
LibFewShot

发行版本 *0.0.1-alpha*

R&L Group

2023 年 04 月 09 日

1	介绍	1
2	贡献者	3
3	安装	5
3.1	获取 LibFewShot	5
3.2	配置 LibFewShot 环境	5
3.3	测试安装是否正确	6
3.4	后续	6
4	入门	7
4.1	准备数据集（以 miniImagenet 为例）	7
4.2	修改配置文件	8
4.3	运行	8
4.4	查看运行日志文件	8
5	编写 .yaml 配置文件	9
5.1	LibFewShot 中配置文件的组成	9
5.2	LibFewShot 中配置文件的设置	10
6	训练/测试 LibFewShot 中已集成的方法	15
6.1	配置文件	15
6.2	训练	16
6.3	测试	16
7	使用数据集	19
7.1	数据集格式	19
7.2	配置数据集	20

8 Transformations	21
9 增加一个新的 Backbone	23
10 增加一个新的分类器	25
10.1 metric based	27
10.2 meta learning	30
10.3 fine tuning	34
11 Model Zoo	37
12 贡献代码	39
12.1 增加一个方法/特性或修复一个错误	39
12.2 使用 pre-commit 检查代码	40
12.3 PR style	40
13 Indices and tables	41

CHAPTER 1

介绍

LibFewShot 是一个主要用于小样本学习的综合算法库。其中以统一框架集成了多种经典的小样本学习方法，包含四种基于微调的方法，六种基于元学习的方法，以及八种基于度量学习的方法。本库代码简洁、结构明了，易于新接触小样本学习的新手学习使用。

[LibFewShot: A Comprehensive Library for Few-shot Learning](#). Wenbin Li, Chuanqi Dong, Pinzhuo Tian, Tiexin Qin, Xuesong Yang, Ziyi Wang, Jing Huo, Yinghuan Shi, Lei Wang, Yang Gao, Jiebo Luo. In arXiv 2021.

CHAPTER 2

贡献者

首先特别感谢 [LegendDong](#)，建立了本库的框架并完成大部分算法。以下贡献者也参与了本库的开发过程：
[WenbinLee](#), [yangcedrus](#), [wZuck](#), [WonderSeven](#), [Pinzhuo Tian](#), [onlyyao](#), [cjl97](#)

本节给出了从零开始搭建 LibFewShot 可运行的环境的教程。

3.1 获取 LibFewShot

使用以下命令获取 LibFewShot:

```
cd ~  
git clone https://github.com/RL-VIG/LibFewShot.git
```

3.2 配置 LibFewShot 环境

可以按照下面方法配置环境

1. 创建 anaconda 环境。

```
cd <path-to-LibFewShot> # 进入clone好的LibFewShot目录  
conda create -n libfewshot python=3.7  
conda activate libfewshot
```

2. 跟随 PyTorch 和 torchvision 的官方引导进行安装。

3. 依赖包安装

- pip

```
cd <path-to-LibFewShot> # cd 进入 `LibFewShot` 目录
pip install -r requirements.txt
```

- 或者其他安装方式，只要满足：

```
numpy >= 1.19.5
pandas >= 1.1.5
Pillow >= 8.1.2
PyYAML >= 5.4.1
scikit-learn >= 0.24.1
scipy >= 1.5.4
tensorboard >= 2.4.1
torch >= 1.5.0
torchvision >= 0.6.0
python >= 3.6.0
```

3.3 测试安装是否正确

1. 修改 `run_trainer.py` 中 `config` 设置的一行为

```
config = Config("./config/test_install.yaml").get_config_dict()
```

2. 修改 `config/headers/data.yaml` 中的 `data_root` 为你的数据集路径
3. 执行

```
python run_trainer.py
```

4. 若第一个 `epoch` 输出正常，则表明 `LibFewShot` 已成功安装

3.4 后续

模型训练和代码修改可参考[训练/测试 LibFewShot](#)中已集成的方法 以及其他部分教程。

本节展示了使用 `LibFewShot` 的流程示例。

4.1 准备数据集（以 `miniImagenet` 为例）

1. 下载并解压 `miniImagnet-ravi`
2. 检查数据集格式：数据集应该具有以下格式（其它数据集如 `tieredImageNet` 等也是）

```
dataset_folder/  
├─ images/  
│   ├─ images_1.jpg  
│   └─ ...  
│   └─ images_n.jpg  
├─ train.csv *  
├─ test.csv *  
└─ val.csv *
```

4.2 修改配置文件

以 ProtoNet 为例：

1. 在 config 目录下新建一个 yaml 文件 getting_started.yaml
2. 在该文件中写入以下指令

```
includes:
- headers/data.yaml
- headers/device.yaml
- headers/losses.yaml
- headers/misc.yaml
- headers/model.yaml
- headers/optimizer.yaml
- classifiers/Proto.yaml
- backbones/Conv64FLeakyReLU.yaml
```

