PRCV 作业3

20337251 伍建霖

半监督图像分类

算法原理

半监督图像分类

半监督图像分类是一种机器学习技术，旨在利用有标签和无标签的图像数据进行分类任务。在传统的监督学习中，我们通常需要大量有标签的数据进行训练，但是标注图像数据是一项费时费力的任务，因此有时很难获得足够的有标签数据。而半监督学习则通过同时使用有标签和无标签的数据，以提高分类模型的性能。

半监督图像分类的核心思想是利用无标签数据的分布信息来帮助分类模型学习更好的特征表示。具体的方法包括以下几种常见的技术：

1. 生成模型：使用生成模型（如生成对抗网络）对无标签图像进行建模，生成与有标签图像类似的样本。这些生成的样本可以与有标签数据一起用于训练分类器，以提高分类性能。
2. 半监督聚类：使用聚类算法对无标签图像进行分组，将相似的图像归为一类。然后，可以将聚类结果作为标签来训练分类模型。
3. 协同训练：将分类任务划分为多个子任务，每个子任务使用有标签数据和部分无标签数据进行训练。然后，使用每个子任务的分类器对无标签数据进行预测，并将预测结果作为标签来训练其他子任务的分类器。通过迭代训练过程，分类性能可以逐渐提升。
4. 自训练：在初始阶段使用有标签数据进行监督训练，然后使用训练好的分类器对无标签数据进行预测，并将置信度较高的预测结果作为伪标签加入到有标签数据中。接着，将扩充后的有标签数据与原始有标签数据一起重新训练分类器，迭代多次以提高性能。

Fixmatch

FixMatch是一种半监督学习算法，专门用于图像分类任务。该算法结合了自监督学习和协同训练的思想，通过使用有标签和无标签的数据进行训练，以提高分类器的性能。

下面是FixMatch算法的主要步骤：

1. 初始训练：使用有标签数据进行监督学习，训练一个基础的分类器模型。
2. 生成伪标签：使用基础模型对无标签数据进行预测，并选择预测概率最高的类别作为伪标签。
3. 数据扩充：对无标签数据和其对应的伪标签进行数据扩充。这可以通过应用随机的数据变换，如随机裁剪、翻转、旋转等，来增加数据的多样性和丰富性。
4. 强化标签：通过使用有标签数据和伪标签数据进行训练，更新分类器模型。在这一步中，通常使用协同训练的思想，将有标签数据和无标签数据的损失函数相结合，以最小化分类误差。
5. 重复迭代：重复执行第2至第4步，直到达到预定的训练迭代次数或其他停止准则。

FixMatch算法的核心思想是通过使用无标签数据和伪标签来扩充训练数据，并结合有标签数据一起进行训练，从而增强分类器的泛化能力。在每个训练迭代中，通过引入伪标签和数据扩充来增加训练样本的多样性，同时使用有标签数据和无标签数据进行训练以提高性能。

FixMatch算法在实践中已经取得了一定的成功，特别是在数据集有限的情况下，它能够通过利用无标签数据来提高分类器的性能。然而，如何选择合适的阈值来确定伪标签的可靠性是FixMatch算法的一个重要挑战，需要根据具体的应用场景和数据集进行调整。

Mixmatch

MixMatch是一种半监督学习算法，旨在改进图像分类任务的性能。该算法是对FixMatch算法的改进和扩展，通过结合混合数据增强和标签插值的方法，进一步提高分类器的泛化能力。

下面是MixMatch算法的主要步骤：

1. 初始训练：使用有标签数据进行监督学习，训练一个基础的分类器模型。
2. 数据增强：对有标签和无标签的数据进行数据增强，包括随机裁剪、翻转、旋转等操作。这可以增加数据的多样性，提高分类器的鲁棒性。
3. 混合数据增强：对有标签和无标签的数据进行混合。将每个有标签样本与一部分无标签样本混合在一起，形成一组带有混合标签的样本。
4. 标签插值：对混合样本的标签进行插值，生成伪标签。使用分类器对混合样本进行预测，将预测结果作为伪标签，并使用插值技术将有标签样本的标签与伪标签进行平滑融合。
5. 强化标签：使用混合样本和对应的伪标签进行训练，更新分类器模型。通过最小化分类器在混合样本上的损失函数，来优化模型参数。
6. 重复迭代：重复执行第2至第5步，进行多轮训练，直到达到预定的训练迭代次数或其他停止准则。

