高置信度样本计算
            loss = Lx + Lu  # 总损失

            loss.backward() # 反向传播
            self.optimizer.step()   # 更新优化器参数
            self.scheduler.step()   # 更新学习率
            self.ema_model.update(self.model)   # 更新EMA模型
            self.model.zero_grad()  # 清零梯度
            print(f'Epoch {epoch+1}/{self.epochs}, Iter {batch_idx+1}/{EVAL_STEP}, Loss: {loss.item()}')

            test_model = self.ema_model.ema
            test_acc = self.test(test_loader, test_model)   # 测试准确率
            is_best = test_acc > best_acc   # 判断当前测试准确率是否为最佳
            best_acc = max(test_acc, best_acc)  # 更新最佳准确率

            model_to_save = self.model.module if hasattr(self.model, "module") else self.model  # 获取需要保存的模型
            ema_to_save = self.ema_model.ema.module if hasattr(self.ema_model.ema, "module") else self.ema_model.ema    # 获取需要保存的EMA模型
            save_checkpoint({
                'epoch': epoch + 1,
                'state_dict': model_to_save.state_dict(),
                'ema_state_dict': ema_to_save.state_dict(),
                'acc': test_acc,
                'best_acc': best_acc,
                'optimizer': self.optimizer.state_dict(),
                'scheduler': self.scheduler.state_dict(),
            }, is_best, 'result')   # 调用save_checkpoint函数保存模型
            test_accs.append(test_acc)  # 将测试准确率添加到列表中
```

4. 模型测试

根据当前模型在测试集中的输出结果，与正确标签进行比较，输出准确率。

```
# 模型测试
def test(self, test_loader, model):
```

```
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = model(images.cuda())  # 模型预测
            _, predicted = torch.max(outputs.data, 1)   # 获取预测结果
            total += labels.size(0) # 总样本数
            correct += (predicted == labels.cuda()).sum().item()    # 计算正确
预测的样本数
    test_acc = correct / total  # 计算测试准确率
    print('Accuracy: %d %%' % (100 * test_acc))
    return test_acc
```

2.2 使用TorchSSL实现FixMatch

2.2.1 TorchSSL的特点

1. 高效的数据加载和处理

   TorchSSL通过使用PyTorch的DataLoader类和多进程数据加载方式，提高了数据加载的效率。特别是对于大型数据集或复杂的数据增强操作，能够显著减少数据准备时间。

2. 多种数据增强策略

   除了FixMatch中使用的弱增强和强增强策略外，TorchSSL还提供了其他数据增强方法，例如随机裁剪、水平翻转、颜色抖动等。用户可以根据需要选择和组合不同的数据增强策略，进一步提升模型的泛化能力。

3. 指数滑动平均（EMA）

   除了常规的模型参数更新，TorchSSL还实现了EMA机制，通过滑动平均来更新模型参数。这种处理方式能够提升模型的稳定性和泛化能力，特别是在小数据集或标签稀缺的情况下。

4. 自动混合精度训练（AMP）

   TorchSSL支持自动混合精度训练（AMP），通过在训练过程中动态选择合适的精度（例如半精度float16和单精度float32），加速模型训练的同时减少显存使用。这对于大型模型和高分辨率图像非常有用。

5. 丰富的日志记录和监控

TorchSSL提供了详细的日志记录和监控功能，包括训练和测试过程中的损失、准确率、学习率等关键指标。用户可以通过这些日志实时监控模型训练的进展，并根据需要调整训练参数。

2.2.2 TorchSSL-FixMatch训练测试

从https://github.com/StephenStorm/TorchSSL下载TorchSSL源代码。

- cifar10-40

```
python fixmatch.py --c config/fixmatch/fixmatch_cifar10_40_0.yaml
```

- cifar10-250

```
python fixmatch.py --c config/fixmatch/fixmatch_cifar10_250_0.yaml
```

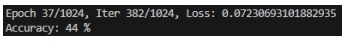
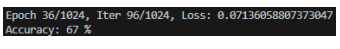
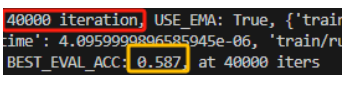
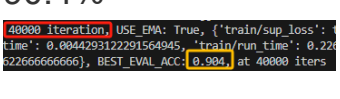
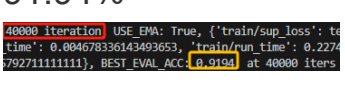
- cifar10-4000

```
python fixmatch.py --c config/fixmatch/fixmatch_cifar10_4000_0.yaml
```

三、实验结果

3.1 对比手动实现和TorchSSL实现的效果

在cifar_10数据集上分别应用手动实现和TorchSSL实现的FitMatch算法，进行训练测试，batch_size均设为64。由于训练时间过长，现只截取40000个iters，进行模型效果对比如下：

实现方式\有标注数量	40	250	4000
手动实现（epoch 40）	44% 	67% 	86% 
TorchSSL（epoch 8）	58.7% 	90.4% 	91.94% 

实现方式\有标注数量	40	250	4000
TorchSSL（最终）	91.21% BEST_EVAL_ACC: 0.9121, at 270000 iters	93.74% BEST_EVAL_ACC: 0.9374, at 270000 iters	94.49% BEST_EVAL_ACC: 0.9449, at 275000 iters

分析：

1. 从训练速度来看，在同等显卡环境下，TorchSSL的训练速度远快于手动实现（并没有展示），推测可能是因为TorchSSL使用了AMP加速训练，也可能因为其模块化和不断优化后性能已经达到了最优状态，这是普通的手动实现无法比拟的。
2. 从FixMatch分类效果来看，TorchSSL实现的FitMatch在测试集上的分类准确率也是远远高于手动实现，推测可能是TorchSSL的优化器参数选择了一组最优的参数，而且数据增强策略也更多元化，使得模型泛化能力更强，等等。
3. 有标签数据量越多，在短时间内，TorchSSL实现的FitMatch分类准确率更接近官方实验结果预期。
4. 总的来说，我的手动实现FitMatch模型是适用于半监督图像分类任务的，只是模型性能有待优化。

3.2 分析对比FixMatch和其他半监督算法的不同点

通过网上的学习和理解，对比FixMatch和其他半监督算法的核心思想、关键步骤和优缺点总结如下：

算法	核心思想	主要步骤	优点	缺点
FixMatch	结合伪标签和一致性正则化	弱增强生成伪标签，强增强应用一致性损失	简单高效，性能优异	伪标签质量依赖于模型
Mean Teacher	教师模型和学生模型，教师参数为EMA	学生模型训练，教师模型提供一致性约束	教师模型稳定，减少噪声	计算开销较大
Pseudo-Labeling	使用模型预测生成伪标签	预测无标签数据，选择概率最高的类作为伪标签	实现简单，易于理解	容易引入错误标签
UDA	一致性训练和无监督数据增强	强增强应用一致性损失	利用数据增强，提升鲁棒性	数据增强策略复杂，参数调节困难

算法	核心思想	主要步骤	优点	缺点
MixMatch	综合多种技术	多次增强，伪标签平均，混合标签	性能优异，充分利用无标签数据	实现复杂，计算开销较大