FGVClib

发布 *0.1.0*

yyq

2023 年 01 月 14 日

Contents

CHAPTER 1

准备工作

在这个部分，我们将展示如何搭建 Pytorch 环境。FGVCLib 工作于 Linux 系统中，并且需要 Python 3.7+, CUDA 10.0+, Pytorch。

```
如果你使用过 Pytroch 并且已经下载好 Pytorch，可以越过这一部分，并跳转到 [下一个部分](
↪#installation)。如果你没有 Pytorch，可以遵循下面的步骤准备环境。
```

Step 0. 从官网下载并安装 Anaconda。

Step 1. 创建一个虚拟环境并且激活它。

```
conda create -n fgvclib python=3.7
conda activate fgvclib
```

Step 2. 从官网上下载并安装 Pytorch。

如果你有 GPU:

```
conda install pytorch torchvision -c python
```

CHAPTER 2

安装

2.1 最佳示例

我们建议开发者遵循我们的最佳示例来安装 FGVCLib，FGVCLib 需要一些要求和安装包。

Step 0. 安装 FGVClib

```
git clone https://github.com/dongliangchang/Fine-grained-Visual-Analysis-Library.git
cd Fine-grained-Visual-Analysis-Library.git
```

Step 1. 安装需要的库

```
pip install -r requirements.txt
```

2.2 问题解答

```
你在安装的过程中可能会遇到一些问题，主要问题是关于安装'fiftyonr'，如果你在安装'fiftyone' 时遇到了问题，你可以参考下面的方法。
```

如果你的 Ubuntu 版本 >=18.04, 你可以执行下面的命令

```
pip install fiftyone
```

如果你的 Ubuntu 版本 < 18.04，你可以执行下面的命令

```
pip install fiftyone-db-ubuntu1604
```

如果你遇到了这样的报错”error while loading shared libraries: libcurl.so.4: cannot open shared object file: No such file”，请检查是否有 `curl`，如果你没有 `curl`, 请执行下面的命令

```
sudo apt-get update
sudo apt-get install libcurl4-openssl-dev
sudo apt-get install curl
```

CHAPTER 3

Benchmark and model zoo

3.1 常规设置

- 所有的模型我们都是在 `CUB_200_2011_train` 上进行训练并且在 `CUB_200_2011_test` 上进行测试的
- 为了与其他代码库进行公平的比较，我们将 GPU 内存报告为所有 8 个 GPU 的’ torch.cuda.max_memory_allocate() ‘的最大值。注意，这个值通常小于’ nvidia-smi ‘显示的值。
- 所有的预训练的 backbone 都是来自于 PyTorch model zoo。

3.2 Backbone 模型

下面列出了 FGVCLib 中常用的骨干模型的详细表:

3.3 方法

3.3.1 MCL

更多细节请参考 MCL

3.3.2 PMG

更多细节请参考 PMG

CHAPTER 4

1：在标准数据集上训练已有模型

为了评估模型的准确性，人们通常在一些标准数据集上测试模型。FGVCLib 支持包括 CUB_200_2011 在内的公共数据集。本节将展示如何在受支持的数据集上测试现有模型。

基本步骤如下：1. 准备标准数据集 2. 准备配置文件 3. 在标准数据集上对模型进行训练、测试和预测

4.1 已有模型

我们提供了多种已有的方法，它们分别是：`baseline_resnet50`, `MCL`, `PMG`, `PMG_v2`, `API-Net`, `CAL`, `PIM`, `TransFG`。

今后我们将会继续复现更多新的方法并将它们更新至 FGVCLib 中。

4.2 准备标准数据集

我们提供了 CUB-200-2011，我们将数据集分为训练文件夹和测试文件夹。

例如，CUB-200-2011 数据集：

```
-/birds/train
                └── 001.Black_footed_Albatross
                        └── Black_Footed_Albatross_0001_796111.jpg
                        └── ...
                └── 002.Laysan_Albatross
```

(下页继续)

(续上页)

```
                └── 003.Sooty_Albatross
                └── ...
  -/birds/test
            └── ...
```

如果你已经准备好数据集了，你可以跳过下面的第一步。

第一步：打开”/fgvclib/datasets/cub.py”，将 `class CUB_200_2011: __init__ : download:bool=False` 修改为 `class CUB_200_2011: __init__ : download:bool=True`

```
参数“download”控制是否下载数据集。通过设置 download=True 直接下载 CUB 数据集。默认为 False。
```

第二步：打开”/configs/xxx/xxx.yml”，将 `DATASET-ROOT` 替换为你自己的路径。

4.3 训练模型

第一步：打开”/configs/xxx/xxx.yml”，将 `WEIGHT-SAVE_DIR` 替换为你自己的路径。**第二步**：打开”/configs/xxx/xxx.yml”，检查模型的配置，你可以自己修改这些配置。**第三步**：执行主程序 main.py 进行训练

```
python main.py --config configs/resnet/resnet50.yml
```

这里存在几类参数控制着程序的运行配置：

- ‘–config’：配置文件路径。
- ‘–task’：默认为 **train**。
- ‘–device’：两种选择是 cuda 和 cpu。默认为 **cuda**。
- ‘–world-size’：分布式进程的数量。默认值是 4。
- ‘–dist-url’:Url 用于设置分布式培训。默认值是’env://’。

如果你想在 cpu 上运行它，你应该执行下面的：

```
python main.py --config configs/resnet/resnet50.yml --device cpu
```

4.4 测试模型

```
python main.py --config configs/resnet/resnet50.yml --task predict
```

CHAPTER 5

2: 在标准数据集上训练自定义模型

我们将模型分解为骨干、编码器、分类器等基本结构，然后将它们组合起来构建完整的方法。在 FGVCLib 中，我们提供了基本结构并复现了最先进的模型。我们致力于为您提供自定义结构，使用分解后的模块重新组装成新的模型。

主要的步骤如下：1. 准备数据集

```
2. 准备自定义模型

3. 准备配置文件

4. 在标准数据集上进行训练、测试和推理
```

5.1 准备数据集

你需要在配置文件中修改对应的数据集路径。你需要将数据集分成训练集和测试集两个文件夹。

例如，CUB-200-2011 数据集：

```
 -/birds/train
                └── 001.Black_footed_Albatross
                        └── Black_Footed_Albatross_0001_796111.jpg
                        └── ...
                └── 002.Laysan_Albatross
                └── 003.Sooty_Albatross
```

(下页继续)

