Fine-grained classification

实验要求

1. Various training tips and tricks to improve model performance
2. Transfer learning: fine-tune pretrained model
3. Synthetic images by generative models for data augmentation

一、模型训练技巧和常见问题

随机种子

TODO: 使用随机的随即种子，重复多次实验，在统计学意义上记录结果

```
# 生成随机种子
random_seed = random.randint(1, 10000)
print("Random seed:", random_seed)
# 设置随机种子
torch.manual_seed(random_seed)
np.random.seed(random_seed)
torch.cuda.manual_seed_all(random_seed)
```

记录结果（准确率）的均值、方差、置信区间?

【待完成：未训练出完善的模型】

GPU资源

1. 优化模型代码和训练代码，任重道远

   对于较小的模型（ResNet18、ResNet34等），参考模型代码、模型结构图，估计模型的参数量以及占用空间，对照函数输出的模型参数量和查询到的模型参数量，看看估计是否准确，相差的参数可能在哪里?

   以预训练模型ResNet18为例运行：与实际统计结果比较，估计准确

   ```
   def get_parameter_number(net):
       total_num = sum(p.numel() for p in net.parameters())
       trainable_num = sum(p.numel() for p in net.parameters() if
   p.requires_grad)
       return {'Total': total_num, 'Trainable': trainable_num}
   # output: {'Total': 11279112, 'Trainable': 102600}
   ```

   ```
   # 使用summary(model, input_size=(3,256,256), batch_size=-1, device='cuda')统计参数量

   ----------------------------------------------------------------
           Layer (type)               Output Shape         Param #
   ================================================================
               Conv2d-1         [-1, 64, 128, 128]           9,408
          BatchNorm2d-2         [-1, 64, 128, 128]             128
                 ReLU-3         [-1, 64, 128, 128]               0
            MaxPool2d-4           [-1, 64, 64, 64]               0
   ```

```
            Conv2d-5           [-1, 64, 64, 64]          36,864
       BatchNorm2d-6           [-1, 64, 64, 64]             128
    ......
    ......
           Conv2d-63            [-1, 512, 8, 8]       2,359,296
      BatchNorm2d-64            [-1, 512, 8, 8]           1,024
             ReLU-65            [-1, 512, 8, 8]               0
       BasicBlock-66            [-1, 512, 8, 8]               0
AdaptiveAvgPool2d-67            [-1, 512, 1, 1]               0
           Linear-68                  [-1, 200]         102,600
================================================================
Total params: 11,279,112
Trainable params: 102,600
Non-trainable params: 11,176,512
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 82.01
Params size (MB): 43.03
Estimated Total Size (MB): 125.78
----------------------------------------------------------------
```

2. 使用混合精度训练：通常tensor以单精度浮点型存储数据（float32），大小32Bits；半精度浮点型（float16）大小为16Bits，使用半精度可以降低一半的显存占用，提高计算速度。在config.py中将‘use_fp16’设置为True使用混合精度训练，具体执行代码在train.py中搜索'use_fp16'

   为什么不全部使用半精度，单精度和半精度的范围、精度分别是什么，混合精度训练中哪些部分使用了半精度，autocast和GradScaler的实现原理?

   模型训练中将模型的参数使用半精度进行计算，减少显存占用，提高计算速度；而梯度计算使用单精度,减少计算精度损失，保持模型训练的准确性

   - autocast：可以用于包装训练过程中的前向传播和反向传播，使得前向传播中使用半精度计算，而反向传播中使用单精度计算。能够节省显存并提高计算速度，同时保持一定的训练精度。
   - GradScaler：用于动态调整梯度的缩放比例，以解决使用半精度计算时梯度下溢（gradient underflow）的问题。在反向传播时对梯度进行缩放，从而避免梯度过小导致的数值计算不稳定性。
3. 降低batch size, 使得一个batch输入更少的数据，代价是更多的计算时间，以及（很可能的）模型性能下降。思考为什么会引起模型性能下降，在资源充足的情况下，batch size是不是越大越好？联系优化器的优化过程
   - 降低batch size使每次batch的样本数减少，随机性上升，导致该批量梯度不能准确反应整个数据集的梯度方向，如此就需要更多次的梯度更新来达到收敛，同时噪声变大也会导致泛化能力下降。
   - 较大的批量大小可能会降低训练速度和内存需求，但也会增加收敛时间和梯度更新的难度
4. 冻结部分模型参数，在某些情形下（特别是迁移学习），可以通过冻结模型的部分参数，使其不参与训练，来降低GPU资源占用，思考这样做的利弊。
   - 利：能够有效利用GPU资源，冻结模型参数能够不受训练影响，保持原有特征，减少过拟合风险
   - 弊：缺乏灵活性，被冻结的参数无法接受训练，需要根据任务合理选择冻结部分

