FGVClib

Release 0.1

yyq

Jan 14, 2023

GET STARTED

1 Prerequisites **1**

2 Installation **3**
- 2.1 Best practices . . . 3
- 2.2 Trouble shooting . . . 3

3 Benchmark and model zoo **5**
- 3.1 Common settings . . . 5
- 3.2 Backbone models . . . 5
- 3.3 Methods . . . 5

4 The conigs of the state-of-the-art methods **7**

5 1: Train with the existing models and satndard datastes **13**
- 5.1 The existing models . . . 13
- 5.2 Prepare the dataset . . . 13
- 5.3 Train . . . 14
- 5.4 Test . . . 14

6 2: Train with the new models and satndard datastes **15**
- 6.1 Prepare the dataset . . . 15
- 6.2 Prepare your own customized model . . . 15
- 6.3 Prepare a config . . . 16
- 6.4 Train a new model . . . 17
- 6.5 Test a new model . . . 17

7 Tutorial 1: Learn about apis **19**
- 7.1 Build . . . 19
- 7.2 Evaluate Model . . . 21
- 7.3 Save Model . . . 21
- 7.4 Update Model . . . 21
- 7.5 The example of using the apis . . . 21

8 Tutorial 2: Learn about configs **23**
- 8.1 Config File Structure . . . 23
- 8.2 The example of the configs . . . 25

9 Tutorial 3: Learn about criterions **27**
- 9.1 Base loss function . . . 27
- 9.2 Mutual channel loss . . . 28
- 9.3 Utils . . . 28

9.4 The example of using the criterions . . . 28

10 Tutorial 4: Learn about datasets **31**
10.1 FGVC Dataset . . . 32

11 Tutorial 5: Learn about metrics **33**
11.1 Accuracy . . . 33
11.2 Precision . . . 33
11.3 Recall . . . 34
11.4 The example . . . 34

12 Tutorial 6: Learn about model **37**
12.1 Backbone . . . 37
12.2 Encoders . . . 39
12.3 Necks . . . 40
12.4 Heads . . . 40
12.5 Sotas . . . 41
12.6 Build a model . . . 42

13 Tutorial 7: Learn about transforms **45**
13.1 The example . . . 46

14 Learn about utils **47**
14.1 Interpreter . . . 47
14.2 Logger . . . 48
14.3 Learning rate schedules . . . 49
14.4 Update strategy . . . 50
14.5 Visualization . . . 51

15 English **53**

16 **55**

CHAPTER

ONE

PREREQUISITES

In this section we demonstrate how to prepare an environment with PyTorch. FGVCLib works on Linux. It requires Python 3.7+,CUDA 10.0+ and PyTorch.

```
If you are experienced with PyTorch and have already installed it, just skip this part␣
↪and jump to the [next section](#installation). Otherwise, you can follow these steps␣
↪for the preparation.
```

Step 0. Download and install Anaconda from the official website.

Step 1. Create a conda environment and activate it.

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

Step 2. Install Pytorch following official website, e.g.

On GPU platforms:

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

CHAPTER

TWO

INSTALLATION

2.1 Best practices

We recommend that users follow our best practices to install FGVCLib. And FGVCLib needs some requirements to install.

Step 0. Install FGVCLib

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

Step 1. Install the requirments

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

2.2 Trouble shooting

```
Maybe you will meet problems when you install the 'fiftyone', if you have the trouble,␣
↪you can refer to the following.
```

If the version of Ubuntu >= 18.04, you can execute

```
pip install fiftyone
```

If the version of Ubuntu < 18.04, you can execute

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

If you have the error "error while loading shared libraries: libcurl.so.4: cannot open shared object file: No such file", please check whether there is `curl` or not. If you don't have the `curl`, please execute

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

CHAPTER

THREE

BENCHMARK AND MODEL ZOO

3.1 Common settings

- All models were trained on `CUB_200_2011_train` and tested on the `CUB_200_2011_test`.
- For fair comparison with other codebases, we report the GPU memory as the maximum value of `torch.cuda.max_memory_allocated()` for all 8 GPUs. Note that this value is usually less than what `nvidia-smi` shows.
- All pytorch-style pretrained backbone are form PyTorch model zoo.

3.2 Backbone models

The detailed table of the commonly used backbone models in FGVCLib is listed below:

3.3 Methods

3.3.1 MCL

Please refer to MCL for details.

3.3.2 PMG

Please refer to PMG for details.

CHAPTER

FOUR

THE CONIGS OF THE STATE-OF-THE-ART METHODS

In "/configs/", we write four file to save the configs of the state-of-the-art methods. And the file type is `.yml` The configs combine `experiment name`, `weight`, `logger`, `dataset`, `model`, `transforms`, `optimizer`, and Validation details.