(续上页)

```
                └── ...
  -/birds/test
            └── ...
```

5.2 准备自定义模型

第二步时使用已有的模块和新的模块构建自定义模型，假设我们想添加一个新的编码器 `xxx`。

5.2.1 1. 定义一个新的编码器（以 xxx 为例）

首先建立新文件 `fgvclib/model/encoders/xxx.py`。

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

from fgvclib.models.encoders import encoder

class xxx(nn.Module):
    def __init__(self)
        pass

    def forward(self,inputs):
        pass

def xxx(cfg:dict):
    pass
```

5.2.2 2. 导入模块

你可以在其他需要的地方导入该编码器

```
from .xxx import xxx
```

5.3 准备配置文件

第三步是为你自己的训练设置准备一个配置文件。在”configs/xxx.yml”中，你可以根据已有的配置文件，新建立配置。

以新编码器 xxx 为例：

```
MODEL:
  NAME: "ResNet50"
  CLASS_NUM: 200
  CRITERIONS:
    - name: "cross_entropy_loss"
      args: []
      w: 1.0
  BACKBONE:
    NAME: "resnet50"
    ARGS:
      - pretrained: True
      - del_keys: []
  ENCODER:
    NAME: "xxx"
  NECKS:
    NAME: ~
  HEADS:
    NAME: "classifier_1fc"
    ARGS:
      - in_dim:
        - 2048
```

5.4 训练新模型

为了能够使用新增配置来训练模型，你可以运行如下命令：

```
python main.py --configs/xxx.yml --task train
```

5.5 测试新模型

为了能够测试训练好的模型，你可以运行如下命令：

```
python main.py --configs/xxx.yml --task predict
```

CHAPTER 6

教程 1: 学习接口文件

在” fgvclib/api” 这个文件夹下，我们为 fgvclib 设置了各类 api 接口。这里有四种类型的 api 接口：`build.py`, `evluate_model.py`, `save_model.py`, 和 `update_model.py`。

“fgvclib/apis/build.py”：提供了各种用于快速构建训练系统或评估系统的 api；

“fgvc/apis/evluate_model.py”：提供了用于评估 FGVC 算法的 api；

“fgvclib/apis/save_model.py”：提供了各种用于保存模型的 api；

“fgvclib/apis/update_model”：提供了各种用于更新模型和记录损失的 api。

6.1 模型构建

build_model: 根据全局配置构建一个 FGVC 模型。

- 参数:

  `model_cfg (CfgNode)`: 根配置的模型配置节点

- 返回值:

  `nn.Module`: FGVC 模型

build_logger: 根据配置构建日志对象。

- 参数:

  `cfg (CfgNode)`: 根配置节点

- 返回值:

  `Logger`: 日志对象 **build_transforms**: 根据配置为训练或测试数据集构建转换

- 参数:

  `transforms_cfg (CfgNode)`: 根配置节点

- 返回值:

  `transforms.Compose`: Pytorch 中的 transforms.Compose 对象

build_dataset: 为训练过程或评估过程构建数据加载器

- 参数:

  `root (str)`: 数据集的目录 `cfg (CfgNode)`: 根配置节点

- 返回值:

  `DataLoader`: Pytorch 数据加载器

build_optimizer: 为训练过程构建优化器

- 参数:

  `optim_cfg (CfgNode)`: 根配置节点的优化配置节点

- 返回值:

  `Optimizer`: Pytorch 优化器

build_criterion : 为训练过程构建损失函数

- 参数:

  `criterion_cfg` (CfgNode): 根配置节点的标准配置节点

- 返回值:

  `nn.Module`: 损失函数

build_interpreter: 为训练过程构建一个解释器

- 参数:

  `cfg (CfgNode)`: 根配置节点

- 返回值:

  `Interpreter`: 一个解释器

build_metrics: 为评估过程构建度量标准

- 参数:

  `metrics_cfg (CfgNode)`: 根配置节点的度量标准配置节点

- 返回值:

  `t.List[NamedMetric]`: NamedMetric 列表

6.2 模型评估

evaluate_model: 对 FGVC 模型进行评估

- 参数:

  `model (nn.Module)`: FGVC 模型 `p_bar (iterable)`: 提供测试数据的迭代器 `metrics (List[NamedMetric])`: 指标的列表 `use_cuda (boolean, optional)`: 是否使用 gpu

- 返回值:

  `dict`: 结果的字典

6.3 模型保存

save_model: 保存被训练的 FGVC 模型

- 参数:

  `cfg (CfgNode)`: 根配置节点 `model (nn.Module)`: FGVC 模型 `logger (Logger)`: 日志对象

6.4 模型更新

update_model: 更新 FGVC 模型并且记录损失

- 参数:

  `model (nn.Module)`: FGVC 模型 `optimizer (Optimizer)`: 日志对象 `pbar (Iterable)`: 提供训练数据的可迭代对象 `strategy (string)`: 更新的策略 `use_cuda (boolean)`: 是否使用 GPU 训练模型 `logger (Logger)`: 日志对象

6.5 API 的应用

当你进行算法设计时，你需要使用 `from fgvclib.apis import *` 导入上述这些 api 去调用这些接口。你可以直接使用以下的函数: `build_logger`, `build_criterion`, `build_model`, `build_metrics`, `build_transforms`, `build_dataset`, `build_optimizer`, `update_model`, `evaluate_model`, `save_model`, `build_interpreter`

- 应用举例：建立模型

```
import os
import torch

from fgvclib.apis import *
from fgvclib.configs import FGVCConfig

model = build_model(cfg.MODEL)
weight_path = os.path.join(cfg.WEIGHT.SAVE_DIR, cfg.WEIGHT.NAME)
assert os.path.exists(weight_path), f"The resume weight {cfg.RESUME_WEIGHT} dosn't␣
↪exists."
state_dict = torch.load(weight_path, map_location="cpu")
model.load_state_dict(state_dict=state_dict)

if cfg.USE_CUDA:
    assert torch.cuda.is_available(), f"Cuda is not available."
    model = torch.nn.DataParallel(model)

transforms = build_transforms(cfg.TRANSFORMS.TEST)
loader = build_dataset(root=os.path.join(cfg.DATASETS.ROOT, 'test'), cfg=cfg.DATASETS.
↪TEST, transforms=transforms)

interpreter = build_interpreter(model, cfg)
voxel = VOXEL(dataset=loader.dataset, name=cfg.FIFTYONE.NAME, interpreter=interpreter)
voxel.predict(model, transforms, 10, cfg.MODEL.NAME)
voxel.launch()
```

CHAPTER 7

教程 2: 学习配置文件

在这个文件夹中 “fgvclib/configs” 我们列出了关于 FGVCLib 的相关配置。在实验中我们对配置进行了模块化，建立了 `FGVCConfig` 类去加载喝存储相关的参数。你可以使用 `FGVCConfig` 加载相关的配置。

7.1 配置文件结构

在这个文件夹下” fgvclib/configs/config.py” 有四种基本组件，`__init__`,`get_cfg`,`load`,`stringfy`.

我们为 FGVC 方法设置了参数，你可以在 `config.py` 查找参数或修改参数设置。

以下是关于基础的参数说明：