MixMatch算法通过混合数据增强和标签插值的方式，利用无标签数据的信息来增强分类器的性能。混合数据增强可以增加数据的多样性，扩展训练数据集。标签插值则利用分类器的预测结果，平滑融合有标签样本和无标签样本的标签，以提高伪标签的质量。

MixMatch算法在实践中取得了良好的性能，尤其在数据集有限的情况下能够有效利用无标签数据来提升分类器的准确性。然而，算法中的超参数设置和对标签插值的调整仍然是需要仔细考虑的关键问题，以获得最佳的性能。

fixmatch和mixmatch的异同

相同点：

1. 半监督学习目标：FixMatch和MixMatch都是半监督学习算法，使用有标签和无标签的数据进行训练，以提高分类器性能。
2. 数据增强：两种算法都使用数据增强技术，如随机裁剪、翻转、旋转等，来增加数据的多样性和丰富性。
3. 伪标签：两种算法都利用无标签数据生成伪标签，用于增加训练数据和扩展训练样本。
4. 迭代训练：两种算法都采用迭代的方式进行训练，每轮迭代都使用有标签和无标签数据进行更新和优化。

不同点：

1. 数据混合与标签插值：MixMatch引入了数据混合和标签插值的方法。在MixMatch中，将有标签样本与一部分无标签样本混合，形成混合样本，并使用标签插值技术将有标签样本的标签与伪标签进行平滑融合。这样可以更好地利用无标签数据，并使伪标签更准确。
2. 伪标签的可靠性：FixMatch在生成伪标签时使用了一个阈值来筛选可靠的预测结果作为伪标签，而MixMatch使用标签插值方法来平滑融合有标签样本和无标签样本的标签，以减少伪标签的噪声。
3. 算法性能：FixMatch和MixMatch在实践中都取得了较好的性能，但在不同的数据集和任务上可能会有所差异。FixMatch侧重于在有限的有标签数据下利用无标签数据提高分类器性能，而MixMatch通过数据混合和标签插值进一步扩展了训练数据集，对数据的多样性和丰富性要求更高。

实验步骤

模型结构：

BasicBlock 类表示 WideResNet 架构的基本模块。它由两个卷积层、批标准化层和 ReLU 激活函数组成，同时还有一个快捷连接。在 __init__ 方法中，使用输入的通道数、输出的通道数和步长来初始化卷积层、批标准化层和快捷连接。forward 方法实现了模块的前向传播过程，输入 x 通过各层后，将残差连接加到输出上，并应用 ReLU 激活函数。

WideResNet 类表示 WideResNet 架构本身。它接受三个参数: depth（每个模块中的层数）、widen_factor（通道数的扩展因子）和 num_classes（输出类别的数量）。在 __init__ 方法中，定义了 WideResNet 的各个层，包括初始卷积层、批标准化层、ReLU 激活函数以及三个不同大小的模块层。_make_layer 方法用于创建模块层，根据输入和输出通道数以及步长来构建 BasicBlock 模块，并将多个模块连接成序列。forward 方法实现了网络的前向传播过程，通过各层后，应用自适应平均池化、展平输出，并通过全连接层得到最终的 logits。

```
# 模型定义
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out += self.shortcut(residual)
        out = self.relu(out)

        return out

class WideResNet(nn.Module):
    def __init__(self, depth, widen_factor, num_classes):
        super(WideResNet, self).__init__()
        self.in_channels = 16
        self.conv1 = nn.Conv2d(3, self.in_channels, kernel_size=3, stride=1,
padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(self.in_channels)
        self.relu = nn.ReLU(inplace=True)
```

```
        self.layer1 = self._make_layer(16 * widen_factor, depth)
        self.layer2 = self._make_layer(32 * widen_factor, depth, stride=2)
        self.layer3 = self._make_layer(64 * widen_factor, depth, stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(64 * widen_factor, num_classes)

    def _make_layer(self, out_channels, blocks, stride=1):
        layers = [BasicBlock(self.in_channels, out_channels, stride)]
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(BasicBlock(out_channels, out_channels, stride=1))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avgpool(out)
        out = torch.flatten(out, 1)
        out = self.fc(out)
        return out
```

