MMSelfSup

Release 0.11.0

MMSelfSup Authors

Apr 06, 2023

GET STARTED

CHAPTER

ONE

PREREQUISITES

In this section we demonstrate how to prepare an environment with PyTorch.

MMselfSup works on Linux (Windows and macOS are not officially supported). It requires Python 3.6+, CUDA 9.2+ and PyTorch 1.5+.

If you are experienced with PyTorch and have already installed it, just skip this part and jump to the *next section*. Otherwise, you can follow these steps for the preparation.

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

Step 1. Create a conda environment and activate it.

```
conda create --name openmmlab python=3.8 -y
conda activate openmmlab
```

Step 2. Install PyTorch following official instructions, e.g.

On GPU platforms:

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

On CPU platforms:

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

CHAPTER

TWO

INSTALLATION

We recommend that users follow our best practices to install MMSelfSup. However, the whole process is highly customizable. See *Customize Installation* section for more information.

2.1 Best Practices

Step 0. Install MMCV using MIM.

```
pip install -U openmim
mim install mmcv-full
```

Step 1. Install MMSelfSup.

Case a: If you develop and run mmselfsup directly, install it from source:

```
git clone https://github.com/open-mmlab/mmselfsup.git
cd mmselfsup
pip install -v -e .
# "-v" means verbose, or more output
# "-e" means installing a project in editable mode,
# thus any local modifications made to the code will take effect without reinstallation.
```

Case b: If you use mmselfsup as a dependency or third-party package, install it with pip:

```
pip install mmselfsup
```

2.2 Verify the installation

To verify whether MMSelfSup is installed correctly, we can run the following sample code to initialize a model and inference a demo image.

```
import torch

from mmselfsup.models import build_algorithm

model_config = dict(
    type='Classification',
    backbone=dict(
        type='ResNet',
```

(continues on next page)

(continued from previous page)

```
        depth=50,
        in_channels=3,
        num_stages=4,
        strides=(1, 2, 2, 2),
        dilations=(1, 1, 1, 1),
        out_indices=[4],  # 0: conv-1, x: stage-x
        norm_cfg=dict(type='BN'),
        frozen_stages=-1),
    head=dict(
        type='ClsHead', with_avg_pool=True, in_channels=2048,
        num_classes=1000))

model = build_algorithm(model_config).cuda()

image = torch.randn((1, 3, 224, 224)).cuda()
label = torch.tensor([1]).cuda()

loss = model.forward_train(image, label)
```

The above code is supposed to run successfully upon you finish the installation.

2.3 Customized installation

2.3.1 Benchmark

The *Best Practices* is for basic usage, if you need to evaluate your pre-training model with some downstream tasks such as detection or segmentation, please also install MMDetection and MMSegmentation.

If you don't run MMDetection and MMSegmentation benchmark, it is unnecessary to install them.

You can simply install MMDetection and MMSegmentation with the following command:

```
pip install mmdet mmsegmentation
```

For more details, you can check the installation page of MMDetection and MMSegmentation.

2.3.2 CUDA versions

When installing PyTorch, you need to specify the version of CUDA. If you are not clear on which to choose, follow our recommendations:

- For Ampere-based NVIDIA GPUs, such as GeForce 30 series and NVIDIA A100, CUDA 11 is a must.
- For older NVIDIA GPUs, CUDA 11 is backward compatible, but CUDA 10.2 offers better compatibility and is more lightweight.

Please make sure the GPU driver satisfies the minimum version requirements. See this table for more information.

Note: Installing CUDA runtime libraries is enough if you follow our best practices, because no CUDA code will be compiled locally. However if you hope to compile MMCV from source or develop other CUDA operators, you need to

install the complete CUDA toolkit from NVIDIA's website, and its version should match the CUDA version of PyTorch. i.e., the specified version of cudatoolkit in `conda install` command.

2.3.3 Install MMCV without MIM

MMCV contains C++ and CUDA extensions, thus depending on PyTorch in a complex way. MIM solves such dependencies automatically and makes the installation easier. However, it is not a must.

To install MMCV with pip instead of MIM, please follow MMCV installation guides. This requires manually specifying a find-url based on PyTorch version and its CUDA version.

For example, the following command install mmcv-full built for PyTorch 1.10.x and CUDA 11.3.

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu113/torch1.10/index.
↪html
```

2.3.4 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 mmselfsup:torch1.10.0-cuda11.3-cudnn8 .
```

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

Run the following cmd:

```
docker run --gpus all --shm-size=8g -it -v {DATA_DIR}:/workspace/mmselfsup/data␣
↪mmselfsup:torch1.10.0-cuda11.3-cudnn8 /bin/bash
```

`{DATA_DIR}` is your local folder containing all these datasets.

2.3.5 Install on Google Colab

Google Colab usually has PyTorch installed, thus we only need to install MMCV and MMSeflSup with the following commands.

Step 0. Install MMCV using MIM.

```
!pip3 install openmim
!mim install mmcv-full
```

Step 1. Install MMSelfSup from the source.

```
!git clone https://github.com/open-mmlab/mmselfsup.git
%cd mmselfsup
!pip install -e .
```

Step 2. Verification.

```
import mmselfsup
print(mmselfsup.__version__)
# Example output: 0.9.0
```

Note: Within Jupyter, the exclamation mark `!` is used to call external executables and `%cd` is a magic command to change the current working directory of Python.

2.4 Trouble shooting

If you have some issues during the installation, please first view the FAQ page. You may open an issue on GitHub if no solution is found.

CHAPTER

THREE

USING MULTIPLE MMSELFSUP VERSIONS

If there are more than one mmselfsup on your machine, and you want to use them alternatively, the recommended way is to create multiple conda environments and use different environments for different versions.

Another way is to insert the following code to the main scripts (`train.py`, `test.py` or any other scripts you run)

```
import os.path as osp
import sys
sys.path.insert(0, osp.join(osp.dirname(osp.abspath(__file__)), '../'))
```

Or run the following command in the terminal of corresponding root folder to temporally use the current one.

```
export PYTHONPATH="$(pwd)":$PYTHONPATH
```

CHAPTER

FOUR

PREPARE DATASETS

MMSelfSup supports multiple datasets. Please follow the corresponding guidelines for data preparation. It is recommended to symlink your dataset root to `$MMSELFSUP/data`. If your folder structure is different, you may need to change the corresponding paths in config files.

- *Prepare ImageNet*
- *Prepare Place205*
- *Prepare iNaturalist2018*
- *Prepare PASCAL VOC*
- *Prepare CIFAR10*
- *Prepare datasets for detection and segmentation*
  - *Detection*
  - *Segmentation*

```
mmselfsup
├── mmselfsup
├── tools
├── configs
├── docs
├── data
    ├── imagenet
        ├── meta
        ├── train
        ├── val
    ├── places205
        ├── meta
        ├── train
        ├── val
    ├── inaturalist2018
        ├── meta
        ├── train
        ├── val
    ├── VOCdevkit
        ├── VOC2007
    ├── cifar
        ├── cifar-10-batches-py
```

4.1 Prepare ImageNet

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

4.2 Prepare Place205

For Places205, you need to:

1. Register an account and login to the download page
2. Download the resized images and the image list of train set and validation set of Places205
3. Untar the downloaded files

4.3 Prepare iNaturalist2018

For iNaturalist2018, you need to:

1. Download the training and validation images and annotations from the download page
2. Untar the downloaded files
3. Convert the original json annotation format to the list format using the script `tools/data_converters/convert_inaturalist.py`

4.4 Prepare PASCAL VOC

Assuming that you usually store datasets in `$YOUR_DATA_ROOT`. The following command will automatically download PASCAL VOC 2007 into `$YOUR_DATA_ROOT`, prepare the required files, create a folder `data` under `$MMSELFSUP` and make a symlink `VOCdevkit`.

```
bash tools/data_converters/prepare_voc07_cls.sh $YOUR_DATA_ROOT
```

4.5 Prepare CIFAR10

CIFAR10 will be downloaded automatically if it is not found. In addition, `dataset` implemented by `MMSelfSup` will also automatically structure CIFAR10 to the appropriate format.

4.6 Prepare datasets for detection and segmentation

4.6.1 Detection

To prepare COCO, VOC2007 and VOC2012 for detection, you can refer to mmdet.

4.6.2 Segmentation

To prepare VOC2012AUG and Cityscapes for segmentation, you can refer to mmseg

CHAPTER

FIVE

GETTING STARTED

This page provides basic tutorials about the usage of MMSelfSup. For installation instructions, please see *install.md*.

5.1 Train existing methods

Note: The default learning rate in config files is for 8 GPUs. If using different number GPUs, the total batch size will change in proportion, you have to scale the learning rate following `new_lr = old_lr * new_ngpus / old_ngpus`. We recommend to use `tools/dist_train.sh` even with 1 gpu, since some methods do not support non-distributed training.

5.1.1 Training with CPU

```
export CUDA_VISIBLE_DEVICES=-1
python tools/train.py ${CONFIG_FILE}
```

Note: We do not recommend users to use CPU for training because it is too slow and some algorithms are using `SyncBN` which is based on distributed training. We support this feature to allow users to debug on machines without GPU for convenience.

5.1.2 Train with single/multiple GPUs

```
bash tools/dist_train.sh ${CONFIG_FILE} ${GPUS} --work-dir ${YOUR_WORK_DIR} [optional␣
↪arguments]
```

Optional arguments are:

- `--resume-from ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.
- `--deterministic`: Switch on "deterministic" mode which slows down training but the results are reproducible.

An example:

```
# checkpoints and logs saved in WORK_DIR=work_dirs/selfsup/odc/odc_resnet50_8xb64-steplr-
↪440e_in1k/
bash tools/dist_train.sh configs/selfsup/odc/odc_resnet50_8xb64-steplr-440e_in1k.py 8 --
↪work-dir work_dirs/selfsup/odc/odc_resnet50_8xb64-steplr-440e_in1k/
```

Note: During training, checkpoints and logs are saved in the same folder structure as the config file under `work_dirs/`. Custom work directory is not recommended since evaluation scripts infer work directories from the config file name. If you want to save your weights somewhere else, please use symlink, for example:

```
ln -s ${YOUR_WORK_DIRS} ${MMSELFSUP}/work_dirs
```

Alternatively, if you run MMSelfSup on a cluster managed with slurm:

```
GPUS_PER_NODE=${GPUS_PER_NODE} GPUS=${GPUS} SRUN_ARGS=${SRUN_ARGS} bash tools/slurm_
↪train.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${YOUR_WORK_DIR} [optional arguments]
```

An example:

```
GPUS_PER_NODE=8 GPUS=8 bash tools/slurm_train.sh Dummy Test_job configs/selfsup/odc/odc_
↪resnet50_8xb64-steplr-440e_in1k.py work_dirs/selfsup/odc/odc_resnet50_8xb64-steplr-
↪440e_in1k/
```

5.1.3 Train with multiple machines

If you launch with multiple machines simply connected with ethernet, you can simply run following commands:

On the first machine:

```
NNODES=2 NODE_RANK=0 PORT=$MASTER_PORT MASTER_ADDR=$MASTER_ADDR bash tools/dist_train.sh
↪$CONFIG $GPUS
```

On the second machine:

```
NNODES=2 NODE_RANK=1 PORT=$MASTER_PORT MASTER_ADDR=$MASTER_ADDR bash tools/dist_train.sh
↪$CONFIG $GPUS
```

Usually it is slow if you do not have high speed networking like InfiniBand.

If you launch with slurm, the command is the same as that on single machine described above, but you need refer to slurm_train.sh to set appropriate parameters and environment variables.

5.1.4 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:

```
CUDA_VISIBLE_DEVICES=0,1,2,3 PORT=29500 sh tools/dist_train.sh ${CONFIG_FILE} 4 --work-
↪dir tmp_work_dir_1
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 sh tools/dist_train.sh ${CONFIG_FILE} 4 --work-
↪dir tmp_work_dir_2
```

If you use launch training jobs with slurm, you have two options to set different communication ports:

Option 1:

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 and config2.py.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py tmp_work_dir_1
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py tmp_work_dir_2
```

Option 2:

You can set different communication ports without the need to modify the configuration file, but have to set the `cfg-options` to overwrite the default port in configuration file.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py tmp_work_dir_1 --cfg-options dist_params.port=29500
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py tmp_work_dir_2 --cfg-options dist_params.port=29501
```

5.2 Benchmarks

We also provide commands to evaluate your pre-trained model on several downstream task, and you can refer to *Benchmarks* for the details.

5.3 Tools and Tips

5.3.1 Count number of parameters

```
python tools/analysis_tools/count_parameters.py ${CONFIG_FILE}
```

5.3.2 Publish a model

Before you publish a model, you may want to

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

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

5.3.3 Use t-SNE

We provide an off-the-shelf tool to visualize the quality of image representations by t-SNE.

```
python tools/analysis_tools/visualize_tsne.py ${CONFIG_FILE} --checkpoint ${CKPT_PATH} --
↪work-dir ${WORK_DIR} [optional arguments]
```

Arguments:

- `CONFIG_FILE`: config file for the pre-trained model.
- `CKPT_PATH`: the path of model's checkpoint.
- `WORK_DIR`: the directory to save the results of visualization.
- `[optional arguments]`: for optional arguments, you can refer to visualize_tsne.py

5.3.4 MAE Visualization

We provide a tool to visualize the mask and reconstruction image of MAE model.

```
python tools/misc/mae_visualization.py ${IMG_PATH} ${CONFIG_FILE} ${CKPT_PATH} ${OUT_
↪FILE} --device ${DEVICE}
```

Arguments:

- `IMG_PATH`: an image path used for visualization.
- `CONFIG_FILE`: config file for the pre-trained model.
- `CKPT_PATH`: the path of model's checkpoint.
- `OUT_FILE`: the image path used for visualization results.
- `DEVICE`: device used for inference.

An example:

```
python tools/misc/mae_visualization.py tests/data/color.jpg configs/selfsup/mae/mae_vit-
↪base-p16_8xb512-coslr-400e_in1k.py mae_epoch_400.pth results.jpg --device 'cuda:0'
```

5.3.5 Reproducibility

If you want to make your performance exactly reproducible, please switch on `--deterministic` to train the final model to be published. Note that this flag will switch off `torch.backends.cudnn.benchmark` and slow down the training speed.

CHAPTER

SIX

MODEL ZOO

All models and part of benchmark results are recorded below.

6.1 Pre-trained models

Remarks:

- The training details are recorded in the config names.
- You can click algorithm name to obtain more information.

6.2 Benchmarks

In the following tables, we only display ImageNet linear evaluation, ImageNet fine-tuning, COCO17 object detection and instance segmentation, and PASCAL VOC12 Aug semantic segmentation. You can click algorithm name above to check more comprehensive benchmark results.

6.2.1 ImageNet Linear Evaluation

If not specified, we use linear evaluation setting from MoCo as default. Other settings are mentioned in Remarks.

6.2.2 ImageNet Fine-tuning

6.2.3 COCO17 Object Detection and Instance Segmentation

In COCO17 object detection and instance segmentation task, we choose the evaluation protocol from MoCo, with Mask-RCNN FPN architecture. The results below are fine-tuned with the same config.

6.2.4 Pascal VOC12 Aug Semantic Segmentation

In Pascal VOC12 Aug semantic segmentation task, we choose the evaluation protocol from MMSeg, with FCN architecture. The results below are fine-tuned with the same config.

CHAPTER

SEVEN

TUTORIAL 0: LEARN ABOUT CONFIGS

MMSelfSup 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. If you wish to inspect the config file in summary, you may run `python tools/misc/print_config.py` to see the complete config.

7.1 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}_{module}_{training_info}_{data_info}.py
```

- `algorithm info`Algorithm information includes algorithm name, such as simclr, mocov2, etc.;
- `module info` Module information is used to represent some backbone, neck, head 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.;

7.1.1 Algorithm information

```
{algorithm}-{misc}
```

`Algorithm` means the abbreviation from the paper and its version. E.g:

- `relative-loc` : The different word is concatenated by dashes `'-'`
- `simclr`
- `mocov2`

`misc` offers some other algorithm related information. E.g.

- `npid-ensure-neg`
- `deepcluster-sobel`

7.1.2 Module information

```
{backbone setting}-{neck setting}-{head_setting}
```

The module information mainly includes the backbone information. E.g:

- `resnet50`
- `vit`will be used in mocov3

Or there are some special settings which is needed to be mentioned in the config name. E.g:

- `resnet50-nofrz`: In some downstream tasksthe backbone will not froze stages while training

7.1.3 Training information

Training related settingsincluding batch size, lr schedule, data augment, etc.

- Batch size, the format is `{gpu x batch_per_gpu}`like `8xb32`;
- Training recipethe methods will be arranged in the order `{pipeline aug}-{train aug}-{loss trick}-{scheduler}-{epochs}`.

E.g:

- `8xb32-mcrop-2-6-coslr-200e` : `mcrop` is proposed in SwAV named multi-croppart of pipeline. 2 and 6 means that 2 pipelines will output 2 and 6 crops correspondinglythe crop size is recorded in data information;
- `8xb32-accum16-coslr-200e` : `accum16` means the gradient will accumulate for 16 iterationsthen the weights will be updated.

7.1.4 Data information

Data information contains the dataset, input size, etc. E.g:

- `in1k` : `ImageNet1k` dataset, default to use the input image size of 224x224
- `in1k-384px` : Indicates that the input image size is 384x384
- `cifar10`
- `inat18` : `iNaturalist2018` datasetit has 8142 classes
- `places205`

7.1.5 Config File Name Example

```
swav_resnet50_8xb32-mcrop-2-6-coslr-200e_in1k-224-96.py
```

- `swav`: Algorithm information
- `resnet50`: Module information
- `8xb32-mcrop-2-6-coslr-200e`: Training information
  - `8xb32`: Use 8 GPUs in totaland the batch size is 32 per GPU
  - `mcrop-2-6`:Use multi-crop data augment method
  - `coslr`: Use cosine learning rate scheduler
  - `200e`: Train the model for 200 epoch
- `in1k-224-96`: Data informationtrain on ImageNet1k datasetthe input sizes are 224x224 and 96x96

7.1.6 Checkpoint Naming Convention

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

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

7.2 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 MoCo v2 as a example and comment the meaning of each line. For more detaile, please refer to the API documentation.

The config file `configs/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-200e_in1k.py` is displayed below.

```
_base_ = [
    '../_base_/models/mocov2.py',                    # model
    '../_base_/datasets/imagenet_mocov2.py',         # data
    '../_base_/schedules/sgd_coslr-200e_in1k.py',    # training schedule
    '../_base_/default_runtime.py',                  # runtime setting
]

# Here we inherit runtime settings and modify the max_keep_ckpts.
# the max_keep_ckpts controls the max number of ckpt file in your work_dirs
# if it is 3, when CheckpointHook (in mmcv) saves the 4th ckpt
# it will remove the oldest one to keep the number of total ckpts as 3
checkpoint_config = dict(interval=10, max_keep_ckpts=3)
```

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

`../_base_/models/mocov2.py` is the base model config for MoCo v2.

```
model = dict(
    type='MoCo',  # Algorithm name
    queue_len=65536,  # Number of negative keys maintained in the queue
    feat_dim=128,  # Dimension of compact feature vectors, equal to the out_channels of␣
↪the neck
    momentum=0.999,  # Momentum coefficient for the momentum-updated encoder
    backbone=dict(
        type='ResNet',  # Backbone name
        depth=50,  # Depth of backbone, ResNet has options of 18, 34, 50, 101, 152
        in_channels=3,  # The channel number of the input images
        out_indices=[4],  # The output index of the output feature maps, 0 for conv-1, x␣
↪for stage-x
        norm_cfg=dict(type='BN')),  # Dictionary to construct and config norm layer
    neck=dict(
        type='MoCoV2Neck',  # Neck name
        in_channels=2048,  # Number of input channels
        hid_channels=2048,  # Number of hidden channels
        out_channels=128,  # Number of output channels
        with_avg_pool=True),  # Whether to apply the global average pooling after␣
↪backbone
    head=dict(
        type='ContrastiveHead',  # Head name, indicates that the MoCo v2 use contrastive␣
↪loss
        temperature=0.2))  # The temperature hyper-parameter that controls the␣
↪concentration level of the distribution.
```

`../_base_/datasets/imagenet_mocov2.py` is the base dataset config for MoCo v2.

```
# dataset settings
data_source = 'ImageNet'  # data source name
dataset_type = 'MultiViewDataset' # dataset type is related to the pipeline composing
img_norm_cfg = dict(
    mean=[0.485, 0.456, 0.406],  # Mean values used to pre-training the pre-trained␣
↪backbone models
    std=[0.229, 0.224, 0.225])  # Standard variance used to pre-training the pre-trained␣
↪backbone models
```

(continues on next page)

(continued from previous page)

```
# The difference between mocov2 and mocov1 is the transforms in the pipeline
train_pipeline = [
    dict(type='RandomResizedCrop', size=224, scale=(0.2, 1.)),  # RandomResizedCrop
    dict(
        type='RandomAppliedTrans',  # Random apply ColorJitter augment method with␣
↪probability 0.8
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.4,
                contrast=0.4,
                saturation=0.4,
                hue=0.1)
        ],
        p=0.8),
    dict(type='RandomGrayscale', p=0.2),  # RandomGrayscale with probability 0.2
    dict(type='GaussianBlur', sigma_min=0.1, sigma_max=2.0, p=0.5),  # Random␣
↪GaussianBlur with probability 0.5
    dict(type='RandomHorizontalFlip'),  # Randomly flip the picture horizontally
]

# prefetch
prefetch = False  # Whether to using prefetch to speed up the pipeline
if not prefetch:
    train_pipeline.extend(
        [dict(type='ToTensor'),
         dict(type='Normalize', **img_norm_cfg)])

# dataset summary
data = dict(
    samples_per_gpu=32,  # Batch size of a single GPU, total 32*8=256
    workers_per_gpu=4,  # Worker to pre-fetch data for each single GPU
    drop_last=True,  # Whether to drop the last batch of data
    train=dict(
        type=dataset_type,  # dataset name
        data_source=dict(
            type=data_source,  # data source name
            data_prefix='data/imagenet/train',  # Dataset root, when ann_file does not␣
↪exist, the category information is automatically obtained from the root folder
            ann_file='data/imagenet/meta/train.txt',  #  ann_file existes, the category␣
↪information is obtained from file
        ),
        num_views=[2],  # The number of different views from pipeline
        pipelines=[train_pipeline],  # The train pipeline
        prefetch=prefetch,  # The boolean value
    ))
```

`../_base_/schedules/sgd_coslr-200e_in1k.py` is the base schedule config for MoCo v2.

```
# optimizer
optimizer = dict(
    type='SGD',  # Optimizer type
```

(continues on next page)

(continued from previous page)

```
    lr=0.03,  # Learning rate of optimizers, see detail usages of the parameters in the␣
↪documentation of PyTorch
    weight_decay=1e-4,  # Momentum parameter
    momentum=0.9)  # 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()  # this config can set grad_clip, coalesce, bucket_size_mb,␣
↪etc.

# learning policy
# Learning rate scheduler config used to register LrUpdater hook
lr_config = dict(
    policy='CosineAnnealing',  # The policy of scheduler, also support Step, Cyclic, etc.
↪ Refer to details of supported LrUpdater from https://github.com/open-mmlab/mmcv/blob/
↪master/mmcv/runner/hooks/lr_updater.py#L9.
    min_lr=0.)  # The minimum lr setting in CosineAnnealing

# runtime settings
runner = dict(
    type='EpochBasedRunner',  # Type of runner to use (i.e. IterBasedRunner or␣
↪EpochBasedRunner)
    max_epochs=200) # Runner that runs the workflow in total max_epochs. For␣
↪IterBasedRunner use `max_iters`
```

../_base_/default_runtime.py is the default runtime settings.