```
# Name of Project
self.cfg.PROJ_NAME = "FGVC"

# Name of experiment
self.cfg.EXP_NAME = None

# Resume last train
self.cfg.RESUME_WEIGHT = None

# Directory of trained weight
self.cfg.WEIGHT = CN()
self.cfg.WEIGHT.NAME = None
self.cfg.WEIGHT.SAVE_DIR = "./checkpoints/"
```

(下页继续)

(续上页)

```
# Use cuda
self.cfg.USE_CUDA = True

# Logger
self.cfg.LOGGER = CN()
self.cfg.LOGGER.NAME = "wandb_logger"
self.cfg.LOGGER.FILE_PATH = "./logs/"
self.cfg.LOGGER.PRINT_FRE = 50
```

以下是关于数据集的参数说明：

```
# Datasets and data loader
self.cfg.DATASET = CN()
self.cfg.DATASET.NAME = None
self.cfg.DATASET.ROOT = None
self.cfg.DATASET.TRAIN = CN()
self.cfg.DATASET.TEST = CN()

# train dataset and data loder
self.cfg.DATASET.TRAIN.BATCH_SIZE = 32
self.cfg.DATASET.TRAIN.POSITIVE = 0
self.cfg.DATASET.TRAIN.PIN_MEMORY = True
self.cfg.DATASET.TRAIN.SHUFFLE = True
self.cfg.DATASET.TRAIN.NUM_WORKERS = 0

# test dataset and data loder
self.cfg.DATASET.TEST.BATCH_SIZE = 32
self.cfg.DATASET.TEST.POSITIVE = 0
self.cfg.DATASET.TEST.PIN_MEMORY = False
self.cfg.DATASET.TEST.SHUFFLE = False
self.cfg.DATASET.TEST.NUM_WORKERS = 0
```

以下是关于模型的参数说明：

```
# Model architecture
self.cfg.MODEL = CN()
self.cfg.MODEL.NAME = None
self.cfg.MODEL.CLASS_NUM = None
self.cfg.MODEL.CRITERIONS = None

# Standard modulars of each model
self.cfg.MODEL.BACKBONE = CN()
self.cfg.MODEL.ENCODING = CN()
```

(下页继续)

(续上页)

```
    self.cfg.MODEL.NECKS = CN()
    self.cfg.MODEL.HEADS = CN()

    # Setting of backbone
    self.cfg.MODEL.BACKBONE.NAME = None
    self.cfg.MODEL.BACKBONE.ARGS = None

    # Setting of encoding
    self.cfg.MODEL.ENCODING.NAME = None
    self.cfg.MODEL.ENCODING.ARGS = None

    # Setting of neck
    self.cfg.MODEL.NECKS.NAME = None
    self.cfg.MODEL.NECKS.ARGS = None

    # Setting of head
    self.cfg.MODEL.HEADS.NAME = None
    self.cfg.MODEL.HEADS.ARGS = None

    # Transforms
    self.cfg.TRANSFORMS = CN()
    self.cfg.TRANSFORMS.TRAIN = None
    self.cfg.TRANSFORMS.TEST = None

    # Optimizer
    self.cfg.OPTIMIZER = CN()
    self.cfg.OPTIMIZER.NAME = "SGD"
    self.cfg.OPTIMIZER.MOMENTUM = 0.9
    self.cfg.OPTIMIZER.WEIGHT_DECAY = 5e-4
    self.cfg.OPTIMIZER.LR = CN()
    self.cfg.OPTIMIZER.LR.backbone = None
    self.cfg.OPTIMIZER.LR.encoding = None
    self.cfg.OPTIMIZER.LR.necks = None
    self.cfg.OPTIMIZER.LR.heads = None
```

以下是关于训练过程的参数说明：

```
    # Train
    self.cfg.ITERATION_NUM = None
    self.cfg.EPOCH_NUM = None
    self.cfg.START_EPOCH = None
    self.cfg.UPDATE_STRATEGY = None

    # Validation
```

(下页继续)

(续上页)

```
  self.cfg.PER_ITERATION = None
  self.cfg.PER_EPOCH = None
  self.cfg.METRICS = None

  # Inference
  self.cfg.FIFTYONE = CN()
  self.cfg.FIFTYONE.NAME = "BirdsTest"
  self.cfg.FIFTYONE.STORE = True

  self.cfg.INTERPRETER = CN()
  self.cfg.INTERPRETER.NAME = "cam"
  self.cfg.INTERPRETER.METHOD = "gradcam"
  self.cfg.INTERPRETER.TARGET_LAYERS = []
```

7.2 关于配置的举例

在这个程序中 `main.py`, 你可以导入关于配置的文件 `from fgvclib.configs import FGVCConfig`, 并且使用它加载模型配置。

```
import os
import torch

from fgvclib.configs import FGVCConfig

# load config
    config = FGVCConfig()
    config.load(args.config)
    cfg = config.cfg
    print(cfg)
```

CHAPTER 8

教程 3: 学习标准文件

在”fgvclib/criterions”这个文件夹下，我们为 fgvclib 提供了不同了损失函数。

我们提供了四个损失函数: `cross_entropy_loss`, `binary_cross_entropy_loss`, `mean_square_error_loss` 和 `mutual_channel_loss`

8.1 基础的损失函数

`cross_entropy_loss`, `binary_cross_entropy_loss`, `mean_square_error_loss` 这三类损失函数是基础的损失函数，在 fgvclib 中，我们从 Pytorch 中调用它们。

“fgvclib/criterions/base_loss.py”中提供了这三类基础的损失函数。

cross_entropy_loss: 构建交叉熵损失函数

- 参数:

  `cfg (CfgNode)`: 配置的根节点

- 返回值:

  `nn.Module`: 损失函数

binary_cross_entropy_loss: 构建二元交叉熵损失函数

- 参数:

  `cfg (CfgNode)`: 配置的根节点

- 返回值:

  nn.Module: 损失函数

mean_square_error_loss: 构建均方差损失函数

- 参数:

  cfg (CfgNode): 配置的根节点

- 返回值:

  nn.Module: 损失函数

8.2 互信道损失函数

“fgvclib/criterions/mutual_channel_loss.py”提供了互信道损失函数, 该方法在”The Devil is in the Channels: Mutual-Channel Loss for Fine-Grained Image Classification” 论文中被提出，关于互信道损失函数的更多细节，参考该篇论文MC-Loss

class MutualChannelLoss: 互信道损失函数类

- 参数:

  height (int): average pooling 的内核大小 cnum (int): 每个类的通道数量 div_weight (float): 多样性部分损失的权重 dis_weight (float): 判别性部分损失的权重

8.3 工具

在” fgvclib/criterions/utils.py” 中，我们设计了一个类：LossItem，两个函数：compute_loss_value 和 detach_loss_value

LossItem: 用于储存训练损失的数据类对象

- 参数:

  name (string): 损失函数名称 value (torch.Tensor): 损失项的值 weight (float, optional): 当前损失项的权重，默认为 1.0

compute_loss_value: 用于储存训练损失的数据类对象

- 参数:

  items (List[LossItem]): 损失项

- 返回值:

  Tensor: 总的损失项的值

detach_loss_value: 从 GPU 分离损失值

- 参数:

  items (List[LossItem]): 损失项

- 返回值:

  Dict: 损失信息字典，key 为损失名称，对应的值为损失值

8.4 Criterion 标准的应用

8.4.1 为训练过程建立损失函数

在”fgvclib/apis/build.py”中，使用”fgvclib.criterions”去为训练过程构建损失函数，你可以从这四类损失函数中选择 cross_entropy_loss，cross_entropy_loss，mean_square_error_loss and mutual_channel_loss 替换损失函数名称 criterion_cfg['name']