fixmatch

```
def fixmatch_train(model, labeled_loader, unlabeled_loader, optimizer, device):
    model.train()
    print("start train")

    labeled_iter = iter(labeled_loader)
    unlabeled_iter = iter(unlabeled_loader)
    num_iterations = min(len(labeled_iter), len(unlabeled_iter))
    # print("get data iterator")

    for i in range(num_iterations):
        # 加载有标签数据
        labeled_images, labels = next(labeled_iter)
        labeled_images, labels = labeled_images.to(device), labels.to(device)
        # print("load labeled data")

        # 加载无标签数据
        unlabeled_images, _ = next(unlabeled_iter)
        unlabeled_images = unlabeled_images.to(device)
        # print("load unlabeled data")

        # 生成强增强数据和预测
        strong_aug_images = torch.stack([strong_augmentation(image) for image in
unlabeled_images]).to(device)
        outputs = model(strong_aug_images)
        _, pseudo_labels = torch.max(outputs.detach(), dim=1)
        # print("get predict")

        # 计算伪标签的置信度
```

```
        confidence = torch.max(torch.softmax(outputs.detach(), dim=1), dim=1)[0]
        mask = confidence.ge(0.95).float()
        # print("get confidence threshold")

        # 有标签损失
        labeled_outputs = model(labeled_images)
        labeled_loss = nn.CrossEntropyLoss()(labeled_outputs, labels)
        # print("get labeled loss")

        # 无标签损失
        pseudo_outputs = model(strong_aug_images)
        pseudo_loss = nn.CrossEntropyLoss(reduction='none')(pseudo_outputs,
pseudo_labels)
        unlabeled_loss = (mask * pseudo_loss).mean()
        # print("get unlabeled loss")

        # 总损失
        loss = labeled_loss + unlabeled_loss
        print("loss=", loss)

        # 更新模型参数
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # print("update model")
```

mixmatch

```
# MixMatch 核心函数
def mixmatch(model, labeled_data, unlabeled_data, num_classes, T, K, alpha):
    labeled_x, labeled_y = labeled_data
    unlabeled_x = unlabeled_data

    # 扩充标记数据
    augmented_labeled_x = data_augmentation(labeled_x)

    # 扩充未标记数据
    augmented_unlabeled_x = data_augmentation(unlabeled_x)

    # 模型训练（标记数据）
    labeled_logits = model(augmented_labeled_x)
    labeled_loss = F.cross_entropy(labeled_logits, labeled_y)

    # 模型训练（未标记数据）
    unlabeled_logits = model(augmented_unlabeled_x)
    unlabeled_probs = F.softmax(unlabeled_logits, dim=1)
    unlabeled_max_probs, unlabeled_pseudo_labels = unlabeled_probs.max(dim=1)

    # MixMatch 样本生成
    mask = (unlabeled_max_probs >= threshold) # 根据阈值选择样本
    mixed_inputs = torch.cat([augmented_labeled_x, augmented_unlabeled_x[mask]])
    mixed_labels = torch.cat([labeled_y, unlabeled_pseudo_labels[mask]])
    mixed_labels = F.one_hot(mixed_labels, num_classes).float()

    # MixMatch 训练
```

```
    mixed_logits = model(mixed_inputs)
    mixed_probs = F.softmax(mixed_logits, dim=1)
    mixed_loss = (alpha * F.kl_div(mixed_probs.log(), mixed_labels,
reduction='batchmean') +
                  (1 - alpha) * F.cross_entropy(mixed_logits,
mixed_labels.argmax(dim=1)))

    # 计算总损失
    total_loss = labeled_loss + mixed_loss
    print("loss=", total_loss)

    # 优化器更新
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()
```

实验结果

下面的每组图上面的是自己实现的算法结果，下面的是TorchSSL实现的结果。可以看到在类似的运行时间下，自己实现的算法表现是不如TorchSSL中算法的表现的，同时mixmatch的表现也不如fixmatch。

fixmatch表现对比

40张标注数据表现对比

```
Epoch 58, Validation Accuracy: 12.48%
start train
loss= tensor(0.0722, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(2.2988, device='cuda:0', grad_fn=<AddBackward0>)
Epoch 59, Validation Accuracy: 11.78%
start train
loss= tensor(0.0990, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(1.4529, device='cuda:0', grad_fn=<AddBackward0>)
Epoch 60, Validation Accuracy: 11.26%
Test Accuracy: 11.36%
```

saved_models > fixmatch_cifar10_40_0 > log.txt

```
2616    [[0.828 0.006 0.021 0.    0.    0.    0.098 0.003 0.029 0.015]
2624
2625
2626     0.8804333222222223}, BEST_EVAL_ACC: 0.5792, at 48000 iters
2627
```

