MMClassification Documentation

Release 0.18.0

MMClassification Contributors

Apr 07, 2023

GET STARTED

You can switch between Chinese and English documentation in the lower-left corner of the layout.
您可以在页面左下角切换中英文文档。

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- Python 3.6+
- PyTorch 1.5+
- MMCV

The compatible MMClassification and MMCV versions are as below. Please install the correct version of MMCV to avoid installation issues.

Note: Since the `master` branch is under frequent development, the `mmcv` version dependency may be inaccurate. If you encounter problems when using the `master` branch, please try to update `mmcv` to the latest version.

1.2 Install MMClassification

a. Create a conda virtual environment and activate it.

```
conda create -n open-mmlab python=3.8 -y
conda activate open-mmlab
```

b. Install PyTorch and torchvision following the official instructions, e.g.,

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

Note: Make sure that your compilation CUDA version and runtime CUDA version match. You can check the supported CUDA version for precompiled packages on the PyTorch website.

`E.g.1` If you have CUDA 10.1 installed under `/usr/local/cuda` and would like to install PyTorch 1.5.1, you need to install the prebuilt PyTorch with CUDA 10.1.

```
conda install pytorch==1.5.1 torchvision==0.6.1 cudatoolkit=10.1 -c pytorch
```

`E.g.2` If you have CUDA 11.3 installed under `/usr/local/cuda` and would like to install PyTorch 1.10.0., you need to install the prebuilt PyTorch with CUDA 11.3.

```
conda install pytorch==1.10.0 torchvision==0.11.1 cudatoolkit=11.3 -c pytorch
```

If you build PyTorch from source instead of installing the prebuilt package, you can use more CUDA versions such as 9.0.

c. Install MMClassification repository.

1.2.1 Release version

We recommend you to install MMClassification with MIM.

```
pip install git+https://github.com/open-mmlab/mim.git
mim install mmcls
```

MIM can automatically install OpenMMLab projects and their requirements, and it can also help us to train, parameter search and pretrain model download.

Or, you can install MMClassification with pip:

```
pip install mmcls
```

1.2.2 Develop version

First, clone the MMClassification repository.

```
git clone https://github.com/open-mmlab/mmclassification.git
cd mmclassification
```

And then, install build requirements and install MMClassification.

```
pip install -e .  # or "python setup.py develop"
```

Note: Following above instructions, MMClassification is installed on `dev` mode, any local modifications made to the code will take effect without the need to reinstall it (unless you submit some commits and want to update the version number).

1.2.3 Another option: Docker Image

We provide a Dockerfile to build an image.

```
# build an image with PyTorch 1.6.0, CUDA 10.1, CUDNN 7.
docker build -f ./docker/Dockerfile --rm -t mmcls:torch1.6.0-cuda10.1-cudnn7 .
```

Important: Make sure you've installed the nvidia-container-toolkit.

Run a container built from mmcls image with command:

```
docker run --gpus all --shm-size=8g -it -v {DATA_DIR}:/workspace/mmclassification/data␣
↪mmcls:torch1.6.0-cuda10.1-cudnn7 /bin/bash
```

1.3 Using multiple MMClassification versions

The train and test scripts already modify the `PYTHONPATH` to ensure the script use the MMClassification in the current directory.

To use the default MMClassification installed in the environment rather than that you are working with, you can remove the following line in those scripts

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

CHAPTER

TWO

GETTING STARTED

This page provides basic tutorials about the usage of MMClassification.

2.1 Prepare datasets

It is recommended to symlink the dataset root to `$MMCLASSIFICATION/data`. If your folder structure is different, you may need to change the corresponding paths in config files.

```
mmclassification
├── mmcls
├── tools
├── configs
├── docs
├── data
│   ├── imagenet
│   │   ├── meta
│   │   ├── train
│   │   ├── val
│   ├── cifar
│   │   ├── cifar-10-batches-py
│   ├── mnist
│   │   ├── train-images-idx3-ubyte
│   │   ├── train-labels-idx1-ubyte
│   │   ├── t10k-images-idx3-ubyte
│   │   ├── t10k-labels-idx1-ubyte
```

For ImageNet, it has multiple versions, but the most commonly used one is ILSVRC 2012. It can be accessed with the following steps.

1. Register an account and login to the download page.
2. Find download links for ILSVRC2012 and download the following two files
   - ILSVRC2012_img_train.tar (~138GB)
   - ILSVRC2012_img_val.tar (~6.3GB)
3. Untar the downloaded files
4. Download meta data using this script

For MNIST, CIFAR10 and CIFAR100, the datasets will be downloaded and unzipped automatically if they are not found.

For using custom datasets, please refer to *Tutorials 2: Adding New Dataset*.

2.2 Inference with pretrained models

We provide scripts to inference a single image, inference a dataset and test a dataset (e.g., ImageNet).

2.2.1 Inference a single image

```
python demo/image_demo.py ${IMAGE_FILE} ${CONFIG_FILE} ${CHECKPOINT_FILE}

# Example
python demo/image_demo.py demo/demo.JPEG configs/resnet/resnet50_8xb32_in1k.py \
  https://download.openmmlab.com/mmclassification/v0/resnet/resnet50_8xb32_in1k_20210831-
↪ea4938fc.pth
```

2.2.2 Inference and test a dataset

- single GPU
- single node multiple GPU
- multiple node

You can use the following commands to infer a dataset.

```
# single-gpu
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--metrics ${METRICS}] [--out $
↪{RESULT_FILE}]

# multi-gpu
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM} [--metrics ${METRICS}]␣
↪[--out ${RESULT_FILE}]

# multi-node in slurm environment
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--metrics ${METRICS}] [--out $
↪{RESULT_FILE}] --launcher slurm
```

Optional arguments:

- `RESULT_FILE`: Filename of the output results. If not specified, the results will not be saved to a file. Support formats include json, yaml and pickle.
- `METRICS`: Items to be evaluated on the results, like accuracy, precision, recall, etc.

Examples:

Assume that you have already downloaded the checkpoints to the directory `checkpoints/`. Infer ResNet-50 on ImageNet validation set to get predicted labels and their corresponding predicted scores.

```
python tools/test.py configs/resnet/resnet50_8xb16_cifar10.py \
  https://download.openmmlab.com/mmclassification/v0/resnet/resnet50_b16x8_cifar10_
↪20210528-f54bfad9.pth \
  --out result.pkl
```

2.3 Train a model

MMClassification implements distributed training and non-distributed training, which uses `MMDistributedDataParallel` and `MMDataParallel` respectively.

All outputs (log files and checkpoints) will be saved to the working directory, which is specified by `work_dir` in the config file.

By default we evaluate the model on the validation set after each epoch, you can change the evaluation interval by adding the interval argument in the training config.

```
evaluation = dict(interval=12)  # Evaluate the model per 12 epochs.
```

2.3.1 Train with a single GPU

```
python tools/train.py ${CONFIG_FILE} [optional arguments]
```

If you want to specify the working directory in the command, you can add an argument `--work_dir ${YOUR_WORK_DIR}`.

2.3.2 Train with multiple GPUs

```
./tools/dist_train.sh ${CONFIG_FILE} ${GPU_NUM} [optional arguments]
```

Optional arguments are:

- `--no-validate` (**not suggested**): By default, the codebase will perform evaluation at every k (default value is 1) epochs during the training. To disable this behavior, use `--no-validate`.
- `--work-dir ${WORK_DIR}`: Override the working directory specified in the config file.
- `--resume-from ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.

Difference between `resume-from` and `load-from`: `resume-from` loads both the model weights and optimizer status, and the epoch is also inherited from the specified checkpoint. It is usually used for resuming the training process that is interrupted accidentally. `load-from` only loads the model weights and the training epoch starts from 0. It is usually used for finetuning.

2.3.3 Train with multiple machines

If you run MMClassification on a cluster managed with slurm, you can use the script `slurm_train.sh`. (This script also supports single machine training.)

```
[GPUS=${GPUS}] ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${WORK_DIR}
```

You can check slurm_train.sh for full arguments and environment variables.

If you have just multiple machines connected with ethernet, you can refer to PyTorch launch utility. Usually it is slow if you do not have high speed networking like InfiniBand.

2.3.4 Launch multiple jobs on a single machine

If you launch multiple jobs on a single machine, e.g., 2 jobs of 4-GPU training on a machine with 8 GPUs, you need to specify different ports (29500 by default) for each job to avoid communication conflict.

If you use `dist_train.sh` to launch training jobs, you can set the port in commands.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 PORT=29500 ./tools/dist_train.sh ${CONFIG_FILE} 4
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 ./tools/dist_train.sh ${CONFIG_FILE} 4
```

If you use launch training jobs with Slurm, you need to modify the config files (usually the 6th line from the bottom in config files) to set different communication ports.

In `config1.py`,

```
dist_params = dict(backend='nccl', port=29500)
```

In `config2.py`,

```
dist_params = dict(backend='nccl', port=29501)
```

Then you can launch two jobs with `config1.py` ang `config2.py`.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py ${WORK_DIR}
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py ${WORK_DIR}
```

2.4 Useful tools

We provide lots of useful tools under `tools/` directory.

2.4.1 Get the FLOPs and params (experimental)

We provide a script adapted from flops-counter.pytorch to compute the FLOPs and params of a given model.

```
python tools/get_flops.py ${CONFIG_FILE} [--shape ${INPUT_SHAPE}]
```

You will get the result like this.

```
==============================
Input shape: (3, 224, 224)
Flops: 4.12 GFLOPs
Params: 25.56 M
==============================
```

> **Warning:** This tool is still experimental and we do not guarantee that the number is correct. You may well use the result for simple comparisons, but double check it before you adopt it in technical reports or papers.
>
> - FLOPs are related to the input shape while parameters are not. The default input shape is (1, 3, 224, 224).

- Some operators are not counted into FLOPs like GN and custom operators. Refer to `mmcv.cnn.get_model_complexity_info()` for details.

2.4.2 Publish a model

Before you publish a model, you may want to

1. Convert model weights to CPU tensors.
2. Delete the optimizer states.
3. Compute the hash of the checkpoint file and append the hash id to the filename.

```
python tools/convert_models/publish_model.py ${INPUT_FILENAME} ${OUTPUT_FILENAME}
```

E.g.,

```
python tools/convert_models/publish_model.py work_dirs/resnet50/latest.pth imagenet_
↪resnet50.pth
```

The final output filename will be `imagenet_resnet50_{date}-{hash id}.pth`.

2.5 Tutorials

Currently, we provide five tutorials for users.

- *learn about config*
- *finetune models*
- *add new dataset*
- *design data pipeline*
- *add new modules*
- *customize schedule*
- *customize runtime settings*.

CHAPTER

THREE

TUTORIAL 1: LEARN ABOUT CONFIGS

MMClassification mainly uses python files as configs. The design of our configuration file system integrates modularity and inheritance, facilitating users to conduct various experiments. All configuration files are placed in the `configs` folder, which mainly contains the primitive configuration folder of `_base_` and many algorithm folders such as `resnet`, `swin_transformer`, `vision_transformer`, etc.

If you wish to inspect the config file, you may run `python tools/analysis/print_config.py /PATH/TO/CONFIG` to see the complete config.

- Config File and Checkpoint Naming Convention
- Config File Structure
- Inherit and Modify Config File
  - Use intermediate variables in configs
  - Ignore some fields in the base configs
  - Use some fields in the base configs
- Modify config through script arguments
- Import user-defined modules
- FAQ

3.1 Config File and Checkpoint Naming Convention

We follow the below convention to name config files. Contributors are advised to follow the same style. The config file names are divided into four parts: algorithm info, module information, training information and data information. Logically, different parts are concatenated by underscores `'_'`, and words in the same part are concatenated by dashes `'-'`.

```
{algorithm info}_{module info}_{training info}_{data info}.py
```

- `algorithm info`: algorithm information, model name and neural network architecture, such as resnet, etc.;
- `module info`: module information is used to represent some special neck, head and pretrain information;
- `training info`: Training information, some training schedule, including batch size, lr schedule, data augment and the like;
- `data info`: Data information, dataset name, input size and so on, such as imagenet, cifar, etc.;

3.1.1 Algorithm information

The main algorithm name and the corresponding branch architecture information. E.g:

- `resnet50`
- `mobilenet-v3-large`
- `vit-small-patch32` : `patch32` represents the size of the partition in `ViT` algorithm;
- `seresnext101-32x4d` : `SeResNet101` network structure, `32x4d` means that `groups` and `width_per_group` are 32 and 4 respectively in `Bottleneck`;

3.1.2 Module information

Some special `neck`, `head` and `pretrain` information. In classification tasks, `pretrain` information is the most commonly used:

- `in21k-pre` : pre-trained on ImageNet21k;
- `in21k-pre-3rd-party` : pre-trained on ImageNet21k and the checkpoint is converted from a third-party repository;

3.1.3 Training information

Training schedule, including training type, `batch size`, `lr schedule`, data augment, special loss functions and so on:

- format `{gpu x batch_per_gpu}`, such as `8xb32`

Training type (mainly seen in the transformer network, such as the `ViT` algorithm, which is usually divided into two training type: pre-training and fine-tuning):

- `ft` : configuration file for fine-tuning
- `pt` : configuration file for pretraining

Training recipe. Usually, only the part that is different from the original paper will be marked. These methods will be arranged in the order `{pipeline aug}-{train aug}-{loss trick}-{scheduler}-{epochs}`.

- `coslr-200e` : use cosine scheduler to train 200 epochs
- `autoaug-mixup-lbs-coslr-50e` : use `autoaug`, `mixup`, `label smooth`, `cosine scheduler` to train 50 epochs

3.1.4 Data information

- `in1k` : `ImageNet1k` dataset, default to use the input image size of 224x224;
- `in21k` : `ImageNet21k` dataset, also called `ImageNet22k` dataset, default to use the input image size of 224x224;
- `in1k-384px` : Indicates that the input image size is 384x384;
- `cifar100`

3.1.5 Config File Name Example

```
repvgg-D2se_deploy_4xb64-autoaug-lbs-mixup-coslr-200e_in1k.py
```

- `repvgg-D2se`: Algorithm information
  - `repvgg`: The main algorithm.
  - `D2se`: The architecture.
- `deploy`: Module information, means the backbone is in the deploy state.
- `4xb64-autoaug-lbs-mixup-coslr-200e`: Training information.
  - `4xb64`: Use 4 GPUs and the size of batches per GPU is 64.
  - `autoaug`: Use `AutoAugment` in training pipeline.
  - `lbs`: Use label smoothing loss.
  - `mixup`: Use `mixup` training augment method.
  - `coslr`: Use cosine learning rate scheduler.
  - `200e`: Train the model for 200 epochs.
- `in1k`: Dataset information. The config is for `ImageNet1k` dataset and the input size is `224x224`.

