模式识别第三次作业：半监督图像分类

21307174 刘俊杰

June 2024

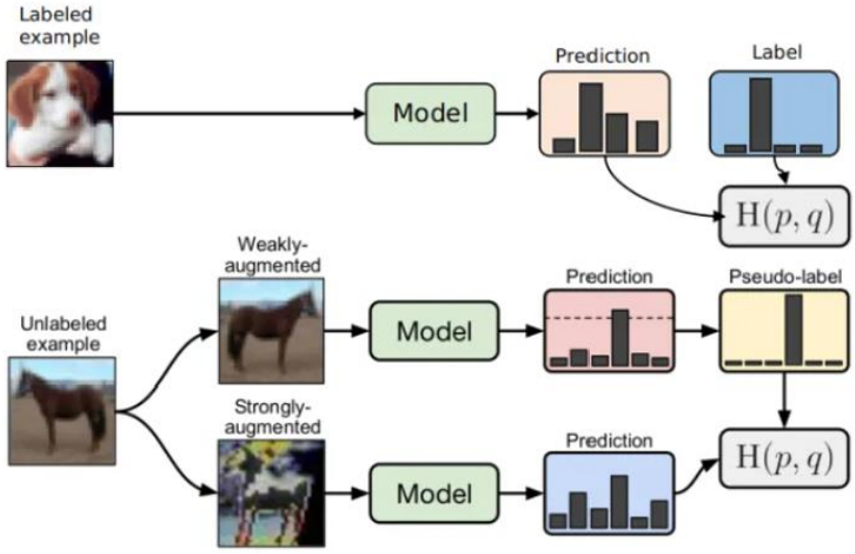
1 实验目的

关于半监督学习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。关于深度学习中半监督学习更全面的总结，可以参考深度学习半监督学习综述 [1]

2 实验内容

基于 FixMatch 的 CIFAR-10 数据集半监督图像分类

FixMatch 结合了伪标签(Pseudo Label)和一致性正则化 (Consistency Regularization) 来实现对无标注数据的高效利用，训练过程包括两个部分，有监督训练和无监督训练。有 label 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 label 的数据，经过首先经过弱增强获取伪标签。然后利用该伪标签去监督强增强的输出值，只有大于一定阈值条件才执行伪标签的生成，并使用伪标签来进行无标注图像的训练。



3 实验要求

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。

4 实验设计

4.1 基于 FixMatch 的半监督图像分类原理

FixMatch 是一种半监督学习算法，旨在通过有效利用标记数据和未标记数据来提高深度学习模型的性能。它结合了一致性正则化（Consistency Regularization）和伪标签（Pseudo-Labeling）技术，这些都是半监督学习中常见且有效的方法。

一致性正则化

一致性正则化的核心思想是：模型对于一个输入及其增强版本应该产生一致的预测。如果模型对某个输入的预测非常自信，那么对于这个输入的强增强版本，它也应该做出相同的预测。

设有一个未标记数据样本 u，对其应用两种类型的数据增强：弱增强和强增强。记 u^w 和 u^s 分别为弱增强和强增强后的样本。我们希望模型对于 u^w 和 u^s 的预测是一致的。

伪标签

伪标签方法通过模型的预测为未标记数据生成标签。具体来说，对于一个未标记数据样本，模型首先进行预测，如果预测结果的置信度高于设定的阈值，则将该预测结果作为伪标签。

对于未标记数据样本 u，模型生成预测分布 $p_m(y|u^w)$，选择置信度最高的类作为伪标签 $\hat{y}$。如果 $\max(p_m(y|u^w)) \geq \tau$，则认为该伪标签是可靠的。

FixMatch 算法工作流程

FixMatch 的工作流程可以分为以下几个步骤：

1. **数据增强**：对于每个未标记数据样本 u，应用弱增强和强增强，得到 u^w 和 u^s。
2. **模型预测**：使用模型对弱增强后的未标记样本 u^w 进行预测，生成预测分布 $p_m(y|u^w)$。选择置信度最高的类作为伪标签 $\hat{y}$。如果预测的最大置信度高于阈值 τ，则将此伪标签视为可靠。

3. **损失计算**：对于标记数据，使用标准的交叉熵损失 $\mathcal{L}_s$ 来计算损失。对于未标记数据，如果伪标签被认为是可靠的，则计算强增强样本的交叉熵损失 $\mathcal{L}_u$，以此作为一致性正则化的一部分。

4. **损失总和**：总损失 $\mathcal{L}$ 是标记数据的损失 $\mathcal{L}_s$ 和未标记数据的一致性损失 $\mathcal{L}_u$ 的加权和：

$$\mathcal{L} = \mathcal{L}_s + \lambda \mathcal{L}_u$$

数据增强

数据增强是指对训练数据进行一系列随机变换以生成新的训练样本。增强方法包括旋转、翻转、裁剪、颜色变换等。弱增强使用简单的变换，而强增强使用更复杂和激进的变换（如 RandAugment）。