Take the **"baseline_resnet50"** for example.

- Experiment name

  EXP_NAME: the name of this method.

```
EXP_NAME: "Baseline_ResNet50"
```

- Resume weight

  RESUME_WEIGHT is set ~

```
RESUME_WEIGHT: ~
```

- Logger

  LOGGER: the logger of this process.

```
LOGGER:
  NAME: "txt_logger"
```

- Dataset

  DATASET: the dataset

  - NAME: the name of the dataset
  - ROOT: the root of the dataset, **you should change it to your own root in advance**
  - TRAIN: the parameters about the processing of training
    - BATCH_SIZE: the batch size
    - POSITIVE: the positive
    - PIN_MEMORY: is bool, True or False
    - SHUFFLE: is bool, True or False
    - NUM_WORKERS: the number of workers
  - TEST: the parameters about the processing of testing
    - BATCH_SIZE: the batch size
    - POSITIVE: the positive

        * PIN_MEMORY: is bool, True or False
        * SHUFFLE: is bool, True or False
        * NUM_WORKERS: the number of workers

```
DATASET:
  NAME: "CUB_200_2011"
  ROOT: "/data/wangxinran/dataset/"
  TRAIN:
    BATCH_SIZE: 2
    POSITIVE: 0
    PIN_MEMORY: True
    SHUFFLE: True
    NUM_WORKERS: 4
  TEST:
    BATCH_SIZE: 2
    POSITIVE: 0
    PIN_MEMORY: False
    SHUFFLE: False
    NUM_WORKERS: 4
```

- Model

  MODEL: the model

  - NAME: the name of the model
  - CLASS_NUM: the number of the class
  - CRITERIONS: the loss
    * name: the name of loss function
    * args: the args are determined by loss
    * w: the weight
  - BACKBONE: the backbone of the model, the parameters about the args are set according the corresonding file.
    * NAME: the name of the bockbone
    * ARGS:
      · pretrained: is bool, True or False
      · del_keys: is a list of the del_keys
  - ENCODING: the encoding of the model, the parameters about the args are set according the corresonding file.
    * NAME: the name of the encoding
  - NECKS: the neck of the model
    * NAME: the name of the neck
  - HEADS: the head of the model, the parameters about the ARGS are set according the corresonding file.
    * NAME: the name of the head
    * ARGS: the args are determined by the classifier.

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

- Transforms

  TRANSFORMS: the transforms

  - TRAIN: the parameters about the process of training
    - name: the name of the transform type
    - other parameters: according the transform type to set the corresonding parameters.
  - TEST: the parameters about the process of testing
    - name: the name of the transform type
    - other parameters: according the transform type to set the corresonding parameters.

```
TRANSFORMS:
  TRAIN:
    - name: "resize"
      size:
        - 600
        - 600
    - name: "random_crop"
      size: 448
      padding: 8
    - name: "random_horizontal_flip"
      prob: 0.5
    - name: "to_tensor"
    - name: "normalize"
      mean:
         - 0.5
         - 0.5
         - 0.5
      std:
```

(continues on next page)

(continued from previous page)

```
      - 0.5
      - 0.5
      - 0.5
  TEST:
    - name: "resize"
      size:
        - 600
        - 600
    - name: "center_crop"
      size: 448
    - name: "to_tensor"
    - name: "normalize"
      mean:
         - 0.5
         - 0.5
         - 0.5
      std:
        - 0.5
        - 0.5
        - 0.5
```

- Optimizer

  OPTIMIZER: the optimizer

  - NAME: the name of the optimizer
  - MOMENTUM: the momentum of the optimizer
  - LR: the learning rate
    - backbone: the learning rate of the backbone
    - encoding: the learning rate of the encoding
    - necks: the learning rate of the neck
    - heads: the learning rate of the head

```
OPTIMIZER:
  NAME: "SGD"
  MOMENTUM: 0.9
  LR:
    backbone: 0.0002
    encoding: 0.002
    necks: 0.002
    heads: 0.002
```

- Iteration number

  ITERATION_NUM: the Iteration number

```
ITERATION_NUM: ~
```

- Epoch number

  EPOCH_NUM: the epcoh number

```
START_EPOCH: 0
```

- Update strategy

  UPDATE_STRATEGY: the update strategy

```
UPDATE_STRATEGY: "general_updating"
```