Note: Some configuration files currently do not follow this naming convention, and related files will be updated in the near future.

3.1.6 Checkpoint Naming Convention

The naming of the weight mainly includes the configuration file name, date and hash value.

```
{config_name}_{date}-{hash}.pth
```

3.2 Config File Structure

There are four kinds of basic component file in the `configs/_base_` folders, namely:

- models
- datasets
- schedules
- runtime

You can easily build your own training config file by inherit some base config files. And the configs that are composed by components from `_base_` are called *primitive*.

For easy understanding, we use ResNet50 primitive config as a example and comment the meaning of each line. For more detaile, please refer to the API documentation.

```
_base_ = [
    '../_base_/models/resnet50.py',           # model
    '../_base_/datasets/imagenet_bs32.py',    # data
    '../_base_/schedules/imagenet_bs256.py',  # training schedule
    '../_base_/default_runtime.py'            # runtime setting
]
```

The four parts are explained separately below, and the above-mentioned ResNet50 primitive config are also used as an example.

3.2.1 model

The parameter `"model"` is a python dictionary in the configuration file, which mainly includes information such as network structure and loss function:

- `type` : Classifier name, MMCls supports `ImageClassifier`, refer to API documentation.
- `backbone` : Backbone configs, refer to API documentation for available options.
- `neck` : Neck network name, MMCls supports `GlobalAveragePooling`, please refer to API documentation.
- `head`: Head network name, MMCls supports single-label and multi-label classification head networks, available options refer to API documentation.
  - `loss`: Loss function type, supports `CrossEntropyLoss`, `LabelSmoothLoss` etc., For available options, refer to API documentation.
- `train_cfg` : Training augment config, MMCls supports `mixup`, `cutmix` and other augments.

Note: The 'type' in the configuration file is not a constructed parameter, but a class name.

```
model = dict(
    type='ImageClassifier',     # Classifier name
    backbone=dict(
        type='ResNet',          # Backbones name
        depth=50,               # depth of backbone, ResNet has options of 18, 34, 50,␣
↪101, 152.
        num_stages=4,           # number of stages, The feature maps generated by these␣
↪states are used as the input for the subsequent neck and head.
        out_indices=(3, ),      # The output index of the output feature maps.
        frozen_stages=-1,       # the stage to be frozen, '-1' means not be forzen
        style='pytorch'),        # The style of backbone, 'pytorch' means that stride 2␣
↪layers are in 3x3 conv, 'caffe' means stride 2 layers are in 1x1 convs.
    neck=dict(type='GlobalAveragePooling'),    # neck network name
    head=dict(
        type='LinearClsHead',     # linear classification head,
        num_classes=1000,         # The number of output categories, consistent with the␣
↪number of categories in the dataset
        in_channels=2048,         # The number of input channels, consistent with the␣
↪output channel of the neck
        loss=dict(type='CrossEntropyLoss', loss_weight=1.0), # Loss function␣
↪configuration information
        topk=(1, 5),              # Evaluation index, Top-k accuracy rate, here is the␣
↪accuracy rate of top1 and top5
```

(continues on next page)

(continued from previous page)

```
    ))
```

3.2.2 data

The parameter "data" is a python dictionary in the configuration file, which mainly includes information to construct dataloader:

- samples_per_gpu : the BatchSize of each GPU when building the dataloader
- workers_per_gpu : the number of threads per GPU when building dataloader
- train | val | test : config to construct dataset
  - type: Dataset name, MMCls supports ImageNet, Cifar etc., refer to API documentation
  - data_prefix : Dataset root directory
  - pipeline : Data processing pipeline, refer to related tutorial CUSTOM DATA PIPELINES

The parameter evaluation is also a dictionary, which is the configuration information of evaluation hook, mainly including evaluation interval, evaluation index, etc..

```
# dataset settings
dataset_type = 'ImageNet'  # dataset name,
img_norm_cfg = dict(        # Image normalization config to normalize the input images
    mean=[123.675, 116.28, 103.53],  # Mean values used to pre-training the pre-trained␣
↪backbone models
    std=[58.395, 57.12, 57.375],     # Standard variance used to pre-training the pre-
↪trained backbone models
    to_rgb=True)                     # Whether to invert the color channel, rgb2bgr or␣
↪bgr2rgb.
# train data pipeline
train_pipeline = [
    dict(type='LoadImageFromFile'),                # First pipeline to load images from␣
↪file path
    dict(type='RandomResizedCrop', size=224),      # RandomResizedCrop
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),  # Randomly flip the␣
↪picture horizontally with a probability of 0.5
    dict(type='Normalize', **img_norm_cfg),        # normalization
    dict(type='ImageToTensor', keys=['img']),      # convert image from numpy into torch.
↪Tensor
    dict(type='ToTensor', keys=['gt_label']),      # convert gt_label into torch.Tensor
    dict(type='Collect', keys=['img', 'gt_label']) # Pipeline that decides which keys in␣
↪the data should be passed to the detector
]
# test data pipeline
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='Resize', size=(256, -1)),
    dict(type='CenterCrop', crop_size=224),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='Collect', keys=['img'])             # do not pass gt_label while testing
]
```

(continues on next page)

(continued from previous page)

```
data = dict(
    samples_per_gpu=32,     # Batch size of a single GPU
    workers_per_gpu=2,      # Worker to pre-fetch data for each single GPU
    train=dict(  # Train dataset config
    train=dict(            # train data config
        type=dataset_type,                  # dataset name
        data_prefix='data/imagenet/train',  # Dataset root, when ann_file does not exist,
↪ the category information is automatically obtained from the root folder
        pipeline=train_pipeline),           # train data pipeline
    val=dict(              # val data config
        type=dataset_type,
        data_prefix='data/imagenet/val',
        ann_file='data/imagenet/meta/val.txt',   #  ann_file existes, the category␣
↪information is obtained from file
        pipeline=test_pipeline),
    test=dict(             # test data config
        type=dataset_type,
        data_prefix='data/imagenet/val',
        ann_file='data/imagenet/meta/val.txt',
        pipeline=test_pipeline))
evaluation = dict(       # The config to build the evaluation hook, refer to https://
↪github.com/open-mmlab/mmdetection/blob/master/mmdet/core/evaluation/eval_hooks.py#L7␣
↪for more details.
    interval=1,          # Evaluation interval
    metric='accuracy')   # Metrics used during evaluation
```

3.2.3 training schedule

Mainly include optimizer settings, `optimizer hook` settings, learning rate schedule and `runner` settings:

- `optimizer`: optimizer setting , support all optimizers in `pytorch`, refer to related mmcv documentation.
- `optimizer_config`: `optimizer hook` configuration file, such as setting gradient limit, refer to related mmcv code.
- `lr_config`: Learning rate scheduler, supports "CosineAnnealing", "Step", "Cyclic", etc. refer to related mmcv documentation for more options.
- `runner`: For `runner`, please refer to `mmcv` for `runner` introduction document.

```
# he configuration file used to build the optimizer, support all optimizers in PyTorch.
optimizer = dict(type='SGD',         # Optimizer type
                 lr=0.1,             # Learning rate of optimizers, see detail usages of␣
↪the parameters in the documentation of PyTorch
                 momentum=0.9,       # Momentum
                 weight_decay=0.0001) # Weight decay of SGD
# Config used to build the optimizer hook, refer to https://github.com/open-mmlab/mmcv/
↪blob/master/mmcv/runner/hooks/optimizer.py#L8 for implementation details.
optimizer_config = dict(grad_clip=None)  # Most of the methods do not use gradient clip
# Learning rate scheduler config used to register LrUpdater hook
lr_config = dict(policy='step',          # The policy of scheduler, also support␣
↪CosineAnnealing, Cyclic, etc. Refer to details of supported LrUpdater from https://
↪github.com/open-mmlab/mmcv/blob/master/mmcv/runner/hooks/lr_updater.py#L9.
```

(continues on next page)

(continued from previous page)

```
                 step=[30, 60, 90])      # Steps to decay the learning rate
runner = dict(type='EpochBasedRunner',   # Type of runner to use (i.e. IterBasedRunner␣
↪or EpochBasedRunner)
            max_epochs=100)    # Runner that runs the workflow in total max_epochs. For␣
↪IterBasedRunner use `max_iters`
```

3.2.4 runtime setting

This part mainly includes saving the checkpoint strategy, log configuration, training parameters, breakpoint weight path, working directory, etc..

```
# Config to set the checkpoint hook, Refer to https://github.com/open-mmlab/mmcv/blob/
↪master/mmcv/runner/hooks/checkpoint.py for implementation.
checkpoint_config = dict(interval=1)    # The save interval is 1
# config to register logger hook
log_config = dict(
    interval=100,                       # Interval to print the log
    hooks=[
        dict(type='TextLoggerHook'),           # The Tensorboard logger is also supported
        # dict(type='TensorboardLoggerHook')
    ])

dist_params = dict(backend='nccl')   # Parameters to setup distributed training, the␣
↪port can also be set.
log_level = 'INFO'             # The output level of the log.
resume_from = None             # Resume checkpoints from a given path, the training will␣
↪be resumed from the epoch when the checkpoint's is saved.
workflow = [('train', 1)]      # Workflow for runner. [('train', 1)] means there is only␣
↪one workflow and the workflow named 'train' is executed once.
work_dir = 'work_dir'          # Directory to save the model checkpoints and logs for␣
↪the current experiments.
```

3.3 Inherit and Modify Config File

For easy understanding, we recommend contributors to inherit from existing methods.

For all configs under the same folder, it is recommended to have only **one** *primitive* config. All other configs should inherit from the *primitive* config. In this way, the maximum of inheritance level is 3.

For example, if your config file is based on ResNet with some other modification, you can first inherit the basic ResNet structure, dataset and other training setting by specifying `_base_ ='./resnet50_8xb32_in1k.py'` (The path relative to your config file), and then modify the necessary parameters in the config file. A more specific example, now we want to use almost all configs in `configs/resnet/resnet50_8xb32_in1k.py`, but change the number of training epochs from 100 to 300, modify when to decay the learning rate, and modify the dataset path, you can create a new config file `configs/resnet/resnet50_8xb32-300e_in1k.py` with content as below:

```
_base_ = './resnet50_8xb32_in1k.py'

runner = dict(max_epochs=300)
lr_config = dict(step=[150, 200, 250])
```

(continues on next page)

(continued from previous page)

```
data = dict(
    train=dict(data_prefix='mydata/imagenet/train'),
    val=dict(data_prefix='mydata/imagenet/train', ),
    test=dict(data_prefix='mydata/imagenet/train', )
)
```

3.3.1 Use intermediate variables in configs

Some intermediate variables are used in the configuration file. The intermediate variables make the configuration file clearer and easier to modify.

For example, `train_pipeline` / `test_pipeline` is the intermediate variable of the data pipeline. We first need to define `train_pipeline` / `test_pipeline`, and then pass them to `data`. If you want to modify the size of the input image during training and testing, you need to modify the intermediate variables of `train_pipeline` / `test_pipeline`.

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', size=384, backend='pillow',),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label'])
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='Resize', size=384, backend='pillow'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='Collect', keys=['img'])
]
data = dict(
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline))
```

3.3.2 Ignore some fields in the base configs

Sometimes, you need to set `_delete_=True` to ignore some domain content in the basic configuration file. You can refer to mmcv for more instructions.

The following is an example. If you wangt to use cosine schedule in the above ResNet50 case, just using inheritance and directly modify it will report `get unexcepected keyword'step'` error, because the `'step'` field of the basic config in `lr_config` domain information is reserved, and you need to add `_delete_ =True` to ignore the content of `lr_config` related fields in the basic configuration file:

```
_base_ = '../../configs/resnet/resnet50_8xb32_in1k.py'
```

(continues on next page)

(continued from previous page)

```
lr_config = dict(
    _delete_=True,
    policy='CosineAnnealing',
    min_lr=0,
    warmup='linear',
    by_epoch=True,
    warmup_iters=5,
    warmup_ratio=0.1
)
```

3.3.3 Use some fields in the base configs

Sometimes, you may refer to some fields in the `_base_` config, so as to avoid duplication of definitions. You can refer to mmcv for some more instructions.

The following is an example of using auto augment in the training data preprocessing pipeline， refer to `configs/_base_/datasets/imagenet_bs64_autoaug.py`. When defining `train_pipeline`, just add the definition file name of auto augment to `_base_`, and then use `{{_base_.auto_increasing_policies}}` to reference the variables:

```
_base_ = ['./pipelines/auto_aug.py']

# dataset settings
dataset_type = 'ImageNet'
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', size=224),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='AutoAugment', policies={{_base_.auto_increasing_policies}}),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label'])
]
test_pipeline = [...]
data = dict(
    samples_per_gpu=64,
    workers_per_gpu=2,
    train=dict(..., pipeline=train_pipeline),
    val=dict(..., pipeline=test_pipeline))
evaluation = dict(interval=1, metric='accuracy')
```

3.4 Modify config through script arguments

When users use the script "tools/train.py" or "tools/test.py" to submit tasks or use some other tools, they can directly modify the content of the configuration file used by specifying the `--cfg-options` parameter.

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options model.backbone.norm_eval=False` changes the all BN modules in model backbones to `train` mode.

- Update keys inside a list of configs.

  Some config dicts are composed as a list in your config. For example, the training pipeline `data.train.pipeline` is normally a list e.g. `[dict(type='LoadImageFromFile'), dict(type='TopDownRandomFlip', flip_prob=0.5), ...]`. If you want to change `'flip_prob=0.5'` to `'flip_prob=0.0'` in the pipeline, you may specify `--cfg-options data.train.pipeline.1.flip_prob=0.0`.

- Update values of list/tuples.

  If the value to be updated is a list or a tuple. For example, the config file normally sets `workflow=[('train', 1)]`. If you want to change this key, you may specify `--cfg-options workflow="[(train,1),(val,1)]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

3.5 Import user-defined modules

Note: This part may only be used when using MMClassification as a third party library to build your own project, and beginners can skip it.

After studying the follow-up tutorials ADDING NEW DATASET, CUSTOM DATA PIPELINES, ADDING NEW MODULES. You may use MMClassification to complete your project and create new classes of datasets, models, data enhancements, etc. in the project. In order to streamline the code, you can use MMClassification as a third-party library, you just need to keep your own extra code and import your own custom module in the configuration files. For examples, you may refer to OpenMMLab Algorithm Competition Project .

Add the following code to your own configuration files:

```
custom_imports = dict(
    imports=['your_dataset_class',
             'your_transforme_class',
             'your_model_class',
             'your_module_class'],
    allow_failed_imports=False)
```

3.6 FAQ

- None

CHAPTER

FOUR

TUTORIAL 2: FINE-TUNE MODELS

Classification models pre-trained on the ImageNet dataset have been demonstrated to be effective for other datasets and other downstream tasks. This tutorial provides instructions for users to use the models provided in the *Model Zoo* for other datasets to obtain better performance.

There are two steps to fine-tune a model on a new dataset.

- Add support for the new dataset following *Tutorial 2: Adding New Dataset*.
- Modify the configs as will be discussed in this tutorial.

Assume we have a ResNet-50 model pre-trained on the ImageNet-2012 dataset and want to take the fine-tuning on the CIFAR-10 dataset, we need to modify five parts in the config.

4.1 Inherit base configs

At first, create a new config file `configs/tutorial/resnet50_finetune_cifar.py` to store our configs. Of course, the path can be customized by yourself.

To reuse the common parts among different configs, we support inheriting configs from multiple existing configs. To fine-tune a ResNet-50 model, the new config needs to inherit `configs/_base_/models/resnet50.py` to build the basic structure of the model. To use the CIFAR-10 dataset, the new config can also simply inherit `configs/_base_/datasets/cifar10_bs16.py`. For runtime settings such as training schedules, the new config needs to inherit `configs/_base_/default_runtime.py`.

To inherit all above configs, put the following code at the config file.

```
_base_ = [
    '../_base_/models/resnet50.py',
    '../_base_/datasets/cifar10_bs16.py', '../_base_/default_runtime.py'
]
```

Besides, you can also choose to write the whole contents rather than use inheritance, like `configs/lenet/lenet5_mnist.py`.

4.2 Modify model

When fine-tuning a model, usually we want to load the pre-trained backbone weights and train a new classification head.

To load the pre-trained backbone, we need to change the initialization config of the backbone and use `Pretrained` initialization function. Besides, in the `init_cfg`, we use `prefix='backbone'` to tell the initialization function to remove the prefix of keys in the checkpoint, for example, it will change `backbone.conv1` to `conv1`. And here we use an online checkpoint, it will be downloaded during training, you can also download the model manually and use a local path.

And then we need to modify the head according to the class numbers of the new datasets by just changing `num_classes` in the head.

```
model = dict(
    backbone=dict(
        init_cfg=dict(
            type='Pretrained',
            checkpoint='https://download.openmmlab.com/mmclassification/v0/resnet/
↪resnet50_8xb32_in1k_20210831-ea4938fc.pth',
            prefix='backbone',
        )),
    head=dict(num_classes=10),
)
```

Tip: Here we only need to set the part of configs we want to modify, because the inherited configs will be merged and get the entire configs.

Sometimes, we want to freeze the first several layers' parameters of the backbone, that will help the network to keep ability to extract low-level information learnt from pre-trained model. In MMClassification, you can simply specify how many layers to freeze by `frozen_stages` argument. For example, to freeze the first two layers' parameters, just use the following config:

```
model = dict(
    backbone=dict(
        frozen_stages=2,
        init_cfg=dict(
            type='Pretrained',
            checkpoint='https://download.openmmlab.com/mmclassification/v0/resnet/
↪resnet50_8xb32_in1k_20210831-ea4938fc.pth',
            prefix='backbone',
        )),
    head=dict(num_classes=10),
)
```

Note: Not all backbones support the `frozen_stages` argument by now. Please check the docs to confirm if your backbone supports it.

4.3 Modify dataset

When fine-tuning on a new dataset, usually we need to modify some dataset configs. Here, we need to modify the pipeline to resize the image from 32 to 224 to fit the input size of the model pre-trained on ImageNet, and some other configs.

```
img_norm_cfg = dict(
    mean=[125.307, 122.961, 113.8575],
    std=[51.5865, 50.847, 51.255],
    to_rgb=False,
)
train_pipeline = [
    dict(type='RandomCrop', size=32, padding=4),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='Resize', size=224),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label']),
]
test_pipeline = [
    dict(type='Resize', size=224),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='Collect', keys=['img']),
]
data = dict(
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline),
)
```

4.4 Modify training schedule

The fine-tuning hyper parameters vary from the default schedule. It usually requires smaller learning rate and less training epochs.

```
# lr is set for a batch size of 128
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optimizer_config = dict(grad_clip=None)
# learning policy
lr_config = dict(policy='step', step=[15])
runner = dict(type='EpochBasedRunner', max_epochs=200)
log_config = dict(interval=100)
```

4.5 Start Training

Now, we have finished the fine-tuning config file as following:

```
_base_ = [
    '../_base_/models/resnet50.py',
    '../_base_/datasets/cifar10_bs16.py', '../_base_/default_runtime.py'
]

# Model config
model = dict(
    backbone=dict(
        frozen_stages=2,
        init_cfg=dict(
            type='Pretrained',
            checkpoint='https://download.openmmlab.com/mmclassification/v0/resnet/
↪resnet50_8xb32_in1k_20210831-ea4938fc.pth',
            prefix='backbone',
        )),
    head=dict(num_classes=10),
)

# Dataset config
img_norm_cfg = dict(
    mean=[125.307, 122.961, 113.8575],
    std=[51.5865, 50.847, 51.255],
    to_rgb=False,
)
train_pipeline = [
    dict(type='RandomCrop', size=32, padding=4),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='Resize', size=224),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label']),
]
test_pipeline = [
    dict(type='Resize', size=224),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='Collect', keys=['img']),
]
data = dict(
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline),
)

# Training schedule config
# lr is set for a batch size of 128
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optimizer_config = dict(grad_clip=None)
```

(continues on next page)

(continued from previous page)

```
# learning policy
lr_config = dict(policy='step', step=[15])
runner = dict(type='EpochBasedRunner', max_epochs=200)
log_config = dict(interval=100)
```

Here we use 8 GPUs on your computer to train the model with the following command:

```
bash tools/dist_train.sh configs/tutorial/resnet50_finetune_cifar.py 8
```

Also, you can use only one GPU to train the model with the following command:

```
python tools/train.py configs/tutorial/resnet50_finetune_cifar.py
```

But wait, an important config need to be changed if using one GPU. We need to change the dataset config as following:

```
data = dict(
    samples_per_gpu=128,
    train=dict(pipeline=train_pipeline),
    val=dict(pipeline=test_pipeline),
    test=dict(pipeline=test_pipeline),
)
```

It's because our training schedule is for a batch size of 128. If using 8 GPUs, just use `samples_per_gpu=16` config in the base config file, and the total batch size will be 128. But if using one GPU, you need to change it to 128 manually to match the training schedule.

CHAPTER

FIVE

TUTORIAL 3: ADDING NEW DATASET

5.1 Customize datasets by reorganizing data

5.1.1 Reorganize dataset to existing format

The simplest way is to convert your dataset to existing dataset formats (ImageNet).

For training, it differentiates classes by folders. The directory of training data is as follows:

```
imagenet
├── ...
├── train
│   ├── n01440764
│   │   ├── n01440764_10026.JPEG
│   │   ├── n01440764_10027.JPEG
│   │   ├── ...
│   ├── ...
│   ├── n15075141
│   │   ├── n15075141_999.JPEG
│   │   ├── n15075141_9993.JPEG
│   │   ├── ...
```

For validation, we provide a annotation list. Each line of the list contrains a filename and its corresponding ground-truth labels. The format is as follows:

```
ILSVRC2012_val_00000001.JPEG 65
ILSVRC2012_val_00000002.JPEG 970
ILSVRC2012_val_00000003.JPEG 230
ILSVRC2012_val_00000004.JPEG 809
ILSVRC2012_val_00000005.JPEG 516
```

Note: The value of ground-truth labels should fall in range `[0, num_classes - 1]`.

5.1.2 An example of customized dataset

You can write a new Dataset class inherited from `BaseDataset`, and overwrite `load_annotations(self)`, like CIFAR10 and ImageNet. Typically, this function returns a list, where each sample is a dict, containing necessary data information, e.g., `img` and `gt_label`.

Assume we are going to implement a `Filelist` dataset, which takes filelists for both training and testing. The format of annotation list is as follows:

```
000001.jpg 0
000002.jpg 1
```

We can create a new dataset in `mmcls/datasets/filelist.py` to load the data.

```
import mmcv
import numpy as np

from .builder import DATASETS
from .base_dataset import BaseDataset


@DATASETS.register_module()
class Filelist(BaseDataset):

    def load_annotations(self):
        assert isinstance(self.ann_file, str)

        data_infos = []
        with open(self.ann_file) as f:
            samples = [x.strip().split(' ') for x in f.readlines()]
            for filename, gt_label in samples:
                info = {'img_prefix': self.data_prefix}
                info['img_info'] = {'filename': filename}
                info['gt_label'] = np.array(gt_label, dtype=np.int64)
                data_infos.append(info)
            return data_infos
```

And add this dataset class in `mmcls/datasets/__init__.py`

```
from .base_dataset import BaseDataset
...
from .filelist import Filelist

__all__ = [
    'BaseDataset', ... ,'Filelist'
]
```

Then in the config, to use `Filelist` you can modify the config as the following

```
train = dict(
    type='Filelist',
    ann_file = 'image_list.txt',
    pipeline=train_pipeline
)
```

5.2 Customize datasets by mixing dataset

MMClassification also supports to mix dataset for training. Currently it supports to concat and repeat datasets.

5.2.1 Repeat dataset

We use `RepeatDataset` as wrapper to repeat the dataset. For example, suppose the original dataset is `Dataset_A`, to repeat it, the config looks like the following

```
dataset_A_train = dict(
        type='RepeatDataset',
        times=N,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

5.2.2 Class balanced dataset

We use `ClassBalancedDataset` as wrapper to repeat the dataset based on category frequency. The dataset to repeat needs to instantiate function `self.get_cat_ids(idx)` to support `ClassBalancedDataset`. For example, to repeat `Dataset_A` with `oversample_thr=1e-3`, the config looks like the following

```
dataset_A_train = dict(
        type='ClassBalancedDataset',
        oversample_thr=1e-3,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

You may refer to source code for details.

CHAPTER

SIX

TUTORIAL 4: CUSTOM DATA PIPELINES

6.1 Design of Data pipelines

Following typical conventions, we use `Dataset` and `DataLoader` for data loading with multiple workers. Indexing `Dataset` returns a dict of data items corresponding to the arguments of models forward method.

The data preparation pipeline and the dataset is decomposed. Usually a dataset defines how to process the annotations and a data pipeline defines all the steps to prepare a data dict. A pipeline consists of a sequence of operations. Each operation takes a dict as input and also output a dict for the next transform.

The operations are categorized into data loading, pre-processing and formatting.

Here is an pipeline example for ResNet-50 training on ImageNet.

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', size=224),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label'])
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='Resize', size=256),
    dict(type='CenterCrop', crop_size=224),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='Collect', keys=['img'])
]
```

For each operation, we list the related dict fields that are added/updated/removed. At the end of the pipeline, we use `Collect` to only retain the necessary items for forward computation.

6.1.1 Data loading

`LoadImageFromFile`

- add: img, img_shape, ori_shape

By default, `LoadImageFromFile` loads images from disk but it may lead to IO bottleneck for efficient small models. Various backends are supported by mmcv to accelerate this process. For example, if the training machines have setup memcached, we can revise the config as follows.

```
memcached_root = '/mnt/xxx/memcached_client/'
train_pipeline = [
    dict(
        type='LoadImageFromFile',
        file_client_args=dict(
            backend='memcached',
            server_list_cfg=osp.join(memcached_root, 'server_list.conf'),
            client_cfg=osp.join(memcached_root, 'client.conf'))),
]
```

More supported backends can be found in mmcv.fileio.FileClient.

6.1.2 Pre-processing

`Resize`

- add: scale, scale_idx, pad_shape, scale_factor, keep_ratio
- update: img, img_shape

`RandomFlip`

- add: flip, flip_direction
- update: img

`RandomCrop`

- update: img, pad_shape

`Normalize`

- add: img_norm_cfg
- update: img

6.1.3 Formatting

`ToTensor`

- update: specified by `keys`.

`ImageToTensor`

- update: specified by `keys`.

`Collect`

- remove: all other keys except for those specified by `keys`

6.2 Extend and use custom pipelines

1. Write a new pipeline in any file, e.g., `my_pipeline.py`, and place it in the folder `mmcls/datasets/pipelines/`. The pipeline class needs to override the `__call__` method which takes a dict as input and returns a dict.

```
from mmcls.datasets import PIPELINES

@PIPELINES.register_module()
class MyTransform(object):

    def __call__(self, results):
        # apply transforms on results['img']
        return results
```

2. Import the new class in `mmcls/datasets/pipelines/__init__.py`.

```
...
from .my_pipeline import MyTransform

__all__ = [
    ..., 'MyTransform'
]
```

3. Use it in config files.

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='RandomResizedCrop', size=224),
    dict(type='RandomFlip', flip_prob=0.5, direction='horizontal'),
    dict(type='MyTransform'),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='ImageToTensor', keys=['img']),
    dict(type='ToTensor', keys=['gt_label']),
    dict(type='Collect', keys=['img', 'gt_label'])
]
```

6.3 Pipeline visualization

After designing data pipelines, you can use the *visualization tools* to view the performance.

CHAPTER

SEVEN

TUTORIAL 5: ADDING NEW MODULES

7.1 Develop new components

We basically categorize model components into 3 types.

- backbone: usually an feature extraction network, e.g., ResNet, MobileNet.
- neck: the component between backbones and heads, e.g., GlobalAveragePooling.
- head: the component for specific tasks, e.g., classification or regression.

7.1.1 Add new backbones

Here we show how to develop new components with an example of ResNet_CIFAR. As the input size of CIFAR is 32x32, this backbone replaces the `kernel_size=7, stride=2` to `kernel_size=3, stride=1` and remove the MaxPooling after stem, to avoid forwarding small feature maps to residual blocks. It inherits from ResNet and only modifies the stem layers.

1. Create a new file `mmcls/models/backbones/resnet_cifar.py`.

