LibFewShot

Release 0.0.1-alpha

R&L Group

Jun 04, 2023

GETTING STARTED

CHAPTER

ONE

INTRODUCTION

`LibFewShot` is a comprehensive library for few-shot learning (FSL), especially for few-shot image classification. It integrates multiple classic FSL methods into a unified framework, including four fine-tuning based methods, six meta-learning based methods, and eight metric-learning based methods. This library is friendly for beginners in few-shot learning with very concise code and clear structure.

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

CHAPTER

TWO

CONTRIBUTORS

Special acknowledgment is first given to LegendDong, who built the foundation of this library and completed the most of algorithms. The following excellent contributors also participated in the development of this library throughout the process: WenbinLee, yangcedrus, wZuck, WonderSeven, Pinzhuo Tian, onlyyao, and cjy97.

CHAPTER

THREE

INSTALLATION

This section provides a tutorial on building a working environment for `LibFewShot` from scratch.

CHAPTER

FOUR

GET THE LIBFEWSHOT LIBRARY

Use the following command to get LibFewShot:

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

CHAPTER

FIVE

CONFIGURE THE LIBFEWSHOT ENVIRONMENT

The environment can be configured in any of the following ways:

1. conda(recommend)

```
cd <path-to-LibFewShot> # cd in `LibFewShot` directory
conda env create -f requirements.yaml
```

2. pip

```
cd <path-to-LibFewShot> # cd in `LibFewShot` directory
pip install -r requirements.txt
```

3. or whatever works for you as long as the following package version conditions are meet:

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

CHAPTER

SIX

TEST THE INSTALLATION

1. set the `config` as follows in `run_trainer.py`:

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

2. modify `data_root` in `config/headers/data.yaml` to the path of the dataset to be used.
3. run code

```
python run_trainer.py
```

4. If the first output is correct, it means that `LibFewShot` has been successfully installed.

CHAPTER

SEVEN

NEXT

For model training and code modification, please see the *train/test methods already integrated in LibFewShot* and other sections of the tutorial.

CHAPTER

EIGHT

GETTING STARTED

This section shows an example of a process of using `LibFewShot`.

8.1 Prepare the dataset (use miniImageNet as an example)

1. download and extract miniimagent–ravi.
2. check the structure of the dataset

   The dataset must be in the following structure:

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

8.2 Modify the config file

Use `ProtoNet` as an example

1. create a new `yaml` file `getting_started.yaml` in `config/`
2. write the following commands into the created file:

```yaml
includes:
  - headers/data.yaml
  - headers/device.yaml
  - headers/losses.yaml
  - headers/misc.yaml
  - headers/model.yaml
  - headers/optimizer.yaml
  - classifiers/Proto.yaml
  - backbones/Conv64F.yaml
```

More details can be referred to *write a config yaml*.

8.3 Run

1. set the `config` as follows in `run_trainer.py`:

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

2. train with the console command:

```
python run_trainer.py
```

3. wait for the end of training.

8.4 View the log files

After running the program, you can find a symlink of `results/ProtoNet-miniImageNet-Conv64F-5-1` and a directory of `results/ProtoNet-miniImageNet-Conv64F-5-1-$TS`, where `TS` means the timestamp. The directory contains two folders: `checkpoint/` and `log_files/`, and a configuration file: `config.yaml`. Note that the symlink will always link to the directory created at the last time, when you train the model with the same few-shot learning configuration for multiple times.

`config.yaml` contains all the settings used in the training phase.

`log_files/` contains tensorboard files, training log files and test log files.

`checkpoints/` contains model checkpoints saved at `$save_insterval` intervals, the last model checkpoint (used to resume) and the best model checkpoint (used to test). The checkpoint files are generally divided into `emb_func.pth`, `classifier.pth`, and `model.pth` (a combination of the first two), respectively.

CHAPTER

NINE

WRITE A `.YAML` CONFIGURATION FILE

Code for this section:

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

9.1 Composition of the configuration file in LibFewShot

The configuration file of LibFewShot uses a yaml format file and it also supports reading the global configuration changes from the command line. We have pre-defined a default configuration `core/config/default.yaml`. The users can put the custom configuration into the `config/` directory, and save this file in the `yaml` format. At parsing, the sequencing relationship of defining the configuration of the method is `default.yaml->config/->console`. The latter definition overrides the same value in the former definition.