模型预测和伪标签生成

对于每个弱增强样本，模型生成预测分布。如果预测的置信度高于设定的阈值，则认为该预测可靠，并将其作为伪标签。这个过程有效地利用了未标记数据。

损失函数

FixMatch 使用两种损失函数：有监督的交叉熵损失和无监督的一致性损失。对于标记数据，使用交叉熵损失来最小化真实标签和预测标签之间的差异。对于未标记数据，如果伪标签可靠，则使用交叉熵损失来最小化伪标签和强增强样本预测之间的差异。

损失加权

总损失是有监督损失和无监督损失的加权和，其中 λ 是一个超参数，用于控制两者的平衡。通过调整 λ，可以控制模型对未标记数据的依赖程度。

4.2 网络模型设计

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

WideResNet 实现:

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    def __init__(self, in_planes, out_planes, stride, dropRate=0.0):
        super(BasicBlock, self).__init__()
        self.bn1 = nn.BatchNorm2d(in_planes)        # Batch Normalization
        self.relu1 = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_planes)       # Batch Normalization
        self.relu2 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_planes, out_planes, kernel_size=3, stride=1,
                               padding=1, bias=False)
        self.droprate = dropRate                    # Dropout
        self.equalInOut = (in_planes == out_planes)
        # 如果输入输出通道数不同，使用1x1卷积匹配维度
        self.convShortcut = (not self.equalInOut)
        and nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,
                               padding=0, bias=False) or None

    def forward(self, x):
        if not self.equalInOut:
            x = self.relu1(self.bn1(x))             # 如果输入输出通道数不同，先做BN和ReLU
        else:
            out = self.relu1(self.bn1(x))           # 如果输入输出通道数相同，直接做BN和ReLU
        out = self.relu2(self.bn2(self.conv1(out if self.equalInOut else x)))
# 第一个卷积层
        if self.droprate > 0:
            out = F.dropout(out, p=self.droprate, training=self.training)
```

```
    # Dropout层
32          out = self.conv2(out)                                   # 第二个卷积层
33          return torch.add(x if self.equalInOut else self.convShortcut(x), out)
    # 残差连接
34
35  class NetworkBlock(nn.Module):
36      def __init__(self, nb_layers, in_planes, out_planes, block, stride, dropRate=0.0):
37          super(NetworkBlock, self).__init__()
38          self.layer = self._make_layer(block, in_planes, out_planes, nb_layers, stride, dropRate)
39
40      def _make_layer(self, block, in_planes, out_planes, nb_layers, stride, dropRate):
41          layers = []
42          for i in range(int(nb_layers)):
43              layers.append(block(i == 0 and in_planes or out_planes,
44              out_planes, i == 0 and stride or 1, dropRate))
45          return nn.Sequential(*layers)
46
47      def forward(self, x):
48          return self.layer(x)
49
50  class WideResNet(nn.Module):
51      def __init__(self, depth, num_classes, widen_factor=1, dropRate=0.0):
52          super(WideResNet, self).__init__()
53          nChannels = [16, 16*widen_factor, 32*widen_factor, 64*widen_factor]
    # 每个阶段的通道数
54          assert((depth - 4) % 6 == 0)                            # 确保深度(depth)可以被6整除，并且至少为4
55          n = (depth - 4) / 6                                     # 每个阶段的块数
56          block = BasicBlock                                      # 使用的基本块类型为BasicBlock
57
58          # 第一个卷积层，输入通道数为3（RGB图像），输出通道数为nChannels[0]，
59          使用3x3的卷积核
60          self.conv1 = nn.Conv2d(3, nChannels[0], kernel_size=3, stride=1, padding=1, bias=False)
61
```