更细节的部分可参考编写.yaml 配置文件。

4.3 运行

1. 修改 run_trainer.py 中 config 配置语句为

```
config = Config("./config/getting_started.yaml").get_config_dict()
```

2. 执行

```
python run_trainer.py
```

3. 等待程序运行结束（你可以去喝 100 杯咖啡）

4.4 查看运行日志文件

程序运行完毕之后，可以找到链接 results/ProtoNet-miniImageNet-Conv64F-5-1 和目录 results/ProtoNet-miniImageNet-Conv64F-5-1-\$TS，其中 TS 表示时间戳。目录包含两个文件夹 checkpoint/和 log_files/和一个配置文件 config.yaml。当你多次训练同一种小样本学习方法，链接总会关联到最后创建的目录。

config.yaml 即本次训练使用的配置文件内容。

log_files 包含 tensorboard 记录文件，以及在本模型上的训练日志以及测试日志。

checkpoints 包含按照 save_interval 保存的模型文件、最后模型文件（用于 resume）和最佳模型文件（用于测试）。模型文件一般分为 emb_func.pth,classifier.pth 以及 model.pth（前两者的组合）

编写 .yaml 配置文件

本节相关代码：

```
core/config/config.py
config/*
```

5.1 LibFewShot 中配置文件的组成

LibFewShot 的配置文件采用了 yaml 格式的文件，同时也支持从命令行中读取一些全局配置的更改。我们预先定义了一个默认的配置 `core/config/default.yaml`。用户可以将自定义的配置放在 `config/` 目录下，保存为 yaml 格式的文件。配置定义在解析时的优先级顺序是 `default.yaml`→`config/`→`console`。后一个定义会覆盖前一个定义中名称相同的值。

尽管 `default.yaml` 中设置的是小样本学习中的一些最基础的配置，无法仅依靠 `default.yaml` 直接运行程序。运行代码前，用户需要在 `config/` 目录下定义已经在 LibFewShot 中实现了的方法的配置。

考虑到小样本方法有一些基本参数例如 `way`，`shot` 或者 `device id`，这样的参数是经常需要改动的。LibFewShot 支持在命令行中对一些简单的配置进行更改而不需要修改 yaml 文件。同样的，在训练和测试过程中，很多不同的小样本学习方法的参数是相同的。为了简洁起见，我们将这些相同的参数包装到了一起，放到了 `config/headers` 目录下，这样就能够通过导入的方式简洁地编写自定义方法的 yaml 文件。

以下是 `config/headers` 目录下文件的构成。

- `data.yaml`：定义了训练所使用的数据的相关配置。
- `device.yaml`：定义了训练所使用的 GPU 的相关配置。

- `losses.yaml`: 定义了训练所用的损失的相关配置。
- `misc.yaml`: 定义了训练过程中一些杂项设置。
- `model.yaml`: 定义了模型训练的相关配置。
- `optimizer.yaml`: 定义了训练所使用的优化器的相关配置。

5.2 LibFewShot 中配置文件的设置

以下详细介绍配置文件中每部分代表的信息以及如何编写。将以 DN4 方法的配置给出示例。

5.2.1 数据设置

- `data_root`: 数据集存放的路径
- `image_size`: 输入图像的尺寸
- `use_memory`: 是否使用内存加速读取
- `augment`: 是否使用数据增强
- `augment_times`: `support_set` 使用数据增强/转换的次数。相当于多次扩充 `support set` 数据。
- `augment_times_query`: `query_set` 使用数据增强/转换的次数。相当于多次扩充了 `query set` 数据。

```
data_root: /data/miniImageNet--ravi
image_size: 84
use_memory: False
augment: True
augment_times: 1
augment_times_query: 1
```

5.2.2 模型设置

- `backbone`: 方法所使用的 `backbone` 信息。
 - `name`: 使用的 `backbone` 的名称, 需要与 `LibFewShot` 中实现的 `backbone` 的大小写一致。
 - `kwargs`: `backbone` 初始化时用到的参数, 必须保持名称与代码中的名称一致。
 - * `is_flatten`: 默认 `False`, 如果为 `True`, 则返回 `flatten` 后的特征向量。
 - * `avg_pool`: 默认 `False`, 如果为 `True`, 则返回 `global average pooling` 后的特征向量。
 - * `is_feature`: 默认 `False`, 如果为 `True`, 则返回 `backbone` 中每一个 `block` 的输出。

```
backbone:
  name: Conv64FLeakyReLU
  kwargs:
    is_flatten: False
```

- classifier: 方法所使用的方法信息。
 - name: 使用的方法的名称，需要与 LibFewShot 中实现的方法的名称一致。
 - kwargs: 方法初始化时用到的参数，必须保持名称与代码中的名称一致。

```
classifier:
  name: DN4
  kwargs:
    n_k: 3
```