Although most of the basic configurations have been set in the `default.yaml`, you can not directly run a program just using the `default.yaml`. Before running the code, the users are required to define a configuration file of one method that has been implemented in LibFewShot in the `config/` directory.

Considering that FSL menthods usually have some basic parameters, such as `way, shot` or `device id`, which are often needed to be changed, LibFewShot also supports making changes to some simple configurations on the command line without modifying the `yaml` file. Similarly, during training and test, because many parameters are the same of different methods, we wrap these same parameters together and put them into the`config/headers` for brevity. In this way, we can write the `yaml` files of the custom methods succinctly by importing them.

The following is the composition of the files in the `config/headers` directory.

- `data.yaml`: The relevant configuration of the data is defined in this file.
- `device.yaml`: The relevant configuration of GPU is defined in this file.
- `losses.yaml`: The relevant configuration of the loss used for training is defined in this file.
- `misc.yaml`: The miscellaneous configuration is defined in this file.
- `model.yaml`: The relevant configuration of the model is defined in this file.
- `optimizer.yaml`: The relevant configuration of the optimizer used for training is defined in this file.

9.2 The settings of the configuration file in LibFewShot

The following details each part of the configuration file and explain how to write them. An example of how the DN4 method is configured is also presented.

9.2.1 The settings for data

- `data_root`: The storage path of the dataset.
- `image_size`: The size of the input image.
- `use_momery`: Whether to use memory to accelerate reading.
- `augment`: Whether to use data augmentation.
- `augment_timessupport_set`: The number of data augmentation/transformations used. Expanding the `support set` data for multiple times.
- `augment_times_queryquery_set`: The number of data augmentation/transformations used. Expanding the `query set` data for multiple times.

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

9.2.2 The settings for model

- `backbone`: The `backbone` information used in the method.
  - `name`: The name of the `backbone`, needs to match the case of the `backbone` implemented in LibFewShot.
  - `kwargs`: The parameters used in the `backbone`, must keep the name consistent with the name in the code.
    - is_flatten: The default is False, and if `True`, the feature vector after flatten is returned.
    - avg_pool: The default is False, and if `True`, the feature vector after `global average pooling` is returned.
    - is_feature: The default is False, and if `True`, the output of each `block` in `backbone` is returned.

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

- `classifier`: The `classifier` information used in the method.
  - `name`: The name of the `classifier`, needs to match the case of the `classifier` implemented in LibFewShot.
  - `kwargs`: The parameters used in the `classifier` initialization, must keep the name consistent with the name in the code.

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

9.2.3 The settings for training

- epoch: The number of epoch during training.
- test_epoch: The number of epoch during testing.
- pretrain_path: The path of the pre-training weights. At the beginning of the training, this setting will be first checked. If it is not empty, the pre-trained weights of the target path will be loaded into the backbone of the current training.
- resume: If set to True, the training status is read from the default address to support continual training.
- way_num: The number of way during training.
- shot_num: The number of shot during training.
- query_num: The number of query during training.
- test_way: The number of way during testing. If not specified, the way_num is assigned to the test_way.
- test_shot: The number of shot during testing. If not specified, the shot_num is assigned to the test_way.
- test_query: The number of query during testing. If not specified, the query_num is assigned to the test_way
- episode_size: The number of tasks/episodes used for the network training at each time.
- batch_size: The batch size used when the pre-training model is pre-trained. In some kinds of methods, this property is useless.
- train_episode: The number of tasks per epoch during training.
- test_episode: The number of tasks per epoch during testing.

```yaml
epoch: 50
test_epoch: 5

pretrain_path: ~
resume: False

way_num: 5
shot_num: 5
query_num: 15
test_way: ~
test_shot: ~
test_query: ~
episode_size: 1
# batch_size only works in pre-train
batch_size: 128
train_episode: 10000
test_episode: 1000
```

9.2.4 The settings for optimizer

- `optimizer`: Optimizer information used during training.
- `name`: The name of the Optimizer, only temporarily supports all Optimizers provided by `PyTorch`.
- `kwargs`: The parameters used in the optimizer, and the name needs to be the same as the parameter name required by the pytorch optimizer.
- `other`: Currently, the framework only supports the learning rate used by each part of a separately specified method, and the name needs to be the same as the variable name used in the method.

```
optimizer:
    name: Adam
    kwargs:
        lr: 0.01
    other:
        emb_func: 0.01
        #For demonstration purposes, there are no additional training parameters for dn4.
        dnf_layer: 0.001
```

`lr_scheduler`: The learning rate adjustment strategy used during training, only temporarily supports all the learning rate adjustment strategies provided by `PyTorch`.

