# MMSelfSup

*Release 1.0.0rc6*

**MMSelfSup Authors**

**Apr 06, 2023**

# GET STARTED

CHAPTER

# ONE

# OVERVIEW



In this section, We would like to give a quick review of the open-source library MMSelfSup.

We will first illustrate the basic idea of the self-supervised learning, then we will briefly describe the design of MMSelfSup. After that, we will provide a hands-on roadmap to help the users to play with MMSelfSup

## 1.1 Introduction of Self-supervised Learning

Self-supervised learning(SSL) is a promising learning paradigm, which aims to leverage the potential of the huge amount of unlabeled data. In SSL, we typically use the label generated automatically without human labor, to learn a model to extract the discriminative representation of the data. Equipped with the powerful pre-trained model by SSL, we are able to improve various downstream vision tasks currently.

The community has witnessed rapid development of SSL in the past few years. Our codebase aims to become an easy-to-use and user-friendly library, to help the research and engineering. We will elaborate the properties and design of MMSelfSup in the following sections.

## 1.2 Design of MMSelfSup

MMSelfSup follows the modular designed architecture as other OpenMMLab projects. the overall framework is illustrated below:

- **Datasets** provides the support for various datasets, with many useful augmentation strategy.
- **Algorithms** consists of many milestone SSL works with easy-to-use interface.
- **Tools** includes the training and analysis tools for SSL
- **Benchmarks** introduces many examples of how to use SSL for various downstream tasks(e.g., classification, detection, segmentation and etc.).

## 1.3 Hands-on Roadmap of MMSelfSup

To help the user to use the MMSelfSup quickly, we recommend the following roadmap for using our library.

### 1.3.1 Play with MMSelfSup

Typically, SSL is considered as the pre-training algorithm for various model architectures. Thus, the complete pipeline consists of the **pre-training** stage and the **benchmark** stage.

- For the user who wants to try MMSelfSup with various SSL algorithms. We first refer the user to *Get Started* for the **environment setup**.
- For the pre-training stage, we refer the user to *Pre-train* for using various SSL algorithms to obtain the pre-trained model.
- For the benchmark stage, we refer the user to Benchmark for examples and usage of applying the pre-trained models in many downstream tasks.
- Also, we provide some analysis tools and visualization tools Useful Tools to help diagnose the algorithm.

### 1.3.2 Learn SSL with MMSelfSup

If you are new to SSL, we recommend using the *Model Zoo* as a reference to learn the representative SSL algorithms.

# CHAPTER TWO

# GET STARTED



## 2.1 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 3.7+, CUDA 9.2+ and PyTorch 1.6+.

**Note:** If you are experienced with PyTorch and have already installed it, just skip this part and jump to the next Installation 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
```

## 2.2 Installation

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

### 2.2.1 Best practices

**Step 0.** Install MMEngine and MMCV using MIM.

```
pip install -U openmim
mim install mmengine
mim install 'mmcv>=2.0.0rc1'
```

**Step 1.** Install MMSelfSup.

According to your needs, we support two installation modes:

- *Install from source (Recommended)*: You want to develop your own self-supervised task or new features based on MMSelfSup framework, e.g., adding new datasets or models. And you can use all tools we provided.
- *Install as a Python package*: You just want to call MMSelfSup's APIs or import MMSelfSup's modules in your project.

#### Install from source

In this case, install mmselfsup from source:

```
git clone https://github.com/open-mmlab/mmselfsup.git
cd mmselfsup
git checkout 1.x
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.
```

Optionally, if you want to contribute to MMSelfSup or experience experimental functions, please checkout to the `dev-1.x` branch:

```
git checkout dev-1.x
```

#### Install as a Python package

Just install with pip.

```
pip install 'mmselfsup>=1.0.0rc0'
```

### 2.2.2 Verify the installation

To verify whether MMSelfSup is installed correctly, you can run the following command.

```
import mmselfsup
print(mmselfsup.__version__)
# Example output: 1.0.0rc0 or newer
```

### 2.2.3 Customize installation

#### Benchmark

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

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

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

```
pip install 'mmdet>=3.0.0rc0' 'mmsegmentation>=1.0.0rc0'
```

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

#### 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.

---

### Install MMEngine without MIM

To install MMEngine with pip instead of MIM, please follow MMEngine installation guides.

For example, you can install MMEngine by the following command.

```
pip install mmengine
```

### 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 installs mmcv-full built for PyTorch 1.12.0 and CUDA 11.6.

```
pip install 'mmcv>=2.0.0rc1' -f https://download.openmmlab.com/mmcv/dist/cu116/torch1.12.
↪0/index.html
```

### Install on CPU-only platforms

MMSelfSup can be built for CPU only environment. In CPU mode, you can train, test or inference a model.

Some functionalities are gone in this mode, usually GPU-compiled ops. But don't worry, almost all models in MMSelfSup don't depend on these ops.

### 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 MMEngine and MMCV using MIM.

```
!pip3 install openmim
!mim install mmengine
!mim install 'mmcv>=2.0.0rc1'
```

**Step 1.** Install MMSelfSup from the source.

```
!git clone https://github.com/open-mmlab/mmselfsup.git
%cd mmselfsup
!git checkout 1.x
!pip install -e .
```

**Step 2.** Verification.

```
import mmselfsup
print(mmselfsup.__version__)
# Example output: 1.0.0rc0 or newer
```

**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.

#### Using MMSelfSup with Docker

We provide a Dockerfile to build an image. Ensure that your docker version >=19.03.

```
# build an image with PyTorch 1.10.0, CUDA 11.3, CUDNN 8.
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 command:

```
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.

### 2.2.4 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.

## 2.3 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

THREE

# PRETRAIN

## 3.1 Tutorial 1: Learn about Configs

We incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config. You may also pass `--cfg-options xxx.yyy=zzz` to see the updated config.



### 3.1.1 Config File and Checkpoint Naming Convention

We follow the convention below to name config files. Contributors are advised to follow the same convention. The name of config file is divided into four parts: `algorithm info`, `module information`, `training information` and `data information`. Logically, different parts are connected with underscore `_`, and info belonging to the same part is connected with dash `-`.

The following example is for illustration:

```
{algorithm_info}_{module_info}_{training_info}_{data_info}.py
```

- `algorithm_info`Algorithm information includes the algorithm name, such as `simclr`, `mocov2`, etc.

- `module_info` Module information denotes backbones, necks, heads and losses.
- `training_info`Training information denotes some training schedules, such as batch size, lr schedule, data augmentation, etc.
- `data_info`Data information includes the dataset name, input size, etc.

We detail the naming convention for each part in the name of the config file:

### Algorithm Information

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

`algorithm` generally denotes the abbreviation for the paper and its version. E.g.:

- `relative-loc`
- `simclr`
- `mocov2`

`misc` provides some other algorithm-related information. E.g.:

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

Note that different words are connected with dash `-`.

### Module Information

```
{backbone_setting}-{neck_setting}-{head_setting}-{loss_setting}
```

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

- `resnet50`
- `vit-base-p16`
- `swin-base`

Sometimes, there are some special settings needed to be mentioned in the config name. E.g.:

- `resnet50-sobel`: In some downstream tasks like linear evaluation, when loading the DeepCluster pre-traiend model, the backbone only takes 2-channel images after the Sobel layer as input.

Note that `neck_setting`, `head_setting` and `loss_setting` are optional.

### Training Information

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

- Batch size: the format is `{gpu x batch_per_gpu}`e.g., `8xb32`.
- Training recipes: they will be arranged in the order `{pipeline aug}-{train aug}-{scheduler}-{epochs}`.

E.g.:

- `8xb32-mcrop-2-6-coslr-200e` : `mcrop` is the multi-crop data augmentation proposed in SwAV. 2 and 6 means that two pipelines output 2 and 6 crops, respectively. The crop size is recorded in data information.

- `8xb32-accum16-coslr-200e` : `accum16` means the weights will be updated after the gradient is accumulated for 16 iterations.
- `8xb512-amp-coslr-300e` : `amp` denotes the automatic mixed precision training.

### Data Information

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

- `in1k` : `ImageNet1k` dataset. The input image size is 224x224 by default
- `in1k-384` : `ImageNet1k` dataset with the input image size of 384x384
- `in1k-384x224` : `ImageNet1k` dataset with the input image size of 384x224 (`HxW`)
- `cifar10`
- `inat18` : `iNaturalist2018` dataset. It has 8142 classes.
- `places205`

### Config File Name Example

Here, we give a specific file name to explain the naming convention.

```
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 the multi-crop data augmentation
  - `coslr`: Use the cosine learning rate decay scheduler
  - `200e`: Train the model for 200 epochs
- `in1k-224-96`: Data information. The model is trained on ImageNet1k dataset with the input size of 224x224 (for 2 crops) and 96x96 (for 6 crops).

## 3.1.2 Config File Structure

There are four kinds of basic files in the `configs/_base_`, namely

- models
- datasets
- schedules
- runtime

All these basic files define the basic elements, such as train/val/test loop and optimizer, to run the experiment. You can easily build your own training config file by inheriting some base config files. The configs that are composed by components from `_base_` are called *primitive*.

For easy understanding, we use MoCo v2 as an example and comment the meaning of each line. For more details, 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
]

# only keeps the latest 3 checkpoints
default_hooks = dict(checkpoint=dict(max_keep_ckpts=3))
```

../_base_/models/mocov2.py is the base configuration file for the model of MoCo v2.

```
# model settings
# type='MoCo' specifies we will use the model of MoCo. And we
# split the model into four parts, which are backbone, neck, head
# and loss. 'queue_len', 'feat_dim' and 'momentum' are required
# by MoCo during the training process.
model = dict(
    type='MoCo',
    queue_len=65536,
    feat_dim=128,
    momentum=0.999,
    data_preprocessor=dict(
        mean=(123.675, 116.28, 103.53),
        std=(58.395, 57.12, 57.375),
        bgr_to_rgb=True),
    backbone=dict(
        type='ResNet',
        depth=50,
        in_channels=3,
        out_indices=[4],  # 0: conv-1, x: stage-x
        norm_cfg=dict(type='BN')),
    neck=dict(
        type='MoCoV2Neck',
        in_channels=2048,
        hid_channels=2048,
        out_channels=128,
        with_avg_pool=True),
    head=dict(
        type='ContrastiveHead',
        loss=dict(type='mmcls.CrossEntropyLoss'),
        temperature=0.2))
```

../_base_/datasets/imagenet_mocov2.py is the base configuration file for the dataset of MoCo v2. The configuration file specifies the configuration for dataset and dataloader.

```
# dataset settings
# We use the ``ImageNet`` dataset implemented by mmclassification, so there
# is a ``mmcls`` prefix.
dataset_type = 'mmcls.ImageNet'
data_root = 'data/imagenet/'
# Since we use ``ImageNet`` from mmclassification, we need to set the
```

(continues on next page)

(continued from previous page)

```
# custom_imports here.
custom_imports = dict(imports='mmcls.datasets', allow_failed_imports=False)
# The difference between mocov2 and mocov1 is the transforms in the pipeline
view_pipeline = [
    dict(
        type='RandomResizedCrop', size=224, scale=(0.2, 1.), backend='pillow'),
    dict(
        type='RandomApply',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.4,
                contrast=0.4,
                saturation=0.4,
                hue=0.1)
        ],
        prob=0.8),
    dict(
        type='RandomGrayscale',
        prob=0.2,
        keep_channels=True,
        channel_weights=(0.114, 0.587, 0.2989)),
    dict(type='RandomGaussianBlur', sigma_min=0.1, sigma_max=2.0, prob=0.5),
    dict(type='RandomFlip', prob=0.5),
]

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='MultiView', num_views=2, transforms=[view_pipeline]),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]

train_dataloader = dict(
    batch_size=32,
    num_workers=8,
    drop_last=True,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    collate_fn=dict(type='default_collate'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='meta/train.txt',
        data_prefix=dict(img_path='train/'),
        pipeline=train_pipeline))
```

`../_base_/schedules/sgd_coslr-200e_in1k.py` is the base configuration file for the training schedules of MoCo v2.

```
# optimizer
optimizer = dict(type='SGD', lr=0.03, weight_decay=1e-4, momentum=0.9)
optim_wrapper = dict(type='OptimWrapper', optimizer=optimizer)
```

(continues on next page)

(continued from previous page)

```
# learning rate scheduler
# use cosine learning rate decay here
param_scheduler = [
    dict(type='CosineAnnealingLR', T_max=200, by_epoch=True, begin=0, end=200)
]

# loop settings
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=200)
```

`../_base_/default_runtime.py` contains the default runtime settings. The runtime settings include some basic components during training, such as default_hooks and log_processor

```
default_scope = 'mmselfsup'

default_hooks = dict(
    runtime_info=dict(type='RuntimeInfoHook'),
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=10),
    sampler_seed=dict(type='DistSamplerSeedHook'),
)

env_cfg = dict(
    cudnn_benchmark=False,
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),
    dist_cfg=dict(backend='nccl'),
)

log_processor = dict(
    window_size=10,
    custom_cfg=[dict(data_src='', method='mean', windows_size='global')])

vis_backends = [dict(type='LocalVisBackend')]
visualizer = dict(
    type='SelfSupVisualizer', vis_backends=vis_backends, name='visualizer')
# custom_hooks = [dict(type='SelfSupVisualizationHook', interval=1)]

log_level = 'INFO'
load_from = None
resume = False
```

### 3.1.3 Inherit and Modify Config File

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

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 modifications, you can first inherit the basic configuration of MoCo v2 by specifying `_base_ ='./mocov2_resnet50_8xb32-coslr-200e_in1k.py'` (The path relative to your config file), and then modify the necessary fields in your customized 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`, except for changing the training epochs from 200 to 800, you can create a new config file `configs/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-800e_in1k.py` with the content as below:

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

# learning rate scheduler
param_scheduler = [
    dict(type='CosineAnnealingLR', T_max=800, by_epoch=True, begin=0, end=800)
]

# runtime settings
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=800)
```

#### Use Intermediate Variables in Configs

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

For example, `dataset_type`, `data_root`, `train_pipeline` are the intermediate variables of `dataset`. We first need to define them and then pass them into `dataset`.

```
# dataset settings

# Since we use ``ImageNet`` from mmclassification, we need to set the
# custom_imports here.
custom_imports = dict(imports='mmcls.datasets', allow_failed_imports=False)

# We use the ``ImageNet`` dataset implemented by mmclassification, so there
# is a ``mmcls`` prefix.
dataset_type = 'mmcls.ImageNet'
data_root = 'data/imagenet/'

# The difference between mocov2 and mocov1 is the transforms in the pipeline
view_pipeline = [
    dict(
        type='RandomResizedCrop', size=224, scale=(0.2, 1.), backend='pillow'),
    dict(
        type='RandomApply',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.4,
```

(continues on next page)

(continued from previous page)

```
                contrast=0.4,
                saturation=0.4,
                hue=0.1)
        ],
        prob=0.8),
    dict(
        type='RandomGrayscale',
        prob=0.2,
        keep_channels=True,
        channel_weights=(0.114, 0.587, 0.2989)),
    dict(type='RandomGaussianBlur', sigma_min=0.1, sigma_max=2.0, prob=0.5),
    dict(type='RandomFlip', prob=0.5),
]

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='MultiView', num_views=2, transforms=[view_pipeline]),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]

train_dataloader = dict(
    batch_size=32,
    num_workers=8,
    drop_last=True,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    collate_fn=dict(type='default_collate'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='meta/train.txt',
        data_prefix=dict(img_path='train/'),
        pipeline=train_pipeline))
```

### Ignore Some Fields in the Base Configs

Sometimes, you may set `_delete_=True` to ignore some of the fields in base configs. You can refer to mmengine for more instructions.

The following is an example. If you want to use `MoCoV2Neck` in SimCLR, directly inheriting and modifying it will report `get unexcepected keyword 'num_layers'` error since `NonLinearNeck` and `MoCoV2Neck` use different keywords to construct. In this case, adding `_delete_=True` would replace all old keys in `neck` field with new keys:

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

model = dict(
    neck=dict(
        _delete_=True,
        type='MoCoV2Neck',
        in_channels=2048,
        hid_channels=2048,
```

(continues on next page)

(continued from previous page)

```
        out_channels=128,
        with_avg_pool=True))
```

### Reuse Some Fields in the Base Configs

Sometimes, you may reuse some fields in base configs, so as to avoid duplication of variables. You can refer to mmengine for more instructions.

The following is an example of reusing the num_classes variable in the base config file. Please refer to configs/selfsup/odc/odc_resnet50_8xb64-steplr-440e_in1k.py for more details.

```
_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}}),
)
```

## 3.1.4 Modify Config through Script Arguments

When using the script tools/train.py/tools/test.py to submit tasks or using some other tools, you can directly modify the content of the configuration file 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 all BN modules in backbone 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, some config files contain param_scheduler = "[dict(type='CosineAnnealingLR',T_max=200,by_epoch=True,begin=0,end=200)]". If you want to change this key, you may specify --cfg-options param_scheduler = "[dict(type='LinearLR', start_factor=1e-4, by_epoch=True,begin=0,end=40,convert_to_iter_based=True)]". Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks for the specified value.

---

**Note:**

```
This modification only supports modifying configuration items of string, int, float,␣
↪boolean, None, list and tuple types.
More specifically, for list and tuple types, the elements inside them must also be one␣
↪of the above seven types.
```

### 3.1.5 Import Modules from Other MM-Codebases

**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 config files. For example, you may refer to OpenMMLab Algorithm Competition Project .

Add the following code to your own config files:

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

## 3.2 Tutorial 2: 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.



```
mmselfsup
├── mmselfsup
├── tools
├── configs
```

(continues on next page)

(continued from previous page)

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

### 3.2.1 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

### 3.2.2 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

### 3.2.3 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/dataset_converters/convert_inaturalist.py`

### 3.2.4 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/dataset_converters/prepare_voc07_cls.sh $YOUR_DATA_ROOT
```

### 3.2.5 Prepare CIFAR10

`MMSelfSup` uses `CIFAR10` implemented by `MMClassification`. In addition, `MMClassification` supports automatic download of the `CIFAR10` dataset, you just need to specify the download folder in the `data_root` field. And specify `test_mode=False` / `test_mode=True` to use the training or test dataset. For more details, please refer to docs in `MMClassification`.

### 3.2.6 Prepare datasets for detection and segmentation

#### Detection

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

#### Segmentation

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

## 3.3 Tutorial 3: Pretrain with Existing Models



This page provides the basic usage about how to run algorithms and how to use some tools in MMSelfSup. For installation instructions and data preparation, please refer to *get_started.md* and *dataset_prepare.md*.

### 3.3.1 Start to Train

**Note**: The default learning rate in config files is for specific number of GPUs, which is indicated in the config names. If you use different number of GPUs, the total batch size will be changed in proportion. In this case, you have to scale the learning rate following `new_lr = old_lr * new_batchsize / old_batchsize`.

#### Train with a single GPU

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

A simple example to start training:

```
python tools/train.py configs/selfsup/mae/mae_vit-base-p16_8xb512-coslr-400e_in1k.py
```

#### Train with CPU

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

**Note**: We do not recommend users to use CPU for training because it is too slow. We support this feature to allow users to debug on machines without GPU for convenience.

#### Train with multiple GPUs

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

Optional arguments:

- `--work-dir`: Indicate your custom work directory to save checkpoints and logs.
- `--resume`: Automatically find the latest checkpoint in your work directory. Or set `--resume ${CHECKPOINT_PATH}` to load the specific checkpoint file.
- `--amp`: Enable automatic-mixed-precision training.
- `--cfg-options`: Setting `--cfg-options` will modify the original configs. For example, setting `--cfg-options randomness.seed=0` will set seed for random number.

An example to start training with 8 GPUs:

```
bash tools/dist_train.sh configs/selfsup/mae/mae_vit-base-p16_8xb512-coslr-400e_in1k.py 8
```

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} [optional arguments]
```

An example to start training with 8 GPUs:

```
# The default setting: GPUS_PER_NODE=8 GPUS=8
bash tools/slurm_train.sh Dummy Test_job configs/selfsup/mae/mae_vit-base-p16_8xb512-
↪coslr-400e_in1k.py
```

### Train with multiple machines

If you launch with multiple machines simply connected with ethernet, you can simply run the 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 to refer to slurm_train.sh to set appropriate parameters and environment variables.

### 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 the communication conflict.

If you use `dist_train.sh` to launch training jobs:

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

CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 bash tools/dist_train.sh ${CONFIG_FILE} 4 --work-
↪dir tmp_work_dir_2
```

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

Option 1:

In `config1.py`:

```
env_cfg = dict(dist_cfg=dict(backend='nccl', port=29500))
```

In `config2.py`:

```
env_cfg = dict(dist_cfg=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 [optional arguments]

CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py [optional arguments]
```

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 the configuration file.

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py --work-dir tmp_work_dir_1 --cfg-options env_cfg.dist_cfg.port=29500

CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 bash tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config2.py --work-dir tmp_work_dir_2 --cfg-options env_cfg.dist_cfg.port=29501
```

## 3.4 Tutorial 4: Pretrain with Custom Dataset



In this tutorial, we provide some tips on how to conduct self-supervised learning on your own dataset(without the need of label).

### 3.4.1 Train MAE on Custom Dataset

In MMSelfSup, We support the `CustomDataset` from MMClassification(similar to the `ImageFolder` in `torchvision`), which is able to read the images within the specified folder directly. You only need to prepare the path information of the custom dataset and edit the config.

#### Step-1: Get the path of custom dataset

It should be like `data/custom_dataset/`

#### Step-2: Choose one config as template

Here, we would like to use `configs/selfsup/mae/mae_vit-base-p16_8xb512-coslr-400e_in1k.py` as the example. We first copy this config file and rename it as `mae_vit-base-p16_8xb512-coslr-400e_${custom_dataset}.py`.

- `custom_dataset`: indicate which dataset you used, e.g.,`in1k` for ImageNet dataset, `coco` for COCO dataset

The content of this config is:

```
_base_ = [
    '../_base_/models/mae_vit-base-p16.py',
    '../_base_/datasets/imagenet_mae.py',
    '../_base_/schedules/adamw_coslr-200e_in1k.py',
    '../_base_/default_runtime.py',
]
```

(continues on next page)

(continued from previous page)

```
# dataset 8 x 512
train_dataloader = dict(batch_size=512, num_workers=8)

# optimizer wrapper
optimizer = dict(
    type='AdamW', lr=1.5e-4 * 4096 / 256, betas=(0.9, 0.95), weight_decay=0.05)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    paramwise_cfg=dict(
        custom_keys={
            'ln': dict(decay_mult=0.0),
            'bias': dict(decay_mult=0.0),
            'pos_embed': dict(decay_mult=0.),
            'mask_token': dict(decay_mult=0.),
            'cls_token': dict(decay_mult=0.)
        }))

# learning rate scheduler
param_scheduler = [
    dict(
        type='LinearLR',
        start_factor=1e-4,
        by_epoch=True,
        begin=0,
        end=40,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=360,
        by_epoch=True,
        begin=40,
        end=400,
        convert_to_iter_based=True)
]

# runtime settings
# pre-train for 400 epochs
train_cfg = dict(max_epochs=400)
default_hooks = dict(
    logger=dict(type='LoggerHook', interval=100),
    # only keeps the latest 3 checkpoints
    checkpoint=dict(type='CheckpointHook', interval=1, max_keep_ckpts=3))

# randomness
randomness = dict(seed=0, diff_rank_seed=True)
resume = True
```

### Step-3: Edit the dataset related config

The dataset related config is defined in `'../_base_/datasets/imagenet_mae.py'` in `_base_`. We then copy the content of dataset config file into our created file `mae_vit-base-p16_8xb512-coslr-400e_${custom_dataset}.py`.

- Then we remove the `'../_base_/datasets/imagenet_mae.py'` in `_base_`.
- Set the `dataset_type = 'mmcls.CustomDataset'`, and the path of the custom dataset `data_root = /dataset/my_custom_dataset`.
- Remove the `ann_file` in `train_dataloader`, and edit the `data_prefix` if needed.

**Note:** The `CustomDataset` is implemented in MMClassification, and we set the `dataset_type=mmcls.CustomDataset`.

And the edited config will be like this:

```
# >>>>>>>>>>>>>>>>>>>>> Start of Changed >>>>>>>>>>>>>>>>>>>>>>>>>
_base_ = [
    '../_base_/models/mae_vit-base-p16.py',
    # '../_base_/datasets/imagenet_mae.py',
    '../_base_/schedules/adamw_coslr-200e_in1k.py',
    '../_base_/default_runtime.py',
]

# custom dataset
dataset_type = 'mmcls.CustomDataset'
data_root = 'data/custom_dataset/'
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='RandomResizedCrop',
        size=224,
        scale=(0.2, 1.0),
        backend='pillow',
        interpolation='bicubic'),
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]

# dataset 8 x 512
train_dataloader = dict(
    batch_size=512,
    num_workers=8,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    collate_fn=dict(type='default_collate'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        # ann_file='meta/train.txt', # removed if you don't have the annotation file
        data_prefix=dict(img_path='./'),
        pipeline=train_pipeline))
```

(continues on next page)

(continued from previous page)

```
# <<<<<<<<<<<<<<<<<<<<<< End of Changed <<<<<<<<<<<<<<<<<<<<<<<<<<<


# optimizer wrapper
optimizer = dict(
    type='AdamW', lr=1.5e-4 * 4096 / 256, betas=(0.9, 0.95), weight_decay=0.05)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    paramwise_cfg=dict(
        custom_keys={
            'ln': dict(decay_mult=0.0),
            'bias': dict(decay_mult=0.0),
            'pos_embed': dict(decay_mult=0.),
            'mask_token': dict(decay_mult=0.),
            'cls_token': dict(decay_mult=0.)
        }))

# learning rate scheduler
param_scheduler = [
    dict(
        type='LinearLR',
        start_factor=1e-4,
        by_epoch=True,
        begin=0,
        end=40,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=360,
        by_epoch=True,
        begin=40,
        end=400,
        convert_to_iter_based=True)
]

# runtime settings
# pre-train for 400 epochs
train_cfg = dict(max_epochs=400)
default_hooks = dict(
    logger=dict(type='LoggerHook', interval=100),
    # only keeps the latest 3 checkpoints
    checkpoint=dict(type='CheckpointHook', interval=1, max_keep_ckpts=3))

# randomness
randomness = dict(seed=0, diff_rank_seed=True)
resume = True
```

By using the edited config file, you are able to train a self-supervised model with MAE algorithm on the custom dataset.

### 3.4.2 Train MAE on COCO Dataset

**Note:** You need to install MMDetection to use the `mmdet.CocoDataset` follow this documentation

Follow the aforementioned idea, we also present an example of how to train MAE on COCO dataset. The edited file will be like this:

```
# >>>>>>>>>>>>>>>>>>>>> Start of Changed >>>>>>>>>>>>>>>>>>>>>>>>>
_base_ = [
    '../_base_/models/mae_vit-base-p16.py',
    # '../_base_/datasets/imagenet_mae.py',
    '../_base_/schedules/adamw_coslr-200e_in1k.py',
    '../_base_/default_runtime.py',
]

# custom dataset
dataset_type = 'mmdet.CocoDataset'
data_root = 'data/coco/'

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='RandomResizedCrop',
        size=224,
        scale=(0.2, 1.0),
        backend='pillow',
        interpolation='bicubic'),
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]

train_dataloader = dict(
    batch_size=128,
    num_workers=8,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    collate_fn=dict(type='default_collate'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_train2017.json',
        data_prefix=dict(img='train2017/'),
        pipeline=train_pipeline))

# <<<<<<<<<<<<<<<<<<<<<<<< End of Changed <<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# optimizer wrapper
optimizer = dict(
    type='AdamW', lr=1.5e-4 * 4096 / 256, betas=(0.9, 0.95), weight_decay=0.05)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
```

(continues on next page)

(continued from previous page)

```
    paramwise_cfg=dict(
        custom_keys={
            'ln': dict(decay_mult=0.0),
            'bias': dict(decay_mult=0.0),
            'pos_embed': dict(decay_mult=0.),
            'mask_token': dict(decay_mult=0.),
            'cls_token': dict(decay_mult=0.)
        }))

# learning rate scheduler
param_scheduler = [
    dict(
        type='LinearLR',
        start_factor=1e-4,
        by_epoch=True,
        begin=0,
        end=40,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=360,
        by_epoch=True,
        begin=40,
        end=400,
        convert_to_iter_based=True)
]

# runtime settings
# pre-train for 400 epochs
train_cfg = dict(max_epochs=400)
default_hooks = dict(
    logger=dict(type='LoggerHook', interval=100),
    # only keeps the latest 3 checkpoints
    checkpoint=dict(type='CheckpointHook', interval=1, max_keep_ckpts=3))

# randomness
randomness = dict(seed=0, diff_rank_seed=True)
resume = True
```

### 3.4.3 Train SimCLR on Custom Dataset

We provide an example of using SimCLR on custom dataset, the main idea is similar to the *Train MAE on Custom Dataset* .

The template config is `configs/selfsup/simclr/simclr_resnet50_8xb32-coslr-200e_in1k.py`. And the edited config is:

```
# >>>>>>>>>>>>>>>>>>>>>> Start of Changed >>>>>>>>>>>>>>>>>>>>>>>>>
_base_ = [
    '../_base_/models/simclr.py',
    # '../_base_/datasets/imagenet_simclr.py',
```

(continues on next page)

(continued from previous page)

```
    '../_base_/schedules/lars_coslr-200e_in1k.py',
    '../_base_/default_runtime.py',
]

# custom dataset
dataset_type = 'mmcls.CustomDataset'
data_root = 'data/custom_dataset/'

view_pipeline = [
    dict(type='RandomResizedCrop', size=224, backend='pillow'),
    dict(type='RandomFlip', prob=0.5),
    dict(
        type='RandomApply',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.8,
                contrast=0.8,
                saturation=0.8,
                hue=0.2)
        ],
        prob=0.8),
    dict(
        type='RandomGrayscale',
        prob=0.2,
        keep_channels=True,
        channel_weights=(0.114, 0.587, 0.2989)),
    dict(type='RandomGaussianBlur', sigma_min=0.1, sigma_max=2.0, prob=0.5),
]

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='MultiView', num_views=2, transforms=[view_pipeline]),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]

train_dataloader = dict(
    batch_size=32,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    collate_fn=dict(type='default_collate'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        # ann_file='meta/train.txt',
        data_prefix=dict(img_path='./'),
        pipeline=train_pipeline))
# <<<<<<<<<<<<<<<<<<<<<< End of Changed <<<<<<<<<<<<<<<<<<<<<<<<<<<


# optimizer
```

(continues on next page)

(continued from previous page)

```
optimizer = dict(type='LARS', lr=0.3, momentum=0.9, weight_decay=1e-6)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    paramwise_cfg=dict(
        custom_keys={
            'bn': dict(decay_mult=0, lars_exclude=True),
            'bias': dict(decay_mult=0, lars_exclude=True),
            # bn layer in ResNet block downsample module
            'downsample.1': dict(decay_mult=0, lars_exclude=True),
        }))

# runtime settings
default_hooks = dict(
    # only keeps the latest 3 checkpoints
    checkpoint=dict(type='CheckpointHook', interval=10, max_keep_ckpts=3))
```

### 3.4.4 Load pre-trained model to speedup convergence

To speedup the convergence of the model on your own dataset. You may use the pre-trained model as the initialization for the model's weight. You just need to specify the url of the pre-trained model via command. You can find our provide pre-trained checkpoint here: Model Zoo

```
bash tools/dist_train.sh ${CONFIG} ${GPUS} --cfg-options model.pretrained=${PRETRAIN}
```

- `CONFIG`: the edited config path
- `GPUS`: the number of GPU
- `PRETRAIN`: the checkpoint url of pre-trained model provided by MMSelfSup

CHAPTER

FOUR

# DOWNSTREAM TASKS

## 4.1 Classification



In MMSelfSup, we provide many benchmarks for classification, thus the models can be evaluated on different classification tasks. Here are comprehensive tutorials and examples to explain how to run all classification benchmarks with MMSelfSup. We provide scripts in folder `tools/benchmarks/classification/`, which has 2 `.sh` files, 1 folder for VOC SVM related classification task and 1 folder for ImageNet nearest-neighbor classification task.

### 4.1.1 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 the 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 the 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

### 4.1.2 Linear Evaluation and Fine-tuning

Linear evaluation and fine-tuning are two of the most general benchmarks. We provide config files and scripts to launch the training and testing for Linear Evaluation and Fine-tuning. The supported datasets are **ImageNet**, **Places205** and **iNaturalist18**.

First, make sure you have installed MIM, which is also a project of OpenMMLab.