5.2.3 训练设置

- epoch: 训练的 epoch 数。
- test_epoch: 测试的 epoch 数。
- pretrain_path: 预训练权重地址。训练开始时会检查该设置。如果不为空，将会把目标地址的预训练权重载入到当前训练的 backbone 中。
- resume: 如果设置为 True，将从默认地址中读取训练状态从而支持断点重训。
- way_num: 训练中的 way 的数量。
- shot_num: 训练中的 shot 的数量。
- query_num: 训练中的 query 的数量。
- test_way: 测试中的 way 的数量。如果未指定，将会把 way_num 赋值给 test_way。
- test_shot: 测试中的 shot 的数量。如果未指定，将会把 shot_num 赋值给 test_way。
- test_query: 测试中的 query 的数量。如果未指定，将会把 query_num 赋值给 test_way。
- episode_size: 网络每次训练所使用的任务数量。
- batch_size: pre-training 的方法在 pre-train 时所使用的 batch size。在某些方法中，该属性是无用的。
- train_episode: 训练阶段每个 epoch 的任务数量。
- test_episode: 测试阶段每个 epoch 的任务数量。

```
epoch: 50
test_epoch: 5
```

(续下页)

(接上页)

```

pretrain_path: ~
resume: False

way_num: 5
shot_num: 5
query_num: 15
test_way: ~
test_shot: ~
test_query: ~
episode_size: 1
# batch_size只在pre-train中起效
batch_size: 128
train_episode: 10000
test_episode: 1000

```

5.2.4 优化器设置

- optimizer: 训练阶段使用的优化器信息。
 - name: 优化器名称, 当前仅支持 Pytorch 提供的所有优化器。
 - kwargs: 传入优化器的参数, 名称需要与 Pytorch 优化器所需要的参数名称相同。
 - other: 当前仅支持单独指定方法中的每一部分所使用的学习率, 名称需要与方法中所使用的变量名相同。

```

optimizer:
  name: Adam
  kwargs:
    lr: 0.01
  other:
    emb_func: 0.01
    #演示用, dn4分类时没有可训练参数
    dn4_layer: 0.001

```

- lr_scheduler: 训练时使用的学习率调整策略, 当前仅支持 Pytorch 提供的所有学习率调整策略。
 - name: 学习率调整策略名称。
 - kwargs: 其他 Pytorch 学习率调整策略所需要的参数。

```

lr_scheduler:
  name: StepLR
  kwargs:

```

(续下页)

(接上页)

```
gamma: 0.5
step_size: 10
```

5.2.5 硬件设置

- `device_ids`: 训练可以用到的 `gpu` 的编号, 与 `nvidia-smi` 命令显示的编号相同。
- `n_gpu`: 训练使用并行训练的 `gpu` 个数, 如果为 1 则不适用并行训练。
- `deterministic`: 是否开启 `torch.backends.cudnn.benchmark` 以及 `torch.backends.cudnn.deterministic` 以及是否使训练随机种子确定。
- `seed`: 训练时 `numpy`, `torch`, `cuda` 使用的种子点。

```
device_ids: 0,1,2,3,4,5,6,7
n_gpu: 4
seed: 0
deterministic: False
```

5.2.6 杂项设置

- `log_name`: 如果为空, 即使用自动生成的 `classifier.name-data_root-backbone-way_num-shot_num` 文件目录。
- `log_level`: 训练中日志输出等级。
- `log_interval`: 日志输出间隔的任务数目。
- `result_root`: 结果存放的根目录
- `save_interval`: 权重保存的 `epoch` 间隔
- `save_part`: 方法中需要保存的部分在方法中的变量名称。这些名称的变量会在模型保存时单独对这些变量保存一次。需要保存的部分在 `save_part` 下以列表的形式给出。

```
log_name: ~
log_level: info
log_interval: 100
result_root: ./results
save_interval: 10
save_part:
  - emb_func
  - dn4_layer
```

训练/测试 LibFewShot 中已集成的方法

本节相关代码：

```
config/dn4.yaml  
run_trainer.py  
run_test.py
```

本部分以 DN4 方法为例，介绍如何训练和测试一个已经实现好的方法。

6.1 配置文件

从编写 `.yaml` 配置文件中我们介绍了如何编写配置文件。并且我们将一部分的常用配置集成成了一个文件，因此可以简单地完成 DN4 配置文件的编写。

```
includes:  
  - headers/data.yaml  
  - headers/device.yaml  
  - headers/misc.yaml  
  - headers/optimizer.yaml  
  - backbones/resnet12.yaml  
  - classifiers/DN4.yaml
```

如果有自定义需要，也可以修改对应的 `includes` 下的导入文件中的内容。也可以删除对应的 `includes` 下的导入文件，自行添加对应的值。

6.2 训练

将上一步编写的配置文件命名为 `dn4.yaml`，放到 `config/` 目录下。

修改根目录下的 `run_trainer.py` 文件。

```
config = Config("./config/dn4.yaml").get_config_dict()
```