250张标注数据表现对比

```
Epoch 59, Validation Accuracy: 22.56%
start train
loss= tensor(0.1901, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.2396, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.2711, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.2637, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.6618, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.2924, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.6389, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.1557, device='cuda:0', grad_fn=<AddBackward0>)
Epoch 60, Validation Accuracy: 24.72%
Test Accuracy: 24.33%
```

saved_models > fixmatch_cifar10_250_0 > log.txt

```
268   [[0.434 0.067 0.072 0.056 0.031 0.016 0.034 0.017 0.223 0.05 ]
274
275
276
277
278   ': 0.8336123500000001}, BEST_EVAL_ACC: 0.4433, at 4000 iters
279
```

4000张标注数据表现对比

```
loss= tensor(0.1988, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.1507, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.4045, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.2374, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.5761, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(0.4221, device='cuda:0', grad_fn=<AddBackward0>)
Epoch 30, Validation Accuracy: 50.34%
Test Accuracy: 50.99%
```

```
[2023-06-20 20:49:57,682 INFO] 2000 iteration, USE_EMA: True, {'train/sup_loss': tensor(0.8044, device='cuda:0'), 'train/unsup_loss': tensor(0.1958, device='cuda:0'), 't
rain/total_loss': tensor(1.0002, device='cuda:0'), 'train/mask_ratio': tensor(0.8080, device='cuda:0'), 'lr': 0.02557489508540388, 'train/prefecth_time': 0.0053828477859
49707, 'train/run_time': 0.3836361389160156, 'eval/loss': tensor(1.3009, device='cuda:0'), 'eval/top-1-acc': 0.5143, 'eval/top-5-acc': 0.9488, 'eval/precision': 0.646694
523288321, 'eval/recall': 0.5143, 'eval/F1': 0.5119413222612138, 'eval/AUC': 0.9234980833333335}, BEST_EVAL_ACC: 0.5143, at 2000 iters
/home/henry/miniconda3/envs/ssl/lib/python3.7/site-packages/torch/optim/lr_scheduler.py:216: UserWarning: Please also save or load the state of the optimizer when saving
 or loading the scheduler.
  warnings.warn(SAVE_STATE_WARNING, UserWarning)
[2023-06-20 20:49:57,739 INFO] model saved: ./saved_models/fixmatch_cifar10_4000_0/model_best.pth
```

mixmatch表现对比

40张标注数据表现对比

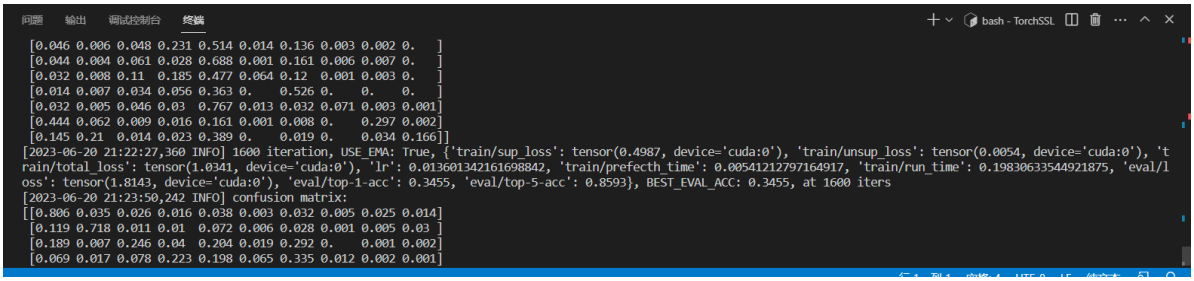
```
epoch 28
loss= tensor(6.4333, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(9.7444, device='cuda:0', grad_fn=<AddBackward0>)
Validation Accuracy: 0.1093
epoch 29
loss= tensor(6.7656, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(7.7645, device='cuda:0', grad_fn=<AddBackward0>)
Validation Accuracy: 0.1077
Test Accuracy: 0.1077
```

saved_models > mixmatch_cifar10_40_0 > log.txt

```
247   [[0.786 0.031 0.021 0.    0.01  0.047 0.014 0.018 0.073 0.   ]
253
254
255
256
257   !349, 'eval/top-5-acc': 0.7713}, BEST_EVAL_ACC: 0.2556, at 2600 iters
258
```

