MMSelfSup

发布 *1.0.0rc6*

MMSelfSup Authors

2023 年 04 月 06 日

新手入门

中文文档在持续翻译中，敬请期待，同时我们也鼓励社区开发者们参与到翻译中来

CHAPTER 1

总体介绍

在本文档中，我们将对MMSelfSup进行整体介绍。我们首先会对自监督学习 (Self-supervised Learning) 的基本概念进行回顾，然后简单介绍整个 MMSelfSup 的基本架构。最后，我们将给出 MMSelfSup 的上手路线图，帮助大家更快的使用 MMSelfSup 助力自己的科学研究和项目实践。

1.1 自监督学习

自监督学习 (Self-supervised learning, SSL) 是一种极具潜力的学习范式，它旨在使用海量的无标注数据来进行表征学习。在 SSL 中，我们通过构造合理的预训练任务（可自动生成标注，即自监督）来进行模型的训练，学习到一个具有强大建模能力的预训练模型。基于自监督学习获得的训练模型，我们可以提升各类下游视觉任务 (图像分类，物体检测，语义分割等) 的性能。

过去几年里，自监督学习的研究获得了快速发展，整个学术社区涌现了一大批优秀的自监督学习算法。我们旨在将 MMSelfSup 打造成为一个功能强大，方便易用的开源算法库，助力学术研究和工程实践。接下来，我们将介绍 MMSelfSup 的架构设计。

1.2 MMSelfSup 的架构设计

与其他 OpenMMLab 的项目类似，MMSelfSup 基于模块化设计的准则，整体的架构设计如下图所示：

- **Datasets** 支持各类数据集，同时提供丰富的数据增强策略。
- **Algorithms** 包括了多个经典的自监督算法，同时提供易用的用户接口。
- **Tools** 包括了自监督学习常用的训练和分析工具。
- **Benchmarks** 提供了使用自监督学习获得的预训练模型进行多种下游任务 ((图像分类，物体检测，语义分割等) 的示例。

1.3 MMSelfSup 的上手路线图

为了帮助用户更快的上手 MMSelfSup，我们推荐参考如下的路线图进行相关学习。

1.3.1 使用 MMSelfSup 玩转自监督实验

通常自监督学习算法被认为是一种适用不同模型架构的预训练算法，因此自监督学习应用通常会包括**预训练阶段**和**下游任务迁移学习阶段**。

- 如果你想尝试 MMSelfSup 提供的各类自监督学习算法，我们推荐你优先参考*Get Started* 来进行**环境配置**。
- 对于**预训练阶段**，我们推荐你参考*Pre-train* 来尝试各类预训练算法，获得预训练模型。
- 对于**下游任务迁移学习阶段**，我们推荐你参考Benchmark 当中提供的示例，来使用预训练模型来尝试各种下游任务。
- 除此之外，我们也提供了多种分析工具和可视化工具Useful Tools来帮助用户更方便地对算法进行诊断和分析。

1.3.2 基于 MMSelfSup 学习自监督算法

如果你是对 SSL 不太了解的新同学，我们推荐你将*Model Zoo*作为一个参考，学习我们已经支持的一些自监督学习的代表性工作。

CHAPTER 2

基础教程

2.1 预备条件

在本节中，我们将演示如何使用 PyTorch 准备环境。

MMSelfSup 在 Linux 上运行（Windows 和 macOS 不受官方支持）。它需要 Python 3.7+、CUDA 9.2+ 和 PyTorch 1.6+。

注解： 如果您有使用 PyTorch 的经验并且已经安装了它，请跳过这一部分并跳到下一个安装环节。否则，您可以按照如下步骤进行准备。

步骤 0. 从官方网站下载并安装 Miniconda。

步骤 1. 创建一个 conda 环境并激活它。

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

步骤 2. 按照官方说明安装 PyTorch，例如：

在 GPU 平台上：

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

在 CPU 平台上：

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

2.2 安装

我们建议用户遵循我们的最佳实践来安装 MMSelfSup。但是，整个过程是高度可定制的。有关详细信息，请参阅自定义安装部分。

2.2.1 最佳实践

步骤 0. 使用 MIM 安装 MMEngine 和 MMCV。

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

步骤 1. 安装 MMSelfSup。

根据您的需要，我们支持两种安装方式：

- *从源代码安装（推荐）*：您想开发自己的自监督任务或基于 MMSelfSup 框架的新功能，例如，添加新的数据集或模型。您可以使用我们提供的所有工具。
- *作为 Python 包安装*: 您只想在项目中调用 MMSelfSup 的 API 或导入 MMSelfSup 的模块。

从源代码安装

在这种情况下，从源代码安装 MMSelfSup：

```
git clone https://github.com/open-mmlab/mmselfsup.git
cd mmselfsup
git checkout 1.x
pip install -v -e .
# "-v" 表示详细，或更多输出
# "-e" 表示以可编辑模式安装项目，
# 因此，对代码所做的任何本地修改都将生效，无需重新安装。
```

或者，如果您想为 MMSelfSup 做出贡献或体验其正在实验中的功能，请查看 `dev-1.x` 分支：

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

作为 Python 包安装

直接用 pip 安装：

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

2.2.2 验证安装

要验证是否正确安装了 MMSelfSup，可以运行以下命令：

```
import mmselfsup
print(mmselfsup.__version__)
# 示例输出：1.0.0rc0 或更新版本
```