```
# checkpoint saving
checkpoint_config = dict(interval=10)  # The save interval is 10

# yapf:disable
log_config = dict(
    interval=50,  # Interval to print the log
    hooks=[
        dict(type='TextLoggerHook'),  # The Tensorboard logger is also supported
        # dict(type='TensorboardLoggerHook'),
    ])
# yapf:enable

# runtime settings
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.
load_from = None  # Runner to load ckpt
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.
persistent_workers = True  # The boolean type to set persistent_workers in Dataloader.␣
↪see detail in the documentation of PyTorch
```

7.3 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 MoCo v2 with some other modification, you can first inherit the basic MoCo v2 structure, dataset and other training setting by specifying `_base_ ='./mocov2_resnet50_8xb32-coslr-200e_in1k.py.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/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-200e_in1k.py.py`, but change the number of training epochs from 200 to 800, modify when to decay the learning rate, and modify the dataset path, you can create a new config file `configs/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-800e_in1k.py.py` with content as below:

```
_base_ = './mocov2_resnet50_8xb32-coslr-200e_in1k.py'

runner = dict(max_epochs=800)
```

7.3.1 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, `data_source`, `dataset_type`, `train_pipeline`, `prefetch` are the intermediate variables of the data. We first need to define them and then pass them to `data`.

```
data_source = 'ImageNet'
dataset_type = 'MultiViewDataset'
img_norm_cfg = dict(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
train_pipeline = [...]

# prefetch
prefetch = False  # Whether to using prefetch to speed up the pipeline
if not prefetch:
    train_pipeline.extend(
        [dict(type='ToTensor'),
         dict(type='Normalize', **img_norm_cfg)])

# dataset summary
data = dict(
    samples_per_gpu=32,
    workers_per_gpu=4,
    drop_last=True,
    train=dict(type=dataset_type, type=data_source, data_prefix=...),
        num_views=[2],
        pipelines=[train_pipeline],
        prefetch=prefetch,
    ))
```

7.3.2 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 want to use `MoCoV2Neck` in simclr setting, just using inheritance and directly modify it will report `get unexcepected keyword 'num_layers'` error, because the `'num_layers'` field of the basic config in `model.neck` domain information is reserved, and you need to add `_delete_=True` to ignore the content of `model.neck` related fields in the basic configuration file:

```
_base_ = 'simclr_resnet50_8xb32-coslr-200e_in1k.py'

model = dict(
    neck=dict(
        _delete_=True,
        type='MoCoV2Neck',
        in_channels=2048,
        hid_channels=2048,
        out_channels=128,
        with_avg_pool=True))
```

7.3.3 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/selfsup/odc/odc_resnet50_8xb64-steplr-440e_in1k.py`. When defining `num_classes`, just add the definition file name of auto augment to `_base_`, and then use `{{_base_.num_classes}}` to reference the variables:

```
_base_ = [
    '../_base_/models/odc.py',
    '../_base_/datasets/imagenet_odc.py',
    '../_base_/schedules/sgd_steplr-200e_in1k.py',
    '../_base_/default_runtime.py',
]

# model settings
model = dict(
    head=dict(num_classes={{_base_.num_classes}}),
    memory_bank=dict(num_classes={{_base_.num_classes}}),
)

# optimizer
optimizer = dict(
    type='SGD',
    lr=0.06,
    momentum=0.9,
    weight_decay=1e-5,
    paramwise_options={'\\Ahead.': dict(momentum=0.)})

# learning policy
lr_config = dict(policy='step', step=[400], gamma=0.4)
```

(continues on next page)

(continued from previous page)

```
# runtime settings
runner = dict(type='EpochBasedRunner', max_epochs=440)
# the max_keep_ckpts controls the max number of ckpt file in your work_dirs
# if it is 3, when CheckpointHook (in mmcv) saves the 4th ckpt
# it will remove the oldest one to keep the number of total ckpts as 3
checkpoint_config = dict(interval=10, max_keep_ckpts=3)
```

7.4 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.

7.5 Import user-defined modules

Note: This part may only be used when using other MM-codebase, like mmcls as a third party library to build your own project, and beginners can skip it.

You may use other MM-codebase 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 MM-codebase 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',
```

(continues on next page)

(continued from previous page)

```
            'your_module_class'],
    allow_failed_imports=False)
```

CHAPTER

EIGHT

TUTORIAL 1: ADDING NEW DATASET

In this tutorial, we introduce the basic steps to create your customized dataset:

If your algorithm does not need any customized dataset, you can use these off-the-shelf datasets under datasets. But to use these existing datasets, you have to convert your dataset to existing dataset format.

8.1 An example of customized dataset

Assuming the format of your dataset's annotation file is:

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

To write a new dataset, you need to implement:

- `DataSource`: inherited from `BaseDataSource` and responsible for loading the annotation files and reading images.
- `Dataset`: inherited from `BaseDataset` and responsible for applying transformation to images and packing these images.

8.2 Creating the `DataSource`

Assume the name of your `DataSource` is `NewDataSource`, you can create a file, named `new_data_source.py` under `mmselfsup/datasets/data_sources` and implement `NewDataSource` in it.

```
import mmcv
import numpy as np

from ..builder import DATASOURCES
from .base import BaseDataSource
```

(continues on next page)

(continued from previous page)

```
@DATASOURCES.register_module()
class NewDataSource(BaseDataSource):

    def load_annotations(self):

        assert isinstance(self.ann_file, str)
        data_infos = []
        # writing your code here.
        return data_infos
```

Then, add NewDataSource in mmselfsup/dataset/data_sources/__init__.py.

```
from .base import BaseDataSource
...
from .new_data_source import NewDataSource

__all__ = [
    'BaseDataSource', ..., 'NewDataSource'
]
```

8.3 Creating the Dataset

Assume the name of your Dataset is NewDataset, you can create a file, named new_dataset.py under mmselfsup/datasets and implement NewDataset in it.

```
# Copyright (c) OpenMMLab. All rights reserved.
import torch
from mmcv.utils import build_from_cfg
from torchvision.transforms import Compose

from .base import BaseDataset
from .builder import DATASETS, PIPELINES, build_datasource
from .utils import to_numpy


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

    def __init__(self, data_source, num_views, pipelines, prefetch=False):
        # writing your code here
    def __getitem__(self, idx):
        # writing your code here
        return dict(img=img)

    def evaluate(self, results, logger=None):
        return NotImplemented
```

Then, add NewDataset in mmselfsup/dataset/__init__.py.

```
from .base import BaseDataset
...
from .new_dataset import NewDataset

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

8.4 Modify config file

To use `NewDataset`, you can modify the config as the following:

```
train=dict(
        type='NewDataset',
        data_source=dict(
            type='NewDataSource',
        ),
        num_views=[2],
        pipelines=[train_pipeline],
        prefetch=prefetch,
    ))
```

CHAPTER

NINE

TUTORIAL 2: CUSTOMIZE DATA PIPELINES

9.1 Overview of `Pipeline`

`DataSource` and `Pipeline` are two important components in `Dataset`. We have introduced `DataSource` in *add_new_dataset*. And the `Pipeline` is responsible for applying a series of data augmentations to images, such as random flip.

Here is a config example of `Pipeline` for `SimCLR` training:

```
train_pipeline = [
    dict(type='RandomResizedCrop', size=224),
    dict(type='RandomHorizontalFlip'),
    dict(
        type='RandomAppliedTrans',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.8,
                contrast=0.8,
                saturation=0.8,
                hue=0.2)
        ],
        p=0.8),
    dict(type='RandomGrayscale', p=0.2),
    dict(type='GaussianBlur', sigma_min=0.1, sigma_max=2.0, p=0.5)
]
```

Every augmentation in the `Pipeline` receives an image as input and outputs an augmented image.

9.2 Creating new augmentations in `Pipeline`

1.Write a new transformation function in transforms.py and overwrite the `__call__` function, which takes a `Pillow` image as input:

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

    def __call__(self, img):
        # apply transforms on img
        return img
```

2.Use it in config files. We reuse the config file shown above and add `MyTransform` to it.

```
train_pipeline = [
    dict(type='RandomResizedCrop', size=224),
    dict(type='RandomHorizontalFlip'),
    dict(type='MyTransform'),
    dict(
        type='RandomAppliedTrans',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.8,
                contrast=0.8,
                saturation=0.8,
                hue=0.2)
        ],
        p=0.8),
    dict(type='RandomGrayscale', p=0.2),
    dict(type='GaussianBlur', sigma_min=0.1, sigma_max=2.0, p=0.5)
]
```

CHAPTER

TEN

TUTORIAL 3: ADDING NEW MODULES

In self-supervised learning domain, each model can be divided into following four parts:

- backbone: used to extract image's feature
- projection head: projects feature extracted by backbone to another space
- loss: loss function the model will optimize
- memory bank(optional): some methods, `e.g. odc`, need extract memory bank to store image's feature.

10.1 Add new backbone

Assuming we are going to create a customized backbone `CustomizedBackbone`

1.Create a new file `mmselfsup/models/backbones/customized_backbone.py` and implement `CustomizedBackbone` in it.

```
import torch.nn as nn
from ..builder import BACKBONES

@BACKBONES.register_module()
class CustomizedBackbone(nn.Module):

    def __init__(self, **kwargs):

        ## TODO

    def forward(self, x):

        ## TODO

    def init_weights(self, pretrained=None):

        ## TODO
```

(continues on next page)

(continued from previous page)

```
    def train(self, mode=True):

        ## TODO
```

2.Import the customized backbone in `mmselfsup/models/backbones/__init__.py`.

```
from .customized_backbone import CustomizedBackbone

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

3.Use it in your config file.

```
model = dict(
    ...
    backbone=dict(
        type='CustomizedBackbone',
        ...),
    ...
)
```

10.2 Add new necks

we include all projection heads in `mmselfsup/models/necks`. Assuming we are going to create a `CustomizedProjHead`.

1.Create a new file `mmselfsup/models/necks/customized_proj_head.py` and implement `CustomizedProjHead` in it.

```
import torch.nn as nn
from mmcv.runner import BaseModule

from ..builder import NECKS


@NECKS.register_module()
class CustomizedProjHead(BaseModule):

    def __init__(self, *args, **kwargs):
        super(CustomizedProjHead, self).__init__(init_cfg)
        ## TODO
    def forward(self, x):
        ## TODO
```

You need to implement the forward function, which takes the feature from the backbone and outputs the projected feature.

2.Import the `CustomizedProjHead` in `mmselfsup/models/necks/__init__`.

```
from .customized_proj_head import CustomizedProjHead

__all__ = [
    ...,
    CustomizedProjHead,
    ...
]
```

3.Use it in your config file.

```
model = dict(
    ...,
    neck=dict(
        type='CustomizedProjHead',
        ...),
   ...)
```

10.3 Add new loss

To add a new loss function, we mainly implement the `forward` function in the loss module.

1.Create a new file `mmselfsup/models/heads/customized_head.py` and implement your customized `CustomizedHead` in it.

```
import torch
import torch.nn as nn
from mmcv.runner import BaseModule

from ..builder import HEADS


@HEADS.register_module()
class CustomizedHead(BaseModule):

    def __init__(self, *args, **kwargs):
        super(CustomizedHead, self).__init__()

        ## TODO

    def forward(self, *args, **kwargs):

        ## TODO
```

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

```
from .customized_head import CustomizedHead

__all__ = [..., CustomizedHead, ...]
```

3.Use it in your config file.

```
model = dict(
    ...,
    head=dict(type='CustomizedHead')
    )
```

10.4 Combine all

After creating each component, mentioned above, we need to create a `CustomizedAlgorithm` to organize them logically. And the `CustomizedAlgorithm` takes raw images as inputs and outputs the loss to the optimizer.

1.Create a new file `mmselfsup/models/algorithms/customized_algorithm.py` and implement `CustomizedAlgorithm` in it.

```
# Copyright (c) OpenMMLab. All rights reserved.
import torch

from ..builder import ALGORITHMS, build_backbone, build_head, build_neck
from ..utils import GatherLayer
from .base import BaseModel


@ALGORITHMS.register_module()
class CustomizedAlgorithm(BaseModel):

    def __init__(self, backbone, neck=None, head=None, init_cfg=None):
        super(SimCLR, self).__init__(init_cfg)

        ## TODO

    def forward_train(self, img, **kwargs):

        ## TODO
```

2.Import the module in `mmselfsup/models/algorithms/__init__.py`

```
from .customized_algorithm import CustomizedAlgorithm

__all__ = [..., CustomizedAlgorithm, ...]
```

3.Use it in your config file.

```
model = dict(
    type='CustomizedAlgorightm',
    backbone=...,
    neck=...,
    head=...)
```

CHAPTER

ELEVEN

TUTORIAL 4: CUSTOMIZE SCHEDULE

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

11.1 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)
```

In addition to optimizers implemented by PyTorch, we also implement a customized LARS in `mmselfsup/core/optimizer/optimizers.py`

11.2 Customize learning rate schedules

11.2.1 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 CosineAnnealing policy to train SimCLR, and the config is:

```
lr_config = dict(
    policy='CosineAnnealing',
    ...)
```

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

We also support many other learning rate schedules here, such as Poly schedule.

11.2.2 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 small value to the expected value.

In MMSelfSup, 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` : whether 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,
    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)
```

11.2.3 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 CyclicLrUpdaterHook and CyclicMomentumUpdaterHook.

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,
)
```

11.2.4 Parameter-wise configuration

Some models may have some parameter-specific settings for optimization, for example, no weight decay to the BatchNorm layer and the bias in each layer. To finely configure them, we can use the `paramwise_options` in optimizer.

For example, if we do not want to apply weight decay to the parameters of BatchNorm or GroupNorm, and the bias in each layer, we can use following config file:

```
optimizer = dict(
    type=...,
    lr=...,
    paramwise_options={
        '(bn|gn)(\\d+)?.(weight|bias)':
        dict(weight_decay=0.),
        'bias': dict(weight_decay=0.)
    })
```

11.3 Gradient clipping and gradient accumulation

11.3.1 Gradient clipping

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

Currently we support `grad_clip` option in `optimizer_config`, and you can refer to PyTorch Documentation for more arguments .

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.

11.3.2 Gradient accumulation

When there is not enough computation resource, the batch size can only be set to a small value, which may degrade the performance of model. Gradient accumulation can be used to solve this problem.

Here is an example:

```
data = dict(samples_per_gpu=64)
optimizer_config = dict(type="DistOptimizerHook", update_interval=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")
```

11.4 Customize self-implemented optimizer

In academic research and industrial practice, it is likely that you need some optimization methods not implemented by MMSelfSup, and you can add them through the following methods.

Implement your `CustomizedOptim` in `mmselfsup/core/optimizer/optimizers.py`

```
import torch
from torch.optim import *  # noqa: F401,F403
from torch.optim.optimizer import Optimizer, required

from mmcv.runner.optimizer.builder import OPTIMIZERS

@OPTIMIZER.register_module()
class CustomizedOptim(Optimizer):

    def __init__(self, *args, **kwargs):

        ## TODO

    @torch.no_grad()
    def step(self):

        ## TODO
```

Import it in `mmselfsup/core/optimizer/__init__.py`

```
from .optimizers import CustomizedOptim
from .builder import build_optimizer

__all__ = ['CustomizedOptim', 'build_optimizer', ...]
```

Use it in your config file

```
optimizer = dict(
    type='CustomizedOptim',
    ...
)
```

CHAPTER

TWELVE

TUTORIAL 5: 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.

12.1 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 we will run training and valiation for one epoch iteratively.

By default, we recommend using `EvalHook` to do evaluation after the training epoch.

12.2 Hooks

The hook mechanism is widely used in the OpenMMLab open-source algorithm library. Inserted in 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

Those hooks are registered by the runner by default. Generally, they fulfill some basic functions, and have default 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.

12.2.1 default training hooks

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

`OptimizerHook`, `MomentumUpdaterHook` and `LrUpdaterHook` have been introduced in *schedule 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-scale 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, during the first 200 epochs, evaluation will not be executed. From the 200th epoch, evaluation will be executed after the training process.

12.3 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')
]
```

12.4 Customize self-implemented hooks

12.4.1 1. Implement a new hook

Here we give an example of creating a new hook in MMSelfSup.

```
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):
        pass

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

Depending on your intention of this hook, you need to implement different functionalities in `before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`.

12.4.2 2. Import the new hook

Then we need to ensure `MyHook` imported. Assuming `MyHook` is in `mmselfsup/core/hooks/my_hook.py`, there are two ways to import it:

- Modify `mmselfsup/core/hooks/__init__.py` as below

```
from .my_hook import MyHook

__all__ = [..., MyHook, ...]
```

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

```
custom_imports = dict(imports=['mmselfsup.core.hooks.my_hook'], allow_failed_
↪imports=False)
```

12.4.3 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.

CHAPTER

THIRTEEN

TUTORIAL 6: RUN BENCHMARKS

In MMSelfSup, we provide many benchmarks, thus the models can be evaluated on different downstream tasks. Here are comprehensive tutorials and examples to explain how to run all benchmarks with MMSelfSup.

First, you are supposed to extract your backbone weights by `tools/model_converters/extract_backbone_weights.py`

```
python ./tools/model_converters/extract_backbone_weights.py {CHECKPOINT} {MODEL_FILE}
```

Arguments:

- `CHECKPOINT`: the checkpoint file of a selfsup method named as epoch_*.pth.
- `MODEL_FILE`: the output backbone weights file. If not mentioned, the `PRETRAIN` below uses this extracted model file.

13.1 Classification

As for classification, we provide scripts in folder `tools/benchmarks/classification/`, which has 4 `.sh` files, 1 folder for VOC SVM related classification task and 1 folder for ImageNet nearest-neighbor classification task.

13.1.1 VOC SVM / Low-shot SVM

To run these benchmarks, you should first prepare your VOC datasets. Please refer to prepare_data.md for the details of data preparation.

To evaluate the pre-trained models, you can run command below.

```
# distributed version
bash tools/benchmarks/classification/svm_voc07/dist_test_svm_pretrain.sh ${SELFSUP_
↪CONFIG} ${GPUS} ${PRETRAIN} ${FEATURE_LIST}

# slurm version
bash tools/benchmarks/classification/svm_voc07/slurm_test_svm_pretrain.sh ${PARTITION} $
↪{JOB_NAME} ${SELFSUP_CONFIG} ${PRETRAIN} ${FEATURE_LIST}
```

Besides, if you want to evaluate the ckpt files saved by runner, you can run command below.

```
# distributed version
bash tools/benchmarks/classification/svm_voc07/dist_test_svm_epoch.sh ${SELFSUP_CONFIG} $
↪{EPOCH} ${FEATURE_LIST}

# slurm version
bash tools/benchmarks/classification/svm_voc07/slurm_test_svm_epoch.sh ${PARTITION} $
↪{JOB_NAME} ${SELFSUP_CONFIG} ${EPOCH} ${FEATURE_LIST}
```

To test with ckpt, the code uses the epoch_*.pth file, there is no need to extract weights.

Remarks:

- `${SELFSUP_CONFIG}` is the config file of the self-supervised experiment.
- `${FEATURE_LIST}` is a string to specify features from layer1 to layer5 to evaluate; e.g., if you want to evaluate layer5 only, then `FEATURE_LIST` is "feat5", if you want to evaluate all features, then `FEATURE_LIST` is "feat1 feat2 feat3 feat4 feat5" (separated by space). If left empty, the default `FEATURE_LIST` is "feat5".
- `PRETRAIN`: the pre-trained model file.
- if you want to change GPU numbers, you could add `GPUS_PER_NODE=4 GPUS=4` at the beginning of the command.
- `EPOCH` is the epoch number of the ckpt that you want to test

13.1.2 Linear Evaluation

The linear evaluation is one of the most general benchmarks, we integrate several papers' config settings, also including multi-head linear evaluation. We write classification model in our own codebase for the multi-head function, thus, to run linear evaluation, we still use `.sh` script to launch training. The supported datasets are **ImageNet**, **Places205** and **iNaturalist18**.

```
# distributed version
bash tools/benchmarks/classification/dist_train_linear.sh ${CONFIG} ${PRETRAIN}

# slurm version
bash tools/benchmarks/classification/slurm_train_linear.sh ${PARTITION} ${JOB_NAME} $
↪{CONFIG} ${PRETRAIN}
```

Remarks:

- The default GPU number is 8. When changing GPUS, please also change `samples_per_gpu` in the config file accordingly to ensure the total batch size is 256.
- `CONFIG`: Use config files under `configs/benchmarks/classification/`. Specifically, `imagenet` (excluding `imagenet_*percent` folders), `places205` and `inaturalist2018`.
- `PRETRAIN`: the pre-trained model file.

13.1.3 ImageNet Semi-Supervised Classification

To run ImageNet semi-supervised classification, we still use `.sh` script to launch training.

```
# distributed version
bash tools/benchmarks/classification/dist_train_semi.sh ${CONFIG} ${PRETRAIN}

# slurm version
bash tools/benchmarks/classification/slurm_train_semi.sh ${PARTITION} ${JOB_NAME} $
↪{CONFIG} ${PRETRAIN}
```

Remarks:

- The default GPU number is 4.
- `CONFIG`: Use config files under `configs/benchmarks/classification/imagenet/`, named `imagenet_*percent` folders.
- `PRETRAIN`: the pre-trained model file.

13.1.4 ImageNet Nearest-Neighbor Classification

To evaluate the pre-trained models using the nearest-neighbor benchmark, you can run command below.

```
# distributed version
bash tools/benchmarks/classification/knn_imagenet/dist_test_knn_pretrain.sh ${SELFSUP_
↪CONFIG} ${PRETRAIN}

# slurm version
bash tools/benchmarks/classification/knn_imagenet/slurm_test_knn_pretrain.sh ${PARTITION}
↪ ${JOB_NAME} ${SELFSUP_CONFIG} ${PRETRAIN}
```

Besides, if you want to evaluate the ckpt files saved by runner, you can run command below.

```
# distributed version
bash tools/benchmarks/classification/knn_imagenet/dist_test_knn_epoch.sh ${SELFSUP_
↪CONFIG} ${EPOCH}

# slurm version
bash tools/benchmarks/classification/knn_imagenet/slurm_test_knn_epoch.sh ${PARTITION} $
↪{JOB_NAME} ${SELFSUP_CONFIG} ${EPOCH}
```

To test with ckpt, the code uses the epoch_*.pth file, there is no need to extract weights.

Remarks:

- `${SELFSUP_CONFIG}` is the config file of the self-supervised experiment.
- `PRETRAIN`: the pre-trained model file.

- if you want to change GPU numbers, you could add `GPUS_PER_NODE=4 GPUS=4` at the beginning of the command.
- `EPOCH` is the epoch number of the ckpt that you want to test

13.2 Detection

Here, we prefer to use MMDetection to do the detection task. First, make sure you have installed MIM, which is also a project of OpenMMLab.

```
pip install openmim
```

It is very easy to install the package.

Besides, please refer to MMDet for installation and data preparation

After installation, you can run MMDet with simple command.

```
# distributed version
bash tools/benchmarks/mmdetection/mim_dist_train.sh ${CONFIG} ${PRETRAIN} ${GPUS}

# slurm version
bash tools/benchmarks/mmdetection/mim_slurm_train.sh ${PARTITION} ${CONFIG} ${PRETRAIN}
```

Remarks:

- `CONFIG`: Use config files under `configs/benchmarks/mmdetection/` or write your own config files
- `PRETRAIN`: the pre-trained model file.