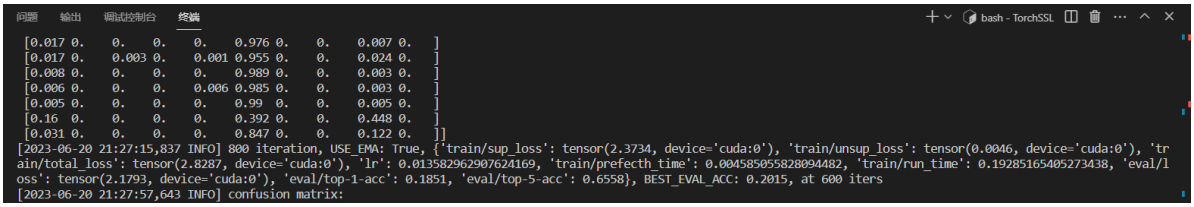
250张标注数据表现对比

```
Validation Accuracy: 0.1147
epoch 29
loss= tensor(4.1675, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(4.3608, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(4.2003, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(4.0211, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(3.6818, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(3.9445, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(4.1782, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(4.8519, device='cuda:0', grad_fn=<AddBackward0>)
Validation Accuracy: 0.1192
Test Accuracy: 0.1192
```



4000张标注数据表现对比

```
loss= tensor(2.3982, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(2.4865, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(2.3576, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(2.1823, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(2.8086, device='cuda:0', grad_fn=<AddBackward0>)
loss= tensor(2.5570, device='cuda:0', grad_fn=<AddBackward0>)
Validation Accuracy: 0.2631
Test Accuracy: 0.2631
PS D:\Course\ComputerVision\hw\hw3>
```



参考资料

[fixmatch-知乎1]

[fixmatch-知乎2]

[mixmatch-知乎]

[mixmatch-CSDN]

[github-TorchSSL]

语义分割

算法原理

语义分割

语义分割是计算机视觉中很重要的一个方向，旨在将图像分割成不同的语义区域。与传统的图像分割方法相比，语义分割不仅仅将图像分割成不同的区域，还对每个像素进行分类，将其标记为属于不同的语义类别，如人、车、树等。

语义分割在许多应用中都起着重要作用，包括自动驾驶、图像编辑、医学图像分析等。它可以帮助计算机理解图像中的不同物体和场景，并为后续的分析和决策提供基础。

在语义分割任务中，通常使用深度学习方法，特别是卷积神经网络。CNN可以通过学习大量标注图像的语义信息，从像素级别上进行分类，从而实现语义分割。

一种常见的语义分割网络是全卷积网络FCN，它使用卷积和反卷积操作，将输入图像映射到像素级别的预测。还有其他的语义分割网络，如U-Net、DeepLab等，它们在网络架构和特征融合等方面有所不同，但都旨在实现准确的语义分割。

通过语义分割，我们可以实现对图像中每个像素的语义理解，从而为许多计算机视觉任务提供支持，如目标检测、实例分割、图像生成等。

FCN

FCN是一种用于语义分割的深度学习网络架构。它在2015年由Jonathan Long等人提出，并在语义分割任务上取得了较好的性能。

传统的卷积神经网络在最后一层通常使用全连接层进行分类，因此无法直接处理变尺寸的输入。而FCN通过将全连接层替换为全卷积层，使网络能够处理任意尺寸的输入图像，并输出与输入图像相同尺寸的语义分割结果。

FCN的基本思想是将卷积神经网络中的池化层进行反池化操作，以实现空间尺寸的上采样。这样，在网络中逐渐进行卷积和上采样操作后，最终得到与输入图像尺寸相同的分割结果。为了丰富特征表达能力，FCN还引入了跳跃连接的概念，即将中间层的特征与上采样后的特征进行融合，以提高分割结果的精度。

在训练阶段，FCN使用像素级别的标注数据进行监督学习。通过最小化预测分割结果与标注之间的差异，网络的参数得到优化，从而使网络能够准确地对图像进行语义分割。

FCN已经被广泛应用于语义分割领域，并在许多公开数据集上取得了优秀的结果。它为语义分割任务提供了一种高效而有效的深度学习解决方案，并为后续的改进和发展奠定了基础。

实验步骤

模型结构，这里我直接使用预训练好的fcn模型

```
model_ft = fcn_resnet50(pretrained=True) # 设置True，表明要加载使用训练好的参数
```