2.2.3 自定义安装

评测基准

最佳实践适用于基本用法。如果您需要使用一些下游任务（例如检测或分割）来评估您的预训练模型，请同时安装 MMDetection 和 MMSegmentation。

如果您不运行 MMDetection 和 MMSegmentation 基准测试，则无需安装它们。

您可以使用以下命令简单地安装 MMDetection 和 MMSegmentation：

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

更多详细信息，您可以查看 MMDetection 和 MMSegmentation 的安装页面。

CUDA 版本

安装 PyTorch 时，您需要指定 CUDA 的版本。如果您不清楚选择哪个，请遵循我们的建议：

- 对于基于 Ampere 的 NVIDIA GPU，例如 GeForce 30 系列和 NVIDIA A100，CUDA 11 是必须的。
- 对于较旧的 NVIDIA GPU，CUDA 11 向后兼容，但 CUDA 10.2 提供更好的兼容性并且更轻量级。

请确保 GPU 驱动程序满足最低版本要求。有关详细信息，请参阅此表。

注解： 如果您遵循我们的最佳实践，安装 CUDA 运行时库就足够了，因为不会在本地编译任何 CUDA 代码。但是，如果您希望从源代码编译 MMCV 或开发其他 CUDA 算子，则需要从 NVIDIA 的网站 安装完整的 CUDA 工具包，其版本应与 PyTorch 的 CUDA 版本相匹配，即 `conda install` 命令中指定的 cudatoolkit 版本。

在不使用 MIM 的情况下安装 MMEngine

想要使用 pip 而不是 MIM 安装 MMEngine，请遵循 MMEngine 安装指南。

例如，您可以通过以下命令安装 MMEngine：

```
pip install mmengine
```

在不使用 MIM 的情况下安装 MMCV

MMCV 包含 C++ 和 CUDA 扩展，因此以一种复杂的方式依赖于 PyTorch。MIM 会自动解决此类依赖关系并使安装更容易。但是，这不是必须的。

要使用 pip 而不是 MIM 安装 MMCV，请遵循 MMCV 安装指南。这需要根据 PyTorch 版本及其 CUDA 版本手动指定 find-url。

例如，以下命令安装以 PyTorch 1.12.0 和 CUDA 11.6 构建的 mmcv-full。

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

在仅有 CPU 的平台上安装

MMSelfSup 可以仅用于 CPU 环境。在 CPU 模式下，您可以训练、测试或推断模型。

在这种模式下，一些功能会消失，通常是 GPU 编译的操作。不过不用担心，MMSelfSup 中的几乎所有模型都不依赖这些操作。

在 Google Colab 上安装

Google Colab 通常会安装 PyTorch，因此我们只需要使用以下命令安装 MMCV 和 MMSeflSup。

步骤 0. 使用 MIM 安装 MMEngine 和 MMCV。

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

步骤 1. 从源代码安装 MMSelfSup。

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

步骤 2. 验证。

```
import mmselfsup
print(mmselfsup.__version__)
# 示例输出：1.0.0rc0 或更新版本
```

注解： 在 Jupyter 中，感叹号 ! 用于调用外部可执行文件，而 %cd 是一个魔法命令 来更改 Python 的当前工作目录。

通过 Docker 使用 MMSelfSup

我们提供了一个 Dockerfile 来构建镜像。请确保您的 docker 版本 >=19.03。

```
# 使用 PyTorch 1.10.0、CUDA 11.3、CUDNN 8 构建镜像。
docker build -f ./docker/Dockerfile --rm -t mmselfsup:torch1.10.0-cuda11.3-cudnn8 .
```