```
pip install openmim
```

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

Then, run the command below.

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

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

Remarks:

- `${CONFIG}`: Use config files under `configs/benchmarks/classification/`. Specifically, `imagenet` (excluding `imagenet_*percent` folders), `places205` and `inaturalist2018`.
- `${PRETRAIN}`: the pre-trained model file.

Example:

```
bash ./tools/benchmarks/classification/mim_dist_train.sh \
configs/benchmarks/classification/imagenet/resnet50_linear-8xb32-coslr-100e_in1k.py \
work_dir/pretrained_model.pth
```

If you want to test the well-trained model, please run the command below.

```
# distributed version
bash tools/benchmarks/classification/mim_dist_test.sh ${CONFIG} ${CHECKPOINT}

# slurm version
bash tools/benchmarks/classification//mim_slurm_test.sh ${PARTITION} ${CONFIG} $
↪{CHECKPOINT}
```

Remarks:

- `${CHECKPOINT}`: The well-trained classification model that you want to test.

Example:

```
bash ./tools/benchmarks/mmsegmentation/mim_dist_test.sh \
configs/benchmarks/classification/imagenet/resnet50_linear-8xb32-coslr-100e_in1k.py \
work_dir/model.pth
```

### 4.1.3 ImageNet Semi-Supervised Classification

To run ImageNet semi-supervised classification, we still use the same `.sh` script as Linear Evaluation and Fine-tuning to launch training.

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.

### 4.1.4 ImageNet Nearest-Neighbor Classification

```
Only support CNN-style backbones (like ResNet50).
```

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

```
# distributed version
bash tools/benchmarks/classification/knn_imagenet/dist_test_knn.sh ${SELFSUP_CONFIG} $
↪{PRETRAIN} [optional arguments]

# slurm version
bash tools/benchmarks/classification/knn_imagenet/slurm_test_knn.sh ${PARTITION} ${JOB_
↪NAME} ${SELFSUP_CONFIG} ${CHECKPOINT} [optional arguments]
```

Remarks:

- `${SELFSUP_CONFIG}` is the config file of the self-supervised experiment.
- `${CHECKPOINT}`: the path of checkpoint model file.
- if you want to change GPU numbers, you could add `GPUS_PER_NODE=4 GPUS=4` at the beginning of the command.
- `[optional arguments]`: for optional arguments, you can refer to the script

An example of command

```
# distributed version
bash tools/benchmarks/classification/knn_imagenet/dist_test_knn.sh \
    configs/selfsup/barlowtwins/barlowtwins_resnet50_8xb256-coslr-300e_in1k.py \
    https://download.openmmlab.com/mmselfsup/1.x/barlowtwins/barlowtwins_resnet50_8xb256-
↪coslr-300e_in1k/barlowtwins_resnet50_8xb256-coslr-300e_in1k_20220825-57307488.pth
```

## 4.2 Detection

- *Detection*
  - *Train*
  - *Test*

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
mim install 'mmdet>=3.0.0rc0'
```

It is very easy to install the package.

Besides, please refer to MMDet for installation and data preparation

### 4.2.1 Train

After installation, you can run MMDetection with simple command.

```
# distributed version
bash tools/benchmarks/mmdetection/mim_dist_train_c4.sh ${CONFIG} ${PRETRAIN} ${GPUS}
bash tools/benchmarks/mmdetection/mim_dist_train_fpn.sh ${CONFIG} ${PRETRAIN} ${GPUS}

# slurm version
bash tools/benchmarks/mmdetection/mim_slurm_train_c4.sh ${PARTITION} ${CONFIG} $
↪{PRETRAIN}
bash tools/benchmarks/mmdetection/mim_slurm_train_fpn.sh ${PARTITION} ${CONFIG} $
↪{PRETRAIN}
```

Remarks:

- `${CONFIG}`: Use config files under `configs/benchmarks/mmdetection/`. Since repositories of OpenMMLab have support referring config files across different repositories, we can easily leverage the configs from MMDetection like:

```
_base_ = 'mmdet::mask_rcnn/mask-rcnn_r50-caffe-c4_1x_coco.py'
```

Writing your config files from scratch is also supported.

- `${PRETRAIN}`: the pre-trained model file.
- `${GPUS}`: The number of GPUs that you want to use to train. We adopt 8 GPUs for detection tasks by default.

Example:

```
bash ./tools/benchmarks/mmdetection/mim_dist_train_c4.sh \
configs/benchmarks/mmdetection/coco/mask-rcnn_r50-c4_ms-1x_coco.py \
https://download.openmmlab.com/mmselfsup/1.x/byol/byol_resnet50_16xb256-coslr-200e_in1k/
↪byol_resnet50_16xb256-coslr-200e_in1k_20220825-de817331.pth 8
```

Or if you want to do detection task with detectron2, we also provide 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 tools/benchmarks/detectron2
python convert-pretrain-to-detectron2.py ${WEIGHT_FILE} ${OUTPUT_FILE} # must use .pkl␣
↪as the output extension.
bash run.sh ${DET_CFG} ${OUTPUT_FILE}
```

### 4.2.2 Test

After training, you can also run the command below to test your model.

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

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

Remarks:

- `${CHECKPOINT}`: The well-trained detection model that you want to test.

Example:

```
bash ./tools/benchmarks/mmdetection/mim_dist_test.sh \
configs/benchmarks/mmdetection/coco/mask-rcnn_r50_fpn_ms-1x_coco.py \
https://download.openmmlab.com/mmselfsup/1.x/byol/byol_resnet50_16xb256-coslr-200e_in1k/
↪byol_resnet50_16xb256-coslr-200e_in1k_20220825-de817331.pth 8
```

## 4.3 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
mim install 'mmsegmentation>=1.0.0rc0'
```

It is very easy to install the package.

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

### 4.3.1 Train

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/. Since repositories of OpenMMLab have support referring config files across different repositories, we can easily leverage the configs from MMSegmentation like:

```
_base_ = 'mmseg::fcn/fcn_r50-d8_4xb2-40k_cityscapes-769x769.py'
```

Writing your config files from scratch is also supported.

- ${PRETRAIN}: the pre-trained model file.
- ${GPUS}: The number of GPUs that you want to use to train. We adopt 4 GPUs for segmentation tasks by default.

Example:

```
bash ./tools/benchmarks/mmsegmentation/mim_dist_train.sh \
configs/benchmarks/mmsegmentation/voc12aug/fcn_r50-d8_4xb4-20k_voc12aug-512x512.py \
https://download.openmmlab.com/mmselfsup/1.x/byol/byol_resnet50_16xb256-coslr-200e_in1k/
↪byol_resnet50_16xb256-coslr-200e_in1k_20220825-de817331.pth 4
```

### 4.3.2 Test

After training, you can also run the command below to test your model.

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

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

Remarks:

- ${CHECKPOINT}: The well-trained segmentation model that you want to test.

Example:

```
bash ./tools/benchmarks/mmsegmentation/mim_dist_test.sh \
configs/benchmarks/mmsegmentation/voc12aug/fcn_r50-d8_4xb4-20k_voc12aug-512x512.py \
https://download.openmmlab.com/mmselfsup/1.x/byol/byol_resnet50_16xb256-coslr-200e_in1k/
↪byol_resnet50_16xb256-coslr-200e_in1k_20220825-de817331.pth 4
```

CHAPTER

FIVE

# USEFUL TOOLS

## 5.1 Visualization

Visualization can give an intuitive interpretation of the performance of the model.



### 5.1.1 How visualization is implemented

It is recommended to learn the basic concept of visualization in documentation.

OpenMMLab 2.0 introduces the visualization object `Visualizer` and several visualization backends `VisBackend`. The diagram below shows the relationship between `Visualizer` and `VisBackend`,

### 5.1.2 What Visualization do in MMSelfsup

(1) Save training data using different storage backends

The backends in MMEngine includes `LocalVisBackend`, `TensorboardVisBackend` and `WandbVisBackend` .

During training, after_train_iter() in the default hook `LoggerHook` will be called, and use `add_scalars` in different backends, as follows:

```
...
def after_train_iter(...):
    ...
    runner.visualizer.add_scalars(
        tag, step=runner.iter + 1, file_path=self.json_log_path)
...
```

(2) Browse dataset

The function `add_datasample()` is impleted in *SelfSupVisualizer*, and it is mainly used in browse_dataset.py for browsing dataset. More tutorial is in section *Visualize Datasets*

### 5.1.3 Use Different Storage Backends

If you want to use a different backend (Wandb, Tensorboard, or a custom backend with a remote window), just change the `vis_backends` in the config, as follows:

**Local**

```
vis_backends = [dict(type='LocalVisBackend')]
```

**Tensorboard**

```
vis_backends = [dict(type='TensorboardVisBackend')]
visualizer = dict(
    type='SelfSupVisualizer', vis_backends=vis_backends, name='visualizer')
```

E.g.

**Wandb**

```
vis_backends = [dict(type='WandbVisBackend')]
visualizer = dict(
    type='SelfSupVisualizer', vis_backends=vis_backends, name='visualizer')
```

E.g.

### 5.1.4 Customize Visualization

The customization of the visualization is similar to other components. If you want to customize `Visualizer`, `VisBackend` or `VisualizationHook`, you can refer to Visualization Doc in MMEngine.

### 5.1.5 Visualize Datasets

`tools/misc/browse_dataset.py` helps the user to browse a mmselfsup dataset (transformed images) visually, or save the image to a designated directory.

```
python tools/misc/browse_dataset.py ${CONFIG} [-h] [--skip-type ${SKIP_TYPE[SKIP_TYPE...
↪]}] [--output-dir ${OUTPUT_DIR}] [--not-show] [--show-interval ${SHOW_INTERVAL}]
```

An example:

```
python tools/misc/browse_dataset.py configs/selfsup/simsiam/simsiam_resnet50_8xb32-coslr-
↪100e_in1k.py
```

An example of visualization:

- The left two pictures are images from contrastive learning data pipeline.
- The right one is a masked image.

### 5.1.6 Visualize 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 t-SNE, which listed in the directory `configs/tsne/`
- `CKPT_PATH`: the path or link of the 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

An example of command:

```
python ./tools/analysis_tools/visualize_tsne.py \
    configs/tsne/resnet50_imagenet.py \
    --checkpoint https://download.openmmlab.com/mmselfsup/1.x/mocov2/mocov2_resnet50_
↪8xb32-coslr-200e_in1k/mocov2_resnet50_8xb32-coslr-200e_in1k_20220825-b6d23c86.pth \
    --work-dir  ./work_dirs/tsne/mocov2/ \
    --max-num-class 100
```

An example of visualization, left is from `MoCoV2_ResNet50` and right is from `MAE_ViT-base`:

### 5.1.7 Visualize Low-level Feature Reconstruction

We provide several reconstruction visualization for listed algorithms:

- MAE
- SimMIM
- MaskFeat

Users can run command below to visualize the reconstruction.

```
python tools/analysis_tools/visualize_reconstruction.py ${CONFIG_FILE} \
    --checkpoint ${CKPT_PATH} \
    --img-path ${IMAGE_PATH} \
    --out-file ${OUTPUT_PATH}
```

Arguments:

- `CONFIG_FILE`: config file for the pre-trained model.
- `CKPT_PATH`: the path of model's checkpoint.
- `IMAGE_PATH`: the input image path.
- `OUTPUT_PATH`: the output image path, including 4 sub-images.
- `[optional arguments]`: for optional arguments, you can refer to visualize_reconstruction.py

An example:

```
python tools/analysis_tools/visualize_reconstruction.py configs/selfsup/mae/mae_vit-huge-
↪p16_8xb512-amp-coslr-1600e_in1k.py \
    --checkpoint https://download.openmmlab.com/mmselfsup/1.x/mae/mae_vit-huge-p16_
↪8xb512-fp16-coslr-1600e_in1k/mae_vit-huge-p16_8xb512-fp16-coslr-1600e_in1k_20220916-
↪ff848775.pth \
    --img-path data/imagenet/val/ILSVRC2012_val_00000003.JPEG \
    --out-file test_mae.jpg \
    --norm-pix


# As for SimMIM, it generates the mask in data pipeline, thus we use '--use-vis-pipeline'␣
↪to apply 'vis_pipeline' defined in config instead of the pipeline defined in script.
python tools/analysis_tools/visualize_reconstruction.py configs/selfsup/simmim/simmim_
↪swin-large_16xb128-amp-coslr-800e_in1k-192.py \
    --checkpoint https://download.openmmlab.com/mmselfsup/1.x/simmim/simmim_swin-large_
↪16xb128-amp-coslr-800e_in1k-192/simmim_swin-large_16xb128-amp-coslr-800e_in1k-192_
↪20220916-4ad216d3.pth \
    --img-path data/imagenet/val/ILSVRC2012_val_00000003.JPEG \
    --out-file test_simmim.jpg \
    --use-vis-pipeline
```

Results of MAE:

Results of SimMIM:

Results of MaskFeat:

### 5.1.8 Visualize Shape Bias

Shape bias measures how a model relies the shapes, compared to texture, to sense the semantics in images. For more details, we recommend interested readers to this paper. MMSelfSup provide an off-the-shelf toolbox to obtain the shape bias of a classification model. You can following these steps below:

#### Prepare the dataset

First you should download the cue-conflict to `data` folder, and then unzip this dataset. After that, you `data` folder should have the following structure:

```
data
├──cue-conflict
|      |──airplane
|      |──bear
|      ...
|      |── truck
```

#### Modify the config for classification

Replace the original test_dataloader and test_evaluation with following configurations

```
test_pipeline = [...] # copy existing test transforms here
test_dataloader = dict(
    dataset=dict(
        type='CustomDataset',
        data_root='data/cue-conflict',
        pipeline=test_pipeline,
        _delete_=True),
    drop_last=False)
test_evaluator = dict(
    type='mmselfsup.ShapeBiasMetric',
    _delete_=True,
    csv_dir='directory/to/save/the/csv/file',
    model_name='your_model_name')
```

Please note you should make custom modifications to the `csv_dir` and `model_name`. You can follow the toy example here to make custom modification to your evaluation.

#### Inference your model with above modified config file

Then you should inferece your model on the `cue-conflict` dataset with the your modified config files.

```
# For Slurm
GPUS_PER_NODE=1 GPUS=1 bash tools/benchmarks/classification/mim_slurm_test.sh $partition
↪$config $checkpoint
```

```
# For PyTorch
GPUS=1 bash tools/benchmarks/classification/mim_dist_test.sh $config $checkpoint
```

After that, you should obtain a csv file, named `cue-conflict_model-name_session-1.csv`. Besides this file, you should also download these csv files to the `csv_dir`.

#### Plot shape bias

Then we can start to plot the shape bias

```
python tools/analysis_tools/visualize_shape_bias.py --csv-dir $CVS_DIR --result-dir $CSV_
↪DIR --colors $RGB --markers o --plotting-names $YOU_MODEL_NAME --model-names $YOU_
↪MODEL_NAME
```

- `--csv-dir`, the same directory to save these csv files
- `--colors`, should be the RGB values, formatted in R G B, e.g. 100 100 100, and can be multiple RGB values, if you want to plot the shape bias of several models
- `--plotting-names`, the name of the legend in the shape bias figure, and you can set it as your model name. If you want to plot several models, plotting_names can be multiple values
- `--model-names`, should be the same name specified in your config, and can be multiple names if you want to plot the shape bias of several models

Please note, every three values for `--colors` corresponds to one value for `--model-names`. After all of above steps, you are expected to obtain the following figure.

## 5.2 Analysis tools



### 5.2.1 Count number of parameters

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

An example:

```
python tools/analysis_tools/count_parameters.py configs/selfsup/mocov2/mocov2_resnet50_
↪8xb32-coslr-200e_in1k.py
```

### 5.2.2 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}
```

An example:

```
python tools/model_converters/publish_model.py YOUR/PATH/epoch_100.pth YOUR/PATH/epoch_
↪100_output.pth
```

### 5.2.3 Reproducibility

If you want to make your performance exactly reproducible, please set `--cfg-options randomness.deterministic=True` to train the final model. Note that this will switch off `torch.backends.cudnn.benchmark` and slow down the training speed.

### 5.2.4 Log Analysis

`tools/analysis_tools/analyze_logs.py` plots loss/lr curves given a training log file. Run `pip install seaborn` first to install the dependency.

```
python tools/analysis_tools/analyze_logs.py plot_curve [--keys ${KEYS}] [--title ${TITLE}
↪] [--legend ${LEGEND}] [--backend ${BACKEND}] [--style ${STYLE}] [--out ${OUT_FILE}]
```

Examples:

- Plot the classification loss of some run.

```
python tools/analysis_tools/analyze_logs.py plot_curve log.json --keys loss_dense --
↪legend loss_dense
```

- Plot the classification and regression loss of some run, and save the figure to a pdf.

```
python tools/analysis_tools/analyze_logs.py plot_curve log.json --keys loss_dense␣
↪loss_single --out losses.pdf
```

- Compare the loss of two runs in the same figure.

```
python tools/analysis_tools/analyze_logs.py plot_curve log1.json log2.json --keys␣
↪loss --legend run1 run2
```

- Compute the average training speed.

```
python tools/analysis_tools/analyze_logs.py cal_train_time log.json [--include-
↪outliers]
```

The output is expected to be like the following.

```
-----Analyze train time of work_dirs/some_exp/20190611_192040.log.json-----
slowest epoch 11, average time is 1.2024
fastest epoch 1, average time is 1.1909
time std over epochs is 0.0028
average iter time: 1.1959 s/iter
```

CHAPTER

SIX

# BASIC CONCEPTS

## 6.1 Data Flow



Data flow defines how data should be passed between two isolated modules, e.g. dataloader and model, as shown below.

In MMSelfSup, we mainly focus on the data flow between dataloader and model, and between model and visualizer. As for the data flow between model and metric, please refer to the docs in other repos, e.g. MMClassification. Also for data flow between model and visualizer, you can refer to *visualization*

### 6.1.1 Data flow between dataloader and model

The data flow between dataloader and model can be generally split into three parts, i) use `PackSelfSupInputs` to pack data from previous transformations into a dictionary, ii) use `collate_fn` to stack a list of tensors into a batched tensor, iii) data preprocessor will move all these data to target device, e.g. GPUS, and unzip the dictionary from the dataloader into a tuple, containing the input images and meta info (`SelfSupDataSample`).

#### Data from dataset

In MMSelfSup, before feeding into the model, data should go through a series of transformations, called `pipeline`, e.g. `RandomResizedCrop` and `ColorJitter`. No matter how many transformations in the pipeline, the last transformation is `PackSelfSupInputs`. `PackSelfSupInputs` will pack these data from previous transformations into a dictionary. The dictionary contains two parts, namely, `inputs` and `data_samples`.

```
# We omit some unimportant code here

class PackSelfSupInputs(BaseTransform):

    def transform(self,
                  results: Dict) -> Dict[torch.Tensor, SelfSupDataSample]:

        packed_results = dict()
        if self.key in results:
```

(continues on next page)

(continued from previous page)

```
            ...
            packed_results['inputs'] = img

        ...
        packed_results['data_samples'] = data_sample

        return packed_results
```

Note: `inputs` contains a list of images, e.g. the multi-views in contrastive learning. Even a single view, `PackSelfSupInputs` will still put it into a list.

### Data from dataloader

After receiving a list of dictionary from dataset, `collect_fn` in dataloader will gather `inputs` in each dict and stack them into a batched tensor. In addition, `data_sample` in each dict will be also collected in a list. Then, it will output a dict, containing the same keys with those of the dict in the received list. Finally, dataloader will output the dict from the `collect_fn`.

### Data from data preprocessor

Data preprocessor is the last step to process the data before feeding into the model. It will apply image normalization, convert BGR to RGB and move all data to the target device, e.g. GPUs. After above steps, it will output a tuple, containing a list of batched images, and a list of data samples.

```
class SelfSupDataPreprocessor(ImgDataPreprocessor):

    def forward(
            self,
            data: dict,
            training: bool = False
    ) -> Tuple[List[torch.Tensor], Optional[list]]:

        assert isinstance(data,
                          dict), 'Please use default_collate in dataloader, \
            instead of pseudo_collate.'

        data = [val for _, val in data.items()]
        batch_inputs, batch_data_samples = self.cast_data(data)
        # channel transform
        if self._channel_conversion:
            batch_inputs = [
                _input[:, [2, 1, 0], ...] for _input in batch_inputs
            ]

        # Convert to float after channel conversion to ensure
        # efficiency
        batch_inputs = [input_.float() for input_ in batch_inputs]

        # Normalization. Here is what is different from
        # :class:`mmengine.ImgDataPreprocessor`. Since there are multiple views
        # for an image for some  algorithms, e.g. SimCLR, each item in inputs
```

(continues on next page)

(continued from previous page)

```
        # is a list, containing multi-views for an image.
        if self._enable_normalize:
            batch_inputs = [(_input - self.mean) / self.std
                            for _input in batch_inputs]

        return batch_inputs, batch_data_samples
```

## 6.2 Structures

- *Structures*
  - *Customized attributes in SelfSupDataSample*
  - *Pack data to SelfSupDataSample in MMSelfSup*

The same as those in other OpenMMLab repositories, MMSelfSup defines a data structure, called `SelfSupDataSample`, which is used to receive and pass data during the whole training/testing process. `SelfSupDataSample` inherits the `BaseDataElement` implemented in MMEngine. We recommend users to refer to BaseDataElement for more in-depth introduction about the basics of `BaseDataElement`. In this tutorials, we mainly discuss some customized features in *SelfSupDataSample*.

### 6.2.1 Customized attributes in SelfSupDataSample

In MMSelfSup, except for images, `SelfSupDataSample` wraps all information required by models, e.g. `mask` requested by mask image modeling(MIM) and `pseudo_label` in pretext tasks. In addition to providing information, it can also accept information generated by models, such as the prediction score. To fulfill these functionalities described above, `SelfSupDataSample` defines five customized attributes:

- gt_label (LabelData), containing the groud-truth label for image.
- sample_idx (InstanceData), containing the index of current image in data list, initialized by dataset in the beginning.
- mask (BaseDataElement), containing the mask in MIM, e.g. SimMIM, CAE.
- pred_label (LabelData), containing the label, predicted by model.
- pseudo_label (BaseDataElement), containing the pseudo label used in pretext tasks, such as the location in Relative Location.

To help users capture the basic idea of SelfSupDataSample, we give a toy example, about how to create a `SelfSupDataSample` instance and set these attributes in it.

```
import torch
from mmselfsup.core import SelfSupDataSample
from mmengine.data import LabelData, InstanceData, BaseDataElement

selfsup_data_sample = SelfSupDataSample()
# set the gt_label in selfsup_data_sample
# gt_label should be the type of LabelData
selfsup_data_sample.gt_label = LabelData(value=torch.tensor([1]))

# setting gt_label to a type, which is not LabelData, will raise an error
selfsup_data_sample.gt_label = torch.tensor([1])
```

(continues on next page)

(continued from previous page)

```
# AssertionError: tensor([1]) should be a <class 'mmengine.data.label_data.LabelData'>␣
↪but got <class 'torch.Tensor'>

# set the sample_idx in selfsup_data_sample
# also, the assigned value of sample_idx should the type of InstanceData
selfsup_data_sample.sample_idx = InstanceData(value=torch.tensor([1]))

# setting the mask in selfsup_data_sample
selfsup_data_sample.mask = BaseDataElement(value=torch.ones((3, 3)))

# setting the pseudo_label in selfsup_data_sample
selfsup_data_sample.pseudo_label = InstanceData(location=torch.tensor([1, 2, 3]))


# After creating these attributes, you can easily fetch values in these attributes
print(selfsup_data_sample.gt_label.value)
# tensor([1])
print(selfsup_data_sample.mask.value.shape)
# torch.Size([3, 3])
```

### 6.2.2 Pack data to SelfSupDataSample in MMSelfSup

Before feeding data into model, MMSelfSup packs data into `SelfSupDataSample` in data pipeline. If you are not familiar with data pipeline, you can consult data transform. To pack data, we implement a data transform, called *PackSelfSupInputs*

```
class PackSelfSupInputs(BaseTransform):
    """Pack data into the format compatible with the inputs of algorithm.

    Required Keys:

    - img

    Added Keys:

    - data_sample
    - inputs

    Args:
        key (str): The key of image inputted into the model. Defaults to 'img'.
        algorithm_keys (List[str]): Keys of elements related
            to algorithms, e.g. mask. Defaults to [].
        pseudo_label_keys (List[str]): Keys set to be the attributes of
            pseudo_label. Defaults to [].
        meta_keys (List[str]): The keys of meta info of an image.
            Defaults to [].
    """

    def __init__(self,
                 key: Optional[str] = 'img',
                 algorithm_keys: Optional[List[str]] = [],
```

(continues on next page)

(continued from previous page)

```
                 pseudo_label_keys: Optional[List[str]] = [],
                 meta_keys: Optional[List[str]] = []) -> None:
        assert isinstance(key, str), f'key should be the type of str, instead \
            of {type(key)}.'

        self.key = key
        self.algorithm_keys = algorithm_keys
        self.pseudo_label_keys = pseudo_label_keys
        self.meta_keys = meta_keys

    def transform(self,
                  results: Dict) -> Dict[torch.Tensor, SelfSupDataSample]:
        """Method to pack the data.

        Args:
            results (Dict): Result dict from the data pipeline.

        Returns:
            Dict:

            - 'inputs' (List[torch.Tensor]): The forward data of models.
            - 'data_sample' (SelfSupDataSample): The annotation info of the
                the forward data.
        """
        packed_results = dict()
        if self.key in results:
            img = results[self.key]
            # if img is not a list, convert it to a list
            if not isinstance(img, List):
                img = [img]
            for i, img_ in enumerate(img):
                if len(img_.shape) < 3:
                    img_ = np.expand_dims(img_, -1)
                img_ = np.ascontiguousarray(img_.transpose(2, 0, 1))
                img[i] = to_tensor(img_)
            packed_results['inputs'] = img

        data_sample = SelfSupDataSample()
        if len(self.pseudo_label_keys) > 0:
            pseudo_label = InstanceData()
            data_sample.pseudo_label = pseudo_label

        # gt_label, sample_idx, mask, pred_label will be set here
        for key in self.algorithm_keys:
            self.set_algorithm_keys(data_sample, key, results)

        # keys, except for gt_label, sample_idx, mask, pred_label, will be
        # set as the attributes of pseudo_label
        for key in self.pseudo_label_keys:
            # convert data to torch.Tensor
            value = to_tensor(results[key])
            setattr(data_sample.pseudo_label, key, value)
```

(continues on next page)

(continued from previous page)

```
        img_meta = {}
        for key in self.meta_keys:
            img_meta[key] = results[key]
        data_sample.set_metainfo(img_meta)
        packed_results['data_sample'] = data_sample

        return packed_results

    @classmethod
    def set_algorithm_keys(self, data_sample: SelfSupDataSample, key: str,
                           results: Dict) -> None:
        """Set the algorithm keys of SelfSupDataSample."""
        value = to_tensor(results[key])
        if key == 'sample_idx':
            sample_idx = InstanceData(value=value)
            setattr(data_sample, 'sample_idx', sample_idx)
        elif key == 'mask':
            mask = InstanceData(value=value)
            setattr(data_sample, 'mask', mask)
        elif key == 'gt_label':
            gt_label = LabelData(value=value)
            setattr(data_sample, 'gt_label', gt_label)
        elif key == 'pred_label':
            pred_label = LabelData(value=value)
            setattr(data_sample, 'pred_label', pred_label)
        else:
            raise AttributeError(f'{key} is not a attribute of \
                SelfSupDataSample')
```

`algorithm_keys` are these attributes, except for `pseudo_label`, in SelfSupDataSample and `pseudo_label_keys` are these sub-keys in pseudo_label of SelfSupDataSample. Thank you for reading the whole tutorial. If you have any problems, you can raise an issue in GitHub, and we will reach you as soon as possible.

## 6.3 Models



Model can be seen as a feature extractor or loss generator for each algorithm. In MMSelfSup, it mainly contains the following fix parts,

- algorithms, containing the full modules of a model and all sub-modules will be constructed in algorithms.
- backbones, containing the backbones for each algorithm, e.g. ViT for MAE, and Swim Transformer for SimMIM.
- necks, some specifial modules, such as decoder, appended directly to the output of the backbone.
- heads, some specifial modules, such as mlp layers, appended to the output of the backbone or neck.

- memories, some memory banks or queues in some algorithms, e.g. MoCo v1/v2.
- losses, used to compute the loss between the predicted output and the target.
- target_generators, generating targets for self-supervised learning optimization, such as HOG, extracted features from other modules(DALL-E, CLIP), etc.

### 6.3.1 Overview of modules in MMSelfSup

First, we will give an overview about existing modules in MMSelfSup. They will be displayed according to the categories described above.

### 6.3.2 Construct algorithms from sub-modules

Just as shown in above table, each algorithm is a combination of backbone, neck, head, loss and memories. You are free to use these existing modules to build your own algorithms. If some customized modules are required, you should follow *add_modules* to meet your own need. MMSelfSup provides a base model, called `BaseModel`, and all algorithms should inherit this base model. And all sub-modules, except for memories, will be built in the base model, during the initialization of each algorithm. Memories will be built in the `__init__` of each specific algorithm. And loss will be built when building the head.