- `name`: The name of the learning rate adjustment strategy.
- `kwargs`: Other parameters used in the learning rate adjustment strategy in `PyTorch`.

```
lr_scheduler:
  name: StepLR
  kwargs:
    gamma: 0.5
    step_size: 10
```

9.2.5 The settings for Hardware

`device_ids`: The `gpu` number, which is the same as the `nvidia-smi` command.

`n_gpu`: The number of parallel `gpu` used during training, if `1`, it can't apply to parallel training.

`deterministic`: Whether to turn on `torch.backend.cudnn.benchmark` and `torch.backend.cudnn.deterministic` and whether to determine random seeds during training.

`seed`: Seed points used in `numpytorch`and `cuda`.

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

9.2.6 The settings for Miscellaneous

`log_name`: If empty, use the auto-generated `classifier.name-data_root-backbone-way_num-shot_num` file directory.

`log_level`: The log output level during training.

`log_interval`: The number of tasks for the log output interval.

`result_root`: The root of the result.

`save_interval`: The epoch interval to save weights.

`save_part`: The name of the variable in the method that needs to be saved. Variables with these names are saved separately when the model is saved. The parts that need to be saved are given as a list under `save_part`.

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

CHAPTER

TEN

TRAIN/TEST EXISTING METHODS IN LIBFEWSHOT

Code for this section

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

In this section, we take the DN4 method as an example to describe how to train and test an implemented method.

10.1 Configuration files

In t0-write_a_config_yaml.md, we have showed how to write a configuration file. We also assemble some of the common configuration into the public file, so that you can easily finish your DN4 configuration file.

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

For specific customer requirements, you can modify the related included files or use other files and add your own configuration.

10.2 Train

Name the configuration file we have finished in previous section as `dn4.yaml`, place it into the `config/` directory.

Modify the `run_trainer.py` file in project root as follow:

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

Next, run this instruction in your shell

```
python run_trainer.py
```

and the training will start.

10.3 Test

Modify the run_test.py file in project root as follow:

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

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

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


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

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

Input in your shell:

```
python run_test.py
```

and the testing will start.

Of course, all of the VAR_DICT variables inrun_test.py can be removed, by running instruction as follows

```
python run_test.py --test_epoch 5 --device_ids 4 --n_gpu 1 --test_episode 600 --episode_
↪size 1
```

to achieve the same effect.

CHAPTER

ELEVEN

USE DATASET

In `LibFewShot`, datasets have a fixed format. We read the data according to the datasets in most few-shot learning settings, like *mini*ImageNet and *tiered*ImageNet. Some datasets like Caltech-UCSD Birds 200 can be downloaded from the internet and unzipped for using directly.

If you want to use a new dataset but its data format is different from the above datasets, you need to transform it into the same dataset format.

11.1 Dataset format

Like *mini*ImageNet, dataset format should be the same as follows:

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

All training, evaluating and testing images should be placed in the `images` directory, by using `train.csvtest.csv` and `val.csv` files to split the dataset, respectively. These three files have similar format, and are organized as follows:

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

The CSV head contains only two columns, one of which is filename and the other is label. The filename should be a relative path from the `images` directory. It means that, for an image with absolute path `.../dataset_folder/images/images_1.jpg`its `filename` should be `images_1.jpg`. In a similar way, for an image with absolute path `.../dataset_folder/images/class_name_1/images_1.jpg`, its `filename` should be `class_name_1/images_1.jpg`.

11.2 Configure Dataset

After downloading a dataset and transforming it into the above dataset format, you only need to change the `data_root` in the configuration file. Notice that `LibeFewShot` will print the data directory's name as well as the dataset's name into the log.

CHAPTER

TWELVE

TRANSFORMATIONS

Code for this section

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

In `LibFewShot`we use a base `transforms` to compare some methods fairly. The base `transforms` could be divided into three sub-transforms:

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

There are some differences in `Resize&Crop` for different dataset and config file (key `augment`):

1. in the training phase, `config.augment` is `True`

   ```
   from torchvision import transforms
   transforms.RandomResizedCrop((config.image_size, config.image_size))
   ```

2. in other phases

   ```
   from torchvision import transforms
   transforms.Resize((96, 96))  # or 256 when config.image_size = 224
   transforms.CenterCrop((84, 84)) # or 224 when config.image_size = 224
   ```

Besides, you may notice that `ToTensor & Norm` always uses the same sets of mean and variance, then you can reset mean and variance for different datasets.