重要提示： 请确保您已安装 nvidia-container-toolkit。

运行以下命令：

```
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} 是包含所有这些数据集的本地文件夹。

2.2.4 故障排除

如果您在安装过程中遇到一些问题，请先查看常见问题页面。如果没有找到解决方案，您可以在 GitHub 上提交一个 issue。

2.3 使用多个 MMSelfSup 版本

如果您的机器上有多个 mmselfsup，并且您想交替使用它们，推荐的方法是创建多个 conda 环境，并为不同的版本使用不同的环境。

另一种方法是将以下代码插入主脚本（train.py、test.py 或您运行的任何其他脚本）：

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

或者在对应根文件夹的终端中运行以下命令来暂时使用当前的版本：

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

CHAPTER 3

预训练

3.1 教程 1: 了解配置文件

MMSelfSup 主要是在 python 文件中来设置各种各样的配置。我们配置文件系统的设计融合了模块化和可继承的设计理念，可以让用户轻松方便地完成各种实验配置。所有的配置文件全部位于 `configs` 目录下。如果您想查看配置文件的全貌，您可以使用以下命令 `python tools/misc/print_config.py`。

3.1.1 配置文件命名规则

我们使用以下规则来命名我们的配置文件，社区贡献者建议遵循这个规则来贡献您的代码。简单来说，配置文件的名字主要划分为四个部分: `algorithm info`, `module information`, `training information` 和 `data information`。不同部分通过下划线 _ 来进行相连，而属于同一个部分的内容，通过中横线 -来进行相连。

我们使用以下一个实例让大家有一个清晰的认识

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

- `algorithm_info`：与算法相关的一些信息，例如算法名;
- `module_info`：模块相关的一些信息，例如与 loss, head 相关的信息;
- `training_info`：训练相关的信息, 例如 batch size, 学习率调整器和数据增强策略。
- `data_info`：数据相关信息, 例如数据集名，输入图片的大小;

在下面几个章节，我们将对文件名中的各个部分进行详细的说明:

算法信息

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

`algorithm` 通常情况下是算法名字的缩写和版本号. 例如:

- `relative-loc` : 算法名中不同的部分通过中横线 -相连
- `simclr`
- `mocov2`

`misc` 描述了算法的一些其他信息

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

模块信息

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

模块信息大部分情况下是有关 backbone 的一些信息. 例如:

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

有时候，有些特殊的配置需要在配置文件名中提及，例如:

- `resnet50-sobel`: 在诸如线性评测之类的下游任务, 当我们使用的是 DeepCluster 的预训练模型，在经过 Sobel 层之后，模型只接受两层输入

而 `neck_setting`, `head_setting` 和 `loss_setting` 这几个选项是可选的。

训练信息

训练相关的一些配置，包括 batch size, 学习率调整方案和数据增强等。

- Batch size, 其格式为 `{gpu x batch_per_gpu}`，如 `8xb32`;
- 训练配置，他们需要以下面这个格式来进行书写 `{pipeline aug}-{train aug}-{scheduler}-{epochs}`

如:

- `8xb32-mcrop-2-6-coslr-200e` : `mcrop` 是 SwAV 提出的 pipeline 中的名为 multi-crop 的一部分。2 和 6 表示 2 个 pipeline 分别输出 2 个和 6 个裁剪图，而且裁剪信息记录在数据信息中;
- `8xb32-accum16-coslr-200e` : `accum16` 表示权重会在梯度累积 16 个迭代之后更新。
- `8xb512-amp-coslr-300e` : `amp` 表示使用混合精度训练。

数据信息

数据信息包含数据集，输入大小等。例如:

- `in1k` : `ImageNet1k` 数据集，默认使用的输入图像大小是 224x224
- `in1k-384px` : 表示输入图像大小是 384x384
- `cifar10`
- `inat18` : `iNaturalist2018` 数据集，包含 8142 类
- `places205`

配置文件命名示例

这一节，我们通过一个具体的例子来说明文件命名的规则:

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

- swav: 算法信息
- resnet50: 模块信息
- 8xb32-mcrop-2-6-coslr-200e: 训练信息
  - 8xb32: 共使用 8 张 GPU，每张 GPU 上的 batch size 是 32
  - mcrop-2-6: 使用 multi-crop 数据增强方法
  - coslr: 使用余弦学习率调度器
  - 200e: 训练模型 200 个周期
- in1k-224-96: 数据信息，在 ImageNet1k 数据集上训练，输入大小是 224x224 和 96x96

3.1.2 配置文件结构

在 configs/_base_ 文件夹中, 有 4 种类型的基础组件文件，即:

- models
- datasets
- schedules
- runtime

所有的基础配置文件定义了训练所需的最基础的元素，例如 train/val/test 循环，优化器。你可以通过继承一些基础配置文件快捷地构建你自己的配置。由 _base_ 下的组件组成的配置被称为原始配置（primitive）。为了易于理解，我们使用 MoCo v2 作为一个例子，并对它的每一行做出注释。若想了解更多细节，请参考 API 文档。

配置文件 configs/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-200e_in1k.py 如下所述:

```
_base_ = [
    '../_base_/models/mocov2.py',                   # 模型
    '../_base_/datasets/imagenet_mocov2.py',        # 数据
    '../_base_/schedules/sgd_coslr-200e_in1k.py',   # 训练调度
    '../_base_/default_runtime.py',                 # 运行时设置
]

# 我们继承了默认的运行时设置，同时修改了 ``CheckpointHook``.
# max_keep_ckpts 控制在 work_dirs 中最多保存多少个 checkpoint 文件
```

(下页继续)

(续上页)

```
# 例如是 3, ``CheckpointHook`` 将会只保存最近的 3 个 checkpoint 文件
# 如果在 work_dirs 中超过了 3 个文件, 将会自动删掉时间最久远的那个 checkpoint
# , 从而保持 checkpoint 文件的数目始终为 3
default_hooks = dict(checkpoint=dict(max_keep_ckpts=3))
```

../_base_/models/mocov2.py 是 MoCo v2 的基础模型配置。

```
# type='MoCo' 指代我们使用 MoCo 这个算法。 我们将改算法分为四个部分:
# backbone, neck, head 和 loss。'queue_len', 'feat_dim' and 'momentum' 是另外
# 几个 MoCo 需要的参数。
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 是 MoCo v2 的基础数据集配置。主要写出了与 dataset 和 dataloader 相关的信息。

```
# dataset 配置
# 我们使用 MMClassification 中实现的 ``ImageNet`` dataset 数据集, 所以
# 这里有一个 ``mmcls`` 前缀.
dataset_type = 'mmcls.ImageNet'
data_root = 'data/imagenet/'
```

(下页继续)

(续上页)

```
# mocov2 和 mocov1 的主要差异在于数据增强的不同
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 是 MoCo v2 的基础调度配置。

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

# 学习率调整策略
# 使用 cosine learning rate decay
param_scheduler = [
    dict(type='CosineAnnealingLR', T_max=200, by_epoch=True, begin=0, end=200)
]

# 循环设置
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=200)
```