接着，在 shell 中输入

```
python run_trainer.py
```

即可开启训练过程。

6.3 测试

修改根目录下的 `run_test.py` 文件。

```
import os
from core.config import Config
from core.test import Test

PATH = "./results/DN4-miniImageNet-resnet12-5-5"
VAR_DICT = {
    "test_epoch": 5,
    "device_ids": "4",
    "n_gpu": 1,
    "test_episode": 600,
    "episode_size": 1,
}

def main(rank, config):
    test = Test(rank, config, PATH)
    test.test_loop()

if __name__ == "__main__":
    config = Config(os.path.join(PATH, "config.yaml"), VAR_DICT).get_config_dict()

    if config["n_gpu"] > 1:
        os.environ["CUDA_VISIBLE_DEVICES"] = config["device_ids"]
        torch.multiprocessing.spawn(main, nprocs=config["n_gpu"], args=(config,))
    else:
        main(0, config)
```

在 shell 中运行

```
python run_test.py
```

即可开始测试过程。

当然，上述 run_test.py 中的 VAR_DICT 变量中的值都可以去掉，然后通过 shell 中运行

```
python run_test.py --test_epoch 5 --device_ids 4 --n_gpu 1 --test_episode 600 --  
→episode_size 1
```

来达到同样的效果。

使用数据集

在 `LibFewShot` 中，数据集有固定的格式。我们按照大多数小样本学习设置下的数据集格式进行数据的读取，例如 *miniImageNet* 和 *tieredImageNet*，因此例如 *Caltech-UCSD Birds 200* 等数据集只需从网络上下载并解压就可以使用。如果你想要使用一个新的数据集，并且该数据集的数据形式与以上数据集不同，那么你需要自己动手将其转换成相同的格式。

7.1 数据集格式

与 *miniImageNet* 一样，数据集的格式应该和下面的示例一样：

```
dataset_folder/
├─ images/
│   └─ images_1.jpg
│   └─ ...
│   └─ images_n.jpg
├─ train.csv *
├─ test.csv *
└─ val.csv *
```

所有的训练、验证以及测试图像都需要放置在 `images` 文件夹下，分别使用 `train.csv`、`test.csv` 和 `val.csv` 文件分割数据集。三个文件的格式都类似，需要以下面的格式进行数据的组织：

```
filename      , label
images_m.jpg, class_name_i
```

(续下页)

(接上页)

```
...  
images_n.jpg, class_name_j
```

CSV 文件的表头仅含文件名和类名两列。这里文件名的路径应是 `images` 文件夹下的相对路径，即对于一张绝对路径为 `.../dataset_folder/images/images_1.jpg` 的图像，其 `filename` 字段就需要填写 `images_1.jpg`，同理，对于绝对路径为 `.../dataset_folder/images/class_name_1/images_1.jpg` 的图像，其 `filename` 字段就需要填写 `class_name_1/images_1.jpg`

7.2 配置数据集

当下载好或按照上述格式整理好数据集后，只需要在配置文件中修改 `data_root` 字段即可，注意 LibFewShot 会将数据集文件夹名当作数据集名称打印在 log 上。

Transformations

本节相关代码：

```
core/data/dataloader.py
core/data/collates/contrib/__init__.py
core/data/collates/collate_functions.py
```

在 LFS 中，我们使用一个基础 Transform 的结构，以公平的比较多种方法。该基础的 Transform 结构可分为三段：

```
Resize&Crop + ExtraTransforms + ToTensor&Norm
```

Resize&Crop 部分根据不同的数据集和配置文件设置 (augment 字段) 存在一些差异：

1. 当数据集为训练数据集 (train) 且 config.augment = True 的时候，使用：

```
from torchvision import transforms
transforms.Resize((96, 96)) # 当 config.image_size 为 224 时，该项为 256
transforms.RandomCrop((84, 84)) # 当 config.image_size 为 224 时，该项为 224
```

2. 其他情况下使用：

```
from torchvision import transforms
transforms.Resize((96, 96)) # 当 config.image_size 为 224 时，该项为 256
transforms.CenterCrop((84, 84)) # 当 config.image_size 为 224 时，该项为 224
```

另外，你可能注意到在

ToTensor & Norm 部分使用同一组均值和方差，你可以根据数据集特性重新设置该值：

```
MEAN = [120.39586422 / 255.0, 115.59361427 / 255.0, 104.54012653 / 255.0]  
STD = [70.68188272 / 255.0, 68.27635443 / 255.0, 72.54505529 / 255.0]
```

增加一个新的 Backbone

本节相关代码：

```
core/model/backbone/*  
config/backbones/*
```

如果想在 LibFewShot 中添加一个新的 backbone，可以将所有与 backbone 有关的文件放到 core/model/backbone/目录下，例如添加 ResNet 网络到 LibFewShot 中，需要将代码写入 core/model/backbone/resnet.py 中，并且在 resnet.py 中提供一个能够生成 backbone 的 class 或者是 function。例如 resnet.py 文件：

```
...  
  
class ResNet(nn.Module):  
    def __init__(self, ...):  
...  
  
def ResNet18():  
    model = ResNet(BasicBlock, [2, 2, 2, 2], **kwargs)  
    return model
```