```
from fgvclib.criterions import get_criterion

def build_criterion(criterion_cfg: CfgNode) -> nn.Module:
    criterion_builder = get_criterion(criterion_cfg['name'])
    criterion = criterion_builder(cfg=tltd(criterion_cfg['args']))
    return criterion
```

8.4.2 计算损失函数

以下展示了如何计算损失，你可以替换其中的损失函数类型。

```
from fgvclib.criterions.utils import LossItem

losses = list()
losses.append(LossItem(name='cross_entropy_loss', value=self.criterions['cross_
↪entropy_loss']['fn'](x, targets)))
```

8.4.3 定义前向传播

以 ResNet50 结构为例:

```
from fgvclib.criterions.utils import LossItem

def forward(self, x, targets=None):
    x = self.infer(x)
    if self.training:
```

(下页继续)

(续上页)

```
        losses = list()
        osses.append(LossItem(name='cross_entropy_loss', value=self.criterions['cross_
↪entropy_loss']['fn'](x, targets)))
        return x, losses

    return x
```

CHAPTER 9

教程 4: 学习数据集加载文件

在 fgvclib 中，我们主要使用鸟类的数据集：`CUB_200_2011` 我们建立这个文件夹去加载数据集，我们定义了 `get_dataset` 函数，通过给定的数据集名称，返回对应的数据集。

```
def get_dataset(dataset_name) -> FGVCDataset:
    r"""Return the dataset with the given name.

        Args:
            dataset_name (str):
                The name of dataset.

        Return:
            The dataset contructor method.
    """

    if dataset_name not in globals():
        raise NotImplementedError(f"Dataset {dataset_name} not found!\nAvailable␣
↪datasets: {available_datasets()}")
    return globals()[dataset_name]
```

9.1 FGVC 数据集

首先，我们应该知道我们具有什么数据集。我们定义了函数 `available_datasets` 来展示所有的可用的 FGVC 数据集，这个函数将会返回所有可用的 FGVC 数据集列表。

然后，我们建立了一个类 `FGVCDataset` 作为 `CUB_200_2011` 类的输入，`CUB_200_2011` 类被用来加载 CUB_200_2011 数据集。

我们列出了和对应数据集相关的下载链接，关于 CUB_200_2011 数据集的分支文件夹、文件。

如果你没有相应的数据集，请讲参数 `download` 设为 **true**

```
    name: str = "Caltech-UCSD Birds-200-2011"
    link: str = "http://www.vision.caltech.edu/datasets/cub_200_2011/"
    download_link: str = "https://data.caltech.edu/records/65de6-vp158/files/CUB_200_
↪2011.tgz?download=1"
    category_file: str = "CUB_200_2011/CUB_200_2011/classes.txt"
    annotation_file: str = "CUB_200_2011/CUB_200_2011/image_class_labels.txt"
    image_dir: str = "CUB_200_2011/CUB_200_2011/images/"
    split_file: str = "CUB_200_2011/CUB_200_2011/train_test_split.txt"
    images_list_file: str = "CUB_200_2011/CUB_200_2011/images.txt"
```

CHAPTER 10

教程 5: 学习评价指标文件

我们提供了三种评价标准：准确率 `accuracy`、精确率 `precision`、召回率 `recall` 作为训练和测试的结果。从”Torchmetrics”中调用者三种评价指标，同时，在”**init**”中设置了评价指标的列表 `__all__ = ["accuracy", "precision", "recall"]`

关于准确率 **accuracy** 参数的更多细节参见torchmetrics.Accuracy object

关于精确率 **precision** 参数的更多细节参见torchmetrics.Precision object

关于召回率 **recall** 参数的更多细节参见torchmetrics.Recall object

10.1 准确率 Accuracy

准确率 `accuracy` 被定义为: `accuracy(name:str="accuracy(top-1)", top_k:int=1, threshold:float=None)`

- 参数:

  `"name(str)"`: 评价指标的名称, 比如 accuracy(top-1) `"top_k (int)"`: 找到正确标签时的最高概率或 logit 分数预测的数量 `"threshhold (float, optional)"`: 在二进制或多标签输入的情况下，将概率或 logit 预测转换为二进制（0，1）预测的阈值

- 返回值:

  `NamedMetirc`: 自定义名称的 torchmetrics 度量

10.2 精确率 Precision

精确率 precision 被定义为 precision(name:str="precision(threshold=0.5)", top_k:int=None, threshold:float=0.5)

- 参数:

  "name(str)": 评价指标的名称, 比如 accuracy(top-1) "top_k (int)": 找到正确标签时的最高概率或 logit 分数预测的数量 "threshhold (float, optional)": 在二进制或多标签输入的情况下, 将概率或 logit 预测转换为二进制（0，1）预测的阈值

- 返回值:

  NamedMetirc: 自定义名称的 torchmetrics 度量

10.3 召回率 Recall

召回率 recall 被定义为 recall(name:str="recall(threshold=0.5)", top_k:int=None, threshold:float=0.5)

- 参数:

  "name(str)": 评价指标的名称, 比如 accuracy(top-1) "top_k (int)": 找到正确标签时的最高概率或 logit 分数预测的数量 "threshhold (float, optional)": 在二进制或多标签输入的情况下, 将概率或 logit 预测转换为二进制（0，1）预测的阈值

- 返回值:

  NamedMetirc: 自定义名称的 torchmetrics 度量

10.4 举例

10.4.1 为评估构建度量标准