Or if you want to do detection task with detectron2, we also provides some config files. Please refer to INSTALL.md for installation and follow the directory structure to prepare your datasets required by detectron2.

```
conda activate detectron2 # use detectron2 environment here, otherwise use open-mmlab␣
↪environment
cd benchmarks/detection
python convert-pretrain-to-detectron2.py ${WEIGHT_FILE} ${OUTPUT_FILE} # must use .pkl␣
↪as the output extension.
bash run.sh ${DET_CFG} ${OUTPUT_FILE}
```

13.3 Segmentation

For semantic segmentation task, we use MMSegmentation. First, make sure you have installed MIM, which is also a project of OpenMMLab.

```
pip install openmim
```

It is very easy to install the package.

Besides, please refer to MMSeg for installation and data preparation.

After installation, you can run MMSeg with simple command.

```
# distributed version
bash tools/benchmarks/mmsegmentation/mim_dist_train.sh ${CONFIG} ${PRETRAIN} ${GPUS}

# slurm version
bash tools/benchmarks/mmsegmentation/mim_slurm_train.sh ${PARTITION} ${CONFIG} $
↪{PRETRAIN}
```

Remarks:

- `CONFIG`: Use config files under `configs/benchmarks/mmsegmentation/` or write your own config files
- `PRETRAIN`: the pre-trained model file.

CHAPTER

FOURTEEN

BYOL

Bootstrap your own latent: A new approach to self-supervised Learning

14.1 Abstract

Bootstrap **Y**our **O**wn **L**atent (BYOL) is a new approach to self-supervised image representation learning. BYOL relies on two neural networks, referred to as online and target networks, that interact and learn from each other. From an augmented view of an image, we train the online network to predict the target network representation of the same image under a different augmented view. At the same time, we update the target network with a slow-moving average of the online network.

14.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

14.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb512-coslr-90e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

14.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

14.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

14.3 Citation

```
@inproceedings{grill2020bootstrap,
  title={Bootstrap your own latent: A new approach to self-supervised learning},
  author={Grill, Jean-Bastien and Strub, Florian and Altch{\'e}, Florent and Tallec,␣
↪Corentin and Richemond, Pierre H and Buchatskaya, Elena and Doersch, Carl and Pires,␣
↪Bernardo Avila and Guo, Zhaohan Daniel and Azar, Mohammad Gheshlaghi and others},
  booktitle={NeurIPS},
  year={2020}
}
```

CHAPTER

FIFTEEN

DEEPCLUSTER

Deep Clustering for Unsupervised Learning of Visual Features

15.1 Abstract

Clustering is a class of unsupervised learning methods that has been extensively applied and studied in computer vision. Little work has been done to adapt it to the end-to-end training of visual features on large scale datasets. In this work, we present DeepCluster, a clustering method that jointly learns the parameters of a neural network and the cluster assignments of the resulting features. DeepCluster iteratively groups the features with a standard clustering algorithm, k-means, and uses the subsequent assignments as supervision to update the weights of the network.

15.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

15.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

15.3 Citation

```
@inproceedings{caron2018deep,
  title={Deep clustering for unsupervised learning of visual features},
  author={Caron, Mathilde and Bojanowski, Piotr and Joulin, Armand and Douze, Matthijs},
  booktitle={ECCV},
  year={2018}
}
```

CHAPTER

SIXTEEN

DENSECL

Dense Contrastive Learning for Self-Supervised Visual Pre-Training

16.1 Abstract

To date, most existing self-supervised learning methods are designed and optimized for image classification. These pre-trained models can be sub-optimal for dense prediction tasks due to the discrepancy between image-level prediction and pixel-level prediction. To fill this gap, we aim to design an effective, dense self-supervised learning method that directly works at the level of pixels (or local features) by taking into account the correspondence between local features. We present dense contrastive learning (DenseCL), which implements self-supervised learning by optimizing a pairwise contrastive (dis)similarity loss at the pixel level between two views of input images.

16.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

16.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

16.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

16.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

16.3 Citation

```
@inproceedings{wang2021dense,
  title={Dense contrastive learning for self-supervised visual pre-training},
  author={Wang, Xinlong and Zhang, Rufeng and Shen, Chunhua and Kong, Tao and Li, Lei},
  booktitle={CVPR},
  year={2021}
}
```

CHAPTER

SEVENTEEN

MOCO V2

Improved Baselines with Momentum Contrastive Learning

17.1 Abstract

Contrastive unsupervised learning has recently shown encouraging progress, e.g., in Momentum Contrast (MoCo) and SimCLR. In this note, we verify the effectiveness of two of SimCLR's design improvements by implementing them in the MoCo framework. With simple modifications to MoCo—namely, using an MLP projection head and more data augmentation—we establish stronger baselines that outperform SimCLR and do not require large training batches. We hope this will make state-of-the-art unsupervised learning research more accessible.

17.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

17.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

17.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

17.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

17.3 Citation

```
@article{chen2020improved,
  title={Improved baselines with momentum contrastive learning},
  author={Chen, Xinlei and Fan, Haoqi and Girshick, Ross and He, Kaiming},
  journal={arXiv preprint arXiv:2003.04297},
  year={2020}
}
```

CHAPTER

EIGHTEEN

NPID

Unsupervised Feature Learning via Non-Parametric Instance Discrimination

18.1 Abstract

Neural net classifiers trained on data with annotated class labels can also capture apparent visual similarity among categories without being directed to do so. We study whether this observation can be extended beyond the conventional domain of supervised learning: Can we learn a good feature representation that captures apparent similar- ity among instances, instead of classes, by merely asking the feature to be discriminative of individual instances?

We formulate this intuition as a non-parametric classification problem at the instance-level, and use noise-contrastive estimation to tackle the computational challenges imposed by the large number of instance classes. Our experimental results demonstrate that, under unsupervised learning settings, our method surpasses the state-of-the-art on ImageNet classification by a large margin.

Our method is also remarkable for consistently improving test performance with more training data and better network architectures. By fine-tuning the learned feature, we further obtain competitive results for semi-supervised learning and object detection tasks. Our non-parametric model is highly compact: With 128 features per image, our method requires only 600MB storage for a million images, enabling fast nearest neighbour retrieval at the run time.

18.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

18.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

18.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

18.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

18.3 Citation

```
@inproceedings{wu2018unsupervised,
  title={Unsupervised feature learning via non-parametric instance discrimination},
  author={Wu, Zhirong and Xiong, Yuanjun and Yu, Stella X and Lin, Dahua},
  booktitle={CVPR},
  year={2018}
}
```

CHAPTER

NINETEEN

ODC

Online Deep Clustering for Unsupervised Representation Learning

19.1 Abstract

Joint clustering and feature learning methods have shown remarkable performance in unsupervised representation learning. However, the training schedule alternating between feature clustering and network parameters update leads to unstable learning of visual representations. To overcome this challenge, we propose Online Deep Clustering (ODC) that performs clustering and network update simultaneously rather than alternatingly. Our key insight is that the cluster centroids should evolve steadily in keeping the classifier stably updated. Specifically, we design and maintain two dynamic memory modules, i.e., samples memory to store samples' labels and features, and centroids memory for centroids evolution. We break down the abrupt global clustering into steady memory update and batch-wise label re-assignment. The process is integrated into network update iterations. In this way, labels and the network evolve shoulder-to-shoulder rather than alternatingly. Extensive experiments demonstrate that ODC stabilizes the training process and boosts the performance effectively.

19.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

19.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

19.3 Citation

```
@inproceedings{zhan2020online,
  title={Online deep clustering for unsupervised representation learning},
  author={Zhan, Xiaohang and Xie, Jiahao and Liu, Ziwei and Ong, Yew-Soon and Loy, Chen␣
↪Change},
  booktitle={CVPR},
  year={2020}
}
```

CHAPTER

TWENTY

RELATIVE LOCATION

Unsupervised Visual Representation Learning by Context Prediction

20.1 Abstract

This work explores the use of spatial context as a source of free and plentiful supervisory signal for training a rich visual representation. Given only a large, unlabeled image collection, we extract random pairs of patches from each image and train a convolutional neural net to predict the position of the second patch relative to the first. We argue that doing well on this task requires the model to learn to recognize objects and their parts. We demonstrate that the feature representation learned using this within-image context indeed captures visual similarity across images. For example, this representation allows us to perform unsupervised visual discovery of objects like cats, people, and even birds from the Pascal VOC 2011 detection dataset. Furthermore, we show that the learned ConvNet can be used in the RCNN framework and provides a significant boost over a randomly-initialized ConvNet, resulting in state-of-the-art performance among algorithms which use only Pascal-provided training set annotations.

20.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

20.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

20.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

20.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

20.3 Citation

```
@inproceedings{doersch2015unsupervised,
  title={Unsupervised visual representation learning by context prediction},
  author={Doersch, Carl and Gupta, Abhinav and Efros, Alexei A},
  booktitle={ICCV},
  year={2015}
}
```

CHAPTER

TWENTYONE

ROTATION PREDICTION

Unsupervised Representation Learning by Predicting Image Rotation

21.1 Abstract

Over the last years, deep convolutional neural networks (ConvNets) have transformed the field of computer vision thanks to their unparalleled capacity to learn high level semantic image features. However, in order to successfully learn those features, they usually require massive amounts of manually labeled data, which is both expensive and impractical to scale. Therefore, unsupervised semantic feature learning, i.e., learning without requiring manual annotation effort, is of crucial importance in order to successfully harvest the vast amount of visual data that are available today. In our work we propose to learn image features by training ConvNets to recognize the 2d rotation that is applied to the image that it gets as input. We demonstrate both qualitatively and quantitatively that this apparently simple task actually provides a very powerful supervisory signal for semantic feature learning. We exhaustively evaluate our method in various unsupervised feature learning benchmarks and we exhibit in all of them state-of-the-art performance. Specifically, our results on those benchmarks demonstrate dramatic improvements w.r.t. prior state-of-the-art approaches in unsupervised representation learning and thus significantly close the gap with supervised feature learning.

21.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

21.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-steplr-100e_in1k.py for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

21.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

21.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

21.3 Citation

```
@inproceedings{komodakis2018unsupervised,
  title={Unsupervised representation learning by predicting image rotations},
  author={Komodakis, Nikos and Gidaris, Spyros},
  booktitle={ICLR},
  year={2018}
}
```

CHAPTER

TWENTYTWO

SIMCLR

A Simple Framework for Contrastive Learning of Visual Representations

22.1 Abstract

This paper presents SimCLR: a simple framework for contrastive learning of visual representations. We simplify recently proposed contrastive self-supervised learning algorithms without requiring specialized architectures or a memory bank. In order to understand what enables the contrastive prediction tasks to learn useful representations, we systematically study the major components of our framework. We show that (1) composition of data augmentations plays a critical role in defining effective predictive tasks, (2) introducing a learnable nonlinear transformation between the representation and the contrastive loss substantially improves the quality of the learned representations, and (3) contrastive learning benefits from larger batch sizes and more training steps compared to supervised learning. By combining these findings, we are able to considerably outperform previous methods for self-supervised and semi-supervised learning on ImageNet. A linear classifier trained on self-supervised representations learned by SimCLR achieves 76.5% top-1 accuracy, which is a 7% relative improvement over previous state-of-the-art, matching the performance of a supervised ResNet-50.

22.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

22.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb512-coslr-90e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

22.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

22.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

22.3 Citation

```
@inproceedings{chen2020simple,
  title={A simple framework for contrastive learning of visual representations},
  author={Chen, Ting and Kornblith, Simon and Norouzi, Mohammad and Hinton, Geoffrey},
  booktitle={ICML},
  year={2020},
}
```

CHAPTER

TWENTYTHREE

SIMSIAM

Exploring Simple Siamese Representation Learning

23.1 Abstract

Siamese networks have become a common structure in various recent models for unsupervised visual representation learning. These models maximize the similarity between two augmentations of one image, subject to certain conditions for avoiding collapsing solutions. In this paper, we report surprising empirical results that simple Siamese networks can learn meaningful representations even using none of the following: (i) negative sample pairs, (ii) large batches, (iii) momentum encoders. Our experiments show that collapsing solutions do exist for the loss and structure, but a stop-gradient operation plays an essential role in preventing collapsing. We provide a hypothesis on the implication of stop-gradient, and further show proof-of-concept experiments verifying it. Our "SimSiam" method achieves competitive results on ImageNet and downstream tasks. We hope this simple baseline will motivate people to rethink the roles of Siamese architectures for unsupervised representation learning.

23.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

23.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb512-coslr-90e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

23.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

23.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

23.3 Citation

```bibtex
@inproceedings{chen2021exploring,
  title={Exploring simple siamese representation learning},
  author={Chen, Xinlei and He, Kaiming},
  booktitle={CVPR},
  year={2021}
}
```

CHAPTER

TWENTYFOUR

SWAV

Unsupervised Learning of Visual Features by Contrasting Cluster Assignments

24.1 Abstract

Unsupervised image representations have significantly reduced the gap with supervised pretraining, notably with the recent achievements of contrastive learning methods. These contrastive methods typically work online and rely on a large number of explicit pairwise feature comparisons, which is computationally challenging. In this paper, we propose an online algorithm, SwAV, that takes advantage of contrastive methods without requiring to compute pairwise comparisons. Specifically, our method simultaneously clusters the data while enforcing consistency between cluster assignments produced for different augmentations (or "views") of the same image, instead of comparing features directly as in contrastive learning. Simply put, we use a "swapped" prediction mechanism where we predict the code of a view from the representation of another view. Our method can be trained with large and small batches and can scale to unlimited amounts of data. Compared to previous contrastive methods, our method is more memory efficient since it does not require a large memory bank or a special momentum network. In addition, we also propose a new data augmentation strategy, multi-crop, that uses a mix of views with different resolutions in place of two full-resolution views, without increasing the memory or compute requirements.

24.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

24.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

VOC SVM / Low-shot SVM

The **Best Layer** indicates that the best results are obtained from which layers feature map. For example, if the **Best Layer** is **feature3**, its best result is obtained from the second stage of ResNet (1 for stem layer, 2-5 for 4 stage layers).

Besides, k=1 to 96 indicates the hyper-parameter of Low-shot SVM.

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_linear-8xb32-steplr-90e_in1k for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_linear-8xb32-coslr-100e_in1k for details of config.

Places205 Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-28e_places205.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

24.2.2 Detection

The detection benchmarks includes 2 downstream task datasets, **Pascal VOC 2007 + 2012** and **COCO2017**. This benchmark follows the evluation protocols set up by MoCo.

Pascal VOC 2007 + 2012

Please refer to faster_rcnn_r50_c4_mstrain_24k_voc0712.py for details of config.

COCO2017

Please refer to mask_rcnn_r50_fpn_mstrain_1x_coco.py for details of config.

24.2.3 Segmentation

The segmentation benchmarks includes 2 downstream task datasets, **Cityscapes** and **Pascal VOC 2012 + Aug**. It follows the evluation protocols set up by MMSegmentation.

Pascal VOC 2012 + Aug

Please refer to fcn_r50-d8_512x512_20k_voc12aug.py for details of config.

24.3 Citation

```
@article{caron2020unsupervised,
  title={Unsupervised Learning of Visual Features by Contrasting Cluster Assignments},
  author={Caron, Mathilde and Misra, Ishan and Mairal, Julien and Goyal, Priya and
↪Bojanowski, Piotr and Joulin, Armand},
  booktitle={NeurIPS},
  year={2020}
}
```

CHAPTER

TWENTYFIVE

MOCO V3

An Empirical Study of Training Self-Supervised Vision Transformers

25.1 Abstract

This paper does not describe a novel method. Instead, it studies a straightforward, incremental, yet must-know baseline given the recent progress in computer vision: self-supervised learning for Vision Transformers (ViT). While the training recipes for standard convolutional networks have been highly mature and robust, the recipes for ViT are yet to be built, especially in the self-supervised scenarios where training becomes more challenging. In this work, we go back to basics and investigate the effects of several fundamental components for training self-supervised ViT. We observe that instability is a major issue that degrades accuracy, and it can be hidden by apparently good results. We reveal that these results are indeed partial failure, and they can be improved when training is made more stable. We benchmark ViT results in MoCo v3 and several other self-supervised frameworks, with ablations in various aspects. We discuss the currently positive evidence as well as challenges and open questions. We hope that this work will provide useful data points and experience for future research.

25.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

25.2.1 Classification

The classification benchmarks includes 4 downstream task datasets, **VOC**, **ImageNet**, **iNaturalist2018** and **Places205**. If not specified, the results are Top-1 (%).

ImageNet Linear Evaluation

The **Linear Evaluation** result is obtained by training a linear head upon the pre-trained backbone. Please refer to vit-small-p16_8xb128-coslr-90e_in1k for details of config.

25.3 Citation

```
@InProceedings{Chen_2021_ICCV,
    title     = {An Empirical Study of Training Self-Supervised Vision Transformers},
    author    = {Chen, Xinlei and Xie, Saining and He, Kaiming},
    booktitle = {Proceedings of the IEEE/CVF International Conference on Computer Vision␣
↪(ICCV)},
    year      = {2021}
}
```

CHAPTER

TWENTYSIX

MAE

Masked Autoencoders Are Scalable Vision Learners

26.1 Abstract

This paper shows that masked autoencoders (MAE) are scalable self-supervised learners for computer vision. Our MAE approach is simple: we mask random patches of the input image and reconstruct the missing pixels. It is based on two core designs. First, we develop an asymmetric encoder-decoder architecture, with an encoder that operates only on the visible subset of patches (without mask tokens), along with a lightweight decoder that reconstructs the original image from the latent representation and mask tokens. Second, we find that masking a high proportion of the input image, e.g., 75%, yields a nontrivial and meaningful self-supervisory task. Coupling these two designs enables us to train large models efficiently and effectively: we accelerate training (by 3× or more) and improve accuracy. Our scalable approach allows for learning high-capacity models that generalize well: e.g., a vanilla ViT-Huge model achieves the best accuracy (87.8%) among methods that use only ImageNet-1K data. Transfer performance in downstream tasks outperforms supervised pretraining and shows promising scaling behavior.

26.2 Models and Benchmarks

Here, we report the results of the model, which is pre-trained on ImageNet1K for 400 epochs, the details are below:

26.3 Citation

```
@article{He2021MaskedAA,
  title={Masked Autoencoders Are Scalable Vision Learners},
  author={Kaiming He and Xinlei Chen and Saining Xie and Yanghao Li and
  Piotr Doll'ar and Ross B. Girshick},
  journal={ArXiv},
  year={2021}
}
```

CHAPTER

TWENTYSEVEN

SIMMIM

SimMIM: A Simple Framework for Masked Image Modeling

27.1 Abstract

This paper presents SimMIM, a simple framework for masked image modeling. We simplify recently proposed related approaches without special designs such as blockwise masking and tokenization via discrete VAE or clustering. To study what let the masked image modeling task learn good representations, we systematically study the major components in our framework, and find that simple designs of each component have revealed very strong representation learning performance: 1) random masking of the input image with a moderately large masked patch size (e.g., 32) makes a strong pre-text task; 2) predicting raw pixels of RGB values by direct regression performs no worse than the patch classification approaches with complex designs; 3) the prediction head can be as light as a linear layer, with no worse performance than heavier ones. Using ViT-B, our approach achieves 83.8% top-1 fine-tuning accuracy on ImageNet-1K by pre-training also on this dataset, surpassing previous best approach by +0.6%. When applied on a larger model of about 650 million parameters, SwinV2H, it achieves 87.1% top-1 accuracy on ImageNet-1K using only ImageNet-1K data. We also leverage this approach to facilitate the training of a 3B model (SwinV2-G), that by 40× less data than that in previous practice, we achieve the state-of-the-art on four representative vision benchmarks. The code and models will be publicly available at https: //github.com/microsoft/SimMIM .

27.2 Models and Benchmarks

Here, we report the results of the model, and more results will be coming soon.

27.3 Citation

```
@inproceedings{xie2021simmim,
  title={SimMIM: A Simple Framework for Masked Image Modeling},
  author={Xie, Zhenda and Zhang, Zheng and Cao, Yue and Lin, Yutong and Bao, Jianmin and␣
↪Yao, Zhuliang and Dai, Qi and Hu, Han},
  booktitle={International Conference on Computer Vision and Pattern Recognition (CVPR)},
  year={2022}
}
```

CHAPTER

TWENTYEIGHT

BARLOWTWINS

Barlow Twins: Self-Supervised Learning via Redundancy Reduction

28.1 Abstract

Self-supervised learning (SSL) is rapidly closing the gap with supervised methods on large computer vision benchmarks. A successful approach to SSL is to learn embeddings which are invariant to distortions of the input sample. However, a recurring issue with this approach is the existence of trivial constant solutions. Most current methods avoid such solutions by careful implementation details. We propose an objective function that naturally avoids collapse by measuring the cross-correlation matrix between the outputs of two identical networks fed with distorted versions of a sample, and making it as close to the identity matrix as possible. This causes the embedding vectors of distorted versions of a sample to be similar, while minimizing the redundancy between the components of these vectors. The method is called Barlow Twins, owing to neuroscientist H. Barlow's redundancy-reduction principle applied to a pair of identical networks. Barlow Twins does not require large batches nor asymmetry between the network twins such as a predictor network, gradient stopping, or a moving average on the weight updates. Intriguingly it benefits from very high-dimensional output vectors. Barlow Twins outperforms previous methods on ImageNet for semi-supervised classification in the low-data regime, and is on par with current state of the art for ImageNet classification with a linear classifier head, and for transfer tasks of classification and object detection.

28.2 Results and Models

Back to model_zoo.md to download models.

In this page, we provide benchmarks as much as possible to evaluate our pre-trained models. If not mentioned, all models are pre-trained on ImageNet-1k dataset.

28.2.1 Classification

The classification benchmarks includes 1 downstream task datasets, **ImageNet**. If not specified, the results are Top-1 (%).

ImageNet Linear Evaluation

The **Feature1 - Feature5** don't have the GlobalAveragePooling, the feature map is pooled to the specific dimensions and then follows a Linear layer to do the classification. Please refer to resnet50_mhead_8xb32-steplr-90e.py for details of config.

The **AvgPool** result is obtained from Linear Evaluation with GlobalAveragePooling. Please refer to resnet50_8xb32-steplr-100e_in1k.py for details of config.

ImageNet Nearest-Neighbor Classification

The results are obtained from the features after GlobalAveragePooling. Here, k=10 to 200 indicates different number of nearest neighbors.

28.3 Citation

```
@inproceedings{zbontar2021barlow,
  title={Barlow twins: Self-supervised learning via redundancy reduction},
  author={Zbontar, Jure and Jing, Li and Misra, Ishan and LeCun, Yann and Deny, St{\'e}
↪phane},
  booktitle={International Conference on Machine Learning},
  year={2021},
}
```

CHAPTER

TWENTYNINE

CAE

Context Autoencoder for Self-Supervised Representation Learning

29.1 Abstract

We present a novel masked image modeling (MIM) approach, context autoencoder (CAE), for self-supervised learning. We randomly partition the image into two sets: visible patches and masked patches. The CAE architecture consists of: (i) an encoder that takes visible patches as input and outputs their latent representations, (ii) a latent context regressor that predicts the masked patch representations from the visible patch representations that are not updated in this regressor, (iii) a decoder that takes the estimated masked patch representations as input and makes predictions for the masked patches, and (iv) an alignment module that aligns the masked patch representation estimation with the masked patch representations computed from the encoder. In comparison to previous MIM methods that couple the encoding and decoding roles, e.g., using a single module in BEiT, our approach attempts to separate the encoding role (content understanding) from the decoding role (making predictions for masked patches) using different modules, improving the content understanding capability. In addition, our approach makes predictions from the visible patches to the masked patches in the latent representation space that is expected to take on semantics. In addition, we present the explanations about why contrastive pretraining and supervised pretraining perform similarly and why MIM potentially performs better. We demonstrate the effectiveness of our CAE through superior transfer performance in downstream tasks: semantic segmentation, and object detection and instance segmentation.