多GPU并行

代码并行：将模型代码的不同部分拆分为流水线，分配不同的GPU进行计算，**TODO**:对模型代码进行流水线上的重构设计，速度瓶颈是流水线中耗时最长的部分

【待完成】

交叉验证

为什么不使用留一法交叉验证，原理和利弊是什么，适用于什么情形?

留一法将数据集中的每个样本依次作为验证集，其余样本作为训练集，重复这个过程直到所有样本都被用于验证。优势在于可以最大程度地利用数据，但是计算成本非常高，不适合大规模数据集，且每次验证只有一个样本，模型评估的方差较大，可能导致评估结果不稳定。适用于适用于数据集较小、计算资源充足的情况。

损失函数

本项目代码默认使用交叉熵损失（cross entropy），不同损失函数使用与不同的问题，针对面临的问题特点，**尝试和设计不同的损失函数**，对比模型性能

【待完成：未训练出完善的模型】

> ## 参考思路
>
> ### 1. 分类问题：
>
> - **交叉熵损失（Cross Entropy Loss）**：适用于多分类问题，尤其是分类任务中类别不平衡的情况下。
> - **Focal Loss**：针对类别不平衡问题，通过调节难易样本的权重，强调困难样本的重要性，可以有效应对类别不平衡和易学样本带来的问题。
> - **Label Smoothing Loss**：通过对真实标签添加噪声，减少模型对训练集的过拟合，提高模型的泛化能力。
> - **Dice Loss**：适用于目标分割任务，尤其是医学图像分割等领域，可以有效处理边缘模糊和类别不平衡问题。
>
> ### 2. 回归问题：
>
> - **均方误差损失（Mean Squared Error Loss）**：适用于回归问题，例如预测房价、股票价格等连续值预测任务。
> - **Huber Loss**：对离群点更加鲁棒，适用于存在噪声或异常值的回归任务。
> - **Smooth L1 Loss**：在均方误差和Huber Loss之间提供了平滑的转换，可以兼顾精度和鲁棒性。
>
> ### 3. 对抗生成网络（GAN）：
>
> - **生成器损失**：通常使用负交叉熵损失或者均方误差损失，目标是使生成器生成的样本更接近真实样本分布。
> - **判别器损失**：通常使用正交叉熵损失或者均方误差损失，目标是使判别器能够准确区分真实样本和生成样本。

> ### 模型性能对比：
>
> - 在设计不同损失函数后，需要通过实验对比模型在测试集上的性能。可以使用准确率、召回率、F1值等指标进行评估，选择性能最优的损失函数。

优化器

优化器（optimizer）辅助根据模型梯度进行模型权重更新，不同的优化器适用于不同问题和模型，针对面临问题和模型特点，**TODO**: 尝试和设计不同的优化器，对比模型性能

【待完成：未训练出完善的模型】

> ### 参考
>
> - **Adam**：适用于数据集规模较大、模型复杂度较高的情况，能够提供较快的收敛速度和较好的泛化能力。
> - **SGD**：在数据集规模较小、模型简单的情况下，SGD可能表现得更稳定且不易过拟合，适用于资源受限或者对模型复杂度要求较低的场景。
> - **AdamW**：如果模型使用了批归一化等正则化技术，并且需要更好地控制权重衰减的影响，可以考虑使用AdamW。