之后为了能够从 backbone 包中调用到 ResNet18 这个 function，需要修改 core/model/backbone/__init__.py 文件，添加如下一行代码

```
...  
  
from resnet import ResNet18
```

这样一个新的 backbone 就添加完成了。

这个新加入的 backbone 和其他的 backbone 是同样的使用方式。举个例子，要将 ResNet18 替换为 DN4 的 backbone，只需要在 config/dn4.yaml 中将修改 backbone 字段如下：

```
# arch info  
backbone:  
  name: resnet18  
  kwargs:  
    avg_pool: False  
    is_flatten: False
```

即可完成替换。

CHAPTER 10

增加一个新的分类器

本节相关代码：

```
core/model/abstract_model.py
core/model/meta/*
core/model/metric/*
core/model/pretrain/*
```

我们需要从论文中分类的三种方法，即 *matric based*，*meta learning*，以及 *fine tuning*，从每种方法中选出一个代表性的方法，描述如何添加这一类别的新的方法。

不过在此之前，需要先了解一下所有分类方法共同的父类 `abstract_model`。

```
class AbstractModel(nn.Module):
    def __init__(self, ...)
        # base info

    @abstractmethod
    def set_forward(self,):
        # inference phase
        pass

    @abstractmethod
    def set_forward_loss(self,):
        # training phase
        pass
```

(续下页)

(接上页)

```

def forward(self, x):
    out = self.emb_func(x)
    return out

def train(self,):
    # override super's function

def eval(self,):
    # override super's function

def _init_network(self,):
    # init all layers

def _generate_local_targets(self,):
    # formate the few shot labels

def split_by_episode(self,):
    # split batch by way, shot and query

def reset_base_info(self,):
    # change way, shot and query

```

- `__init__`: 初始化函数, 用于初始化一些小样本学习中常用的如 `way`, `shot`, `query` 这样的参数设置。
- `set_forward`: 用于推理阶段调用, 返回分类输出以及准确率。
- `set_forward_loss`: 用于训练阶段调用, 返回分类输出、准确率以及前向损失。
- `forward`: 重写 `pytorch` 的 `Module` 中的 `forward` 函数, 返回 `backbone` 的输出。
- `train`: 重写 `pytorch` 的 `Module` 中的 `train` 函数, 用于解除 `BatchNorm` 层的参数固定。
- `eval`: 重写 `pytorch` 的 `Module` 中的 `eval` 函数, 用于固定 `BatchNorm` 层的参数。
- `_init_network`: 用于初始化所有网络。
- `_generate_local_targets`: 用于生成小样本学习的任务中所使用的 `target`。
- `split_by_episode`: 将输入按照 `episode_size`, `way`, `shot`, `query` 切分好便于后续处理。提供了几种切分方式。
- `reset_base_info`: 改变小样本学习的 `way`, `shot`, `query` 等设置。

其中, 添加新的方法必须要重写 `set_forward` 以及 `set_forward_loss` 这两个函数, 其他的函数都可以根据所实现方法的需要来调用。

注意, 为了新添加的方法能够通过反射机制调用到, 需要在对应方法类型的目录下的 `__init__.py` 文件中加上一行:

```
from NewMethodFileName import *
```

10.1 metric based

接下来将以 DN4 为例，描述如何在 LibFewShot 中添加一个新的 metric based classifier。

metric based 方法有一个共同的父类 MetricModel，继承了 AbstractModel。

```
class MetricModel(AbstractModel):
    def __init__(self,):
        ...

    @abstractmethod
    def set_forward(self, *args, **kwargs):
        pass

    @abstractmethod
    def set_forward_loss(self, *args, **kwargs):
        pass

    def forward(self, x):
        out = self.emb_func(x)
        return out
```

由于 metric based 方法的 pipeline 的方法大多比较简单，因此只是继承了 abstract_model，并没有做其他修改。

建立模型

首先创建 DN4 的模型类，在 core/model/metric/下添加 dn4.py 文件：（这部分代码与源码略有不同）

```
class DN4(MetricModel):
    def __init__(self, way_num, shot_num, query_num, emb_func, device, n_k=3):
        # base info
        super(DN4Layer, self).__init__()
        self.way_num = way_num
        self.shot_num = shot_num
        self.query_num = query_num
        self.n_k = n_k
        self.loss_func = nn.CrossEntropyLoss()

    def set_forward(self, batch):
        # inference phase
        """
```

(续下页)