../_base_/default_runtime.py 是运行时的默认配置。运行时设置主要包含一些训练中需要使用的基础配置, 例如 default_hooks 和 log_processor

```
default_scope = 'mmselfsup'

default_hooks = dict(
    runtime_info=dict(type='RuntimeInfoHook'),
    optimizer=dict(type='OptimizerHook', grad_clip=None),
    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(
    interval=50,
    custom_keys=[dict(data_src='', method='mean', windows_size='global')])

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

(下页继续)

(续上页)

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

3.1.3 继承和修改配置文件

为了易于理解，我们推荐贡献者从现有方法继承。

对于同一个文件夹下的所有配置，我们推荐只使用一个原始（primitive）配置。其他所有配置应当从原始（primitive）配置继承。这样最大的继承层次为 3。

例如，如果你的配置文件是基于 MoCo v2 做一些修改，首先你可以通过指定 `_base_ ='./mocov2_resnet50_8xb32-coslr-200e_in1k.py.py'`（相对于你的配置文件的路径）继承基本的 MoCo v2 结构，接着在配置文件中修改一些必要的参数。现在，我们举一个更具体的例子，我们想使用 `configs/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-200e_in1k.py.py` 中几乎所有的配置，但是将训练周期数从 200 修改为 800，修改学习率衰减的时机和数据集路径，你可以创建一个名为 `configs/selfsup/mocov2/mocov2_resnet50_8xb32-coslr-800e_in1k.py.py` 的新配置文件，内容如下：

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


# 学习率调整器
param_scheduler = [
    dict(type='CosineAnnealingLR', T_max=800, by_epoch=True, begin=0, end=800)
]

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

使用配置中的中间变量

在配置文件中使用一些中间变量会使配置文件更加清晰和易于修改。

例如 `dataset_type`, `train_pipeline`, 是数据中的中间变量。我们先定义它们再将它们传进 `data`.

```
# 数据集配置
# 我们使用来源于 MMClassification 中的 ``ImageNet``, 所以有一个 ``mmcls`` 的前缀
dataset_type = 'mmcls.ImageNet'
data_root = 'data/imagenet/'

# mocov2 和 mocov1 的不同主要来自于数据增强
```

(下页继续)

(续上页)

```
view_pipeline = [
    dict(type='RandomResizedCrop', size=224, scale=(0.2, 1.)),
    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),
    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))
```

忽略基础配置中的字段

有时候，你需要设置 _delete_=True 来忽略基础配置文件中一些域的内容。您可以参考 mmengine 获得更多说明。接下来是一个例子。如果你希望在 SimCLR 使用中 MoCoV2Neck, 仅仅继承并直接修改将会报 get unexcepected keyword 'num_layers' 错误, 因为在 model.neck 域信息中, 基础配置 num_layers 字段被保存下来了，你需要添加 _delete_=True 来忽略 model.neck 在基础配置文件中的有关字段的内容:

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

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

使用基础配置中的字段

有时候，你可能引用 _base_ 配置中一些字段，以避免重复定义。你可以参考 mmengine 获取更多的说明。下面是一个使用基础配置文件中 num_classes 的例子，请参考 configs/selfsup/odc/odc_resnet50_8xb64-steplr-440e_in1k.py.

```
_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 通过脚本参数修改配置

当用户使用脚本 “tools/train.py” 或 “tools/test.py” 提交任务，或者其他工具时，可以通过指定 `--cfg-options` 参数来直接修改配置文件中内容。

- 更新字典链中的配置的键

  配置项可以通过遵循原始配置中键的层次顺序指定。例如，`--cfg-options model.backbone.norm_eval=False` 改变模型 backbones 中的所有 BN 模块为 `train` 模式。

- 更新列表中配置的键

  你的配置中的一些配置字典是由列表组成。例如，训练 pipeline `data.train.pipeline` 通常是一个列表。例如 `[dict(type='LoadImageFromFile'), dict(type='TopDownRandomFlip', flip_prob=0.5), ...]`。如果你想要在 pipeline 中将 `'flip_prob=0.5'` 修改为 `'flip_prob=0.0'`，您可以指定 `--cfg-options data.train.pipeline.1.flip_prob=0.0`.

- 更新 list/tuples 中的值

  如果想要更新的值是一个列表或者元组。例如，一些配置文件中包含 `param_scheduler = "[dict(type='CosineAnnealingLR',T_max=200,by_epoch=True,begin=0, end=200)]"`。如果你想要改变这个键，你可以指定 `--cfg-options param_scheduler = "[dict(type='LinearLR',start_factor=1e-4, by_epoch=True,begin=0,end=40, convert_to_iter_based=True)]"`。注意，” 是必要的，并且在指定值的时候，在引号中不能存在空白字符。

3.1.5 导入用户定义模块

注解： 这部分内容初学者可以跳过，只在使用其他 MM-codebase 时会用到，例如使用 mmcls 作为第三方库来构建你的工程。

这部分内容初学者可以跳过，只在使用其他 MM-codebase 时会用到，例如使用 mmcls 作为第三方库来构建你的工程。i 为了简化代码，你可以使用 MM-codebase 作为第三方库，只需要保存你自己额外的代码，并在配置文件中导入自定义模块。你可以参考 OpenMMLab Algorithm Competition Project 中的例子.