```
class BaseModel(_BaseModel):

    def __init__(self,
                 backbone: dict,
                 neck: Optional[dict] = None,
                 head: Optional[dict] = None,
                 target_generator: Optional[dict] = None,
                 pretrained: Optional[str] = None,
                 data_preprocessor: Optional[Union[dict, nn.Module]] = None,
                 init_cfg: Optional[dict] = None):

        if pretrained is not None:
            init_cfg = dict(type='Pretrained', checkpoint=pretrained)

        if data_preprocessor is None:
            data_preprocessor = {}
        # The build process is in MMEngine, so we need to add scope here.
        data_preprocessor.setdefault('type',
                                     'mmselfsup.SelfSupDataPreprocessor')

        super().__init__(
            init_cfg=init_cfg, data_preprocessor=data_preprocessor)

        self.backbone = MODELS.build(backbone)

        if neck is not None:
            self.neck = MODELS.build(neck)

        if head is not None:
            self.head = MODELS.build(head)
```

Just as shown above, you should provide the config to build the backbone, but neck and head are optional. In addition to building your algorithm, you should overwrite some abstract functions in the base model to get the correct results, which we will discuss in the following section.

### 6.3.3 Overview these abstract functions in base model

The `forward` function is the entrance to the results. However, it is different from the default `forward` function in most PyTorch code, which only has one mode. You will mess all your logic in the `forward` function, limiting the scalability. Just as shown in the code below, `forward` function in MMSelfSup has three modes, i) tensor, ii) loss and iii) predict.

```
def forward(self,
                batch_inputs: torch.Tensor,
                data_samples: Optional[List[SelfSupDataSample]] = None,
                mode: str = 'tensor'):
    if mode == 'tensor':
        feats = self.extract_feat(batch_inputs)
        return feats
    elif mode == 'loss':
        return self.loss(batch_inputs, data_samples)
    elif mode == 'predict':
        return self.predict(batch_inputs, data_samples)
    else:
        raise RuntimeError(f'Invalid mode "{mode}".')
```

- tensor, if the mode is `tensor`, the forward function will return the extracted features for images. You should overwrite the `extract_feat` to implement your customized extracting process.
- loss, if the mode is `loss`, the forward function will return the loss between the prediction and the target. You should overview the `loss` to implement your customized loss function.
- predict, if the mode is `predict`, the forward function will return the prediction, e.g. the predicted label, from your algorithm. If should also overwrite the `predict` function.

Now we have introduce the basic components related to models in MMSelfSup, if you want to dive in , please refer the API doc of each algorithm.

## 6.4 Datasets



The `datasets` folder under `mmselfsup` contains all kinds of modules, related to loading data. It can be roughly split into three parts, namely,

- cutomized datasets to read images
- cutomized dataset samplers to read index before loading images
- data transforms, e.g. `RandomResizedCrop`, to augment data before feeding into models.

In this tutorial, we will explain the above three parts in details.

### 6.4.1 Datasets

OpenMMLab provides a lot of off-the-shelf datasets, and all these datasets inherit the BaseDataset implemented in MMEngine. To have a full knowledge about all these functionalities implemented in `BaseDataset`, we recommend interested readers to refer to the documents in MMEngine. `ImageNet`, `ADE20KDataset` and `CocoDataset` are the three commonly used datasets `MMSelfSup`. Before using them, you should refactor your local folder according to the following format.

#### Refactor your datasets

To use these existing datasets, you need to refactor your datasets into following dataset format.

```
mmselfsup
├── mmselfsup
├── tools
├── configs
├── docs
├── data
│   ├── imagenet
│   │   ├── meta
│   │   ├── train
│   │   ├── val
│   │
│   │── ade
│   │   ├── ADEChallengeData2016
│   │   │   ├── annotations
│   │   │   │   ├── training
│   │   │   │   ├── validation
│   │   │   ├── images
│   │   │   │   ├── training
│   │   │   │   ├── validation
│   │
│   │── coco
│   │   ├── annotations
│   │   ├── train2017
│   │   ├── val2017
│   │   ├── test2017
```

For more details about the annotation files and the structure of each subfolder, you can consult MMClassfication, MMSegmentation and MMDetection.

#### Use datasets from other MM-repos in your config

```
# Use ImageNet dataset from MMClassification
# Use ImageNet in your dataloader
# For simplicity, we only provide the config related to importting ImageNet
# from MMClassification, instead of the full configuration for the dataloader.
# The ``mmcls`` prefix tells the ``Registry`` to search ``ImageNet`` in
# MMClassification
train_dataloader=dict(dataset=dict(type='mmcls.ImageNet', ...), ...)
```

```
# Use ADE20KDataset dataset from MMSegmentation
# Use ADE20KDataset in your dataloader
# For simplicity, we only provide the config related to importting ADE20KDataset
# from MMSegmentation, instead of the full configuration for the dataloader.
# The ``mmseg`` prefix tells the ``Registry`` to search ``ADE20KDataset`` in
# MMSegmentation
train_dataloader=dict(dataset=dict(type='mmseg.ADE20KDataset', ...), ...)
```

```
# Use CocoDataset in your dataloader
# For simplicity, we only provide the config related to importting CocoDataset
# from MMDetection, instead of the full configuration for the dataloader.
# The ``mmdet`` prefix tells the ``Registry`` to search ``CocoDataset`` in
# MMDetection
train_dataloader=dict(dataset=dict(type='mmdet.CocoDataset', ...), ...)
```

```
# Use dataset in MMSelfSup, for example ``DeepClusterImageNet``
train_dataloader=dict(dataset=dict(type='DeepClusterImageNet', ...), ...)
```

Till now, we have introduced two key steps, in order to use existing datasets successfully. We hope you can grasp the basic idea about how to use datasets in `MMSelfSup`. If you want to create you customized datasets, you can refer to another useful document, *add_datasets*.

### 6.4.2 Samplers

In pytorch, `Sampler` is used to sample the index of data before loading. `MMEngine` has already implemented `DefaultSampler` and `InfiniteSampler`. In most situation, we can directly use them, instead of implementing customized sampler. But the `DeepClusterSampler` is a special case, in which we implement the unique index sampling logic. We recommend interested user to refer to the API doc for more details about this sampler. If you want to implement your customized sampler, you can follow `DeepClusterSampler`and implement it under the folder of `samplers`.

### 6.4.3 Transforms

In short, `transform` refer to data augmentation in `MM-repos` and we compose a series of transforms into a list, called `pipeline`. `MMCV` already provides some useful transforms, covering most of scenarios. But every `MM-repo` defines their own transforms, following the User Guide in `MMCV`. Concretely, every customized dataset: i) inherits BaseTransform, ii) overwrite the `transform` function and implement your key logic in it. In MMSelfSup, we implement these `transforms` below:

For interested users, you can refer to the API doc to have a full understanding of these transforms. Now, we have introduced the basic concepts about transform. If you want to know how to use them in your config or implement your customed transforms, you can refer to *transforms* and *add_transforms*.

## 6.5 Transforms

- Transforms
  - *Overview of transforms*
  - *Introduction of* `MultiView`
  - *Introduction of* `PackSelfSupInputs`

### 6.5.1 Overview of transforms

We have introduced how to build a `Pipeline` in *add_transforms*. A `Pipeline` contains a series of `transforms`. There are three main categories of `transforms` in MMSelfSup:

1. Transforms about processing the data. The unique transforms in MMSelfSup are defined in processing.py, e.g. `RandomCrop`, `RandomResizedCrop` and `RandomGaussianBlur`. We may also use some transforms from other repositories, e.g. `LoadImageFromFile` from MMCV.
2. The transform wrapper for multiple views of an image. It is defined in wrappers.py.
3. The transform to pack data into a format compatible with the inputs of the algorithm. It is defined in formatting.py.

In summary, we implement these `transforms` below. The last two transforms will be introduced in detail.

### 6.5.2 Introduction of `MultiView`

We build a wrapper named *`MultiView`* for some algorithms e.g. MOCO, SimCLR and SwAV with multi-view image inputs. In the config file, we can define it as:

```
pipeline = [
     dict(type='MultiView',
          num_views=2,
          transforms=[
            [dict(type='Resize', scale=224),]
          ])
]
```

, which means that there are two views in the pipeline.

We can also define pipeline with different views like:

```
pipeline = [
     dict(type='MultiView',
          num_views=[2, 6],
          transforms=[
            [
              dict(type='Resize', scale=224)],
            [
              dict(type='Resize', scale=224),
              dict(type='RandomSolarize')],
          ])
]
```

This means that there are two pipelines, which contain 2 views and 6 views, respectively. More examples can be found in imagenet_mocov1.py, imagenet_mocov2.py and imagenet_swav_mcrop-2-6.py etc.

### 6.5.3 Introduction of PackSelfSupInputs

We build a class named *PackSelfSupInputs* to pack data into a format compatible with the inputs of an algorithm. This transform is usually put at the end of the pipeline like:

```
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='MultiView', num_views=2, transforms=[view_pipeline]),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]
```

## 6.6 Evaluation



### 6.6.1 Evaluation in MMEngine

During model validation and testing, quantitative evaluation is often required. `Metric` and `Evaluator` have been implemented in MMEngine to perform this function. See MMEngine Doc.

Model evaluation is divided into online evaluation and offline evaluation.

#### Online evaluation

Online evaluation is used in `ValLoop` and `TestLoop`.

Take `ValLoop` for example:

```
...
class ValLoop(BaseLoop):
    ...
    def run(self) -> dict:
        """Launch validation."""
        self.runner.call_hook('before_val')
        self.runner.call_hook('before_val_epoch')
        self.runner.model.eval()
        for idx, data_batch in enumerate(self.dataloader):
            self.run_iter(idx, data_batch)

        # compute metrics
        metrics = self.evaluator.evaluate(len(self.dataloader.dataset))
        self.runner.call_hook('after_val_epoch', metrics=metrics)
        self.runner.call_hook('after_val')
```

(continues on next page)

(continued from previous page)

```
        return metrics

    @torch.no_grad()
    def run_iter(self, idx, data_batch: Sequence[dict]):
        ...
        self.runner.call_hook(
            'before_val_iter', batch_idx=idx, data_batch=data_batch)
        # outputs should be sequence of BaseDataElement
        with autocast(enabled=self.fp16):
            outputs = self.runner.model.val_step(data_batch)
        self.evaluator.process(data_samples=outputs, data_batch=data_batch)
        self.runner.call_hook(
            'after_val_iter',
            batch_idx=idx,
            data_batch=data_batch,
            outputs=outputs)
```

#### Offline evaluation

Offline evaluation uses the predictions saved in a file. In this case, since there is no `Runner`, we need to build the `Evaluator` and call `offline_evaluate()` function.

An example:

```
from mmengine.evaluator import Evaluator
from mmengine.fileio import load

evaluator = Evaluator(metrics=dict(type='Accuracy', top_k=(1, 5)))

data = load('test_data.pkl')
predictions = load('prediction.pkl')

results = evaluator.offline_evaluate(data, predictions, chunk_size=128)
```

### 6.6.2 Evaluation In MMSelfSup

During pretrain, validation and testing are not included, so it is no need to use evaluation.

During benchmark, the pre-trained models need other downstream tasks to evaluate the performance, e.g. classification, detection, segmentation, etc. It is recommended to run downstream tasks with other OpenMMLab repos, such as MMClassification or MMDetection, which have already implemented their own evaluation functionalities.

But MMSelfSup also implements some custom evaluation functionalities to support downstream tasks, shown as below:

- `knn_classifier()`

It compute accuracy of knn classifier predictions, and is used in KNN evaluation.

```
...
top1, top5 = knn_classifier(train_feats, train_labels, val_feats,
                                        val_labels, k, args.temperature)
...
```

- ResLayerExtraNorm

It add extra norm to original ResLayer, and is used in mmdetection benchmark config.

```
model = dict(
    backbone=...,
    roi_head=dict(
        shared_head=dict(
            type='ResLayerExtraNorm',
            norm_cfg=norm_cfg,
            norm_eval=False,
            style='pytorch')))
```

### 6.6.3 Customize Evaluation

Custom Metric and Evaluator are also supported, see MMEngine Doc

## 6.7 Engine

### 6.7.1 Hook

#### Introduction

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 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.

#### Default hooks

The following common hooks are already reigistered by default, which is implemented through `register_default_hooks` in MMEngine:

#### Common Hooks implemented in MMEngine

Some hooks have been already implemented in MMEngine, they are:

#### Hooks implemented in MMSelfsup

Some hooks have been already implemented in MMSelfsup, they are:

- *DeepClusterHook*
- *DenseCLHook*
- *ODCHook*
- *SimSiamHook*
- *SwAVHook*
- ……

An example:

Take *DenseCLHook* for example, this hook includes `loss_lambda` warmup in DenseCL.

`loss_lambda` is loss weight for the single and dense contrastive loss. Defaults to 0.5.

```
losses = dict()
losses['loss_single'] = loss_single * (1 - self.loss_lambda)
losses['loss_dense'] = loss_dense * self.loss_lambda
```

DenseCLHook is implemented as follows:

```
...
@HOOKS.register_module()
class DenseCLHook(Hook):
...
    def before_train_iter(self,
                          runner,
                          batch_idx: int,
                          data_batch: Optional[Sequence[dict]] = None) -> None:
...
        cur_iter = runner.iter
        if cur_iter >= self.start_iters:
            get_model(runner.model).loss_lambda = self.loss_lambda
        else:
            get_model(runner.model).loss_lambda = 0.
```

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

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

such as using DenseCLHook, start_iters is 500:

```
custom_hooks = [
    dict(type='DenseCLHook', start_iters=500)
]
```

### 6.7.2 Optimizer

We will introduce Optimizer section through 3 different parts: Optimizer, Optimizer wrapper, and Constructor.

#### Optimizer

##### Customize optimizer supported by PyTorch

We have already supported all the optimizers implemented by PyTorch, see mmengine/optim/optimizer/builder.py. 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)
```

### 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_cfg in optimizer.

For example, in MAE, we do not want to apply weight decay to the parameters of ln, bias, pos_embed, mask_token and cls_token, so we can use following config file:

```
optimizer = dict(
    type='AdamW', lr=1.5e-4 * 4096 / 256, betas=(0.9, 0.95), weight_decay=0.05)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    paramwise_cfg=dict(
        custom_keys={
            'ln': dict(decay_mult=0.0),
            'bias': dict(decay_mult=0.0),
            'pos_embed': dict(decay_mult=0.),
            'mask_token': dict(decay_mult=0.),
            'cls_token': dict(decay_mult=0.)
        }))
```

### Implemented optimizers in MMSelfsup

- *LARS*

In addition to optimizers implemented by PyTorch, we also implement a customized *LARS* in mmselfsup/engine/optimizers/lars.py. It implements layer-wise adaptive rate scaling for SGD.

```
optimizer = dict(type='LARS', lr=4.8, momentum=0.9, weight_decay=1e-6)
```

### Optimizer wrapper

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

### Gradient clipping

Currently we support clip_grad option in optim_wrapper, and you can refer to OptimWrapper and PyTorch Documentationfor more arguments . Here is an example:

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(
    type='OptimWrapper',
        optimizer=optimizer,
    clip_grad=dict(
        max_norm=0.2,
        norm_type=2))
# norm_type: type of the used p-norm, here norm_type is 2.
```

If clip_grad is not None, it will be the arguments of torch.nn.utils.clip_grad.clip_grad_norm_().

### 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:

```
train_dataloader = dict(batch_size=64)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    accumulative_counts=4)
```

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

```
train_dataloader = dict(batch_size=256)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    accumulative_counts=1)
```

### Automatic mixed precision(AMP) training

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(type='AmpOptimWrapper', optimizer=optimizer)
```

The default setting of loss_scale of AmpOptimWrapper is dynamic.

### Constructor

The constructor aims to build optimizer, optimizer wrapper and customize hyper-parameters of different layers. The key `paramwise_cfg` of `optim_wrapper` in configs controls this customization.

### Constructors implemented in MMSelfsup

- *LearningRateDecayOptimWrapperConstructor*

`LearningRateDecayOptimWrapperConstructor` sets different learning rates for different layers of backbone. Note: Currently, this optimizer constructor is built for ViT , Swin and MixMIN.

An example:

```
optim_wrapper = dict(
    type='AmpOptimWrapper',
    optimizer=dict(
        type='AdamW', lr=5e-3, model_type='swin', layer_decay_rate=0.9),
    clip_grad=dict(max_norm=5.0),
    paramwise_cfg=dict(
        norm_decay_mult=0.0,
        bias_decay_mult=0.0,
        custom_keys={
            '.absolute_pos_embed': dict(decay_mult=0.0),
            '.relative_position_bias_table': dict(decay_mult=0.0)
        }),
    constructor='mmselfsup.LearningRateDecayOptimWrapperConstructor')
```

Note: `paramwise_cfg` only supports the customization of `weight_decay` in `LearningRateDecayOptimWrapperConstructor`.

## 6.8 Conventions

Please check the following conventions if you would like to modify MMSelfSup as your own project.

### 6.8.1 Losses

When the algorithm is implemented, the returned losses is supposed to be `dict` type.

Take `MAE` as an example:

```
class MAE(BaseModel):
    """MAE.

    Implementation of `Masked Autoencoders Are Scalable Vision Learners
    <https://arxiv.org/abs/2111.06377>`_.
    """

    def extract_feat(self, inputs: List[torch.Tensor],
                     **kwarg) -> Tuple[torch.Tensor]:
        ...
```

(continues on next page)

(continued from previous page)

```
    def loss(self, inputs: List[torch.Tensor],
             data_samples: List[SelfSupDataSample],
             **kwargs) -> Dict[str, torch.Tensor]:
        """The forward function in training.

        Args:
            inputs (List[torch.Tensor]): The input images.
            data_samples (List[SelfSupDataSample]): All elements required
                during the forward function.

        Returns:
            Dict[str, torch.Tensor]: A dictionary of loss components.
        """
        # ids_restore: the same as that in original repo, which is used
        # to recover the original order of tokens in decoder.
        latent, mask, ids_restore = self.backbone(inputs[0])
        pred = self.neck(latent, ids_restore)
        loss = self.head(pred, inputs[0], mask)
        losses = dict(loss=loss)
        return losses
```

The `MAE.loss()` function will be called during model forward to compute the loss and return its value.

By default, only values whose keys contain `'loss'` will be back propagated, if your algorithm need more than one loss value, you could pack losses dict with several keys:

```
class YourAlgorithm(BaseModel):

    def loss():
        ...

        losses['loss_1'] = loss_1
        losses['loss_2'] = loss_2
```

CHAPTER

SEVEN

# COMPONENT CUSTOMIZATION

## 7.1 Add Modules

In this tutorial, we introduce the basic steps to create your customized modules. Before learning to create your customized modules, it is recommended to learn the basic concept of models in file *models.md*. You can customize all the components introduced in *models.md*, such as **backbone**, **neck**, **head** and **loss**.



### 7.1.1 Add a new backbone

Assume you are going to create a new backbone `NewBackbone`.

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

```
import torch.nn as nn

from mmselfsup.registry import MODELS


@MODELS.register_module()
class NewBackbone(nn.Module):

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

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

    def init_weights(self):
        pass
```

(continues on next page)

(continued from previous page)

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

2. Import the new backbone module in mmselfsup/models/backbones/__init__.py.

```
...
from .new_backbone import NewBackbone

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

3. Use it in your config file.

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

### 7.1.2 Add a new neck

You can write a new neck inherited from BaseModule from mmengine, and overwrite forward. We have a unified interface for weight initialization in mmengine, you can use init_cfg to specify the initialization function and arguments, or overwrite init_weights if you prefer customized initialization.

We include all necks in mmselfsup/models/necks. Assume you are going to create a new neck NewNeck.

1. Create a new file mmselfsup/models/necks/new_neck.py and implement NewNeck in it.

```
from mmengine.model import BaseModule

from mmselfsup.registry import MODELS


@MODELS.register_module()
class NewNeck(BaseModule):

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

    def forward(self, x):
        pass
```

You need to implement the forward function, which applies some operations on the output from the backbone and forwards the results to the head.

2. Import the new neck module in mmselfsup/models/necks/__init__.py.

```
...
from .new_neck import NewNeck

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

3. Use it in your config file.

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

### 7.1.3 Add a new head

You can write a new head inherited from `BaseModule` from mmengine, and overwrite `forward`.

We include all heads in `mmselfsup/models/heads`. Assume you are going to create a new head `NewHead`.

1. Create a new file `mmselfsup/models/heads/new_head.py` and implement `NewHead` in it.

```
from mmengine.model import BaseModule

from mmselfsup.registry import MODELS


@MODELS.register_module()
class NewHead(BaseModule):

    def __init__(self, loss, **kwargs):
        super().__init__()
        # build loss
        self.loss = MODELS.build(loss)
        # other specific initializations

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

You need to implement the `forward` function, which applies some operations on the output from the neck/backbone and computes the loss. Please note that the loss module should be built in the head module for the loss computation.

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

```
...
from .new_head import NewHead

__all__ = [
```

(continues on next page)

(continued from previous page)

```
    ...,
    'NewHead',
    ...
]
```

3. Use it in your config file.

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

### 7.1.4 Add a new loss

To add a new loss function, we mainly implement the `forward` function in the loss module. We should register the loss module as `MODELS` as well.

We include all losses in `mmselfsup/models/losses`. Assume you are going to create a new loss `NewLoss`.

1. Create a new file `mmselfsup/models/losses/new_loss.py` and implement `NewLoss` in it.

```
from mmengine.model import BaseModule

from mmselfsup.registry import MODELS


@MODELS.register_module()
class NewLoss(BaseModule):

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

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

2. Import the new loss module in `mmselfsup/models/losses/__init__.py`

```
...
from .new_loss import NewLoss

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

3. Use it in your config file.

```
model = dict(
    ...
    head=dict(
        ...
        loss=dict(
            type='NewLoss',
            ...),
        ...),
    ...
)
```

### 7.1.5 Combine all

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

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

```
from mmselfsup.registry import MODELS
from .base import BaseModel


@MODELS.register_module()
class NewAlgorithm(BaseModel):

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

    def extract_feat(self, inputs, **kwargs):
        pass

    def loss(self, inputs, data_samples, **kwargs):
        pass

    def predict(self, inputs, data_samples, **kwargs):
        pass
```

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

```
...
from .new_algorithm import NewAlgorithm

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

3. Use it in your config file.

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

## 7.2 Add Datasets

In this tutorial, we introduce the basic steps to create your customized dataset. Before learning to create your customized datasets, it is recommended to learn the basic concept of datasets in file *datasets.md*.

- *Add Datasets*
  - *Step 1: Creating the Dataset*
  - *Step 2: Add NewDataset to __init__py*
  - *Step 3: Modify the config file*

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

As for image pretraining, it is recommended to follow the format of MMClassification.

### 7.2.1 Step 1: Creating the Dataset

You could implement a new dataset class, inherited from `CustomDataset` from MMClassification for image pretraining.

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.

```python
from typing import List, Optional, Union

from mmcls.datasets import CustomDataset

from mmselfsup.registry import DATASETS


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

    IMG_EXTENSIONS = ('.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm', '.tif')

    def __init__(self,
                 ann_file: str = '',
                 metainfo: Optional[dict] = None,
                 data_root: str = '',
                 data_prefix: Union[str, dict] = '',
                 **kwargs) -> None:
        kwargs = {'extensions': self.IMG_EXTENSIONS, **kwargs}
```

(continues on next page)

(continued from previous page)

```
        super().__init__(
            ann_file=ann_file,
            metainfo=metainfo,
            data_root=data_root,
            data_prefix=data_prefix,
            **kwargs)

    def load_data_list(self) -> List[dict]:
        # Rewrite load_data_list() to satisfy your specific requirement.
        # The returned data_list could include any information you need from
        # data or transforms.

        # writing your code here
        return data_list
```

### 7.2.2 Step 2: Add NewDataset to __init__py

Then, add NewDataset in mmselfsup/dataset/__init__.py. If it is not imported, the NewDataset will not be registered successfully.

```
...
from .new_dataset import NewDataset

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

### 7.2.3 Step 3: Modify the config file

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

```
train_dataloader = dict(
    ...
    dataset=dict(
        type='NewDataset',
        data_root=your_data_root,
        ann_file=your_data_root,
        data_prefix=dict(img_path='train/'),
        pipeline=train_pipeline))
```

## 7.3 Add Transforms

In this tutorial, we introduce the basic steps to create your customized transforms. Before learning to create your customized transforms, it is recommended to learn the basic concept of transforms in file *transforms.md*.



### 7.3.1 Overview of Pipeline

`Pipeline` is an important component in `Dataset`, which is responsible for applying a series of data augmentations to images, such as `RandomResizedCrop`, `RandomFlip`, etc.

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

```
view_pipeline = [
    dict(type='RandomResizedCrop', size=224, backend='pillow'),
    dict(type='RandomFlip', prob=0.5),
    dict(
        type='RandomApply',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.8,
                contrast=0.8,
                saturation=0.8,
                hue=0.2)
        ],
        prob=0.8),
    dict(
        type='RandomGrayscale',
        prob=0.2,
        keep_channels=True,
        channel_weights=(0.114, 0.587, 0.2989)),
    dict(type='RandomGaussianBlur', sigma_min=0.1, sigma_max=2.0, prob=0.5),
]

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='MultiView', num_views=2, transforms=[view_pipeline]),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]
```

Every augmentation in the `Pipeline` receives a `dict` as input and outputs a `dict` containing the augmented image and other related information.

### 7.3.2 Creating a new transform in Pipeline

Here are the steps to create a new transform.

#### Step 1: Creating the transform

Write a new transform in processing.py and overwrite the `transform` function, which takes a `dict` as input:

```
@TRANSFORMS.register_module()
class NewTransform(BaseTransform):
    """Docstring for transform.
    """

    def transform(self, results: dict) -> dict:
        # apply transform
        return results
```

**Note:** For the implementation of transforms, you could apply functions in mmcv.

#### Step 2: Add NewTransform to __init__py

Then, add the transform to __init__.py.

```
...
from .processing import NewTransform, ...

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

#### Step 3: Modify the config file

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

```
view_pipeline = [
    dict(type='RandomResizedCrop', size=224, backend='pillow'),
    dict(type='RandomFlip', prob=0.5),
    # add `NewTransform`
    dict(type='NewTransform'),
    dict(
        type='RandomApply',
        transforms=[
            dict(
                type='ColorJitter',
                brightness=0.8,
                contrast=0.8,
                saturation=0.8,
                hue=0.2)
        ],
        prob=0.8),
    dict(
```

(continues on next page)

(continued from previous page)

```
        type='RandomGrayscale',
        prob=0.2,
        keep_channels=True,
        channel_weights=(0.114, 0.587, 0.2989)),
    dict(type='RandomGaussianBlur', sigma_min=0.1, sigma_max=2.0, prob=0.5),
]

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='MultiView', num_views=2, transforms=[view_pipeline]),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]
```

## 7.4 Customize Runtime



In this tutorial, we will introduce some methods about how to customize runtime settings for the project.

### 7.4.1 Loop

`Loop` means the workflow of training, validation or testing and we use `train_cfg`, `val_cfg` and `test_cfg` to build `Loop`.

E.g.:

```
# Use EpochBasedTrainLoop to train 200 epochs.
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=200)
```

MMEngine defines several basic loops. Users could implement customized loops if the defined loops are not satisfied.

### 7.4.2 Hook

Before learning to create your customized hooks, it is recommended to learn the basic concept of hooks in file *engine.md*.

#### Step 1: Create a new hook

Depending on your intention of this hook, you need to implement corresponding functions according to the hook point of your expectation.

For example, if you want to modify the value of a hyper-parameter according to the training iter and two other hyper-parameters after every train iter, you could implement a hook like:

```python
# Copyright (c) OpenMMLab. All rights reserved.
from typing import Optional, Sequence

from mmengine.hooks import Hook

from mmselfsup.registry import HOOKS
from mmselfsup.utils import get_model


@HOOKS.register_module()
class NewHook(Hook):
    """Docstring for NewHook.
    """

    def __init__(self, a: int, b: int) -> None:
        self.a = a
        self.b = b

    def before_train_iter(self,
                          runner,
                          batch_idx: int,
                          data_batch: Optional[Sequence[dict]] = None) -> None:
        cur_iter = runner.iter
        get_model(runner.model).hyper_parameter = self.a * cur_iter + self.b
```

#### Step 2: Import the new hook

Then we need to ensure NewHook imported. Assuming NewHook is in mmselfsup/engine/hooks/new_hook.py, modify mmselfsup/engine/hooks/__init__.py as below

```python
...
from .new_hook import NewHook

__all__ = [..., NewHook]
```

#### Step 3: Modify the config

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

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

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

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

### 7.4.3 Optimizer

Before customizing the optimizer config, it is recommended to learn the basic concept of optimizer in file *engine.md*.

Here is an example of SGD optimizer:

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
```

We support all optimizers of PyTorch. For more details, please refer to MMEngine optimizer document.

#### Optimizer Wrapper

Optimizer wrapper provides a unified interface for single precision training and automatic mixed precision training with different hardware. Here is an example of `optim_wrapper` setting:

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(type='OptimWrapper', optimizer=optimizer)
```

Besides, if you want to apply automatic mixed precision training, you could modify the config above like:

```
optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
optim_wrapper = dict(type='AmpOptimWrapper', optimizer=optimizer)
```

The default setting of `loss_scale` of `AmpOptimWrapper` is `dynamic`.