读入voc数据集

```
def read_voc_images(root="./data/VOCdevkit/VOC2007", is_train=True,
max_num=None):
    if is_train == False:
        root = "./data/VOCdevkit/VOC2007_test"
    txt_fname = '%s/ImageSets/Segmentation/%s' % (root, 'train.txt' if is_train
else 'test.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split() # 拆分成一个个名字组成list
    if max_num is not None:
```

```
        images = images[:min(max_num, len(images))]
    features, labels = [None] * len(images), [None] * len(images)
    for i, fname in tqdm(enumerate(images)):
        # 读入数据并且转为RGB的 PIL image
        features[i] = Image.open('%s/JPEGImages/%s.jpg' % (root,
fname)).convert("RGB")
        labels[i] = Image.open('%s/SegmentationClass/%s.png' % (root,
fname)).convert("RGB")
    return features, labels # PIL image 0-255
```

将标签图转成具体的label，由于语义分割的目标是一副图，所以label和以往实验不一样，花了我不少时间在这步上

```
# 标签中每个RGB颜色的值
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]
# 标签其标注的类别
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']

colormap2label = torch.zeros(256**3, dtype=torch.uint8) # torch.Size([16777216])
for i, colormap in enumerate(VOC_COLORMAP):
    # 每个通道的进制是256，这样可以保证每个 rgb 对应一个下标 i
    colormap2label[(colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i

# 构造标签矩阵
def voc_label_indices(colormap, colormap2label):
    colormap = np.array(colormap.convert("RGB")).astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256 + colormap[:, :,
2])
    return colormap2label[idx] # colormap 映射 到colormaplabel中计算的下标
```

数据预处理

```
def voc_rand_crop(feature, label, height, width):
    """
    随机裁剪feature(PIL image) 和 label(PIL image).
    为了使裁剪的区域相同，不能直接使用RandomCrop，而要像下面这样做
    Get parameters for ``crop`` for a random crop.
    Args:
        img (PIL Image): Image to be cropped.
        output_size (tuple): Expected output size of the crop.
    Returns:
        tuple: params (i, j, h, w) to be passed to ``crop`` for random crop.
    """
    i,j,h,w = torchvision.transforms.RandomCrop.get_params(feature, output_size=
(height, width))
    feature = torchvision.transforms.functional.crop(feature, i, j, h, w)
```

```
    label = torchvision.transforms.functional.crop(label, i, j, h, w)
    return feature, label
```

创建数据集

```
class VOCSegDataset(torch.utils.data.Dataset):
    def __init__(self, is_train, crop_size, voc_dir, colormap2label,
max_num=None):
        """
        crop_size: (h, w)
        """
        # 对输入图像的RGB三个通道的值分别做标准化
        self.rgb_mean = np.array([0.485, 0.456, 0.406])
        self.rgb_std = np.array([0.229, 0.224, 0.225])
        self.tsf = torchvision.transforms.Compose([
            torchvision.transforms.ToTensor(),
            torchvision.transforms.Normalize(mean=self.rgb_mean,
std=self.rgb_std)])
        self.crop_size = crop_size # (h, w)
        features, labels = read_voc_images(root=voc_dir, is_train=is_train,
 max_num=max_num)
# 由于数据集中有些图像的尺寸可能小于随机裁剪所指定的输出尺寸，这些样本需要通过自定义的filter函
数所移除
        self.features = self.filter(features) # PIL image
        self.labels = self.filter(labels)     # PIL image
        self.colormap2label = colormap2label
        print('read ' + str(len(self.features)) + ' valid examples')

    def filter(self, imgs):
        return [img for img in imgs if (
            img.size[1] >= self.crop_size[0] and img.size[0] >=
self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
*self.crop_size)
                                # float32 tensor           uint8 tensor (b,h,w)
        return (self.tsf(feature), voc_label_indices(label,
self.colormap2label))

    def __len__(self):
        return len(self.features)

batch_size = 4
crop_size = (320, 480)
max_num = 20000

# 创建训练集和测试集的实例
voc_train = VOCSegDataset(True, crop_size, voc_dir, colormap2label, max_num)
voc_test = VOCSegDataset(False, crop_size, voc_dir, colormap2label, max_num)

# 设批量大小为32，分别定义【训练集】和【测试集】的数据迭代器
num_workers = 0 if sys.platform.startswith('win32') else 4
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
                                drop_last=True, num_workers=num_workers)
```

```
test_iter = torch.utils.data.DataLoader(voc_test, batch_size, drop_last=True,
                                        num_workers=num_workers)

# 方便封装，把训练集和验证集保存在dict里
dataloaders = {'train':train_iter, 'val':test_iter}
dataset_sizes = {'train':len(voc_train), 'val':len(voc_test)}
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

创建模型并开始训练