```
import torch.nn as nn

from ..builder import BACKBONES
from .resnet import ResNet


@BACKBONES.register_module()
class ResNet_CIFAR(ResNet):

    """ResNet backbone for CIFAR.

    short description of the backbone

    Args:
        depth(int): Network depth, from {18, 34, 50, 101, 152}.
        ...
    """

    def __init__(self, depth, deep_stem, **kwargs):
        # call ResNet init
        super(ResNet_CIFAR, self).__init__(depth, deep_stem=deep_stem, **kwargs)
        # other specific initialization
```

(continues on next page)

(continued from previous page)

```
        assert not self.deep_stem, 'ResNet_CIFAR do not support deep_stem'

    def _make_stem_layer(self, in_channels, base_channels):
        # override ResNet method to modify the network structure
        self.conv1 = build_conv_layer(
            self.conv_cfg,
            in_channels,
            base_channels,
            kernel_size=3,
            stride=1,
            padding=1,
            bias=False)
        self.norm1_name, norm1 = build_norm_layer(
            self.norm_cfg, base_channels, postfix=1)
        self.add_module(self.norm1_name, norm1)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):  # should return a tuple
        pass  # implementation is ignored

    def init_weights(self, pretrained=None):
        pass  # override ResNet init_weights if necessary

    def train(self, mode=True):
        pass  # override ResNet train if necessary
```

2. Import the module in `mmcls/models/backbones/__init__.py`.

```
...
from .resnet_cifar import ResNet_CIFAR

__all__ = [
    ..., 'ResNet_CIFAR'
]
```

3. Use it in your config file.

```
model = dict(
    ...
    backbone=dict(
        type='ResNet_CIFAR',
        depth=18,
        other_arg=xxx),
    ...
```

7.1.2 Add new necks

Here we take `GlobalAveragePooling` as an example. It is a very simple neck without any arguments. To add a new neck, we mainly implement the `forward` function, which applies some operation on the output from backbone and forward the results to head.

1. Create a new file in `mmcls/models/necks/gap.py`.

```
import torch.nn as nn

from ..builder import NECKS

@NECKS.register_module()
class GlobalAveragePooling(nn.Module):

    def __init__(self):
        self.gap = nn.AdaptiveAvgPool2d((1, 1))

    def forward(self, inputs):
        # we regard inputs as tensor for simplicity
        outs = self.gap(inputs)
        outs = outs.view(inputs.size(0), -1)
        return outs
```

2. Import the module in `mmcls/models/necks/__init__.py`.

```
...
from .gap import GlobalAveragePooling

__all__ = [
    ..., 'GlobalAveragePooling'
]
```

3. Modify the config file.

```
model = dict(
    neck=dict(type='GlobalAveragePooling'),
)
```

7.1.3 Add new heads

Here we show how to develop a new head with the example of `LinearClsHead` as the following. To implement a new head, basically we need to implement `forward_train`, which takes the feature maps from necks or backbones as input and compute loss based on ground-truth labels.

1. Create a new file in `mmcls/models/heads/linear_head.py`.

```
from ..builder import HEADS
from .cls_head import ClsHead


@HEADS.register_module()
class LinearClsHead(ClsHead):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 num_classes,
                 in_channels,
                 loss=dict(type='CrossEntropyLoss', loss_weight=1.0),
                 topk=(1, )):
        super(LinearClsHead, self).__init__(loss=loss, topk=topk)
        self.in_channels = in_channels
        self.num_classes = num_classes

        if self.num_classes <= 0:
            raise ValueError(
                f'num_classes={num_classes} must be a positive integer')

        self._init_layers()

    def _init_layers(self):
        self.fc = nn.Linear(self.in_channels, self.num_classes)

    def init_weights(self):
        normal_init(self.fc, mean=0, std=0.01, bias=0)

    def forward_train(self, x, gt_label):
        cls_score = self.fc(x)
        losses = self.loss(cls_score, gt_label)
        return losses
```

2. Import the module in `mmcls/models/heads/__init__.py`.

```
...
from .linear_head import LinearClsHead

__all__ = [
    ..., 'LinearClsHead'
]
```

3. Modify the config file.

Together with the added GlobalAveragePooling neck, an entire config for a model is as follows.

```
model = dict(
    type='ImageClassifier',
    backbone=dict(
        type='ResNet',
        depth=50,
        num_stages=4,
        out_indices=(3, ),
        style='pytorch'),
    neck=dict(type='GlobalAveragePooling'),
    head=dict(
        type='LinearClsHead',
        num_classes=1000,
        in_channels=2048,
```

(continues on next page)

(continued from previous page)

```
        loss=dict(type='CrossEntropyLoss', loss_weight=1.0),
        topk=(1, 5),
    ))
```

7.1.4 Add new loss

To add a new loss function, we mainly implement the `forward` function in the loss module. In addition, it is helpful to leverage the decorator `weighted_loss` to weight the loss for each element. Assuming that we want to mimic a probabilistic distribution generated from another classification model, we implement a L1Loss to fulfil the purpose as below.

1. Create a new file in `mmcls/models/losses/l1_loss.py`.

```
import torch
import torch.nn as nn

from ..builder import LOSSES
from .utils import weighted_loss

@weighted_loss
def l1_loss(pred, target):
    assert pred.size() == target.size() and target.numel() > 0
    loss = torch.abs(pred - target)
    return loss

@LOSSES.register_module()
class L1Loss(nn.Module):

    def __init__(self, reduction='mean', loss_weight=1.0):
        super(L1Loss, self).__init__()
        self.reduction = reduction
        self.loss_weight = loss_weight

    def forward(self,
                pred,
                target,
                weight=None,
                avg_factor=None,
                reduction_override=None):
        assert reduction_override in (None, 'none', 'mean', 'sum')
        reduction = (
            reduction_override if reduction_override else self.reduction)
        loss = self.loss_weight * l1_loss(
            pred, target, weight, reduction=reduction, avg_factor=avg_factor)
        return loss
```

2. Import the module in `mmcls/models/losses/__init__.py`.

```
...
from .l1_loss import L1Loss, l1_loss
```

(continues on next page)

(continued from previous page)

```
__all__ = [
    ..., 'L1Loss', 'l1_loss'
]
```

3. Modify loss field in the config.

```
loss=dict(type='L1Loss', loss_weight=1.0))
```

CHAPTER

EIGHT

TUTORIAL 6: CUSTOMIZE SCHEDULE

In this tutorial, we will introduce some methods about how to construct optimizers, customize learning rate and momentum schedules, parameter-wise finely configuration, gradient clipping, gradient accumulation, and customize self-implemented methods for the project.

- Customize optimizer supported by PyTorch
- Customize learning rate schedules
  - Learning rate decay
  - Warmup strategy
- Customize momentum schedules
- Parameter-wise finely configuration
- Gradient clipping and gradient accumulation
  - Gradient clipping
  - Gradient accumulation
- Customize self-implemented methods
  - Customize self-implemented optimizer
  - Customize optimizer constructor

8.1 Customize optimizer supported by PyTorch

We already support to use all the optimizers implemented by PyTorch, and to use and modify them, please change the `optimizer` field of config files.

For example, if you want to use `SGD`, the modification could be as the following.

```
optimizer = dict(type='SGD', lr=0.0003, weight_decay=0.0001)
```

To modify the learning rate of the model, just modify the `lr` in the config of optimizer. You can also directly set other arguments according to the API doc of PyTorch.

For example, if you want to use `Adam` with the setting like `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)` in PyTorch, the config should looks like.

```
optimizer = dict(type='Adam', lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0,␣
↪amsgrad=False)
```

8.2 Customize learning rate schedules

8.2.1 Learning rate decay

Learning rate decay is widely used to improve performance. And to use learning rate decay, please set the `lr_confg` field in config files.

For example, we use step policy as the default learning rate decay policy of ResNet, and the config is:

```
lr_config = dict(policy='step', step=[100, 150])
```

Then during training, the program will call `StepLRHook` periodically to update the learning rate.

We also support many other learning rate schedules here, such as `CosineAnnealing` and `Poly` schedule. Here are some examples

- ConsineAnnealing schedule:

```
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

- Poly schedule:

```
lr_config = dict(policy='poly', power=0.9, min_lr=1e-4, by_epoch=False)
```

8.2.2 Warmup strategy

In the early stage, training is easy to be volatile, and warmup is a technique to reduce volatility. With warmup, the learning rate will increase gradually from a minor value to the expected value.

In MMClassification, we use `lr_config` to configure the warmup strategy, the main parameters are as follows:

- `warmup`: The warmup curve type. Please choose one from 'constant', 'linear', 'exp' and `None`, and `None` means disable warmup.
- `warmup_by_epoch` : if warmup by epoch or not, default to be True, if set to be False, warmup by iter.
- `warmup_iters` : the number of warm-up iterations, when `warmup_by_epoch=True`, the unit is epoch; when `warmup_by_epoch=False`, the unit is the number of iterations (iter).
- `warmup_ratio` : warm-up initial learning rate will calculate as `lr = lr * warmup_ratio`。

Here are some examples

1. linear & warmup by iter

```
lr_config = dict(
    policy='CosineAnnealing',
    by_epoch=False,
    min_lr_ratio=1e-2,
    warmup='linear',
    warmup_ratio=1e-3,
```

(continues on next page)

(continued from previous page)

```
    warmup_iters=20 * 1252,
    warmup_by_epoch=False)
```

2. exp & warmup by epoch

```
lr_config = dict(
    policy='CosineAnnealing',
    min_lr=0,
    warmup='exp',
    warmup_iters=5,
    warmup_ratio=0.1,
    warmup_by_epoch=True)
```

Tip: After completing your configuration file， you could use learning rate visualization tool to draw the corresponding learning rate adjustment curve.

8.3 Customize momentum schedules

We support the momentum scheduler to modify the model's momentum according to learning rate, which could make the model converge in a faster way.

Momentum scheduler is usually used with LR scheduler, for example, the following config is used to accelerate convergence. For more details, please refer to the implementation of CyclicLrUpdater and CyclicMomentumUpdater.

Here is an example

```
lr_config = dict(
    policy='cyclic',
    target_ratio=(10, 1e-4),
    cyclic_times=1,
    step_ratio_up=0.4,
)
momentum_config = dict(
    policy='cyclic',
    target_ratio=(0.85 / 0.95, 1),
    cyclic_times=1,
    step_ratio_up=0.4,
)
```

8.4 Parameter-wise finely configuration

Some models may have some parameter-specific settings for optimization, for example, no weight decay to the BatchNorm layer or using different learning rates for different network layers. To finely configuration them, we can use the `paramwise_cfg` option in `optimizer`.

We provide some examples here and more usages refer to DefaultOptimizerConstructor.

- Using specified options

The `DefaultOptimizerConstructor` provides options including `bias_lr_mult`, `bias_decay_mult`, `norm_decay_mult`, `dwconv_decay_mult`, `dcn_offset_lr_mult` and `bypass_duplicate` to configure special optimizer behaviors of bias, normalization, depth-wise convolution, deformable convolution and duplicated parameter. E.g:

1. No weight decay to the BatchNorm layer

```
optimizer = dict(
    type='SGD',
    lr=0.8,
    weight_decay=1e-4,
    paramwise_cfg=dict(norm_decay_mult=0.))
```

- Using `custom_keys` dict

  MMClassification can use `custom_keys` to specify different parameters to use different learning rates or weight decays, for example:

  1. No weight decay for specific parameters

```
paramwise_cfg = dict(
    custom_keys={
        'backbone.cls_token': dict(decay_mult=0.0),
        'backbone.pos_embed': dict(decay_mult=0.0)
    })

optimizer = dict(
    type='SGD',
    lr=0.8,
    weight_decay=1e-4,
    paramwise_cfg=paramwise_cfg)
```

  2. Using a smaller learning rate and a weight decay for the backbone layers

```
optimizer = dict(
    type='SGD',
    lr=0.8,
    weight_decay=1e-4,
    # 'lr' for backbone and 'weight_decay' are 0.1 * lr and 0.9 * weight_decay
    paramwise_cfg=dict(
        custom_keys={'backbone': dict(lr_mult=0.1, decay_mult=0.9)}))
```

8.5 Gradient clipping and gradient accumulation

Besides the basic function of PyTorch optimizers, we also provide some enhancement functions, such as gradient clipping, gradient accumulation, etc., refer to MMCV.

8.5.1 Gradient clipping

During the training process, the loss function may get close to a cliffy region and cause gradient explosion. And gradient clipping is helpful to stabilize the training process. More introduction can be found in this page.

Currently we support `grad_clip` option in `optimizer_config`, and the arguments refer to PyTorch Documentation.

Here is an example:

```
optimizer_config = dict(grad_clip=dict(max_norm=35, norm_type=2))
# norm_type: type of the used p-norm, here norm_type is 2.
```

When inheriting from base and modifying configs, if `grad_clip=None` in base, `_delete_=True` is needed. For more details about `_delete_` you can refer to TUTORIAL 1: LEARN ABOUT CONFIGS. For example,

```
_base_ = [./_base_/schedules/imagenet_bs256_coslr.py]

optimizer_config = dict(grad_clip=dict(max_norm=35, norm_type=2), _delete_=True, type=
↪'OptimizerHook')
# you can ignore type if type is 'OptimizerHook', otherwise you must add "type=
↪'xxxxxOptimizerHook'" here
```

8.5.2 Gradient accumulation

When computing resources are lacking, the batch size can only be set to a small value, which may affect the performance of models. Gradient accumulation can be used to solve this problem.