学习率下降策略

学习率通过修改`config.py`中的`lr`参数修改初始学习率，学习率下降策略（lr scheduler）决定学习率如何下降，于`trainer.py`中的`_get_lr_scheduler`设置。针对面临问题和模型特点，**TODO**尝试和设计不同学习率下降策略，对比模型性能

【待完成：未训练出完善的模型】

> 1. **ReduceLROnPlateau**：
>   - **原理**：监控验证集损失（或指定指标）的变化，在损失不再下降时降低学习率。
>   - **适用情况**：适用于需要动态调整学习率以应对训练过程中损失波动的情况，可以提高模型收敛速度和稳定性。例如训练过程中出现了震荡或者验证集损失长时间不降时。
> 2. **MultiStepLR**：
>   - **原理**：根据预先指定的里程碑（milestone）来调整学习率，在指定的里程碑处降低学习率。
>   - **适用情况**：适用于需要在训练的特定时期降低学习率的情况，例如在训练过程中逐渐减小学习率以加速收敛。
> 3. **CosineAnnealingLR**：
>   - **原理**：使用余弦退火调整学习率，学习率在每个周期内按照余弦函数变化。
>   - **适用情况**：适用于需要平滑降低学习率并在训练后期加强探索能力的情况，有助于跳出局部最优解。
> 4. **CosineAnnealingWarmRestarts**：
>   - **原理**：带热重启的余弦退火调整学习率，周期性地降低学习率，并在每个周期开始前预热学习率。
>   - **适用情况**：适用于需要周期性调整学习率，同时希望在每个周期开始时加快学习率的变化，有助于探索更广泛的解空间。

评价指标

评价指标（metrics）用于评价模型的性能表现，对比分类问题，准确率是最普遍的评价指标，不同评价指标适用于不同问题以及评价模型的不同方面性能。例如，精确度（precision）、召回率（recall）、F1值（F-Score）、AUROC，混淆模型等常用指标。**TODO**针对问题特点和模型了解和尝试不同的评价指标

【待完成：未训练出完善的模型】

细粒度分类问题可能对各个类别的识别准确性都比较看重，因此可以考虑使用F1值或AUC-ROC作为评价指标

```
# 参考代码
from sklearn.metrics import f1_score, roc_auc_score

# 示例数据，假设y_true为真实标签，y_pred为模型预测概率
y_true = [0, 1, 0, 1]
y_pred = [0.1, 0.9, 0.3, 0.8]
# 计算F1值
f1 = f1_score(y_true, [1 if pred > 0.5 else 0 for pred in y_pred])

# 计算AUC-ROC
auc_roc = roc_auc_score(y_true, y_pred)
```

1. **准确率（Accuracy）**：
    - **适用场景**：当各个类别的样本数量相对平衡，且每个类别的重要性相同时，可以使用准确率作为评价指标。准确率可以反映模型正确预测的样本占总样本数的比例。
2. **精确度（Precision）**：
    - **适用场景**：当对某个类别的识别准确性更为关注时，可以使用精确度作为评价指标。精确度衡量的是模型预测为某一类别的样本中，真实属于该类别的比例。
3. **召回率（Recall）**：
    - **适用场景**：当对某个类别的查全率更为关注时，可以使用召回率作为评价指标。召回率衡量的是模型正确识别为某一类别的样本数占该类别总样本数的比例。
4. **F1值（F-Score）**：
    - **适用场景**：综合考虑精确度和召回率时，可以使用F1值作为评价指标。F1值是精确度和召回率的调和平均数，既考虑了模型的准确性又考虑了模型的查全率。
5. **AUC-ROC（Area Under the Receiver Operating Characteristic Curve）**：
    - **适用场景**：当需要评估模型在不同阈值下的分类性能时，可以使用AUC-ROC作为评价指标。AUC-ROC反映的是模型在正例和负例之间判别能力的强弱，值越大表示模型性能越好。
6. **混淆矩阵（Confusion Matrix）**：
    - **适用场景**：用于详细分析模型在每个类别上的分类结果，包括真正例、假正例、真负例、假负例等，有助于了解模型的分类效果和误判情况。