- Per iteration

  PER_ITERATION: the per iteration

```
PER_ITERATION: ~
```

- Per epoch

  PER_EPOCH the per per iteration

```
PER_EPOCH: ~
```

- Metrics

  METRICS: the metrics

  - name: the name of the metric type
  - other parameters: according the metrics type to set the corresonding parameters.

```
METRICS:
  - name: "accuracy(topk=1)"
    metric: "accuracy"
    top_k: 1
    threshold: ~
  - name: "accuracy(topk=5)"
    metric: "accuracy"
    top_k: 5
    threshold: ~
  - name: "recall(threshold=0.5)"
    metric: "recall"
    top_k: ~
    threshold: 0.5
  - name: "precision(threshold=0.5)"
    metric: "precision"
    top_k: ~
    threshold: 0.5
```

- Interpreter

  INTERPRETER: the interpreter

  - NAME: the name of the interpreter
  - METHOD: the method of the interpreter
  - TARGET_LAYERS: the target layers

CHAPTER

FIVE

1: TRAIN WITH THE EXISTING MODELS AND SATNDARD DATASTES

To evaluate a model's accuracy, one usually tests the model on some standard datasets. FGVCLib supports the public datasets including CUB_200_2011. This section will show how to test existing models on supported datasets.

The basic steps are as below: 1.Prepare the dataset 2.Prepare a config 3.Train, test models on the dataset

5.1 The existing models

We provide a variety of existing methods, they are `baseline_resnet50`, `MCL`, `PMG`, `PMG_v2`, `API-Net`, `CAL`, `PIM`, `TransFG`.

In the future we will continue to reproduce new methods and add them into FGVCLib.

5.2 Prepare the dataset

We provide the CUB-200-2011, and we split the dataset into train and test folder.

e.g., CUB-200-2011 dataset

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

If you have prepared the dataset, you can skip the step1.

step1: open the "/fgvclib/datasets/cub.py", and modify the `class CUB_200_2011: __init__ : download:bool=False` to `class CUB_200_2011: __init__ : download:bool=True`

```
The parameter 'download' controls whether the dataset is downloaded. Directly␣
↪downloading CUB dataset by setting download=True. Default is False.
```

step2: open the "/configs/xxx/xxx.yml", and replace the `DATASET-ROOT` with your own path.

5.3 Train

step1: open the "/configs/xxx/xxx.yml", and replace the `WEIGHT-SAVE_DIR` with your own path. **step2**: open the "/configs/xxx/xxx.yml", and check the configs about the model. You can change the configs by yourself. **stpe3**: execute main program to train.

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

There are several arguments to control the program.

- '–config': the path of configuration file.
- '–task': train or predict. The default is **train**.
- '–device': two choices are cuda and cpu. The default is **cuda**.
- '–world-size': the number of distributed processes. The default is 4.
- '–dist-url': url used to set up distributed training. The default is 'env://'.

If you want to run it on cpu, you should execute the following

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

5.4 Test

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

CHAPTER

SIX

2: TRAIN WITH THE NEW MODELS AND SATNDARD DATASTES

We disassemble the model into backbone, encoder, classifier and other basic structures, and then combine them to build the complete method. In the FGVCLib, we have provide the basic structures and reproduce the state-of-art model. We are committed to providing you with a custom structure, using the disassembled modules to reassemble a new model.

The basic steps are as below: 1.Prepare the dataset 2.Prepare you own customized model 3.Prepare a config 4.Train, test and predict models on the dataset

6.1 Prepare the dataset

You need to change the corresponding dataset paths in the config files. And you need to split the dataset into train and test folder.

e.g., CUB-200-2011 dataset

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

6.2 Prepare your own customized model

The second step is to use your own module or training setting. Assume that we want to add a new encoder `xxx`.

6.2.1 1.Define a new encoder(e.g. xxx)

Firstly we create a new file `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
```

6.2.2 2.Import the module

You can import the encoder in other parts

```
from .xxx import xxx
```

6.3 Prepare a config

The third step is to prepare a config for your own training setting. In "configs/xxx.yml", you should prepare a complete config file. Refer to the existing config file, you can create a new config file. Take the new encoder xxx as an example:

```
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:
```

(continues on next page)

(continued from previous page)