```
62          # 三个网络阶段，每个阶段有n个块
63          self.block1 = NetworkBlock(n, nChannels[0], nChannels[1], block, 1, dropRate)
64          self.block2 = NetworkBlock(n, nChannels[1], nChannels[2], block, 2, dropRate)
65          self.block3 = NetworkBlock(n, nChannels[2], nChannels[3], block, 2, dropRate)
66
67          # 全局平均池化和分类器
68          self.bn1 = nn.BatchNorm2d(nChannels[3])
69          self.relu  = nn.ReLU(inplace=True)
70          self.fc  = nn.Linear(nChannels[3], num_classes)   # 全连接层，输出为num_classes个类别
71          self.nChannels = nChannels[3]                     # 最后一个阶段的输出通道数
72
73          # 初始化所有层的权重和偏置
74          for m in self.modules():
75              if isinstance(m, nn.Conv2d):
76                  nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
    # 卷积层权重初始化
77              elif isinstance(m, nn.BatchNorm2d):
78                  m.weight.data.fill_(1)              # BN层权重初始化
79                  m.bias.data.zero_()                 # BN层偏置初始化
80              elif isinstance(m, nn.Linear):
81                  m.bias.data.zero_()                 # 全连接层偏置初始化
82
83      def forward(self, x):
84          out = self.conv1(x)
85          out = self.block1(out)                      # 第一个网络阶段
86          out = self.block2(out)                      # 第二个网络阶段
87          out = self.block3(out)                      # 第三个网络阶段
88          out = self.relu(self.bn1(out))              # BN和ReLU
89          out = F.avg_pool2d(out, 8)                  # 全局平均池化，kernel_size为8
90          out = out.view(-1, self.nChannels)          # 展平操作
91          return self.fc(out)                         # 分类器，输出分类结果
```

4.3 数据增强

在训练深度学习模型时，数据增强是一种常用的技术，通过对训练数据进行随机变换来生成新的样本，从而提高模型的泛化能力。以下是对不同增强模式的解释以及它们在数据集上的应用。

针对不同的数据用不同的数据增强方法：

弱增强（Weak Transform）

弱增强是一种较为保守的数据增强方法，主要通过一些简单的随机变换来增加数据的多样性。

随机水平翻转：`transforms.RandomHorizontalFlip()`，以 0.5 的概率随机水平翻转图像。增加数据多样性，适用于对称物体。

随机裁剪并填充：`transforms.RandomCrop(32, padding=4)`，裁剪尺寸为 32×32 像素，填充 4 个像素。这种方法能模拟不同的视角和位置。

转换为张量：`transforms.ToTensor()`，将 PIL 图像或 numpy 数组转换为张量。

图像标准化：`transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))`，使用 CIFAR-10 数据集的均值和标准差，使得每个通道的数据分布相对一致。

```
weak_transform = 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)),
])
```

强增强（Strong Transform）

强增强包括更多复杂的随机变换，从而显著增加数据多样性，适用于对模型鲁棒性要求较高的场景。

弱增强中的所有变换：`RandomHorizontalFlip` 和 `RandomCrop`。

颜色抖动：`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。这种方法能模拟不同的光照条件和色彩变化。

```
strong_transform = 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)),
])
```

测试增强（Test Transform）

测试增强用于验证和测试集，不包含任何随机变换，以确保评估的稳定性和一致性。

转换为张量：`transforms.ToTensor()`，将 PIL 图像或 numpy 数组转换为张量。

图像标准化：`transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))`，使用 CIFAR-10 数据集的均值和标准差。

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

从 10 个类别中随机挑选有标签数据，对有标签数据进行弱增强、对无标签数据分别进行弱增强和强增强：

```
# 获取标记和未标记的数据集，确保有标签数据从10个类别中随机挑选
def get_semi_supervised_datasets(train_dataset, num_labeled):
```

```
3
4       labeled_indices = []
5       unlabeled_indices = []
6
7       # 创建一个字典用于存储每个类别的索引列表
8       class_indices = {i: [] for i in range(10)}
9
10      # 将训练集中每个类别的索引分别存储到字典中
11      for idx, (_, label) in enumerate(train_dataset):
12          class_indices[label].append(idx)
13
14      # 随机选择 num_labeled 个样本作为有标签数据
15      labeled_indices = []
16      for label in range(10):
17          indices = class_indices[label]
18          random.shuffle(indices)
19          labeled_indices.extend(indices[:num_labeled//10])
20          unlabeled_indices.extend(indices[num_labeled//10:])
21
22      labeled_dataset = CustomDataset(Subset(train_dataset,
23      labeled_indices), weak_transform=weak_transform)
24      unlabeled_dataset = CustomDataset(Subset(train_dataset,
25      unlabeled_indices), weak_transform=weak_transform, strong_transform=strong_transform)
26
27      return labeled_dataset, unlabeled_dataset
```

4.4 训练过程

由于有标签数据和无标签数据的数据量不同，于是我学习 fixmatch 源代码的方法，每次迭代对有标签数据数据取 batch_size 个数据，对无标签数据取 mu * batch_size 个数据（源代码中 mu = 7 ）：

```
1       # 数据加载器
2       labeled_loader = DataLoader(labeled_dataset,
```

```
    batch_size=batch_size, shuffle =True, num_workers=2)

    unlabeled_loader = DataLoader(unlabeled_dataset,
    batch_size=batch_size * 7, shuffle =True, num_workers=2)
```

同时为了避免有标签数据或无标签数据取完报错异常，我也采用了 fixmatch 源代码的方法，使用迭代器获取数据：