```
num_classes = 21 # 21分类，1个背景，20个物体
model_ft = fcn_resnet50(pretrained=True) # 设置True，表明要加载使用训练好的参数
model_ft = model_ft.to(device)

def train_model(model:nn.Module, criterion, optimizer, scheduler,
num_epochs=20):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    # 每个epoch都有一个训练和验证阶段
    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs-1))
        print('-'*10)
        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train()
            else:
                model.eval()
            runing_loss = 0.0
            runing_corrects = 0.0
            # 迭代一个epoch
            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)
                optimizer.zero_grad() # 零参数梯度
                # 前向，只在训练时跟踪参数
                with torch.set_grad_enabled(phase=='train'):
                    logits = model(inputs)  # [5, 21, 320, 480]
                    logits = list(logits.values())[0]
                    loss = criterion(logits, labels.long())
                    # 后向，只在训练阶段进行优化
                    if phase=='train':
                        loss.backward()
                        optimizer.step()
                # 统计loss和correct
                runing_loss += loss.item()*inputs.size(0)
                runing_corrects +=
torch.sum((torch.argmax(logits.data,1))==labels.data)/(480*320)

            epoch_loss = runing_loss / dataset_sizes[phase]
            epoch_acc = runing_corrects.double() / dataset_sizes[phase]
            print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss,
epoch_acc))
            # 深度复制model参数
            if phase=='val' and epoch_acc>best_acc:
```

```
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
        print()
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed//60,
time_elapsed%60))
    # 加载最佳模型权重
    model.load_state_dict(best_model_wts)
    return model

epochs = 5 # 训练5个epoch
criteon = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_ft.parameters(), lr=0.001, weight_decay=1e-4,
momentum=0.9)
# 每3个epochs衰减LR通过设置gamma=0.1
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)

# 开始训练
model_ft = train_model(model_ft, criteon, optimizer, exp_lr_scheduler,
num_epochs=epochs)
```

测试模型

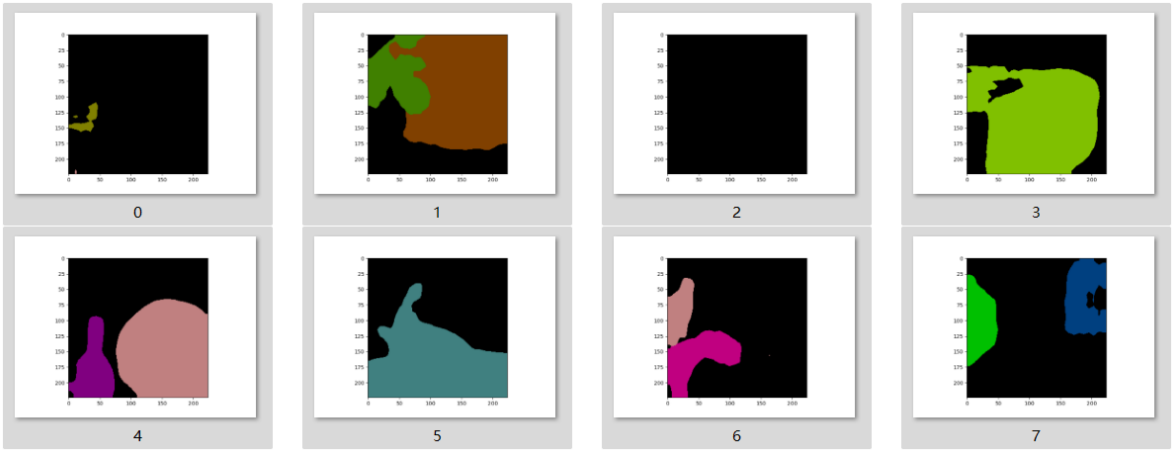
```
def label2image(pred):
    # pred: [320,480]
    colormap = torch.tensor(VOC_COLORMAP,device=device,dtype=int)
    x = pred.long()
    return (colormap[x,:]).data.cpu().numpy()