```
    - in_dim:
      - 2048
```

6.4 Train a new model

To train a model with the new config, you can simply run

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

6.5 Test a new model

To train a model with the new config, you can simply run

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

CHAPTER

SEVEN

TUTORIAL 1: LEARN ABOUT APIS

In this folder "fgvclib/api" we set up the various apis interfaces for the fgvclib. There are 4 types apis interfaces in this folder, `build.py`, `evluate_model.py`, `save_model.py`, and `update_model.py`

"fgvclib/apis/build.py" provides various apis for building a training or evaluation system fast.

"fgvclib/apis/evaluate_model.py" provides a api for evaluating FGVC algorithms.

"fgvclib/apis/save_model.py"provides various apis for saving a model.

"fgvclib/apis/update_model"provides various apis for updating models and recording losses.

7.1 Build

We import the other modules to build the model, in this part, there are eight functions. The configs about the model are saved in the folder "./configs". For more detailes about the **configs** , see FGVC Configs

build_model: Build a FGVC model according to config.

- Args:

  `model_cfg (CfgNode)`: The model config node of root config.

- Returns:

  `fgvclib.models.sota.FGVCSOTA`: The FGVC model.

build_logger: Build a Logger object according to config.

- Args:

  `cfg (CfgNode)`: The root config node.

- Returns:

  `Logger`: The Logger object.

build_transforms: Build transforms for train or test dataset according to config.

- Args:

  `transforms_cfg (CfgNode)`: The root config node.

- Returns:

  `PyTorch transforms.Compose`: The transforms.Compose object in Pytorch.

build_dataset: Build a dataloader for training or evaluation.

- Args:

  `name(str)`: The dataset name. `root (str)`: The directory of dataset. `cfg (CfgNode)`: The mode config of the dataset config. `mode(str)`: The split of the dataset. `transform`: Pytorch Transformer Compose.

- Returns:

  A FGVCDataset.

build_dataset: Build a dataloader for training or evaluation.

- Args:

  `dataset (FGVCDataset)`: A FGVCDataset. `mode_cfg (CfgNode)`: The mode config of the dataset config. `sampler (Sampler)`: The dataloder sampler.

- Returns:

  `DataLoader`: A Pytorch Dataloader.

build_optimizer: Build a optimizer for training.

- Args:

  `optim_cfg (CfgNode)`: The optimizer config node of root config node.

- Returns:

  `Optimizer`: A Pytorch Optimizer.

build_criterion :Build loss function for training.

- Args:

  `criterion_cfg` (CfgNode): The criterion config node of root config node.

- Returns:

  `nn.Module`: A loss function.

build_interpreter: Build loss function for training.

- Args:

  `cfg (CfgNode)`: The root config node.

- Returns:

  `Interpreter`: A Interpreter.

build_metrics: Build metrics for evaluation.

- Args:

  `metrics_cfg (CfgNode)`: The metric config node of root config node.

- Returns:

  `t.List[NamedMetric]`: A List of NamedMetric.

7.2 Evaluate Model

We import the other modules to evaluate the model, in this part, there is one function. The configs about the model are saved in the folder "./configs". For more detailes about the **configs** , see FGVC Configs

evaluate_model: Evaluate the FGVC model according to config.

- Args:

  `model(nn.Module)`: The FGVC model. `p_bar(iterable)`: A iterable provide test data. `metrics(List[NamedMetric])`: List of metrics. `use_cuda(boolean, optional)`: Whether to use gpu.

- Returns:

  `dict`: The result dict.

7.3 Save Model

We design this module to save the trained FGVC model.

save_model: Save the trained FGVC model.

- Args:

  `cfg (CfgNode)`: The root config node. `model (nn.Module)`: The FGVC model. `logger (Logger)`: The Logger object.

7.4 Update Model

We design this module tp update the FGVC model and record losses.

update_model: Update the FGVC model and record losses.

- Args:

  `model (nn.Module)`: The FGVC model. `optimizer (Optimizer)`: The Logger object. `pbar (Iterable)`: A iterable object provide training data. `strategy (string)`: The update strategy. `use_cuda (boolean)`: Whether to use GPU to train the model. `logger (Logger)`: The Logger object.

7.5 The example of using the apis

When you do algorithm design, you need to import the apis `from fgvclib.apis import *` and call the interfaces. You can use the following functions directly, `build_logger`, `build_criterion`, `build_model`, `build_metrics`, `build_transforms`, `build_dataset`, `build_optimizer`, `update_model`, `evaluate_model`, `save_model`, `build_interpreter`

7.5.1 The example of building model

```
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

EIGHT

TUTORIAL 2: LEARN ABOUT CONFIGS

In this folder "fgvclib/configs" we show the configs about the fgvclib. We modularized the config in the experiment, creating `FGVCConfig` class to load and store the parameters.And you can load config by using `FGVCConfig`

8.1 Config File Structure

There are 4 basic component types under "fgvclib/configs/config.py",`__init__`,`get_cfg`,`load`,`stringfy`.

We set the parameters for the fgvclib, and you can search or modify the parameters in `config.py`.

The following is about the basic parameters.

```
# 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/"

# 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
```

The following parameters are about datasets.

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

(continues on next page)

(continued from previous page)

```
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
```

The following parameters are about the model.

```
# 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()
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
```

(continues on next page)

(continued from previous page)

```
# 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
```

The following parameters are about the processing.