过拟合和欠拟合

处理过拟合有早停机制`EarlyStopping`，Lp正则、Dropout、批标准化、数据增强、集成学习等

处理欠拟合有数据归一化，增加迭代次数（结合使用较小的lr），增大batch size，换用更复杂的模型

Lp正则

Lp正则（Lp regularization）指的是给损失函数加入Lp范数（Lp norm）形式的惩罚因子。`torch.optim`集成的优化器自带L2正则，在定义优化器时设置`weight_decay`参数可设置L2正则的λ参数，例如修改`config.py`中超参数`INIT_TRAINER`中的`weight_decay`。在`train.py`搜索`_get_loss`，修改损失函数的相关代码，**TODO**:加入L1正则或自定义Lp正则

在训练和验证过程的计算loss下添加如下代码加入L1正则和Lp正则（此处计算无穷范数），使用时将`config.py`中的`weight_decay`置0关闭L2正则，设置`l1_reg`和`lp_reg`调整正则的权重参数

```
loss = criterion(output, target)

# 添加L1正则
if self.l1_reg is not None:
   l1_loss = 0
   for param in self.net.parameters():
        l1_loss += torch.sum(torch.abs(param)).to("cuda")
        loss += self.l1_reg * l1_loss

# 添加自定义Lp正则
if self.lp_reg is not None:
    lp_loss = 0
    for param in self.net.parameters():
        lp_loss += torch.norm(param, p=float('inf')).to("cuda")
        loss += self.lp_reg * lp_loss
```

Dropout

Dropout是神经网络中一种特殊的层，一般在全连接后插入。在每次训练批次的前向传播中，随机丢弃一部分神经元（同时丢弃其对应的连接边，即这些神经元不工作），来减少过拟合。

`vision_transformer.py`已经实现dropout正则化，通过设置`config.py`的超参数`INIT_TRAINER`的`dropout`设置dropout的比例。

`vision_transformer.py`中还提供了`attention_dropout`，查看代码（和论文），分析两种dropout的作用区别，并在`trainer.py`和`config.py`中搜索`dropout`，**TODO**:修改相关代码，添加`attention_dropout`设置。

- Dropout是一种常用的正则化技术，用于在训练过程中随机将神经元的输出置为零，从而减少过拟合现象
- Attention Dropout是一种专门针对注意力机制的正则化技术，用于在注意力机制中随机屏蔽一部分注意力权重，减少注意力机制的过度依赖

【待完成：第四部分深入ViTs时完成】

标准化

批标准化（Batch Normalization）是神经网络的一种特殊层，在激活层前插入，用于加速网络收敛，也有降低过拟合的效果。通过调用`torch.nn.BatchNorm2d`实现，批标准化出现以后，dropout已经较少被主流网络使用，查询论文并思考这是为什么？

- 批标准化将每个批次的数据进行标准化，使得网络更加稳定，减少梯度消失或梯度爆炸的问题，而dropout通过随机地屏蔽神经元来减少模型复杂度，可能会导致网络在训练过程中的一些信息损失。
- 批标准化可以加速网络的收敛速度，dropout会随机地丢弃一部分神经元的输出，在训练过程中需要更多的迭代次数来收敛

数据增强

数据增强（Data Augmentation），指的是对原数据旋转、反转、高斯噪音等随机小扰动，或者通过超采样、负采样等技术。使输入模型的数据在原数据集基础上更具多样化，提升模型训练效果。通过`torchvision.transforms.compose`实现，在`train.py`中`transforms.Compose`设置数据集使用的数据增强方法及其顺序，**TODO**:修改和设计新的数据增强方法。

```
import data_utils.transform as tr
train_transformer = transforms.Compose([
            tr.ToCVImage(),
            tr.Resize((448,448),'linear'),
            tr.RandomHorizontalFlip(),  # 随机水平翻转
            tr.ToTensor(),
            tr.Normalize(self.train_mean,self.train_std),
        ])
```

集成学习

集成学习（ensemble learning）是指将多个训练好的基学习器（base learners）通过多种方式结合进行推理的模型，集成学习有效的理论前提是学习器好且好得不同：1. 基学习器的推理优于随机推理；2. 基学习器的错误互相独立。尝试理解这是为什么？