在你自己的配置文件中添加如下所述的代码：

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

3.2 教程 2: 准备数据集

MMSelfSup 支持多个数据集。请遵循相应的数据准备指南。建议将您的数据集根目录软链接到 `$MMSELFSUP/data`。如果您的文件夹结构不同，您可能需要更改配置文件中的相应路径。


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

3.2.1 准备 ImageNet 数据集

对于 ImageNet，它有多个版本，但最常用的是 ILSVRC 2012。可以通过以下步骤得到：

1. 注册账号并登录 下载页面
2. 找到 ILSVRC2012 的下载链接，下载以下两个文件
   - ILSVRC2012_img_train.tar (~138GB)
   - ILSVRC2012_img_val.tar (~6.3GB)
3. 解压下载的文件
4. 使用这个 脚本 下载元数据

3.2.2 准备 Places205 数据集

对于 Places205，您需要：

1. 注册账号并登录 下载页面
2. 下载 Places205 经过缩放的图片以及训练集和验证集的图片列表
3. 解压下载的文件

3.2.3 准备 iNaturalist2018 数据集

对于 iNaturalist2018，您需要：

1. 从 下载页面 下载训练集和验证集图像及标注
2. 解压下载的文件
3. 使用脚本 `tools/data_converters/convert_inaturalist.py` 将原来的 json 标注格式转换为列表格式

3.2.4 准备 PASCAL VOC 数据集

假设您通常将数据集存储在 `$YOUR_DATA_ROOT` 中。下面的命令会自动将 PASCAL VOC 2007 下载到 `$YOUR_DATA_ROOT` 中，准备好所需的文件，在 `$MMSELFSUP` 下创建一个文件夹 `data`，并制作一个软链接 `VOCdevkit`。

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

3.2.5 准备 CIFAR10 数据集

MMSelfSup 使用由 MMClassification 实现的CIFAR10。此外，MMClassification 支持自动下载 CIFAR10 数据集，您只需在 data_root 字段中指定下载文件夹即可。并且通过指定 test_mode=False / test_mode=True 来使用训练数据集或测试数据集。对于更多细节，请参考 MMClassification 中的文档。

3.2.6 准备检测和分割数据集

检测

您可以参考 mmdetection 来准备 COCO，VOC2007 和 VOC2012 检测数据集。

分割

您可以参考 mmsegmentation 来准备 VOC2012AUG 和 Cityscapes 分割数据集。

3.3 教程 3: 使用现有模型进行预训练

本文档提供有关如何运行算法以及如何使用 MMSelfSup 中的一些工具的基本用法。有关安装说明和数据准备，请参阅 install.md 和 prepare_data.md。

开始训练

注意：配置文件中的默认学习率是针对特定数量的 GPU（GPU 数量已在配置文件名称中注明）。如果使用不同数量的 GPUs，总的 batch size 将按比例变化，您必须按照 new_lr = old_lr * new_ngpus / old_ngpus 缩放学习率。

3.3.1 使用单卡训练

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

一个简单的例子来开启训练：

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

3.3.2 使用 CPU 训练

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

注意。我们不建议用户使用 CPU 进行训练，因为它太慢了。我们支持这个功能，是为了方便用户在没有 GPU 的机器上进行调试。

3.3.3 使用多卡训练

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

可选参数：

- `--work-dir`：指示您的自定义工作目录以保存 checkpoints 和日志。
- `--resume`：自动在你的工作目录中查找最新的 checkpoints。或者设置 `--resume ${CHECKPOINT_PATH}` 来加载特定的 checkpoints 文件。
- `--amp`：启用自动混合精度训练。
- `--cfg-options`：设置 `--cfg-options` 将修改原始配置。例如，设置 `--cfg-options randomness.seed=0` 将为随机数设置种子。

使用 8 个 GPUs 开始训练的示例：

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

或者，如果您在使用 **slurm** 管理的集群上运行 MMSelfSup：

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

使用 8 个 GPUs 开始训练的示例：

```
# 默认设置：GPUS_PER_NODE=8 GPUS=8
sh tools/slurm_train.sh Dummy Test_job configs/selfsup/mae/mae_vit-base-p16_8xb512-
↪coslr-400e_in1k.py
```

3.3.4 使用用多台机器训练

如果您想使用由以太网连接起来的多台机器，您可以使用以下命令:

在第一台机器上:

```
NNODES=2 NODE_RANK=0 PORT=${MASTER_PORT} MASTER_ADDR=${MASTER_ADDR} sh tools/dist_
↪train.sh ${CONFIG} ${GPUS}
```

在第二台机器上:

```
NNODES=2 NODE_RANK=1 PORT=${MASTER_PORT} MASTER_ADDR=${MASTER_ADDR} sh tools/dist_
↪train.sh ${CONFIG} ${GPUS}
```

但是，如果您不使用高速网路连接这几台机器的话，训练将会非常慢。

如果您使用的是 **slurm** 来管理多台机器，您可以使用同在单台机器上一样的命令来启动任务，但是您必须得设置合适的环境变量和参数，具体可以参考 slurm_train.sh。

3.3.5 在一台机器上启动多个任务

如果你在一台机器上启动多个任务，例如，在一台有 8 个 GPU 的机器上启动 2 个分别使用 4 块 GPU 的训练任务，你需要为每个任务指定不同的端口（默认为 29500）以避免通信冲突。

如果你使用 `dist_train.sh` 来启动训练任务。

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

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

如果你用 slurm 启动训练任务，你有两种方法来设置不同的通信端口。

方法 1 :

在 `config1.py` 中:

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

在 `config2.py` 中:

```
env_cfg = dict(dist_cfg=dict(backend='nccl', port=29501))
```

然后你可以用 config1.py 和 config2.py 启动两个任务。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 sh tools/slurm_train.sh ${PARTITION} ${JOB_NAME}␣
↪config1.py [optional arguments]

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

方法 2 :

你可以设置不同的通信端口，而不需要修改配置文件，但必须设置 `--cfg-options` 来覆盖配置文件中的默认端口。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 sh 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 sh 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 教程 4: 使用自定义数据集进行预训练