```
for batch_idx in tqdm(range(1024), desc='Training', leave=False):

        try:
            inputs_x, targets_x = labeled_iter.__next__()
        except StopIteration:
            labeled_epoch += 1
            labeled_loader = DataLoader(labeled_loader.dataset,
            batch_size=labeled_loader.batch_size,
            shuffle =True, num_workers=labeled_loader.num_workers)
            labeled_iter = iter(labeled_loader)
            inputs_x, targets_x = labeled_iter.__next__()

        try:
            inputs_u_w, inputs_u_s, _ = unlabeled_iter.__next__()
        except StopIteration:
            unlabeled_epoch += 1
            unlabeled_loader = DataLoader(unlabeled_loader.dataset,
            batch_size=unlabeled_loader.batch_size,
            shuffle =True, num_workers=unlabeled_loader.num_workers)
            unlabeled_iter = iter(unlabeled_loader)
            inputs_u_w, inputs_u_s, _ = unlabeled_iter.__next__()
```

使用迭代器获取数据，try 块尝试从数据集中获取下一个批次的数据；如果 StopIteration 异常被捕获，说明当前迭代器已经遍历完了一个完整数据集的数据。在 except 块中，重新创建一个新的 DataLoader 对象，确保数据随机化并重新加载，然后重新生成迭代器，并从中获取下一个批次的数据。

这样做的目的是确保在训练过程中能够循环遍历有标签和无标签数据集，即使一个迭代结束时数据集迭代器也能正确地被重置和重新加载，以确保训练数据的随机性和完整性

其中需要说明的是每个 epoch 要进行代码中所示的 1024 次迭代，整个训练过程一个用了 20 个 epoch。

训练预测以及反向传播：

模型对有标签数据进行前向传播，生成预测 logits，然后使用交叉熵损失函数计算有监督损失。对于无标签数据的弱增强结果，首先利用模型生成预测 logits，并通过 softmax 函数生成伪标签。接着根据预测概率的最大值和设定的阈值 tau 过滤掉低置信度的伪标签，形成一个掩码。基于这个掩码，再次利用强增强结果和交叉熵损失函数计算无监督损失，并加权到总损失中。最后，进行执行反向传播和优化步骤。

实现代码：

```
        inputs_x, targets_x = inputs_x.to(device), targets_x.to(device)
        inputs_u_w, inputs_u_s = inputs_u_w.to(device), inputs_u_s.to(device)

        # 模型预测
        logits_x = model(inputs_x)
        logits_u_w = model(inputs_u_w)

        # 生成伪标签
        pseudo_labels = torch.softmax(logits_u_w, dim=-1)
        max_probs, targets_u = torch.max(pseudo_labels, dim=-1)

        # 过滤掉低置信度的伪标签
        mask = max_probs.ge(tau).float()

        # 计算有监督损失
        loss_x = criterion(logits_x, targets_x)

        # 计算无监督损失
        logits_u_s = model(inputs_u_s)
        loss_u = (F.cross_entropy(logits_u_s, targets_u, reduction='none')
```

```
        * mask).mean()

        # 总损失
        loss = loss_x + lambda_u * loss_u

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

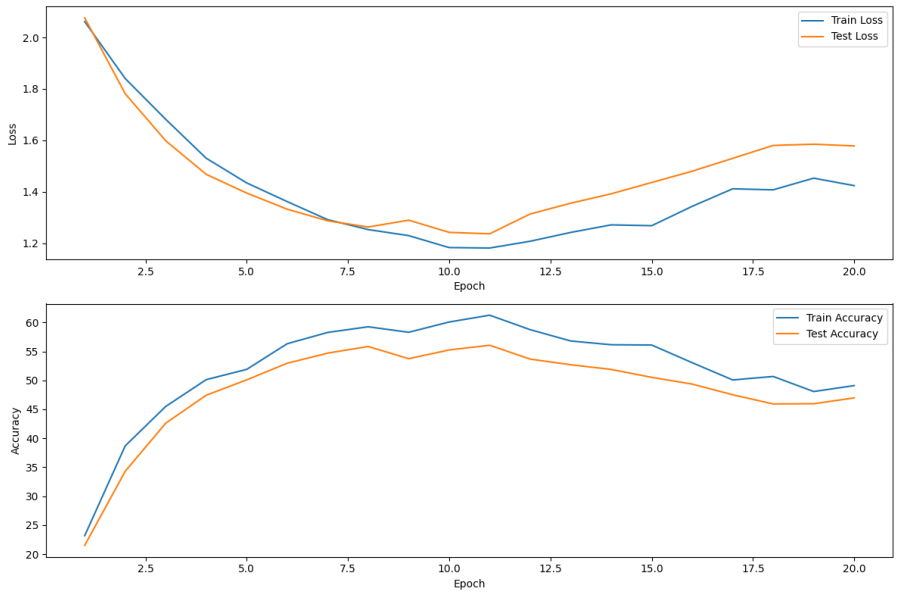
4.5 超参数设置

- ‘batch_size = 40 ‘: 每个批次中的样本数。
- WideResNet-28-2 模型的定义: ‘WideResNet28_2 ‘ 类。
- ‘learning_rate = 0.03 ‘: 优化器中使用的学习率。
- ‘weight_decay = 0.0005 ‘: 优化器中的权重衰减 (L2 正则化参数)。
- ‘num_epochs = 20 ‘: 训练的总轮数。
- 每个 epoch 有 1024 次迭代。
- 损失函数: 交叉熵损失函数 (‘nn.CrossEntropyLoss() ‘)。
- 优化器: 随机梯度下降 (‘optim.SGD ‘)。
- ‘lambda_u = 1.0 ‘: 无监督损失的权重因子, 平衡有监督损失和无监督损失。
- ‘tau = 0.95 ‘: 生成伪标签时的阈值, 用于过滤低置信度的伪标签。