#### Constructor

The constructor aims to build optimizer, optimizer wrapper and customize hyper-parameters of different layers. The key `paramwise_cfg` of `optim_wrapper` in configs controls this customization.

The example and detailed information can be found in MMEngine optimizer document.

Besides, We could use `custom_keys` to set different hyper-parameters of different modules.

Here is the `optim_wrapper` example of MAE. The config below sets weight decay multiplication to be 0 of `pos_embed`, `mask_token`, `cls_token` modules and those layers whose name contains `ln` and `bias`. During training, the weight decay of these modules will be `weight_decay * decay_mult`.

```
optimizer = dict(
    type='AdamW', lr=1.5e-4 * 4096 / 256, betas=(0.9, 0.95), weight_decay=0.05)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    paramwise_cfg=dict(
        custom_keys={
            'ln': dict(decay_mult=0.0),
            'bias': dict(decay_mult=0.0),
            'pos_embed': dict(decay_mult=0.),
            'mask_token': dict(decay_mult=0.),
            'cls_token': dict(decay_mult=0.)
        }))
```

Furthermore, for some specific settings, we could use boolean type arguments to control the optimization process or parameters. For example, here is an example config of SimCLR:

```
optimizer = dict(type='LARS', lr=0.3, momentum=0.9, weight_decay=1e-6)
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=optimizer,
    paramwise_cfg=dict(
        custom_keys={
            'bn': dict(decay_mult=0, lars_exclude=True),
            'bias': dict(decay_mult=0, lars_exclude=True),
            # bn layer in ResNet block downsample module
            'downsample.1': dict(decay_mult=0, lars_exclude=True),
        }))
```

In LARS optimizer, we have `lars_exclude` to decide whether the named layers apply the LARS optimization methods or not.

### 7.4.4 Scheduler

Before customizing the scheduler config, it is recommended to learn the basic concept of scheduler in MMEngine document.

Here is an example of scheduler:

```
param_scheduler = [
    dict(
        type='LinearLR',
        start_factor=1e-4,
        by_epoch=True,
        begin=0,
        end=40,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=360,
        by_epoch=True,
        begin=40,
        end=400,
```

(continues on next page)

(continued from previous page)

```
        convert_to_iter_based=True)
]
```

**Note:** When you change the max_epochs in train_cfg, make sure that the args in param_scheduler are modified simultanuously.

CHAPTER

EIGHT

# MODEL ZOO STATISTICS

- Number of papers: 23
  - Algorithm: 23
- Number of checkpoints: 75
  - [Algorithm] *Bootstrap your own latent: A new approach to self-supervised Learning* (2 ckpts)
  - [Algorithm] *Deep clustering for unsupervised learning of visual features* (1 ckpts)
  - [Algorithm] *Dense contrastive learning for self-supervised visual pre-training* (2 ckpts)
  - [Algorithm] *Momentum Contrast for Unsupervised Visual Representation Learning* (1 ckpts)
  - [Algorithm] *Improved Baselines with Momentum Contrastive Learning* (2 ckpts)
  - [Algorithm] *An Empirical Study of Training Self-Supervised Vision Transformers* (13 ckpts)
  - [Algorithm] *Unsupervised Feature Learning via Non-Parametric Instance Discrimination* (2 ckpts)
  - [Algorithm] *Online deep clustering for unsupervised representation learning* (1 ckpts)
  - [Algorithm] *Unsupervised visual representation learning by context prediction* (2 ckpts)
  - [Algorithm] *Unsupervised representation learning by predicting image rotations* (2 ckpts)
  - [Algorithm] *A simple framework for contrastive learning of visual representations* (6 ckpts)
  - [Algorithm] *Exploring simple siamese representation learning* (4 ckpts)
  - [Algorithm] *Unsupervised Learning of Visual Features by Contrasting Cluster Assignments* (2 ckpts)
  - [Algorithm] *Masked Autoencoders Are Scalable Vision Learners* (11 ckpts)
  - [Algorithm] *SimMIM: A Simple Framework for Masked Image Modeling* (6 ckpts)
  - [Algorithm] *Barlow Twins: Self-Supervised Learning via Redundancy Reduction* (2 ckpts)
  - [Algorithm] *Context Autoencoder for Self-Supervised Representation Learning* (2 ckpts)
  - [Algorithm] *Masked Feature Prediction for Self-Supervised Visual Pre-Training* (2 ckpts)
  - [Algorithm] *BEiT: BERT Pre-Training of Image Transformers* (2 ckpts)
  - [Algorithm] *MILAN: Masked Image Pretraining on Language Assisted Representation* (3 ckpts)
  - [Algorithm] *BEiT v2: Masked Image Modeling with Vector-Quantized Visual Tokenizers* (2 ckpts)
  - [Algorithm] *EVA: Exploring the Limits of Masked Visual Representation Learning at Scale* (3 ckpts)
  - [Algorithm] *MixMIM: Mixed and Masked Image Modeling for Efficient Visual Representation Learning* (2 ckpts)

CHAPTER

NINE

# MODEL ZOO

All models and part of benchmark results are recorded below.



## 9.1 Benchmarks

### 9.1.1 ImageNet

ImageNet has multiple versions, but the most commonly used one is ILSVRC 2012. The classification results below are reported by linear evaluation or fine-tuning with pre-trained weights provided by various algorithms.

CHAPTER

TEN

# BARLOWTWINS

Barlow Twins: Self-Supervised Learning via Redundancy Reduction

## 10.1 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.

## 10.2 Models and Benchmarks

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.

### 10.2.1 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.

### ImageNet Nearest-Neighbor Classification

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

## 10.3 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

# ELEVEN

# BEIT

BEiT: BERT Pre-Training of Image Transformers

## 11.1 Abstract

We introduce a self-supervised vision representation model BEiT, which stands for Bidirectional Encoder representation from Image Transformers. Following BERT developed in the natural language processing area, we propose a masked image modeling task to pretrain vision Transformers. Specifically, each image has two views in our pre-training, i.e, image patches (such as 16x16 pixels), and visual tokens (i.e., discrete tokens). We first "tokenize" the original image into visual tokens. Then we randomly mask some image patches and fed them into the backbone Transformer. The pre-training objective is to recover the original visual tokens based on the corrupted image patches. After pre-training BEiT, we directly fine-tune the model parameters on downstream tasks by appending task layers upon the pretrained encoder. Experimental results on image classification and semantic segmentation show that our model achieves competitive results with previous pre-training methods. For example, base-size BEiT achieves 83.2% top-1 accuracy on ImageNet-1K, significantly outperforming from-scratch DeiT training (81.8%) with the same setup. Moreover, large-size BEiT obtains 86.3% only using ImageNet-1K, even outperforming ViT-L with supervised pre-training on ImageNet-22K (85.2%).

## 11.2 Models and Benchmarks

Here, we report the results of the model on ImageNet, the details are below:

## 11.3 Citation

```
@inproceedings{bao2022beit,
    title={{BE}iT: {BERT} Pre-Training of Image Transformers},
    author={Hangbo Bao and Li Dong and Songhao Piao and Furu Wei},
    booktitle={International Conference on Learning Representations},
    year={2022},
}
```

CHAPTER

# TWELVE

# BEIT V2

BEiT v2: Masked Image Modeling with Vector-Quantized Visual Tokenizers

## 12.1 Abstract

Masked image modeling (MIM) has demonstrated impressive results in self-supervised representation learning by recovering corrupted image patches. However, most existing studies operate on low-level image pixels, which hinders the exploitation of high-level semantics for representation models. In this work, we propose to use a semantic-rich visual tokenizer as the reconstruction target for masked prediction, providing a systematic way to promote MIM from pixel-level to semantic-level. Specifically, we propose vector-quantized knowledge distillation to train the tokenizer, which discretizes a continuous semantic space to compact codes. We then pretrain vision Transformers by predicting the original visual tokens for the masked image patches. Furthermore, we introduce a patch aggregation strategy which associates discrete image patches to enhance global semantic representation. Experiments on image classification and semantic segmentation show that BEiT v2 outperforms all compared MIM methods. On ImageNet-1K (224 size), the base-size BEiT v2 achieves 85.5% top-1 accuracy for fine-tuning and 80.1% top-1 accuracy for linear probing. The large-size BEiT v2 obtains 87.3% top-1 accuracy for ImageNet-1K (224 size) fine-tuning, and 56.7% mIoU on ADE20K for semantic segmentation.

## 12.2 Models and Benchmarks

During trainig, the `VQKD` target generator will download **VQ-KD** model automatically. Besides, You could also download **VQ-KD** model from this link manually.

Here, we report the results of the model on ImageNet, the details are below:

## 12.3 Citation

```
@article{beitv2,
    title={{BEiT v2}: Masked Image Modeling with Vector-Quantized Visual Tokenizers},
    author={Zhiliang Peng and Li Dong and Hangbo Bao and Qixiang Ye and Furu Wei},
    journal={ArXiv},
    year={2022}
}
```

CHAPTER

THIRTEEN

# BYOL

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

## 13.1 Abstract

**B**ootstrap **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.

## 13.2 Models and Benchmarks

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.

### 13.2.1 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.

#### 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.

### 13.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 13.2.3 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 config for details.

## 13.3 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

FOURTEEN

# CAE

Context Autoencoder for Self-Supervised Representation Learning

## 14.1 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.

## 14.2 Prerequisite

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

## 14.3 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:

## 14.4 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

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 Models and Benchmarks

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 Models and Benchmarks

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.

#### 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 config for details.

#### COCO2017

Please refer to config for details.

### 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 config for details.

## 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

# EVA

EVA: Exploring the Limits of Masked Visual Representation Learning at Scale

## 17.1 Abstract

We launch EVA, a vision-centric foundation model to explore the limits of visual representation at scale using only publicly accessible data. EVA is a vanilla ViT pre-trained to reconstruct the masked out image-text aligned vision features conditioned on visible image patches. Via this pretext task, we can efficiently scale up EVA to one billion parameters, and sets new records on a broad range of representative vision downstream tasks, such as image recognition, video action recognition, object detection, instance segmentation and semantic segmentation without heavy supervised training. Moreover, we observe quantitative changes in scaling EVA result in qualitative changes in transfer learning performance that are not present in other models. For instance, EVA takes a great leap in the challenging large vocabulary instance segmentation task: our model achieves almost the same state-of-the-art performance on LVISv1.0 dataset with over a thousand categories and COCO dataset with only eighty categories. Beyond a pure vision encoder, EVA can also serve as a vision-centric, multi-modal pivot to connect images and text. We find initializing the vision tower of a giant CLIP from EVA can greatly stabilize the training and outperform the training from scratch counterpart with much fewer samples and less compute, providing a new direction for scaling up and accelerating the costly training of multi-modal foundation models. To facilitate future research, we release all the code and models at this https URL.

## 17.2 Models and Benchmarks

Here, we report the results of the model, which is pre-trained on ImageNet-1k for 400 epochs, the details are below:

## 17.3 Citation

```
@article{fang2022eva,
  title={Eva: Exploring the limits of masked visual representation learning at scale},
  author={Fang, Yuxin and Wang, Wen and Xie, Binhui and Sun, Quan and Wu, Ledell and␣
↪Wang, Xinggang and Huang, Tiejun and Wang, Xinlong and Cao, Yue},
  journal={arXiv preprint arXiv:2211.07636},
  year={2022}
}
```

CHAPTER

EIGHTEEN

# MAE

Masked Autoencoders Are Scalable Vision Learners

## 18.1 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.

## 18.2 Models and Benchmarks

## 18.3 Evaluating MAE on Detection and Segmentation

If you want to evaluate your model on detection or segmentation task, we provide a script to convert the model keys from MMClassification style to timm style.

```
cd $MMSELFSUP
python tools/model_converters/mmcls2timm.py $src_ckpt $dst_ckpt
```

Then, using this converted ckpt, you can evaluate your model on detection task, following Detectron2 and on semantic segmentation task, following this project. Besides, using the unconverted ckpt, you can use MMSegmentation to evaluate your model.

## 18.4 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

# NINETEEN

# MASKFEAT

Masked Feature Prediction for Self-Supervised Visual Pre-Training

## 19.1 Abstract

We present Masked Feature Prediction (MaskFeat) for self-supervised pre-training of video models. Our approach first randomly masks out a portion of the input sequence and then predicts the feature of the masked regions. We study five different types of features and find Histograms of Oriented Gradients (HOG), a hand-crafted feature descriptor, works particularly well in terms of both performance and efficiency. We observe that the local contrast normalization in HOG is essential for good results, which is in line with earlier work using HOG for visual recognition. Our approach can learn abundant visual knowledge and drive large-scale Transformer-based models. Without using extra model weights or supervision, MaskFeat pre-trained on unlabeled videos achieves unprecedented results of 86.7% with MViT-L on Kinetics-400, 88.3% on Kinetics-600, 80.4% on Kinetics-700, 38.8 mAP on AVA, and 75.0% on SSv2. MaskFeat further generalizes to image input, which can be interpreted as a video with a single frame and obtains competitive results on ImageNet.

## 19.2 Models and Benchmarks

Here, we report the results of the model on ImageNet, the details are below:

## 19.3 Citation

```
@InProceedings{wei2022masked,
    author    = {Wei, Chen and Fan, Haoqi and Xie, Saining and Wu, Chao-Yuan and Yuille,␣
↪Alan and Feichtenhofer, Christoph},
    title     = {Masked Feature Prediction for Self-Supervised Visual Pre-Training},
    booktitle = {CVPR},
    year      = {2022},
}
```

CHAPTER

TWENTY

# MILAN

MILAN: Masked Image Pretraining on Language Assisted Representation

## 20.1 Abstract

Self-attention based transformer models have been dominating many computer vision tasks in the past few years. Their superb model qualities heavily depend on the excessively large labeled image datasets. In order to reduce the reliance on large labeled datasets, reconstruction based masked autoencoders are gaining popularity, which learn high quality transferable representations from unlabeled images. For the same purpose, recent weakly supervised image pretraining methods explore language supervision from text captions accompanying the images. In this work, we propose masked image pretraining on language assisted representation, dubbed as MILAN. Instead of predicting raw pixels or low level features, our pretraining objective is to reconstruct the image features with substantial semantic signals that are obtained using caption supervision. Moreover, to accommodate our reconstruction target, we propose a more efficient prompting decoder architecture and a semantic aware mask sampling mechanism, which further advance the transfer performance of the pretrained model. Experimental results demonstrate that MILAN delivers higher accuracy than the previous works. When the masked autoencoder is pretrained and finetuned on ImageNet-1K dataset with an input resolution of 224×224, MILAN achieves a top-1 accuracy of 85.4% on ViTB/16, surpassing previous state-of-the-arts by 1%. In the downstream semantic segmentation task, MILAN achieves 52.7 mIoU using ViT-B/16 backbone on ADE20K dataset, outperforming previous masked pretraining results by 4 points.

## 20.2 Models and Benchmarks

Here, we report the results of the model, which is pre-trained on ImageNet-1k for 400 epochs, the details are below:

## 20.3 Citation

```
@article{Hou2022MILANMI,
  title={MILAN: Masked Image Pretraining on Language Assisted Representation},
  author={Zejiang Hou and Fei Sun and Yen-Kuang Chen and Yuan Xie and S. Y. Kung},
  journal={ArXiv},
  year={2022}
}
```

CHAPTER

TWENTYONE

# MIXMIM

MixMIM: Mixed and Masked Image Modeling for Efficient Visual Representation Learning

## 21.1 Abstract

In this study, we propose Mixed and Masked Image Modeling (MixMIM), a simple but efficient MIM method that is applicable to various hierarchical Vision Transformers. Existing MIM methods replace a random subset of input tokens with a special [MASK] symbol and aim at reconstructing original image tokens from the corrupted image. However, we find that using the [MASK] symbol greatly slows down the training and causes training-finetuning inconsistency, due to the large masking ratio (e.g., 40% in BEiT). In contrast, we replace the masked tokens of one image with visible tokens of another image, i.e., creating a mixed image. We then conduct dual reconstruction to reconstruct the original two images from the mixed input, which significantly improves efficiency. While MixMIM can be applied to various architectures, this paper explores a simpler but stronger hierarchical Transformer, and scales with MixMIM-B, -L, and -H. Empirical results demonstrate that MixMIM can learn high-quality visual representations efficiently. Notably, MixMIM-B with 88M parameters achieves 85.1% top-1 accuracy on ImageNet-1K by pretraining for 600 epochs, setting a new record for neural networks with comparable model sizes (e.g., ViT-B) among MIM methods. Besides, its transferring performances on the other 6 datasets show MixMIM has better FLOPs / performance tradeoff than previous MIM methods

## 21.2 Models and Benchmarks

Here, we report the results of the model on ImageNet, the details are below:

## 21.3 Citation

```
@article{MixMIM2022,
  author  = {Jihao Liu, Xin Huang, Yu Liu, Hongsheng Li},
  journal = {arXiv:2205.13137},
  title   = {MixMIM: Mixed and Masked Image Modeling for Efficient Visual Representation␣
↪Learning},
  year    = {2022},
}
```

# CHAPTER TWENTYTWO

# MOCO V1

Momentum Contrast for Unsupervised Visual Representation Learning

## 22.1 Abstract

We present Momentum Contrast (MoCo) for unsupervised visual representation learning. From a perspective on contrastive learning as dictionary look-up, we build a dynamic dictionary with a queue and a moving-averaged encoder. This enables building a large and consistent dictionary on-the-fly that facilitates contrastive unsupervised learning. MoCo provides competitive results under the common linear protocol on ImageNet classification. More importantly, the representations learned by MoCo transfer well to downstream tasks.

## 22.2 Citation

```
@inproceedings{he2020momentum,
  title={Momentum contrast for unsupervised visual representation learning},
  author={He, Kaiming and Fan, Haoqi and Wu, Yuxin and Xie, Saining and Girshick, Ross},
  booktitle={CVPR},
  year={2020}
}
```

CHAPTER

TWENTYTHREE

# MOCO V2

Improved Baselines with Momentum Contrastive Learning

## 23.1 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.

## 23.2 Models and Benchmarks

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.

#### 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 config for details.

#### COCO2017

Please refer to config for details.

### 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 config for details.

## 23.3 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 TWENTYFOUR

# MOCO V3

An Empirical Study of Training Self-Supervised Vision Transformers

## 24.1 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.

## 24.2 Models and Benchmarks

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.

## 24.3 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

# TWENTYFIVE

# NPID

Unsupervised Feature Learning via Non-Parametric Instance Discrimination

## 25.1 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.

## 25.2 Results and 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 (%).

#### 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.

#### 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.

### 25.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 25.2.3 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 config for details.

## 25.3 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

# TWENTYSIX

# ODC

Online Deep Clustering for Unsupervised Representation Learning

## 26.1 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.

## 26.2 Models and Benchmarks

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.

### 26.2.1 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.

## 26.3 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

TWENTYSEVEN

# RELATIVE LOCATION

Unsupervised Visual Representation Learning by Context Prediction

## 27.1 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.

## 27.2 Models and Benchmarks

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.

### 27.2.1 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.

#### 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.

### 27.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 27.2.3 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 config for details.

## 27.3 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

# TWENTYEIGHT

# ROTATION PREDICTION

Unsupervised Representation Learning by Predicting Image Rotation

## 28.1 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.

## 28.2 Models and Benchmarks

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 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.

#### 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.

### 28.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 28.2.3 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 config for details.

## 28.3 Citation

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

CHAPTER

# TWENTYNINE

# SIMCLR

A Simple Framework for Contrastive Learning of Visual Representations

## 29.1 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.

## 29.2 Results and 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.

### 29.2.1 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.

#### 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.

### 29.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 29.2.3 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 config for details.

## 29.3 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

THIRTY

# SIMMIM

SimMIM: A Simple Framework for Masked Image Modeling

## 30.1 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 .

## 30.2 Models and Benchmarks

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

## 30.3 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

THIRTYONE

# SIMSIAM

Exploring Simple Siamese Representation Learning

## 31.1 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.

## 31.2 Models and Benchmarks

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.

### 31.2.1 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.

#### 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.

### 31.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 31.2.3 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 config for details.

## 31.3 Citation

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

CHAPTER

# THIRTYTWO

# SWAV

Unsupervised Learning of Visual Features by Contrasting Cluster Assignments

## 32.1 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.

## 32.2 Models and Benchmarks

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.

### 32.2.1 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.

#### 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.

### 32.2.2 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 config for details.

#### COCO2017

Please refer to config for details.

### 32.2.3 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 config for details.

## 32.3 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

THIRTYTHREE

# MIGRATION



## 33.1 Migration from MMSelfSup 0.x

> **Warning:** MMSelfSup 1.x depends on some new packages, you should create a new environment for MMSelfSup 1.x even if you have a well-rounded MMSelfSup 0.x environment before. Please refer to the *install tutorial* for required packages installation.

We introduce some modifications of MMSelfSup 1.x, to help users to migrate their projects based on MMSelfSup 0.x to 1.x smoothly.

Three important packages are listed below,

1. MMEngine: MMEngine is the base of all OpenMMLab 2.0 repos. Some modules, which are not specific to Computer Vision, are migrated from MMCV to this repo.
2. MMCV: The computer vision package of OpenMMLab. This is not a new dependency, but you need to upgrade it to above `2.0.0rc1` version.
3. MMClassification: The image classification package of OpenMMLab. This is not a new dependency, but you need to upgrade it to above `1.0.0rc0` version.

## 33.2 Config

This section illustrates the changes of our config files in `_base_` folder, which includes three parts

- Datasets: `mmselfsup/configs/selfsup/_base_/datasets`
- Models: `mmselfsup/configs/selfsup/_base_/models`
- Schedules: `mmselfsup/configs/selfsup/_base_/schedules`

### 33.2.1 Datasets

In **MMSelfSup 0.x**, we use key `data` to summarize all information, such as `samples_per_gpu`, `train`, `val`, etc.

In **MMSelfSup 1.x**, we separate `train_dataloader`, `val_dataloader` to summarize information correspodingly and the key `data` has been **removed**.

```
data = dict(
    samples_per_gpu=32,  # total 32*8(gpu)=256
    workers_per_gpu=4,
    train=dict(
        type=dataset_type,
        data_source=dict(
            type=data_source,
            data_prefix='data/imagenet/train',
            ann_file='data/imagenet/meta/train.txt',
        ),
        num_views=[1, 1],
        pipelines=[train_pipeline1, train_pipeline2],
        prefetch=prefetch,
    ),
    val=...)
```

```
train_dataloader = dict(
    batch_size=32,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(type='DefaultSampler', shuffle=True),
    collate_fn=dict(type='default_collate'),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='meta/train.txt',
        data_prefix=dict(img_path='train/'),
        pipeline=train_pipeline))
val_dataloader = ...
```

Besides, we **remove** the key of `data_source` to keep the pipeline format consistent with that in other OpenMMLab projects. Please refer to *Config* for more details.

Changes in **`pipeline`**:

Take MAE as an example of `pipeline`:

```
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='RandomResizedCrop',
        size=224,
        scale=(0.2, 1.0),
        backend='pillow',
        interpolation='bicubic'),
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackSelfSupInputs', meta_keys=['img_path'])
]
```

### 33.2.2 Models

In the config of models, there are two main different parts from MMSeflSup 0.x.

1. There is a new key called `data_preprocessor`, which is responsible for preprocessing the data, like normalization, channel conversion, etc. For example:

```
model = dict(
    type='MAE',
    data_preprocessor=dict(
        mean=[123.675, 116.28, 103.53],
        std=[58.395, 57.12, 57.375],
        bgr_to_rgb=True),
    backbone=...,
    neck=...,
    head=...,
    init_cfg=...)
```

**NOTE:** `data_preprocessor` can be defined outside the model dict, which has higher priority than it in model dict.

For example bwlow, `Runner` would build `data_preprocessor` based on `mean=[123.675, 116.28, 103.53]` and `std=[58.395, 57.12, 57.375]`, but omit the `127.5` of mean and std.

```
data_preprocessor=dict(
    mean=[123.675, 116.28, 103.53],
    std=[58.395, 57.12, 57.375],
    bgr_to_rgb=True)
model = dict(
    type='MAE',
    data_preprocessor=dict(
        mean=[127.5, 127.5, 127.5],
        std=[127.5, 127.5, 127.5],
        bgr_to_rgb=True)
    backbone=...,
    neck=...,
    head=...,
    init_cfg=...)
```

Related codes in MMEngine: Runner could get key `cfg.data_preprocessor` in `cfg` directly and merge it to `cfg.model`.

2. There is a new key `loss` in `head` in MMSelfSup 1.x, to determine the loss function of the algorithm. For example:

```
model = dict(
    type='MAE',
    data_preprocessor=...,
    backbone=...,
    neck=...,
    head=dict(
        type='MAEPretrainHead',
        norm_pix=True,
        patch_size=16,
        loss=dict(type='MAEReconstructionLoss')),
    init_cfg=...)
```

### 33.2.3 Schedules

1. Changes in **optimizer** and **optimizer_config**:

- Now we use `optim_wrapper` field to specify all configuration about the optimization process. And the `optimizer` is a sub field of `optim_wrapper` now.
- `paramwise_cfg` is also a sub field of `optim_wrapper`, instead of `optimizer`.
- `optimizer_config` is removed now, and all configurations of it are moved to `optim_wrapper`.
- `grad_clip` is renamed to `clip_grad`.

```
optimizer = dict(
    type='AdamW',
    lr=0.0015,
    weight_decay=0.3,
    paramwise_options = dict(
        norm_decay_mult=0.0,
        bias_decay_mult=0.0,
    ))
optimizer_config = dict(grad_clip=dict(max_norm=1.0))
```

```
optim_wrapper = dict(
    optimizer=dict(type='AdamW', lr=0.0015, weight_decay=0.3),
    paramwise_cfg = dict(
        norm_decay_mult=0.0,
        bias_decay_mult=0.0,
    ),
    clip_gard=dict(max_norm=1.0),
)
```

2. Changes in **lr_config**:

- The `lr_config` field is removed and we use new `param_scheduler` to replace it.
- The `warmup` related arguments are removed, since we use a separate lr scheduler to implement this functionality. These introduced lr schedulers are very flexible, and you can use them to design many kinds of learning rate / momentum curves. See the tutorial for more details.

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

(continues on next page)

(continued from previous page)

```
    min_lr=0,
    warmup='linear',
    warmup_iters=5,
    warmup_ratio=0.01,
    warmup_by_epoch=True)
```

```
param_scheduler = [
    # warmup
    dict(
        type='LinearLR',
        start_factor=0.01,
        by_epoch=True,
        end=5,
        # Update the learning rate after every iters.
        convert_to_iter_based=True),
    # main learning rate scheduler
    dict(type='CosineAnnealingLR', by_epoch=True, begin=5 end=200),
]
```

1. Changes in **runner**:

Most configuration in the original runner field is moved to train_cfg, val_cfg and test_cfg, which configure the loop in training, validation and test.

```
runner = dict(type='EpochBasedRunner', max_epochs=200)
```

```
train_cfg = dict(by_epoch=True, max_epochs=200)
```

### 33.2.4 Runtime settings

1. Changes in **checkpoint_config** and **log_config**:

The checkpoint_config are moved to default_hooks.checkpoint and the log_config are moved to default_hooks.logger.

And we move many hooks settings from the script code to the default_hooks field in the runtime configuration.

```
default_hooks = dict(
    # record the time of every iterations.
    timer=dict(type='IterTimerHook'),
    # print log every 100 iterations.
    logger=dict(type='LoggerHook', interval=100),
    # enable the parameter scheduler.
    param_scheduler=dict(type='ParamSchedulerHook'),
    # save checkpoint per epoch, and automatically save the best checkpoint.
    checkpoint=dict(type='CheckpointHook', interval=1, save_best='auto'),
    # set sampler seed in distributed evrionment.
    sampler_seed=dict(type='DistSamplerSeedHook'),
    # validation results visualization, set True to enable it.
    visualization=dict(type='VisualizationHook', enable=False),
)
```

In addition, we splited the original logger to logger and visualizer. The logger is used to record information and the visualizer is used to show the logger in different backends, like terminal, TensorBoard and Wandb.

```
log_config = dict(
    interval=100,
    hooks=[
        dict(type='TextLoggerHook'),
        dict(type='TensorboardLoggerHook'),
    ])
```

```
default_hooks = dict(
    ...
    logger=dict(type='LoggerHook', interval=100),
)
visualizer = dict(
    type='SelfSupVisualizer',
    vis_backends=[dict(type='LocalVisBackend'), dict(type='TensorboardVisBackend')],
)
```

2. Changes in **load_from** and **resume_from**:

- The resume_from is removed. And we use resume and load_from to replace it.
  - If resume=True and load_from is not None, resume training from the checkpoint in load_from.
  - If resume=True and load_from is None, try to resume from the latest checkpoint in the work directory.
  - If resume=False and load_from is not None, only load the checkpoint, not resume training.
  - If resume=False and load_from is None, do not load nor resume.

3. Changes in **dist_params**:

The dist_params field is a sub field of env_cfg now. And there are some new configurations in the env_cfg.

```
env_cfg = dict(
    # whether to enable cudnn benchmark
    cudnn_benchmark=False,
    # set multi process parameters
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0),
    # set distributed parameters
    dist_cfg=dict(backend='nccl'),
)
```

4. Changes in **workflow**: workflow related functionalities are **removed**.

5. New field **visualizer**:

The visualizer is a new design in OpenMMLab 2.0 architecture. We use a visualizer instance in the runner to handle results & log visualization and save to different backends. See the MMEngine visualization tutorial for more details.

```
visualizer = dict(
    type='SelfSupVisualizer',
    vis_backends=[
        dict(type='LocalVisBackend'),
        # Uncomment the below line to save the log and visualization results to␣
↪TensorBoard.
        # dict(type='TensorboardVisBackend')
```

(continues on next page)

(continued from previous page)

```
    ]
)
```

1. New field **default_scope**: The start point to search module for all registries. The `default_scope` in MMSelfSup is `mmselfsup`. See the registry tutorial for more details.

## 33.3 Package

The table below records the general modification of the folders and files.

CHAPTER

# THIRTYFOUR

# MMSELFSUP.DATASETS

## 34.1 datasets

class mmselfsup.datasets.**DeepClusterImageNet**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: Union[str, dict] = ''*, ***kwargs*)

ImageNet Dataset.