在本教程中，我们将介绍如何使用自定义数据集 (无需标注) 进行自监督预训练。

3.4.1 在自定义数据集上使用 MAE 算法进行预训练

在 MMSelfSup 中, 我们支持用户直接调用 MMClassification 的 `CustomDataset`(类似于 `torchvision` 的 `ImageFolder`), 该数据集能自动的读取给的路径下的图片。你只需要准备你的数据集路径，并修改配置文件，即可轻松使用 MMSelfSup 进行预训练。

第一步：获取自定义数据路径

路径应类似这种形式：`data/custom_dataset/`

第二步：选择一个配置文件作为模板

在本教程中，我们使用 `configs/selfsup/mae/mae_vit-base-p16_8xb512-coslr-400e_in1k.py` 作为一个示例进行讲解。我们首先复制这个配置文件，将新复制的文件命名为 `mae_vit-base-p16_8xb512-coslr-400e_${custom_dataset}.py`.

- `custom_dataset`: 表明你用的那个数据集。例如，用 `in1k` 代表 ImageNet 数据集，`coco` 代表 COCO 数据集。

这个配置文件的内容如下：

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

# 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.)
```

(下页继续)

(续上页)

```
        }))

# 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
```

第三步：修改数据集相关的配置

数据集相关的配置是定义在 _base_ 的'../_base_/datasets/imagenet_mae.py' 文件内。我们直接将其内容复制到刚刚创建的新的配置文件mae_vit-base-p16_8xb512-coslr-400e_${custom_dataset}.py 中.

- 此时我们删除 _base_ 的 '../_base_/datasets/imagenet_mae.py'。
- 修 改 dataset_type = 'mmcls.CustomDataset' 和 data_root = /dataset/my_custom_dataset.
- 删除 train_dataloader 中的 ann_file ，同时根据自己的实际情况决定是否需要设定

data_prefix。

注解： CustomDataset 是在 MMClassification 实现的，因此我们使用这种方式 dataset_type=mmcls.CustomDataset 来使用这个类。

此时，修改后的文件应如下：

```
# >>>>>>>>>>>>>>>>>>>>> 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='./'),
```

(下页继续)

(续上页)

```
        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,
    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
```

(下页继续)

(续上页)

```
    checkpoint=dict(type='CheckpointHook', interval=1, max_keep_ckpts=3))

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

使用上述配置文件，你就能够轻松的在自定义数据集上使用 MAE 算法来进行预训练了。

3.4.2 在 COCO 数据集上使用 MAE 算法进行预训练

注解： 你可能需要参考文档安装 MMDetection 来使用 `mmdet.CocoDataset`。

与在自定义数据集上进行预训练类似，我们在本教程中也提供了一个使用 COCO 数据集进行预训练的示例。修改后的文件如下：

```
# >>>>>>>>>>>>>>>>>>>>>> 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,
```

(下页继续)

(续上页)

```
    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,
    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,
```

(下页继续)

(续上页)

```
        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 在自定义数据集上使用 SimCLR 算法进行预训练

我们也提供了一个使用 SimCLR 在自定义数据集上进行预训练的配置文件，主要思路与 在自定义数据集上使用 MAE 算法进行预训练是类似的。

我们使用的模板是 `configs/selfsup/simclr/simclr_resnet50_8xb32-coslr-200e_in1k.py`，你可以根据自己的需要从配置文件仓库里选择合适的文件作为模板，其修改后的内容如下:

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

(下页继续)

(续上页)

```
                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
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),
```

(下页继续)

(续上页)

```
            '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 使用 MMSelfSup 提供的预训练模型来加速收敛

在具体应用中，我们可以使用 MMSelfSup 已经提供的预训练模型来加速自定义数据集上的训练速度。你可以考虑使用这些预训练模型作为初始化。具体来讲，你只需要从 模型库 中选择一个合适模型，获取模型权重的 URL 链接，并在启动训练的时候，指定这个链接作为预训练模型。

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

- `CONFIG`: 修改后的配置文件
- `GPUS`: 使用的 GPU 数
- `PRETRAIN`: MMSelfSup 提供的预训练模型文件的 URL

CHAPTER 4

下游任务

4.1 分类

在 MMSelfSup 中，我们为分类任务提供了许多基线，因此模型可以在不同分类任务上进行评估。这里有详细的教程和例子来阐述如何使用 MMSelfSup 来运行所有的分类基线。我们在 `tools/benchmarks/classification/`文件夹中提供了所有的脚本，包含 2 个`.sh` 文件，一个文件夹用于与 VOC SVM 相关的分类任务，另一个文件夹用于 ImageNet 最近邻分类任务。

4.1.1 VOC SVM / Low-shot SVM

为了运行这些基准，您首先应该准备好您的 VOC 数据集。请参考*prepare_data.md* 来获取数据准备的详细信息。

为了评估这些预训练的模型, 您可以运行如下指令。

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

(下页继续)

(续上页)

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

此外，如果您想评估由 runner 保存的 ckpt 文件，您可以运行如下指令。

```
# 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}
```

使用 ckpt 进行测试，代码使用 epoch_*.pth 文件，这里不需要提取权重。

备注：

- `${SELFSUP_CONFIG}` 是自监督实验的配置文件.
- `${FEATURE_LIST}` 是一个字符串，用于指定从 layer1 到 layer5 的要评估特征；例如，如果您只想评估 layer5，那么 `FEATURE_LIST` 是 “feat5”，如果您想要评估所有的特征，那么 `FEATURE_LIST` 是 “feat1 feat2 feat3 feat4 feat5” (用空格分隔)。如果为空，那么 `FEATURE_LIST` 默认是 “feat5”。
- `${PRETRAIN}`：预训练模型文件。
- 如果您想改变 GPU 个数, 您可以在命令的前面加上 `GPUS_PER_NODE=4 GPUS=4`。
- `${EPOCH}` 是您想要测试的 ckpt 的轮数