Here is an example:

```
data = dict(samples_per_gpu=64)
optimizer_config = dict(type="GradientCumulativeOptimizerHook", cumulative_iters=4)
```

Indicates that during training, back-propagation is performed every 4 iters. And the above is equivalent to:

```
data = dict(samples_per_gpu=256)
optimizer_config = dict(type="OptimizerHook")
```

Note: When the optimizer hook type is not specified in `optimizer_config`, `OptimizerHook` is used by default.

8.6 Customize self-implemented methods

In academic research and industrial practice, it may be necessary to use optimization methods not implemented by MMClassification, and you can add them through the following methods.

Note: This part will modify the MMClassification source code or add code to the MMClassification framework, beginners can skip it.

8.6.1 Customize self-implemented optimizer

1. Define a new optimizer

A customized optimizer could be defined as below.

Assume you want to add an optimizer named `MyOptimizer`, which has arguments `a`, `b`, and `c`. You need to create a new directory named `mmcls/core/optimizer`. And then implement the new optimizer in a file, e.g., in `mmcls/core/optimizer/my_optimizer.py`:

```
from mmcv.runner import OPTIMIZERS
from torch.optim import Optimizer


@OPTIMIZERS.register_module()
class MyOptimizer(Optimizer):

    def __init__(self, a, b, c):
```

2. Add the optimizer to registry

To find the above module defined above, this module should be imported into the main namespace at first. There are two ways to achieve it.

- Modify `mmcls/core/optimizer/__init__.py` to import it into `optimizer` package, and then modify `mmcls/core/__init__.py` to import the new `optimizer` package.

  Create the `mmcls/core/optimizer` folder and the `mmcls/core/optimizer/__init__.py` file if they don't exist. The newly defined module should be imported in `mmcls/core/optimizer/__init__.py` and `mmcls/core/__init__.py` so that the registry will find the new module and add it:

```
# In mmcls/core/optimizer/__init__.py
from .my_optimizer import MyOptimizer # MyOptimizer maybe other class name

__all__ = ['MyOptimizer']
```

```
# In mmcls/core/__init__.py
...
from .optimizer import *  # noqa: F401, F403
```

- Use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmcls.core.optimizer.my_optimizer'], allow_failed_
↪imports=False)
```

The module `mmcls.core.optimizer.my_optimizer` will be imported at the beginning of the program and the class `MyOptimizer` is then automatically registered. Note that only the package containing the class `MyOptimizer` should be imported. `mmcls.core.optimizer.my_optimizer.MyOptimizer` **cannot** be imported directly.

3. Specify the optimizer in the config file

Then you can use `MyOptimizer` in `optimizer` field of config files. In the configs, the optimizers are defined by the field `optimizer` like the following:

```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
```

To use your own optimizer, the field can be changed to

```
optimizer = dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value)
```

8.6.2 Customize optimizer constructor

Some models may have some parameter-specific settings for optimization, e.g. weight decay for BatchNorm layers.

Although our `DefaultOptimizerConstructor` is powerful, it may still not cover your need. If that, you can do those fine-grained parameter tuning through customizing optimizer constructor.

```
from mmcv.runner.optimizer import OPTIMIZER_BUILDERS


@OPTIMIZER_BUILDERS.register_module()
class MyOptimizerConstructor:

    def __init__(self, optimizer_cfg, paramwise_cfg=None):
        pass

    def __call__(self, model):
        ...      # Construct your optimzier here.
        return my_optimizer
```

The default optimizer constructor is implemented here, which could also serve as a template for new optimizer constructor.

CHAPTER

NINE

TUTORIAL 7: CUSTOMIZE RUNTIME SETTINGS

In this tutorial, we will introduce some methods about how to customize workflow and hooks when running your own settings for the project.

- Customize Workflow
- Hooks
  - Default training hooks
    - CheckpointHook
    - LoggerHooks
    - EvalHook
  - Use other implemented hooks
  - Customize self-implemented hooks
    - 1. Implement a new hook
    - 2. Register the new hook
    - 3. Modify the config
- FAQ

9.1 Customize Workflow

Workflow is a list of (phase, duration) to specify the running order and duration. The meaning of "duration" depends on the runner's type.

For example, we use epoch-based runner by default, and the "duration" means how many epochs the phase to be executed in a cycle. Usually, we only want to execute training phase, just use the following config.

```
workflow = [('train', 1)]
```

Sometimes we may want to check some metrics (e.g. loss, accuracy) about the model on the validate set. In such case, we can set the workflow as

```
[('train', 1), ('val', 1)]
```

so that 1 epoch for training and 1 epoch for validation will be run iteratively.

By default, we recommend using **`EvalHook`** to do evaluation after the training epoch, but you can still use `val` workflow as an alternative.

Note:

1. The parameters of model will not be updated during the val epoch.
2. Keyword `max_epochs` in the config only controls the number of training epochs and will not affect the validation workflow.
3. Workflows `[('train', 1), ('val', 1)]` and `[('train', 1)]` will not change the behavior of `EvalHook` because `EvalHook` is called by `after_train_epoch` and validation workflow only affect hooks that are called through `after_val_epoch`. Therefore, the only difference between `[('train', 1), ('val', 1)]` and `[('train', 1)]` is that the runner will calculate losses on the validation set after each training epoch.

9.2 Hooks

The hook mechanism is widely used in the OpenMMLab open-source algorithm library. Combined with the `Runner`, the entire life cycle of the training process can be managed easily. You can learn more about the hook through related article.

Hooks only work after being registered into the runner. At present, hooks are mainly divided into two categories:

- default training hooks

The default training hooks are registered by the runner by default. Generally, they are hooks for some basic functions, and have a certain priority, you don't need to modify the priority.

- custom hooks

The custom hooks are registered through `custom_hooks`. Generally, they are hooks with enhanced functions. The priority needs to be specified in the configuration file. If you do not specify the priority of the hook, it will be set to 'NORMAL' by default.

Priority list

The priority determines the execution order of the hooks. Before training, the log will print out the execution order of the hooks at each stage to facilitate debugging.

9.2.1 default training hooks

Some common hooks are not registered through `custom_hooks`, they are

`OptimizerHook`, `MomentumUpdaterHook` and `LrUpdaterHook` have been introduced in *sehedule strategy*. `IterTimerHook` is used to record elapsed time and does not support modification.

Here we reveal how to customize `CheckpointHook`, `LoggerHooks`, and `EvalHook`.

CheckpointHook

The MMCV runner will use `checkpoint_config` to initialize `CheckpointHook`.

```
checkpoint_config = dict(interval=1)
```

We could set `max_keep_ckpts` to save only a small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are here

LoggerHooks

The `log_config` wraps multiple logger hooks and enables to set intervals. Now MMCV supports `TextLoggerHook`, `WandbLoggerHook`, `MlflowLoggerHook`, `NeptuneLoggerHook`, `DvcliveLoggerHook` and `TensorboardLoggerHook`. The detailed usages can be found in the doc.

```
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        dict(type='TensorboardLoggerHook')
    ])
```

EvalHook

The config of `evaluation` will be used to initialize the `EvalHook`.

The `EvalHook` has some reserved keys, such as `interval`, `save_best` and `start`, and the other arguments such as `metrics` will be passed to the `dataset.evaluate()`

```
evaluation = dict(interval=1, metric='accuracy', metric_options={'topk': (1, )})
```

You can save the model weight when the best verification result is obtained by modifying the parameter `save_best`:

```
# "auto" means automatically select the metrics to compare.
# You can also use a specific key like "accuracy_top-1".
evaluation = dict(interval=1, save_best="auto", metric='accuracy', metric_options={'topk
↪': (1, )})
```

When running some large experiments, you can skip the validation step at the beginning of training by modifying the parameter `start` as below:

```
evaluation = dict(interval=1, start=200, metric='accuracy', metric_options={'topk': (1,␣
↪)})
```

This indicates that, before the 200th epoch, evaluations would not be executed. Since the 200th epoch, evaluations would be executed after the training process.

Note: In the default configuration files of MMClassification, the evaluation field is generally placed in the datasets configs.

9.2.2 Use other implemented hooks

Some hooks have been already implemented in MMCV and MMClassification, they are:

- EMAHook
- SyncBuffersHook
- EmptyCacheHook
- ProfilerHook
- .....

If the hook is already implemented in MMCV, you can directly modify the config to use the hook as below

```
mmcv_hooks = [
    dict(type='MMCVHook', a=a_value, b=b_value, priority='NORMAL')
]
```

such as using `EMAHook`, interval is 100 iters:

```
custom_hooks = [
    dict(type='EMAHook', interval=100, priority='HIGH')
]
```

9.3 Customize self-implemented hooks

9.3.1 1. Implement a new hook

Here we give an example of creating a new hook in MMClassification and using it in training.

```
from mmcv.runner import HOOKS, Hook


@HOOKS.register_module()
class MyHook(Hook):

    def __init__(self, a, b):
        pass

    def before_run(self, runner):
        pass

    def after_run(self, runner):
        pass

    def before_epoch(self, runner):
        pass

    def after_epoch(self, runner):
        pass

    def before_iter(self, runner):
```

(continues on next page)

(continued from previous page)

```
        pass

    def after_iter(self, runner):
        pass
```

Depending on the functionality of the hook, the users need to specify what the hook will do at each stage of the training in `before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`.

9.3.2 2. Register the new hook

Then we need to make `MyHook` imported. Assuming the file is in `mmcls/core/utils/my_hook.py` there are two ways to do that:

- Modify `mmcls/core/utils/__init__.py` to import it.

  The newly defined module should be imported in `mmcls/core/utils/__init__.py` so that the registry will find the new module and add it:

```
from .my_hook import MyHook
```

- Use `custom_imports` in the config to manually import it

```
custom_imports = dict(imports=['mmcls.core.utils.my_hook'], allow_failed_imports=False)
```

9.3.3 3. Modify the config

```
custom_hooks = [
    dict(type='MyHook', a=a_value, b=b_value)
]
```

You can also set the priority of the hook as below:

```
custom_hooks = [
    dict(type='MyHook', a=a_value, b=b_value, priority='ABOVE_NORMAL')
]
```

By default, the hook's priority is set as `NORMAL` during registration.

9.4 FAQ

9.4.1 1. `resume_from` and `load_from` and `init_cfg.Pretrained`

- `load_from` : only imports model weights, which is mainly used to load pre-trained or trained models;
- `resume_from` : not only import model weights, but also optimizer information, current epoch information, mainly used to continue training from the checkpoint.
- `init_cfg.Pretrained` : Load weights during weight initialization, and you can specify which module to load. This is usually used when fine-tuning a model, refer to *Tutorial 2: Fine-tune Models*.

CHAPTER

TEN

MODEL ZOO SUMMARY

- Number of papers: 16
  - ALGORITHM: 16
- Number of checkpoints: 80
  - [ALGORITHM] *MLP-Mixer: An all-MLP Architecture for Vision* (2 ckpts)
  - [ALGORITHM] *MobileNetV2: Inverted Residuals and Linear Bottlenecks* (1 ckpts)
  - [ALGORITHM] *Searching for MobileNetV3* (2 ckpts)
  - [ALGORITHM] *Designing Network Design Spaces* (8 ckpts)
  - [ALGORITHM] *Repvgg: Making vgg-style convnets great again* (12 ckpts)
  - [ALGORITHM] *Res2Net: A New Multi-scale Backbone Architecture* (3 ckpts)
  - [ALGORITHM] *Deep Residual Learning for Image Recognition* (15 ckpts)
  - [ALGORITHM] *Aggregated Residual Transformations for Deep Neural Networks* (4 ckpts)
  - [ALGORITHM] *Squeeze-and-Excitation Networks* (2 ckpts)
  - [ALGORITHM] *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices* (1 ckpts)
  - [ALGORITHM] *Shufflenet v2: Practical guidelines for efficient cnn architecture design* (1 ckpts)
  - [ALGORITHM] *Swin Transformer: Hierarchical Vision Transformer using Shifted Windows* (11 ckpts)
  - [ALGORITHM] *Tokens-to-Token ViT: Training Vision Transformers from Scratch on ImageNet* (3 ckpts)
  - [ALGORITHM] *Transformer in Transformer* (1 ckpts)
  - [ALGORITHM] *Very Deep Convolutional Networks for Large-Scale Image Recognition* (8 ckpts)
  - [ALGORITHM] *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* (6 ckpts)

CHAPTER

ELEVEN

MODEL ZOO

11.1 ImageNet

ImageNet has multiple versions, but the most commonly used one is ILSVRC 2012. The ResNet family models below are trained by standard data augmentations, i.e., RandomResizedCrop, RandomHorizontalFlip and Normalize.

Models with * are converted from other repos, others are trained by ourselves.

11.2 CIFAR10

CHAPTER

TWELVE

MLP-MIXER: AN ALL-MLP ARCHITECTURE FOR VISION

12.1 Abstract

Convolutional Neural Networks (CNNs) are the go-to model for computer vision. Recently, attention-based networks, such as the Vision Transformer, have also become popular. In this paper we show that while convolutions and attention are both sufficient for good performance, neither of them are necessary. We present MLP-Mixer, an architecture based exclusively on multi-layer perceptrons (MLPs). MLP-Mixer contains two types of layers: one with MLPs applied independently to image patches (i.e. "mixing" the per-location features), and one with MLPs applied across patches (i.e. "mixing" spatial information). When trained on large datasets, or with modern regularization schemes, MLP-Mixer attains competitive scores on image classification benchmarks, with pre-training and inference cost comparable to state-of-the-art models. We hope that these results spark further research beyond the realms of well established CNNs and Transformers.

12.2 Citation