The dataset inherit ImageNet dataset from MMClassification as the DeepCluster and Online Deep Clustering algorithm need to initialize clustering labels and assign them during training.

**Parameters**

- **ann_file** (*str*) – Annotation file path. Defaults to None.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str*) – The root directory for `data_prefix` and `ann_file`. Defaults to None.
- **data_prefix** (*str | dict*) – Prefix for training data. Defaults to None.
- ****kwargs** – Other keyword arguments in `CustomDataset` and `BaseDataset`.

**assign_labels**(*labels: list*) → None

Assign new labels to *self.clustering_labels*.

**Parameters** **labels** (*list*) – The new labels.

**Returns** None

**prepare_data**(*idx: int*) → Any

Get data processed by `self.pipeline`.

**Parameters** **idx** (*int*) – The index of `data_info`.

**Returns** Depends on `self.pipeline`.

**Return type** Any

class mmselfsup.datasets.**ImageList**(*ann_file: str*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: Union[str, dict] = ''*, ***kwargs*)

The dataset 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.

An annotation file should be provided, and each line indicates a sample:

The sample files:

```
data_prefix/
├── folder_1
│   ├── xxx.png
│   ├── xxy.png
│   └── ...
└── folder_2
    ├── 123.png
    ├── nsdf3.png
    └── ...
```

1. If data is labeled, the annotation file (the first column is the image path and the second column is the index of category):

```
    folder_1/xxx.png 0
    folder_1/xxy.png 1
    folder_2/123.png 5
    folder_2/nsdf3.png 3
    ...

2. If data is unlabeled, the annotation file is: ::

    folder_1/xxx.png
    folder_1/xxy.png
    folder_2/123.png
    folder_2/nsdf3.png
    ...
```

**Parameters**

- **ann_file** (*str*) – Annotation file path.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.
- **data_root** (*str*) – The root directory for `data_prefix` and `ann_file`. Defaults to None.
- **data_prefix** (*str | dict*) – Prefix for training data. Defaults to None.
- ****kwargs** – Other keyword arguments in `CustomDataset` and `BaseDataset`.

**load_data_list**() → List[dict]

Rewrite load_data_list() function for supporting annotation files with unlabeled data.

**Returns** A list of data information.

**Return type** List[dict]

**class** mmselfsup.datasets.**Places205**(*ann_file: str = ''*, *metainfo: Optional[dict] = None*, *data_root: str = ''*, *data_prefix: Union[str, dict] = ''*, ***kwargs*)

Places205 Dataset.

The dataset supports two kinds of annotation format. More details can be found in `CustomDataset`.

**Parameters**

- **ann_file** (*str*) – Annotation file path. Defaults to None.
- **metainfo** (*dict, optional*) – Meta information for dataset, such as class information. Defaults to None.

- **data_root** (*str*) – The root directory for data_prefix and ann_file. Defaults to None.
- **data_prefix** (*str | dict*) – Prefix for training data. Defaults to None.
- ****kwargs** – Other keyword arguments in CustomDataset and BaseDataset.

mmselfsup.datasets.**build_dataset**(*cfg*)
Build dataset.

## 34.2 transforms

class mmselfsup.datasets.transforms.**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.

Added Keys:

- mask

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.

**get_shape**() → Tuple[int, int]
Get the shape of mask.

**Returns** The shape of mask.

**Return type** Tuple[int, int]

**transform**(*results: dict*) → dict
Method to generate random block mask for each Image in BEiT.

**Parameters** **results** (*dict*) – Result dict from previous pipeline.

**Returns** Result dict with added key mask.

**Return type** dict

class mmselfsup.datasets.transforms.**ColorJitter**(*brightness: Union[float, List[float]] = 0*, *contrast: Union[float, List[float]] = 0*, *saturation: Union[float, List[float]] = 0*, *hue: Union[float, List[float]] = 0*, *backend: str = 'pillow'*)

Randomly change the brightness, contrast, saturation and hue of an image.

Modified from https://github.com/pytorch/vision/blob/main/torchvision/transforms/transforms.py

Required Keys:

- img

Modified Keys:

- img

**Parameters**

- **brightness** (*float or tuple of float (min, max)*) – How much to jitter brightness. brightness_factor is chosen uniformly from [max(0, 1 - brightness), 1 + brightness] or the given [min, max]. Should be non negative numbers.
- **contrast** (*float or tuple of float (min, max)*) – How much to jitter contrast. contrast_factor is chosen uniformly from [max(0, 1 - contrast), 1 + contrast] or the given [min, max]. Should be non negative numbers.
- **saturation** (*float or tuple of float (min, max)*) – How much to jitter saturation. saturation_factor is chosen uniformly from [max(0, 1 - saturation), 1 + saturation] or the given [min, max]. Should be non negative numbers.
- **hue** (*float or tuple of float (min, max)*) – How much to jitter hue. hue_factor is chosen uniformly from [-hue, hue] or the given [min, max]. Should have 0 <= hue <= 0.5 or -0.5 <= min <= max <= 0.5. To jitter hue, the pixel values of the input image has to be non-negative for conversion to HSV space; thus it does not work if you normalize your image to an interval with negative values, or use an interpolation that generates negative values before using this function.
- **backend** (*str*) – The type of image processing backend. Options are *cv2*, *pillow*. Defaults to *pillow*.

**static get_params**(*brightness: Optional[List[float]]*, *contrast: Optional[List[float]]*, *saturation: Optional[List[float]]*, *hue: Optional[List[float]]*) → Tuple[numpy.ndarray, Optional[float], Optional[float], Optional[float], Optional[float]]

Get the parameters for the randomized transform to be applied on image.

**Parameters**

- **brightness** (*tuple of float (min, max), optional*) – The range from which the brightness_factor is chosen uniformly. Pass None to turn off the transformation.
- **contrast** (*tuple of float (min, max), optional*) – The range from which the contrast_factor is chosen uniformly. Pass None to turn off the transformation.
- **saturation** (*tuple of float (min, max), optional*) – The range from which the saturation_factor is chosen uniformly. Pass None to turn off the transformation.
- **hue** (*tuple of float (min, max), optional*) – The range from which the hue_factor is chosen uniformly. Pass None to turn off the transformation.

**Returns**

**The parameters used to apply the randomized transform** along with their random order.

**Return type** tuple

**transform**(*results: dict*) → dict

Randomly change the brightness, contrast, saturation and hue of an image. # noqa: E501.

**Parameters** **results** (*dict*) – The results dict from previous pipeline.

**Returns** Results after applying this transformation.

**Return type** dict

class mmselfsup.datasets.transforms.**MultiView**(*transforms: List[List[Union[dict, Callable[[dict], dict]]]], num_views: Union[int, List[int]]*)

A transform wrapper for multiple views of an image.

**Parameters**

- **transforms** (*list[dict | callable], optional*) – Sequence of transform object or config dict to be wrapped.
- **mapping** (*dict*) – A dict that defines the input key mapping. The keys corresponds to the inner key (i.e., kwargs of the `transform` method), and should be string type. The values corresponds to the outer keys (i.e., the keys of the data/results), and should have a type of string, list or dict. None means not applying input mapping. Default: None.
- **allow_nonexist_keys** (*bool*) – If False, the outer keys in the mapping must exist in the input data, or an exception will be raised. Default: False.

### Examples

```
>>> # Example 1: MultiViews 1 pipeline with 2 views
>>> pipeline = [
>>>     dict(type='MultiView',
>>>         num_views=2,
>>>         transforms=[
>>>             [
>>>                 dict(type='Resize', scale=224))],
>>>         ])
>>> ]
>>> # Example 2: MultiViews 2 pipelines, the first with 2 views,
>>> # the second with 6 views
>>> pipeline = [
>>>     dict(type='MultiView',
>>>         num_views=[2, 6],
>>>         transforms=[
>>>             [
>>>                 dict(type='Resize', scale=224)],
>>>             [
>>>                 dict(type='Resize', scale=224),
>>>                 dict(type='RandomSolarize')],
>>>         ])
>>> ]
```

**transform**(*results: dict*) → dict

Apply transformation to inputs.

**Parameters** **results** (*dict*) – Result dict from previous pipelines.

**Returns** Transformed results.

**Return type** dict

class mmselfsup.datasets.transforms.**PackSelfSupInputs**(*key: str = 'img'*, *algorithm_keys: List[str] = []*, *pseudo_label_keys: List[str] = []*, *meta_keys: List[str] = []*)

Pack data into the format compatible with the inputs of algorithm.

Required Keys:

- img

Added Keys:

- data_samples
- inputs

**Parameters**

- **key** (*str*) – The key of image inputted into the model. Defaults to 'img'.
- **algorithm_keys** (*List[str]*) – Keys of elements related to algorithms, e.g. mask. Defaults to [].
- **pseudo_label_keys** (*List[str]*) – Keys set to be the attributes of pseudo_label. Defaults to [].
- **meta_keys** (*List[str]*) – The keys of meta info of an image. Defaults to [].

classmethod **set_algorithm_keys**(*data_sample: mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*, *key: str*, *results: dict*) → None

Set the algorithm keys of SelfSupDataSample.

**Parameters**

- **data_sample** (SelfSupDataSample) – An instance of SelfSupDataSample.
- **key** (*str*) – The key, which may be used by the algorithm, such as gt_label, sample_idx, mask, pred_label. For more keys, please refer to the attribute of SelfSupDataSample.
- **results** (*dict*) – The results from the data pipeline.

**transform**(*results: Dict*) → Dict[torch.Tensor, *mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]

Method to pack the data.

**Parameters** **results** (*Dict*) – Result dict from the data pipeline.

**Returns**

- inputs (List[torch.Tensor]): The forward data of models.
- data_samples (SelfSupDataSample): The annotation info of the forward data.

**Return type** Dict

class mmselfsup.datasets.transforms.**RandomCrop**(*size: Union[int, Sequence[int]]*, *padding: Optional[Union[int, Sequence[int]]] = None*, *pad_if_needed: bool = False*, *pad_val: Union[numbers.Number, Sequence[numbers.Number]] = 0*, *padding_mode: str = 'constant'*)

Crop the given Image at a random location.

Required Keys:

- img

Modified Keys:

- img
- img_shape

**Parameters**

- **size** (*int or Sequence*) – Desired output size of the crop. If size is an int instead of sequence like (h, w), a square crop (size, size) is made.
- **padding** (*int or Sequence, optional*) – Optional padding on each border of the image. If a sequence of length 4 is provided, it is used to pad left, top, right, bottom borders respectively. If a sequence of length 2 is provided, it is used to pad left/right, top/bottom borders, respectively. Default: None, which means no padding.
- **pad_if_needed** (*boolean*) – It will pad the image if smaller than the desired size to avoid raising an exception. Since cropping is done after padding, the padding seems to be done at a random offset. Default: False.
- **pad_val** (*Number | Sequence[Number]*) – Pixel pad_val value for constant fill. If a tuple of length 3, it is used to pad_val R, G, B channels respectively. Default: 0.
- **padding_mode** (*str*) – Type of padding. Defaults to "constant". Should be one of the following:
  - constant: Pads with a constant value, this value is specified with pad_val.
  - edge: pads with the last value at the edge of the image.
  - reflect: Pads with reflection of image without repeating the last value on the edge. For example, padding [1, 2, 3, 4] with 2 elements on both sides in reflect mode will result in [3, 2, 1, 2, 3, 4, 3, 2].
  - symmetric: Pads with reflection of image repeating the last value on the edge. For example, padding [1, 2, 3, 4] with 2 elements on both sides in symmetric mode will result in [2, 1, 1, 2, 3, 4, 4, 3].

**static get_params**(*img: numpy.ndarray*, *output_size: Tuple*) → Tuple

Get parameters for `crop` for a random crop.

**Parameters**

- **img** (*np.ndarray*) – Image to be cropped.
- **output_size** (*Tuple*) – Expected output size of the crop.

**Returns**

**Params (xmin, ymin, target_height, target_width) to be** passed to `crop` for random crop.

**Return type** tuple

**transform**(*results: dict*) → dict

Randomly crop the image.

**Parameters results** (*dict*) – Result dict from previous pipeline.

**Returns** Result dict with the transformed image.

**Return type** dict

**class** mmselfsup.datasets.transforms.**RandomGaussianBlur**(*sigma_min: float*, *sigma_max: float*, *prob: Optional[float] = 0.5*)

GaussianBlur augmentation refers to SimCLR.

Paper link.

Required Keys:

- img

Modified Keys:

- img

**Parameters**

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

**transform**(*results: dict*) → dict

Apply GaussianBlur augmentation to the given image.

**Parameters** **results** (*dict*) – Results from previous pipeline.

**Returns** Results after applying this transformation.

**Return type** dict

class mmselfsup.datasets.transforms.**RandomPatchWithLabels**

Relative patch location.

Required Keys:

- img

Modified Keys:

- img

Added Keys:

- patch_label
- patch_box
- unpatched_img

Crops image into several patches and concatenates every surrounding patch with center one. Finally gives labels *0, 1, 2, 3, 4, 5, 6, 7* and patch positions.

**transform**(*results: dict*) → dict

Apply random patch augmentation to the given image.

**Parameters** **results** (*dict*) – Results from previous pipeline.

**Returns** Results after applying this transformation.

**Return type** dict

class mmselfsup.datasets.transforms.**RandomResizedCrop**(*size: Union[int, Sequence[int]], scale: Tuple = (0.08, 1.0), ratio: Tuple = (0.75, 1.3333333333333333), max_attempts: int = 10, interpolation: str = 'bilinear', backend: str = 'cv2'*)

Crop the given image to random size and aspect ratio.

A crop of random size (default: of 0.08 to 1.0) of the original size and a random aspect ratio (default: of 3/4 to 4/3) of the original aspect ratio is made. This crop is finally resized to given size.

Required Keys:

- img

Modified Keys:

- img
- img_shape

**Parameters**

- **size** (*Sequence | int*) – Desired output size of the crop. If size is an int instead of sequence like (h, w), a square crop (size, size) is made.
- **scale** (*Tuple*) – Range of the random size of the cropped image compared to the original image. Defaults to (0.08, 1.0).
- **ratio** (*Tuple*) – Range of the random aspect ratio of the cropped image compared to the original image. Defaults to (3. / 4., 4. / 3.).
- **max_attempts** (*int*) – Maximum number of attempts before falling back to Central Crop. Defaults to 10.
- **interpolation** (*str*) – Interpolation method, accepted values are 'nearest', 'bilinear', 'bicubic', 'area', 'lanczos'. Defaults to 'bilinear'.
- **backend** (*str*) – The image resize backend type, accepted values are *cv2* and *pillow*. Defaults to *cv2*.

static get_params(*img: numpy.ndarray*, *scale: Tuple*, *ratio: Tuple*, *max_attempts: int = 10*) → Tuple[int, int, int, int]

Get parameters for crop for a random sized crop.

**Parameters**

- **img** (*np.ndarray*) – Image to be cropped.
- **scale** (*Tuple*) – Range of the random size of the cropped image compared to the original image size.
- **ratio** (*Tuple*) – Range of the random aspect ratio of the cropped image compared to the original image area.
- **max_attempts** (*int*) – Maximum number of attempts before falling back to central crop. Defaults to 10.

**Returns**

**Params (ymin, xmin, ymax, xmax) to be passed to *crop* for** a random sized crop.

**Return type** tuple

transform(*results: dict*) → dict

Randomly crop the image and resize the image to the target size.

**Parameters** **results** (*dict*) – Result dict from previous pipeline.

**Returns** Result dict with the transformed image.

**Return type** dict

**class** mmselfsup.datasets.transforms.**RandomResizedCropAndInterpolationWithTwoPic**(*size: Union[tuple, int]*, *second_size=None*, *scale=(0.08, 1.0)*, *ratio=(0.75, 1.3333333333333333)*, *interpolation='bilinear'*, *second_interpolation='lanczos'*)

Crop the given PIL Image to random size and aspect ratio with random interpolation.

Required Keys:

- img

Modified Keys:

- img

Added Keys:

- target_img

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

A crop of random size (default: of 0.08 to 1.0) of the original size and a random aspect ratio (default: of 3/4 to 4/3) of the original aspect ratio is made. This crop is finally resized to given size. This is popularly used to train the Inception networks. This module first crops the image and resizes the crop to two different sizes.

**Parameters**

- **size** (*Union[tuple, int]*) – Expected output size of each edge of the first image.
- **second_size** (*Union[tuple, int], optional*) – Expected output size of each edge of the second image.
- **scale** (*tuple[float, float]*) – Range of size of the origin size cropped. Defaults to (0.08, 1.0).
- **ratio** (*tuple[float, float]*) – Range of aspect ratio of the origin aspect ratio cropped. Defaults to (3./4., 4./3.).
- **interpolation** (*str*) – The interpolation for the first image. Defaults to `bilinear`.
- **second_interpolation** (*str*) – The interpolation for the second image. Defaults to `lanczos`.

**static get_params**(*img: numpy.ndarray*, *scale: tuple*, *ratio: tuple*) → Sequence[int]

Get parameters for `crop` for a random sized crop.

**Parameters**

- **img** (*np.ndarray*) – Image to be cropped.
- **scale** (*tuple*) – range of size of the origin size cropped
- **ratio** (*tuple*) – range of aspect ratio of the origin aspect ratio cropped

**Returns**

**params (i, j, h, w) to be passed to `crop` for a random** sized crop.

**Return type** tuple

`transform`(*results: dict*) → dict

Crop the given image and resize it to two different sizes.

This module crops the given image randomly and resize the crop to two different sizes. This is popularly used in BEiT-style masked image modeling, where an off-the-shelf model is used to provide the target.

**Parameters** **results** (*dict*) – Results from previous pipeline.

**Returns** Results after applying this transformation.

**Return type** dict

**class** mmselfsup.datasets.transforms.**RandomRotation**(*degrees: Union[int, Sequence[int]], interpolation: str = 'nearest', expand: bool = False, center: Optional[Tuple[float]] = None, fill: int = 0*)

Rotate the image by angle.

Required Keys:

- img

Modified Keys:

- img

**Parameters**

- **degrees** (*sequence | int*) – Range of degrees to select from. If degrees is an int instead of sequence like (min, max), the range of degrees will be (-degrees, +degrees).
- **interpolation** (*str, optional*) – Interpolation method, accepted values are 'nearest', 'bilinear', 'bicubic', 'area', 'lanczos'. Defaults to 'nearest'.
- **expand** (*bool, optional*) – Optional expansion flag. If true, expands the output to make it large enough to hold the entire rotated image. If false or omitted, make the output image the same size as the input image. Note that the expand flag assumes rotation around the center and no translation. Defaults to False.
- **center** (*Tuple[float], optional*) – Center point (w, h) of the rotation in the source image. If not specified, the center of the image will be used. Defaults to None.
- **fill** (*int, optional*) – Pixel fill value for the area outside the rotated image. Default to 0.

**static get_params**(*degrees: List[float]*) → float

Get parameters for `rotate` for a random rotation.

**Parameters** **degrees** (*List[float]*) – Range of degrees to select from.

**Returns**

**angle parameter to be passed to `rotate` for** random rotation.

**Return type** float

`transform`(*results: dict*) → dict

Randomly rotate the image.

**Parameters** **results** (*dict*) – Result dict from previous pipeline.

**Returns** Result dict with the transformed image.

**Return type** dict

class mmselfsup.datasets.transforms.**RandomSolarize**(*threshold: int = 128*, *prob: float = 0.5*)

Solarization augmentation refers to BYOL.

Paper link.

Required Keys:

- img

Modified Keys:

- img

**Parameters**

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

**transform**(*results: dict*) → dict

Apply Solarize augmentation to the given image.

**Parameters** **results** (*dict*) – Results from previous pipeline.

**Returns** Results after applying this transformation.

**Return type** dict

class mmselfsup.datasets.transforms.**RotationWithLabels**

Rotation prediction.

Required Keys:

- img

Modified Keys:

- img

Added Keys:

- rot_label

Rotate each image with 0, 90, 180, and 270 degrees and give labels *0, 1, 2, 3* correspodingly.

**transform**(*results: dict*) → dict

Apply rotation augmentation to the given image.

**Parameters** **results** (*dict*) – Results from previous pipeline.

**Returns** Results after applying this transformation.

**Return type** dict

class mmselfsup.datasets.transforms.**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.

Added Keys:

- mask

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.

**transform**(*results: dict*) → dict

Method to generate random block mask for each Image in SimMIM.

**Parameters** **results** (*dict*) – Result dict from previous pipeline.

**Returns** Result dict with added key `mask`.

**Return type** dict

## 34.3 samplers

**class** `mmselfsup.datasets.samplers.`**DeepClusterSampler**(*dataset: Sized*, *shuffle: bool = True*, *seed: Optional[int] = None*, *replace: bool = False*, *round_up: bool = True*)

The sampler inherits `DefaultSampler` from mmengine.

This sampler supports to set replace to be `True` to get indices. Besides, it defines function `set_uniform_indices`, which is applied in `DeepClusterHook`.

**Parameters**

- **dataset** (*Sized*) – The dataset.
- **shuffle** (*bool*) – Whether shuffle the dataset or not. Defaults to True.
- **seed** (*int, optional*) – Random seed used to shuffle the sampler if `shuffle=True`. This number should be identical across all processes in the distributed group. Defaults to None.
- **replace** (*bool*) – Replace or not in random shuffle. It works on when shuffle is True. Defaults to False.
- **round_up** (*bool*) – Whether to add extra samples to make the number of samples evenly divisible by the world size. Defaults to True.

**set_uniform_indices**(*labels: list*, *num_classes: int*) → None

The function is applied in DeepClusterHook for uniform sampling.

**Parameters**

- **labels** (*list*) – The updated labels after clustering.
- **num_classes** (*int*) – number of clusters.

**Returns** None

CHAPTER

THIRTYFIVE

# MMSELFSUP.ENGINE

## 35.1 hooks

**class** mmselfsup.engine.hooks.**DeepClusterHook**(*extract_dataloader: dict*, *clustering: dict*, *unif_sampling: bool*, *reweight: bool*, *reweight_pow: float*, *init_memory: bool = False*, *initial: bool = True*, *interval: int = 1*, *seed: Optional[int] = 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.
- **seed** (*int, optional*) – Random seed. Defaults to None.

**after_train_epoch**(*runner*) → None

Run cluster after indicated epoch.

**before_train**(*runner*) → None

Run cluster before training.

**deepcluster**(*runner*) → None

Call cluster algorithm.

**evaluate**(*runner*, *new_labels: numpy.ndarray*) → None

Evaluate with labels histogram.

**set_reweight**(*runner*, *labels: numpy.ndarray*, *reweight_pow: float = 0.5*)

Loss re-weighting.

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

**Parameters**

- **runner** (*mmengine.Runner*) – mmengine Runner.

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

class mmselfsup.engine.hooks.**DenseCLHook**(*start_iters: int = 1000*)

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*) – The number of warmup iterations to set `loss_lambda=0`. Defaults to 1000.

**before_train**(*runner*) → None

Obtain `loss_lambda` from algorithm.

**before_train_iter**(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*) → None

Adjust `loss_lambda` every train iter.

class mmselfsup.engine.hooks.**ODCHook**(*centroids_update_interval: int*, *deal_with_small_clusters_interval: int*, *evaluate_interval: int*, *reweight: bool*, *reweight_pow: float*, *dist_mode: bool = 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.

**after_train_epoch**(*runner*) → None

Save cluster.

**after_train_iter**(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*, *outputs: Optional[dict] = None*) → None

Update cluster centroids and the loss_weight.

**evaluate**(*runner*, *new_labels: numpy.ndarray*) → None

Evaluate with labels histogram.

**set_reweight**(*runner*, *labels: Optional[numpy.ndarray] = None*, *reweight_pow: float = 0.5*)

Loss re-weighting.

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

**Parameters**

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

class mmselfsup.engine.hooks.**SimSiamHook**(*fix_pred_lr: bool*, *lr: float*, *adjust_by_epoch: Optional[bool] = True*)

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*) → None

fix lr of predictor by epoch.

**before_train_iter**(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*) → None

fix lr of predictor by iter.

class mmselfsup.engine.hooks.**SwAVHook**(*batch_size: int*, *epoch_queue_starts: Optional[int] = 15*, *crops_for_assign: Optional[List[int]] = [0, 1]*, *feat_dim: Optional[int] = 128*, *queue_length: Optional[int] = 0*, *interval: Optional[int] = 1*, *frozen_layers_cfg: Optional[Dict] = {}*)

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.
- **frozen_layers_cfg** (*dict, optional*) – Dict to config frozen layers. The key-value pair is layer name and its frozen iters. If frozen, the layers don't need gradient. Defaults to dict().

**after_train_epoch**(*runner*) → None

Save the queues locally.

**before_run**(*runner*) → None

Check whether the queues exist locally or not.

**before_train_epoch**(*runner*) → None

Check the queues' state.

**before_train_iter**(*runner*, *batch_idx: int*, *data_batch: Optional[Sequence[dict]] = None*) → None

Freeze layers before specific iters according to the config.

## 35.2 optimizers

class mmselfsup.engine.optimizers.**LARS**(*params: Iterable*, *lr: float*, *momentum: float = 0*, *weight_decay: float = 0*, *dampening: float = 0*, *eta: float = 0.001*, *nesterov: bool = False*, *eps: float = 1e-08*)

Implements layer-wise adaptive rate scaling for SGD.

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

**Parameters**

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

### 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*) → torch.Tensor

Performs a single optimization step.

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

class mmselfsup.engine.optimizers.**LearningRateDecayOptimWrapperConstructor**(*optim_wrapper_cfg: dict*, *paramwise_cfg: Optional[dict] = None*)

Different learning rates are set for different layers of backbone.

Note: Currently, this optimizer constructor is built for ViT and Swin.

In addition to applying layer-wise learning rate decay schedule, the paramwise_cfg only supports weight decay customization.

**add_params**(*params: List[dict]*, *module: torch.nn.modules.module.Module*, *optimizer_cfg: dict*, ***kwargs*) → None

Add all parameters of module to the params list.

The parameters of the given module will be added to the list of param groups, with specific rules defined by paramwise_cfg.

**Parameters**

- **params** (*List[dict]*) – A list of param groups, it will be modified in place.
- **module** (*nn.Module*) – The module to be added.
- **optimizer_cfg** (*dict*) – The configuration of optimizer.
- **prefix** (*str*) – The prefix of the module.

CHAPTER

# THIRTYSIX

# MMSELFSUP.EVALUATION

## 36.1 functional

mmselfsup.evaluation.functional.**knn_eval**(*train_features: torch.Tensor*, *train_labels: torch.Tensor*, *test_features: torch.Tensor*, *test_labels: torch.Tensor*, *k: int*, *T: float*, *num_classes: int = 1000*) → Tuple[float, float]

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.

**Returns** The top1 and top5 accuracy.

**Return type** Tuple[float, float]

CHAPTER

# THIRTYSEVEN

# MMSELFSUP.MODELS

## 37.1 algorithms

**class** mmselfsup.models.algorithms.**BEiT**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

BEiT v1/v2.

Implementation of BEiT: BERT Pre-Training of Image Transformers and BEiT v2: Masked Image Modeling with Vector-Quantized Visual Tokenizers.

**loss**(*batch_inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **batch_inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**BYOL**(*backbone: dict*, *neck: dict*, *head: dict*, *base_momentum: float = 0.996*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[dict] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

BYOL.

Implementation of Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning.

**Parameters**

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors.
- **head** (*dict*) – Config dict for module of head functions.
- **base_momentum** (*float*) – The base momentum coefficient for the target network. Defaults to 0.996.

- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See `SelfSupDataPreprocessor` for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]
Function to extract features from backbone.

**Parameters** **batch_inputs** (*List[torch.Tensor]*) – The input images.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]
The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**BarlowTwins**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

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.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]
Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]
The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**BaseModel**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

BaseModel for SelfSup.

All algorithms should inherit this module.

**Parameters**

- **backbone** (*dict*) – The backbone module. See `mmcls.models.backbones`.
- **neck** (*dict, optional*) – The neck module to process features from backbone. See `mmcls.models.necks`. Defaults to None.
- **head** (*dict, optional*) – The head module to do prediction and calculate loss from processed features. See `mmcls.models.heads`. Notice that if the head is not set, almost all methods cannot be used except *extract_feat()*. Defaults to None.
- **target_generator** – (dict, optional): The target_generator module to generate targets for self-supervised learning optimization, such as HOG, extracted features from other modules(DALL-E, CLIP), etc.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*Union[dict, nn.Module], optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See `SelfSupDataPreprocessor` for more details. Defaults to None.
- **init_cfg** (*dict, optional*) – the config to control the initialization. Defaults to None.

**extract_feat**(*inputs: torch.Tensor*)

Extract features from the input tensor with shape (N, C, . . . ).

This is a abstract method, and subclass should overwrite this methods if needed.

**Parameters** **inputs** (*Tensor*) – A batch of inputs. The shape of it should be `(num_samples, num_channels, *img_shape)`.

**Returns** The output of specified stage. The output depends on detailed implementation.

**Return type** tuple | Tensor

**forward**(*inputs: torch.Tensor*, *data_samples: Optional[List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]] = None*, *mode: str = 'tensor'*)

Returns losses or predictions of training, validation, testing, and simple inference process.

This module overwrites the abstract method in `BaseModel`.

**Parameters**

- **inputs** (*torch.Tensor*) – batch input tensor collated by `data_preprocessor`.
- **data_samples** (*List[BaseDataElement], optional*) – data samples collated by `data_preprocessor`.
- **mode** (*str*) – mode should be one of `loss`, `predict` and `tensor`.
  - `loss`: Called by `train_step` and return loss `dict` used for logging
  - `predict`: Called by `val_step` and `test_step` and return list of `BaseDataElement` results used for computing metric.
  - `tensor`: Called by custom use to get `Tensor` type results.

**Returns**

- If `mode == loss`, return a `dict` of loss tensor used for backward and logging.
- If `mode == predict`, return a `list` of `BaseDataElement` for computing metric and getting inference result.
- If `mode == tensor`, return a tensor or `tuple` of tensor or ``dict of tensor for custom use.

**Return type** ForwardResults (dict or list)

`loss`(*inputs: torch.Tensor*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*) → dict

Calculate losses from a batch of inputs and data samples.

This is a abstract method, and subclass should overwrite this methods if needed.