5 实验结果

5.1 手动实现的 fixmatch 实验结果

(每个 epoch 有 1024 次迭代)



4000 张有标签数据

4000 张有标签数据

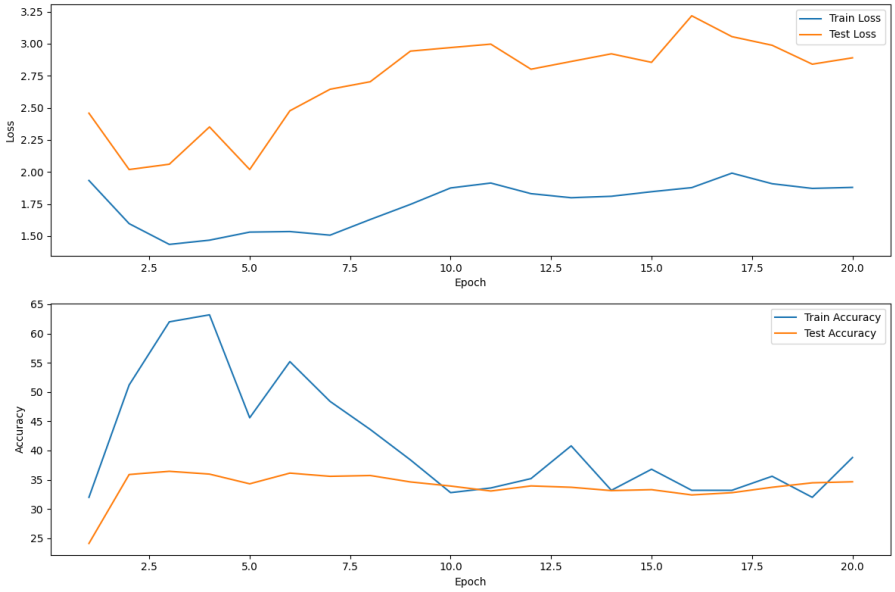
可以看到我的模型对 4000 张有标签数据的数据集上有 60% 多的准确率，可以看到在前 13 个 epoch 前，训练集和测试集上的损失在减少，准确率在升高，这表明模型此时已经学习到了训练数据中的主要特征，并且在这些特征上有较好的预测能力。

但之后准确率发生了下降，再进一步提高预测能力上出现了问题，可能是对无标签数据的预测伪标签不正确，导致无标签数据的伪标签错误对模型的性能提高产生了影响。

250 张有标签数据

我的模型在有 250 张有标签数据集上在训练刚开始时，在训练集上的效果特别好，最高能达到 60% 多的准确率，但可以看到测试集上的准确率并不高，只有 30% 左右，说明模型发生了过拟合，导致模型的泛化能力并不好，在测试集上的表现不优。