1. 基学习器的推理能力低于随机推理，那么集成学习将无法带来性能提升，因为基学习器的错误可能会导致集成结果更糟糕
2. 基学习器之间的错误不相关，对于某个错误存在结果正确的基学习器，集成学习才能根据正确的基学习器减少错误率，如果基学习器的错误相关，那么集成学习会根据相关的错误强化错误。

二、微调预训练模型

模型加载方法

1. 加载并使用预训练模型参数

以`resnet.py`为例修改原代码

```
def _resnet(arch, pretrained=False, progress=True, **kwargs):
    if pretrained:
        num_classes = kwargs['num_classes']
        kwargs['num_classes'] = 1000
        model = ResNet(**kwargs)
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
        model.fc = nn.Linear(in_features=model.fc.in_features,
out_features=num_classes).to("cuda")
    else:
        model = ResNet(**kwargs)

    return model
```

在`train.py`的`_get_net`函数中修改加入`self.model_pretrained`

```
net = resnet.__dict__[net_name](
                pretrained=self.model_pretrained,
                num_classes=self.num_classes
            )
```

在`config.py`通过`MODEL_PRETRAINED`修改即可。

2. 同等效果的方法

在`train.py`定义一个新函数`_get_net_model`获取训练模型

```
def _get_net_model(self,net_name):
        model_class =getattr(models,net_name)   # import torchvision.models as
models
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model = model_class(pretrained = True).to(device)
        #冻结全部层
        for param in model.parameters():
            param.requires_grad = False
        # 重置全连接层
        model.fc = nn.Linear(in_features=model.fc.in_features,
out_features=self.num_classes).to(device)

        summary(model,input_size=(3,448,448),batch_size=-1,device='cuda') # 查看
模型参数量
        return model
```

3. 加载更多预训练模型

使用`torch.hub`加载其他模型，如下用例`ntsnet`.

```
def _get_net_hub(self):
        torch.hub._validate_not_a_forked_repo=lambda a,b,c: True
        model = torch.hub.load('nicolalandro/ntsnet-cub200', 'ntsnet',
pretrained=True,trust_repo=True, **{'topN': 6, 'device':'cuda','num_classes':
200})

        return model
```

通过`config.py`中的`MODEL_METHOD`选择对应模型加载方法

迁移学习策略

1. Transfer Learning

冻结预训练模型的全部卷积层（主要是提取特征的层，消耗资源较多），只训练自己定制的全连接层（主要是分类器层，消耗资源较少）。

```
#冻结全部层
for param in model.parameters():
    param.requires_grad = False
# 重置全连接层
model.fc = nn.Linear(in_features=model.fc.in_features,
out_features=self.num_classes).to(device)
```

2. Extract Feature Vector

先计算出预训练模型的卷积层对所有训练和测试数据的特征向量，然后抛开预训练模型，只训练自己定制的简配版全连接网络（行为上和上一条类似）

3. Fine-tune

冻结预训练模型的部分层（通常是靠近输入的多数卷积层），训练剩下的层（通常是靠近输出的部分卷积层和全连接层），为什么冻结的是靠近输入的多数卷积层，训练的是靠近输出的部分卷积层和全连接层?

- 底层特征的通用性：靠近输入的多数卷积层（底层卷积层）通常学习到的是低级特征，如边缘、纹理、颜色、形状等。这些特征在不同的任务和数据集中往往都是通用的。因此，保留预训练模型的底层卷积层可以使模型在新的任务中受益于这些通用特征的提取能力。
- 数据需求的适应性：靠近输出的部分卷积层和全连接层（高层特征提取器）通常学习到的是更高级和任务特定的特征。这些层根据训练数据和任务的不同需要进行调整和训练，以适应新的任务。这些高层特征提取器对模型在新任务中的表现更为重要。
- 稳定性：预训练模型的底层卷积层已经过广泛的训练，并在各种任务中表现稳定。因此，保留这些层可以使微调过程更加稳定，减少训练的不确定性。