**Parameters**

- **inputs** (*torch.Tensor*) – The input tensor with shape (N, C, ...) in general.
- **data_samples** (*List[SelfSupDataSample]*) – The annotation data of every samples.

**Returns** A dictionary of loss components.

**Return type** dict[str, Tensor]

`predict`(*inputs: tuple*, *data_samples: Optional[List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]] = None*, ***kwargs*) → List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]

Predict results from the extracted features.

This module returns the logits before loss, which are used to compute all kinds of metrics. This is a abstract method, and subclass should overwrite this methods if needed.

**Parameters**

- **feats** (*tuple*) – The features extracted from the backbone.
- **data_samples** (*List[BaseDataElement], optional*) – The annotation data of every samples. Defaults to None.
- ****kwargs** – Other keyword arguments accepted by the `predict` method of `head`.

**property with_head: bool**

Check if the model has a head module.

**property with_neck: bool**

Check if the model has a neck module.

**property with_target_generator: bool**

Check if the model has a target_generator module.

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

CAE.

Implementation of Context Autoencoder for Self-Supervised Representation Learning.

**Parameters**

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of neck.
- **head** (*dict*) – Config dict for module of head functions.
- **target_generator** – (dict, optional): The target_generator module to generate targets for self-supervised learning optimization, such as HOG, extracted features from other modules(DALL-E, CLIP), etc.
- **base_momentum** (*float*) – The base momentum coefficient for the target network. Defaults to 0.0.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See SelfSupDataPreprocessor for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**init_weights**() → None

Initialize weights.

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**momentum_update**() → None

Momentum update of the teacher network.

class mmselfsup.models.algorithms.**DeepCluster**(*backbone: dict*, *neck: dict*, *head: dict*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[dict] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

DeepCluster.

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

**Parameters**

- **backbone** (*dict*) – Config dict for module of backbone.

- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors.
- **head** (*dict*) – Config dict for module of head functions.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See `SelfSupDataPreprocessor` for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwarg*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**predict**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]

The forward function in testing.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** List[*SelfSupDataSample*] class mmselfsup.models.algorithms.**DenseCL**(*backbone: dict*, *neck: dict*, *head: dict*, *queue_len: int = 65536*, *feat_dim: int = 128*, *momentum: float = 0.999*, *loss_lambda: float = 0.5*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[dict] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

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 *engine/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.
- **head** (*dict*) – Config dict for module of head functions.
- **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.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See SelfSupDataPreprocessor for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**predict**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → *mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*

Predict results from the extracted features.

**Parameters**

- **batch_inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** *SelfSupDataSample*

**class** mmselfsup.models.algorithms.**EVA**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

EVA.

Implementation of EVA: Exploring the Limits of Masked Visual Representation Learning at Scale.

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**MAE**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

MAE.

Implementation of Masked Autoencoders Are Scalable Vision Learners.

**extract_feat**(*inputs: List[torch.Tensor]*, *data_samples: Optional[List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]] = None*, ***kwarg*) → Tuple[torch.Tensor]

The forward function to extract features from neck.

**Parameters** **inputs** (*List[torch.Tensor]*) – The input images.

**Returns** Neck outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**reconstruct**(*features: torch.Tensor*, *data_samples: Optional[List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]] = None*, ***kwargs*) → *mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*

The function is for image reconstruction.

**Parameters**

- **features** (*torch.Tensor*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** *SelfSupDataSample*

**class** mmselfsup.models.algorithms.**MILAN**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

MILAN.

Implementation of MILAN: Masked Image Pretraining on Language Assisted Representation.

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**MaskFeat**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

MaskFeat.

Implementation of Masked Feature Prediction for Self-Supervised Visual Pre-Training.

**extract_feat**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, *compute_hog: bool = True*, ***kwarg*) → Tuple[torch.Tensor]

The forward function to extract features from neck.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images and mask.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.
- **compute_hog** (*bool*) – Whether to compute hog during extraction. If True, the batch size of inputs need to be 1. Defaults to True.

**Returns** Neck outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**reconstruct**(*features: List[torch.Tensor]*, *data_samples: Optional[List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]] = None*, ***kwargs*) → *mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*

The function is for image reconstruction.

**Parameters**

- **features** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** *SelfSupDataSample*

**class** mmselfsup.models.algorithms.**MixMIM**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

MiXMIM.

Implementation of MixMIM: Mixed and Masked Image Modeling for Efficient Visual Representation Learning..

**loss**(*inputs: List[torch.Tensor], data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample], **kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

class mmselfsup.models.algorithms.**MoCo**(*backbone: dict, neck: dict, head: dict, queue_len: int = 65536, feat_dim: int = 128, momentum: float = 0.999, pretrained: Optional[str] = None, data_preprocessor: Optional[dict] = None, init_cfg: Optional[Union[dict, List[dict]]] = None*)

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.
- **head** (*dict*) – Config dict for module of head functions.
- **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.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See SelfSupDataPreprocessor for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**extract_feat**(*inputs: List[torch.Tensor], **kwarg*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**MoCoV3**(*backbone: dict*, *neck: dict*, *head: dict*, *base_momentum: float = 0.99*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[dict] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

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.
- **head** (*dict*) – Config dict for module of head functions.
- **base_momentum** (*float*) – Momentum coefficient for the momentum-updated encoder. Defaults to 0.99.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See `SelfSupDataPreprocessor` for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwarg*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.

- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**NPID**(*backbone: dict*, *neck: dict*, *head: dict*, *memory_bank: dict*, *neg_num: int = 65536*, *ensure_neg: bool = False*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[dict] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = 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.
- **head** (*dict*) – Config dict for module of head functions.
- **memory_bank** (*dict*) – Config dict for module of memory bank.
- **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.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See SelfSupDataPreprocessor for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwarg*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, Tensor]

class mmselfsup.models.algorithms.ODC(*backbone: dict*, *neck: dict*, *head: dict*, *memory_bank: dict*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[dict] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = 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 *engine/hooks/odc_hook.py*.

**Parameters**

- **backbone** (*dict*) – Config dict for module of backbone.
- **neck** (*dict*) – Config dict for module of deep features to compact feature vectors.
- **head** (*dict*) – Config dict for module of head functions.
- **memory_bank** (*dict*) – Config dict for module of memory bank.
- **pretrained** (*str, optional*) – The pretrained checkpoint path, support local path and remote path. Defaults to None.
- **data_preprocessor** (*dict, optional*) – The config for preprocessing input data. If None or no specified type, it will use "SelfSupDataPreprocessor" as type. See SelfSupDataPreprocessor for more details. Defaults to None.
- **init_cfg** (*Union[List[dict], dict], optional*) – Config dict for weight initialization. Defaults to None.

extract_feat(*inputs: List[torch.Tensor]*, ***kwarg*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** Backbone outputs.

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

loss(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**predict**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]

The forward function in testing.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** List[*SelfSupDataSample*]

**class** mmselfsup.models.algorithms.**RelativeLoc**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

Relative patch location.

Implementation of Unsupervised Visual Representation Learning by Context Prediction.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters** **inputs** (*List[torch.Tensor]*) – The input images.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**predict**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]

The forward function in testing.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** List[*SelfSupDataSample*]

**class** mmselfsup.models.algorithms.**RotationPred**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

Rotation prediction.

Implementation of Unsupervised Representation Learning by Predicting Image Rotations.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters** **inputs** (*List[torch.Tensor]*) – The input images.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[*SelfSupDataSample*]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**predict**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]

The forward function in testing.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[*SelfSupDataSample*]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** List[*SelfSupDataSample*]

**class** mmselfsup.models.algorithms.**SimCLR**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

SimCLR.

Implementation of A Simple Framework for Contrastive Learning of Visual Representations.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters** **inputs** (*List[torch.Tensor]*) – The input images.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[*SelfSupDataSample*]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

**class** mmselfsup.models.algorithms.**SimMIM**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

SimMIM.

Implementation of SimMIM: A Simple Framework for Masked Image Modeling.

**extract_feat**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwarg*) → torch.Tensor

The forward function to extract features.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[*SelfSupDataSample*]*) – All elements required during the forward function.

**Returns** The reconstructed images.

**Return type** torch.Tensor

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[*SelfSupDataSample*]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, Tensor]

**reconstruct**(*features: torch.Tensor*, *data_samples: Optional[List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]] = None*, ***kwargs*) → *mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*

The function is for image reconstruction.

**Parameters**

- **features** (*torch.Tensor*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** The prediction from model.

**Return type** *SelfSupDataSample*

**class** mmselfsup.models.algorithms.**SimSiam**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

SimSiam.

Implementation of Exploring Simple Siamese Representation Learning. The operation of fixing learning rate of predictor is in *engine/hooks/simsiam_hook.py*.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwarg*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters** **inputs** (*List[torch.Tensor]*) – The input images.

**Returns** Backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*]*, ***kwargs*) → Dict[str, torch.Tensor]

The forward function in training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, Tensor]

**class** mmselfsup.models.algorithms.**SwAV**(*backbone: dict*, *neck: Optional[dict] = None*, *head: Optional[dict] = None*, *target_generator: Optional[dict] = None*, *pretrained: Optional[str] = None*, *data_preprocessor: Optional[Union[dict, torch.nn.modules.module.Module]] = None*, *init_cfg: Optional[dict] = None*)

SwAV.

Implementation of Unsupervised Learning of Visual Features by Contrasting Cluster Assignments. The queue is built in *engine/hooks/swav_hook.py*.

**extract_feat**(*inputs: List[torch.Tensor]*, ***kwargs*) → Tuple[torch.Tensor]

Function to extract features from backbone.

**Parameters** **inputs** (*List[torch.Tensor]*) – The input images.

**Returns** backbone outputs.

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

**loss**(*inputs: List[torch.Tensor]*, *data_samples: List[mmselfsup.structures.selfsup_data_sample.SelfSupDataSample]*, ***kwargs*) → Dict[str, torch.Tensor]

Forward computation during training.

**Parameters**

- **inputs** (*List[torch.Tensor]*) – The input images.
- **data_samples** (*List[SelfSupDataSample]*) – All elements required during the forward function.

**Returns** A dictionary of loss components.

**Return type** Dict[str, torch.Tensor]

## 37.2 backbones

**class** mmselfsup.models.backbones.**BEiTViT**(*arch: str = 'base'*, *img_size: int = 224*, *patch_size: int = 16*, *in_channels: int = 3*, *out_indices: int = - 1*, *drop_rate: float = 0*, *drop_path_rate: float = 0*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *final_norm: bool = True*, *avg_token: bool = False*, *frozen_stages: int = - 1*, *output_cls_token: bool = True*, *use_abs_pos_emb: bool = False*, *use_rel_pos_bias: bool = False*, *use_shared_rel_pos_bias: bool = True*, *layer_scale_init_value: int = 0.1*, *interpolate_mode: str = 'bicubic'*, *patch_cfg: dict = {'padding': 0}*, *layer_cfgs: dict = {}*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

Vision Transformer for BEiT pre-training.

Rewritten version of: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

**Parameters**

- **arch** (*str | dict*) – Vision Transformer architecture. If use string, choose from 'small', 'base' and 'large'. If use dict, it should have below keys:
  - **embed_dims** (int): The dimensions of embedding.
  - **num_layers** (int): The number of transformer encoder layers.
  - **num_heads** (int): The number of heads in attention modules.
  - **feedforward_channels** (int): The hidden dimensions in feedforward modules.

  Defaults to 'base'.
- **img_size** (*int | tuple*) – The expected input image shape. Because we support dynamic input shape, just set the argument to the most common input image shape. Defaults to 224.
- **patch_size** (*int | tuple*) – The patch size in patch embedding. Defaults to 16.
- **in_channels** (*int*) – The num of input channels. Defaults to 3.
- **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.
- **qkv_bias** (*bool*) – Whether to add bias for qkv in attention modules. Defaults to True.
- **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.
- **with_cls_token** (*bool*) – Whether concatenating class token into image tokens as transformer input. Defaults to True.
- **avg_token** (*bool*) – Whether or not to use the mean patch token for classification. If True, the model will only take the average of all patch tokens. 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.
- **output_cls_token** (*bool*) – Whether output the cls_token. If set True, `with_cls_token` must be True. Defaults to True.
- **use_abs_pos_emb** (*bool*) – Whether or not use absolute position embedding. Defaults to False.
- **use_rel_pos_bias** (*bool*) – Whether or not use relative position bias. Defaults to False.
- **use_shared_rel_pos_bias** (*bool*) – Whether or not use shared relative position bias. Defaults to True.
- **layer_scale_init_value** (*float*) – The initialization value for the learnable scaling of attention and FFN. Defaults to 0.1.
- **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*) → Tuple[torch.Tensor]

The BEiT style forward function.

**Parameters**

- **x** (*torch.Tensor*) – Input images, which is of shape (B x C x H x W).
- **mask** (*torch.Tensor*) – Mask for input, which is of shape (B x patch_resolution[0] x patch_resolution[1]).

**Returns** Hidden features.

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

**init_weights**() → None

Initialize position embedding, patch embedding and cls token.

**rescale_init_weight**() → None

Rescale the initialized weights.

**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

Generate features for masked images.

This function generates mask images and get the hidden features for visible patches.

**Parameters**

- **x** (*torch.Tensor*) – Input images, which is of shape B x C x H x W.
- **mask** (*torch.Tensor*) – Mask for input, which is of shape B x L.

**Returns** hidden features.

**Return type** torch.Tensor

**init_weights**() → None

Initialize position embedding, patch embedding and cls token.

**class** mmselfsup.models.backbones.**MAEViT**(*arch: Union[str, dict] = 'b'*, *img_size: int = 224*, *patch_size: int = 16*, *out_indices: Union[Sequence, int] = - 1*, *drop_rate: float = 0*, *drop_path_rate: float = 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 = {}*, *mask_ratio: float = 0.75*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

Vision Transformer for MAE pre-training.

A PyTorch implement of: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. This module implements the patch masking in MAE and initialize the position embedding with sine-cosine position embedding.

**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** (*Union[List[dict], dict], optional*) – Initialization config dict. Defaults to None.

**forward**(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Generate features for masked images.

This function generates mask and masks some patches randomly and get the hidden features for visible patches.

**Parameters** **x** (*torch.Tensor*) – Input images, which is of shape B x C x H x W.

**Returns**

Hidden features, mask and the ids to restore original image.

- x (torch.Tensor): hidden features, which is of shape B x (L * mask_ratio) x C.
- mask (torch.Tensor): mask used to mask image.
- ids_restore (torch.Tensor): ids to restore original image.

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

`init_weights()` → None

Initialize position embedding, patch embedding and cls token.

`random_masking`(*x: torch.Tensor*, *mask_ratio: float = 0.75*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Generate the mask for MAE Pre-training.

**Parameters**

- **x** (*torch.Tensor*) – Image with data augmentation applied, which is of shape B x L x C.
- **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 (torch.Tensor): masked image.
- mask (torch.Tensor): mask used to mask image.
- ids_restore (torch.Tensor): ids to restore original image.

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

class mmselfsup.models.backbones.**MILANViT**(*arch: Union[str, dict] = 'b'*, *img_size: int = 224*, *patch_size: int = 16*, *out_indices: Union[Sequence, int] = - 1*, *drop_rate: float = 0*, *drop_path_rate: float = 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 = {}*, *mask_ratio: float = 0.75*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

MILANViT.

Implementation of the encoder for MILAN: Masked Image Pretraining on Language Assisted Representation. This module inherits from MAEViT and only overrides the forward function and replace random masking with attention masking.

`attention_masking`(*x: torch.Tensor*, *mask_ratio: float*, *importance: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Generate attention mask for MILAN.

This is what is different from MAEViT, which uses random masking. Attention masking generates attention mask for MILAN, according to importance. The higher the importance, the more likely the patch is kept.

**Parameters**

- **x** (*torch.Tensor*) – Input images, which is of shape B x L x C.
- **mask_ratio** (*float*) – The ratio of patches to be masked.
- **importance** (*torch.Tensor*) – Importance of each patch, which is of shape B x L.

**Returns** masked image, mask, the ids to restore original image, ids of the shuffled patches, ids of the kept patches, ids of the removed patches.

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

`forward`(*x: torch.Tensor*, *importance: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Generate features for masked images.

This function generates mask and masks some patches randomly and get the hidden features for visible patches. The mask is generated by importance. The higher the importance, the more likely the patch is kept. The importance is calculated by CLIP. The higher the CLIP score, the more likely the patch is kept. The CLIP score is calculated by by cross attention between the class token and all other tokens from the last layer.

**Parameters**

- **x** (*torch.Tensor*) – Input images, which is of shape B x C x H x W.
- **importance** (*torch.Tensor*) – Importance of each patch, which is of shape B x L.

**Returns**

masked image, the ids to restore original image, ids of the kept patches, ids of the removed patches.

- x (torch.Tensor): hidden features, which is of shape B x (L * mask_ratio) x C.
- ids_restore (torch.Tensor): ids to restore original image.
- ids_keep (torch.Tensor): ids of the kept patches.
- ids_dump (torch.Tensor): ids of the removed patches.

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

class mmselfsup.models.backbones.**MaskFeatViT**(*arch: Union[str, dict] = 'b'*, *img_size: int = 224*, *patch_size: int = 16*, *out_indices: Union[Sequence, int] = - 1*, *drop_rate: float = 0*, *drop_path_rate: float = 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[Union[dict, List[dict]]] = None*)

Vision Transformer for MaskFeat pre-training.

A PyTorch implement of: Masked Feature Prediction for Self-Supervised Visual Pre-Training.

**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.
- **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 position embedding, mask token and cls token.

**class** mmselfsup.models.backbones.**MixMIMTransformerPretrain**(*arch: Union[str, dict] = 'base'*, *mlp_ratio: float = 4*, *img_size: int = 224*, *patch_size: int = 4*, *in_channels: int = 3*, *window_size: List = [14, 14, 14, 7]*, *qkv_bias: bool = True*, *patch_cfg: dict = {}*, *norm_cfg: dict = {'type': 'LN'}*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *attn_drop_rate: float = 0.0*, *use_checkpoint: bool = False*, *range_mask_ratio: float = 0.0*, *init_cfg: Optional[dict] = None*)

MixMIM backbone during pretraining.

A PyTorch implement of : ` MixMIM: Mixed and Masked Image Modeling for Efficient Visual Representation Learning <https://arxiv.org/abs/2205.13137>`_

**Parameters**

- **arch** (*str | dict*) – MixMIM architecture. If use string, choose from 'base','large' and 'huge'. If use dict, it should have below keys:
  - **embed_dims** (int): The dimensions of embedding.
  - **depths** (int): The number of transformer encoder layers.
  - **num_heads** (int): The number of heads in attention modules.

  Defaults to 'base'.
- **mlp_ratio** (*int*) – The mlp ratio in FFN. Defaults to 4.
- **img_size** (*int | tuple*) – The expected input image shape. Because we support dynamic input shape, just set the argument to mlp_ratio the most common input image shape. Defaults to 224.
- **patch_size** (*int | tuple*) – The patch size in patch embedding. Defaults to 16.
- **in_channels** (*int*) – The num of input channels. Defaults to 3.
- **window_size** (*list*) – The height and width of the window.
- **qkv_bias** (*bool*) – Whether to add bias for qkv in attention modules. Defaults to True.

- **patch_cfg** (*dict*) – Extra config dict for patch embedding. Defaults to an empty dict.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **drop_rate** (*float*) – Probability of an element to be zeroed. Defaults to 0.
- **drop_path_rate** (*float*) – Stochastic depth rate. Defaults to 0.
- **attn_drop_rate** (*float*) – Attention drop rate. Defaults to 0.
- **use_checkpoint** (*bool*) – Whether use the checkpoint to
- **GPU memory cost** (*reduce*) –
- **range_mask_ratio** (*float*) – The range of mask ratio. Defaults to 0.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

**forward**(*x: torch.Tensor*, *mask_ratio=0.5*)

Generate features for masked images.

This function generates mask and masks some patches randomly and get the hidden features for visible patches.

**Parameters** **x** (*torch.Tensor*) – Input images, which is of shape B x C x H x W.

**Returns**

- x (torch.Tensor): hidden features, which is of shape B x L x C.
- mask_s4 (torch.Tensor): the mask tensor for the last layer.

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

**init_weights**()

Initialize position embedding, patch embedding.

**random_masking**(*x: torch.Tensor*, *mask_ratio: float = 0.5*)

Generate the mask for MixMIM Pretraining.

**Parameters**

- **x** (*torch.Tensor*) – Image with data augmentation applied, which is of shape B x L x C.
- **mask_ratio** (*float*) – The mask ratio of total patches. Defaults to 0.5.

**Returns**

- mask_s1 (torch.Tensor): mask with stride of self.encoder_stride // 8.
- mask_s2 (torch.Tensor): mask with stride of self.encoder_stride // 4.
- mask_s3 (torch.Tensor): mask with stride of self.encoder_stride // 2.
- mask (torch.Tensor): mask with stride of self.encoder_stride.

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

**class** mmselfsup.models.backbones.**MoCoV3ViT**(*stop_grad_conv1: bool = False*, *frozen_stages: int = - 1*, *norm_eval: bool = False*, *init_cfg: Optional[Union[dict, List[dict]]] = 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*) – 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**() → None

Initialize position embedding, patch embedding, qkv layers and cls token.

**train**(*mode: bool = True*) → None

Set module status before forward computation.

**Parameters** **mode** (*bool*) – Whether it is train_mode or test_mode

class mmselfsup.models.backbones.**ResNeXt**(*depth: int*, *groups: int = 32*, *width_per_group: int = 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*) → torch.nn.modules.module.Module
Redefine the function for ResNeXt related args.

class mmselfsup.models.backbones.**ResNet**(*depth: int, in_channels: int = 3, stem_channels: int = 64, base_channels: int = 64, expansion: Optional[int] = None, num_stages: int = 4, strides: Tuple[int] = (1, 2, 2, 2), dilations: Tuple[int] = (1, 1, 1, 1), out_indices: Tuple[int] = (4), style: str = 'pytorch', deep_stem: bool = False, avg_down: bool = False, frozen_stages: int = - 1, conv_cfg: Optional[dict] = None, norm_cfg: Optional[dict] = {'requires_grad': True, 'type': 'BN'}, norm_eval: bool = False, with_cp: bool = False, zero_init_residual: bool = False, init_cfg: Optional[dict] = [{'type': 'Kaiming', 'layer': ['Conv2d']}, {'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}], drop_path_rate: float = 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: torch.Tensor*) → Tuple[torch.Tensor]

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 will be used for downstream evaluation.

**class** mmselfsup.models.backbones.**ResNetSobel**(***kwargs*)

ResNet with Sobel layer.

This variant is used in clustering-based methods like DeepCluster to avoid color shortcut.

**forward**(*x: torch.Tensor*) → Tuple[torch.Tensor]

Forward function.

**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 = {}*, *pad_small_map: bool = False*, *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.
- **pad_small_map** (*bool*) – If True, pad the small feature map to the window size, which is common used in detection and segmentation. If False, avoid shifting window and shrink the window size to the size of feature map, which is common used in classification. Defaults to False.
- **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.

## 37.3 necks

**class** mmselfsup.models.necks.**AvgPool2dNeck**(*output_size: int = 1*)

The average pooling 2d neck.

**forward**(*x: List[torch.Tensor]*) → List[torch.Tensor]

Forward function.

**class** mmselfsup.models.necks.**BEiTV2Neck**(*num_layers: int = 2*, *early_layers: int = 9*, *backbone_arch: str = 'base'*, *drop_rate: float = 0.0*, *drop_path_rate: float = 0.0*, *layer_scale_init_value: float = 0.1*, *use_rel_pos_bias: bool = False*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *init_cfg: Optional[Union[dict, List[dict]]] = {'bias': 0, 'layer': 'Linear', 'std': 0.02, 'type': 'TruncNormal'}*)

Neck for BEiTV2 Pre-training.

This module construct the decoder for the final prediction.

**Parameters**

- **num_layers** (*int*) – Number of encoder layers of neck. Defaults to 2.
- **early_layers** (*int*) – The layer index of the early output from the backbone. Defaults to 9.
- **backbone_arch** (*str*) – Vision Transformer architecture. Defaults to base.
- **drop_rate** (*float*) – Probability of an element to be zeroed. Defaults to 0.
- **drop_path_rate** (*float*) – stochastic depth rate. Defaults to 0.
- **layer_scale_init_value** (*float*) – The initialization value for the learnable scaling of attention and FFN. Defaults to 0.1.
- **use_rel_pos_bias** (*bool*) – Whether to use unique relative position bias, if False, use shared relative position bias defined in backbone.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Defaults to `dict(type='LN')`.
- **init_cfg** (*dict, optional*) – Initialization config dict. Defaults to None.

**forward**(*inputs: Tuple[torch.Tensor]*, *rel_pos_bias: torch.Tensor*, ***kwargs*) → Tuple[torch.Tensor, torch.Tensor]

Get the latent prediction and final prediction.

**Parameters**

- **x** (*Tuple[torch.Tensor]*) – Features of tokens.
- **rel_pos_bias** (*torch.Tensor*) – Shared relative position bias table.

**Returns**

- x: The final layer features from backbone, which are normed in BEiTV2Neck.
- x_cls_pt: The early state features from backbone, which are consist of final layer cls_token and early state patch_tokens from backbone and sent to PatchAggregation layers in the neck.

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

**rescale_patch_aggregation_init_weight**()
Rescale the initialized weights.

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

Initialization.

class mmselfsup.models.necks.**ClsBatchNormNeck**(*input_features: int*, *affine: bool = False*, *eps: float = 1e-06*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

Normalize cls token across batch before head.

This module is proposed by MAE, when running linear probing.

**Parameters**

- **input_features** (*int*) – The dimension of features.
- **affine** (*bool*) – a boolean value that when set to True, this module has learnable affine parameters. Defaults to False.
- **eps** (*float*) – a value added to the denominator for numerical stability. Defaults to 1e-6.
- **init_cfg** (*Dict or List[Dict], optional*) – Config dict for weight initialization. Defaults to None.

**forward**(*inputs: Tuple[List[torch.Tensor]]*) → Tuple[List[torch.Tensor]]

The forward function.

class mmselfsup.models.necks.**DenseCLNeck**(*in_channels: int*, *hid_channels: int*, *out_channels: int*, *num_grid: Optional[int] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

The non-linear neck of DenseCL.

Single and dense neck in parallel: fc-relu-fc, conv-relu-conv. Borrowed from the authors' code.

**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: List[torch.Tensor]*) → List[torch.Tensor]

Forward function of neck.

**Parameters** **x** (*List[torch.Tensor]*) – feature map of backbone.

**Returns**

**The global feature** vectors and dense feature vectors. - avgpooled_x: Global feature vectors. - x: Dense feature vectors. - avgpooled_x2: Dense feature vectors for queue.

**Return type** List[torch.Tensor, torch.Tensor, torch.Tensor] class mmselfsup.models.necks.**LinearNeck**(*in_channels: int*, *out_channels: int*, *with_avg_pool: bool = True*, *init_cfg: Optional[Union[dict, List[dict]]] = 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: Tuple[torch.Tensor]*) → List[torch.Tensor]

Forward function.

**Parameters** **x** (*List[torch.Tensor]*) – The feature map of backbone.

**Returns** The output features.

**Return type** List[torch.Tensor]

class mmselfsup.models.necks.**MAEPretrainDecoder**(*num_patches: int = 196*, *patch_size: int = 16*, *in_chans: int = 3*, *embed_dim: int = 1024*, *decoder_embed_dim: int = 512*, *decoder_depth: int = 8*, *decoder_num_heads: int = 16*, *mlp_ratio: int = 4*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *predict_feature_dim: Optional[float] = None*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

Decoder for MAE Pre-training.

Some of the code is borrowed from *https://github.com/facebookresearch/mae*. # noqa

**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.
- **init_cfg** (*Union[List[dict], dict], optional*) – Initialization config dict. Defaults to None.