但之后训练集上准确率发生了下降，说明可能也是对无标签数据的预测伪标签不正确，导致无标签数据的伪标签错误对模型的性能提高产生了

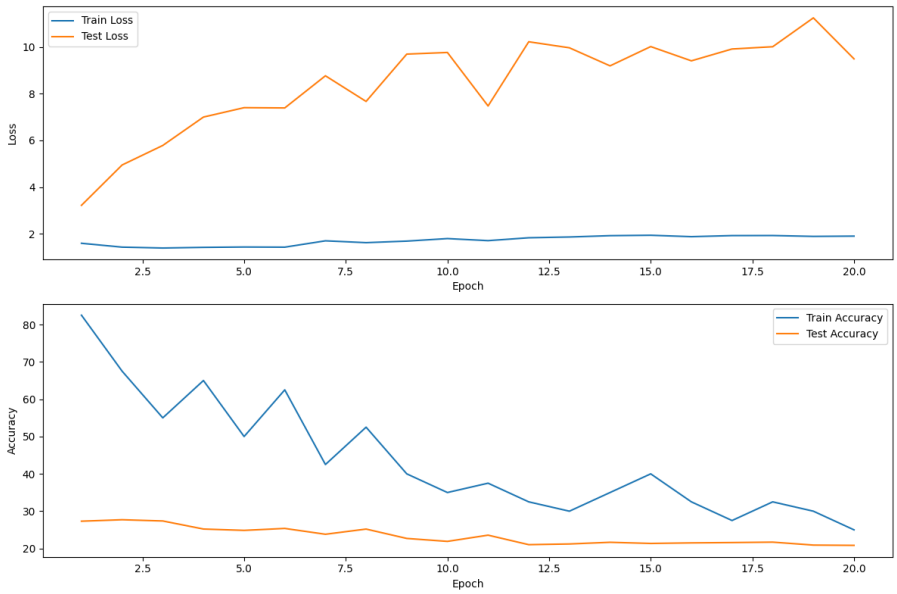


250 张有标签数据

影响。

40 张有标签数据

可以看到随着有标签数据的数量减少到了 40 张，训练过程中，模型的准确率只有 20% 多。同样可以看出 40 张有标签上的结果和 250 张有标签数据的结果有些类似，都发生了过拟合和后期训练中的缺少有标签数据而导致的性能下降。



40 张有标签数据

最终模型的预测效果：

40 张：

Epoch	训练损失	训练准确率	测试准确率
1	1.9336	32.00%	27.32%
2	1.4272	67.50%	27.71%
3	1.3889	55.00%	27.38%
4	1.4166	65.00%	25.22%
5	1.4329	50.00%	24.86%
6	1.4263	62.50%	25.38%
7	1.6978	42.50%	23.82%
8	1.6196	52.50%	25.21%
9	1.6852	40.00%	22.70%
10	1.7939	35.00%	21.91%
11	1.7053	37.50%	23.58%
12	1.8286	32.50%	21.03%
13	1.8617	30.00%	21.22%
14	1.9188	35.00%	21.67%
15	1.9360	40.00%	21.36%
16	1.8757	32.50%	21.51%
17	1.9217	27.50%	21.60%
18	1.9244	32.50%	21.71%
19	1.8878	30.00%	20.92%
20	1.8996	25.00%	20.85%

250 张：

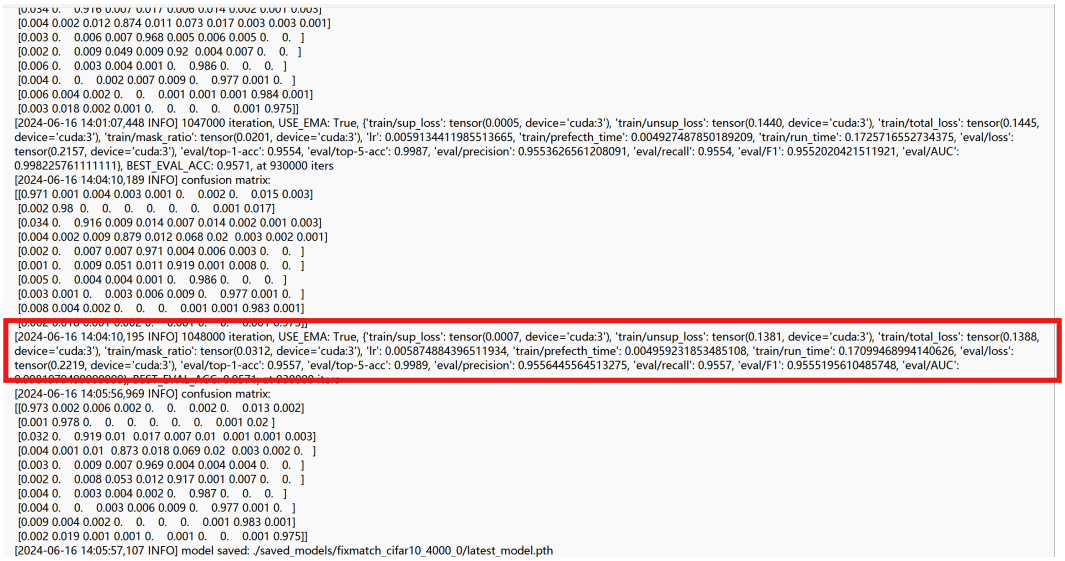
Epoch	**训练损失**	**训练准确率**	**测试准确率**
1	1.9336	32.00%	24.12%
2	1.5971	51.20%	35.90%
3	1.4351	62.00%	36.45%
4	1.4682	63.20%	35.97%
5	1.5311	45.60%	34.31%
6	1.5354	55.20%	36.14%
7	1.5072	48.40%	35.59%
8	1.6293	43.60%	35.73%
9	1.7473	38.40%	34.63%
10	1.8754	32.80%	33.93%
11	1.9137	33.60%	33.08%
12	1.8309	35.20%	33.95%
13	1.7991	40.80%	33.72%
14	1.8105	33.20%	33.15%
15	1.8461	36.80%	33.31%
16	1.8779	33.20%	32.41%
17	1.9909	33.20%	32.79%
18	1.9084	35.60%	33.72%
19	1.8717	32.00%	34.48%
20	1.8799	38.80%	34.66%