29.2 Prerequisite

Create a new folder `cae_ckpt` under the root directory and download the weights for `dalle` encoder to that folder

29.3 Models and Benchmarks

Here, we report the results of the model, which is pre-trained on ImageNet-1k for 300 epochs, the details are below:

29.4 Citation

```
@article{CAE,
  title={Context Autoencoder for Self-Supervised Representation Learning},
  author={Xiaokang Chen, Mingyu Ding, Xiaodi Wang, Ying Xin, Shentong Mo,
  Yunhao Wang, Shumin Han, Ping Luo, Gang Zeng, Jingdong Wang},
  journal={ArXiv},
  year={2022}
}
```

CHAPTER

THIRTY

CHANGELOG

30.1 MMSelfSup

30.1.1 v0.11.0 (30/12/2022)

New Features

- Support InterCLR (#609)

Bug Fixes

- Fix potential bug of hook registration (#647)
- Fix sampling_replace config kwargs bug (#646)
- Change sklearn to scikit-learn in requirements (#583)

Improvements

- Update CI check rules (#581)
- Update assignee schedule (#606)

Docs

- Add global notes and the version switcher menu (#573)

30.1.2 v0.10.1 (01/11/2022)

Improvements

- Update issue template and management file (#550, #512)
- Move res-layer to models.utils (#537)

Docs

- Add MMYOLO description for README (#541)
- Fix typo for README (#545)
- Fix lint of MaskFeat (#520)

30.1.3 v0.10.0 (30/09/2022)

Highlight

- Support MaskFeat (#485)
- Update README to announce 1.0.0rc version (#474)

New Features

- Support MaskFeat (#485)

Bug Fixes

- Fix DenseCL init weights bug (#411)
- Fix norm typo in config (#418)
- Fix read image bug (#386)

Improvements

- Change hook_cfg type access (#409)
- Support to dump training config (#410)
- Support to save MAE visualization results (#388)
- Remove default value of deprecated option (#490)

Docs

- Update the link of MAE (#497)
- Update README to announce 1.0.0rc version (#474)
- Update get_started.md (#402)

30.1.4 v0.9.2 (28/07/2022)

New Features

- Support MAE Reconstructed Image Visualization (#376)

Bug Fixes

- Fix args/cfg bug in extract.py, use cfg.work_dir to save files (#357)
- Fix SimMIM mask generator config bug (#360)

Improvements

- Update mdformat settings (#323)
- Add circleci (#374)

Docs

- Fix the link of switch language (#327)
- Update lr_updater.py links in tutorials/4_schedule.md (#354)

30.1.5 v0.9.1 (31/05/2022)

Highlight

- Update **BYOL** model and results (#319)
- Refine some documentation

New Features

- Update **BYOL** models and results (#319)

Bug Fixes

- Set qkv bias to False for cae and True for mae (#303)
- Fix spelling errors in MAE config (#307)

Improvements

- Change the file name of cosine annealing hook (#304)
- Replace markdownlint with mdformat (#311)

Docs

- Fix typo in tutotial (#308)
- Configure Myst-parser to parse anchor tag (#309)
- Update readthedocs algorithm README (#310)
- Rewrite install.md (#317)
- refine README.md file (#318)

30.1.6 v0.9.0 (29/04/2022)

Highlight

- Support **CAE** (#284)
- Support **Barlow Twins** (#207)

New Features

- Support CAE (#284)
- Support Barlow twins (#207)
- Add SimMIM 192 pretrain and 224 fine-tuning results (#280)
- Add MAE pretrain with fp16 (#271)

Bug Fixes

- Fix args error (#290)
- Change imgs_per_gpu to samples_per_gpu in MAE config (#278)
- Avoid GPU memory leak with prefetch dataloader (#277)
- Fix key error bug when registering custom hooks (#273)

Improvements

- Update SimCLR models and results (#295)
- Reduce memory usage while running unit test (#291)
- Remove pytorch1.5 test (#288)
- Rename linear probing config file names (#281)
- add unit test for apis (#276)

Docs

- Fix SimMIM config link, and add SimMIM to model_zoo (#272)

30.1.7 v0.8.0 (31/03/2022)

Highlight

- Support **SimMIM** (#239)
- Add **KNN** benchmark, support KNN test with checkpoint and extracted backbone weights (#243)
- Support ImageNet-21k dataset (#225)

New Features

- Support SimMIM (#239)
- Add KNN benchmark, support KNN test with checkpoint and extracted backbone weights (#243)
- Support ImageNet-21k dataset (#225)
- Resume latest checkpoint automatically (#245)

Bug Fixes

- Add seed to distributed sampler (#250)
- Fix positional parameter error in dist_test_svm_epoch.sh (#260)
- Fix 'mkdir' error in prepare_voc07_cls.sh (#261)

Improvements

- Update args format from command line (#253)

Docs

- Fix command errors in 6_benchmarks.md (#263)
- Translate 6_benchmarks.md to Chinese (#262)

30.1.8 v0.7.0 (03/03/2022)

Highlight

- Support MAE (#221)
- Add Places205 benchmarks (#210)
- Add test Windows in workflows (#215)

New Features

- Support MAE (#221)
- Add Places205 benchmarks (#210)

Bug Fixes

- Fix config typos for rotation prediction and deepcluster (#200)
- Fix image channel bgr/rgb bug and update benchmarks (#210)
- Fix the bug when using prefetch under multi-view methods (#218)
- Fix tsne 'no init_cfg' error (#222)

Improvements

- Deprecate `imgs_per_gpu` and use `samples_per_gpu` (#204)
- Update the installation of MMCV (#208)
- Add pre-commit hook for algo-readme and copyright (#213)
- Add test Windows in workflows (#215)

Docs

- Translate 0_config.md into Chinese (#216)
- Reorganizing OpenMMLab projects and update algorithms in readme (#219)

30.1.9 v0.6.0 (02/02/2022)

Highlight

- Support vision transformer based MoCo v3 (#194)
- Speed up training and start time (#181)
- Support cpu training (#188)

New Features

- Support vision transformer based MoCo v3 (#194)
- Support cpu training (#188)

Bug Fixes

- Fix issue (#159, #160) related bugs (#161)
- Fix missing prob assignment in `RandomAppliedTrans` (#173)
- Fix bug of showing k-means losses (#182)
- Fix bug in non-distributed multi-gpu training/testing (#189)
- Fix bug when loading cifar dataset (#191)
- Fix `dataset.evaluate` args bug (#192)

Improvements

- Cancel previous runs that are not completed in CI (#145)
- Enhance MIM function (#152)
- Skip CI when some specific files were changed (#154)
- Add `drop_last` when building eval optimizer (#158)
- Deprecate the support for "python setup.py test" (#174)
- Speed up training and start time (#181)
- Upgrade `isort` to 5.10.1 (#184)

Docs

- Refactor the directory structure of docs (#146)
- Fix readthedocs (#148, #149, #153)
- Fix typos and dead links in some docs (#155, #180, #195)
- Update training logs and benchmark results in model zoo (#157, #165, #195)
- Update and translate some docs into Chinese (#163, #164, #165, #166, #167, #168, #169, #172, #176, #178, #179)
- Update algorithm README with the new format (#177)

30.1.10 v0.5.0 (16/12/2021)

Highlight

- Released with code refactor.
- Add 3 new self-supervised learning algorithms.
- Support benchmarks with MMDet and MMSeg.
- Add comprehensive documents.

Refactor

- Merge redundant dataset files.
- Adapt to new version of MMCV and remove old version related codes.
- Inherit MMCV BaseModule.
- Optimize directory.
- Rename all config files.

New Features

- Add SwAV, SimSiam, DenseCL algorithms.
- Add t-SNE visualization tools.
- Support MMCV version fp16.

Benchmarks

- More benchmarking results, including classification, detection and segmentation.
- Support some new datasets in downstream tasks.
- Launch MMDet and MMSeg training with MIM.

Docs

- Refactor README, getting_started, install, model_zoo files.
- Add data_prepare file.
- Add comprehensive tutorials.

30.2 OpenSelfSup (History)

30.2.1 v0.3.0 (14/10/2020)

Highlight

- Support Mixed Precision Training
- Improvement of GaussianBlur doubles the training speed
- More benchmarking results

Bug Fixes

- Fix bugs in moco v2, now the results are reproducible.
- Fix bugs in byol.

New Features

- Mixed Precision Training
- Improvement of GaussianBlur doubles the training speed of MoCo V2, SimCLR, BYOL
- More benchmarking results, including Places, VOC, COCO

30.2.2 v0.2.0 (26/6/2020)

Highlights

- Support BYOL
- Support semi-supervised benchmarks

Bug Fixes

- Fix hash id in publish_model.py

New Features

- Support BYOL.
- Separate train and test scripts in linear/semi evaluation.
- Support semi-supevised benchmarks: benchmarks/dist_train_semi.sh.
- Move benchmarks related configs into configs/benchmarks/.
- Provide benchmarking results and model download links.
- Support updating network every several iterations.
- Support LARS optimizer with nesterov.
- Support excluding specific parameters from LARS adaptation and weight decay required in SimCLR and BYOL.

CHAPTER

THIRTYONE

DIFFERENCES BETWEEN MMSELFSUP AND OPENSELFSUP

This file records differences between the newest version of MMSelfSup with older versions and OpenSelfSup.

MMSelfSup goes through a refactoring and addresses many legacy issues. It is not compatitible with OpenSelfSup, i.e. the old config files are supposed to be updated as some arguments of the class or names of the components have been modified.

The major differences are in two folds: codebase conventions, modular design.

31.1 Modular Design

In order to build more clear directory structure, MMSelfSup redesigns some of the modules.

31.1.1 Datasets

- MMSelfSup merges some datasets to reduce some redundant codes.
  - Classification, Extraction, NPID -> OneViewDataset
  - BYOL, Contrastive -> MultiViewDataset
- The `data_sources` folder has been refactored, thus the loading function is more robust.

In addition, this part is still under refactoring, it will be released in following version.

31.1.2 Models

- The registry mechanism is updated. Currently, the parts under the `models` folder are built with a parent called `MMCV_MODELS` that is imported from `MMCV`. Please check mmselfsup/models/builder.py and refer to mmcv/utils/registry.py for more details.
- The `models` folder includes `algorithms`, `backbones`, `necks`, `heads`, `memories` and some required utils. The `algorithms` integrates the other main components to build the self-supervised learning algorithms, which is like `classifiers` in `MMCls` or `detectors` in `MMDet`.
- In OpenSelfSup, the names of `necks` are kind of confused and all in one file. Now, the `necks` are refactored, managed with one folder and renamed for easier understanding. Please check `mmselfsup/models/necks` for more details.

31.2 Codebase Conventions

MMSelfSup renews codebase conventions as OpenSelfSup has not been updated for some time.

31.2.1 Configs

- MMSelfSup renames all config files to use new name convention. Please refer to *0_config* for more details.
- In the configs, some arguments of the class or names of the components have been modified.
  - One algorithm name has been modified: MOCO -> MoCo
  - As all models' components inherit `BaseModule` from `MMCV`, the models are initialized with `init_cfg`. Please use it to set your initialization. Besides, `init_weights` can also be used.
  - Please use new neck names to compose your algorithms, check it before write your own configs.
  - The normalization layers are all built with arguments `norm_cfg`.

31.2.2 Scripts

- The directory of `tools` is modified, thus it has more clear structure. It has several folders to manage different scripts. For example, it has two converter folders for models and data format. Besides, the benchmark related scripts are all in `benchmarks` folder, which has the same structure as `configs/benchmarks`.
- The arguments in `train.py` has been updated. Two major modifications are listed below.
  - Add `--cfg-options` to modify the config from cmd arguments.
  - Remove `--pretrained` and use `--cfg-options` to set the pretrained models.

CHAPTER

THIRTYTWO

ENGLISH

CHAPTER

THIRTYTHREE

CHAPTER

THIRTYFOUR

MMSELFSUP.APIS

mmselfsup.apis.**inference_model**(*model: torch.nn.modules.module.Module*, *data: <module 'PIL.Image' from '/home/docs/checkouts/readthedocs.org/user_builds/mmselfsup/envs/0.x/lib/python3.7/site-packages/PIL/Image.py'>*) → Tuple[torch.Tensor, Union[torch.Tensor, dict]]

Inference an image with the model. :param model: The loaded model. :type model: nn.Module :param data: The loaded image. :type data: PIL.Image

Returns

Output of model inference. - data (torch.Tensor): The loaded image to input model. - output (torch.Tensor, dict[str, torch.Tensor]): the output

of test model.

Return type Tuple[torch.Tensor, Union(torch.Tensor, dict)]

mmselfsup.apis.**init_model**(*config: Union[str, mmcv.utils.config.Config]*, *checkpoint: Optional[str] = None*, *device: str = 'cuda:0'*, *options: Optional[dict] = None*) → torch.nn.modules.module.Module

Initialize an model from config file.

Parameters

- **config** (str or :obj:mmcv.Config) – Config file path or the config object.
- **checkpoint** (*str, optional*) – Checkpoint path. If left as None, the model will not load any weights. Defaults to None.
- **device** (*str*) – The device where the model will be put on. Default to 'cuda:0'.
- **options** (*dict, optional*) – Options to override some settings in the used config. Defaults to None.

Returns The initialized model.

Return type nn.Module

mmselfsup.apis.**init_random_seed**(*seed=None*, *device='cuda'*)

Initialize random seed.

If the seed is not set, the seed will be automatically randomized, and then broadcast to all processes to prevent some potential bugs. :param seed: The seed. Default to None. :type seed: int, Optional :param device: The device where the seed will be put on.

Default to 'cuda'.

Returns Seed to be used.

Return type int

mmselfsup.apis.**set_random_seed**(*seed*, *deterministic=False*)
Set random seed.

Parameters

- **seed** (*int*) – Seed to be used.
- **deterministic** (*bool*) – Whether to set the deterministic option for CUDNN backend, i.e., set *torch.backends.cudnn.deterministic* to True and *torch.backends.cudnn.benchmark* to False. Defaults to False.

CHAPTER

THIRTYFIVE

MMSELFSUP.CORE

35.1 hooks

class mmselfsup.core.hooks.**DeepClusterHook**(*extractor*, *clustering*, *unif_sampling*, *reweight*, *reweight_pow*, *init_memory=False*, *initial=True*, *interval=1*, *dist_mode=True*, *data_loaders=None*)

Hook for DeepCluster.

This hook includes the global clustering process in DC.

Parameters

- **extractor** (*dict*) – Config dict for feature extraction.
- **clustering** (*dict*) – Config dict that specifies the clustering algorithm.
- **unif_sampling** (*bool*) – Whether to apply uniform sampling.
- **reweight** (*bool*) – Whether to apply loss re-weighting.
- **reweight_pow** (*float*) – The power of re-weighting.
- **init_memory** (*bool*) – Whether to initialize memory banks used in ODC. Defaults to False.
- **initial** (*bool*) – Whether to call the hook initially. Defaults to True.
- **interval** (*int*) – Frequency of epochs to call the hook. Defaults to 1.
- **dist_mode** (*bool*) – Use distributed training or not. Defaults to True.
- **data_loaders** (*DataLoader*) – A PyTorch dataloader. Defaults to None.

class mmselfsup.core.hooks.**DenseCLHook**(*start_iters=1000*, ***kwargs*)

Hook for DenseCL.

This hook includes `loss_lambda` warmup in DenseCL. Borrowed from the authors' code: https://github.com/WXinlong/DenseCL.

Parameters **start_iters** (*int, optional*) – The number of warmup iterations to set `loss_lambda=0`. Defaults to 1000.

class mmselfsup.core.hooks.**DistOptimizerHook**(*update_interval=1*, *grad_clip=None*, *coalesce=True*, *bucket_size_mb=- 1*, *frozen_layers_cfg={}*)

Optimizer hook for distributed training.

This hook can accumulate gradients every n intervals and freeze some layers for some iters at the beginning.

Parameters

- **update_interval** (*int, optional*) – The update interval of the weights, set > 1 to accumulate the grad. Defaults to 1.

- **grad_clip** (*dict, optional*) – Dict to config the value of grad clip. E.g., grad_clip = dict(max_norm=10). Defaults to None.
- **coalesce** (*bool, optional*) – Whether allreduce parameters as a whole. Defaults to True.
- **bucket_size_mb** (*int, optional*) – Size of bucket, the unit is MB. Defaults to -1.
- **frozen_layers_cfg** (*dict, optional*) – Dict to config frozen layers. The key-value pair is layer name and its frozen iters. If frozen, the layer gradient would be set to None. Defaults to dict().

class mmselfsup.core.hooks.**GradAccumFp16OptimizerHook**(*update_interval=1*, *frozen_layers_cfg={}*, ***kwargs*)

Fp16 optimizer hook (using PyTorch's implementation).

This hook can accumulate gradients every n intervals and freeze some layers for some iters at the beginning. If you are using PyTorch >= 1.6, torch.cuda.amp is used as the backend, to take care of the optimization procedure.

Parameters

- **update_interval** (*int, optional*) – The update interval of the weights, set > 1 to accumulate the grad. Defaults to 1.
- **frozen_layers_cfg** (*dict, optional*) – Dict to config frozen layers. The key-value pair is layer name and its frozen iters. If frozen, the layer gradient would be set to None. Defaults to dict().

after_train_iter(*runner*)

Backward optimization steps for Mixed Precision Training. For dynamic loss scaling, please refer to https://pytorch.org/docs/stable/amp.html#torch.cuda.amp.GradScaler.

1. Scale the loss by a scale factor.
2. Backward the loss to obtain the gradients.
3. Unscale the optimizer's gradient tensors.
4. Call optimizer.step() and update scale factor.
5. Save loss_scaler state_dict for resume purpose.

class mmselfsup.core.hooks.**InterCLRHook**(*extractor*, *clustering*, *centroids_update_interval*, *deal_with_small_clusters_interval*, *evaluate_interval*, *warmup_epochs=0*, *init_memory=True*, *initial=True*, *online_labels=True*, *interval=1*, *dist_mode=True*, *data_loaders=None*)

Hook for InterCLR.

This hook includes the clustering process in InterCLR.

Parameters

- **extractor** (*dict*) – Config dict for feature extraction.
- **clustering** (*dict*) – Config dict that specifies the clustering algorithm.
- **centroids_update_interval** (*int*) – Frequency of iterations to update centroids.
- **deal_with_small_clusters_interval** (*int*) – Frequency of iterations to deal with small clusters.
- **evaluate_interval** (*int*) – Frequency of iterations to evaluate clusters.
- **warmup_epochs** (*int, optional*) – The number of warmup epochs to set `intra_loss_weight=1` and `inter_loss_weight=0`. Defaults to 0.

- **init_memory** (*bool*) – Whether to initialize memory banks used in online labels. Defaults to True.
- **initial** (*bool*) – Whether to call the hook initially. Defaults to True.
- **online_labels** (*bool*) – Whether to use online labels. Defaults to True.
- **interval** (*int*) – Frequency of epochs to call the hook. Defaults to 1.
- **dist_mode** (*bool*) – Use distributed training or not. Defaults to True.
- **data_loaders** (*DataLoader*) – A PyTorch dataloader. Defaults to None.

class mmselfsup.core.hooks.**MomentumUpdateHook**(*end_momentum=1.0*, *update_interval=1*, ***kwargs*)

Hook for updating momentum parameter, used by BYOL, MoCoV3, etc.

This hook includes momentum adjustment following:

$$m = 1 - (1 - m_0) * (cos(pi * k/K) + 1)/2$$

where k is the current step, K is the total steps.

Parameters

- **end_momentum** (*float*) – The final momentum coefficient for the target network. Defaults to 1.
- **update_interval** (*int, optional*) – The momentum update interval of the weights. Defaults to 1.

class mmselfsup.core.hooks.**ODCHook**(*centroids_update_interval*, *deal_with_small_clusters_interval*, *evaluate_interval*, *reweight*, *reweight_pow*, *dist_mode=True*)

Hook for ODC.

This hook includes the online clustering process in ODC.

Parameters

- **centroids_update_interval** (*int*) – Frequency of iterations to update centroids.
- **deal_with_small_clusters_interval** (*int*) – Frequency of iterations to deal with small clusters.
- **evaluate_interval** (*int*) – Frequency of iterations to evaluate clusters.
- **reweight** (*bool*) – Whether to perform loss re-weighting.
- **reweight_pow** (*float*) – The power of re-weighting.
- **dist_mode** (*bool*) – Use distributed training or not. Defaults to True.

class mmselfsup.core.hooks.**SimSiamHook**(*fix_pred_lr*, *lr*, *adjust_by_epoch=True*, ***kwargs*)

Hook for SimSiam.

This hook is for SimSiam to fix learning rate of predictor.

Parameters

- **fix_pred_lr** (*bool*) – whether to fix the lr of predictor or not.
- **lr** (*float*) – the value of fixed lr.
- **adjust_by_epoch** (*bool, optional*) – whether to set lr by epoch or iter. Defaults to True.

before_train_epoch(*runner*)

fix lr of predictor.

class mmselfsup.core.hooks.**StepFixCosineAnnealingLrUpdaterHook**(*min_lr: Optional[float] = None*, *min_lr_ratio: Optional[float] = None*, ***kwargs*)

class mmselfsup.core.hooks.**SwAVHook**(*batch_size*, *epoch_queue_starts=15*, *crops_for_assign=[0, 1]*, *feat_dim=128*, *queue_length=0*, *interval=1*, ***kwargs*)

Hook for SwAV.

This hook builds the queue in SwAV according to `epoch_queue_starts`. The queue will be saved in `runner.work_dir` or loaded at start epoch if the path folder has queues saved before.

Parameters

- **batch_size** (*int*) – the batch size per GPU for computing.
- **epoch_queue_starts** (*int, optional*) – from this epoch, starts to use the queue. Defaults to 15.
- **crops_for_assign** (*list[int], optional*) – list of crops id used for computing assignments. Defaults to [0, 1].
- **feat_dim** (*int, optional*) – feature dimension of output vector. Defaults to 128.
- **queue_length** (*int, optional*) – length of the queue (0 for no queue). Defaults to 0.
- **interval** (*int, optional*) – the interval to save the queue. Defaults to 1.

35.2 optimizer

class mmselfsup.core.optimizer.**DefaultOptimizerConstructor**(*optimizer_cfg*, *paramwise_cfg=None*)

Rewrote default constructor for optimizers. By default each parameter share the same optimizer settings, and we provide an argument `paramwise_cfg` to specify parameter-wise settings. It is a dict and may contain the following fields: :param model: The model with parameters to be optimized. :type model: `nn.Module` :param optimizer_cfg: The config dict of the optimizer.

Positional fields are

- *type*: class name of the optimizer.

Optional fields are

- any arguments of the corresponding optimizer type, e.g., lr, weight_decay, momentum, etc.

Parameters **paramwise_cfg** (*dict, optional*) – Parameter-wise options. Defaults to None.

Example 1:

```
>>> model = torch.nn.modules.Conv1d(1, 1, 1)
>>> optimizer_cfg = dict(type='SGD', lr=0.01, momentum=0.9,
>>>                      weight_decay=0.0001)
>>> paramwise_cfg = dict('bias': dict(weight_decay=0.,
↪       lars_exclude=True))
>>> optim_builder = DefaultOptimizerConstructor(
>>>     optimizer_cfg, paramwise_cfg)
>>> optimizer = optim_builder(model)
```

class mmselfsup.core.optimizer.**LARS**(*params*, *lr=<required parameter>*, *momentum=0*, *weight_decay=0*, *dampening=0*, *eta=0.001*, *nesterov=False*, *eps=1e-08*)

Implements layer-wise adaptive rate scaling for SGD.