### 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)
```

**property decoder_norm**
The normalization layer of decoder.

**forward**(*x: torch.Tensor*, *ids_restore: torch.Tensor*) → torch.Tensor
The forward function.

The process computes the visible patches' features vectors and the mask tokens to output feature vectors, which will be used for reconstruction.

**Parameters**

- **x** (*torch.Tensor*) – hidden features, which is of shape B x (L * mask_ratio) x C.
- **ids_restore** (*torch.Tensor*) – ids to restore original image.

**Returns**

**The reconstructed feature vectors, which is of** shape B x (num_patches) x C.

**Return type** x (torch.Tensor)

**init_weights**() → None
Initialize position embedding and mask token of MAE decoder.

**class** mmselfsup.models.necks.**MILANPretrainDecoder**(*num_patches: int = 196*, *patch_size: int = 16*, *in_chans: int = 3*, *embed_dim: int = 1024*, *decoder_embed_dim: int = 512*, *decoder_depth: int = 8*, *decoder_num_heads: int = 16*, *predict_feature_dim: int = 512*, *mlp_ratio: int = 4*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

Prompt decoder for MILAN.

This decoder is used in MILAN pretraining, which will not update these visible tokens from the encoder.

**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.

- **predict_feature_dim** (*int*) – The dimension of the feature to be predicted. Defaults to 512.
- **mlp_ratio** (*int*) – Ratio of mlp hidden dim to decoder's embedding dim. Defaults to 4.
- **norm_cfg** (*dict*) – Normalization layer. Defaults to LayerNorm.
- **init_cfg** (*Union[List[dict], dict], optional*) – Initialization config dict. Defaults to None.

**forward**(*x: torch.Tensor*, *ids_restore: torch.Tensor*, *ids_keep: torch.Tensor*, *ids_dump: torch.Tensor*) → torch.Tensor

Forward function.

**Parameters**

- **x** (*torch.Tensor*) – The input features, which is of shape (N, L, C).
- **ids_restore** (*torch.Tensor*) – The indices to restore these tokens to the original image.
- **ids_keep** (*torch.Tensor*) – The indices of tokens to be kept.
- **ids_dump** (*torch.Tensor*) – The indices of tokens to be masked.

**Returns**

**The reconstructed features, which is of shape** (N, L, C).

**Return type** torch.Tensor

class mmselfsup.models.necks.**MixMIMPretrainDecoder**(*num_patches: int = 196*, *patch_size: int = 16*, *in_chans: int = 3*, *embed_dim: int = 1024*, *encoder_stride: int = 32*, *decoder_embed_dim: int = 512*, *decoder_depth: int = 8*, *decoder_num_heads: int = 16*, *mlp_ratio: int = 4*, *norm_cfg: dict = {'eps': 1e-06, 'type': 'LN'}*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

Decoder for MixMIM Pretraining.

Some of the code is borrowed from *https://github.com/Sense-X/MixMIM*. # noqa

**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.
- **encoder_stride** (*int*) – The output stride of MixMIM backbone. Defaults to 32.
- **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.
- **init_cfg** (*Union[List[dict], dict], optional*) – Initialization config dict. Defaults to None.

**forward**(*x: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Forward function.

**Parameters**

- **x** (*torch.Tensor*) – The input features, which is of shape (N, L, C).
- **mask** (*torch.Tensor*) – The tensor to indicate which tokens a re masked.

**Returns**

**The reconstructed features, which is of shape** (N, L, C).

**Return type** torch.Tensor

**init_weights**() → None
Initialize position embedding and mask token of MixMIM decoder.

class mmselfsup.models.necks.**MoCoV2Neck**(*in_channels: int*, *hid_channels: int*, *out_channels: int*, *with_avg_pool: bool = True*, *init_cfg: Optional[Union[dict, List[dict]]] = 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: List[torch.Tensor]*) → List[torch.Tensor]
Forward function.

**Parameters** **x** (*List[torch.Tensor]*) – The feature map of backbone.

**Returns** The output features.

**Return type** List[torch.Tensor]

class mmselfsup.models.necks.**NonLinearNeck**(*in_channels: int*, *hid_channels: int*, *out_channels: int*, *num_layers: int = 2*, *with_bias: bool = False*, *with_last_bn: bool = True*, *with_last_bn_affine: bool = True*, *with_last_bias: bool = False*, *with_avg_pool: bool = True*, *vit_backbone: bool = False*, *norm_cfg: dict = {'type': 'SyncBN'}*, *init_cfg: Optional[Union[dict, List[dict]]] = [{'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.
- **vit_backbone** (*bool*) – The key to indicate whether the upstream backbone is ViT. Defaults to False.
- **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: Tuple[torch.Tensor]*) → List[torch.Tensor]

Forward function.

**Parameters** **x** (*List[torch.Tensor]*) – The feature map of backbone.

**Returns** The output features.

**Return type** List[torch.Tensor]

class mmselfsup.models.necks.**ODCNeck**(*in_channels: int*, *hid_channels: int*, *out_channels: int*, *with_avg_pool: bool = True*, *norm_cfg: dict = {'type': 'SyncBN'}*, *init_cfg: Optional[Union[dict, List[dict]]] = [{'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: List[torch.Tensor]*) → List[torch.Tensor]

Forward function.

**Parameters** **x** (*List[torch.Tensor]*) – The feature map of backbone.

**Returns** The output features.

**Return type** List[torch.Tensor]

class mmselfsup.models.necks.**RelativeLocNeck**(*in_channels: int*, *out_channels: int*, *with_avg_pool: bool = True*, *norm_cfg: dict = {'type': 'BN1d'}*, *init_cfg: Optional[Union[dict, List[dict]]] = [{'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: List[torch.Tensor]*) → List[torch.Tensor]

Forward function.

**Parameters** **x** (*List[torch.Tensor]*) – The feature map of backbone.

**Returns** The output features.

**Return type** List[torch.Tensor]

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

Forward function.

class mmselfsup.models.necks.**SwAVNeck**(*in_channels: int*, *hid_channels: int*, *out_channels: int*, *with_avg_pool: bool = True*, *with_l2norm: bool = True*, *norm_cfg: dict = {'type': 'SyncBN'}*, *init_cfg: Optional[Union[dict, List[dict]]] = [{'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: List[torch.Tensor]*) → List[torch.Tensor]
Forward function.

**Parameters x** (*List[torch.Tensor]*) – list of feature maps, len(x) according to len(num_crops).

**Returns** The projection vectors.

**Return type** List[torch.Tensor]

**forward_projection**(*x: torch.Tensor*) → torch.Tensor
Compute projection.

**Parameters x** (*torch.Tensor*) – The feature vectors after pooling.

**Returns** The output features with projection or L2-norm.

**Return type** torch.Tensor

## 37.4 heads

class mmselfsup.models.heads.**BEiTV1Head**(*embed_dims: int, num_embed: int, loss: dict, init_cfg: Optional[Union[dict, List[dict]]] = {'bias': 0, 'layer': 'Linear', 'std': 0.02, 'type': 'TruncNormal'}*)

Pretrain Head for BEiT v1.

Compute the logits and the cross entropy loss.

**Parameters**

- **embed_dims** (*int*) – The dimension of embedding.
- **num_embed** (*int*) – The number of classification types.
- **loss** (*dict*) – The config of loss.
- **init_cfg** (*dict or List[dict], optional*) – Initialization config dict. Defaults to None.

**forward**(*feats: torch.Tensor, target: torch.Tensor, mask: torch.Tensor*) → torch.Tensor
Generate loss.

**Parameters**

- **feats** (*torch.Tensor*) – Features from backbone.
- **target** (*torch.Tensor*) – Target generated by target_generator.
- **mask** (*torch.Tensor*) – Generated mask for pretraing.

class mmselfsup.models.heads.**BEiTV2Head**(*embed_dims: int, num_embed: int, loss: dict, init_cfg: Optional[Union[dict, List[dict]]] = {'bias': 0, 'layer': 'Linear', 'std': 0.02, 'type': 'TruncNormal'}*)

Pretrain Head for BEiT.

Compute the logits and the cross entropy loss.

**Parameters**

- **embed_dims** (*int*) – The dimension of embedding.
- **num_embed** (*int*) – The number of classification types.

- **loss** (*dict*) – The config of loss.
- **init_cfg** (*dict or List[dict], optional*) – Initialization config dict. Defaults to None.

**forward**(*feats: torch.Tensor*, *feats_cls_pt: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Generate loss.

**Parameters**

- **feats** (*torch.Tensor*) – Features from backbone.
- **feats_cls_pt** (*torch.Tensor*) – Features from class late layers for pretraining.
- **target** (*torch.Tensor*) – Target generated by target_generator.
- **mask** (*torch.Tensor*) – Generated mask for pretraing.

class mmselfsup.models.heads.**CAEHead**(*loss: dict*, *init_cfg: Optional[Union[dict, List[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**

- **loss** (*dict*) – The config of loss.
- **tokenizer_path** (*str*) – The path of the tokenizer.
- **init_cfg** (*dict or List[dict], optional*) – Initialization config dict. Defaults to None.

**forward**(*logits: torch.Tensor*, *logits_target: torch.Tensor*, *latent_pred: torch.Tensor*, *latent_target: torch.Tensor*, *mask: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Generate loss.

**Parameters**

- **logits** (*torch.Tensor*) – Logits generated by decoder.
- **logits_target** (*img_target*) – Target generated by dalle for decoder prediction.
- **latent_pred** (*torch.Tensor*) – Latent prediction by regressor.
- **latent_target** (*torch.Tensor*) – Target for latent prediction, generated by teacher.

**Returns**

**The tuple of loss.**

- loss_main (torch.Tensor): Cross entropy loss.
- loss_align (torch.Tensor): MSE loss.

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

class mmselfsup.models.heads.**ClsHead**(*loss: dict*, *with_avg_pool: bool = False*, *in_channels: int = 2048*, *num_classes: int = 1000*, *vit_backbone: bool = False*, *init_cfg: Optional[Union[dict, List[dict]]] = [{'type': 'Normal', 'std': 0.01, 'layer': 'Linear'}, {'type': 'Constant', 'val': 1, 'layer': ['_BatchNorm', 'GroupNorm']}]*)

Simplest classifier head, with only one fc layer.

**Parameters**

- **loss** (*dict*) – Config of the loss.
- **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: Union[List[torch.Tensor], Tuple[torch.Tensor]]*, *label: torch.Tensor*) → torch.Tensor

Get the loss.

**Parameters**

- **x** (*List[Tensor] | Tuple[Tensor]*) – Feature maps of backbone, each tensor has shape (N, C, H, W).
- **label** (*torch.Tensor*) – The label for cross entropy loss.

**Returns** The cross entropy loss.

**Return type** torch.Tensor

**logits**(*x: Union[List[torch.Tensor], Tuple[torch.Tensor]]*) → List[torch.Tensor]

Get the logits before the cross_entropy loss.

This module is used to obtain the logits before the loss.

**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]

**class** mmselfsup.models.heads.**ContrastiveHead**(*loss: dict*, *temperature: float = 0.1*)

Head for contrastive learning.

The contrastive loss is implemented in this head and is used in SimCLR, MoCo, DenseCL, etc.

**Parameters**

- **loss** (*dict*) – Config dict for module of loss functions.
- **temperature** (*float*) – The temperature hyper-parameter that controls the concentration level of the distribution. Defaults to 0.1.

**forward**(*pos: torch.Tensor*, *neg: torch.Tensor*) → torch.Tensor

Forward function to compute contrastive loss.

**Parameters**

- **pos** (*torch.Tensor*) – Nx1 positive similarity.
- **neg** (*torch.Tensor*) – Nxk negative similarity.

**Returns** The contrastive loss.

**Return type** torch.Tensor

**class** mmselfsup.models.heads.**LatentCrossCorrelationHead**(*in_channels: int*, *loss: dict*)

Head for latent feature cross correlation.

Part of the code is borrowed from script.

**Parameters**

- **in_channels** (*int*) – Number of input channels.
- **loss** (*dict*) – Config dict for module of loss functions.

**forward**(*input: torch.Tensor*, *target: torch.Tensor*) → torch.Tensor
Forward head.

**Parameters**

- **input** (*torch.Tensor*) – NxC input features.
- **target** (*torch.Tensor*) – NxC target features.

**Returns** The cross correlation loss.

**Return type** torch.Tensor

class mmselfsup.models.heads.**LatentPredictHead**(*loss: dict*, *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**

- **loss** (*dict*) – Config dict for the loss.
- **predictor** (*dict*) – Config dict for the predictor.

**forward**(*input: torch.Tensor*, *target: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Forward head.

**Parameters**

- **input** (*torch.Tensor*) – NxC input features.
- **target** (*torch.Tensor*) – NxC target features.

**Returns** The latent predict loss.

**Return type** torch.Tensor

class mmselfsup.models.heads.**MAEPretrainHead**(*loss: dict*, *norm_pix: bool = False*, *patch_size: int = 16*)
Pre-training head for MAE.

**Parameters**

- **loss** (*dict*) – Config of loss.
- **norm_pix_loss** (*bool*) – Whether or not normalize target. Defaults to False.
- **patch_size** (*int*) – Patch size. Defaults to 16.

**construct_target**(*target: torch.Tensor*) → torch.Tensor
Construct the reconstruction target.

In addition to splitting images into tokens, this module will also normalize the image according to `norm_pix`.

**Parameters** **target** (*torch.Tensor*) – Image with the shape of B x 3 x H x W

**Returns** Tokenized images with the shape of B x L x C

**Return type** torch.Tensor

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Forward function of MAE head.

**Parameters**

- **pred** (*torch.Tensor*) – The reconstructed image.
- **target** (*torch.Tensor*) – The target image.
- **mask** (*torch.Tensor*) – The mask of the target image.

**Returns** The reconstruction loss.

**Return type** torch.Tensor

**patchify**(*imgs: torch.Tensor*) → torch.Tensor
Split images into non-overlapped patches.

**Parameters imgs** (*torch.Tensor*) – A batch of images, of shape B x H x W x C.

**Returns** Patchified images. The shape is B x L x D.

**Return type** torch.Tensor

**unpatchify**(*x: torch.Tensor*) → torch.Tensor
Combine non-overlapped patches into images.

**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.**MILANPretrainHead**(*loss: dict*)
MILAN pretrain head.

**Parameters loss** (*dict*) – Config of loss.

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: Optional[torch.Tensor] = None*) → torch.Tensor
Forward function.

**Parameters**

- **pred** (*torch.Tensor*) – Predicted features, of shape (N, L, D).
- **target** (*torch.Tensor*) – Target features, of shape (N, L, D).
- **mask** (*torch.Tensor*) – The mask of the target image of shape.

**Returns** the reconstructed loss.

**Return type** torch.Tensor

class mmselfsup.models.heads.**MaskFeatPretrainHead**(*loss: dict*)
Pre-training head for MaskFeat.

It computes reconstruction loss between prediction and target in masked region.

**Parameters loss** (*dict*) – Config dict for module of loss functions.

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Forward head.

**Parameters**

- **latent** (*torch.Tensor*) – Predictions, which is of shape B x (1 + L) x C.
- **target** (*torch.Tensor*) – Hog features, which is of shape B x L x C.
- **mask** (*torch.Tensor*) – The mask of the hog features, which is of shape B x H x W.

**Returns** The loss tensor.

**Return type** torch.Tensor

class mmselfsup.models.heads.**MixMIMPretrainHead**(*loss: dict*, *norm_pix: bool = False*, *patch_size: int = 16*)

MixMIM pretrain head.

**Parameters**

- **loss** (*dict*) – Config of loss.
- **norm_pix_loss** (*bool*) – Whether or not normalize target. Defaults to False.
- **patch_size** (*int*) – Patch size. Defaults to 16.

**forward**(*x_rec: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Forward function of MixMIM head.

**Parameters**

- **pred** (*torch.Tensor*) – The reconstructed image.
- **target** (*torch.Tensor*) – The target image.
- **mask** (*torch.Tensor*) – The mask of the target image.

**Returns** The reconstruction loss.

**Return type** torch.Tensor

class mmselfsup.models.heads.**MoCoV3Head**(*predictor: dict*, *loss: dict*, *temperature: float = 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.
- **loss** (*dict*) – Config dict for module of loss functions.
- **temperature** (*float*) – The temperature hyper-parameter that controls the concentration level of the distribution. Defaults to 1.0.

**forward**(*base_out: torch.Tensor*, *momentum_out: torch.Tensor*) → torch.Tensor

Forward head.

**Parameters**

- **base_out** (*torch.Tensor*) – NxC features from base_encoder.
- **momentum_out** (*torch.Tensor*) – NxC features from momentum_encoder.

**Returns** The loss tensor.

**Return type** torch.Tensor

class mmselfsup.models.heads.**MultiClsHead**(*backbone: str = 'resnet50'*, *in_indices: Sequence[int] = (0, 1, 2, 3, 4)*, *pool_type: str = 'adaptive'*, *num_classes: int = 1000*, *loss: dict = {'loss_weight': 1.0, 'type': 'mmcls.CrossEntropyLoss'}*, *with_last_layer_unpool: bool = False*, *cal_acc: bool = False*, *topk: Union[int, Tuple[int]] = (1)*, *norm_cfg: dict = {'type': 'BN'}*, *init_cfg: Union[dict, List[dict]] = [{'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**

- **backbone** (*str*) – Specify which backbone to use, only support ResNet50. Defaults to 'resnet50'.
- **in_indices** (*Sequence[int]*) – Input from which stages. Defaults to (0, 1, 2, 3, 4).
- **pool_type** (*str*) – 'adaptive' or 'specified'. If set to 'adaptive', use adaptive average pooling, otherwise use specified pooling params. Defaults to 'adaptive'.
- **num_classes** (*int*) – Number of classes. Defaults to 1000.
- **loss** (*dict*) – The dict of loss information. Defaults to 'mmcls.models.CrossEntro): Whether to unpool the features from last layer. Defaults to False.
- **cal_acc** (*bool*) – Whether to calculate accuracy during training. If you use batch augmentations like Mixup and CutMix during training, it is pointless to calculate accuracy. Defaults to False.
- **topk** (*int | Tuple[int]*) – Top-k accuracy. Defaults to `(1, )`.
- **norm_cfg** (*dict*) – Dict to construct and config norm layer. Defaults to `dict(type='BN')`.
- **init_cfg** (*dict or List[dict]*) – Initialization config dict. Defaults to `[ dict(type='Normal', std=0.01, layer='Linear'), dict(type='Constant', val=1, layer=['_BatchNorm', 'GroupNorm']) ]`

**forward**(*feats: Union[list, tuple]*) → list

Compute multi-head scores.

**Parameters** **feats** (*Sequence[torch.Tensor]*) – Feature maps of backbone, each tensor has shape (N, C, H, W).

**Returns** A list of class scores.

**Return type** List[torch.Tensor]

**loss**(*feats: Sequence[torch.Tensor]*, *data_samples: List[mmcls.structures.cls_data_sample.ClsDataSample]*, ***kwargs*) → dict

Calculate losses from the extracted features.

**Parameters**

- **x** (*Sequence[torch.Tensor]*) – Feature maps of backbone, each tensor has shape (N, C, H, W).
- **gt_label** (*torch.Tensor*) – The ground truth label.

**Returns** Dict of loss and accuracy.

**Return type** Dict[str, torch.Tensor]

**predict**(*feats: Sequence[torch.Tensor]*, *data_samples: List[mmcls.structures.cls_data_sample.ClsDataSample]*) → List[mmcls.structures.cls_data_sample.ClsDataSample]

Inference without augmentation.

**Parameters**

- **feats** (*tuple[Tensor]*) – The extracted features.
- **data_samples** (*List[BaseDataElement], optional*) – The annotation data of every samples. If not None, set `pred_label` of the input data samples.

**Returns**

**The data samples containing annotation,** prediction, etc.

**Return type** List[BaseDataElement]

`class mmselfsup.models.heads.`**`SimMIMHead`**(*patch_size: int*, *loss: dict*)

Pretrain Head for SimMIM.

**Parameters**

- **`patch_size`** (*`int`*) – Patch size of each token.
- **`loss`** (*`dict`*) – The config for loss.

**`forward`**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Forward function of MAE Loss.

This method will expand mask to the size of the original image.

**Parameters**

- **`pred`** (*`torch.Tensor`*) – The reconstructed image.
- **`target`** (*`torch.Tensor`*) – The target image.
- **`mask`** (*`torch.Tensor`*) – The mask of the target image.

**Returns** The reconstruction loss.

**Return type** torch.Tensor

`class mmselfsup.models.heads.`**`SwAVHead`**(*loss: dict*)

Head for SwAV.

**Parameters** **`loss`** (*`dict`*) – Config dict for module of loss functions.

**`forward`**(*pred: torch.Tensor*) → torch.Tensor

Forward function of SwAV head.

**Parameters** **`pred`** (*`torch.Tensor`*) – NxC input features.

**Returns** The SwAV loss.

**Return type** torch.Tensor

## 37.5 losses

`class mmselfsup.models.losses.`**`BEiTLoss`**

Loss function for BEiT.

The BEiTLoss supports 2 diffenrent logits shared 1 target, like BEiT v2.

**`forward`**(*logits: Union[Tuple[torch.Tensor], torch.Tensor]*, *target: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward function of BEiT Loss.

**Parameters**

- **`logits`** (*`torch.Tensor`*) – The outputs from the decoder.
- **`target`** (*`torch.Tensor`*) – The targets generated by dalle.

**Returns** The main loss.

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

**class** mmselfsup.models.losses.**CAELoss**(*lambd: float*)

Loss function for CAE.

Compute the align loss and the main loss.

**Parameters** **lambd** (*float*) – The weight for the align loss.

**forward**(*logits: torch.Tensor*, *target: torch.Tensor*, *latent_pred: torch.Tensor*, *latent_target: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Forward function of CAE Loss.

**Parameters**

- **logits** (*torch.Tensor*) – The outputs from the decoder.
- **target** (*torch.Tensor*) – The targets generated by dalle.
- **latent_pred** (*torch.Tensor*) – The latent prediction from the regressor.
- **latent_target** (*torch.Tensor*) – The latent target from the teacher network.

**Returns** The main loss and align loss.

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

**class** mmselfsup.models.losses.**CosineSimilarityLoss**(*shift_factor: float = 0.0*, *scale_factor: float = 1.0*)

Cosine similarity loss function.

Compute the similarity between two features and optimize that similarity as loss.

**Parameters**

- **shift_factor** (*float*) – The shift factor of cosine similarity. Default: 0.0.
- **scale_factor** (*float*) – The scale factor of cosine similarity. Default: 1.0.

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: Optional[torch.Tensor] = None*) → torch.Tensor

Forward function of cosine similarity loss.

**Parameters**

- **pred** (*torch.Tensor*) – The predicted features.
- **target** (*torch.Tensor*) – The target features.

**Returns** The cosine similarity loss.

**Return type** torch.Tensor

**class** mmselfsup.models.losses.**CrossCorrelationLoss**(*lambd: float = 0.0051*)

Cross correlation loss function.

Compute the on-diagnal and off-diagnal loss.

**Parameters** **lambd** (*float*) – The weight for the off-diag loss.

**forward**(*cross_correlation_matrix: torch.Tensor*) → torch.Tensor

Forward function of cross correlation loss.

**Parameters** **cross_correlation_matrix** (*torch.Tensor*) – The cross correlation matrix.

**Returns** cross correlation loss.

**Return type** torch.Tensor

**off_diagonal**(*x: torch.Tensor*) → torch.Tensor

Rreturn a flattened view of the off-diagonal elements of a square matrix.

class mmselfsup.models.losses.**MAEReconstructionLoss**

Loss function for MAE.

Compute the loss in masked region.

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Forward function of MAE Loss.

**Parameters**

- **pred** (*torch.Tensor*) – The reconstructed image.
- **target** (*torch.Tensor*) – The target image.
- **mask** (*torch.Tensor*) – The mask of the target image.

**Returns** The reconstruction loss.

**Return type** torch.Tensor

class mmselfsup.models.losses.**PixelReconstructionLoss**(*criterion: str*, *channel: Optional[int] = None*)

Loss for the reconstruction of pixel in Masked Image Modeling.

This module measures the distance between the target image and the reconstructed image and compute the loss to optimize the model. Currently, This module only provides L1 and L2 loss to penalize the reconstructed error. In addition, a mask can be passed in the `forward` function to only apply loss on visible region, like that in MAE.

**Parameters**

- **criterion** (*str*) – The loss the penalize the reconstructed error. Currently, only supports L1 and L2 loss
- **channel** (*int, optional*) – The number of channels to average the reconstruction loss. If not None, the reconstruction loss will be divided by the channel. Defaults to None.

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: Optional[torch.Tensor] = None*) → torch.Tensor

Forward function to compute the reconstrction loss.

**Parameters**

- **pred** (*torch.Tensor*) – The reconstructed image.
- **target** (*torch.Tensor*) – The target image.
- **mask** (*torch.Tensor*) – The mask of the target image.

**Returns** The reconstruction loss.

**Return type** torch.Tensor

class mmselfsup.models.losses.**SimMIMReconstructionLoss**(*encoder_in_channels: int*)

Loss function for MAE.

Compute the loss in masked region.

**Parameters** **encoder_in_channels** (*int*) – Number of input channels for encoder.

**forward**(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Forward function of MAE Loss.

**Parameters**

- **pred** (*torch.Tensor*) – The reconstructed image.
- **target** (*torch.Tensor*) – The target image.
- **mask** (*torch.Tensor*) – The mask of the target image.

**Returns** The reconstruction loss.

**Return type** torch.Tensor

class mmselfsup.models.losses.**SwAVLoss**(*feat_dim: int*, *sinkhorn_iterations: int = 3*, *epsilon: float = 0.05*, *temperature: float = 0.1*, *crops_for_assign: List[int] = [0, 1]*, *num_crops: List[int] = [2]*, *num_prototypes: int = 3000*, *init_cfg: Optional[Union[dict, List[dict]]] = None*)

The Loss for SwAV.

This Loss contains clustering and sinkhorn algorithms to compute Q codes. Part of the code is borrowed from script. The queue is built in *engine/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: torch.Tensor*) → torch.Tensor

Forward function of SwAV loss.

**Parameters** **x** (*torch.Tensor*) – NxC input features.

**Returns** The returned loss.

**Return type** torch.Tensor

## 37.6 memories

class mmselfsup.models.memories.**ODCMemory**(*length: int*, *feat_dim: int*, *momentum: float*, *num_classes: int*, *min_cluster: int*, ***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 the 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**() → None
Deal with small clusters.

**init_memory**(*feature: numpy.ndarray*, *label: numpy.ndarray*) → None
Initialize memory modules.

**update_centroids_memory**(*cinds: Optional[List] = None*) → None
Update centroids memory.

**update_samples_memory**(*idx: torch.Tensor*, *feature: torch.Tensor*) → torch.Tensor
Update samples memory.

class mmselfsup.models.memories.**SimpleMemory**(*length: int*, *feat_dim: int*, *momentum: float*, ***kwargs*)
Simple feature memory bank.

This module includes the memory bank that stores running average features of all samples in the dataset. It is used in algorithms like NPID.

**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**(*idx: torch.Tensor*, *feature: torch.Tensor*) → None
Update features in the memory bank.

**Parameters**

- **idx** (*torch.Tensor*) – Indices for the batch of features.
- **feature** (*torch.Tensor*) – Batch of features.

## 37.7 target_generators

class mmselfsup.models.target_generators.**CLIPGenerator**(*tokenizer_path: str*)
Get the features and attention from the last layer of CLIP.

This module is used to generate target features in masked image modeling.

**Parameters** **tokenizer_path** (*str*) – The path of the checkpoint of CLIP.

**forward**(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Get the features and attention from the last layer of CLIP.

**Parameters** **x** (*torch.Tensor*) – The input image, which is of shape (N, 3, H, W).

**Returns** The features and attention from the last layer of CLIP, which are of shape (N, L, C) and (N, L, L), respectively.

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

class mmselfsup.models.target_generators.**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*, *init_cfg: Optional[Union[dict, List[dict]]] = 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.target_generators.**HOGGenerator**(*nbins: int = 9*, *pool: int = 8*, *gaussian_window: int = 16*)

Generate HOG feature for images.

This module is used in MaskFeat to generate HOG feature. The code is modified from file slow-fast/models/operators.py. Here is the link of HOG wikipedia.

**Parameters**

- **nbins** (*int*) – Number of bin. Defaults to 9.
- **pool** (*float*) – Number of cell. Defaults to 8.
- **gaussian_window** (*int*) – Size of gaussian kernel. Defaults to 16.

**forward**(*x: torch.Tensor*) → torch.Tensor

Generate hog feature for each batch images.

**Parameters** **x** (*torch.Tensor*) – Input images of shape (N, 3, H, W).

**Returns** Hog features.

**Return type** torch.Tensor

**generate_hog_image**(*hog_out: torch.Tensor*) → numpy.ndarray

Generate HOG image according to HOG features.

**get_gaussian_kernel**(*kernlen: int*, *std: int*) → torch.Tensor

Returns a 2D Gaussian kernel array.

**class** mmselfsup.models.target_generators.**VQKD**(*encoder_config: dict*, *decoder_config: Optional[dict] = None*, *num_embed: int = 8192*, *embed_dims: int = 32*, *decay: float = 0.99*, *beta: float = 1.0*, *quantize_kmeans_init: bool = True*, *init_cfg: Optional[dict] = None*)

Vector-Quantized Knowledge Distillation.

The module only contains encoder and VectorQuantizer part Modified from https://github.com/microsoft/unilm/blob/master/beit2/modeling_vqkd.py

**Parameters**

- **encoder_config** (*dict*) – The config of encoder.
- **decoder_config** (*dict, optional*) – The config of decoder. Currently, VQKD only support to build encoder. Defaults to None.
- **num_embed** (*int*) – Number of embedding vectors in the codebook. Defaults to 8192.
- **embed_dims** (*int*) – The dimension of embedding vectors in the codebook. Defaults to 32.
- **decay** (*float*) – The decay parameter of EMA. Defaults to 0.99.
- **beta** (*float*) – The mutiplier for VectorQuantizer loss. Defaults to 1.

- **quantize_kmeans_init** (*bool*) – Whether to use k-means to initialize the VectorQuantizer. Defaults to True.
- **init_cfg** (*dict or List[dict], optional*) – Initialization config dict. Defaults to None.

**encode**(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]
Encode the input images and get corresponding results.

**forward**(*x: torch.Tensor*) → torch.Tensor
The forward function.

Currently, only support to get tokens.

**get_tokens**(*x: torch.Tensor*) → dict
Get tokens for beit pre-training.

## 37.8 utils

class mmselfsup.models.utils.**CAEDataPreprocessor**(*mean: Optional[Sequence[Union[int, float]]] = None*, *std: Optional[Sequence[Union[int, float]]] = None*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_blocking: Optional[bool] = False*)

Image pre-processor for CAE.

Compared with the `mmselfsup.SelfSupDataPreprocessor`, this module will normalize the prediction image and target image with different normalization parameters.

**forward**(*data: dict*, *training: bool = False*) → Tuple[List[torch.Tensor], Optional[list]]
Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

**Parameters**

- **data** (*dict*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

**Returns** Data in the same format as the model input.

**Return type** Tuple[torch.Tensor, Optional[list]]

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
Forward function.

class mmselfsup.models.utils.**CosineEMA**(*model: torch.nn.modules.module.Module*, *momentum: float = 0.996*, *end_momentum: float = 1.0*, *interval: int = 1*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)

CosineEMA is implemented for updating momentum parameter, used in BYOL, MoCoV3, etc.

The momentum parameter is updated with cosine annealing, including momentum adjustment following:

$$m = m_1 - (m_1 - m_0) * (cos(pi * k/K) + 1)/2$$

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

**Parameters**

- **model** (*nn.Module*) – The model to be averaged.
- **momentum** (*float*) – The momentum used for updating ema parameter. Ema's parameter are updated with the formula: *averaged_param = momentum * averaged_param + (1-momentum) * source_param*. Defaults to 0.996.
- **end_momentum** (*float*) – The end momentum value for cosine annealing. Defaults to 1.
- **interval** (*int*) – Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) – If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) – if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

**avg_func**(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*) → None
Compute the moving average of the parameters using the cosine momentum strategy.

**Parameters**

- **averaged_param** (*Tensor*) – The averaged parameters.
- **source_param** (*Tensor*) – The source parameters.
- **steps** (*int*) – The number of times the parameters have been updated.

**Returns** The averaged parameters.

**Return type** Tensor

class mmselfsup.models.utils.**Extractor**(*extract_dataloader: Union[torch.utils.data.dataloader.DataLoader, dict]*, *seed: Optional[int] = None*, *dist_mode: bool = False*, *pool_cfg: Optional[dict] = None*, ***kwargs*)

Feature extractor.

The extractor support to build its own DataLoader, customized models, pooling type. It also has distributed and non-distributed mode.

**Parameters**

- **extract_dataloader** (*dict*) – A dict to build DataLoader object.
- **seed** (*int, optional*) – Random seed. Defaults to None.
- **dist_mode** (*bool*) – Use distributed extraction or not. Defaults to False.
- **pool_cfg** (*dict, optional*) – The configs of pooling. Defaults to dict(type='AvgPool2d', output_size=1).

class mmselfsup.models.utils.**GatherLayer**(**args*, ***kwargs*)

Gather tensors from all process, supporting backward propagation.

static **backward**(*ctx: Any*, **grads: torch.Tensor*) → torch.Tensor

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: Any*, *input: torch.Tensor*) → Tuple[List]

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.**MultiPooling**(*pool_type: str = 'adaptive'*, *in_indices: tuple = (0)*, *backbone: str = '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: Union[List, Tuple]*) → None
Forward function.

class mmselfsup.models.utils.**MultiPrototypes**(*output_dim: int*, *num_prototypes: List[int]*)
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: torch.Tensor*) → List[torch.Tensor]
Run forward for every prototype.

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
Forward function.

class mmselfsup.models.utils.**NormEMAVectorQuantizer**(*num_embed: int*, *embed_dims: int*, *beta: float*, *decay: float = 0.99*, *statistic_code_usage: bool = True*, *kmeans_init: bool = True*, *codebook_init_path: Optional[str] = None*)

Normed EMA vector quantizer module.

**Parameters**

- **num_embed** (*int*) – Number of embedding vectors in the codebook. Defaults to 8192.
- **embed_dims** (*int*) – The dimension of embedding vectors in the codebook. Defaults to 32.
- **beta** (*float*) – The mutiplier for VectorQuantizer embedding loss. Defaults to 1.
- **decay** (*float*) – The decay parameter of EMA. Defaults to 0.99.
- **statistic_code_usage** (*bool*) – Whether to use cluster_size to record statistic. Defaults to True.
- **kmeans_init** (*bool*) – Whether to use k-means to initialize the VectorQuantizer. Defaults to True.
- **codebook_init_path** (*str*) – The initialization checkpoint for codebook. Defaults to None.

**forward**(*z*)

Forward function.

class mmselfsup.models.utils.**PromptTransformerEncoderLayer**(*embed_dims: int*, *num_heads: int*, *feedforward_channels=<class 'int'>*, *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_cfg: Optional[Union[dict, List[dict]]] = None*)

Prompt Transformer Encoder Layer for MILAN.

This module is specific for the prompt encoder in MILAN. It will not update the visible tokens from the encoder.

**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 0.0.
- **attn_drop_rate** (*float*) – The drop out rate for attention layer. Defaults to 0.0.
- **drop_path_rate** (*float*) – Stochastic depth rate. Defaults to 0.0.
- **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')`.

- **batch_first** (*bool*) – Key, Query and Value are shape of (batch, n, embed_dim) or (n, batch, embed_dim). Defaults to False.
- **init_cfg** (*dict, optional*) – The Config for initialization. Defaults to None.

**forward**(*x: torch.Tensor*, *visible_tokens: torch.Tensor*, *ids_restore: torch.Tensor*) → torch.Tensor

Forward function for *PromptMultiheadAttention*.

**Parameters**

- **x** (*torch.Tensor*) – Mask token features with shape N x L_m x C.
- **visible_tokens** (*torch.Tensor*) – The visible tokens features from encoder with shape N x L_v x C.
- **ids_restore** (*torch.Tensor*) – The ids of all tokens in the original image with shape N x L.

**Returns** Output features with shape N x L x C.

**Return type** torch Tensor

**class** mmselfsup.models.utils.**RelativeLocDataPreprocessor**(*mean: Optional[Sequence[Union[int, float]]] = None*, *std: Optional[Sequence[Union[int, float]]] = None*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_blocking: Optional[bool] = False*)

Image pre-processor for Relative Location.

**forward**(*data: dict*, *training: bool = False*) → Tuple[List[torch.Tensor], Optional[list]]

Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

**Parameters**

- **data** (*dict*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

**Returns** Data in the same format as the model input.

**Return type** Tuple[torch.Tensor, Optional[list]]

**class** mmselfsup.models.utils.**RotationPredDataPreprocessor**(*mean: Optional[Sequence[Union[int, float]]] = None*, *std: Optional[Sequence[Union[int, float]]] = None*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_blocking: Optional[bool] = False*)

Image pre-processor for Relative Location.

**forward**(*data: dict*, *training: bool = False*) → Tuple[List[torch.Tensor], Optional[list]]

Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

**Parameters**

- **data** (*dict*) – data sampled from dataloader.

- **training** (*bool*) – Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

**Returns** Data in the same format as the model input.

**Return type** Tuple[torch.Tensor, Optional[list]]

class mmselfsup.models.utils.**SelfSupDataPreprocessor**(*mean: Optional[Sequence[Union[int, float]]] = None*, *std: Optional[Sequence[Union[int, float]]] = None*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_blocking: Optional[bool] = False*)

Image pre-processor for operations, like normalization and bgr to rgb.

Compared with the `mmengine.ImgDataPreprocessor`, this module treats each item in *inputs* of input data as a list, instead of torch.Tensor.

**forward**(*data: dict*, *training: bool = False*) → Tuple[List[torch.Tensor], Optional[list]]

Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

**Parameters**

- **data** (*dict*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

**Returns** Data in the same format as the model input.

**Return type** Tuple[torch.Tensor, Optional[list]]

class mmselfsup.models.utils.**Sobel**

Sobel layer.

The layer reduces channels from 3 to 2.

**forward**(*x: torch.Tensor*) → torch.Tensor

Run sobel layer.

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
Forward function.

class mmselfsup.models.utils.**TwoNormDataPreprocessor**(*mean: Optional[Sequence[Union[int, float]]] = None*, *std: Optional[Sequence[Union[int, float]]] = None*, *second_mean: Optional[Sequence[Union[int, float]]] = None*, *second_std: Optional[Sequence[Union[int, float]]] = None*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *rgb_to_bgr: bool = False*, *non_blocking: Optional[bool] = False*)

Image pre-processor for CAE, BEiT v1/v2, etc.

Compared with the `mmselfsup.SelfSupDataPreprocessor`, this module will normalize the prediction image and target image with different normalization parameters.

**Parameters**

- **mean** (*Sequence[float or int], optional*) – The pixel mean of image channels. If `bgr_to_rgb=True` it means the mean value of R, G, B channels. If the length of *mean* is 1, it means all channels have the same mean value, or the input is a gray image. If it is not specified, images will not be normalized. Defaults None.
- **std** (*Sequence[float or int], optional*) – The pixel standard deviation of image channels. If `bgr_to_rgb=True` it means the standard deviation of R, G, B channels. If the length of *std* is 1, it means all channels have the same standard deviation, or the input is a gray image. If it is not specified, images will not be normalized. Defaults None.
- **second_mean** (*Sequence[float or int], optional*) – The description is like `mean`, it can be customized for targe image. Defaults None.
- **second_std** (*Sequence[float or int], optional*) – The description is like `std`, it can be customized for targe image. Defaults None.
- **pad_size_divisor** (*int*) – The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **pad_value** (*float or int*) – The padded pixel value. Defaults to 0.
- **bgr_to_rgb** (*bool*) – whether to convert image from BGR to RGB. Defaults to False.
- **rgb_to_bgr** (*bool*) – whether to convert image from RGB to RGB. Defaults to False.
- **non_blocking** (*bool*) – Whether block current process when transferring data to device.

**forward**(*data: dict*, *training: bool = False*) → Tuple[List[torch.Tensor], Optional[list]]

Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

**Parameters**

- **data** (*dict*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

**Returns**

**Data in the same format as the** model input.

**Return type** Tuple[torch.Tensor, Optional[list]]

**class** mmselfsup.models.utils.**VideoDataPreprocessor**(*mean: Optional[Sequence[Union[int, float]]] = None*, *std: Optional[Sequence[Union[int, float]]] = None*, *pad_size_divisor: int = 1*, *pad_value: Union[float, int] = 0*, *bgr_to_rgb: bool = False*, *format_shape: str = 'NCHW'*)

Video pre-processor for operations, like normalization and bgr to rgb conversion .

Compared with the `mmaction.ActionDataPreprocessor`, this module treats each item in *inputs* of input data as a list, instead of torch.Tensor.

**Parameters**

- **mean** (*Sequence[float or int, optional*) – The pixel mean of channels of images or stacked optical flow. Defaults to None.
- **std** (*Sequence[float or int], optional*) – The pixel standard deviation of channels of images or stacked optical flow. Defaults to None.
- **pad_size_divisor** (*int*) – The size of padded image should be divisible by `pad_size_divisor`. Defaults to 1.
- **pad_value** (*float or int*) – The padded pixel value. Defaults to 0.
- **bgr_to_rgb** (*bool*) – Whether to convert image from BGR to RGB. Defaults to False.
- **format_shape** (*str*) – Format shape of input data. Defaults to `'NCHW'`.

**forward**(*data: dict*, *training: bool = False*) → Tuple[List[torch.Tensor], Optional[list]]

Performs normalizationpadding and bgr2rgb conversion based on `BaseDataPreprocessor`.

**Parameters**

- **data** (*dict*) – data sampled from dataloader.
- **training** (*bool*) – Whether to enable training time augmentation. If subclasses override this method, they can perform different preprocessing strategies for training and testing based on the value of `training`.

**Returns**

**Data in the same format** as the model input.

**Return type** Tuple[List[torch.Tensor], Optional[list]] mmselfsup.models.utils.**build_2d_sincos_position_embedding**(*patches_resolution: Union[int, Sequence[int]]*, *embed_dims: int*, *temperature: Optional[int] = 10000.0*, *cls_token: Optional[bool] = False*) → torch.Tensor

The function is to build position embedding for model to obtain the position information of the image patches.

**Parameters**

- **patches_resolution** (*Union[int, Sequence[int]]*) – The resolution of each patch.
- **embed_dims** (*int*) – The dimension of the embedding vector.
- **temperature** (*int, optional*) – The temperature parameter. Defaults to 10000.
- **cls_token** (*bool, optional*) – Whether to concatenate class token. Defaults to False.

**Returns** The position embedding vector.

**Return type** torch.Tensor

mmselfsup.models.utils.**build_clip_model**(*state_dict: dict*, *finetune: bool = False*, *average_targets: int = 1*) → torch.nn.modules.module.Module

Build the CLIP model.

**Parameters**

- **state_dict** (*dict*) – The pretrained state dict.
- **finetune** (*bool*) – Whether to fineturn the model.
- **average_targets** (*bool*) – Whether to average the target.

**Returns** The CLIP model.

**Return type** nn.Module

CHAPTER

THIRTYEIGHT

# MMSELFSUP.STRUCTURES

**class** mmselfsup.structures.**SelfSupDataSample**(**, metainfo: Optional[dict] = None, **kwargs*)

A data structure interface of MMSelfSup. They are used as interfaces between different components.

Meta field:

- `img_shape` (Tuple): The shape of the corresponding input image. Used for visualization.
- `ori_shape` (Tuple): The original shape of the corresponding image. Used for visualization.
- `img_path` (str): The path of original image.

Data field:

- `gt_label` (LabelData): The ground truth label of an image.
- `sample_idx` (InstanceData): The idx of an image in the dataset.
- `mask` (BaseDataElement): Mask used in masks image modeling.
- `pred_label` (LabelData): The predicted label.
- `pseudo_label` (InstanceData): Label used in pretext task, e.g. Relative Location.

### Examples