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

# Validation
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 = []
```

8.2 The example of the configs

In the `main.py`, you can import the configs `from fgvclib.configs import FGVCConfig`, and use it to load config.

```
import os
import torch

from fgvclib.configs import FGVCConfig

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

CHAPTER

NINE

TUTORIAL 3: LEARN ABOUT CRITERIONS

In this folder "fgvclib/criterions" we provide different loss functions for the fgvclib.

We provide four loss functions, `cross_entropy_loss`, `binary_cross_entropy_loss`, `mean_square_error_loss` and `mutual_channel_loss`

You can choose which loss function you want to use, and you should set it in the "./configs". For more details about the configs please see FGVC Configs

9.1 Base loss function

`cross_entropy_loss`, `binary_cross_entropy_loss`, `mean_square_error_loss` are the base loss functions, and they are from PyTorch. "fgvclib/criterions/base_loss.py": provides the base loss functions.

cross_entropy_loss: Build the cross entropy loss function.

- Args:

  `cfg (CfgNode)`: The root node of config.

- Return:

  `nn.Module`: The loss function.

binary_cross_entropy_loss:Build the binary cross entropy loss function.

- Args:

  `cfg (CfgNode)`: The root node of config.

- Return:

  `nn.Module`: The loss function.

mean_square_error_loss: Build the mean square error loss function.

- Args:

  `cfg (CfgNode)`: The root node of config.

- Return:

  `nn.Module`: The loss function.

9.2 Mutual channel loss

"fgvclib/criterions/mutual_channel_loss.py": provides the mutual channel loss function which was proposed on "The Devil is in the Channels: Mutual-Channel Loss for Fine-Grained Image Classification".

`class MutualChannelLoss`: The mutual channel loss function.

- Args:

  `height (int)`: The kernel size of average pooling. `cnum (int)`: Channel numbers per class. `div_weight (float)`: The weight for diversity part loss. `dis_weight (float)`: The weight for discriminality part loss.

9.3 Utils

In the "fgvclib/criterions/utils.py", we design a class named `LossItem`, and two functions, `compute_loss_value` and `detach_loss_value`.

LossItem: A dataclass object for store training loss

- Args:

  name (string): The loss item name. value (torch.Tensor): The value of loss. weight (float, optional): The weight of current loss item, default is 1.0.

compute_loss_value: A dataclass object for store training loss

- Args:

  items (List[LossItem]): The loss items.

- Return:

  Tensor: The total loss value.

detach_loss_value: Detach loss value from GPU.

- Args:

  items (List[LossItem]): The loss items.

- Return:

  Dict: A loss information dict whose key is loss name, value is loss value.

9.4 The example of using the criterions

9.4.1 Build loss functions for training.

In the "fgvclib/apis/build.py", use the "fgvclib.criterions" to build loss functions for training. You can choose the loss function name `criterion_cfg['name']` from `cross_entropy_loss`, `cross_entropy_loss`, `mean_square_error_loss` and `mutual_channel_loss`.

```
from fgvclib.criterions import get_criterion

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

(continues on next page)

(continued from previous page)

```
    criterion = criterion_builder(cfg=tltd(criterion_cfg['args']))
    return criterion
```

9.4.2 Calculate loss functions.

Following is about how to calculate the loss, and you can replace the loss functions.

```
from fgvclib.criterions.utils import LossItem

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

9.4.3 Define the forward.

Set the ResNet50 for example.

```
from fgvclib.criterions.utils import LossItem

def forward(self, x, targets=None):
    x = self.infer(x)
    if self.training:
        losses = list()
        osses.append(LossItem(name='cross_entropy_loss', value=self.criterions['cross_
↪entropy_loss']['fn'](x, targets)))
        return x, losses

    return x
```

CHAPTER

TEN

TUTORIAL 4: LEARN ABOUT DATASETS

In FGVCLib, we mainly use the Birds dataset, `CUB_200_2011`. We build this folder to load the dataset. We define the function `get_dataset` to get the dataset dict, and using the function `dataset` to return the dataset

```
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.
    """
```

```
def dataset(name):

    def register_function_fn(cls):
        if name in __DATASET_DICT__:
            raise ValueError("Name %s already registered!" % name)
        if not issubclass(cls, FGVCDataset):
            raise ValueError("Class %s is not a subclass of %s" % (cls, FGVCDataset))
        __DATASET_DICT__[name] = cls
        return cls

    return register_function_fn

for file in os.listdir(os.path.dirname(__file__)):
    if file.endswith('.py') and not file.startswith('_'):
        module_name = file[:file.find('.py')]
        module = importlib.import_module('fgvclib.datasets.' + module_name)
```

10.1 FGVC Dataset

Firstly, we should know what dataset we have. We define the function `available_datasets` to show all available FGVC datasets, and this function will return a list wih all available FGVC datasets.