Parameters

- **params** (*iterable*) – Iterable of parameters to optimize or dicts defining parameter groups.
- **lr** (*float*) – Base learning rate.
- **momentum** (*float, optional*) – Momentum factor. Defaults to 0 ('m')
- **weight_decay** (*float, optional*) – Weight decay (L2 penalty). Defaults to 0. ('beta')
- **dampening** (*float, optional*) – Dampening for momentum. Defaults to 0.
- **eta** (*float, optional*) – LARS coefficient. Defaults to 0.001.
- **nesterov** (*bool, optional*) – Enables Nesterov momentum. Defaults to False.
- **eps** (*float, optional*) – A small number to avoid dviding zero. Defaults to 1e-8.

Based on Algorithm 1 of the following paper by You, Gitman, and Ginsburg. `Large Batch Training of Convolutional Networks:

<https://arxiv.org/abs/1708.03888>`_.

Example

```
>>> optimizer = LARS(model.parameters(), lr=0.1, momentum=0.9,
>>>                  weight_decay=1e-4, eta=1e-3)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

step(*closure=None*)

Performs a single optimization step.

Parameters **closure** (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class mmselfsup.core.optimizer.**TransformerFinetuneConstructor**(*optimizer_cfg*, *paramwise_cfg=None*)

Rewrote default constructor for optimizers.

By default each parameter share the same optimizer settings, and we provide an argument `paramwise_cfg` to specify parameter-wise settings. In addition, we provide two optional parameters, `model_type` and `layer_decay` to set the commonly used layer-wise learning rate decay schedule. Currently, we only support layer-wise learning rate schedule for swin and vit.

Parameters

- **optimizer_cfg** (*dict*) – The config dict of the optimizer. Positional fields are
  - *type*: class name of the optimizer.

  Optional fields are

  - any arguments of the corresponding optimizer type, e.g., lr, weight_decay, momentum, model_type, layer_decay, etc.
- **paramwise_cfg** (*dict, optional*) – Parameter-wise options. Defaults to None.

Example 1:

```
>>> model = torch.nn.modules.Conv1d(1, 1, 1)
>>> optimizer_cfg = dict(type='SGD', lr=0.01, momentum=0.9,
>>>                      weight_decay=0.0001, model_type='vit')
>>> paramwise_cfg = dict('bias': dict(weight_decay=0.,
↪       lars_exclude=True))
>>> optim_builder = TransformerFinetuneConstructor(
>>>     optimizer_cfg, paramwise_cfg)
>>> optimizer = optim_builder(model)
```

mmselfsup.core.optimizer.**build_optimizer**(*model*, *optimizer_cfg*)

Build optimizer from configs.

Parameters

- **model** (nn.Module) – The model with parameters to be optimized.
- **optimizer_cfg** (*dict*) – The config dict of the optimizer. Positional fields are:
  - type: class name of the optimizer.
  - lr: base learning rate.

  Optional fields are:
  - any arguments of the corresponding optimizer type, e.g., weight_decay, momentum, etc.
  - paramwise_options: a dict with regular expression as keys to match parameter names and a dict containing options as values. Options include 6 fields: lr, lr_mult, momentum, momentum_mult, weight_decay, weight_decay_mult.

Returns The initialized optimizer.

Return type torch.optim.Optimizer

Example

```
>>> model = torch.nn.modules.Conv1d(1, 1, 1)
>>> paramwise_options = {
>>>     '(bn|gn)(\d+)?.(weight|bias)': dict(weight_decay_mult=0.1),
>>>     '\Ahead.': dict(lr_mult=10, momentum=0)}
>>> optimizer_cfg = dict(type='SGD', lr=0.01, momentum=0.9,
>>>                      weight_decay=0.0001,
>>>                      paramwise_options=paramwise_options)
>>> optimizer = build_optimizer(model, optimizer_cfg)
```

CHAPTER

THIRTYSIX

MMSELFSUP.DATASETS

36.1 data_sources

class mmselfsup.datasets.data_sources.**BaseDataSource**(*data_prefix*, *classes=None*, *ann_file=None*, *test_mode=False*, *color_type='color'*, *channel_order='rgb'*, *file_client_args={'backend': 'disk'}*)

Datasource base class to load dataset information.

Parameters

- **data_prefix** (*str*) – the prefix of data path.
- **classes** (*str | Sequence[str], optional*) – Specify classes to load.
- **ann_file** (*str | None*) – the annotation file. When ann_file is str, the subclass is expected to read from the ann_file. When ann_file is None, the subclass is expected to read according to data_prefix.
- **test_mode** (*bool*) – in train mode or test mode. Defaults to False.
- **color_type** (*str*) – The flag argument for `mmcv.imfrombytes()`. Defaults to color.
- **channel_order** (*str*) – The channel order of images when loaded. Defaults to rgb.
- **file_client_args** (*dict*) – Arguments to instantiate a FileClient. See `mmcv.fileio.FileClient` for details. Defaults to dict(backend='disk').

get_cat_ids(*idx*)

Get category id by index.

Parameters **idx** (*int*) – Index of data.

Returns Image category of specified index.

Return type int

classmethod **get_classes**(*classes=None*)

Get class names of current dataset.

Parameters **classes** (*Sequence[str] | str | None*) – If classes is None, use default CLASSES defined by builtin dataset. If classes is a string, take it as a file name. The file contains the name of classes where each line contains one class name. If classes is a tuple or list, override the CLASSES defined by the dataset.

Returns Names of categories of the dataset.

Return type tuple[str] or list[str]

get_gt_labels()
Get all ground-truth labels (categories).

Returns categories for all images.

Return type list[int]

get_img(*idx*)
Get image by index.

Parameters **idx** (*int*) – Index of data.

Returns PIL Image format.

Return type Image

class mmselfsup.datasets.data_sources.**CIFAR10**(*data_prefix*, *classes=None*, *ann_file=None*, *test_mode=False*, *color_type='color'*, *channel_order='rgb'*, *file_client_args={'backend': 'disk'}*)
CIFAR10 Dataset.

This implementation is modified from https://github.com/pytorch/vision/blob/master/torchvision/datasets/cifar.py

class mmselfsup.datasets.data_sources.**CIFAR100**(*data_prefix*, *classes=None*, *ann_file=None*, *test_mode=False*, *color_type='color'*, *channel_order='rgb'*, *file_client_args={'backend': 'disk'}*)
CIFAR100 Dataset.

class mmselfsup.datasets.data_sources.**ImageList**(*data_prefix*, *classes=None*, *ann_file=None*, *test_mode=False*, *color_type='color'*, *channel_order='rgb'*, *file_client_args={'backend': 'disk'}*)
The implementation for loading any image list file.

The *ImageList* can load an annotation file or a list of files and merge all data records to one list. If data is unlabeled, the gt_label will be set -1.

class mmselfsup.datasets.data_sources.**ImageNet**(*data_prefix*, *classes=None*, *ann_file=None*, *test_mode=False*, *color_type='color'*, *channel_order='rgb'*, *file_client_args={'backend': 'disk'}*)
ImageNet Dataset.

This implementation is modified from https://github.com/pytorch/vision/blob/master/torchvision/datasets/imagenet.py

class mmselfsup.datasets.data_sources.**ImageNet21k**(*data_prefix*, *classes=None*, *ann_file=None*, *multi_label=False*, *recursion_subdir=False*, *test_mode=False*)
ImageNet21k Dataset. Since the dataset ImageNet21k is extremely big, cantains 21k+ classes and 1.4B files. This class has improved the following points on the basis of the class `ImageNet`, in order to save memory usage and time required :

- Delete the samples attribute
- using 'slots' create a Data_item tp replace dict
- Modify setting `info` dict from function `load_annotations` to function `prepare_data`
- using int instead of np.array(..., np.int64)

Parameters

- **data_prefix** (*str*) – the prefix of data path
- **ann_file** (*str | None*) – the annotation file. When ann_file is str, the subclass is expected to read from the ann_file. When ann_file is None, the subclass is expected to read according to data_prefix
- **test_mode** (*bool*) – in train mode or test mode
- **multi_label** (*bool*) – use multi label or not.
- **recursion_subdir** (*bool*) – whether to use sub-directory pictures, which are meet the conditions in the folder under category directory.

load_annotations()
load dataset annotations.

36.2 pipelines

class mmselfsup.datasets.pipelines.**BEiTMaskGenerator**(*input_size: int*, *num_masking_patches: int*, *min_num_patches: int = 4*, *max_num_patches: Optional[int] = None*, *min_aspect: float = 0.3*, *max_aspect: Optional[float] = None*)

Generate mask for image.

This module is borrowed from https://github.com/microsoft/unilm/tree/master/beit

Parameters

- **input_size** (*int*) – The size of input image.
- **num_masking_patches** (*int*) – The number of patches to be masked.
- **min_num_patches** (*int*) – The minimum number of patches to be masked in the process of generating mask. Defaults to 4.
- **max_num_patches** (*int, optional*) – The maximum number of patches to be masked in the process of generating mask. Defaults to None.
- **min_aspect** (*float, optional*) – The minimum aspect ratio of mask blocks. Defaults to 0.3.
- **min_aspect** – The minimum aspect ratio of mask blocks. Defaults to None.

class mmselfsup.datasets.pipelines.**GaussianBlur**(*sigma_min*, *sigma_max*, *p=0.5*)

GaussianBlur augmentation refers to `SimCLR.

<https://arxiv.org/abs/2002.05709>`_.

Parameters

- **sigma_min** (*float*) – The minimum parameter of Gaussian kernel std.
- **sigma_max** (*float*) – The maximum parameter of Gaussian kernel std.
- **p** (*float, optional*) – Probability. Defaults to 0.5.

class mmselfsup.datasets.pipelines.**Lighting**(*alphastd=0.1*)

Lighting noise(AlexNet - style PCA - based noise).

Parameters **alphastd** (*float, optional*) – The parameter for Lighting. Defaults to 0.1.

class mmselfsup.datasets.pipelines.**MaskFeatMaskGenerator**(*mask_window_size: int = 14*, *mask_ratio: float = 0.4*, *min_num_patches: int = 15*, *max_num_patches: Optional[int] = None*, *min_aspect: float = 0.3*, *max_aspect: Optional[float] = None*)

Generate random block mask for each image.

This module is borrowed from https://github.com/facebookresearch/SlowFast/blob/main/slowfast/datasets/transform.py :param mask_window_size: Size of input image. Defaults to 14. :type mask_window_size: int :param mask_ratio: The mask ratio of image. Defaults to 0.4. :type mask_ratio: float :param min_num_patches: Minimum number of patches that require masking.

Defaults to 15.

Parameters

- **max_num_patches** (*int, optional*) – Maximum number of patches that require masking. Defaults to None.
- **min_aspect** (*int*) – Minimum aspect of patches. Defaults to 0.3.
- **max_aspect** (*float, optional*) – Maximum aspect of patches. Defaults to None.

class mmselfsup.datasets.pipelines.**RandomAppliedTrans**(*transforms*, *p=0.5*)

Randomly applied transformations.

Parameters

- **transforms** (*list[dict]*) – List of transformations in dictionaries.
- **p** (*float, optional*) – Probability. Defaults to 0.5.

class mmselfsup.datasets.pipelines.**RandomAug**(*input_size=None*, *color_jitter=None*, *auto_augment=None*, *interpolation=None*, *re_prob=None*, *re_mode=None*, *re_count=None*, *mean=None*, *std=None*)

RandAugment data augmentation method based on "RandAugment: Practical automated data augmentation with a reduced search space".

This code is borrowed from <https://github.com/pengzhiliang/MAE-pytorch>

class mmselfsup.datasets.pipelines.**SimMIMMaskGenerator**(*input_size: int = 192*, *mask_patch_size: int = 32*, *model_patch_size: int = 4*, *mask_ratio: float = 0.6*)

Generate random block mask for each Image.

This module is used in SimMIM to generate masks.

Parameters

- **input_size** (*int*) – Size of input image. Defaults to 192.
- **mask_patch_size** (*int*) – Size of each block mask. Defaults to 32.
- **model_patch_size** (*int*) – Patch size of each token. Defaults to 4.
- **mask_ratio** (*float*) – The mask ratio of image. Defaults to 0.6.

class mmselfsup.datasets.pipelines.**Solarization**(*threshold=128*, *p=0.5*)

Solarization augmentation refers to `BYOL.

<https://arxiv.org/abs/2006.07733>`_.

Parameters

- **threshold** (*float, optional*) – The solarization threshold. Defaults to 128.
- **p** (*float, optional*) – Probability. Defaults to 0.5.

class mmselfsup.datasets.pipelines.**ToTensor**
Convert image or a sequence of images to tensor.

This module can not only convert a single image to tensor, but also a sequence of images.

36.3 samplers

class mmselfsup.datasets.samplers.**DistributedGivenIterationSampler**(*dataset*, *total_iter*, *batch_size*, *num_replicas=None*, *rank=None*, *last_iter=- 1*)

gen_new_list()
Each process shuffle all list with same seed, and pick one piece according to rank.

class mmselfsup.datasets.samplers.**DistributedGroupSampler**(*dataset*, *samples_per_gpu=1*, *num_replicas=None*, *rank=None*)
Sampler that restricts data loading to a subset of the dataset.

It is especially useful in conjunction with `torch.nn.parallel.DistributedDataParallel`. In such case, each process can pass a DistributedSampler instance as a DataLoader sampler, and load a subset of the original dataset that is exclusive to it.

Note: Dataset is assumed to be of constant size.

Parameters

- **dataset** – Dataset used for sampling.
- **num_replicas** (*optional*) – Number of processes participating in distributed training.
- **rank** (*optional*) – Rank of the current process within num_replicas.

class mmselfsup.datasets.samplers.**DistributedSampler**(*dataset*, *num_replicas=None*, *rank=None*, *shuffle=True*, *replace=False*, *seed=0*)

class mmselfsup.datasets.samplers.**GroupSampler**(*dataset*, *samples_per_gpu=1*)

36.4 datasets

class mmselfsup.datasets.**BaseDataset**(*data_source*, *pipeline*, *prefetch=False*)
Base dataset class.

The base dataset can be inherited by different algorithm's datasets. After *__init__*, the data source and pipeline will be built. Besides, the algorithm specific dataset implements different operations after obtaining images from data sources.

Parameters

- **data_source** (*dict*) – Data source defined in *mmselfsup.datasets.data_sources*.

- **pipeline** (*list[dict]*) – A list of dict, where each element represents an operation defined in *mmselfsup.datasets.pipelines*.
- **prefetch** (*bool, optional*) – Whether to prefetch data. Defaults to False.

class mmselfsup.datasets.**ConcatDataset**(*datasets*)

A wrapper of concatenated dataset.

Same as `torch.utils.data.dataset.ConcatDataset`, but concat the group flag for image aspect ratio.

Parameters **datasets** (list[Dataset]) – A list of datasets.

class mmselfsup.datasets.**DeepClusterDataset**(*data_source*, *pipeline*, *prefetch=False*)

Dataset for DC and ODC.

The dataset initializes clustering labels and assigns it during training.

Parameters

- **data_source** (*dict*) – Data source defined in *mmselfsup.datasets.data_sources*.
- **pipeline** (*list[dict]*) – A list of dict, where each element represents an operation defined in *mmselfsup.datasets.pipelines*.
- **prefetch** (*bool, optional*) – Whether to prefetch data. Defaults to False.

class mmselfsup.datasets.**MultiViewDataset**(*data_source*, *num_views*, *pipelines*, *prefetch=False*)

The dataset outputs multiple views of an image.

The number of views in the output dict depends on *num_views*. The image can be processed by one pipeline or multiple piepelines.

Parameters

- **data_source** (*dict*) – Data source defined in *mmselfsup.datasets.data_sources*.
- **num_views** (*list*) – The number of different views.
- **pipelines** (*list[list[dict]]*) – A list of pipelines, where each pipeline contains elements that represents an operation defined in *mmselfsup.datasets.pipelines*.
- **prefetch** (*bool, optional*) – Whether to prefetch data. Defaults to False.

Examples

```
>>> dataset = MultiViewDataset(data_source, [2], [pipeline])
>>> output = dataset[idx]
The output got 2 views processed by one pipeline.
```