```
>>> import torch
>>> import numpy as np
>>> from mmengine.structure import InstanceData
>>> from mmselfsup.structures import SelfSupDataSample
```

```
>>> data_sample = SelfSupDataSample()
>>> gt_label = LabelData()
>>> gt_label.value = [1]
>>> data_sample.gt_label = gt_label
>>> len(data_sample.gt_label)
1
>>> print(data_sample)
<SelfSupDataSample(
    META INFORMATION
    DATA FIELDS
    gt_label: <InstanceData(
            META INFORMATION
            DATA FIELDS
            value: [1]
```

(continues on next page)

(continued from previous page)

```
        ) at 0x7f15c08f9d10>
    _gt_label: <InstanceData(
            META INFORMATION
            DATA FIELDS
            value: [1]
        ) at 0x7f15c08f9d10>
 ) at 0x7f15c077ef10>
```

```
>>> idx = InstanceData()
>>> idx.value = [0]
>>> data_sample = SelfSupDataSample(idx=idx)
>>> assert 'idx' in data_sample
```

```
>>> data_sample = SelfSupDataSample()
>>> mask = dict(value=np.random.rand(48, 48))
>>> mask = PixelData(**mask)
>>> data_sample.mask = mask
>>> assert 'mask' in data_sample
>>> assert 'value' in data_sample.mask
```

```
>>> data_sample = SelfSupDataSample()
>>> pred_label = dict(pred_label=[3])
>>> pred_label = LabelData(**pred_label)
>>> data_sample.pred_label = pred_label
>>> print(data_sample)
<SelfSupDataSample(
    META INFORMATION
    DATA FIELDS
    _pred_label: <InstanceData(
            META INFORMATION
            DATA FIELDS
            pred_label: [3]
        ) at 0x7f15c06a3990>
    pred_label: <InstanceData(
            META INFORMATION
            DATA FIELDS
            pred_label: [3]
        ) at 0x7f15c06a3990>
) at 0x7f15c07b8bd0>
```

CHAPTER

THIRTYNINE

# MMSELFSUP.VISUALIZATION

**class** mmselfsup.visualization.**SelfSupVisualizer**(*name: str = 'visualizer'*, *image: Optional[numpy.ndarray] = None*, *vis_backends: Optional[List[Dict]] = None*, *save_dir: Optional[str] = None*, *line_width: Union[int, float] = 3*, *alpha: Union[int, float] = 0.8*)

MMSelfSup Visualizer.

**Parameters**

- **name** (*str*) – Name of the instance. Defaults to 'visualizer'.
- **image** (*np.ndarray, optional*) – the origin image to draw. The format should be RGB. Defaults to None.
- **vis_backends** (*list, optional*) – Visual backend config list. Defaults to None.
- **save_dir** (*str, optional*) – Save file dir for all storage backends. If it is None, the backend storage will not save any data.
- **line_width** (*int, float*) – The linewidth of lines. Defaults to 3.
- **alpha** (*int, float*) – The transparency of boxes or mask. Defaults to 0.8.

### Examples

```
>>> import numpy as np
>>> import torch
>>> from mmengine.structures import InstanceData
>>> from mmselfsup.structures import SelfSupDataSample
>>> from mmselfsup.visualization import SelfSupVisualizer
```

```
>>> selfsup_visualizer = SelfSupVisualizer()
>>> image = np.random.randint(0, 256,
...                     size=(10, 12, 3)).astype('uint8')
>>> pseudo_label = InstanceData()
>>> pseudo_label.patch_box = torch.Tensor([[1, 2, 2, 5]])
>>> gt_selfsup_data_sample = SelfSupDataSample()
>>> gt_selfsup_data_sample.pseudo_label = pseudo_label
>>> selfsup_visualizer.add_datasample('image', image,
...                         gt_selfsup_data_sample)
>>> selfsup_visualizer.add_datasample(
...                        'image', image, gt_selfsup_data_sample,
```

(continues on next page)

(continued from previous page)

```
...                          out_file='out_file.jpg')
>>> selfsup_visualizer.add_datasample(
...                         'image', image, gt_selfsup_data_sample,
...                          show=True)
>>> pseudo_label = InstanceData()
>>> pseudo_label.patch_box = torch.Tensor([[1, 2, 2, 5]])
>>> pred_selfsup_data_sample = SelfSupDataSample()
>>> pred_selfsup_data_sample.pseudo_label = pseudo_label
>>> selfsup_visualizer.add_datasample('image', image,
...                          gt_selfsup_data_sample,
...                          pred_selfsup_data_sample)
```

**add_datasample**(*name: str*, *image: numpy.ndarray*, *gt_sample: Optional[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*] = None*, *pred_sample: Optional[*mmselfsup.structures.selfsup_data_sample.SelfSupDataSample*] = None*, *draw_gt: bool = True*, *draw_pred: bool = True*, *show: bool = False*, *wait_time: float = 0*, *out_file: Optional[str] = None*, *step: int = 0*) → None

Draw datasample and save to all backends.

- If GT and prediction are plotted at the same time, they are displayed in a stitched image where the left image is the ground truth and the right image is the prediction.
- If `show` is True, all storage backends are ignored, and the images will be displayed in a local window.
- If `out_file` is specified, the drawn image will be saved to `out_file`. t is usually used when the display is not available.

**Parameters**

- **name** (*str*) – The image identifier.
- **image** (*np.ndarray*) – The image to draw.
- **gt_sample** (`SelfSupDataSample`, optional) – GT SelfSupDataSample. Defaults to None.
- **pred_sample** (`SelfSupDataSample`, optional) – Prediction SelfSupDataSample. Defaults to None.
- **draw_gt** (*bool*) – Whether to draw GT SelfSupDataSample. Default to True.
- **draw_pred** (*bool*) – Whether to draw Prediction SelfSupDataSample. Defaults to True.
- **show** (*bool*) – Whether to display the drawn image. Default to False.
- **wait_time** (*float*) – The interval of show (s). Defaults to 0.
- **out_file** (*str*) – Path to output file. Defaults to None.
- **step** (*int*) – Global step value to record. Defaults to 0.

# CHAPTER FORTY

# MMSELFSUP.UTILS

**class** mmselfsup.utils.**AliasMethod**(*probs: torch.Tensor*)
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** (*torch.Tensor*) – Sampling probabilities.

**draw**(*N: int*) → None
Draw N samples from multinomial.

**Parameters** **N** (*int*) – Number of samples.

**Returns** Samples.

**Return type** torch.Tensor

mmselfsup.utils.**batch_shuffle_ddp**(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]
Batch shuffle, for making use of BatchNorm.

**Parameters** **x** (*torch.Tensor*) – Data in each GPU.

**Returns**

**Output of shuffle operation.**

- x_gather[idx_this]: Shuffled data.
- idx_unshuffle: Index for restoring.

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

mmselfsup.utils.**batch_unshuffle_ddp**(*x: torch.Tensor*, *idx_unshuffle: torch.Tensor*) → torch.Tensor
Undo batch shuffle.

**Parameters**

- **x** (*torch.Tensor*) – Data in each GPU.
- **idx_unshuffle** (*torch.Tensor*) – Index for restoring.

**Returns** Output of unshuffle operation.

**Return type** torch.Tensor

mmselfsup.utils.**collect_env**()
Collect the information of the running environments.

mmselfsup.utils.**concat_all_gather**(*tensor: torch.Tensor*) → torch.Tensor
Performs all_gather operation on the provided tensors.

**Parameters** **tensor** (*torch.Tensor*) – Tensor to be broadcast from current process.

**Returns** The concatnated tensor.

**Return type** torch.Tensor

`mmselfsup.utils.`**`dist_forward_collect`**(*func: object*, *data_loader: torch.utils.data.dataloader.DataLoader*, *length: int*) → dict

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.
- **data_loader** (*DataLoader*) – the torch DataLoader to yield data.
- **length** (*int*) – Expected length of output arrays.

**Returns** The collected outputs.

**Return type** Dict[str, torch.Tensor]

`mmselfsup.utils.`**`distributed_sinkhorn`**(*out: torch.Tensor*, *sinkhorn_iterations: int*, *world_size: int*, *epsilon: float*) → torch.Tensor

Apply the distributed sinknorn optimization on the scores matrix to find the assignments.

**Parameters**

- **out** (*torch.Tensor*) – The scores matrix
- **sinkhorn_iterations** (*int*) – Number of iterations in Sinkhorn-Knopp algorithm.
- **world_size** (*int*) – The world size of the process group.
- **epsilon** (*float*) – regularization parameter for Sinkhorn-Knopp algorithm.

**Returns** Output of sinkhorn algorithm.

**Return type** torch.Tensor

`mmselfsup.utils.`**`get_model`**(*model: torch.nn.modules.module.Module*) → mmengine.model.base_model.base_model.BaseModel

Get model if the input model is a model wrapper.

**Parameters** **model** (*nn.Module*) – A model may be a model wrapper.

**Returns** The model without model wrapper.

**Return type** *BaseModel*

`mmselfsup.utils.`**`nondist_forward_collect`**(*func: object*, *data_loader: torch.utils.data.dataloader.DataLoader*, *length: int*) → dict

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.
- **data_loader** (*DataLoader*) – the torch DataLoader to yield data.
- **length** (*int*) – Expected length of output arrays.

**Returns** The concatenated outputs.

**Return type** Dict[str, torch.Tensor]

mmselfsup.utils.**register_all_modules**(*init_default_scope: bool = True*) → None

Register all modules in mmselfsup into the registries.

**Parameters** **init_default_scope** (*bool*) – Whether initialize the mmselfsup default scope. When *init_default_scope=True*, the global default scope will be set to *mmselfsup*, and all registries will build modules from mmselfsup's registry node. To understand more about the registry, please refer to https://github.com/open-mmlab/mmengine/blob/main/docs/en/tutorials/registry.md Defaults to True.

CHAPTER

FORTYONE

# CONTRIBUTING TO MMSELFSUP



Thanks for your interest in contributing to MMSelfSup! All kinds of contributions are welcome, including but not limited to the following.

- Fix typo or bugs
- Add documentation or translate the documentation into other languages
- Add new features and components

## 41.1 Workflow

We recommend the potential contributors follow this workflow for contribution.

1. Fork and pull the latest MMSelfSup repository, follow *get_started* to setup the environment.
2. Checkout a new branch (**do not use master/dev branch** for PRs)

Please checkout a new branch from `dev-1.x` branch, you could follow the commands below:

```
git clone git@github.com:open-mmlab/mmselfsup.git
cd mmselfsup
git checkout dev-1.x
git checkout -b xxxx # xxxx is the name of new branch
```

3. Edit the related files follow the code style mentioned below
4. Use **pre-commit hook** to check and format your changes.
5. Commit your changes
6. Create a PR to merge it into dev-1.x branch

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

## 41.2 Code style

### 41.2.1 Python

We adopt PEP8 as the preferred code style.

We use the following tools for linting and formatting:

- flake8: A wrapper around some linter tools.
- isort: A Python utility to sort imports.
- yapf: A formatter for Python files.
- codespell: A Python utility to fix common misspellings in text files.
- mdformat: Mdformat is an opinionated Markdown formatter that can be used to enforce a consistent style in Markdown files.
- docformatter: A formatter to format docstring.

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

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

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

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

From the repository folder

```
pre-commit install
pre-commit run
```

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

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

### 41.2.2 C++ and CUDA

We follow the Google C++ Style Guide.

CHAPTER

FORTYTWO

# CHANGELOG

## 42.1 MMSelfSup

### 42.1.1 v1.0.0rc6 (10/02/2023)

The `master` branch is still 0.x version and we will checkout a new `1.x` branch to release 1.x version. The two versions will be maintained simultaneously in the future.

We briefly list the major breaking changes here. Please refer to the *migration guide* for details and migration instructions.

#### Highlight

- Support `MaskFeat` with video dataset in `projects/maskfeat_video/`
- Translate documentation to Chinese.

#### New Features

- Support `MaskFeat` with video dataset in `projects/maskfeat_video/` (#678)

#### Bug Fixes

- Fix distributed setting for shape bias (#689)
- Update link of beitv2 (#676)
- Pass param by explicitly setting location (#654)
- Update default_runtime.py (#681)
- Rename metafile.yaml to metafile.yml (#680)
- Fix bugs in configs/selfsup/eva/metafile.yaml (#669)

#### Improvements

- Switch default branch to 1.x (#686)
- Update pre-commit (#685)
- Deprecate the support of python 3.6 (#657)

#### Docs

- Translate add_transforms.md and conventions.md (#674)
- Translate classification.md, detection.md, segmentation.md (#665)
- Update link of knn script (#661)
- Translate two docs (#653)
- Translate three docs (#651)

### 42.1.2 v1.0.0rc5 (30/12/2022)

The `master` branch is still 0.x version and we will checkout a new `1.x` branch to release 1.x version. The two versions will be maintained simultaneously in the future.

We briefly list the major breaking changes here. Please refer to the *migration guide* for details and migration instructions.

#### Highlight

- Support `BEiT v2`, `MixMIM`, `EVA`
- Support `ShapeBias` for model analysis
- Add Solution of FGIA ACCV 2022 (1st Place)
- Refactor t-SNE

#### New Features

- Support `BEiT v2` (#627)
- Support `MixMIM` (#626)
- Support `EVA` (#632)
- Support `ShapeBias` metric (#635)
- Add convert scripts and instructions on seg and det (#621)
- Add pretraining for FGIA (#607)

#### Bug Fixes

- Change `pseudo_collect` to `default_collect` (#616)
- Fix the link of SimMIM 800pt 100ft (#622)
- Change `map_location` to `cpu` (#623)
- Fix import error (#631)
- Fix key error in configs (#630)
- Change `np.int` to `int` (#636)
- Fix knn multi-gpu bug (#634)

#### Improvements

- Refactor `projects/` folder (#620)
- Refactor `t-SNE` (#629)
- Refactor `CAE` (#645)
- Refactor benchmark script and update files (#637)

#### Docs

- Update data_flow.md (#612)
- Update datasets.md (#633)

### 42.1.3 v1.0.0rc4 (07/12/2022)

The `master` branch is still 0.x version and we will checkout a new `1.x` branch to release 1.x version. The two versions will be maintained simultaneously in the future.

We briefly list the major breaking changes here. Please refer to the *migration guide* for details and migration instructions.

#### Highlight

- Support `BEiT` and `MILAN`
- Support low-level reconstruction visualization

#### New Features

- Support `BEiT` (#425)
- Support `MILAN` (#600)
- Support low-level reconstruction visualization (#570)

#### Bug Fixes

- Fix registry of data preprocessor (#603)
- Fix dependence and key bug (#611)

#### Improvements

- Refactor file io (#582))
- Add './projects' folder and an example (#586))
- Update CI and UT (#601))

#### Docs

- Update readthedocs rst and menu button (#572)
- Add readthedocs algorithm pages and fix some displaying error (#599)

### 42.1.4 v1.0.0rc3 (01/11/2022)

The `master` branch is still 0.x version and we will checkout a new `1.x` branch to release 1.x version. The two versions will be maintained simultaneously in the future.

We briefly list the major breaking changes here. Please refer to the *migration guide* for details and migration instructions.

#### Highlight

- Support `MaskFeat`

#### New Features

- Support `MaskFeat` (#494)
- Add Hog generator (#518)

#### Bug Fixes

- Fix fine-tuning config of MAE-H-448 (#509)

#### Improvements

- Refactor evaluation folder and related configs (#538))
- Refine configs (#547))

#### Docs

- Add custom dataset tutorial (#522)
- Refactor add_modules.md (#524)
- Translate some documentation to Chinese

### 42.1.5 v1.0.0rc2 (12/10/2022)

The `master` branch is still 0.x version and we will checkout a new `1.x` branch to release 1.x version. The two versions will be maintained simultaneously in the future.

We briefly list the major breaking changes here. Please refer to the *migration guide* for details and migration instructions.

#### Highlight

- Full support of `MAE`, `SimMIM`, `MoCoV3`.

#### New Features

- Full support of `MAE` (#483)
- Full support of `SimMIM` (#487)
- Full support of `MoCoV3` (#496)

#### Bug Fixes

- Fix classification configs (#488)
- Fix MAE config name error (#498)

#### Improvements

- Refactor colab tutorial (#470))
- Update readthedocs requirements (#472)
- Update CI (#476)
- Refine `mim_slurm_test.sh` and `mim_dist_test.sh` for benchmarks (#477)
- Update Metafile format and content (#478)

### Docs

- Add advanced_guides/engine.md (#454)
- Add advanced_guides/evaluation.md (#456)
- add advanced_guides/transforms.md (#463)
- Add dataset docs (#437)
- Refine contribution guide (#492)
- update convention (#475)

## 42.1.6 v1.0.0rc1 (01/09/2022)

We are excited to announce the release of MMSelfSup v1.0.0rc1. MMSelfSup v1.0.0rc1 is the first version of MMSelfSup 1.x, a part of the OpenMMLab 2.0 projects. The `master` branch is still 0.x version and we will checkout a new `1.x` branch to release 1.x version. The two versions will be maintained simultaneously in the future.

We briefly list the major breaking changes here. Please refer to the *migration guide* for details and migration instructions.

### Highlight

- Based on MMEngine and MMCV.
- Released with refactor.
  - Datasets
  - Models
  - Config
  - ...
- Refine all documents.

### New Features

- Add `SelfSupDataSample` to unify the components' interface.
- Add `SelfSupVisualizer` for visualization.
- Add `SelfSupDataPreprocessor` for data preprocess in model.

### Improvements

- Most algorithms now support non-distributed training.
- Change the interface of different data augmentation transforms to `dict`.
- Run classification downstream task with MMClassification.

#### Docs

- Refine all documents and reorganize the directory.
- Add concepts for different components.

### 42.1.7 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)

### 42.1.8 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)

### 42.1.9 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)

### 42.1.10 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)

### 42.1.11 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)

### 42.1.12 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.

## 42.2 OpenSelfSup (History)

### 42.2.1 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

### 42.2.2 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 FORTYTHREE

# FAQ

We list some common troubles faced by many users and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them. If the contents here do not cover your issue, please create an issue using the provided templates and make sure you fill in all required information in the template.



## 43.1 Installation

Compatible MMEngine, MMCV, MMClassification, MMDetection and MMSegmentation versions are shown below. Please install the correct version of them to avoid installation issues.

**Note:**

- MMDetection and MMSegmentation are optional.
- If you still have version problem, please create an issue and provide your package versions.

## 43.2 DeepCluster on A100 GPU

Problem: If you want to try DeepCluster algorithm on A100 GPU, use the `faiss` installed by pip will raise error, which is mentioned in here.

Please install `faiss` by conda like this:

```
conda install -c pytorch faiss-gpu cudatoolkit=11.3
```

Also, you need to install PyTorch with the support of CUDA 11.3, and the faiss-gpu==1.7.2 requires python 3.6-3.8.

CHAPTER

# FORTYFOUR

# ENGLISH

# CHAPTER

# FORTYFIVE

CHAPTER

FORTYSIX

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## m

# INDEX

## G



## H



## I



## K



## L

## M

# N



# O



# P



# R

## S



## T

## U



## V



## W