```
@misc{tolstikhin2021mlpmixer,
      title={MLP-Mixer: An all-MLP Architecture for Vision},
      author={Ilya Tolstikhin and Neil Houlsby and Alexander Kolesnikov and Lucas Beyer␣
↪and Xiaohua Zhai and Thomas Unterthiner and Jessica Yung and Andreas Steiner and␣
↪Daniel Keysers and Jakob Uszkoreit and Mario Lucic and Alexey Dosovitskiy},
      year={2021},
      eprint={2105.01601},
      archivePrefix={arXiv},
      primaryClass={cs.CV}
}
```

12.3 Pretrain model

The pre-trained modles are converted from timm.

12.3.1 ImageNet-1k

*Models with * are converted from other repos.*

CHAPTER

THIRTEEN

MOBILENETV2: INVERTED RESIDUALS AND LINEAR BOTTLENECKS

13.1 Abstract

In this paper we describe a new mobile architecture, MobileNetV2, that improves the state of the art performance of mobile models on multiple tasks and benchmarks as well as across a spectrum of different model sizes. We also describe efficient ways of applying these mobile models to object detection in a novel framework we call SSDLite. Additionally, we demonstrate how to build mobile semantic segmentation models through a reduced form of DeepLabv3 which we call Mobile DeepLabv3.

The MobileNetV2 architecture is based on an inverted residual structure where the input and output of the residual block are thin bottleneck layers opposite to traditional residual models which use expanded representations in the input an MobileNetV2 uses lightweight depthwise convolutions to filter features in the intermediate expansion layer. Additionally, we find that it is important to remove non-linearities in the narrow layers in order to maintain representational power. We demonstrate that this improves performance and provide an intuition that led to this design. Finally, our approach allows decoupling of the input/output domains from the expressiveness of the transformation, which provides a convenient framework for further analysis. We measure our performance on Imagenet classification, COCO object detection, VOC image segmentation. We evaluate the trade-offs between accuracy, and number of operations measured by multiply-adds (MAdd), as well as the number of parameters

13.2 Citation

```
@INPROCEEDINGS{8578572,
  author={M. {Sandler} and A. {Howard} and M. {Zhu} and A. {Zhmoginov} and L. {Chen}},
  booktitle={2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition},
  title={MobileNetV2: Inverted Residuals and Linear Bottlenecks},
  year={2018},
  volume={},
  number={},
  pages={4510-4520},
  doi={10.1109/CVPR.2018.00474}}
}
```

13.3 Results and models

13.3.1 ImageNet

CHAPTER

FOURTEEN

SEARCHING FOR MOBILENETV3

14.1 Abstract

We present the next generation of MobileNets based on a combination of complementary search techniques as well as a novel architecture design. MobileNetV3 is tuned to mobile phone CPUs through a combination of hardware-aware network architecture search (NAS) complemented by the NetAdapt algorithm and then subsequently improved through novel architecture advances. This paper starts the exploration of how automated search algorithms and network design can work together to harness complementary approaches improving the overall state of the art. Through this process we create two new MobileNet models for release: MobileNetV3-Large and MobileNetV3-Small which are targeted for high and low resource use cases. These models are then adapted and applied to the tasks of object detection and semantic segmentation. For the task of semantic segmentation (or any dense pixel prediction), we propose a new efficient segmentation decoder Lite Reduced Atrous Spatial Pyramid Pooling (LR-ASPP). We achieve new state of the art results for mobile classification, detection and segmentation. MobileNetV3-Large is 3.2% more accurate on ImageNet classification while reducing latency by 15% compared to MobileNetV2. MobileNetV3-Small is 4.6% more accurate while reducing latency by 5% compared to MobileNetV2. MobileNetV3-Large detection is 25% faster at roughly the same accuracy as MobileNetV2 on COCO detection. MobileNetV3-Large LR-ASPP is 30% faster than MobileNetV2 R-ASPP at similar accuracy for Cityscapes segmentation.

14.2 Citation

```
@inproceedings{Howard_2019_ICCV,
    author = {Howard, Andrew and Sandler, Mark and Chu, Grace and Chen, Liang-Chieh and
↪Chen, Bo and Tan, Mingxing and Wang, Weijun and Zhu, Yukun and Pang, Ruoming and
↪Vasudevan, Vijay and Le, Quoc V. and Adam, Hartwig},
    title = {Searching for MobileNetV3},
    booktitle = {Proceedings of the IEEE/CVF International Conference on Computer Vision
↪(ICCV)},
    month = {October},
    year = {2019}
}
```

14.3 Pretrain model

The pre-trained modles are converted from torchvision.

14.3.1 ImageNet

14.4 Results and models

Waiting for adding.

CHAPTER

FIFTEEN

DESIGNING NETWORK DESIGN SPACES

15.1 Abstract

In this work, we present a new network design paradigm. Our goal is to help advance the understanding of network design and discover design principles that generalize across settings. Instead of focusing on designing individual network instances, we design network design spaces that parametrize populations of networks. The overall process is analogous to classic manual design of networks, but elevated to the design space level. Using our methodology we explore the structure aspect of network design and arrive at a low-dimensional design space consisting of simple, regular networks that we call RegNet. The core insight of the RegNet parametrization is surprisingly simple: widths and depths of good networks can be explained by a quantized linear function. We analyze the RegNet design space and arrive at interesting findings that do not match the current practice of network design. The RegNet design space provides simple and fast networks that work well across a wide range of flop regimes. Under comparable training settings and flops, the RegNet models outperform the popular EfficientNet models while being up to 5x faster on GPUs.

15.2 Citation

```
@article{radosavovic2020designing,
    title={Designing Network Design Spaces},
    author={Ilija Radosavovic and Raj Prateek Kosaraju and Ross Girshick and Kaiming He␣
↪and Piotr Dollár},
    year={2020},
    eprint={2003.13678},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

15.3 Pretrain model

The pre-trained modles are converted from model zoo of pycls.

15.3.1 ImageNet

15.4 Results and models

Waiting for adding.

CHAPTER

SIXTEEN

REPVGG: MAKING VGG-STYLE CONVNETS GREAT AGAIN

16.1 Abstract

We present a simple but powerful architecture of convolutional neural network, which has a VGG-like inference-time body composed of nothing but a stack of 3x3 convolution and ReLU, while the training-time model has a multi-branch topology. Such decoupling of the training-time and inference-time architecture is realized by a structural re-parameterization technique so that the model is named RepVGG. On ImageNet, RepVGG reaches over 80% top-1 accuracy, which is the first time for a plain model, to the best of our knowledge. On NVIDIA 1080Ti GPU, RepVGG models run 83% faster than ResNet-50 or 101% faster than ResNet-101 with higher accuracy and show favorable accuracy-speed trade-off compared to the state-of-the-art models like EfficientNet and RegNet.

16.2 Citation

```
@inproceedings{ding2021repvgg,
  title={Repvgg: Making vgg-style convnets great again},
  author={Ding, Xiaohan and Zhang, Xiangyu and Ma, Ningning and Han, Jungong and Ding,
↪Guiguang and Sun, Jian},
  booktitle={Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern
↪Recognition},
  pages={13733--13742},
  year={2021}
}
```

16.3 Pretrain model

*Models with * are converted from other repos.*

16.4 Reparameterize RepVGG

The checkpoints provided are all in `train` form. Use the reparameterize tool to switch them to more efficient `deploy` form, which not only has fewer parameters but also less calculations.

```
python ./tools/convert_models/reparameterize_repvgg.py ${CFG_PATH} ${SRC_CKPT_PATH} $
↪{TARGET_CKPT_PATH}
```

`${CFG_PATH}` is the config file, `${SRC_CKPT_PATH}` is the source chenpoint file, `${TARGET_CKPT_PATH}` is the target deploy weight file path.

To use reparameterized repvgg weight, the config file must switch to the deploy config files as below:

```
python ./tools/test.py ${RapVGG_Deploy_CFG} ${CHECK_POINT}
```

CHAPTER

SEVENTEEN

RES2NET: A NEW MULTI-SCALE BACKBONE ARCHITECTURE

17.1 Abstract

Representing features at multiple scales is of great importance for numerous vision tasks. Recent advances in backbone convolutional neural networks (CNNs) continually demonstrate stronger multi-scale representation ability, leading to consistent performance gains on a wide range of applications. However, most existing methods represent the multi-scale features in a layer-wise manner. In this paper, we propose a novel building block for CNNs, namely Res2Net, by constructing hierarchical residual-like connections within one single residual block. The Res2Net represents multi-scale features at a granular level and increases the range of receptive fields for each network layer. The proposed Res2Net block can be plugged into the state-of-the-art backbone CNN models, e.g., ResNet, ResNeXt, and DLA. We evaluate the Res2Net block on all these models and demonstrate consistent performance gains over baseline models on widely-used datasets, e.g., CIFAR-100 and ImageNet. Further ablation studies and experimental results on representative computer vision tasks, i.e., object detection, class activation mapping, and salient object detection, further verify the superiority of the Res2Net over the state-of-the-art baseline methods.

17.2 Citation

```
@article{gao2019res2net,
  title={Res2Net: A New Multi-scale Backbone Architecture},
  author={Gao, Shang-Hua and Cheng, Ming-Ming and Zhao, Kai and Zhang, Xin-Yu and Yang,␣
↪Ming-Hsuan and Torr, Philip},
  journal={IEEE TPAMI},
  year={2021},
  doi={10.1109/TPAMI.2019.2938758},
}
```

17.3 Pretrain model

The pre-trained models are converted from official repo.

17.3.1 ImageNet 1k

*Models with * are converted from other repos.*

CHAPTER

EIGHTEEN

DEEP RESIDUAL LEARNING FOR IMAGE RECOGNITION

18.1 Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.

The depth of representations is of central importance for many visual recognition tasks. Solely due to our extremely deep representations, we obtain a 28% relative improvement on the COCO object detection dataset. Deep residual nets are foundations of our submissions to ILSVRC & COCO 2015 competitions, where we also won the 1st places on the tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.

18.2 Citation

```
@inproceedings{he2016deep,
  title={Deep residual learning for image recognition},
  author={He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={770--778},
  year={2016}
}
```

18.3 Results and models

18.4 Cifar10

18.5 Cifar100

18.5.1 ImageNet

CHAPTER

NINETEEN

AGGREGATED RESIDUAL TRANSFORMATIONS FOR DEEP NEURAL NETWORKS

19.1 Abstract

We present a simple, highly modularized network architecture for image classification. Our network is constructed by repeating a building block that aggregates a set of transformations with the same topology. Our simple design results in a homogeneous, multi-branch architecture that has only a few hyper-parameters to set. This strategy exposes a new dimension, which we call "cardinality" (the size of the set of transformations), as an essential factor in addition to the dimensions of depth and width. On the ImageNet-1K dataset, we empirically show that even under the restricted condition of maintaining complexity, increasing cardinality is able to improve classification accuracy. Moreover, increasing cardinality is more effective than going deeper or wider when we increase the capacity. Our models, named ResNeXt, are the foundations of our entry to the ILSVRC 2016 classification task in which we secured 2nd place. We further investigate ResNeXt on an ImageNet-5K set and the COCO detection set, also showing better results than its ResNet counterpart. The code and models are publicly available online.

19.2 Citation

```
@inproceedings{xie2017aggregated,
  title={Aggregated residual transformations for deep neural networks},
  author={Xie, Saining and Girshick, Ross and Doll{\'a}r, Piotr and Tu, Zhuowen and He,␣
↪Kaiming},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={1492--1500},
  year={2017}
}
```

19.3 Results and models

19.3.1 ImageNet

CHAPTER

TWENTY

SQUEEZE-AND-EXCITATION NETWORKS

20.1 Abstract

The central building block of convolutional neural networks (CNNs) is the convolution operator, which enables networks to construct informative features by fusing both spatial and channel-wise information within local receptive fields at each layer. A broad range of prior research has investigated the spatial component of this relationship, seeking to strengthen the representational power of a CNN by enhancing the quality of spatial encodings throughout its feature hierarchy. In this work, we focus instead on the channel relationship and propose a novel architectural unit, which we term the "Squeeze-and-Excitation" (SE) block, that adaptively recalibrates channel-wise feature responses by explicitly modelling interdependencies between channels. We show that these blocks can be stacked together to form SENet architectures that generalise extremely effectively across different datasets. We further demonstrate that SE blocks bring significant improvements in performance for existing state-of-the-art CNNs at slight additional computational cost. Squeeze-and-Excitation Networks formed the foundation of our ILSVRC 2017 classification submission which won first place and reduced the top-5 error to 2.251%, surpassing the winning entry of 2016 by a relative improvement of ~25%.

20.2 Citation

```
@inproceedings{hu2018squeeze,
  title={Squeeze-and-excitation networks},
  author={Hu, Jie and Shen, Li and Sun, Gang},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern␣
↪recognition},
  pages={7132--7141},
  year={2018}
}
```

20.3 Results and models

20.3.1 ImageNet

CHAPTER

TWENTYONE

SHUFFLENET: AN EXTREMELY EFFICIENT CONVOLUTIONAL NEURAL NETWORK FOR MOBILE DEVICES

21.1 Abstract

We introduce an extremely computation-efficient CNN architecture named ShuffleNet, which is designed specially for mobile devices with very limited computing power (e.g., 10-150 MFLOPs). The new architecture utilizes two new operations, pointwise group convolution and channel shuffle, to greatly reduce computation cost while maintaining accuracy. Experiments on ImageNet classification and MS COCO object detection demonstrate the superior performance of ShuffleNet over other structures, e.g. lower top-1 error (absolute 7.8%) than recent MobileNet on ImageNet classification task, under the computation budget of 40 MFLOPs. On an ARM-based mobile device, ShuffleNet achieves ~13x actual speedup over AlexNet while maintaining comparable accuracy.

21.2 Citation

```
@inproceedings{zhang2018shufflenet,
  title={Shufflenet: An extremely efficient convolutional neural network for mobile devices},
  author={Zhang, Xiangyu and Zhou, Xinyu and Lin, Mengxiao and Sun, Jian},
  booktitle={Proceedings of the IEEE conference on computer vision and pattern recognition},
  pages={6848--6856},
  year={2018}
}
```

21.3 Results and models

21.3.1 ImageNet

CHAPTER

TWENTYTWO

SHUFFLENET V2: PRACTICAL GUIDELINES FOR EFFICIENT CNN ARCHITECTURE DESIGN

22.1 Abstract

Currently, the neural network architecture design is mostly guided by the *indirect* metric of computation complexity, i.e., FLOPs. However, the *direct* metric, e.g., speed, also depends on the other factors such as memory access cost and platform characterics. Thus, this work proposes to evaluate the direct metric on the target platform, beyond only considering FLOPs. Based on a series of controlled experiments, this work derives several practical *guidelines* for efficient network design. Accordingly, a new architecture is presented, called *ShuffleNet V2*. Comprehensive ablation experiments verify that our model is the state-of-the-art in terms of speed and accuracy tradeoff.