4.1.2 线性评估和微调

线性评估和微调是最常见的两个基准。我们为线性评估和微调提供了配置文件和脚本来进行训练和测试。支持的数据集有 **ImageNet**，**Places205** 和 **iNaturalist18**。

首先，确保您已经安装 MIM，这也是 OpenMMLab 的一个项目.

```
pip install openmim
```

此外，请参考 MMClassification 的安装和数据准备。

然后运行如下命令。

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

(下页继续)

(续上页)

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

备注:

- ${CONFIG}: 使用 configs/benchmarks/classification/下的配置文件。具体来说，imagenet (除了 imagenet_*percent 文件), places205 and inaturalist2018。
- ${PRETRAIN}: 预训练模型文件。

例子:

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

如果您想测试训练好的模型，请运行如下命令。

```
# 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}
```

备注:

- ${CHECKPOINT}: 您想测试的训练好的分类模型

例子:

```
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 半监督分类

为了运行 ImageNet 半监督分类，我们将使用和线性评估和微调一样的 .sh 脚本进行训练。

备注:

- 默认 GPU 数量是 4.
- ${CONFIG}: 使用 configs/benchmarks/classification/imagenet/下的配置文件，命名为 imagenet_*percent 的文件。
- ${PRETRAIN}: 预训练模型文件。

4.1.4 ImageNet 最近邻分类

仅支持 CNN 形式的主干网络 (例如 ResNet50)。

为评估用于 ImageNet 最近邻分类基准的预训练模型，您可以运行如下命令。

```
# 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]
```

备注：

- ${SELFSUP_CONFIG}：是自监督实验的配置文件。
- ${CHECKPOINT}：检查点模型文件的路径。
- 如果您想改变 GPU 的数量，您可以在命令的前面加上 GPUS_PER_NODE=4 GPUS=4。
- [optional arguments]：用于可选参数，您可以参考这个脚本

命令的一个例子

```
# 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 检测

- 检测
  - 训练
  - 测试

这里，我们倾向使用 MMDetection 做检测任务。首先确保您已经安装了 MIM，这也是 OpenMMLab 的一个项目。

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

非常容易安装这个包。

此外，请参考 MMDetection 的安装和数据准备

4.2.1 训练

安装完后，您可以使用如下的简单命令运行 MMDetection。

```
# 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}
```

备注：

- ${CONFIG}: 使用 configs/benchmarks/mmdetection/下的配置文件。由于 OpenMMLab 的算法库支持跨不同存储库引用配置文件，因此我们可以轻松使用 MMDetection 的配置文件，例如：

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

从头开始写您的配置文件也是支持的。

- ${PRETRAIN}：预训练模型文件
- ${GPUS}: 您想用于训练的 GPU 数量，对于检测任务，我们默认采用 8 块 GPU。

例子：

```
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
```

或者您想用 detectron2 来做检测任务，我们也提供了一些配置文件。请参考 INSTALL.md 用于安装并按照 detectron2 需要的目录结构准备您的数据集。

```
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 测试

在训练之后，您可以运行如下命令测试您的模型。

```
# 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}
```

备注：

- ${CHECKPOINT}：您想测试的训练好的检测模型。

例子：

```
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 分割

对于语义分割任务我们使用 MMSegmentation。首先确保您已经安装了 MIM，这也是 OpenMMLab 的一个项目。

```
pip install openmim
mim install 'mmsegmentation>=1.0.0rc0'
```

非常容易安装这个包。

此外，请参考 MMSegmentation 的安装和数据准备。

4.3.1 训练

在安装完后，可以使用如下简单命令运行 MMSegmentation。

```
# 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}
```

备注:

- ${CONFIG}: 使用 configs/benchmarks/mmsegmentation/下的配置文件。由于 OpenMMLab 的算法库支持跨不同存储库引用配置文件，因此我们可以轻松使用 MMSegmentation 的配置文件，例如:

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

从头开始写您的配置文件也是支持的。

- ${PARTITION}: 预训练模型文件
- ${GPUS}: 您想用于训练的 GPU 数量，对于分割任务，我们默认采用 4 块 GPU。

例子:

```
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 测试

在训练之后，您可以运行如下命令测试您的模型。

```
# 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}
```

备注:

- ${CHECKPOINT}: 您想测试的训练好的分割模型。

例子:

```
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 5

便捷工具

5.1 可视化

可视化能直观反映模型性能表现。

5.1.1 如何实现可视化

建议先学习 文档 里关于可视化的基本概念。

OpenMMLab 2.0 引入可视化对象 `Visualizer` 和一些可视化后端 `VisBackend` 。如下图表展示了 `Visualizer` 和 `VisBackend` 的关系。

5.1.2 MMSelfSup 的可视化做什么

(1) 用不同的存储后端存训练数据

MMEngine 的后端包括 `LocalVisBackend`, `TensorboardVisBackend` 和 `WandbVisBackend`。

在训练过程中，默认钩子 `LoggerHook` 中的 after_train_iter() 会被调用，并且会在不同后端中用到 `add_scalars`，例如：

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

(2) 浏览数据集

`add_datasample()` 函数位于*`SelfSupVisualizer`*, 常用于在 browse_dataset.py 中浏览数据集。更多细节可以参考数据集可视化。