```
>>> dataset = MultiViewDataset(
>>>     data_source, [2, 6], [pipeline1, pipeline2])
>>> output = dataset[idx]
The output got 8 views processed by two pipelines, the first two views
were processed by pipeline1 and the remaining views by pipeline2.
```

class mmselfsup.datasets.**RelativeLocDataset**(*data_source*, *pipeline*, *format_pipeline*, *prefetch=False*)

Dataset for relative patch location.

The dataset crops image into several patches and concatenates every surrounding patch with center one. Finally it also outputs corresponding labels *0, 1, 2, 3, 4, 5, 6, 7*.

Parameters

- **data_source** (*dict*) – Data source defined in *mmselfsup.datasets.data_sources*.
- **pipeline** (*list[dict]*) – A list of dict, where each element represents an operation defined in *mmselfsup.datasets.pipelines*.
- **format_pipeline** (*list[dict]*) – A list of dict, it converts input format from PIL.Image to Tensor. The operation is defined in *mmselfsup.datasets.pipelines*.
- **prefetch** (*bool, optional*) – Whether to prefetch data. Defaults to False.

class mmselfsup.datasets.**RepeatDataset**(*dataset*, *times*)
A wrapper of repeated dataset.

The length of repeated dataset will be *times* larger than the original dataset. This is useful when the data loading time is long but the dataset is small. Using RepeatDataset can reduce the data loading time between epochs.

Parameters

- **dataset** (Dataset) – The dataset to be repeated.
- **times** (*int*) – Repeat times.

class mmselfsup.datasets.**RotationPredDataset**(*data_source*, *pipeline*, *prefetch=False*)
Dataset for rotation prediction.

The dataset rotates the image with 0, 90, 180, and 270 degrees and outputs labels *0, 1, 2, 3* correspodingly.

Parameters

- **data_source** (*dict*) – Data source defined in *mmselfsup.datasets.data_sources*.
- **pipeline** (*list[dict]*) – A list of dict, where each element represents an operation defined in *mmselfsup.datasets.pipelines*.
- **prefetch** (*bool, optional*) – Whether to prefetch data. Defaults to False.

class mmselfsup.datasets.**SingleViewDataset**(*data_source*, *pipeline*, *prefetch=False*)
The dataset outputs one view of an image, containing some other information such as label, idx, etc.

Parameters

- **data_source** (*dict*) – Data source defined in *mmselfsup.datasets.data_sources*.
- **pipeline** (*list[dict]*) – A list of dict, where each element represents an operation defined in *mmselfsup.datasets.pipelines*.
- **prefetch** (*bool, optional*) – Whether to prefetch data. Defaults to False.

evaluate(*results*, *logger=None*, *topk=(1, 5)*)
The evaluation function to output accuracy.

Parameters

- **results** (*dict*) – The key-value pair is the output head name and corresponding prediction values.
- **logger** (*logging.Logger | str | None, optional*) – The defined logger to be used. Defaults to None.
- **topk** (*tuple(int)*) – The output includes topk accuracy.

mmselfsup.datasets.**build_dataloader**(*dataset*, *imgs_per_gpu=None*, *samples_per_gpu=None*, *workers_per_gpu=1*, *num_gpus=1*, *dist=True*, *shuffle=True*, *replace=False*, *seed=None*, *pin_memory=True*, *persistent_workers=True*, ***kwargs*)
Build PyTorch DataLoader.

In distributed training, each GPU/process has a dataloader. In non-distributed training, there is only one dataloader for all GPUs.

Parameters

- **dataset** (*Dataset*) – A PyTorch dataset.
- **imgs_per_gpu** (*int*) – (Deprecated, please use samples_per_gpu) Number of images on each GPU, i.e., batch size of each GPU. Defaults to None.
- **samples_per_gpu** (*int*) – Number of images on each GPU, i.e., batch size of each GPU. Defaults to None.
- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU. *persistent_workers* option needs num_workers > 0. Defaults to 1.
- **num_gpus** (*int*) – Number of GPUs. Only used in non-distributed training.
- **dist** (*bool*) – Distributed training/test or not. Defaults to True.
- **shuffle** (*bool*) – Whether to shuffle the data at every epoch. Defaults to True.
- **replace** (*bool*) – Replace or not in random shuffle. It works on when shuffle is True. Defaults to False.
- **seed** (*int*) – set seed for dataloader.
- **pin_memory** (*bool, optional*) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them. Defaults to True.
- **persistent_workers** (*bool*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers Dataset instances alive. The argument also has effect in PyTorch>=1.7.0. Defaults to True.
- **kwargs** – any keyword argument to be used to initialize DataLoader

Returns A PyTorch dataloader.

Return type DataLoader

CHAPTER

THIRTYSEVEN

MMSELFSUP.MODELS

37.1 algorithms

class mmselfsup.models.algorithms.**BYOL**(*backbone*, *neck=None*, *head=None*, *base_momentum=0.996*, *init_cfg=None*, ***kwargs*)

BYOL.

Implementation of Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning. The momentum adjustment is in *core/hooks/byol_hook.py*.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **base_momentum** (*float*) – The base momentum coefficient for the target network. Defaults to 0.996.

extract_feat(*img*)

Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, ***kwargs*)

Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

momentum_update()

Momentum update of the target network.

class mmselfsup.models.algorithms.**BarlowTwins**(*backbone: Optional[dict] = None*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *init_cfg: Optional[dict] = None*, ***kwargs*)

BarlowTwins.

Implementation of Barlow Twins: Self-Supervised Learning via Redundancy Reduction. Part of the code is borrowed from: https://github.com/facebookresearch/barlowtwins/blob/main/main.py.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone. Defaults to None.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **init_cfg** (*dict*) – Config dict for weight initialization. Defaults to None.

extract_feat(*img: torch.Tensor*) → torch.Tensor
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img: List[torch.Tensor]*) → dict
Forward computation during training.

Parameters **img** (*List[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**BaseModel**(*init_cfg=None*)
Base model class for self-supervised learning.

abstract extract_feat(*imgs*)
Function to extract features from backbone.

Parameters

- **img** (*Tensor*) – Input images. Typically these should be mean centered
- **std scaled.** (*and*) –

forward(*img*, *mode='train'*, ***kwargs*)
Forward function of model.

Calls either forward_train, forward_test or extract_feat function according to the mode.

forward_test(*imgs*, ***kwargs*)

Parameters

- **img** (*Tensor*) – List of tensors. Typically these should be mean centered and std scaled.
- **kwargs** (*keyword arguments*) – Specific to concrete implementation.

abstract forward_train(*imgs*, ***kwargs*)

Parameters

- **img** (*[Tensor*) – List of tensors. Typically these should be mean centered and std scaled.
- **kwargs** (*keyword arguments*) – Specific to concrete implementation.

train_step(*data*, *optimizer*)
The iteration step during training.

This method defines an iteration step during training, except for the back propagation and optimizer updating, which are done in an optimizer hook. Note that in some complicated cases or models, the whole process including back propagation and optimizer updating are also defined in this method, such as GAN.

Parameters

- **data** (*dict*) – The output of dataloader.
- **optimizer** (torch.optim.Optimizer | dict) – The optimizer of runner is passed to `train_step()`. This argument is unused and reserved.

Returns

Dict of outputs. The following fields are contained.

- loss (torch.Tensor): A tensor for back propagation, which can be a weighted sum of multiple losses.
- log_vars (dict): Dict contains all the variables to be sent to the logger.
- num_samples (int): Indicates the batch size (when the model is DDP, it means the batch size on each GPU), which is used for averaging the logs.

Return type dict

val_step(*data*, *optimizer*)
The iteration step during validation.

This method shares the same signature as *train_step()*, but used during val epochs. Note that the evaluation after training epochs is not implemented with this method, but an evaluation hook.

class mmselfsup.models.algorithms.**CAE**(*backbone: Optional[dict] = None*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *base_momentum: float = 0.0*, *init_cfg: Optional[dict] = None*, ***kwargs*)

CAE.

Implementation of Context Autoencoder for Self-Supervised Representation Learning.

Parameters

- **backbone** (*dict, optional*) – Config dict for module of backbone.
- **neck** (*dict, optional*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict, optional*) – Config dict for module of loss functions. Defaults to None.
- **base_momentum** (*float*) – The base momentum coefficient for the target network. Defaults to 0.0.
- **init_cfg** (*dict, optional*) – the config to control the initialization.

extract_feat(*img: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Function to extract features from backbone.

Parameters

- **img** (*Tensor*) – Input images. Typically these should be mean centered
- **std scaled.** (*and*) –

forward_train(*samples: Sequence*, ***kwargs*) → dict
Args: img ([Tensor): List of tensors. Typically these should be

mean centered and std scaled.

kwargs (keyword arguments): Specific to concrete implementation.

init_weights() → None
Initialize the weights.

momentum_update() → None
Momentum update of the teacher network.

class mmselfsup.models.algorithms.**Classification**(*backbone*, *with_sobel=False*, *head=None*, *train_cfg=None*, *init_cfg=None*)

Simple image classification.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **with_sobel** (*bool*) – Whether to apply a Sobel filter. Defaults to False.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_test(*img*, ***kwargs*)
Forward computation during test.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of output features.

Return type dict[str, Tensor]

forward_train(*img*, *label*, ***kwargs*)
Forward computation during training.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **label** (*Tensor*) – Ground-truth labels.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**DeepCluster**(*backbone*, *with_sobel=True*, *neck=None*, *head=None*, *init_cfg=None*)

DeepCluster.

Implementation of Deep Clustering for Unsupervised Learning of Visual Features. The clustering operation is in *core/hooks/deepcluster_hook.py*.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **with_sobel** (*bool*) – Whether to apply a Sobel filter on images. Defaults to True.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_test(*img*, ***kwargs*)
Forward computation during test.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of output features.

Return type dict[str, Tensor]

forward_train(*img*, *pseudo_label*, ***kwargs*)
Forward computation during training.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **pseudo_label** (*Tensor*) – Label assignments.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

set_reweight(*labels*, *reweight_pow=0.5*)
Loss re-weighting.

Re-weighting the loss according to the number of samples in each class.

Parameters

- **labels** (*numpy.ndarray*) – Label assignments.
- **reweight_pow** (*float*) – The power of re-weighting. Defaults to 0.5.

class mmselfsup.models.algorithms.**DenseCL**(*backbone*, *neck=None*, *head=None*, *queue_len=65536*, *feat_dim=128*, *momentum=0.999*, *loss_lambda=0.5*, *init_cfg=None*, ***kwargs*)
DenseCL.

Implementation of Dense Contrastive Learning for Self-Supervised Visual Pre-Training. Borrowed from the authors' code: https://github.com/WXinlong/DenseCL. The loss_lambda warmup is in *core/hooks/densecl_hook.py*.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **queue_len** (*int*) – Number of negative keys maintained in the queue. Defaults to 65536.
- **feat_dim** (*int*) – Dimension of compact feature vectors. Defaults to 128.
- **momentum** (*float*) – Momentum coefficient for the momentum-updated encoder. Defaults to 0.999.
- **loss_lambda** (*float*) – Loss weight for the single and dense contrastive loss. Defaults to 0.5.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_test(*img*, ***kwargs*)
Forward computation during test.

Parameters **img** (*Tensor*) – Input of two concatenated images of shape (N, 2, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of normalized output features.

Return type dict(Tensor)

forward_train(*img*, ***kwargs*)
Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

init_weights()
Init weights and copy query encoder init weights to key encoder.

class mmselfsup.models.algorithms.**InterCLRMoCo**(*backbone*, *neck=None*, *head=None*, *queue_len=65536*, *feat_dim=128*, *momentum=0.999*, *memory_bank=None*, *online_labels=True*, *neg_num=16384*, *neg_sampling='semihard'*, *semihard_neg_pool_num=128000*, *semieasy_neg_pool_num=128000*, *intra_cos_marign_loss=False*, *intra_cos_margin=0*, *intra_arc_marign_loss=False*, *intra_arc_margin=0*, *inter_cos_marign_loss=True*, *inter_cos_margin=- 0.5*, *inter_arc_marign_loss=False*, *inter_arc_margin=0*, *intra_loss_weight=0.75*, *inter_loss_weight=0.25*, *share_neck=True*, *num_classes=10000*, *init_cfg=None*, ***kwargs*)

MoCo-InterCLR.

Official implementation of Delving into Inter-Image Invariance for Unsupervised Visual Representations. The clustering operation is in *core/hooks/interclr_hook.py*.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **queue_len** (*int*) – Number of negative keys maintained in the queue. Defaults to 65536.
- **feat_dim** (*int*) – Dimension of compact feature vectors. Defaults to 128.
- **momentum** (*float*) – Momentum coefficient for the momentum-updated encoder. Defaults to 0.999.
- **memory_bank** (*dict*) – Config dict for module of memory banks. Defaults to None.
- **online_labels** (*bool*) – Whether to use online labels. Defaults to True.
- **neg_num** (*int*) – Number of negative samples for inter-image branch. Defaults to 16384.
- **neg_sampling** (*str*) – Negative sampling strategy. Support 'hard', 'semihard', 'random', 'semieasy'. Defaults to 'semihard'.
- **semihard_neg_pool_num** (*int*) – Number of negative samples for semi-hard nearest neighbor pool. Defaults to 128000.
- **semieasy_neg_pool_num** (*int*) – Number of negative samples for semi-easy nearest neighbor pool. Defaults to 128000.
- **intra_cos_marign_loss** (*bool*) – Whether to use a cosine margin for intra-image branch. Defaults to False.
- **intra_cos_marign** (*float*) – Intra-image cosine margin. Defaults to 0.
- **intra_arc_marign_loss** (*bool*) – Whether to use an arc margin for intra-image branch. Defaults to False.
- **intra_arc_marign** (*float*) – Intra-image arc margin. Defaults to 0.
- **inter_cos_marign_loss** (*bool*) – Whether to use a cosine margin for inter-image branch. Defaults to True.
- **inter_cos_marign** (*float*) – Inter-image cosine margin. Defaults to -0.5.
- **inter_arc_marign_loss** (*bool*) – Whether to use an arc margin for inter-image branch. Defaults to False.
- **inter_arc_marign** (*float*) – Inter-image arc margin. Defaults to 0.
- **intra_loss_weight** (*float*) – Loss weight for intra-image branch. Defaults to 0.75.
- **inter_loss_weight** (*float*) – Loss weight for inter-image branch. Defaults to 0.25.
- **share_neck** (*bool*) – Whether to share the neck for intra- and inter-image branches. Defaults to True.
- **num_classes** (*int*) – Number of clusters. Defaults to 10000.

contrast_inter(*q*, *idx*)

Inter-image invariance learning.

Parameters

- **q** (*Tensor*) – Query features with shape (N, C).
- **idx** (*Tensor*) – Index corresponding to each query.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

contrast_intra(*q*, *k*)
Intra-image invariance learning.

Parameters

- **q** (*Tensor*) – Query features with shape (N, C).
- **k** (*Tensor*) – Key features with shape (N, C).

Returns A dictionary of loss components.

Return type dict[str, Tensor]

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, *idx*, ***kwargs*)
Forward computation during training.

Parameters

- **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **idx** (*Tensor*) – Index corresponding to each image.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

init_weights()
Initialize base_encoder with init_cfg defined in backbone.

class mmselfsup.models.algorithms.**MAE**(*backbone: dict*, *neck: dict*, *head: dict*, *init_cfg: Optional[dict] = None*)
MAE.

Implementation of Masked Autoencoders Are Scalable Vision Learners.

Parameters

- **backbone** (*dict*) – Config dict for encoder. Defaults to None.
- **neck** (*dict*) – Config dict for encoder. Defaults to None.
- **head** (*dict*) – Config dict for loss functions. Defaults to None.
- **init_cfg** (*dict, optional*) – Config dict for weight initialization. Defaults to None.

extract_feat(*img: torch.Tensor*) → Tuple[torch.Tensor]
Function to extract features from backbone.

Parameters **img** (*torch.Tensor*) – Input images of shape (N, C, H, W).

Returns backbone outputs.

Return type Tuple[torch.Tensor]

forward_test(*img: torch.Tensor*, ***kwargs*) → Tuple[torch.Tensor, torch.Tensor]
Forward computation during testing.

Parameters

- **img** (*torch.Tensor*) – Input images of shape (N, C, H, W).
- **kwargs** – Any keyword arguments to be used to forward.

Returns

Output of model test.

- mask: Mask used to mask image.
- pred: The output of neck.

Return type Tuple[torch.Tensor, torch.Tensor]

forward_train(*img: torch.Tensor*, ***kwargs*) → Dict[str, torch.Tensor]
Forward computation during training.

Parameters

- **img** (*torch.Tensor*) – Input images of shape (N, C, H, W).
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type Dict[str, torch.Tensor]

init_weights()
Initialize the weights.

class mmselfsup.models.algorithms.**MMClsImageClassifierWrapper**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *pretrained: Optional[str] = None*, *train_cfg: Optional[dict] = None*, *init_cfg: Optional[dict] = None*)
Workaround to use models from mmclassificaiton.

Since the output of classifier from mmclassification is not compatible with mmselfsup's evaluation function. We rewrite some key components from mmclassification.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict, optional*) – Config dict for module of neck. Defaults to None.
- **head** (*dict, optional*) – Config dict for module of loss functions. Defaults to None.
- **pretrained** (*str, optional*) – The path of pre-trained checkpoint. Defaults to None.
- **train_cfg** (*dict, optional*) – Config dict for pre-processing utils, e.g. mixup. Defaults to None.
- **init_cfg** (*dict, optional*) – Config dict for initialization. Defaults to None.

forward(*img*, *mode='train'*, ***kwargs*)
Forward function of model.

Calls either forward_train, forward_test or extract_feat function according to the mode.

forward_test(*imgs*, ***kwargs*)

Parameters **imgs** (*List[Tensor]*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch.

forward_train(*img*, *label*, ***kwargs*)
Forward computation during training.

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **label** (*Tensor*) – It should be of shape (N, 1) encoding the ground-truth label of input images for single label task. It shoulf be of shape (N, C) encoding the ground-truth label of input images for multi-labels task.

Returns a dictionary of loss components

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**MaskFeat**(*backbone: dict*, *head: dict*, *hog_para: dict*, *init_cfg: Optional[dict] = None*)
MaskFeat.

Implementation of Masked Feature Prediction for Self-Supervised Visual Pre-Training. :param backbone: Config dict for encoder. :type backbone: dict :param head: Config dict for loss functions. :type head: dict :param hog_para: Config dict for hog layer.

dict['nbins', int]: Number of bin. Defaults to 9. dict['pool', float]: Number of cell. Defaults to 8. dict['gaussian_window', int]: Size of gaussian kernel.

Defaults to 16.

Parameters **init_cfg** (*dict*) – Config dict for weight initialization. Defaults to None.

extract_feat(*input: List[torch.Tensor]*) → torch.Tensor
Function to extract features from backbone.

Parameters **input** (*List[torch.Tensor, torch.Tensor]*) – Input images and masks.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*input: List[torch.Tensor]*, ***kwargs*) → dict
Forward computation during training.

Parameters

- **input** (*List[torch.Tensor, torch.Tensor]*) – Input images and masks.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**MoCo**(*backbone*, *neck=None*, *head=None*, *queue_len=65536*, *feat_dim=128*, *momentum=0.999*, *init_cfg=None*, ***kwargs*)

MoCo.

Implementation of Momentum Contrast for Unsupervised Visual Representation Learning. Part of the code is borrowed from: https://github.com/facebookresearch/moco/blob/master/moco/builder.py.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **queue_len** (*int*) – Number of negative keys maintained in the queue. Defaults to 65536.
- **feat_dim** (*int*) – Dimension of compact feature vectors. Defaults to 128.
- **momentum** (*float*) – Momentum coefficient for the momentum-updated encoder. Defaults to 0.999.

extract_feat(*img*)

Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, ***kwargs*)

Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**MoCoV3**(*backbone*, *neck*, *head*, *base_momentum=0.99*, *init_cfg=None*, ***kwargs*)

MoCo v3.

Implementation of An Empirical Study of Training Self-Supervised Vision Transformers.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **base_momentum** (*float*) – Momentum coefficient for the momentum-updated encoder. Defaults to 0.99.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None

extract_feat(*img*)

Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images. Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, ***kwargs*)
Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images. Typically these should be mean centered and std scaled.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

init_weights()
Initialize base_encoder with init_cfg defined in backbone.

momentum_update()
Momentum update of the momentum encoder.

class mmselfsup.models.algorithms.**NPID**(*backbone*, *neck=None*, *head=None*, *memory_bank=None*, *neg_num=65536*, *ensure_neg=False*, *init_cfg=None*)
NPID.

Implementation of Unsupervised Feature Learning via Non-parametric Instance Discrimination.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **memory_bank** (*dict*) – Config dict for module of memory banks. Defaults to None.
- **neg_num** (*int*) – Number of negative samples for each image. Defaults to 65536.
- **ensure_neg** (*bool*) – If False, there is a small probability that negative samples contain positive ones. Defaults to False.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, *idx*, ***kwargs*)
Forward computation during training.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **idx** (*Tensor*) – Index corresponding to each image.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**ODC**(*backbone*, *with_sobel=False*, *neck=None*, *head=None*, *memory_bank=None*, *init_cfg=None*)

ODC.

Official implementation of Online Deep Clustering for Unsupervised Representation Learning. The operation w.r.t. memory bank and loss re-weighting is in

core/hooks/odc_hook.py.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **with_sobel** (*bool*) – Whether to apply a Sobel filter on images. Defaults to False.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.
- **memory_bank** (*dict*) – Module of memory banks. Defaults to None.

extract_feat(*img*)

Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_test(*img*, ***kwargs*)

Forward computation during test.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of output features.

Return type dict[str, Tensor]

forward_train(*img*, *idx*, ***kwargs*)

Forward computation during training.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **idx** (*Tensor*) – Index corresponding to each image.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

set_reweight(*labels=None*, *reweight_pow=0.5*)

Loss re-weighting.

Re-weighting the loss according to the number of samples in each class.

Parameters

- **labels** (*numpy.ndarray*) – Label assignments. Defaults to None.
- **reweight_pow** (*float*) – The power of re-weighting. Defaults to 0.5.

class mmselfsup.models.algorithms.**RelativeLoc**(*backbone*, *neck=None*, *head=None*, *init_cfg=None*)

Relative patch location.

Implementation of Unsupervised Visual Representation Learning by Context Prediction.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)

Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward(*img*, *patch_label=None*, *mode='train'*, ***kwargs*)

Forward function to select mode and modify the input image shape.

Parameters **img** (*Tensor*) – Input images, the shape depends on mode. Typically these should be mean centered and std scaled.

forward_test(*img*, ***kwargs*)

Forward computation during training.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of output features.

Return type dict[str, Tensor]

forward_train(*img*, *patch_label*, ***kwargs*)

Forward computation during training.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **patch_label** (*Tensor*) – Labels for the relative patch locations.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**RotationPred**(*backbone*, *head=None*, *init_cfg=None*)

Rotation prediction.

Implementation of Unsupervised Representation Learning by Predicting Image Rotations.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward(*img*, *rot_label=None*, *mode='train'*, ***kwargs*)
Forward function to select mode and modify the input image shape.

Parameters **img** (*Tensor*) – Input images, the shape depends on mode. Typically these should be mean centered and std scaled.

forward_test(*img*, ***kwargs*)
Forward computation during training.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of output features.

Return type dict[str, Tensor]

forward_train(*img*, *rot_label*, ***kwargs*)
Forward computation during training.

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **rot_label** (*Tensor*) – Labels for the rotations.
- **kwargs** – Any keyword arguments to be used to forward.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**SimCLR**(*backbone*, *neck=None*, *head=None*, *init_cfg=None*)
SimCLR.

Implementation of A Simple Framework for Contrastive Learning of Visual Representations.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, ***kwargs*)
Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.algorithms.**SimMIM**(*backbone: dict*, *neck: dict*, *head: dict*, *init_cfg: Optional[dict] = None*)
SimMIM.

Implementation of SimMIM: A Simple Framework for Masked Image Modeling.

Parameters

- **backbone** (*dict*) – Config dict for encoder. Defaults to None.
- **neck** (*dict*) – Config dict for encoder. Defaults to None.
- **head** (*dict*) – Config dict for loss functions. Defaults to None.
- **init_cfg** (*dict, optional*) – Config dict for weight initialization. Defaults to None.

extract_feat(*img: torch.Tensor*) → tuple
Function to extract features from backbone.

Parameters **img** (*torch.Tensor*) – Input images of shape (N, C, H, W).

Returns Latent representations of images.

Return type tuple[Tensor]

forward_train(*x: List[torch.Tensor]*, ***kwargs*) → dict
Forward the masked image and get the reconstruction loss.

Parameters **x** (*List[torch.Tensor, torch.Tensor]*) – Images and masks.

Returns Reconstructed loss.

Return type dict

class mmselfsup.models.algorithms.**SimSiam**(*backbone*, *neck=None*, *head=None*, *init_cfg=None*, ***kwargs*)
SimSiam.

Implementation of Exploring Simple Siamese Representation Learning. The operation of fixing learning rate of predictor is in *core/hooks/simsiam_hook.py*.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns backbone outputs.

Return type tuple[Tensor]

forward_train(*img*)
Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components

Return type loss[str, Tensor]

class mmselfsup.models.algorithms.**SwAV**(*backbone*, *neck=None*, *head=None*, *init_cfg=None*, ***kwargs*)
SwAV.

Implementation of Unsupervised Learning of Visual Features by Contrasting Cluster Assignments. The queue is built in *core/hooks/swav_hook.py*.

Parameters

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors. Defaults to None.
- **head** (*dict*) – Config dict for module of loss functions. Defaults to None.

extract_feat(*img*)
Function to extract features from backbone.

Parameters **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns Backbone outputs.

Return type tuple[Tensor]

forward_train(*img*, ***kwargs*)
Forward computation during training.

Parameters **img** (*list[Tensor]*) – A list of input images with shape (N, C, H, W). Typically these should be mean centered and std scaled.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

37.2 backbones

class mmselfsup.models.backbones.**CAEViT**(*arch: str = 'b'*, *img_size: int = 224*, *patch_size: int = 16*, *out_indices: int = - 1*, *drop_rate: float = 0*, *drop_path_rate: float = 0*, *qkv_bias: bool = True*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *final_norm: bool = True*, *output_cls_token: bool = True*, *interpolate_mode: str = 'bicubic'*, *init_values: Optional[float] = None*, *patch_cfg: dict = {}*, *layer_cfgs: dict = {}*, *init_cfg: Optional[dict] = None*)
Vision Transformer for CAE pre-training.

Rewritten version of: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Parameters

- **arch** (*str | dict*) – Vision Transformer architecture. Default: 'b'
- **img_size** (*int | tuple*) – Input image size
- **patch_size** (*int | tuple*) – The patch size
- **out_indices** (*Sequence | int*) – Output from which stages. Defaults to -1, means the last stage.
- **drop_rate** (*float*) – Probability of an element to be zeroed. Defaults to 0.
- **drop_path_rate** (*float*) – stochastic depth rate. Defaults to 0.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **final_norm** (*bool*) – Whether to add a additional layer to normalize final feature map. Defaults to True.
- **output_cls_token** (*bool*) – Whether output the cls_token. If set True, *with_cls_token* must be True. Defaults to True.
- **interpolate_mode** (*str*) – Select the interpolate mode for position embeding vector resize. Defaults to "bicubic".
- **init_values** (*float, optional*) – The init value of gamma in TransformerEncoderLayer.
- **patch_cfg** (*dict*) – Configs of patch embeding. Defaults to an empty dict.
- **layer_cfgs** (*Sequence | dict*) – Configs of each transformer layer in encoder. Defaults to an empty dict.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*img: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Forward computation.

Parameters **x** (*tensor | tuple[tensor]*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.

init_weights() → None
Initialize the weights.

class mmselfsup.models.backbones.**MAEViT**(*arch='b'*, *img_size=224*, *patch_size=16*, *out_indices=- 1*, *drop_rate=0*, *drop_path_rate=0*, *norm_cfg={'eps': 1e-06, 'type': 'LN'}*, *final_norm=True*, *output_cls_token=True*, *interpolate_mode='bicubic'*, *patch_cfg={}*, *layer_cfgs={}*, *mask_ratio=0.75*, *init_cfg=None*)
Vision Transformer for MAE pre-training.

A PyTorch implement of: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Parameters

- **arch** (*str | dict*) – Vision Transformer architecture Default: 'b'
- **img_size** (*int | tuple*) – Input image size
- **patch_size** (*int | tuple*) – The patch size
- **out_indices** (*Sequence | int*) – Output from which stages. Defaults to -1, means the last stage.
- **drop_rate** (*float*) – Probability of an element to be zeroed. Defaults to 0.
- **drop_path_rate** (*float*) – stochastic depth rate. Defaults to 0.

- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **final_norm** (*bool*) – Whether to add a additional layer to normalize final feature map. Defaults to True.
- **output_cls_token** (*bool*) – Whether output the cls_token. If set True, *with_cls_token* must be True. Defaults to True.
- **interpolate_mode** (*str*) – Select the interpolate mode for position embeding vector resize. Defaults to "bicubic".
- **patch_cfg** (*dict*) – Configs of patch embeding. Defaults to an empty dict.
- **layer_cfgs** (*Sequence | dict*) – Configs of each transformer layer in encoder. Defaults to an empty dict.
- **mask_ratio** (*bool*) – The ratio of total number of patches to be masked. Defaults to 0.75.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*x*)

Forward computation.

Parameters **x** (*tensor | tuple[tensor]*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.

init_weights()

Initialize the weights.

random_masking(*x*, *mask_ratio=0.75*)

Generate the mask for MAE Pre-training.

Parameters

- **x** (*torch.tensor*) – Image with data augmentation applied.
- **mask_ratio** (*float*) – The mask ratio of total patches. Defaults to 0.75.

Returns

masked image, mask and the ids to restore original image.

- x_masked (Tensor): masked image.
- mask (Tensor): mask used to mask image.
- ids_restore (Tensor): ids to restore original image.

Return type tuple[Tensor, Tensor, Tensor]

class mmselfsup.models.backbones.**MIMVisionTransformer**(*arch='b'*, *img_size=224*, *patch_size=16*, *out_indices=- 1*, *use_window=False*, *drop_rate=0*, *drop_path_rate=0*, *qkv_bias=True*, *norm_cfg={'eps': 1e-06, 'type': 'LN'}*, *final_norm=True*, *output_cls_token=True*, *interpolate_mode='bicubic'*, *init_values=0.0*, *patch_cfg={}*, *layer_cfgs={}*, *finetune=True*, *init_cfg=None*)

Vision Transformer for MIM-style model (Mask Image Modeling) classification (fine-tuning or linear probe).

A PyTorch implement of : An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Parameters

- **arch** (*str | dict*) – Vision Transformer architecture Default: 'b'
- **img_size** (*int | tuple*) – Input image size
- **patch_size** (*int | tuple*) – The patch size
- **out_indices** (*Sequence | int*) – Output from which stages. Defaults to -1, means the last stage.
- **drop_rate** (*float*) – Probability of an element to be zeroed. Defaults to 0.
- **drop_path_rate** (*float*) – stochastic depth rate. Defaults to 0.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **final_norm** (*bool*) – Whether to add a additional layer to normalize final feature map. Defaults to True.
- **output_cls_token** (*bool*) – Whether output the cls_token. If set True, *with_cls_token* must be True. Defaults to True.
- **interpolate_mode** (*str*) – Select the interpolate mode for position embeding vector resize. Defaults to "bicubic".
- **patch_cfg** (*dict*) – Configs of patch embeding. Defaults to an empty dict.
- **layer_cfgs** (*Sequence | dict*) – Configs of each transformer layer in encoder. Defaults to an empty dict.
- **finetune** (*bool*) – Whether or not do fine-tuning. Defaults to True.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*x*)

Forward computation.

Parameters **x** (*tensor | tuple[tensor]*) – x could be a Torch.tensor or a tuple of Torch.tensor, containing input data for forward computation.

train(*mode=True*)

Set module status before forward computation.

Parameters **mode** (*bool*) – Whether it is train_mode or test_mode

class mmselfsup.models.backbones.**MaskFeatViT**(*arch: Union[str, dict] = 'b'*, *img_size: Union[Tuple[int, int], int] = 224*, *patch_size: int = 16*, *out_indices: int = -1*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *final_norm: bool = True*, *output_cls_token: bool = True*, *interpolate_mode: str = 'bicubic'*, *patch_cfg: dict = {}*, *layer_cfgs: dict = {}*, *init_cfg: Optional[dict] = None*)

Vision Transformer for MaskFeat pre-training.

A PyTorch implement of: Masked Feature Prediction for Self-Supervised Visual Pre-Training. :param arch: Vision Transformer architecture

Default: 'b'

Parameters

- **img_size** (*int | tuple*) – Input image size
- **patch_size** (*int | tuple*) – The patch size

- **out_indices** (*Sequence | int*) – Output from which stages. Defaults to -1, means the last stage.
- **drop_rate** (*float*) – Probability of an element to be zeroed. Defaults to 0.
- **drop_path_rate** (*float*) – stochastic depth rate. Defaults to 0.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **final_norm** (*bool*) – Whether to add a additional layer to normalize final feature map. Defaults to True.
- **output_cls_token** (*bool*) – Whether output the cls_token. If set True, *with_cls_token* must be True. Defaults to True.
- **interpolate_mode** (*str*) – Select the interpolate mode for position embeding vector resize. Defaults to "bicubic".
- **patch_cfg** (*dict*) – Configs of patch embeding. Defaults to an empty dict.
- **layer_cfgs** (*Sequence | dict*) – Configs of each transformer layer in encoder. Defaults to an empty dict.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Generate features for masked images.

Parameters

- **x** (*torch.Tensor*) – Input images.
- **mask** (*torch.Tensor*) – Input masks.

Returns Features with cls_tokens.

Return type torch.Tensor

init_weights() → None
Initialize the weights.

class mmselfsup.models.backbones.**ResNeXt**(*depth*, *groups=32*, *width_per_group=4*, ***kwargs*)
ResNeXt backbone.

Please refer to the paper for details.

As the behavior of forward function in MMSelfSup is different from MMCls, we register our own ResNeXt, inheriting from *mmselfsup.model.backbone.ResNet*.

Parameters

- **depth** (*int*) – Network depth, from {50, 101, 152}.
- **groups** (*int*) – Groups of conv2 in Bottleneck. Defaults to 32.
- **width_per_group** (*int*) – Width per group of conv2 in Bottleneck. Defaults to 4.
- **in_channels** (*int*) – Number of input image channels. Defaults to 3.
- **stem_channels** (*int*) – Output channels of the stem layer. Defaults to 64.
- **num_stages** (*int*) – Stages of the network. Defaults to 4.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage. Defaults to `(1, 2, 2, 2)`.
- **dilations** (*Sequence[int]*) – Dilation of each stage. Defaults to `(1, 1, 1, 1)`.

- **out_indices** (*Sequence[int]*) – Output from which stages. If only one stage is specified, a single tensor (feature map) is returned, otherwise multiple stages are specified, a tuple of tensors will be returned. Defaults to `(3, )`.
- **style** (*str*) – *pytorch* or *caffe*. If set to "pytorch", the stride-two layer is the 3x3 conv layer, otherwise the stride-two layer is the first 1x1 conv layer.
- **deep_stem** (*bool*) – Replace 7x7 conv in input stem with 3 3x3 conv. Defaults to False.
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck. Defaults to False.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **conv_cfg** (*dict | None*) – The config dict for conv layers. Defaults to None.
- **norm_cfg** (*dict*) – The config dict for norm layers.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Defaults to False.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity. Defaults to False.

Example