```
from fgvclib.metrics import get_metric
from fgvclib.metrics import NamedMetric

def build_metrics(metrics_cfg: CfgNode, use_cuda:bool=True) -> t.List[NamedMetric]:

    metrics = []
    for cfg in metrics_cfg:
        metric = get_metric(cfg["metric"])(name=cfg["name"], top_k=cfg["top_k"],␣
↪threshold=cfg["threshold"])
        if use_cuda:
```

(下页继续)

(续上页)

```
            metric = metric.cuda()
        metrics.append(metric)
    return metrics
```

10.4.2 评估 FGVC 模型

```
def evaluate_model(model:nn.Module, p_bar:t.Iterable, metrics:t.List[NamedMetric],␣
↪use_cuda:bool=True) -> t.Dict:

    model.eval()
    results = dict()

    with torch.no_grad():
        for _, (inputs, targets) in enumerate(p_bar):
            if use_cuda:
                inputs, targets = inputs.cuda(), targets.cuda()
            inputs, targets = Variable(inputs), Variable(targets)
            for metric in metrics:
                _ = metric.update(model(inputs), targets)

    for metric in metrics:
        result = metric.compute()
        results.update({
            metric.name: round(result.item(), 3)
        })

    return results
```

10.4.3 准确率的输出

In the processing of train:

```
acc = evaluate_model(model, test_bar, metrics=cfg.METRICS, use_cuda=cfg.USE_CUDA)
logger("Evalution Result:")
logger(acc)
```

In the processing of predict:

```
metrics = build_metrics(cfg.METRICS)
acc = evaluate_model(model, pbar, metrics=metrics, use_cuda=cfg.USE_CUDA)
```

(下页继续)

(续上页)

```
print(acc)
```

CHAPTER 11

教程 6: 学习模型文件

在这一部分，我们将模型进行拆分，并对其进行封装。我们为模型提供了 `backbones`, `encoders`, `heads`, `necks`, `sotas`, 和 `utils` 这些组件，你可以分别选择它们组装模型。

- **Backbone**: 骨干网大多书的时候指特征提取网络，它的作用是提取图片中的信息，然后使用该网络, 常见的有：**ResNet**, **VGG** 等.
- **Encoder**: 池化层可以减小数据体的空间大小, 从而减少网络中的参数数量, 进而减少计算资源消耗, 有效控制过拟合。
- **Neck**: backbone 和 head 之间的组成部分
- **Head**: 特定任务的组件
- **Sotas**: 最先进的模型

11.1 Backbone

我们主要提供两类 backbone，ResNet 和 VGG.

| ResNet | VGG | | resnet18 | vgg11 | | resnet34 | vgg13 | | resnet50 | vgg16 | | resnet101 | vgg19 | | resnet152 |———————————————| | resnet50_32x4d |———————————————| | resnet101_32x8d |———————————————| | resnet50_bc |———————————————| | resnet101_bc |———————————————|

在”fgvclib/models/backbones/**init**.py”中，我们定义了 `get_backbone` 函数，根据给出的 backbone 名称返回对应的 backbone。backbone 名称如下：`resnet18`, `resnet34`, `resnet50`, `resnet101`, `resnet152`,

`resnext50_32x4d`, `resnext101_32x8d`, `resnet50_bc`, `resnet101_bc`, `vgg11`, `vgg13`, `vgg16`, `vgg19`

```
def get_backbone(backbone_name):
    if backbone_name not in globals():
        raise NotImplementedError(f"Backbone {backbone_name} not found!\nAvailable␣
↪backbones: {__all__}")
    return globals()[backbone_name]
```

11.1.1 ResNet

我们从 Pytorch 中加载了 ResNet-x 模型，并且定义了函数用于构造 ResNet-x 模型。

resnet18:

- 参数:

  pretrained (bool): 如果该值为 True，则返回在 ImageNet 上的预训练模型 progress (bool): 如果该值为 True，则显示下载的进度条

- 返回值:

  _resnet（‘resnet18’, BasicBlock, [2, 2, 2, 2], cfg, progress, **kwargs)

该函数返回 `_resnet`，`_resnet` 返回对应的模型，`_resnet` 中包含关于模型类别的输入参数

其他的 backbone 和 resnet18 类似，不同的地方在于返回值

resnet50_32x4d

- 返回值:

  _resnet（‘resnext50_32x4d’, Bottleneck, [3, 4, 6, 3], cfg, progress=True, **kwargs)

resnet50_32x4d needs to add the folllowing code:

```
kwargs['groups'] = 32
kwargs['width_per_group'] = 4
```

resnet101_32x8d

- 返回值: _resnet（‘resnext101_32x8d’, Bottleneck, [3, 4, 23, 3], cfg, progress=True, **kwargs)

resnet101_32x8d needs to add the folllowing code:

```
kwargs['groups'] = 32
kwargs['width_per_group'] = 8
```

11.1.2 VGG

我们从 Pytorch 中加载了 VGG-x 模型，并且定义了函数用于构造 VGG-x 模型。

vgg11:

- 参数:

  pretrained (bool): If True, returns a model pre-trained on ImageNet progress (bool): If True, displays a progress bar of the download to stderr

- 返回值:

  _vgg(“vgg11”, cfg, progress)

该函数返回 `_vgg`，`_vgg` 返回对应的模型，`_vgg` 中包含关于模型类别的输入参数

其他的 backbone 和 vgg11 类似，不同的地方在于返回值

11.1.3 举例

当你需要建立一个 FGVC 模型，你可以使用它得到一个骨干网。在 FGVCLib，我们根据配置构建 FGVC 模型，有关配置 **configs** 的更多细节，请参考FGVC Configs.

在 `fgvclib/apis/build.py`，函数 `build_model` 根据配置构建 FGVC 模型，在 `model_cfg` 中，我们提前设置了 backbone 名称。

```
from fgvclib.models.backbones import get_backbone

backbone_builder = get_backbone(model_cfg.BACKBONE.NAME)
backbone = backbone_builder(cfg=tltd(model_cfg.BACKBONE.ARGS))
```

11.2 Encoders

我们提供了三种类型的池化层 `global average pooling`, `global max pooling` 和 `max pooling 2d`

在” fgvclib/models/encoders/**init**.py” 中，我们定义了 `get_encoding` 函数，根据提供的池化层类型返回对应的编码器。给出的池化层名称有：`global_avg_pooling`, `global_max_pooling`, `max_pooling_2d`