微调库

PEFT库是一个很好的微调库，包含许多SOTA方法。**TODO**：尝试了解、学习并调用、移植来提高你的模型性能。

【待完成】

PEFT（PyTorch Efficient Fine-Tuning）是一个用于 PyTorch 模型的迁移学习工具库，旨在简化 Fine-tuning 过程。PEFT 提供了一种简单的方式来对预训练模型进行 Fine-tuning，并自动处理数据加载、模型构建、训练、验证和测试等步骤。

三、使用生成模型

通过生成模型来生成额外数据，广义上属于超采样的数据增强技术，最经典的生成模型为生成对抗网络（GAN）和扩散模型（Diffusion Models），**TODO**:查询并深入了解这二者代表的思想异同

> 1. **目标**：
>   - **GAN**：生成对抗网络旨在通过训练一个生成器网络和一个判别器网络，让生成器能够生成逼真的数据，而判别器则负责区分真实数据和生成数据。
>   - **扩散模型**：扩散模型是一种生成模型，其目标是通过多步骤的反向传播过程，将噪声逐步扩散到真实数据分布，逐步优化生成的数据，使其接近真实数据分布。
> 2. **损失函数**：
>   - **GAN**：生成对抗网络通常使用对抗损失函数（adversarial loss），即判别器的损失和生成器的损失。判别器希望最大化正确分类真实和生成样本的概率，而生成器希望最小化被判别为生成样本的概率。
>   - **扩散模型**：扩散模型通常使用像素级别的损失函数，例如均方误差（MSE）损失或对比损失（contrastive loss），用于比较生成的图像和真实图像之间的差异。

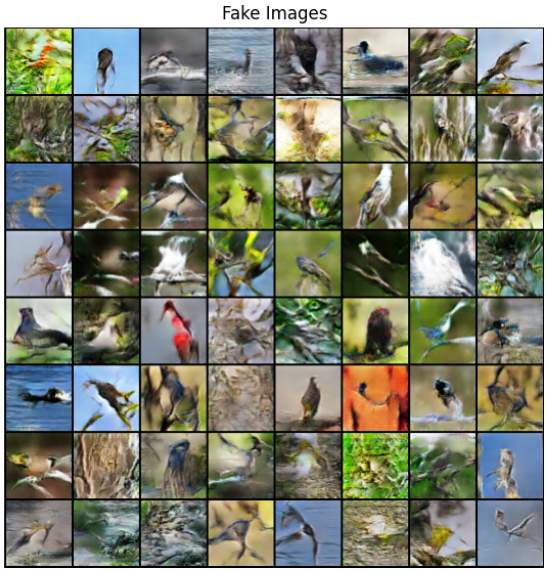
本项目实现了简单的DCGAN，**TODO**：尝试对照`train.py`、`main.py`、`config.py`，规范代码并补全相关功能

使用`config_gan.py`设置参数，内容如下，而`main_GAN.py`根据配置文件修改规范代码

```
from utils import get_weight_path
VERSION = 'v1.0'
```

```
DEVICE = 2
PRE_TRAINED = False

GEN_PIC = False
if GEN_PIC:
    PRE_TRAINED = True

if PRE_TRAINED:
    GEN_PATH = './ckpt/{}_gan/generator.pth'.format(VERSION)
    DIS_PATH = './ckpt/{}_gan/discriminator.pth'.format(VERSION)

DATA_PATH = 'gen_dataset/{}/'.format(VERSION)
DATA_NUM = 10

INIT_TRAINER = {
    'image_size':256,
    'encoding_dims':100,
    'batch_size':128,
    'epochs':10,
    'num_workers':1,
    'gen_path':GEN_PATH,
    'dis_path':DIS_PATH
}
```

生成结果：



TODO：如何获得生成的图像的标签，查阅并尝试其他生成模型是如何解决这个问题的。

- 生成器接收额外的条件信息，比如类别标签，设计成能够根据给定的条件生成对应的图像，这样生成的图像就带有了相应的标签信息。
- 同时判别器可以包含对标签信息的学习，即使生成器不直接接收标签信息，也能够通过从判别器中提取的标签信息来间接获得生成图像的标签。