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

CHAPTER

THIRTEEN

ADD A NEW BACKBONE

Code for this section

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

If you want to add a new backbone into `LibFewShot`, you should put all files about this new backbone in the directory of `core/model/backbone/`. For example, to add a `ResNet` to `LibFewShot`, you need provide a `resnet.py` in the directory of `core/model/backbone/`, and provide a class or function that can return a ResNet model like following:

```
...

class ResNet(nn.Module):
        def __init(self,...):
...

def ResNet18():
        model = ResNet(BasicBlock, [2,2,2,2], **kwargs)
        return model
```

After that, to make sure `trainer.py` could call `ResNet18`, you need add a line in `core/model/backbone/__init__.py` as follows:

```
...

from resnet import ResNet18
```

At this point, the addition of a new backbone is finished.

The new backbone shares the same way to use as other backbones. For example, to change `DN4` backbone to the new backbone, you just modify `backbone`'s value in `config/dn4.yaml` as follows:

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

CHAPTER

FOURTEEN

ADD A NEW CLASSIFIER

Code for this section

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

We need to select one representative method from `matric based` methods, `meta learning` methods and `fine-tuning` methods, respectively, and describe how to add new methods of the three categories.

Before thiswe need to introduce a parent class of all methods: `abstract_model`.

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

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

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

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

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

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

    def _init_network(self,):
        # init all layers

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

(continues on next page)

(continued from previous page)

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

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

- `__init__`init funcused to initialize the few shot learning settings like way, shot, query and other train parameters.
- `set_forward`used to be called in inference phase, return classifier's output and accuracy.
- `set_forward_loss`used to be called in training phase, return classifier's output, accuracy and loss.
- `forward`override the forward function `forward` of `Module` in `pytorch`, return the ouput of `backbone`.
- `train`override the forward function `train` of `Module` in `pytorch`, used to unfix the `BatchNorm` layer parameter.
- `eval`override the forward function `test` of `Module` in `pytorch`, used to fix the `BatchNorm` layer parameter.
- `_init_network`used to initialize all network parameters.
- `_generate_local_targets`used to generate `target` for few shot learning.
- `split_by_episode`used to split batch in shape:[episode_size, way, shot+query, ...]. It has several split modes.
- `reset_base_info`used to change the few shot learning settings.

New methods must override the `set_forward` and `set_forward_loss` functions, and all other functions can be called according to the needs of the implemented methods.

Note that in order for the newly added method to be called through reflection, add a line to the `__init__.py` file in the directory of the corresponding method type:

```
from NewMethodFileName import *
```

14.1 metric based

Using `DN4` as an example, we will describe how to add a new `metric based classifier` to `LibFewShot`.

`metric based` methods have a common parent class `MetricModel`, which is inherited from `AbstractModel`.

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

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

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

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

Since the `pipeline` of `metric based` methods are mostly simple, `MetricModel` just inherites `AbstractModel` and no other changes are made.

14.1.1 build model

First, create `DN4` model class, add file `dn4.py` under `core/model/metric/`: (this code have some differences with source code)

```
class DN4(MetricModel):
    def __init__(self, n_k=3, **kwargs):
        # base info
        super(DN4Layer, self).__init__(**kwargs)
        self.n_k = n_k
        self.loss_func = nn.CrossEntropyLoss()

    def set_forward(self, batch):
        # inference phase
        """
        :param batch: (images, labels)
        :param batch.images: shape: [episodeSize*way*(shot*augment_times+query*augment_
↪times_query),C,H,W]
        :param batch.labels: shape: [episodeSize*way*(shot*augment_times+query*augment_
↪times_query), ]
        :return: net output and accuracy
        """
        image, global_target = batch
        image = image.to(self.device)
        episode_size = image.size(0) // (
            self.way_num * (self.shot_num + self.query_num)
        )
        feat = self.emb_func(image)
        support_feat, query_feat, support_target, query_target = self.split_by_episode(
            feat, mode=2
        )

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

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

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

(continues on next page)

(continued from previous page)

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

        output = score.view(episode_size * self.way_num * self.query_num, self.way_num)
        acc = accuracy(output, query_target)

        return output, acc

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
        output = score.view(episode_size * self.way_num * self.query_num, self.way_num)
        loss = self.loss_func(output, query_target)
        acc = accuracy(output, query_target)

        return output, acc, loss
```

`__init__` function call `super.__init__()` to initialize few shot learning settings, and initialize `DN4` method's super parameter `n_k`.