4000 张：

Epoch	训练损失	训练准确率	测试准确率
1	2.0613	23.20%	21.53%
2	1.8404	38.70%	34.31%
3	1.6819	45.50%	42.61%
4	1.5308	50.12%	47.44%
5	1.4349	51.90%	50.09%
6	1.3623	56.33%	52.96%
7	1.2921	58.27%	54.70%
8	1.2536	59.25%	55.85%
9	1.2301	58.30%	53.74%
10	1.1837	60.08%	55.25%
11	1.1820	61.25%	56.06%
12	1.2083	58.75%	53.67%
13	1.2426	56.80%	52.70%
14	1.2719	56.15%	51.89%
15	1.2686	56.10%	50.52%
16	1.3439	53.05%	49.36%
17	1.4120	50.08%	47.52%
18	1.4081	50.67%	45.93%
19	1.4533	48.08%	45.97%
20	1.4243	49.10%	46.98%

5.2 torchSSL 的 fixmatch 实验结果

由于 torchSSL 源代码设置的迭代次数很多，这里我减少了对 40 张和 250 张有标签数据的实验的迭代次数，对 4000 张有标签数据进行了完整迭代次数的实验。



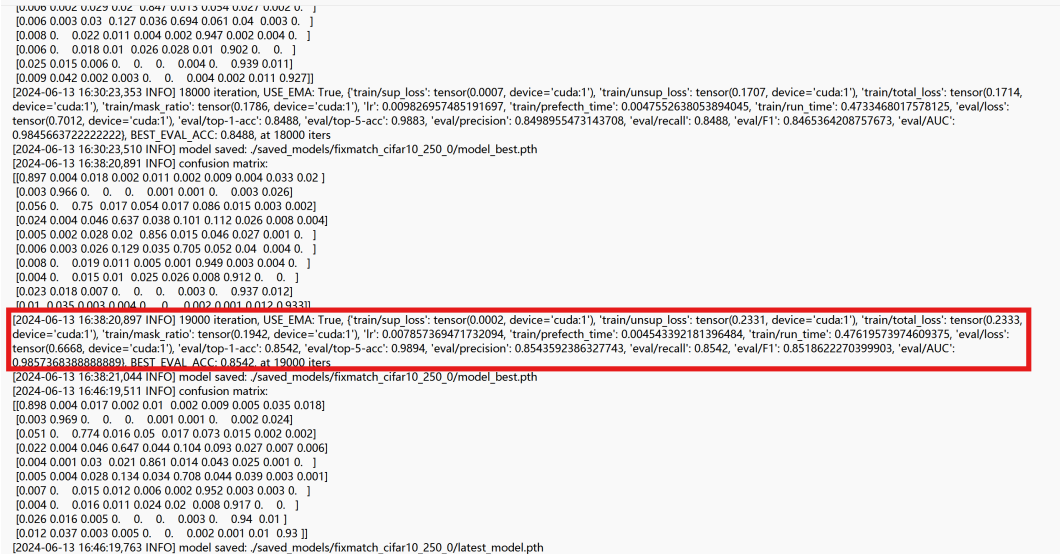
4000 张有标签数据（torchSSL）

4000 张有标签数据

信息	数值
Iteration	1048000
USE_EMA	True
train/sup_loss	0.0007
train/unsup_loss	0.1381
train/total_loss	0.1388
train/mask_ratio	0.0312
lr	0.005874884396511934
train/prefetch_time	0.004959231853485108
train/run_time	0.17099468994140626
eval/loss	0.2219
eval/top-1-acc	0.9557
eval/top-5-acc	0.9989
eval/precision	0.9556445564513275
eval/recall	0.9557
eval/F1	0.9555195610485748
eval/AUC	

可以看到 torchSSL 中的 fixmatch 对 4000 张有标签数据的 cifar10 图像分类的准确率，在经过 1048000 次迭代后可以达到 95.56%。