(接上页)

```

        :param batch: (images, labels)
        :param batch.images: shape: [episodeSize*way*(shot*augment_
→times+query*augment_times_query), C, H, W]
        :param batch.labels: shape: [episodeSize*way*(shot*augment_
→times+query*augment_times_query), ]
        :return: net output and accuracy
        """
        image, global_target = batch
        image = image.to(self.device)
        episode_size = image.size(0) // (
            self.way_num * (self.shot_num + self.query_num)
        )
        feat = self.emb_func(image)
        support_feat, query_feat, support_target, query_target = self.split_by_
→episode(
            feat, mode=2
        )

        t, wq, c, h, w = query_feat.size()
        _, ws, _, _, _ = support_feat.size()

        # t, wq, c, hw -> t, wq, hw, c -> t, wq, 1, hw, c
        query_feat = query_feat.view(
            t, self.way_num * self.query_num, c, h * w
        ).permute(0, 1, 3, 2)
        query_feat = F.normalize(query_feat, p=2, dim=2).unsqueeze(2)

        # t, ws, c, h, w -> t, w, s, c, hw -> t, 1, w, c, shw
        support_feat = (
            support_feat.view(t, self.way_num, self.shot_num, c, h * w)
            .permute(0, 1, 3, 2, 4)
            .contiguous()
            .view(t, self.way_num, c, self.shot_num * h * w)
        )
        support_feat = F.normalize(support_feat, p=2, dim=2).unsqueeze(1)

        # t, wq, w, hw, shw -> t, wq, w, hw, n_k -> t, wq, w
        relation = torch.matmul(query_feat, support_feat)
        topk_value, _ = torch.topk(relation, self.n_k, dim=-1)
        score = torch.sum(topk_value, dim=[3, 4])

        output = score.view(episode_size * self.way_num * self.query_num, self.way_
→num)

```

(续下页)

(接上页)

```

    acc = accuracy(output, query_target)

    return output, acc

def set_forward_loss(self, batch):
    # training phase
    """
    :param batch: (images, labels)
    :param batch.images: shape: [episodeSize*way*(shot*augment_
→times+query*augment_times_query), C, H, W]
    :param batch.labels: shape: [episodeSize*way*(shot*augment_
→times+query*augment_times_query), ]
    :return: net output, accuracy and train loss
    """
    image, global_target = batch
    image = image.to(self.device)
    episode_size = image.size(0) // (
        self.way_num * (self.shot_num + self.query_num)
    )
    emb = self.emb_func(image)
    support_feat, query_feat, support_target, query_target = self.split_by_
→episode(
        emb, mode=2
    )

    t, wq, c, h, w = query_feat.size()
    _, ws, _, _, _ = support_feat.size()

    # t, wq, c, hw -> t, wq, hw, c -> t, wq, 1, hw, c
    query_feat = query_feat.view(
        t, self.way_num * self.query_num, c, h * w
    ).permute(0, 1, 3, 2)
    query_feat = F.normalize(query_feat, p=2, dim=2).unsqueeze(2)

    # t, ws, c, h, w -> t, w, s, c, hw -> t, 1, w, c, shw
    support_feat = (
        support_feat.view(t, self.way_num, self.shot_num, c, h * w)
        .permute(0, 1, 3, 2, 4)
        .contiguous()
        .view(t, self.way_num, c, self.shot_num * h * w)
    )
    support_feat = F.normalize(support_feat, p=2, dim=2).unsqueeze(1)

```

(续下页)

(接上页)

```

    # t, wq, w, hw, shw -> t, wq, w, hw, n_k -> t, wq, w
    relation = torch.matmul(query_feat, support_feat)
    topk_value, _ = torch.topk(relation, self.n_k, dim=-1)
    score = torch.sum(topk_value, dim=[3, 4])

    output = score.view(episode_size * self.way_num * self.query_num, self.way_
↪ num)

    loss = self.loss_func(output, query_target)
    acc = accuracy(output, query_target)

    return output, acc, loss

```

在 `__init__` 中，对分类器可能用到的小样本学习的基本设置进行了初始化，还传入了 DN4 方法的一个超参数 `n_k`。

在 `set_forward` 与 `set_forward_loss` 中，需要注意的是 19-27, 65-73 行，这部分代码对输入的 `batch` 进行处理，提取特征，最后切分为小样本学习中需要使用的 `support set` 和 `query set` 的特征。具体来说，为了最大化利用计算资源，我们将所有图像同时经过 `backbone`，之后对特征向量进行 `support set`, `query set` 的切分。29-50, 75-96 行为 DN4 方法的计算过程。最终 `set_forward` 的输出为 `$output.shape:[episode_size*way*query,way]`, `acc:float`，`set_forward_loss` 的输出为 `$output.shape:[episode_size*way*query,way]`, `acc:float`, `loss:tensor`。其中 `output` 需要用户根据方法进行生成，`acc` 可以调用 LibFewShot 提供的 `accuracy` 函数，输入 `output`, `target` 就可以得到分类准确率。而 `loss` 可以使用用户在方法开始时初始化的损失函数，在 `set_forward_loss` 中使用来得到分类损失。

`metric` 方法中只需要根据自己设计的方法，将输入处理为对应的形式就可以开始训练了。

10.2 meta learning

接下来将以 MAML 为例，描述如何在 LibFewShot 中添加一个新的 meta learning classifier。

meta learning 方法有一个共同的父类 `MetaModel`，继承了 `AbstractModel`。

```

class MetaModel(AbstractModel):
    def __init__(self,):
        super(MetaModel, self).__init__(init_type, ModelType.META, **kwargs)

    @abstractmethod
    def set_forward(self, *args, **kwargs):
        pass

    @abstractmethod
    def set_forward_loss(self, *args, **kwargs):
        pass