Please notice line `19-27,65-73`, these lines aim to split batch feature vectors into correct shape that fit few shot learning setting. In deatils, in order to maximize the useage of computing resources, we first get all images' feature vectors, and then divide the feature vectors into `support set`, `suery set`. `29-50` lines are used to calculate DN4 method's output. Finally, the ouput shape of `set_forward` is $output.shape:[episode_size*way*query,way]acc:float$, the output shape of `set_forward_loss` is $output.shape:[episode_size*way*query,way], acc:float, loss:tensor$. Where `output` needs to be cabculated according to the method, `acc` can call the `accuracy` function provided by `LibFewShot` and input `output, target` to get the classification accuracy.While `loss` can use the loss function that the user initializes at the start of the method, used in `set_forward_loss` to get the classification loss.

The metric based method simply needs to process the input images into the corresponding form according to the method, and then begin the training.

14.2 meta learning

Using `MAML` as an example, we will describe how to add a new `meta learning classifier` to `LibFewShot`.

`meta learning` methods have a common parent class `MetaModel`, which is inherited from `AbstractModel`.

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

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

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

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

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

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

        if config["kwargs"] is not None:
```

(continues on next page)

(continued from previous page)

```
            kwargs.update(config["kwargs"])
        return getattr(torch.optim, config["name"])(parameters, **kwargs)
```

The meta-learning method adds two new functions, `set_forward_adaptation` and `sub_optimizer`. `set_forward_adaptation` is the logic that deals with the need to fine-tune the network during the classification process, and `sub_optimizer` is to provide a new sub-optimizer for the fine-tuning.

14.2.1 build model

First, create `MAML` model class, add file `maml.py` under `core/model/meta/`: (this code have some differences with source code)

```
from ..backbone.utils import convert_maml_module

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

        convert_maml_module(self)

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

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

        output_list = []
        for i in range(episode_size):
            episode_support_image = support_image[i].contiguous().reshape(-1, c, h, w)
```

(continues on next page)

(continued from previous page)

```
            episode_query_image = query_image[i].contiguous().reshape(-1, c, h, w)
            episode_support_target = support_target[i].reshape(-1)
            self.set_forward_adaptation(episode_support_image, episode_support_target)

            output = self.forward_output(episode_query_image)

            output_list.append(output)

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

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

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

            output = self.forward_output(episode_query_image)

            output_list.append(output)

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

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

        self.emb_func.train()
        self.classifier.train()
```

(continues on next page)

(continued from previous page)

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

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

The most important parts of `MAML` are the two parts. The first part is the `convert_maml_module` function on line `10`, which changes all the layers in the network to MAML format layers for easy parameter updating. The other part is the `set_forward_adaptation` function, which updates the fast parameters of the network. `MAML` is a common meta learning method, so we will use MAML as an example to show how to add `meta learning` method to `LibFewShot`.

14.3 fine-tuning

Using `Baseline` as an example, we will describe how to add a new `fine-tuning classifier` to `LibFewShot`.

`fine-tuning` methods have a common parent class `FinetuningModel`, which is inherited from `AbstractModel`.

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

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

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

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

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

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

The main aim of finetuning method train phase is to train a good feature extractor, while using the few shot learning

setting in the test phase to finetune the model by the support set. Another method is to use the training setting of few shot learning to fine-tune the whole model after the feature extractor is trained. In line with the `meta learning` method, a `set_forward_adaptation` abstract function is added to handle the forward process during test phase. In addition, since there are some `fine-tuning` methods in which the classifier needs to be trained, a `sub_optimizer` method is added, passing in the parameters to be optimized and the optimized configuration parameters, and returning the optimizer for easy call.

14.3.1 build model

First, create `Baseline` model class, add file `baseline.py` under `core/model/finetuning/`: (this code have some differences with source code)