22.2 Citation

```
@inproceedings{ma2018shufflenet,
  title={Shufflenet v2: Practical guidelines for efficient cnn architecture design},
  author={Ma, Ningning and Zhang, Xiangyu and Zheng, Hai-Tao and Sun, Jian},
  booktitle={Proceedings of the European conference on computer vision (ECCV)},
  pages={116--131},
  year={2018}
}
```

22.3 Results and models

22.3.1 ImageNet

CHAPTER

TWENTYTHREE

SWIN TRANSFORMER: HIERARCHICAL VISION TRANSFORMER USING SHIFTED WINDOWS

23.1 Abstract

This paper presents a new vision Transformer, called Swin Transformer, that capably serves as a general-purpose backbone for computer vision. Challenges in adapting Transformer from language to vision arise from differences between the two domains, such as large variations in the scale of visual entities and the high resolution of pixels in images compared to words in text. To address these differences, we propose a hierarchical Transformer whose representation is computed with **S**hifted **win**dows. The shifted windowing scheme brings greater efficiency by limiting self-attention computation to non-overlapping local windows while also allowing for cross-window connection. This hierarchical architecture has the flexibility to model at various scales and has linear computational complexity with respect to image size. These qualities of Swin Transformer make it compatible with a broad range of vision tasks, including image classification (87.3 top-1 accuracy on ImageNet-1K) and dense prediction tasks such as object detection (58.7 box AP and 51.1 mask AP on COCO test-dev) and semantic segmentation (53.5 mIoU on ADE20K val). Its performance surpasses the previous state-of-the-art by a large margin of +2.7 box AP and +2.6 mask AP on COCO, and +3.2 mIoU on ADE20K, demonstrating the potential of Transformer-based models as vision backbones. The hierarchical design and the shifted window approach also prove beneficial for all-MLP architectures.

23.2 Citation

```
@article{liu2021Swin,
  title={Swin Transformer: Hierarchical Vision Transformer using Shifted Windows},
  author={Liu, Ze and Lin, Yutong and Cao, Yue and Hu, Han and Wei, Yixuan and Zhang,␣
↪Zheng and Lin, Stephen and Guo, Baining},
  journal={arXiv preprint arXiv:2103.14030},
  year={2021}
}
```

23.3 Pretrain model

The pre-trained modles are converted from model zoo of Swin Transformer.

23.3.1 ImageNet 1k

23.4 Results and models

23.4.1 ImageNet 1k

Model	Pretrain	resolution	Params(M)	Flops(G)	Top-1 (%)	Top-5 (%)	Config	Download
Swin-T	ImageNet-1k	224x224	28.29	4.36	81.18	95.61	config	model | log
Swin-S	ImageNet-1k	224x224	49.61	8.52	83.02	96.29	config	model | log
Swin-B	ImageNet-1k	224x224	87.77	15.14	83.36	96.44	config	model | log

CHAPTER

TWENTYFOUR

TOKENS-TO-TOKEN VIT: TRAINING VISION TRANSFORMERS FROM SCRATCH ON IMAGENET

24.1 Abstract

Transformers, which are popular for language modeling, have been explored for solving vision tasks recently, \eg, the Vision Transformer (ViT) for image classification. The ViT model splits each image into a sequence of tokens with fixed length and then applies multiple Transformer layers to model their global relation for classification. However, ViT achieves inferior performance to CNNs when trained from scratch on a midsize dataset like ImageNet. We find it is because: 1) the simple tokenization of input images fails to model the important local structure such as edges and lines among neighboring pixels, leading to low training sample efficiency; 2) the redundant attention backbone design of ViT leads to limited feature richness for fixed computation budgets and limited training samples. To overcome such limitations, we propose a new Tokens-To-Token Vision Transformer (T2T-ViT), which incorporates 1) a layer-wise Tokens-to-Token (T2T) transformation to progressively structurize the image to tokens by recursively aggregating neighboring Tokens into one Token (Tokens-to-Token), such that local structure represented by surrounding tokens can be modeled and tokens length can be reduced; 2) an efficient backbone with a deep-narrow structure for vision transformer motivated by CNN architecture design after empirical study. Notably, T2T-ViT reduces the parameter count and MACs of vanilla ViT by half, while achieving more than 3.0% improvement when trained from scratch on ImageNet. It also outperforms ResNets and achieves comparable performance with MobileNets by directly training on ImageNet. For example, T2T-ViT with comparable size to ResNet50 (21.5M parameters) can achieve 83.3% top1 accuracy in image resolution 384×384 on ImageNet.

24.2 Citation

```
@article{yuan2021tokens,
  title={Tokens-to-token vit: Training vision transformers from scratch on imagenet},
  author={Yuan, Li and Chen, Yunpeng and Wang, Tao and Yu, Weihao and Shi, Yujun and Tay,
↪ Francis EH and Feng, Jiashi and Yan, Shuicheng},
  journal={arXiv preprint arXiv:2101.11986},
  year={2021}
}
```

24.3 Pretrain model

The pre-trained modles are converted from official repo.

24.3.1 ImageNet-1k

*Models with * are converted from other repos.*

24.4 Results and models

Waiting for adding.

CHAPTER

TWENTYFIVE

TRANSFORMER IN TRANSFORMER

25.1 Abstract

Transformer is a new kind of neural architecture which encodes the input data as powerful features via the attention mechanism. Basically, the visual transformers first divide the input images into several local patches and then calculate both representations and their relationship. Since natural images are of high complexity with abundant detail and color information, the granularity of the patch dividing is not fine enough for excavating features of objects in different scales and locations. In this paper, we point out that the attention inside these local patches are also essential for building visual transformers with high performance and we explore a new architecture, namely, Transformer iN Transformer (TNT). Specifically, we regard the local patches (e.g., 16×16) as "visual sentences" and present to further divide them into smaller patches (e.g., 4×4) as "visual words". The attention of each word will be calculated with other words in the given visual sentence with negligible computational costs. Features of both words and sentences will be aggregated to enhance the representation ability. Experiments on several benchmarks demonstrate the effectiveness of the proposed TNT architecture, e.g., we achieve an 81.5% top-1 accuracy on the ImageNet, which is about 1.7% higher than that of the state-of-the-art visual transformer with similar computational cost.

25.2 Citation

```
@misc{han2021transformer,
      title={Transformer in Transformer},
      author={Kai Han and An Xiao and Enhua Wu and Jianyuan Guo and Chunjing Xu and
↪Yunhe Wang},
      year={2021},
      eprint={2103.00112},
      archivePrefix={arXiv},
      primaryClass={cs.CV}
}
```

25.3 Pretrain model

The pre-trained modles are converted from timm.

25.3.1 ImageNet

*Models with * are converted from other repos.*

25.4 Results and models

Waiting for adding.

CHAPTER

TWENTYSIX

VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION

26.1 Abstract

In this work we investigate the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. Our main contribution is a thorough evaluation of networks of increasing depth using an architecture with very small (3x3) convolution filters, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16-19 weight layers. These findings were the basis of our ImageNet Challenge 2014 submission, where our team secured the first and the second places in the localisation and classification tracks respectively. We also show that our representations generalise well to other datasets, where they achieve state-of-the-art results. We have made our two best-performing ConvNet models publicly available to facilitate further research on the use of deep visual representations in computer vision.

26.2 Citation

```
@article{simonyan2014very,
  title={Very deep convolutional networks for large-scale image recognition},
  author={Simonyan, Karen and Zisserman, Andrew},
  journal={arXiv preprint arXiv:1409.1556},
  year={2014}
}
```

26.3 Results and models

26.3.1 ImageNet

CHAPTER

TWENTYSEVEN

AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

27.1 Abstract

While the Transformer architecture has become the de-facto standard for natural language processing tasks, its applications to computer vision remain limited. In vision, attention is either applied in conjunction with convolutional networks, or used to replace certain components of convolutional networks while keeping their overall structure in place. We show that this reliance on CNNs is not necessary and a pure transformer applied directly to sequences of image patches can perform very well on image classification tasks. When pre-trained on large amounts of data and transferred to multiple mid-sized or small image recognition benchmarks (ImageNet, CIFAR-100, VTAB, etc.), Vision Transformer (ViT) attains excellent results compared to state-of-the-art convolutional networks while requiring substantially fewer computational resources to train.

27.2 Citation

```
@inproceedings{
  dosovitskiy2021an,
  title={An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale},
  author={Alexey Dosovitskiy and Lucas Beyer and Alexander Kolesnikov and Dirk␣
↪Weissenborn and Xiaohua Zhai and Thomas Unterthiner and Mostafa Dehghani and Matthias␣
↪Minderer and Georg Heigold and Sylvain Gelly and Jakob Uszkoreit and Neil Houlsby},
  booktitle={International Conference on Learning Representations},
  year={2021},
  url={https://openreview.net/forum?id=YicbFdNTTy}
}
```

The training step of Vision Transformers is divided into two steps. The first step is training the model on a large dataset, like ImageNet-21k, and get the pretrain model. And the second step is training the model on the target dataset, like ImageNet-1k, and get the finetune model. Here, we provide both pretrain models and finetune models.

27.3 Pretrain model

The pre-trained models are converted from model zoo of Google Research.

27.3.1 ImageNet 21k

*Models with * are converted from other repos.*

27.4 Finetune model

The finetune models are converted from model zoo of Google Research.

27.4.1 ImageNet 1k

Model	Pretrain	resolution	Params(M)	Flops(G)	Top-1 (%)	Top-5 (%)	Config	Download
ViT-B16*	ImageNet-21k	384x384	86.86	33.03	85.43	97.77	config	model
ViT-B32*	ImageNet-21k	384x384	88.30	8.56	84.01	97.08	config	model
ViT-L16*	ImageNet-21k	384x384	304.72	116.68	85.63	97.63	config	model

*Models with * are converted from other repos.*

CHAPTER

TWENTYEIGHT

PYTORCH TO ONNX (EXPERIMENTAL)

28.1 How to convert models from Pytorch to ONNX

28.1.1 Prerequisite

1. Please refer to install for installation of MMClassification.
2. Install onnx and onnxruntime

```
pip install onnx onnxruntime==1.5.1
```

28.1.2 Usage

```
python tools/deployment/pytorch2onnx.py \
    ${CONFIG_FILE} \
    --checkpoint ${CHECKPOINT_FILE} \
    --output-file ${OUTPUT_FILE} \
    --shape ${IMAGE_SHAPE} \
    --opset-version ${OPSET_VERSION} \
    --dynamic-export \
    --show \
    --simplify \
    --verify \
```

28.1.3 Description of all arguments:

- `config` : The path of a model config file.
- `--checkpoint` : The path of a model checkpoint file.
- `--output-file`: The path of output ONNX model. If not specified, it will be set to `tmp.onnx`.
- `--shape`: The height and width of input tensor to the model. If not specified, it will be set to `224 224`.
- `--opset-version` : The opset version of ONNX. If not specified, it will be set to `11`.
- `--dynamic-export` : Determines whether to export ONNX with dynamic input shape and output shapes. If not specified, it will be set to `False`.
- `--show`: Determines whether to print the architecture of the exported model. If not specified, it will be set to `False`.
- `--simplify`: Determines whether to simplify the exported ONNX model. If not specified, it will be set to `False`.
- `--verify`: Determines whether to verify the correctness of an exported model. If not specified, it will be set to `False`.

Example:

```
python tools/deployment/pytorch2onnx.py \
    configs/resnet/resnet18_8xb16_cifar10.py \
    --checkpoint checkpoints/resnet/resnet18_8xb16_cifar10.pth \
    --output-file checkpoints/resnet/resnet18_8xb16_cifar10.onnx \
    --dynamic-export \
    --show \
    --simplify \
    --verify \
```

28.2 How to evaluate ONNX models with ONNX Runtime

We prepare a tool `tools/deployment/test.py` to evaluate ONNX models with ONNXRuntime or TensorRT.

28.2.1 Prerequisite

- Install onnx and onnxruntime-gpu

```
pip install onnx onnxruntime-gpu
```

28.2.2 Usage

```
python tools/deployment/test.py \
    ${CONFIG_FILE} \
    ${ONNX_FILE} \
    --backend ${BACKEND} \
    --out ${OUTPUT_FILE} \
    --metrics ${EVALUATION_METRICS} \
    --metric-options ${EVALUATION_OPTIONS} \
    --show
    --show-dir ${SHOW_DIRECTORY} \
    --cfg-options ${CFG_OPTIONS} \
```

28.2.3 Description of all arguments

- `config`: The path of a model config file.
- `model`: The path of a ONNX model file.
- `--backend`: Backend for input model to run and should be `onnxruntime` or `tensorrt`.
- `--out`: The path of output result file in pickle format.
- `--metrics`: Evaluation metrics, which depends on the dataset, e.g., "accuracy", "precision", "recall", "f1_score", "support" for single label dataset, and "mAP", "CP", "CR", "CF1", "OP", "OR", "OF1" for multi-label dataset.
- `--show`: Determines whether to show classifier outputs. If not specified, it will be set to `False`.
- `--show-dir`: Directory where painted images will be saved
- `--metrics-options`: Custom options for evaluation, the key-value pair in `xxx=yyy` format will be kwargs for `dataset.evaluate()` function
- `--cfg-options`: Override some settings in the used config file, the key-value pair in `xxx=yyy` format will be merged into config file.

28.2.4 Results and Models

This part selects ImageNet for onnxruntime verification. ImageNet has multiple versions, but the most commonly used one is ILSVRC 2012.

28.3 List of supported models exportable to ONNX

The table below lists the models that are guaranteed to be exportable to ONNX and runnable in ONNX Runtime.

Notes:

- *All models above are tested with Pytorch==1.6.0*

28.4 Reminders

- If you meet any problem with the listed models above, please create an issue and it would be taken care of soon. For models not included in the list, please try to dig a little deeper and debug a little bit more and hopefully solve them by yourself.