```

(续下页)

(接上页)

```

def forward(self, x):
    out = self.emb_func(x)
    return out

@abstractmethod
def set_forward_adaptation(self, *args, **kwargs):
    pass

def sub_optimizer(self, parameters, config):
    kwargs = dict()

    if config["kwargs"] is not None:
        kwargs.update(config["kwargs"])
    return getattr(torch.optim, config["name"])(parameters, **kwargs)

```

meta-learning 方法加入了两个新函数, `set_forward_adaptation` 和 `sub_optimizer`。`set_forward_adaptation` 是微调网络阶段的分类过程所采用的逻辑, 而 `sub_optimizer` 用于在微调时提供新的局部优化器。

建立模型

首先创建 MAML 的模型类, 在 `core/model/meta/` 下添加 `maml.py` 文件: (这部分代码与源码略有不同)

```

from ..backbone.utils import convert_maml_module

class MAML(MetaModel):
    def __init__(self, inner_param, feat_dim, **kwargs):
        super(MAML, self).__init__(**kwargs)
        self.loss_func = nn.CrossEntropyLoss()
        self.classifier = nn.Sequential(nn.Linear(feat_dim, self.way_num))
        self.inner_param = inner_param

        convert_maml_module(self)

    def forward_output(self, x):
        """
        :param x: feature vectors, shape: [batch, C]
        :return: probability of classification
        """
        out1 = self.emb_func(x)
        out2 = self.classifier(out1)
        return out2

```

(续下页)

(接上页)

```

def set_forward(self, batch):
    """
    :param batch: (images, labels)
    :param batch.images: shape: [episodeSize*way*(shot*augment_
↪times+query*augment_times_query), C, H, W]
    :param batch.labels: shape: [episodeSize*way*(shot*augment_
↪times+query*augment_times_query), ]
    :return: net output, accuracy and train loss
    """
    image, global_target = batch # unused global_target
    image = image.to(self.device)
    support_image, query_image, support_target, query_target = self.split_by_
↪episode(
    image, mode=2
    )
    episode_size, _, c, h, w = support_image.size()

    output_list = []
    for i in range(episode_size):
        episode_support_image = support_image[i].contiguous().reshape(-1, c, h, w)
        episode_query_image = query_image[i].contiguous().reshape(-1, c, h, w)
        episode_support_target = support_target[i].reshape(-1)
        self.set_forward_adaptation(episode_support_image, episode_support_target)

        output = self.forward_output(episode_query_image)

        output_list.append(output)

    output = torch.cat(output_list, dim=0)
    acc = accuracy(output, query_target.contiguous().view(-1))
    return output, acc

def set_forward_loss(self, batch):
    """
    :param batch: (images, labels)
    :param batch.images: shape: [episodeSize*way*(shot*augment_
↪times+query*augment_times_query), C, H, W]
    :param batch.labels: shape: [episodeSize*way*(shot*augment_
↪times+query*augment_times_query), ]
    :return: net output, accuracy and train loss
    """
    image, global_target = batch # unused global_target
    image = image.to(self.device)

```

(续下页)

(接上页)

```

        support_image, query_image, support_target, query_target = self.split_by_
→episode(
            image, mode=2
        )
    episode_size, _, c, h, w = support_image.size()

    output_list = []
    for i in range(episode_size):
        episode_support_image = support_image[i].contiguous().reshape(-1, c, h, w)
        episode_query_image = query_image[i].contiguous().reshape(-1, c, h, w)
        episode_support_target = support_target[i].reshape(-1)
        self.set_forward_adaptation(episode_support_image, episode_support_target)

        output = self.forward_output(episode_query_image)

        output_list.append(output)

    output = torch.cat(output_list, dim=0)
    loss = self.loss_func(output, query_target.contiguous().view(-1))
    acc = accuracy(output, query_target.contiguous().view(-1))
    return output, acc, loss

def set_forward_adaptation(self, support_set, support_target):
    lr = self.inner_param["lr"]
    fast_parameters = list(self.parameters())
    for parameter in self.parameters():
        parameter.fast = None

    self.emb_func.train()
    self.classifier.train()
    for i in range(self.inner_param["iter"]):
        output = self.forward_output(support_set)
        loss = self.loss_func(output, support_target)
        grad = torch.autograd.grad(loss, fast_parameters, create_graph=True)
        fast_parameters = []

        for k, weight in enumerate(self.parameters()):
            if weight.fast is None:
                weight.fast = weight - lr * grad[k]
            else:
                weight.fast = weight.fast - lr * grad[k]
        fast_parameters.append(weight.fast)