Then, we set a class `FGVCDataset` as the input of class `CUB_200_2011` which is used for loading the CUB_200_2011 dataset.

CUB_200_2011 is the Caltech-UCSD Birds-200-2011 dataset. We list the relevant link, file, and dir about CUB_200_2011.

If you don't have the dataset, please set the `download` **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"
```

- Args:

  root (str): The root directory of CUB dataset. mode (str): The split of CUB dataset. download (bool): Directly downloading CUB dataset by setting download=True. Default is False. transforms (torchvision.transforms.Compose): The PyTorch transforms Compose class used to preprocessing the data. positive (int): If positive = n > 0, the **getitem** method will an extra list of n images of same category.

CHAPTER

ELEVEN

TUTORIAL 5: LEARN ABOUT METRICS

We provide 3 metrics, `accuracy`, `precision`, `recall` as the results of training and testing. They are from the Torchmetrics, and in "**init**" we set the list of them `__all__ = ["accuracy", "precision", "recall"]`

For details about the meanings of the **accuracy** parameters, see torchmetrics.Accuracy object.

For details about the meanings of the **precision** parameters, see torchmetrics.Precision object.

For details about the meanings of the **recall** parameters, see torchmetrics.Recall object.

11.1 Accuracy

The `accuracy` is defined as `accuracy(name:str="accuracy(top-1)", top_k:int=1, threshold:float=None)`

- Args:

  `"name(str)"`: The name of metric, e.g. accuracy(top-1) `"top_k (int)"`: Number of the highest probability or logit score predictions considered finding the correct label. `"threshhold (float, optional)"`: Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.

- Return:

  `NamedMetirc`: A torchmetrics metric with customed name.

11.2 Precision

The `precision` is defined as `precision(name:str="precision(threshold=0.5)", top_k:int=None, threshold:float=0.5)`

- Args:

  `"name(str)"`: The name of metric, e.g. accuracy(top-1) `"top_k (int)"`: Number of the highest probability or logit score predictions considered finding the correct label. `"threshhold (float, optional)"`: Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.

- Return:

  `NamedMetirc`: A torchmetrics metric with customed name.

11.3 Recall

The `recall` is defined as `recall(name:str="recall(threshold=0.5)", top_k:int=None, threshold:float=0.5)`

- Args:

  `"name(str)"`: The name of metric, e.g. accuracy(top-1) `"top_k (int)"`: Number of the highest probability or logit score predictions considered finding the correct label. `"threshhold (float, optional)"`: Threshold for transforming probability or logit predictions to binary (0,1) predictions, in the case of binary or multi-label inputs.

- Return:

  `NamedMetirc`: A torchmetrics metric with customed name.

11.4 The example

11.4.1 Build metrics for evaluation.

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

11.4.2 Evaluate the FGVC model.

```
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:
```

(continues on next page)

(continued from previous page)

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

    return results
```

11.4.3 Output the accuracy.

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)

print(acc)
```

CHAPTER

TWELVE

TUTORIAL 6: LEARN ABOUT MODEL

In this part, we took the model apart and encapsulated each part of the model. We provide `backbones`, `encoders`, `heads`, `necks`, `sotas`, and `utils` for the model. You can select them separately to go to the component model.

- **Backbone**: The backbone network most of time refers to the feature extraction network, its role is to extract the information in the picture, and then use the network, e.g. **ResNet**, **VGG**.
- **Encoder**: The pooling layer can reduce the space size of the data body, so that the number of parameters in the network can be reduced, so that the computing resource consumption is reduced, and the overfitting can be effectively controlled.
- **Neck**: The component between backbones and heads.
- **Head**: The component for specific tasks.
- **Sotas**: State-of-the-art model.

12.1 Backbone

We mainly provide two categories backbone, ResNet and VGG.

In the fgvclib/models/backbones/**init**.py", we define a function `get_backbone` to return the backbone with the givenname. The given names are `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]
```

12.1.1 ResNet

We download the ResNet-x model from pytorch, and define the functions to construct the ResNet-x model.

resnet18:

- Args:

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

- Return:

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

This function return the `_resnet`, and `_resnet` return the model. The `_resnet` has some input parameters about the model category.