```
>>> from mmselfsup.models import ResNeXt
>>> import torch
>>> self = ResNeXt(depth=50)
>>> self.eval()
>>> inputs = torch.rand(1, 3, 32, 32)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
(1, 256, 8, 8)
(1, 512, 4, 4)
(1, 1024, 2, 2)
(1, 2048, 1, 1)
```

make_res_layer(***kwargs*)

Redefine the function for ResNeXt related args.

class mmselfsup.models.backbones.**ResNet**(*depth*, *in_channels=3*, *stem_channels=64*, *base_channels=64*, *expansion=None*, *num_stages=4*, *strides=(1, 2, 2, 2)*, *dilations=(1, 1, 1, 1)*, *out_indices=(4)*, *style='pytorch'*, *deep_stem=False*, *avg_down=False*, *frozen_stages=- 1*, *conv_cfg=None*, *norm_cfg={'requires_grad': True, 'type': 'BN'}*, *norm_eval=False*, *with_cp=False*, *zero_init_residual=False*, *init_cfg=[{'type': 'Kaiming', 'layer': ['Conv2d']}, {'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*, *drop_path_rate=0.0*, ***kwargs*)

ResNet backbone.

Please refer to the paper for details.

Parameters

- **depth** (*int*) – Network depth, from {18, 34, 50, 101, 152}.
- **in_channels** (*int*) – Number of input image channels. Defaults to 3.
- **stem_channels** (*int*) – Output channels of the stem layer. Defaults to 64.
- **base_channels** (*int*) – Middle channels of the first stage. Defaults to 64.
- **num_stages** (*int*) – Stages of the network. Defaults to 4.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage. Defaults to `(1, 2, 2, 2)`.
- **dilations** (*Sequence[int]*) – Dilation of each stage. Defaults to `(1, 1, 1, 1)`.
- **out_indices** (*Sequence[int]*) – Output from which stages. Defaults to `(4, )`.
- **style** (*str*) – *pytorch* or *caffe*. If set to "pytorch", the stride-two layer is the 3x3 conv layer, otherwise the stride-two layer is the first 1x1 conv layer.
- **deep_stem** (*bool*) – Replace 7x7 conv in input stem with 3 3x3 conv. Defaults to False.
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck. Defaults to False.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **conv_cfg** (*dict | None*) – The config dict for conv layers. Defaults to None.
- **norm_cfg** (*dict*) – The config dict for norm layers.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Defaults to False.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity. Defaults to False.
- **of the path to be zeroed. Defaults to 0.1** (*Probability*) –

Example

```
>>> from mmselfsup.models import ResNet
>>> import torch
>>> self = ResNet(depth=18)
>>> self.eval()
>>> inputs = torch.rand(1, 3, 32, 32)
>>> level_outputs = self.forward(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
(1, 64, 8, 8)
(1, 128, 4, 4)
(1, 256, 2, 2)
(1, 512, 1, 1)
```

forward(*x*)
Forward function.

As the behavior of forward function in MMSelfSup is different from MMCls, we rewrite the forward function. MMCls does not output the feature map from the 'stem' layer, which we will use for downstream evaluation.

class mmselfsup.models.backbones.**ResNetV1d**(***kwargs*)
ResNetV1d variant described in Bag of Tricks.

Compared with default ResNet(ResNetV1b), ResNetV1d replaces the 7x7 conv in the input stem with three 3x3 convs. And in the downsampling block, a 2x2 avg_pool with stride 2 is added before conv, whose stride is changed to 1.

class mmselfsup.models.backbones.**SimMIMSwinTransformer**(*arch: Union[str, dict] = 'T', img_size: Union[Tuple[int, int], int] = 224, in_channels: int = 3, drop_rate: float = 0.0, drop_path_rate: float = 0.1, out_indices: tuple = (3), use_abs_pos_embed: bool = False, with_cp: bool = False, frozen_stages: bool = - 1, norm_eval: bool = False, norm_cfg: dict = {'type': 'LN'}, stage_cfgs: Union[Sequence, dict] = {}, patch_cfg: dict = {}, init_cfg: Optional[dict] = None*)
Swin Transformer for SimMIM.

Parameters

- **Args** –
- **arch** (*str | dict*) – Swin Transformer architecture Defaults to 'T'.
- **img_size** (*int | tuple*) – The size of input image. Defaults to 224.
- **in_channels** (*int*) – The num of input channels. Defaults to 3.
- **drop_rate** (*float*) – Dropout rate after embedding. Defaults to 0.
- **drop_path_rate** (*float*) – Stochastic depth rate. Defaults to 0.1.
- **out_indices** (*tuple*) – Layers to be outputted. Defaults to (3,).
- **use_abs_pos_embed** (*bool*) – If True, add absolute position embedding to the patch embedding. Defaults to False.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Defaults to False.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **norm_cfg** (*dict*) – Config dict for normalization layer at end of backone. Defaults to dict(type='LN')
- **stage_cfgs** (*Sequence | dict*) – Extra config dict for each stage. Defaults to empty dict.
- **patch_cfg** (*dict*) – Extra config dict for patch embedding. Defaults to empty dict.
- **init_cfg** (*dict, optional*) – The Config for initialization. Defaults to None.

forward(*x: torch.Tensor*, *mask: torch.Tensor*) → Sequence[torch.Tensor]
Generate features for masked images.

This function generates mask images and get the hidden features for them.

Parameters

- **x** (*torch.Tensor*) – Input images.
- **mask** (*torch.Tensor*) – Masks used to construct masked images.

Returns A tuple containing features from multi-stages.

Return type tuple

init_weights() → None
Initialize weights.

class mmselfsup.models.backbones.**VisionTransformer**(*stop_grad_conv1=False*, *frozen_stages=- 1*, *norm_eval=False*, *init_cfg=None*, ***kwargs*)
Vision Transformer.

A pytorch implement of: An Images is Worth 16x16 Words: Transformers for Image Recognition at Scale.

Part of the code is modified from: https://github.com/facebookresearch/moco-v3/blob/main/vits.py.

Parameters

- **stop_grad_conv1** (*bool, optional*) – whether to stop the gradient of convolution layer in *PatchEmbed*. Defaults to False.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

init_weights()
Initialize the weights.

train(*mode=True*)
Set module status before forward computation.

Parameters **mode** (*bool*) – Whether it is train_mode or test_mode

37.3 heads

class mmselfsup.models.heads.**CAEHead**(*tokenizer_path: str*, *lambd: float*, *init_cfg: Optional[dict] = None*)
Pretrain Head for CAE.

Compute the align loss and the main loss. In addition, this head also generates the prediction target generated by dalle.

Parameters

- **tokenizer_path** (*str*) – The path of the tokenizer.
- **lambd** (*float*) – The weight for the align loss.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*img_target: torch.Tensor*, *outputs: torch.Tensor*, *latent_pred: torch.Tensor*, *latent_target: torch.Tensor*, *mask: torch.Tensor*) → dict
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.heads.**ClsHead**(*with_avg_pool=False*, *in_channels=2048*, *num_classes=1000*, *vit_backbone=False*, *init_cfg=[{'type': 'Normal', 'std': 0.01, 'layer': 'Linear'}, {'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)
Simplest classifier head, with only one fc layer.

Parameters

- **with_avg_pool** (*bool*) – Whether to apply the average pooling after neck. Defaults to False.
- **in_channels** (*int*) – Number of input channels. Defaults to 2048.
- **num_classes** (*int*) – Number of classes. Defaults to 1000.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*x*)
Forward head.

Parameters **x** (*list[Tensor] | tuple[Tensor]*) – Feature maps of backbone, each tensor has shape (N, C, H, W).

Returns A list of class scores.

Return type list[Tensor]

loss(*cls_score*, *labels*)
Compute the loss.

class mmselfsup.models.heads.**ContrastiveHead**(*temperature=0.1*)
Head for contrastive learning.

The contrastive loss is implemented in this head and is used in SimCLR, MoCo, DenseCL, etc.

Parameters **temperature** (*float*) – The temperature hyper-parameter that controls the concentration level of the distribution. Defaults to 0.1.

forward(*pos*, *neg*)
Forward function to compute contrastive loss.

Parameters

- **pos** (*Tensor*) – Nx1 positive similarity.
- **neg** (*Tensor*) – Nxk negative similarity.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.heads.**LatentClsHead**(*in_channels: int*, *num_classes: int*, *init_cfg: dict = {'layer': 'Linear', 'std': 0.01, 'type': 'Normal'}*)

Head for latent feature classification.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **num_classes** (*int*) – Number of classes.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*input: torch.Tensor*, *target: torch.Tensor*) → dict

Forward head.

Parameters

- **input** (*Tensor*) – NxC input features.
- **target** (*Tensor*) – NxC target features.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.heads.**LatentCrossCorrelationHead**(*in_channels: int*, *lambd: float = 0.0051*)

Head for latent feature cross correlation. Part of the code is borrowed from: **`https://github.com/facebookresearch/barlowtwins/blob/main/main.py>`_**.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **lambd** (*float*) – Weight on off-diagonal terms. Defaults to 0.0051.

forward(*input: torch.Tensor*, *target: torch.Tensor*) → dict

Forward head.

Parameters

- **input** (*Tensor*) – NxC input features.
- **target** (*Tensor*) – NxC target features.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

off_diagonal(*x: torch.Tensor*) → torch.Tensor

Rreturn a flattened view of the off-diagonal elements of a square matrix.

class mmselfsup.models.heads.**LatentPredictHead**(*predictor: dict*)

Head for latent feature prediction.

This head builds a predictor, which can be any registered neck component. For example, BYOL and SimSiam call this head and build NonLinearNeck. It also implements similarity loss between two forward features.

Parameters **predictor** (*dict*) – Config dict for the predictor.

forward(*input: torch.Tensor*, *target: torch.Tensor*) → dict

Forward head.

Parameters

- **input** (*Tensor*) – NxC input features.
- **target** (*Tensor*) – NxC target features.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.heads.**MAEFinetuneHead**(*embed_dim*, *num_classes=1000*, *label_smooth_val=0.1*)
Fine-tuning head for MAE.

Parameters

- **embed_dim** (*int*) – The dim of the feature before the classifier head.
- **num_classes** (*int*) – The total classes. Defaults to 1000.

forward(*x*)
"Get the logits.

init_weights()
Initialize the weights.

loss(*outputs*, *labels*)
Compute the loss.

class mmselfsup.models.heads.**MAELinprobeHead**(*embed_dim*, *num_classes=1000*)
Linear probing head for MAE.

Parameters

- **embed_dim** (*int*) – The dim of the feature before the classifier head.
- **num_classes** (*int*) – The total classes. Defaults to 1000.

forward(*x*)
"Get the logits.

init_weights()
Initialize the weights.

loss(*outputs*, *labels*)
Compute the loss.

class mmselfsup.models.heads.**MAEPretrainHead**(*norm_pix: bool = False*, *patch_size: int = 16*)
Pre-training head for MAE.

Parameters

- **norm_pix_loss** (*bool*) – Whether or not normalize target. Defaults to False.
- **patch_size** (*int*) – Patch size. Defaults to 16.

forward(*x: torch.Tensor*, *pred: torch.Tensor*, *mask: torch.Tensor*) → dict
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

patchify(*imgs: torch.Tensor*) → torch.Tensor

Parameters **imgs** (*torch.Tensor*) – The shape is (N, 3, H, W)

Returns The shape is (N, L, patch_size**2 *3)

Return type x (torch.Tensor)

unpatchify(*x: torch.Tensor*) → torch.Tensor

Parameters **x** (*torch.Tensor*) – The shape is (N, L, patch_size**2 *3)

Returns The shape is (N, 3, H, W)

Return type imgs (torch.Tensor)

class mmselfsup.models.heads.**MaskFeatFinetuneHead**(*embed_dim: int*, *num_classes: int = 1000*, *label_smooth_val: float = 0.1*)

Fine-tuning head for MaskFeat.

Parameters

- **embed_dim** (*int*) – The dim of the feature before the classifier head.
- **num_classes** (*int*) – The total classes. Defaults to 1000.
- **label_smooth_val** (*float*) – The degree of label smoothing. Defaults to 0.1.

forward(*x: torch.Tensor*) → list
"Get the logits.

init_weights() → None
Initialize the weights.

loss(*outputs: torch.Tensor*, *labels: torch.Tensor*) → dict
Compute the loss.

class mmselfsup.models.heads.**MaskFeatPretrainHead**(*embed_dim: int = 768*, *hog_dim: int = 108*)

Pre-training head for MaskFeat.

Parameters

- **embed_dim** (*int*) – The dim of the feature before the classifier head. Defaults to 768.
- **hog_dim** (*int*) – The dim of the hog feature. Defaults to 108.

forward(*latent: torch.Tensor*, *hog: torch.Tensor*, *mask: torch.Tensor*) → dict
Pre-training head for MaskFeat.

Parameters

- **latent** (*torch.Tensor*) – Input latent of shape (N, 1+L, C).
- **hog** (*torch.Tensor*) – Input hog feature of shape (N, L, C).
- **mask** (*torch.Tensor*) – Input mask of shape (N, H, W).

Returns A dictionary of loss components.

Return type Dict[str, torch.Tensor]

init_weights() → None
Initialize the weights.

loss(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → dict
Compute the loss.

Parameters

- **pred** (*torch.Tensor*) – Input prediction of shape (N, L, C).
- **target** (*torch.Tensor*) – Input target of shape (N, L, C).

- **mask** (*torch.Tensor*) – Input mask of shape (N, L, 1).

Returns A dictionary of loss components.

Return type dict[str, torch.Tensor]

class mmselfsup.models.heads.**MoCoV3Head**(*predictor*, *temperature=1.0*)

Head for MoCo v3 algorithms.

This head builds a predictor, which can be any registered neck component. It also implements latent contrastive loss between two forward features. Part of the code is modified from: https://github.com/facebookresearch/moco-v3/blob/main/moco/builder.py.

Parameters

- **predictor** (*dict*) – Config dict for module of predictor.
- **temperature** (*float*) – The temperature hyper-parameter that controls the concentration level of the distribution. Defaults to 1.0.

forward(*base_out*, *momentum_out*)

Forward head.

Parameters

- **base_out** (*Tensor*) – NxC features from base_encoder.
- **momentum_out** (*Tensor*) – NxC features from momentum_encoder.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmselfsup.models.heads.**MultiClsHead**(*pool_type='adaptive'*, *in_indices=(0)*, *with_last_layer_unpool=False*, *backbone='resnet50'*, *norm_cfg={'type': 'BN'}*, *num_classes=1000*, *init_cfg=[{'type': 'Normal', 'std': 0.01, 'layer': 'Linear'}, {'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)

Multiple classifier heads.

This head inputs feature maps from different stages of backbone, average pools each feature map to around 9000 dimensions, and then appends a linear classifier at each stage to predict corresponding class scores.

Parameters

- **pool_type** (*str*) – 'adaptive' or 'specified'. If set to 'adaptive', use adaptive average pooling, otherwise use specified pooling params.
- **in_indices** (*Sequence[int]*) – Input from which stages.
- **with_last_layer_unpool** (*bool*) – Whether to unpool the features from last layer. Defaults to False.
- **backbone** (*str*) – Specify which backbone to use. Defaults to 'resnet50'.
- **norm_cfg** (*dict*) – dictionary to construct and config norm layer.
- **num_classes** (*int*) – Number of classes. Defaults to 1000.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*x*)

Forward head.

Parameters **x** (*list[Tensor] | tuple[Tensor]*) – Feature maps of backbone, each tensor has shape (N, C, H, W).

Returns A list of class scores.

Return type list[Tensor]

loss(*cls_score*, *labels*)
Compute the loss.

class mmselfsup.models.heads.**SimMIMHead**(*patch_size: int*, *encoder_in_channels: int*)
Pretrain Head for SimMIM.

Parameters

- **patch_size** (*int*) – Patch size of each token.
- **encoder_in_channels** (*int*) – Number of input channels for encoder.

forward(*x: torch.Tensor*, *x_rec: torch.Tensor*, *mask: torch.Tensor*) → dict
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.heads.**SwAVHead**(*feat_dim*, *sinkhorn_iterations=3*, *epsilon=0.05*, *temperature=0.1*, *crops_for_assign=[0, 1]*, *num_crops=[2]*, *num_prototypes=3000*, *init_cfg=None*)
The head for SwAV.

This head contains clustering and sinkhorn algorithms to compute Q codes. Part of the code is borrowed from: `<https://github.com/facebookresearch/swav`_. The queue is built in *core/hooks/swav_hook.py*.

Parameters

- **feat_dim** (*int*) – feature dimension of the prototypes.
- **sinkhorn_iterations** (*int*) – number of iterations in Sinkhorn-Knopp algorithm. Defaults to 3.
- **epsilon** (*float*) – regularization parameter for Sinkhorn-Knopp algorithm. Defaults to 0.05.
- **temperature** (*float*) – temperature parameter in training loss. Defaults to 0.1.
- **crops_for_assign** (*list[int]*) – list of crops id used for computing assignments. Defaults to [0, 1].
- **num_crops** (*list[int]*) – list of number of crops. Defaults to [2].
- **num_prototypes** (*int*) – number of prototypes. Defaults to 3000.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*)
Forward head of swav to compute the loss.

Parameters **x** (*Tensor*) – NxC input features.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

37.4 memories

class mmselfsup.models.memories.**InterCLRMemory**(*length*, *feat_dim*, *momentum*, *num_classes*, *min_cluster*, ***kwargs*)

Memory bank for InterCLR.

Parameters

- **length** (*int*) – Number of features stored in the memory bank.
- **feat_dim** (*int*) – Dimension of stored features.
- **momentum** (*float*) – Momentum coefficient for updating features.
- **num_classes** (*int*) – Number of clusters.
- **min_cluster** (*int*) – Minimal cluster size.

assign_label(*label*)
Assign offline labels for each epoch.

deal_with_small_clusters()
Deal with small clusters.

update_centroids_memory(*cinds=None*)
Update centroids in the memory bank.

update_samples_memory(*ind*, *feature*)
Update features and labels in the memory bank.

update_simple_memory(*ind*, *feature*)
Update features in the memory bank.

class mmselfsup.models.memories.**ODCMemory**(*length*, *feat_dim*, *momentum*, *num_classes*, *min_cluster*, ***kwargs*)

Memory module for ODC.

This module includes the samples memory and the centroids memory in ODC. The samples memory stores features and pseudo-labels of all samples in the dataset; while the centroids memory stores features of cluster centroids.

Parameters

- **length** (*int*) – Number of features stored in samples memory.
- **feat_dim** (*int*) – Dimension of stored features.
- **momentum** (*float*) – Momentum coefficient for updating features.
- **num_classes** (*int*) – Number of clusters.
- **min_cluster** (*int*) – Minimal cluster size.

deal_with_small_clusters()
Deal with small clusters.

init_memory(*feature*, *label*)
Initialize memory modules.

update_centroids_memory(*cinds=None*)
Update centroids memory.

update_samples_memory(*ind*, *feature*)
Update samples memory.

class mmselfsup.models.memories.**SimpleMemory**(*length*, *feat_dim*, *momentum*, ***kwargs*)
Simple memory bank (e.g., for NPID).

This module includes the memory bank that stores running average features of all samples in the dataset.

Parameters

- **length** (*int*) – Number of features stored in the memory bank.
- **feat_dim** (*int*) – Dimension of stored features.
- **momentum** (*float*) – Momentum coefficient for updating features.

update(*ind*, *feature*)
Update features in memory bank.

Parameters

- **ind** (*Tensor*) – Indices for the batch of features.
- **feature** (*Tensor*) – Batch of features.

37.5 necks

class mmselfsup.models.necks.**AvgPool2dNeck**(*output_size=1*)
The average pooling 2d neck.

forward(*x*)
Forward function.

class mmselfsup.models.necks.**CAENeck**(*patch_size: int = 16*, *num_classes: int = 8192*, *embed_dims: int = 768*, *regressor_depth: int = 6*, *decoder_depth: int = 8*, *num_heads: int = 12*, *mlp_ratio: int = 4*, *qkv_bias: bool = True*, *qk_scale: Optional[float] = None*, *drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *init_values: Optional[float] = None*, *mask_tokens_num: int = 75*, *init_cfg: Optional[dict] = None*)
Neck for CAE Pre-training.

This module construct the latent prediction regressor and the decoder for the latent prediction and final prediction.

Parameters

- **patch_size** (*int*) – The patch size of each token. Defaults to 16.
- **num_classes** (*int*) – The number of classes for final prediction. Defaults to 8192.
- **embed_dims** (*int*) – The embed dims of latent feature in regressor and decoder. Defaults to 768.
- **regressor_depth** (*int*) – The number of regressor blocks. Defaults to 6.
- **decoder_depth** (*int*) – The number of decoder blocks. Defaults to 8.
- **num_heads** (*int*) – The number of head in multi-head attention. Defaults to 12.
- **mlp_ratio** (*int*) – The expand ratio of latent features in MLP. defaults to 4.
- **qkv_bias** (*bool*) – Whether or not to use qkv bias. Defaults to True.
- **qk_scale** (*float, optional*) – The scale applied to the results of qk. Defaults to None.

- **drop_rate** (*float*) – The dropout rate. Defaults to 0.
- **attn_drop_rate** (*float*) – The dropout rate in attention block. Defaults to 0.
- **norm_cfg** (*dict*) – The config of normalization layer. Defaults to dict(type='LN', eps=1e-6).
- **init_values** (*float, optional*) – The init value of gamma. Defaults to None.
- **mask_tokens_num** (*int*) – The number of mask tokens. Defaults to 75.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*x_unmasked: torch.Tensor*, *pos_embed_masked: torch.Tensor*, *pos_embed_unmasked: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Get the latent prediction and final prediction.

Parameters

- **x_unmasked** (*torch.Tensor*) – Features of unmasked tokens.
- **pos_embed_masked** (*torch.Tensor*) – Position embedding of masked tokens.
- **pos_embed_unmasked** (*torch.Tensor*) – Position embedding of unmasked tokens.

Returns

Final prediction and latent prediction.

Return type Tuple[torch.Tensor, torch.Tensor]

init_weights() → None

Initialize the weights.

class mmselfsup.models.necks.**DenseCLNeck**(*in_channels*, *hid_channels*, *out_channels*, *num_grid=None*, *init_cfg=None*)

The non-linear neck of DenseCL.

Single and dense neck in parallel: fc-relu-fc, conv-relu-conv. Borrowed from the authors' code: **`<https://github.com/WXinlong/DenseCL`_**.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **hid_channels** (*int*) – Number of hidden channels.
- **out_channels** (*int*) – Number of output channels.
- **num_grid** (*int*) – The grid size of dense features. Defaults to None.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*)

Forward function of neck.

Parameters **x** (*list[tensor]*) – feature map of backbone.

class mmselfsup.models.necks.**LinearNeck**(*in_channels*, *out_channels*, *with_avg_pool=True*, *init_cfg=None*)

The linear neck: fc only.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.

- **with_avg_pool** (*bool*) – Whether to apply the global average pooling after backbone. Defaults to True.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.necks.**MAEPretrainDecoder**(*num_patches=196*, *patch_size=16*, *in_chans=3*, *embed_dim=1024*, *decoder_embed_dim=512*, *decoder_depth=8*, *decoder_num_heads=16*, *mlp_ratio=4.0*, *norm_cfg={'eps': 1e-06, 'type': 'LN'}*)

Decoder for MAE Pre-training.

Parameters

- **num_patches** (*int*) – The number of total patches. Defaults to 196.
- **patch_size** (*int*) – Image patch size. Defaults to 16.
- **in_chans** (*int*) – The channel of input image. Defaults to 3.
- **embed_dim** (*int*) – Encoder's embedding dimension. Defaults to 1024.
- **decoder_embed_dim** (*int*) – Decoder's embedding dimension. Defaults to 512.
- **decoder_depth** (*int*) – The depth of decoder. Defaults to 8.
- **decoder_num_heads** (*int*) – Number of attention heads of decoder. Defaults to 16.
- **mlp_ratio** (*int*) – Ratio of mlp hidden dim to decoder's embedding dim. Defaults to 4.
- **norm_cfg** (*dict*) – Normalization layer. Defaults to LayerNorm.

Some of the code is borrowed from *https://github.com/facebookresearch/mae*.

Example