28.5 FAQs

- None

CHAPTER

TWENTYNINE

ONNX TO TENSORRT (EXPERIMENTAL)

29.1 How to convert models from ONNX to TensorRT

29.1.1 Prerequisite

1. Please refer to install.md for installation of MMClassification from source.
2. Use our tool *pytorch2onnx.md* to convert the model from PyTorch to ONNX.

29.1.2 Usage

```
python tools/deployment/onnx2tensorrt.py \
    ${MODEL} \
    --trt-file ${TRT_FILE} \
    --shape ${IMAGE_SHAPE} \
    --max-batch-size ${MAX_BATCH_SIZE} \
    --workspace-size ${WORKSPACE_SIZE} \
    --fp16 \
    --show \
    --verify \
```

Description of all arguments:

- `model` : The path of an ONNX model file.
- `--trt-file`: The Path of output TensorRT engine file. If not specified, it will be set to `tmp.trt`.
- `--shape`: The height and width of model input. If not specified, it will be set to `224 224`.
- `--max-batch-size`: The max batch size of TensorRT model, should not be less than 1.

- `--fp16`: Enable fp16 mode.
- `--workspace-size` : The required GPU workspace size in GiB to build TensorRT engine. If not specified, it will be set to 1 GiB.
- `--show`: Determines whether to show the outputs of the model. If not specified, it will be set to `False`.
- `--verify`: Determines whether to verify the correctness of models between ONNXRuntime and TensorRT. If not specified, it will be set to `False`.

Example:

```
python tools/deployment/onnx2tensorrt.py \
    checkpoints/resnet/resnet18_b16x8_cifar10.onnx \
    --trt-file checkpoints/resnet/resnet18_b16x8_cifar10.trt \
    --shape 224 224 \
    --show \
    --verify \
```

29.2 List of supported models convertible to TensorRT

The table below lists the models that are guaranteed to be convertible to TensorRT.

Notes:

- *All models above are tested with Pytorch==1.6.0 and TensorRT-7.2.1.6.Ubuntu-16.04.x86_64-gnu.cuda-10.2.cudnn8.0*

29.3 Reminders

- If you meet any problem with the listed models above, please create an issue and it would be taken care of soon. For models not included in the list, we may not provide much help here due to the limited resources. Please try to dig a little deeper and debug by yourself.

29.4 FAQs

- None

CHAPTER

THIRTY

PYTORCH TO TORCHSCRIPT (EXPERIMENTAL)

30.1 How to convert models from Pytorch to TorchScript

30.1.1 Usage

```
python tools/deployment/pytorch2torchscript.py \
    ${CONFIG_FILE} \
    --checkpoint ${CHECKPOINT_FILE} \
    --output-file ${OUTPUT_FILE} \
    --shape ${IMAGE_SHAPE} \
    --verify \
```

30.1.2 Description of all arguments

- `config` : The path of a model config file.
- `--checkpoint` : The path of a model checkpoint file.
- `--output-file`: The path of output TorchScript model. If not specified, it will be set to `tmp.pt`.
- `--shape`: The height and width of input tensor to the model. If not specified, it will be set to `224 224`.
- `--verify`: Determines whether to verify the correctness of an exported model. If not specified, it will be set to `False`.

Example:

```
python tools/deployment/pytorch2onnx.py \
    configs/resnet/resnet18_8xb16_cifar10.py \
    --checkpoint checkpoints/resnet/resnet18_8xb16_cifar10.pth \
```

(continues on next page)

(continued from previous page)

```
    --output-file checkpoints/resnet/resnet18_8xb16_cifar10.pt \
    --verify \
```

Notes:

- *All models are tested with Pytorch==1.8.1*

30.2 Reminders

- For torch.jit.is_tracing() is only supported after v1.6. For users with pytorch v1.3-v1.5, we suggest early returning tensors manually.
- If you meet any problem with the models in this repo, please create an issue and it would be taken care of soon.

30.3 FAQs

- None

CHAPTER

THIRTYONE

MODEL SERVING

In order to serve an `MMClassification` model with `TorchServe`, you can follow the steps:

31.1 1. Convert model from MMClassification to TorchServe

```
python tools/deployment/mmcls2torchserve.py ${CONFIG_FILE} ${CHECKPOINT_FILE} \
--output-folder ${MODEL_STORE} \
--model-name ${MODEL_NAME}
```

Note: ${MODEL_STORE} needs to be an absolute path to a folder.

Example:

```
python tools/deployment/mmcls2torchserve.py \
  configs/resnet/resnet18_8xb32_in1k.py \
  checkpoints/resnet18_8xb32_in1k_20210831-fbbb1da6.pth \
  --output-folder ./checkpoints \
  --model-name resnet18_in1k
```

31.2 2. Build `mmcls-serve` docker image

```
docker build -t mmcls-serve:latest docker/serve/
```

31.3 3. Run `mmcls-serve`

Check the official docs for running TorchServe with docker.

In order to run in GPU, you need to install nvidia-docker. You can omit the `--gpus` argument in order to run in GPU.

Example:

```
docker run --rm \
--cpus 8 \
--gpus device=0 \
-p8080:8080 -p8081:8081 -p8082:8082 \
```

(continues on next page)

(continued from previous page)

```
--mount type=bind,source=`realpath ./checkpoints`,target=/home/model-server/model-store \
mmcls-serve:latest
```

Note: `realpath ./checkpoints` points to the absolute path of "./checkpoints", and you can replace it with the absolute path where you store torchserve models.

Read the docs about the Inference (8080), Management (8081) and Metrics (8082) APis

31.4 4. Test deployment

```
curl http://127.0.0.1:8080/predictions/${MODEL_NAME} -T demo/demo.JPEG
```

You should obtain a response similar to:

```
{
  "pred_label": 58,
  "pred_score": 0.38102269172668457,
  "pred_class": "water snake"
}
```

And you can use `test_torchserver.py` to compare result of TorchServe and PyTorch, and visualize them.

```
python tools/deployment/test_torchserver.py ${IMAGE_FILE} ${CONFIG_FILE} ${CHECKPOINT_
↪FILE} ${MODEL_NAME}
[--inference-addr ${INFERENCE_ADDR}] [--device ${DEVICE}]
```

Example:

```
python tools/deployment/test_torchserver.py \
  demo/demo.JPEG \
  configs/resnet/resnet18_8xb32_in1k.py \
  checkpoints/resnet18_8xb32_in1k_20210831-fbbb1da6.pth \
  resnet18_in1k
```

CHAPTER

THIRTYTWO

VISUALIZATION

- Visualization
  - Pipeline Visualization
  - Learning Rate Schedule Visualization
  - FAQs

32.1 Pipeline Visualization

```
python tools/visualizations/vis_pipeline.py \
    ${CONFIG_FILE} \
    --output-dir ${OUTPUT_DIR} \
    --phase ${DATASET_PHASE} \
    --number ${BUNBER_IMAGES_DISPLAY} \
    --skip-type ${SKIP_TRANSFORM_TYPE}
    --mode ${DISPLAY_MODE} \
    --show \
    --adaptive \
    --min-edge-length ${MIN_EDGE_LENGTH} \
    --max-edge-length ${MAX_EDGE_LENGTH} \
    --bgr2rgb \
    --window-size ${WINDOW_SIZE}
```

Description of all arguments:

- `config` : The path of a model config file.
- `--output-dir`: The output path for visualized images. If not specified, it will be set to `''`, which means not to save.
- `--phase`: Phase of visualizing dataset, must be one of `[train, val, test]`. If not specified, it will be set to `train`.
- `--number`: The number of samples to visualize. If not specified, display all images in the dataset.
- `--skip-type`: The pipelines to be skipped. If not specified, it will be set to `['ToTensor', 'Normalize', 'ImageToTensor', 'Collect']`.
- `--mode`: The display mode, can be one of `[original, pipeline, concat]`. If not specified, it will be set to `concat`.
- `--show`: If set, display pictures in pop-up windows.
- `--adaptive`: If set, automatically adjust the size of the visualization images.

- `--min-edge-length`: The minimum edge length, used when `--adaptive` is set. When any side of the picture is smaller than `${MIN_EDGE_LENGTH}`, the picture will be enlarged while keeping the aspect ratio unchanged, and the short side will be aligned to `${MIN_EDGE_LENGTH}`. If not specified, it will be set to 200.
- `--max-edge-length`: The maximum edge length, used when `--adaptive` is set. When any side of the picture is larger than `${MAX_EDGE_LENGTH}`, the picture will be reduced while keeping the aspect ratio unchanged, and the long side will be aligned to `${MAX_EDGE_LENGTH}`. If not specified, it will be set to 1000.
- `--bgr2rgb`: If set, flip the color channel order of images.
- `--window-size`: The shape of the display window. If not specified, it will be set to `12*7`. If used, it must be in the format `'W*H'`.

Note:

1. If the `--mode` is not specified, it will be set to `concat` as default, get the pictures stitched together by original pictures and transformed pictures; if the `--mode` is set to `original`, get the original pictures; if the `--mode` is set to `pipeline`, get the transformed pictures.
2. When `--adaptive` option is set, images that are too large or too small will be automatically adjusted, you can use `--min-edge-length` and `--max-edge-length` to set the adjust size.

Examples:

1. Visualize all the transformed pictures of the `ImageNet` training set and display them in pop-up windows:

```
python ./tools/visualizations/vis_pipeline.py ./configs/resnet/resnet50_8xb32_in1k.py --
↪show --mode pipeline
```

2. Visualize 10 comparison pictures in the `ImageNet` train set and save them in the `./tmp` folder:

```
python ./tools/visualizations/vis_pipeline.py configs/swin_transformer/swin_base_224_
↪b16x64_300e_imagenet.py --phase train --output-dir tmp --number 10 --adaptive
```

3. Visualize 100 original pictures in the `CIFAR100` validation set, then display and save them in the `./tmp` folder:

```
python ./tools/visualizations/vis_pipeline.py configs/resnet/resnet50_8xb16_cifar100.py -
↪-phase val --output-dir tmp --mode original --number 100  --show --adaptive --bgr2rgb
```

32.2 Learning Rate Schedule Visualization

```
python tools/visualizations/vis_lr.py \
    ${CONFIG_FILE} \
    --dataset-size ${DATASET_SIZE} \
    --ngpus ${NUM_GPUs}
    --save-path ${SAVE_PATH} \
    --title ${TITLE} \
    --style ${STYLE} \
    --window-size ${WINDOW_SIZE}
    --cfg-options
```

Description of all arguments:

- `config` : The path of a model config file.

- `dataset-size` : The size of the datasets. If set， `build_dataset` will be skipped and `${DATASET_SIZE}` will be used as the size. Default to use the function `build_dataset`.
- `ngpus` : The number of GPUs used in training, default to be 1.
- `save-path` : The learning rate curve plot save path, default not to save.
- `title` : Title of figure. If not set, default to be config file name.
- `style` : Style of plt. If not set, default to be `whitegrid`.
- `window-size`: The shape of the display window. If not specified, it will be set to `12*7`. If used, it must be in the format `'W*H'`.
- `cfg-options` : Modifications to the configuration file, refer to Tutorial 1: Learn about Configs.

Note: Loading annotations maybe consume much time, you can directly specify the size of the dataset with `dataset-size` to save time.

Examples:

```
python tools/visualizations/vis_lr.py configs/resnet/resnet50_b16x8_cifar100.py
```

When using ImageNet, directly specify the size of ImageNet, as below:

```
python tools/visualizations/vis_lr.py configs/repvgg/repvgg-B3g4_4xb64-autoaug-lbs-mixup-
↪coslr-200e_in1k.py --dataset-size 1281167 --ngpus 4 --save-path ./repvgg-B3g4_4xb64-lr.
↪jpg
```

32.3 FAQs

- None

CHAPTER

THIRTYTHREE

CONTRIBUTING TO OPENMMLAB

All kinds of contributions are welcome, including but not limited to the following.

- Fixes (typo, bugs)
- New features and components

33.1 Workflow

1. fork and pull the latest OpenMMLab repository (mmclassification)
2. checkout a new branch (do not use master branch for PRs)
3. commit your changes
4. create a PR

Note: If you plan to add some new features that involve large changes, it is encouraged to open an issue for discussion first.

33.2 Code style

33.2.1 Python

We adopt PEP8 as the preferred code style.

We use the following tools for linting and formatting:

- flake8: A wrapper around some linter tools.
- yapf: A formatter for Python files.
- isort: A Python utility to sort imports.
- markdownlint: A linter to check markdown files and flag style issues.
- docformatter: A formatter to format docstring.

Style configurations of yapf and isort can be found in setup.cfg.

We use pre-commit hook that checks and formats for `flake8`, `yapf`, `isort`, `trailing whitespaces`, `markdown files`, fixes `end-of-files`, `double-quoted-strings`, `python-encoding-pragma`, `mixed-line-ending`, sorts `requirments.txt` automatically on every commit. The config for a pre-commit hook is stored in .pre-commit-config.

After you clone the repository, you will need to install initialize pre-commit hook.

```
pip install -U pre-commit
```

From the repository folder

```
pre-commit install
```

Try the following steps to install ruby when you encounter an issue on installing markdownlint

```
# install rvm
curl -L https://get.rvm.io | bash -s -- --autolibs=read-fail
[[ -s "$HOME/.rvm/scripts/rvm" ]] && source "$HOME/.rvm/scripts/rvm"
rvm autolibs disable

# install ruby
rvm install 2.7.1
```

Or refer to this repo and take `zzruby.sh` according its instruction.

After this on every commit check code linters and formatter will be enforced.

Important: Before you create a PR, make sure that your code lints and is formatted by yapf.

33.2.2 C++ and CUDA

We follow the Google C++ Style Guide.

CHAPTER

THIRTYFOUR

MMCLS.APIS

CHAPTER

THIRTYFIVE

MMCLS.CORE

35.1 evaluation

CHAPTER

THIRTYSIX

MMCLS.MODELS

36.1 models

36.2 classifiers

36.3 backbones

36.4 necks

36.5 heads

36.6 losses

36.7 utils

CHAPTER

THIRTYSEVEN

MMCLS.DATASETS

37.1 datasets

37.2 pipelines

CHAPTER

THIRTYEIGHT

MMCLS.UTILS

CHAPTER

THIRTYNINE

INDICES AND TABLES

- genindex
- search