Other backbones are similar to the resnet18, the difference lies on the **return**.

resnet50_32x4d

- Return:

  _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

- Return: _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
```

12.1.2 VGG

We download the VGG-x model from pytorch, and define the functions to construct the VGG-x model.

vgg11:

- Args:

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

- Return:

  _vgg("vgg11", cfg, progress)

This function return the `_vgg`, and `_vgg` return the model. The `_vgg` has some input parameters about the model category.

Other backbones are similar to the vgg11, the difference lies on the **return**.

12.1.3 The example

When you need to build a FGVC model, you can use it to get a backbone. In the FGVCLib, we build a FGVC model according to config. For detailes about the **configs** , see FGVC Configs.

In `fgvclib/apis/build.py`, there is a function build_model to build a FGVC model according to config. In the `model_cfg`, we have set the backbone name.

```
from fgvclib.models.backbones import get_backbone

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

12.2 Encoders

We provide three kind of pooling layer, `global average pooling`, `global max pooling` and `max pooling 2d`.

In the fgvclib/models/encoders/**init**.py", we define a function `get_encoding` to return the encoder with the given name. And the given names are `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]
```

12.2.1 Global average pooling

Firstly, we define a class named `GlobalAvgPooling` as global average pooling encoding. Then, we define a function named `global_avg_pooling`.

12.2.2 Global max pooling

Firstly, we define a class named `GlobalMaxPooling` as global average pooling encoding. Then, we define a function named `global_max_pooling`.

12.2.3 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'])
```

12.2.4 The example

When you need to build a FGVC model, you can use it to get a encoding. In the FGVCLib, we build a FGVC model according to config. For detailes about the **configs** , see FGVC Configs.

In `fgvclib/apis/build.py`, there is a function build_model to build a FGVC model according to config. In the `model_cfg`, we have set the encoding name.

```
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
```

12.3 Necks

We provide one kind neck for the fgvclib, `Multi-scale Convolution neck`.

In the fgvclib/models/necks/**init**.py", we define a function `get_neck` to return the neck with the given name. And the given name is `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]
```

12.3.1 Multi-scale Convolution neck

Firstly, we define a class named `MultiScaleConv` as a Multi-scale Convolution neck. Then, we define a function named `multi_scale_conv`.

12.3.2 The example

When you need to build a FGVC model, you can use it to get a neck. In the FGVCLib, we build a FGVC model according to config. For detailes about the **configs** , see FGVC Configs.

In `fgvclib/apis/build.py`, there is a function build_model to build a FGVC model according to config. In the `model_cfg`, we have set the neck name.

```
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
```

12.4 Heads

We mainly provide two classifier, `classifier_1fc`, and `classifier_2fc`

In the fgvclib/models/heads/**init**.py", we define a function `get_head` to return the head with the given name. And the given names are `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]
```

12.4.1 Classifier_1FC

Firstly, we define a class named `Classifier_1FC` as a classifier with one fully connected layer. Then, we define a function named `classifier_1fc`.

12.4.2 Classifier_2FC

Firstly, we define a class named `Classifier_2FC` as a classifier with two fully connected layer. Then, we define a function named `classifier_2fc`.

12.4.3 The example

When you need to build a FGVC model, you can use it to get a head. In the FGVCLib, we build a FGVC model according to config. For detailes about the **configs** , see FGVC Configs.

In `fgvclib/apis/build.py`, there is a function build_model to build a FGVC model according to config. In the `model_cfg`, we have set the head name.

```
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))
```

12.5 Sotas

We reproduced state-of-the-art models, `baseline_resnet50`, `mcl`, `pmg_resnet50`, `pmg_v2_resnet50`.

In the fgvclib/models/sotas/**init**.py", we define a function `get_model` to return the head with the given name. And the given names are `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():
        raise NotImplementedError(f"Model {model_name} not found!\nAvailable models: {__
↪all__}")
    return globals()[model_name]
```

- Baseline_resnet50: Using the resnet50 as the backbone to build a model which is the baseline.
- MCL: This model was proposed in the paper "The Devil is in the Channels: Mutual-Channel Loss for Fine-Grained Image Classification", for more details about this method, seeMCL
- PMG: This model was proposed in the paper " Fine-Grained Visual Classication via Progressive Multi-Granularity Training of Jigsaw Patches", for more details about this method, see PMG

12.5.1 The example

When you need to build a FGVC model, you can use it to get a model. In the FGVCLib, we build a FGVC model according to config. For detailes about the **configs** , see FGVC Configs.

In `fgvclib/apis/build.py`, there is a function build_model to build a FGVC model according to config. In the `model_cfg`, we have set the model name.

```
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)
```

12.6 Build a model

A model is made up **backbone**, **encoder**, **neck**, **head**, and **loss**. We take the model apart and you can combine them to build a new model or replicate other network. You should configure the model parameters in the configs in advance and then you can invoking these moudles to build model.

12.6.1 Take an example to show the process of building a model.