```
>>> from mmselfsup.models import MAEPretrainDecoder
>>> import torch
>>> self = MAEPretrainDecoder()
>>> self.eval()
>>> inputs = torch.rand(1, 50, 1024)
>>> ids_restore = torch.arange(0, 196).unsqueeze(0)
>>> level_outputs = self.forward(inputs, ids_restore)
>>> print(tuple(level_outputs.shape))
(1, 196, 768)
```

forward(*x*, *ids_restore*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

init_weights()
Initialize the weights.

class mmselfsup.models.necks.**MoCoV2Neck**(*in_channels*, *hid_channels*, *out_channels*, *with_avg_pool=True*, *init_cfg=None*)
The non-linear neck of MoCo v2: fc-relu-fc.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **hid_channels** (*int*) – Number of hidden channels.
- **out_channels** (*int*) – Number of output channels.
- **with_avg_pool** (*bool*) – Whether to apply the global average pooling after backbone. Defaults to True.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Defaults to None.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmselfsup.models.necks.**NonLinearNeck**(*in_channels*, *hid_channels*, *out_channels*, *num_layers=2*, *with_bias=False*, *with_last_bn=True*, *with_last_bn_affine=True*, *with_last_bias=False*, *with_avg_pool=True*, *vit_backbone=False*, *norm_cfg={'type': 'SyncBN'}*, *init_cfg=[{'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)
The non-linear neck.

Structure: fc-bn-[relu-fc-bn] where the substructure in [] can be repeated. For the default setting, the repeated time is 1. The neck can be used in many algorithms, e.g., SimCLR, BYOL, SimSiam.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **hid_channels** (*int*) – Number of hidden channels.
- **out_channels** (*int*) – Number of output channels.
- **num_layers** (*int*) – Number of fc layers. Defaults to 2.
- **with_bias** (*bool*) – Whether to use bias in fc layers (except for the last). Defaults to False.
- **with_last_bn** (*bool*) – Whether to add the last BN layer. Defaults to True.

- **with_last_bn_affine** (*bool*) – Whether to have learnable affine parameters in the last BN layer (set False for SimSiam). Defaults to True.
- **with_last_bias** (*bool*) – Whether to use bias in the last fc layer. Defaults to False.
- **with_avg_pool** (*bool*) – Whether to apply the global average pooling after backbone. Defaults to True.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='SyncBN').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.necks.**ODCNeck**(*in_channels*, *hid_channels*, *out_channels*, *with_avg_pool=True*, *norm_cfg={'type': 'SyncBN'}*, *init_cfg=[{'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)

The non-linear neck of ODC: fc-bn-relu-dropout-fc-relu.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **hid_channels** (*int*) – Number of hidden channels.
- **out_channels** (*int*) – Number of output channels.
- **with_avg_pool** (*bool*) – Whether to apply the global average pooling after backbone. Defaults to True.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='SyncBN').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.necks.**RelativeLocNeck**(*in_channels*, *out_channels*, *with_avg_pool=True*, *norm_cfg={'type': 'BN1d'}*, *init_cfg=[{'type': 'Normal', 'std': 0.01, 'layer': 'Linear'}, {'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)

The neck of relative patch location: fc-bn-relu-dropout.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **out_channels** (*int*) – Number of output channels.
- **with_avg_pool** (*bool*) – Whether to apply the global average pooling after backbone. Defaults to True.
- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='BN1d').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.necks.**SimMIMNeck**(*in_channels: int*, *encoder_stride: int*)

Pre-train Neck For SimMIM.

This neck reconstructs the original image from the shrunk feature map.

Parameters

- **in_channels** (*int*) – Channel dimension of the feature map.
- **encoder_stride** (*int*) – The total stride of the encoder.

forward(*x: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.necks.**SwAVNeck**(*in_channels*, *hid_channels*, *out_channels*, *with_avg_pool=True*, *with_l2norm=True*, *norm_cfg={'type': 'SyncBN'}*, *init_cfg=[{'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)

The non-linear neck of SwAV: fc-bn-relu-fc-normalization.

Parameters

- **in_channels** (*int*) – Number of input channels.
- **hid_channels** (*int*) – Number of hidden channels.
- **out_channels** (*int*) – Number of output channels.
- **with_avg_pool** (*bool*) – Whether to apply the global average pooling after backbone. Defaults to True.
- **with_l2norm** (*bool*) – whether to normalize the output after projection. Defaults to True.

- **norm_cfg** (*dict*) – Dictionary to construct and config norm layer. Defaults to dict(type='SyncBN').
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

37.6 utils

class mmselfsup.models.utils.**Accuracy**(*topk=(1)*)

Implementation of accuracy computation.

forward(*pred*, *target*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**CAETransformerRegressorLayer**(*embed_dims: int*, *num_heads: int*, *feedforward_channels: int*, *num_fcs: int = 2*, *qkv_bias: bool = False*, *qk_scale: Optional[float] = None*, *drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *init_values: float = 0.0*, *act_cfg: dict = {'type': 'GELU'}*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*)

Transformer layer for the regressor of CAE.

This module is different from conventional transformer encoder layer, for its queries are the masked tokens, but its keys and values are the concatenation of the masked and unmasked tokens.

Parameters

- **embed_dims** (*int*) – The feature dimension.
- **num_heads** (*int*) – The number of heads in multi-head attention.
- **feedforward_channels** (*int*) – The hidden dimension of FFNs. Defaults: 1024.
- **num_fcs** (*int, optional*) – The number of fully-connected layers in FFNs. Default: 2.
- **qkv_bias** (*bool*) – If True, add a learnable bias to q, k, v. Defaults to True.
- **qk_scale** (*float, optional*) – Override default qk scale of `head_dim ** -0.5` if set. Defaults to None.

- **drop_rate** (*float*) – The dropout rate. Defaults to 0.0.
- **attn_drop_rate** (*float*) – The drop out rate for attention output weights. Defaults to 0.
- **drop_path_rate** (*float*) – Stochastic depth rate. Defaults to 0.
- **init_values** (*float*) – The init values of gamma. Defaults to 0.0.
- **act_cfg** (*dict*) – The activation config for FFNs. Defaluts to `dict(type='GELU')`.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.

forward(*x_q: torch.Tensor*, *x_kv: torch.Tensor*, *pos_q: torch.Tensor*, *pos_k: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**Encoder**(*n_hid: int = 256*, *n_blk_per_group: int = 2*, *input_channels: int = 3*, *vocab_size: int = 8192*, *device: torch.device = device(type='cpu')*, *requires_grad: bool = False*, *use_mixed_precision: bool = True*)

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**ExtractProcess**
Global average-pooled feature extraction process.

This process extracts the global average-pooled features from the last layer of resnet backbone.

extract(*model*, *data_loader*, *distributed=False*)
The extract function to apply forward function and choose distributed or not.

class mmselfsup.models.utils.**GatherLayer**(**args*, ***kwargs*)
Gather tensors from all process, supporting backward propagation.

static backward(*ctx*, **grads*)
Defines a formula for differentiating the operation with backward mode automatic differentiation (alias to the vjp function).

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by as many outputs as the *forward()* returned (None will be passed in for non tensor outputs of the forward function), and it should return as many tensors, as there were inputs to *forward()*. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input. If an input is not a Tensor or is a Tensor not requiring grads, you can just pass None as a gradient for that input.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g.,

backward() will have `ctx.needs_input_grad[0] = True` if the first input to *forward()* needs gradient computated w.r.t. the output.

static forward(*ctx*, *input*)
Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store arbitrary data that can be then retrieved during the backward pass. Tensors should not be stored directly on *ctx* (though this is not currently enforced for backward compatibility). Instead, tensors should be saved either with `ctx.save_for_backward()` if they are intended to be used in `backward` (equivalently, `vjp`) or `ctx.save_for_forward()` if they are intended to be used for in `jvp`.

class mmselfsup.models.utils.**MultiExtractProcess**(*pool_type='specified'*, *backbone='resnet50'*, *layer_indices=(0, 1, 2, 3, 4)*)
Multi-stage intermediate feature extraction process for *extract.py* and *tsne_visualization.py* in tools.

This process extracts feature maps from different stages of backbone, and average pools each feature map to around 9000 dimensions.

Parameters

- **pool_type** (*str*) – Pooling type in *MultiPooling*. Options are "adaptive" and "specified". Defaults to "specified".
- **backbone** (*str*) – Backbone type, now only support "resnet50". Defaults to "resnet50".
- **layer_indices** (*Sequence[int]*) – Output from which stages. 0 for stem, 1, 2, 3, 4 for res layers. Defaults to (0, 1, 2, 3, 4).

extract(*model*, *data_loader*, *distributed=False*)
The extract function to apply forward function and choose distributed or not.

class mmselfsup.models.utils.**MultiPooling**(*pool_type='adaptive'*, *in_indices=(0)*, *backbone='resnet50'*)
Pooling layers for features from multiple depth.

Parameters

- **pool_type** (*str*) – Pooling type for the feature map. Options are 'adaptive' and 'specified'. Defaults to 'adaptive'.
- **in_indices** (*Sequence[int]*) – Output from which backbone stages. Defaults to (0,).
- **backbone** (*str*) – The selected backbone. Defaults to 'resnet50'.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**MultiPrototypes**(*output_dim*, *num_prototypes*)
Multi-prototypes for SwAV head.

Parameters

- **output_dim** (*int*) – The output dim from SwAV neck.

- **num_prototypes** (*list[int]*) – The number of prototypes needed.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**MultiheadAttention**(*embed_dims: int*, *num_heads: int*, *input_dims: Optional[int] = None*, *attn_drop: float = 0.0*, *proj_drop: float = 0.0*, *qkv_bias: bool = True*, *qk_scale: Optional[float] = None*, *proj_bias: bool = True*, *init_cfg: Optional[dict] = None*)

Multi-head Attention Module.

This module rewrite the MultiheadAttention by replacing qkv bias with customized qkv bias, in addition to removing the drop path layer.

Parameters

- **embed_dims** (*int*) – The embedding dimension.
- **num_heads** (*int*) – Parallel attention heads.
- **input_dims** (*int, optional*) – The input dimension, and if None, use `embed_dims`. Defaults to None.
- **attn_drop** (*float*) – Dropout rate of the dropout layer after the attention calculation of query and key. Defaults to 0.
- **proj_drop** (*float*) – Dropout rate of the dropout layer after the output projection. Defaults to 0.
- **dropout_layer** (*dict*) – The dropout config before adding the shortcut. Defaults to `dict(type='Dropout', drop_prob=0.)`.
- **qkv_bias** (*bool*) – If True, add a learnable bias to q, k, v. Defaults to True.
- **qk_scale** (*float, optional*) – Override default qk scale of `head_dim ** -0.5` if set. Defaults to None.
- **proj_bias** (*bool*) – Defaults to True.
- **init_cfg** (*dict, optional*) – The Config for initialization. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**Sobel**
Sobel layer.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmselfsup.models.utils.**TransformerEncoderLayer**(*embed_dims: int*, *num_heads: int*, *feedforward_channels: int*, *window_size: Optional[int] = None*, *drop_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *num_fcs: int = 2*, *qkv_bias: bool = True*, *act_cfg: dict = {'type': 'GELU'}*, *norm_cfg: dict = {'type': 'LN'}*, *init_values: float = 0.0*, *init_cfg: Optional[dict] = None*)

Implements one encoder layer in Vision Transformer.

This module is the rewritten version of the TransformerEncoderLayer in MMClassification by adding the gamma and relative position bias in Attention module.

Parameters

- **embed_dims** (*int*) – The feature dimension.
- **num_heads** (*int*) – Parallel attention heads
- **feedforward_channels** (*int*) – The hidden dimension for FFNs
- **drop_rate** (*float*) – Probability of an element to be zeroed after the feed forward layer. Defaults to 0.
- **attn_drop_rate** (*float*) – The drop out rate for attention output weights. Defaults to 0.
- **drop_path_rate** (*float*) – Stochastic depth rate. Defaults to 0.
- **num_fcs** (*int*) – The number of fully-connected layers for FFNs. Defaults to 2.
- **qkv_bias** (*bool*) – enable bias for qkv if True. Defaults to True.
- **act_cfg** (*dict*) – The activation config for FFNs. Defaluts to `dict(type='GELU')`.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **init_values** (*float*) – The init values of gamma. Defaults to 0.0.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

mmselfsup.models.utils.**accuracy**(*pred*, *target*, *topk=1*)
Compute accuracy of predictions.

Parameters

- **pred** (*Tensor*) – The output of the model.
- **target** (*Tensor*) – The labels of data.
- **topk** (*int | list[int]*) – Top-k metric selection. Defaults to 1.

mmselfsup.models.utils.**build_2d_sincos_position_embedding**(*patches_resolution*, *embed_dims*, *temperature=10000.0*, *cls_token=False*)

The function is to build position embedding for model to obtain the position information of the image patches.

mmselfsup.models.utils.**knn_classifier**(*train_features*, *train_labels*, *test_features*, *test_labels*, *k*, *T*, *num_classes=1000*)

Compute accuracy of knn classifier predictions.

Parameters

- **train_features** (*Tensor*) – Extracted features in the training set.
- **train_labels** (*Tensor*) – Labels in the training set.
- **test_features** (*Tensor*) – Extracted features in the testing set.
- **test_labels** (*Tensor*) – Labels in the testing set.
- **k** (*int*) – Number of NN to use.
- **T** (*float*) – Temperature used in the voting coefficient.
- **num_classes** (*int*) – Number of classes. Defaults to 1000.

CHAPTER

THIRTYEIGHT

MMSELFSUP.UTILS

class mmselfsup.utils.**AliasMethod**(*probs*)

The alias method for sampling.

From: https://hips.seas.harvard.edu/blog/2013/03/03/the-alias-method-efficient-sampling-with-many-discrete-outcomes/

Parameters **probs** (*Tensor*) – Sampling probabilities.

draw(*N*)

Draw N samples from multinomial.

Parameters **N** (*int*) – Number of samples.

Returns Samples.

Return type Tensor

class mmselfsup.utils.**Extractor**(*dataset*, *samples_per_gpu*, *workers_per_gpu*, *dist_mode=False*, *persistent_workers=True*, ***kwargs*)

Feature extractor.

Parameters

- **dataset** (*Dataset | dict*) – A PyTorch dataset or dict that indicates the dataset.
- **samples_per_gpu** (*int*) – Number of images on each GPU, i.e., batch size of each GPU.
- **workers_per_gpu** (*int*) – How many subprocesses to use for data loading for each GPU.
- **dist_mode** (*bool*) – Use distributed extraction or not. Defaults to False.
- **persistent_workers** (*bool*) – If True, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers Dataset instances alive. The argument also has effect in PyTorch>=1.7.0. Defaults to True.

mmselfsup.utils.**batch_shuffle_ddp**(*x*)

Batch shuffle, for making use of BatchNorm.

*** Only support DistributedDataParallel (DDP) model. ***

mmselfsup.utils.**batch_unshuffle_ddp**(*x*, *idx_unshuffle*)

Undo batch shuffle.

*** Only support DistributedDataParallel (DDP) model. ***

mmselfsup.utils.**collect_env**()

Collect the information of the running environments.

mmselfsup.utils.**concat_all_gather**(*tensor*)

Performs all_gather operation on the provided tensors.

*** Warning ***: torch.distributed.all_gather has no gradient.

mmselfsup.utils.**dist_forward_collect**(*func*, *data_loader*, *rank*, *length*, *ret_rank=- 1*)
Forward and collect network outputs in a distributed manner.

This function performs forward propagation and collects outputs. It can be used to collect results, features, losses, etc.

Parameters

- **func** (*function*) – The function to process data. The output must be a dictionary of CPU tensors.
- **data_loader** (*Dataloader*) – the torch Dataloader to yield data.
- **rank** (*int*) – This process id.
- **length** (*int*) – Expected length of output arrays.
- **ret_rank** (*int*) – The process that returns. Other processes will return None.

Returns The concatenated outputs.

Return type results_all (dict(np.ndarray))

mmselfsup.utils.**distributed_sinkhorn**(*out*, *sinkhorn_iterations*, *world_size*, *epsilon*)
Apply the distributed sinknorn optimization on the scores matrix to find the assignments.

mmselfsup.utils.**find_latest_checkpoint**(*path*, *suffix='pth'*)
Find the latest checkpoint from the working directory. :param path: The path to find checkpoints. :type path: str :param suffix: File extension.

Defaults to pth.

Returns File path of the latest checkpoint.

Return type latest_path(str | None)

References

mmselfsup.utils.**gather_tensors**(*input_array*)
Gather tensor from all GPUs.

mmselfsup.utils.**gather_tensors_batch**(*input_array*, *part_size=100*, *ret_rank=- 1*)
batch-wise gathering to avoid CUDA out of memory.

mmselfsup.utils.**get_root_logger**(*log_file=None*, *log_level=20*)
Get root logger.

Parameters

- **log_file** (*str, optional*) – File path of log. Defaults to None.
- **log_level** (*int, optional*) – The level of logger. Defaults to logging.INFO.

Returns The obtained logger.

Return type logging.Logger

mmselfsup.utils.**nondist_forward_collect**(*func*, *data_loader*, *length*)
Forward and collect network outputs.

This function performs forward propagation and collects outputs. It can be used to collect results, features, losses, etc.

Parameters

- **func** (*function*) – The function to process data. The output must be a dictionary of CPU tensors.
- **data_loader** (*Dataloader*) – the torch Dataloader to yield data.
- **length** (*int*) – Expected length of output arrays.

Returns The concatenated outputs.

Return type results_all (dict(np.ndarray))

mmselfsup.utils.**setup_multi_processes**(*cfg*)
Setup multi-processing environment variables.

mmselfsup.utils.**sync_random_seed**(*seed=None*, *device='cuda'*)
Make sure different ranks share the same seed. All workers must call this function, otherwise it will deadlock. This method is generally used in *DistributedSampler*, because the seed should be identical across all processes in the distributed group.

In distributed sampling, different ranks should sample non-overlapped data in the dataset. Therefore, this function is used to make sure that each rank shuffles the data indices in the same order based on the same seed. Then different ranks could use different indices to select non-overlapped data from the same data list.

Parameters

- **seed** (*int, Optional*) – The seed. Default to None.
- **device** (*str*) – The device where the seed will be put on. Default to 'cuda'.

Returns Seed to be used.

Return type int

References

CHAPTER

THIRTYNINE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V