```
def get_encoding(encoding_name):
    if encoding_name not in globals():
        raise NotImplementedError(f"Encoding not found: {encoding_name}\nAvailable␣
↪encodings: {__all__}")
    return globals()[encoding_name]
```

11.2.1 全局平均池化

首先我们定义了一个类：GlobalAvgPooling 作为全局平均池化编码器。然后我们定义了一个函数 global_avg_pooling

11.2.2 全局最大池化

首先，我们定义了一个类：GlobalMaxPooling 作为全局最大池化编码器。然后，我们定义了一个函数 global_max_pooling

11.2.3 Max pooling 2d

```
def max_pooling_2d(cfg):
    assert 'kernel_size' in cfg.keys()
    assert isinstance(cfg['kernel_size'], int)
    assert 'stride' in cfg.keys()
    assert isinstance(cfg['stride'], int)
    return nn.MaxPool2d(kernel_size=cfg['kernel_size'], stride=cfg['stride'])
```

11.2.4 举例

当你需要构建一个 FGVC 模型时，你可以使用它构建一个编码器。在 FGVCLib 中，我们根据配置构建 FGVC 模型，关于配置 **configs** 的细节，请参考FGVC Configs.

在 fgvclib/apis/build.py 中，函数 build_model 根据配置构建 FGVC 模型，在 model_cfg 中，我们提前设置了编码器名称。

```
from fgvclib.models.encoders import get_encoding

if model_cfg.ENCODING.NAME:
        encoding_builder = get_encoding(model_cfg.ENCODING.NAME)
        encoding = encoding_builder(cfg=tltd(model_cfg.ENCODING.ARGS))
    else:
        encoding = None
```

11.3 Necks

我们为 fgvclib 提供了一种 neck，`Multi-scale Convolution neck`，在” fgvclib/models/necks/**init**.py” 中，我们定义了一个函数 `get_neck`，根据给出的 neck 名称返回对应的 neck。给出的 neck 名称有: `multi_scale_conv`

```
def get_neck(neck_name):
    """Return the backbone with the given name."""
    if neck_name not in globals():
        raise NotImplementedError(f"Neck not found: {neck_name}\nAvailable necks: {__
↪all__}")
    return globals()[neck_name]
```

11.3.1 Multi-scale Convolution neck

首先，我们定义一个类 `MultiScaleConv` 作为 Multi-scale Convolution neck，然后，我们定义了一个函数 `multi_scale_conv`。

11.3.2 举例

当你需要构建一个 FGVC 模型时，你可以使用它构建一个 neck。在 FGVCLib 中，我们根据配置构建 FGVC 模型，关于配置 **configs** 的细节，请参考FGVC Configs.

在 `fgvclib/apis/build.py` 中，函数 `build_model` 根据配置构建 FGVC 模型，在 `model_cfg` 中，我们提前设置了 neck 名称。

```
from fgvclib.models.necks import get_neck

if model_cfg.NECKS.NAME:
    neck_builder = get_neck(model_cfg.NECKS.NAME)
    necks = neck_builder(cfg=tltd(model_cfg.NECKS.ARGS))
else:
    necks = None
```

11.4 Heads

我们主要提供两种分类器，`classifier_1fc`, and `classifier_2fc`，在” fgvclib/models/heads/**init**.py” 中，我们定义了一个函数 `get_head`，根据给出的 head 名称返回对应的 head。给出的 head 名称有: `classifier_1fc`, and `classifier_2fc`

```
def get_head(head_name):
    """Return the backbone with the given name."""
    if head_name not in globals():
        raise NotImplementedError(f"Head not found: {head_name}\nAvailable heads: {__
↪all__}")
    return globals()[head_name]
```

11.4.1 Classifier_1FC

首先，我们定义一个类：`Classifier_1FrC` 作为具有一个全连接层的分类器，然后，我们定义一个函数 `classifier_1fc`

11.4.2 Classifier_2FC

首先，我们定义一个类：`Classifier_1FrC` 作为具有两个全连接层的分类器，然后，我们定义一个函数 `classifier_1fc`

11.4.3 举例

当你需要构建一个 FGVC 模型时，你可以使用它构建一个 head。在 FGVCLib 中，我们根据配置构建 FGVC 模型，关于配置 **configs** 的细节，请参考FGVC Configs.

在 `fgvclib/apis/build.py` 中，函数 `build_model` 根据配置构建 FGVC 模型，在 `model_cfg` 中，我们提前设置了 head 名称。

```
from fgvclib.models.heads import get_head

head_builder = get_head(model_cfg.HEADS.NAME)
    heads = head_builder(class_num=model_cfg.CLASS_NUM, cfg=tltd(model_cfg.HEADS.
↪ARGS))
```

11.5 Sotas

我们复现了几个最先进的模型，`baseline_resnet50`, `mcl`, `pmg_resnet50`, `pmg_v2_resnet50`，在” fgvclib/models/heads/**init**.py” 中，我们定义了一个函数 `get_model`，根据给出的 model 名称返回对应的 model。给出的 model 名称有：`PMG_ResNet50`, `PMG_V2_ResNet50`, `Baseline_ResNet50`, `MCL`

```
def get_model(model_name):
    """Return the model class with the given name."""
    if model_name not in globals():
```

(下页继续)

(续上页)

```
        raise NotImplementedError(f"Model {model_name} not found!\nAvailable models:
↪{__all__}")
    return globals()[model_name]
```

- Baseline_resnet50: 使用 resnet50 作为主干网络去构建模型作为基准模型
- MCL: 这个模型在”The Devil is in the Channels: Mutual-Channel Loss for Fine-Grained Image Classification”论文中被提出，关于此模型的更多细节参考MCL
- PMG: 这个模型在”Fine-Grained Visual Classification via Progressive Multi-Granularity Training of Jigsaw Patches”论文中被提出, 关于此模型的更多细节参考PMG

11.5.1 举例

当你需要构建一个 FGVC 模型时，你可以使用它获得模型，在 FGVCLib 中，我们根据配置构建 FGVC 模型，关于配置 **configs** 的更多细节，请参考FGVC Configs.

在 `fgvclib/apis/build.py` 中，函数 `build_model` 根据配置构建 FGVC 模型，在 `model_cfg` 中，我们提前设置了 model 名称

```
from fgvclib.models.sotas import get_model

model_builder = get_model(model_cfg.NAME)
model = model_builder(backbone=backbone, encoding=encoding, necks=necks, heads=heads,␣
↪criterions=criterions)
```

11.6 构建一个模型

一个完整的模型由 **backbone**, **encoder**, **neck**, **head**, 和 **loss** 这几部分组成，我们将模型的各个部分进行拆分，你可以自由的组合它们去构建一个新的模型，或者复现其他的工作，你需要事先在配置中设置好模型的参数，才能调用这些模块来构建模型。

11.6.1 举例说明构建模型的过程

```
from fgvclib.metrics import get_metric
from fgvclib.models.sotas import get_model
from fgvclib.models.backbones import get_backbone
from fgvclib.models.encoders import get_encoding
from fgvclib.models.necks import get_neck
from fgvclib.models.heads import get_head
```

(下页继续)

(续上页)

```
def build_model(model_cfg: CfgNode) -> nn.Module:
    r"""Build a FGVC model according to config.