```
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

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))
```

(continues on next page)

(continued from previous page)

```
    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

THIRTEEN

TUTORIAL 7: LEARN ABOUT TRANSFORMS

We import the `transforms` to process the image. The transfomers are from the torchvision. We add six categories transform method, `resize`, `random crop`,`center crop`, `random horizontal flip`, `to tensor`. `normaliza`.

- **Resize**: Resize the input image to the given size. If the image is torch Tensor, it is expected to have [..., H, W] shape, where ... means an arbitrary number of leading dimensions.
- **Random crop**: Crop the given image at a random location. 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**: Crops the given image at the center. 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**: Horizontally flip the given image randomly with a given probability. If the image is torch Tensor, it is expected to have [..., H, W] shape, where ... means an arbitrary number of leading dimensions
- **To tensor**: Convert a PIL Image or numpy.ndarray to tensor. This transform does not support torchscript.
- **Normalize**: Normalize a tensor image with mean and standard deviation.

For more details about transforms, see torchvision.transforms

We import the `torchvision` and `PIL` to define the transform functions.

And in "fgvclib/transforms/**init**.py", we define the function `get_transform` to return the transforms with the given name. The given names are `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]
```

13.1 The example

The parameters about the network are saved in the configs in advance, so we can build transforms for train or test dataset according to config.

```
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

FOURTEEN

LEARN ABOUT UTILS

We add some tools for the fgvc, and the tools are about interpreter, logger, learning rate schedules, updating strategy, and visualization.

14.1 Interpreter

We chose the class activation map tool. We design a class named CAM, the class actication map tool is for explaning the classification result. All methods are from (pytorch_grad_cam)[git@github.com:jacobgil/pytorch-grad-cam.git]. The methods are gradcam, hirescam, scorecam, gradcam++, xgradcam, eigencam, eigengrafcam, layercam, fullgrad, gradcamelementeise.

There are some args for the class CAM:

- `model (nn.Module)`: The FGVC model
- `target_layers (list)`: The layers used to get CAM weights
- `use_cuda (bool)`: Wheter use gpu
- `method (str)`: The available CAM methods
- `aug_smooth (str)`: The smooth method has the effect of better centering the CAM around the objects
- `eigen_smooth (str)`: The smooth method has the effect of removing a lot of noise.

In "fgvclib/utils/interpreter/**init**.py", we define a function named get_interpreter to return the interpreter with the given name. And the given name is `cam`.

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

14.1.1 The example

It is used to build interpreter.

```
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)
```

14.2 Logger

We define two types logger, `txt logger` and `wandb logger`.

In "fgvclib/utils/logger/**init**.py" we define a function named `get_logger` to return the logger with the given name, and the given names are `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.

        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]
```

14.2.1 The example

It can be used to build a logger object or generate the logger.

```
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.
    """
```

(continues on next page)

(continued from previous page)

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

14.3 Learning rate schedules

In "fgvclib/utils/lr_schedules/**init**.py" we define a function named `get_lr_schedule` to return the learning rate schedule with the given name, and the given name is `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():
        raise NotImplementedError(f"Learning rate schedule not found: {lr_schedule_name}\
↪nAvailable learning rate schedules: {__all__}")
    return globals()[lr_schedule_name]
```

And we define the function named `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)
```

14.3.1 The example

It can be used in the file `main.py` for the processing of training.

```
from fgvclib.utils.lr_schedules import cosine_anneal_schedule

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

14.4 Update strategy

We provide three types update strategy contructor methods, `progressive updating with jigsaw`, `progressive updating consistency constraint`, and `general updating`.

progressive updating with jigsaw: For more details about progressive updating with jigsaw, see "fgvclib/utils/update_strategy/progressibe_updating_with_jigsaw.py".

progressive updating consistency constraint: For more details about progressive updating consistency constraint, see "fgvclib/utils/update_strategy/progressive_updating_consistency_constraint.py".

general updating: For more details about general updating, see "fgvclib/utils/update_strategy/general_updating.py".

In "fgvclib/utils/update_strategy/**init**.py", we define a function named `get_update_strategy` to return the update stratrgy contructor method with the given name. And the given names are `progressive_updating_with_jigsaw`, `progressive_updating_consistency_constraint`, `general_updating`

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

14.4.1 The example

The update stratrgy contructor method is used to update the FGCV model, so we can import it when update model.

In "fgvclib/apis/update_model.py", we import 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)
```

14.5 Visualization

We designed this module to visualize the results. This module can help to show the heat map, which is better for the result. In this module, `fiftyone` is mainly imported and we create a class named `VOXEL`.

```
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

        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)
```

(continues on next page)

(continued from previous page)

```
        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

FIFTEEN

ENGLISH

CHAPTER

SIXTEEN