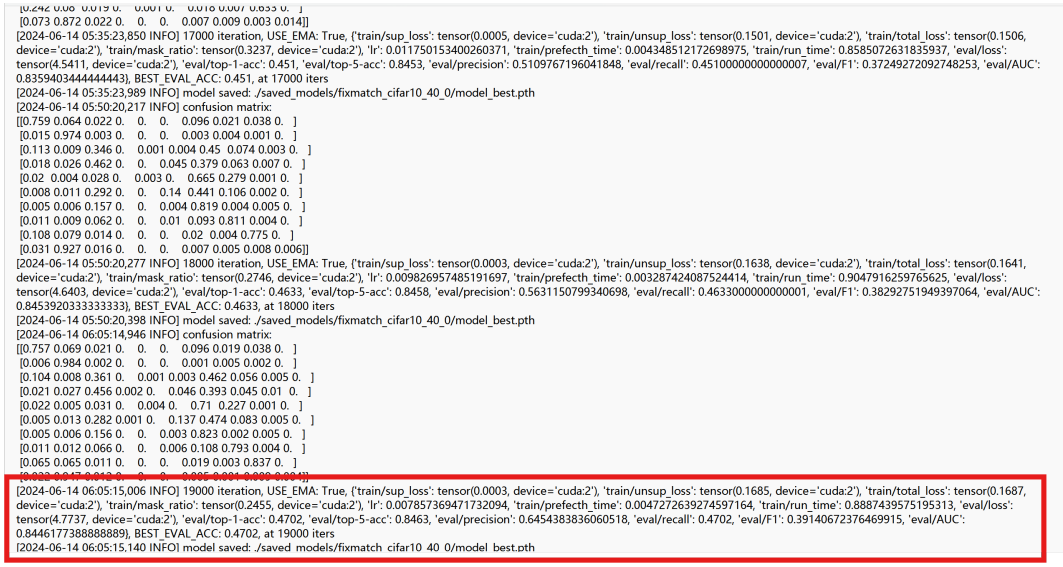
250 张有标签数据



250 张有标签数据（torchSSL）

信息	数值
Iteration	19000
USE_EMA	True
train/sup_loss	0.0002
train/unsup_loss	0.2331
train/total_loss	0.2333
train/mask_ratio	0.1942
lr	0.007857369471732094
train/prefetch_time	0.004543392181396484
train/run_time	0.47619573974609375
eval/loss	0.6668
eval/top-1-acc	0.8542
eval/top-5-acc	0.9894
eval/precision	0.8543592386327743
eval/recall	0.8542
eval/F1	0.8518622270399903
eval/AUC	0.9857368388888889
BEST_EVAL_ACC	0.8542

可以看到 torchSSL 中的 fixmatch 对 250 张有标签数据的 cifar10 图像分类的准确率，在经过 19000 次迭代后可以达到 85.43%。



40 张有标签数据（torchSSL）

40 张有标签数据

信息	数值
Iteration	19000
USE_EMA	True
train/sup_loss	0.0003
train/unsup_loss	0.1685
train/total_loss	0.1687
train/mask_ratio	0.2455
lr	0.007857369471732094
train/prefetch_time	0.0047272639274597164
train/run_time	0.8887439575195313
eval/loss	4.7737
eval/top-1-acc	0.4702
eval/top-5-acc	0.8463
eval/precision	0.6454383836060518
eval/recall	0.4702
eval/F1	0.39140672376469915
eval/AUC	0.8446177388888889
BEST_EVAL_ACC	0.4702

可以看到 torchSSL 中的 fixmatch 对 40 张有标签数据的 cifar10 图像分类的准确率，在经过 19000 次迭代后可以达到 64.54%。

所以从总体上来看,torchSSL 实现的 fixmatch 图像分类结果比我手动实现效果好很多。

6 心得体会

通过这次学习和实践，我深刻理解了 FixMatch 算法。这是一种高效且有效的半监督学习方法，其主要优势在于如何有效利用未标记数据以提升模型性能。通过结合一致性正则化和伪标签技术，FixMatch 能够在标记数据有限的情况下显著改善模型的泛化能力。这种方法不需要复杂的生成模型或对抗训练，使得实现简单且计算高效。关键的一点是通过设定置信度阈值 τ，FixMatch 能够过滤掉不可靠的伪标签，从而进一步提升训练效果。

在动手实现 FixMatch 用于图像的半监督分类过程中，我不仅加深了对 ResNet 的了解，还学习到了新的模型 WideResNet。WideResNet 在结构上进行了扩展，使得模型在处理高维度图像数据时具备更强的表达能力和泛化性能。

为了进一步提高模型的泛化能力，我研读了 FixMatch 的 PyTorch 源代码，学习到了一些训练中的小技巧。这些技巧包括如何有效地进行数据增强、优化训练过程中的超参数，以及如何处理和利用未标记数据。这些实践经验不仅增强了我对 FixMatch 算法的理解，也提高了我在半监督学习领域的实践能力。

同时通过运行 torchSSL 实现的 fixmatch 图像分类结果，可以看到其实现的效果较我手动实现的效果要好很多，故在 ifxmatch 算法实现的细节上我还要学习研究来提高自己模型的效果。

总的来说，通过这次学习和实践，我不仅掌握了 FixMatch 算法的理论基础，还通过动手实践和代码研读提升了自己的模型实现和优化能力。这次经历让我认识到理论与实践相结合的重要性，也激励我在今后的学习中不断探索和应用先进的机器学习算法。

7 参考资料

[1]. A Survey on Deep Semi-supervised Learning 2021

[2]. Temporal Ensembling for Semi-Supervised Learning

[3]. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results

[4]. Mixmatch: A holistic approach to semi-supervised learning

[5]. Fixmatch: Simplifying semi-supervised learning with consistency and confidence

[6]. https://github.com/StephenStorm/TorchSSL

[7]. https://blog.csdn.net/u011984148/article/details/105384080