    Args:
        model_cfg (CfgNode): The model config node of root config.
    Returns:
        nn.Module: The FGVC model.
    """

    backbone_builder = get_backbone(model_cfg.BACKBONE.NAME)
    backbone = backbone_builder(cfg=tltd(model_cfg.BACKBONE.ARGS))

    if model_cfg.ENCODING.NAME:
        encoding_builder = get_encoding(model_cfg.ENCODING.NAME)
        encoding = encoding_builder(cfg=tltd(model_cfg.ENCODING.ARGS))
    else:
        encoding = None

    if model_cfg.NECKS.NAME:
        neck_builder = get_neck(model_cfg.NECKS.NAME)
        necks = neck_builder(cfg=tltd(model_cfg.NECKS.ARGS))
    else:
        necks = None

    head_builder = get_head(model_cfg.HEADS.NAME)
    heads = head_builder(class_num=model_cfg.CLASS_NUM, cfg=tltd(model_cfg.HEADS.
↪ARGS))

    criterions = {}
    for item in model_cfg.CRITERIONS:
        criterions.update({item["name"]: {"fn": build_criterion(item), "w": item["w"]}
↪})

    model_builder = get_model(model_cfg.NAME)
    model = model_builder(backbone=backbone, encoding=encoding, necks=necks,␣
↪heads=heads, criterions=criterions)

    return model
```

CHAPTER 12

教程 7: 学习转换文件

我们引入 `transforms` 来处理图片，我们导入了六类转换的方法 `resize`, `random crop`,`center crop`, `random horizontal flip`, `to tensor`. `normaliza`

- **Resize**: 将图像调整为给定的大小.
- **Random crop**: 在随机位置裁剪给定的图像. If the image is torch Tensor, it is expected to have [···, H, W] shape, where ···means an arbitrary number of leading dimensions, but if non-constant padding is used, the input is expected to have at most 2 leading dimensions
- **Center crop**: 在中心裁剪给定的图像 If the image is torch Tensor, it is expected to have [···, H, W] shape, where ···means an arbitrary number of leading dimensions. If image size is smaller than output size along any edge, image is padded with 0 and then center cropped.
- **Random horizontal flip**: 以给定的概率随机地水平翻转给定的图像. If the image is torch Tensor, it is expected to have [···, H, W] shape, where ···means an arbitrary number of leading dimensions
- **To tensor**: 转换 PIL Image 或 numpy.ndarry 到 tensor
- **Normalize**: 用均值和标准超归一化图像

关于转换的更多细节请参见torchvision.transforms

我们导入了 `torchvision` 和 `PIL` 去定义转换函数

在”fgvclib/transforms/**init**.py”中，我们定义了函数 `get_transform`，根据给定的转换类型返回转换函数，给定的转换类型有：`resize`, `center_crop`, `random_crop`, `random_horizontal_flip`, `to_tensor`, `normalize`

```
def get_transform(transform_name):
    """Return the backbone with the given name."""
    if transform_name not in globals():
        raise NotImplementedError(f"Transform not found: {transform_name}\nAvailable␣
↪transforms: {__all__}")
    return globals()[transform_name]
```

12.1 举例

网络参数事先保存在配置中，可以根据配置对训练数据集或测试数据集进行变换。

```
from fgvclib.transforms import get_transform

def build_transforms(transforms_cfg: CfgNode) -> transforms.Compose:
    """
    Args:
        transforms_cfg (CfgNode): The root config node.
    Returns:
        transforms.Compose: The transforms.Compose object in Pytorch.
    """
    return transforms.Compose([get_transform(item['name'])(item) for item in␣
↪transforms_cfg])
```

CHAPTER 13

Learn about utils

我们为 FGVCLib 添加了一些工具，这些工具包括解释器、记录器、学习率表、更新策略和可视化。

13.1 解释器

我们选择了类激活映射工具，我们设计了一个名为 CAM 的类，累计或映射工具用于解释分类结果。所有的方法君来自于 (pytorch_grad_cam)[git@github.com:jacobgil/pytorch-grad-cam.git]。方法有：gradcam, hirescam, scorecam, gradcam++, xgradcam, eigencam, eigengrafcam, layercam, fullgrad, gradcamelementeise.

以下是关于类 CAM 的一些参数：

- `model (nn.Module)`: FGVC 模型
- `target_layers (list)`: 该层用于得到 CAM 权重
- `use_cuda (bool)`: 是否使用 gpu
- `method (str)`: 可用的 CAM 方法
- `aug_smooth (str)`: 平滑法具有更好的使 CAM 围绕物体居中的作用
- `eigen_smooth (str)`: 平滑法具有移动噪声的作用

在” fgvclib/utils/interpreter/**init**.py” 中，我们定义了函数 `get_interpreter`，根据给定的名称返回对应的解释器，给定的名称有：`cam`

```
def get_interpreter(interpreter_name):
    r"""
```

(下页继续)

(续上页)

```
        Args:
            interpreter_name (str):
                The name of interpreter.

        Return:
            The interpreter contructor method.
    """
    if interpreter_name not in globals():
        raise NotImplementedError(f"Interpreter not found: {interpreter_name}\
↪nAvailable interpreters: {__all__}")
    return globals()[interpreter_name]
```

13.1.1 举例

以构建一个解释器为例

```
gvclib.utils.interpreter import get_interpreter, Interpreter

def build_interpreter(model: nn.Module, cfg: CfgNode) -> Interpreter:
    r"""
    Args:
        cfg (CfgNode): The root config node.
    Returns:
        Interpreter: A Interpreter.
    """
    return get_interpreter(cfg.INTERPRETER.NAME)(model, cfg)
```

13.2 记录器

我们定义了两种记录器，`txt logger` 和 `wandb logger`

在”fgvclib/utils/logger/**init**.py”中，我们定义了一个函数 `get_logger`，根据给定的名称返回对应的记录器，个 icing 的名称有：`wandb_logger`, `txt_logger`

```
def get_logger(logger_name):
    r"""Return the logger with the given name.