```

MAML 中最重要的有两部分。第一部分是第 10 行的 `convert_maml_module` 函数，用于将网络中的所有

层转换为 MAML 格式的层以便于参数更新。另一部分是 `set_forward_adaption` 函数，用于更新网络的快参数。MAML 是一种常用的 meta learning 方法，因此我们使用 MAML 作为例子来展示如何添加一个 meta learning 方法到 LibFewShot 库中。

10.3 fine tuning

接下来将以 Baseline 为例，描述如何在 LibFewShot 中添加一个新的 fine-tuning classifier。fine-tuning 方法有一个共同的父类 `FinetuningModel`，继承了 `AbstractModel`。

```
class FinetuningModel(AbstractModel):
    def __init__(self,):
        super(FinetuningModel, self).__init__()
        # ...

    @abstractmethod
    def set_forward(self, *args, **kwargs):
        pass

    @abstractmethod
    def set_forward_loss(self, *args, **kwargs):
        pass

    def forward(self, x):
        out = self.emb_func(x)
        return out

    @abstractmethod
    def set_forward_adaptation(self, *args, **kwargs):
        pass

    def sub_optimizer(self, model, config):
        kwargs = dict()
        if config["kwargs"] is not None:
            kwargs.update(config["kwargs"])
        return getattr(torch.optim, config["name"])(model.parameters(), **kwargs)
```

fine-tuning 方法训练时的目标是训练出一个好的特征抽取器，在测试时使用小样本学习的设置，通过 support set 来对模型进行微调。也有的方法是在训练完毕特征抽取器后，再使用小样本学习的训练设置来进行整个模型的微调。为了与 meta learning 的方法统一，我们添加了一个 `set_forward_adaptation` 抽象函数，用于处理在测试时的前向过程。另外，由于有一些 fine-tuning 方法的测试过程中，也需要训练分类器，因此，添加了一个 `sub_optimizer` 方法，传入需要优化的参数以及优化的配置参数，返回优化器，用以方便调用。

建立模型

首先创建 Baseline 的模型类, 在 `core/model/finetuning/` 下添加 `baseline.py` 文件: (这部分代码与源码略有不同)

```
class FinetuningModel(AbstractModel):
    def __init__(self,):
        super(FinetuningModel, self).__init__()
        # ...

    @abstractmethod
    def set_forward(self, *args, **kwargs):
        pass

    @abstractmethod
    def set_forward_loss(self, *args, **kwargs):
        pass

    def forward(self, x):
        out = self.emb_func(x)
        return out

    @abstractmethod
    def set_forward_adaptation(self, *args, **kwargs):
        pass

    def sub_optimizer(self, model, config):
        kwargs = dict()
        if config["kwargs"] is not None:
            kwargs.update(config["kwargs"])
        return getattr(torch.optim, config["name"])(model.parameters(), **kwargs)
```

`set_forward_loss` 方法与经典有监督分类方法相同, 而 `set_forward` 方法与 meta learning 方法相同。`set_forward_loss` 函数的内容是测试阶段的主要过程。由 `backbone` 从 support set 中提取的特征被用于训练一个分类器, 而从 query set 中提取的特征被该分类器进行分类。

CHAPTER 11

Model Zoo

Coming Soon. 这里将会包含方法和精度，对每个方法提供 `checkpoint` 下载和配置文件下载，乃至代码解析。

可自由贡献 classifiers, backbones, functions 及其他任何改进。

12.1 增加一个方法/特性或修复一个错误

我们建议参考如下方法：

1. fork main 分支的最新 LibFewShot 代码
2. checkout 一个新的分支，分支的名字应该能够直观表现本次 contribution 的主要内容，如 add-method-demo 或者 fix-doc-typos
3. commit
4. create a PR

注意，如果你添加了一个方法，你需要做的是

1. 测试该方法的表现是否正常
2. 提供你复现该方法使用的 config 文件，以及对应的在 miniImageNet 上的 1-shot 和 5-shot 精度。以及，您最好能提供：
3. 在其它数据集（如 *tieredImageNet*）上的 1-shot 和 5-shot 精度
4. 对应的 model_best.pth 文件（以 zip 文件格式）我们会在 README 和其它显眼的地方感谢您的贡献。

12.2 使用 pre-commit 检查代码

在提交你的代码之前，你的代码需能够通过 `black` 的格式化以及 `flake8`，我们使用 `pre-commit` 进行测试和自动修订：

1. 首先安装 `pre-commit`

```
cd <path-to-LibFewShot>
pip install pre-commit
```

1. `run pre-commit install`
2. `run pre-commit run --all-files`
3. 根据 `pre-commit` 给出的 `warning` 修改代码

12.3 PR style

你的 PR 请求标题应该如下：

```
[Method] XXXX XXXX
# 或者
[Feature] XXXX XXXX
# 或者
[FIX] XXXX XXXX
```

PR 请求的正文内容应该使用英文/中文简要描述本次 PR 的主要内容。

CHAPTER 13

Indices and tables

- `genindex`
- `modindex`
- `search`