```
class Baseline(FinetuningModel):
    def __init__(self, feat_dim, num_class, inner_param, **kwargs):
        super(Baseline, self).__init__(**kwargs)
        self.feat_dim = feat_dim
        self.num_class = num_class
        self.inner_param = inner_param

        self.classifier = nn.Linear(self.feat_dim, self.num_class)
        self.loss_func = nn.CrossEntropyLoss()

    def set_forward(self, batch):
        """
        :param batch: (images, labels)
        :param batch.images: shape: [episodeSize*way*(shot*augment_times+query*augment_
↪times_query),C,H,W]
        :param batch.labels: shape: [episodeSize*way*(shot*augment_times+query*augment_
↪times_query), ]
        :return: net output, accuracy and train loss
        """
        image, global_target = batch
        image = image.to(self.device)
        feat = self.emb_func(image)

        support_feat, query_feat, support_target, query_target = self.split_by_
↪episode(feat, mode=1)
        episode_size = support_feat.size(0)

        support_target = support_target.reshape(episode_size, self.way_num, self.shot_
↪num)
        query_target = query_target.reshape(episode_size, self.way_num, self.query_num)

        output_list = []
        for i in range(episode_size):
            output = self.set_forward_adaptation(support_feat, support_target, query_
↪feat)
            output_list.append(output)

        output = torch.stack(output_list, dim=0)
        acc = accuracy(output, query_target)

        return output, acc
```

(continues on next page)

(continued from previous page)

```
    def set_forward_loss(self, batch):
        """
        :param batch: (images, labels)
        :param batch.images: shape: [batch_size,C,H,W]
        :param batch.labels: shape: [batch_size, ]
        :return: net output, accuracy and train loss
        """
        image, target = batch
        image = image.to(self.device)
        target = target.to(self.device)

        feat = self.emb_func(image)
        output = self.classifier(feat)
        loss = self.loss_func(output, target)
        acc = accuracy(output, target)
        return output, acc, loss

    def set_forward_adaptation(self, support_feat, support_target, query_feat):
        """
        support_feat: shape: [way_num, shot_num, C]
        support_target: shape: [way_num*shot_num, ]
        query_feat: shape: [way_num, shot_num, C]
        """
        classifier = nn.Linear(self.feat_dim, self.way_num)
        optimizer = self.sub_optimizer(classifier, self.inner_param["inner_optim"])

        classifier = classifier.to(self.device)

        classifier.train()
        support_size = support_feat.size(0)
        for epoch in range(self.inner_param["inner_train_iter"]):
            rand_id = torch.randperm(support_size)
            for i in range(0, support_size, self.inner_param["inner_batch_size"]):
                select_id = rand_id[i : min(i + self.inner_param["inner_batch_size"],␣
↪support_size)]
                batch = support_feat[select_id]
                target = support_target[select_id]

                output = classifier(batch)

                loss = self.loss_func(output, target)

                optimizer.zero_grad()
                loss.backward(retain_graph=True)
                optimizer.step()

        output = classifier(query_feat)
        return output
```

The `set_forward_loss` is the same as the classical supervised classification method, while the `set_forward` is the same as the `meta learning` method. The contents of the `set_forward_adaptation` function is the main part of the test phase. The feature vectors of `support set` and `query set` extracted by backbone is used to train a classifier,

and the feature vectors of `query set` is used to classify by the classifier.

CHAPTER

FIFTEEN

MODEL ZOO

Coming Soon.

This file will contains methods' precision results, provides checkpoint downloads and configuration file downloads, and even provides code parsing for each method.

CHAPTER

SIXTEEN

HOW TO CONTRIBUTE TO THIS REPOSITORY?

Feel free to contribute classifiers, backbones, functions and any enhancements.

16.1 Add a method/feature or fix a bug

We recommend using the following guidelines:

1. fork the `main` branch of the latest `LibFewShot`;
2. checkout a new branchwhose name should reflects the content intuitively, like `add-method-ProtoNet` of `fix-doc-contribution`;
3. add a new method/feature or fix a bug;
4. check and commit;
5. create a pull request.

Note that if you add a new method, you need:

1. test if the method works properly;
2. provide a config file of this new method, and the corresponding 5-way 1-shot and 5-way 5-shot accuracy on the miniImageNet dataset.

Also, it will be better if you can provide:

1. the 5-way 1-shot and 5-way 5-shot accuracy on other datasets (like tieredImageNet);
2. `model_best.pth` of each setting on each dataset.

We will thank you for your contributions in `README` or other prominent places.

16.2 Use `pre-commit` to check code

Before committing the code, you may need to make sure that your code could pass black and flake test. We use **pre-commit** to do test and automatic code revision:

1. first, install pre-commit;

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

1. run `pre-commit install`;
2. run `pre-commit run --all-files;`

3. modify the code by the warning gived by pre-commit.

16.3 Pull request style

The title of your PR should like followings:

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

The body of your PR should describe the main content of this PR in EN OR CN.

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search