        Args:
            logger_name (str):
                The name of logger.
```

(下页继续)

(续上页)

```
        Return:
            The logger contructor method.
    """

    if logger_name not in globals():
        raise NotImplementedError(f"Logger not found: {logger_name}\nAvailable␣
↪loggers: {__all__}")
    return globals()[logger_name]
```

13.2.1 举例

它可以用于构建记录器对象或生成记录器

```
def build_logger(cfg: CfgNode) -> Logger:
    r"""Build a Logger object according to config.

    Args:
        cfg (CfgNode): The root config node.
    Returns:
        Logger: The Logger object.
    """

    return get_logger(cfg.LOGGER.NAME)(cfg)
```

13.3 学习率表

在”fgvclib/utils/lr_schedules/**init**.py”中，我们定义了一个函数 `get_lr_schedule`，根据给定的名称返回对应的学习率表，给定的名称有：`cosine_anneal_schedule`

```
def get_lr_schedule(lr_schedule_name):
    r"""Return the learning rate schedule with the given name.

        Args:
            lr_schedule_name (str):
                The name of learning rate schedule.

        Return:
            The learning rate schedule contructor method.
    """

    if lr_schedule_name not in globals():
```

(下页继续)

(续上页)

```
        raise NotImplementedError(f"Learning rate schedule not found: {lr_schedule_
↪name}\nAvailable learning rate schedules: {__all__}")
    return globals()[lr_schedule_name]
```

并且，我们定义了函数 cosine_anneal_schedule

```
def cosine_anneal_schedule(optimizer, current_epoch, total_epoch):
    cos_inner = np.pi * (current_epoch % (total_epoch))
    cos_inner /= (total_epoch)
    cos_out = np.cos(cos_inner) + 1

    for i in range(len(optimizer.param_groups)):
        current_lr = optimizer.param_groups[i]['lr']
        optimizer.param_groups[i]['lr'] = float(current_lr / 2 * cos_out)
```

13.3.1 举例

可以在 main.py 文件中，在训练过程中使用它

```
from fgvclib.utils.lr_schedules import cosine_anneal_schedule

   cosine_anneal_schedule(optimizer, epoch, cfg.EPOCH_NUM)
```

13.4 更新策略

我们提供了三种类型的更新策略构造方法：progressive updating with jigsaw, progressive updating consistency constraint, 和 general updating

progressive updating with jigsaw: 有关用 jigsaw 渐进式更新的更多详细信息，参见文件” fgvclib/utils/update_strategy/progressibe_updating_with_jigsaw.py”

progressive updating consistency constraint: 有关渐进式更新一致性约束的详细信息，参见文件” fgvclib/utils/update_strategy/progressive_updating_consistency_constraint.py”

general updating: 有关一般更新的详细信息，参见” fgvclib/utils/update_strategy/general_updating.py”

在” fgvclib/utils/update_strategy/**init**.py” 中，我们定义了一个函数 get_update_strategy，根据给定的名称返回对应的更新策略方法，给出的名称有：progressive_updating_with_jigsaw, progressive_updating_consistency_constraint, general_updating

```
def get_update_strategy(strategy_name):
    r"""
```

(下页继续)

(续上页)

```
        Args:
            strategy_name (str):
                The name of the update strategy.

        Return:
            The update strategy contructor method.
    """

    if strategy_name not in globals():
        raise NotImplementedError(f"Strategy not found: {strategy_name}\nAvailable␣
↪strategy: {__all__}")
    return globals()[strategy_name]
```

13.4.1 举例

在更新模型时导入该模块，使用更新策略构造方法更新 FGVC 模型

在”fgvclib/apis/update_model.py”中，我们导入了 fgvclib.utils.update_strategy

```
from fgvclib.utils.update_strategy import get_update_strategy
from fgvclib.utils.logger import Logger

def update_model(model: nn.Module, optimizer: Optimizer, pbar:Iterable, strategy:str=
↪"general_updating", use_cuda:bool=True, logger:Logger=None):
    model.train()
    mean_loss = 0.
    for batch_idx, train_data in enumerate(pbar):
        losses_info = get_update_strategy(strategy)(model, train_data, optimizer, use_
↪cuda)
        mean_loss = (mean_loss * batch_idx + losses_info['iter_loss']) / (batch_idx +␣
↪1)
        losses_info.update({"mean_loss": mean_loss})
        logger(losses_info, step=batch_idx)
        pbar.set_postfix(losses_info)
```

13.5 可视化

我们设计该模块将结果进行可视化, 这个模块可以帮助显示热图，帮助我们更好的理解实验结果。在这个模块中，我们导入了’fiftyone’，并且我们创建了一个名为’VOXEL’的类。

```
class VOXEL:

    def __init__(self, dataset, name:str, persistent:bool=False, cuda:bool=True,␣
↪interpreter:Interpreter=None) -> None:
        self.dataset = dataset
        self.name = name
        self.persistent = persistent
        self.cuda = cuda
        self.interpreter = interpreter

        if self.name not in self.loaded_datasets():
            self.fo_dataset = self.create_dataset()
            self.load()
        else:
            self.fo_dataset = fo.load_dataset(self.name)

        self.view = self.fo_dataset.view()

    def create_dataset(self) -> fo.Dataset:
        return fo.Dataset(self.name)

    def loaded_datasets(self) -> t.List:
        return fo.list_datasets()

    def load(self):

        samples = []

        for i in tqdm(range(len(self.dataset))):
            path, anno = self.dataset.get_imgpath_anno_pair(i)

            sample = fo.Sample(filepath=path)

            # Store classification in a field name of your choice
            sample["ground_truth"] = fo.Classification(label=anno)

            samples.append(sample)

            # Create dataset
```

(下页继续)

(续上页)

```
        self.fo_dataset.add_samples(samples)
        self.fo_dataset.persistent = self.persistent

    def predict(self, model:nn.Module, transforms, n:int=inf, name="prediction",␣
↪seed=51, explain:bool=False):
        model.eval()
        if n < inf:
            self.view = self.fo_dataset.take(n, seed=seed)

        with fo.ProgressBar() as pb:
            for sample in pb(self.view):
                image = Image.open(sample.filepath)
                image = transforms(image).unsqueeze(0)

                if self.cuda:
                    image = image.cuda()
                    pred = model(image)
                    index = torch.argmax(pred).item()
                    confidence = pred[:, index].item()


                sample[name] = fo.Classification(
                    label=str(index),
                    confidence=confidence
                )

                if self.interpreter:
                    heatmap = self.interpreter(image_path=sample.filepath, image_
↪tensor=image, transforms=transforms)
                    sample["heatmap"] = fo.Heatmap(map=heatmap)

                sample.save()
        print("Finished adding predictions")
```

CHAPTER 14

English

CHAPTER 15

简体中文