# 预测前将图像标准化，并转换成(b,c,h,w)的tensor
def predict(img, model):
    tsf = transforms.Compose([
            transforms.ToTensor(), # 好像会自动转换channel
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])])
    x = tsf(img).unsqueeze(0).to(device) # (3,320,480) -> (1,3,320,480)
    pred = torch.argmax(list(model(x).values())[0], dim=1) # 每个通道选择概率最大的那
个像素点 -> (1,320,480)
    return pred.reshape(pred.shape[1],pred.shape[2]) # reshape成(320,480)

def evaluate(model:nn.Module):
    model.eval()
    test_images, test_labels = read_voc_images(voc_dir, is_train=False)
    n, imgs = 4, []
    for i in range(10):
        xi, yi = voc_rand_crop(test_images[i], test_labels[i], 224, 224) # Image
        pred = label2image(predict(xi, model))
        imgs += [xi, pred, yi]
        plt.imshow(pred)
        plt.savefig(f'result/trained/{i}.png')
        # plt.show()
    print("finished")


# 开始测试
```

```
evaluate(model_ft)
```

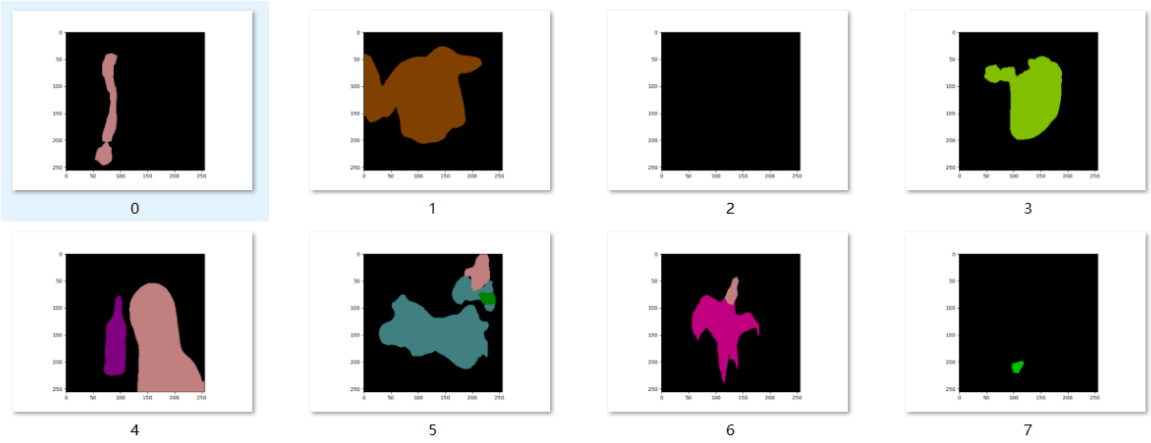
实验结果

下图中的1到8依次为000068，000175，000243，000333，000346，000364，000392和000452。

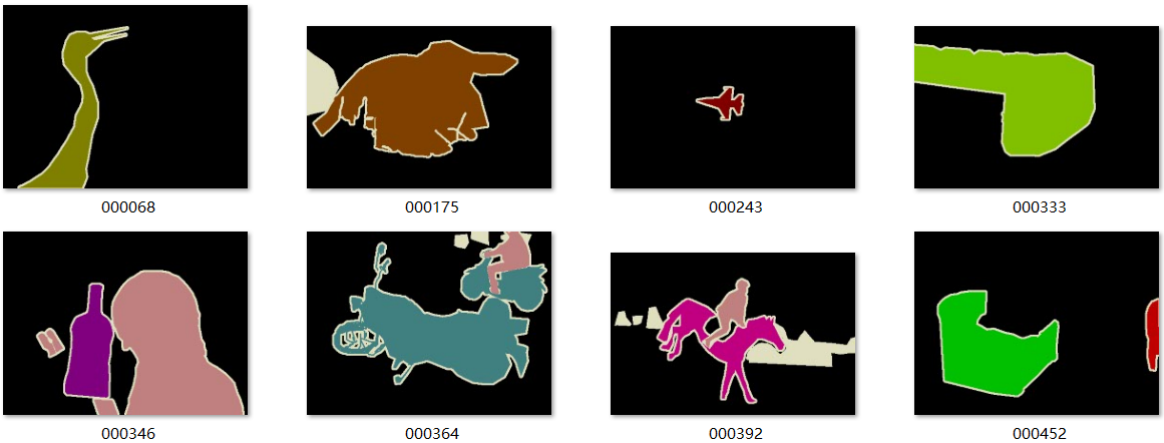
在预训练好的模型上训练后的结果：



直接使用预训练好的模型进行语义分割的结果：



正确的分割结果：



参考资料

[语义分割-知乎]

[FCN-CSDN]