5.1.3 用不同的存储后端

如果想用不同的存储后端（ Wandb, Tensorboard, 或者远程窗口里常规的后端），像以下这样改配置文件的 `vis_backends` 就行了：

Local

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

Tensorboard

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

例如

Wandb

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

例如：

5.1.4 定制化的可视化

定制化可视化就像定制化其他组成部分那样。想定制化 `Visualizer`, `VisBackend` 或者 `VisualizationHook` 的话可以参考 MMEngine 里的 可视化文档

5.1.5 数据集可视化

`tools/misc/browse_dataset.py` 帮助用户可视化浏览 MMSelfSup 数据集，或者也可以把图像存到指定的目录里。

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

例子如下：

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

一个可视化的例子如下：

- 左边两张图来自对比学习数据流。
- 右边那张图是添加了掩码的图像。

5.1.6 t-SNE 可视化

我们提供可视化 t-SNE 展示图片表征的现成工具。

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

参数:

- `CONFIG_FILE`: 位于 `configs/tsne/` 中的 t-SNE 的配置文件。
- `CKPT_PATH`: 模型检查点的目录或链接。
- `WORK_DIR`: 拿来存可视化结果的目录。
- `[optional arguments]`: 可选项，可以参考 visualize_tsne.py

一个命令示例如下：

```
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
```

下面是可视化的例子, 左边来自 MoCoV2_ResNet50，右边来自 MAE_ViT-base:

5.1.7 可视化低级特征重建

我们提供如下算法的重建可视化：

- MAE
- SimMIM
- MaskFeat

用户可以通过如下命令可视化重建。

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

参数:

- CONFIG_FILE: 预训练模型配置文件。
- CKPT_PATH: 模型检查点的路径。
- IMAGE_PATH: 输入图像的路径。
- OUTPUT_PATH: 输出图像的路径，包含 4 个子图。
- [optional arguments]: for optional arguments, 您可以参考 visualize_reconstruction.py 了解可选参数。

例子如下:

```
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 \
```

(下页继续)

(续上页)

```
    --norm-pix


# SimMIM 在数据流里生成掩码，所以我们不用脚本里定义好的管道而用 '--use-vis-pipeline' 来应用配置里
定义的 'vis_pipeline'
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
```

MAE 结果如下:

SimMIM 结果如下:

MaskFeat 结果如下:

5.1.8 可视化 shape bias

shape bias 衡量在感知图像特征的过程中，与纹理相比，模型依赖 shape 的程度。感兴趣的话可以参考 paper 了解更多信息。MMSelfSup 提供一个现有的用于得到分类模型 shape bias 的工具箱。可以按以下步骤来做:

准备数据集

首先把 cue-conflict 下载到 `data` 文件夹里, 然后解压数据集。然后，您的 `data` 文件夹的结构应该像这样:

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

为分类调整配置

用以下配置代替原来的 test_dataloader 和 test_evaluation

```
test_dataloader = dict(
    dataset=dict(
        type='CustomDataset',
        data_root='data/cue-conflict',
        _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')
```

请记得自己修改一下 `csv_dir` 和 `model_name`。

用上述调整过的配置文件推理模型

然后您需要做的是用调整过的配置文件在 `cue-conflict` 数据集上推理模型。

```
# 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
```

在这之后，可以获得名为 `cue-conflict_model-name_session-1.csv` 的 csv 文件。除了这个文件之外，您应该下载 csv 文件 到对应的 `csv_dir`。

画出 shape bias

然后我们就可以开始画出 shape bias 了。

```
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`, 相同目录下，用于存储 csv 文件。
- `--colors`, 应为以 RGB 为格式的 RGB 值, 比如 100 100 100, 如果您想画若干模型的 shape bias 的话多个 RGB 值也行。

- `--plotting-names`, 偏好形状里图例的名称，您可将之设为模型名字。如果您想画若干模型的 shape bias 的话名字设多个值也行。
- `--model-names`，应该跟配置文件里的一样，如果您想画若干模型的 shape bias 的话多个名字也行。

请注意，每三个 `--colors` 对应一个 `--model-names` 。上面步骤做完后您会得到如下图像：

5.2 分析工具

5.2.1 统计参数量

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

一个例子如下：

```
python tools/analysis_tools/count_parameters.py configs/selfsup/mocov2/mocov2_
↪resnet50_8xb32-coslr-200e_in1k.py
```

5.2.2 发布模型

发布模型之前，你可能是想：

- 把模型权重转换为 CPU 张量。
- 删除优化器相关状态。
- 计算检查点文件的哈希值并把哈希 ID 加到文件名上。

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

例子如下：

```
python tools/model_converters/publish_model.py YOUR/PATH/epoch_100.pth YOUR/PATH/
↪epoch_100_output.pth
```

5.2.3 结果复现

想让你的结果完全可以复现的话，训练最终模型时请设置 `--cfg-options randomness.deterministic=True` 。值得一提的是，这会关掉 `torch.backends.cudnn.benchmark` 并降低训练速度。

5.2.4 日志分析

`tools/analysis_tools/analyze_logs.py` 用训练日志文件画损失/学习率曲线。首先 `pip install seaborn` 安装依赖库。

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

例子如下:

- 画部分运行过程中分类的损失函数图像。

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

- 画部分运行过程中分类和倒退的损失函数图像并存到 pdf 文件里。

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

- 在同一张图内，比较两次训练的损失。

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

- 计算平均训练速度。

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

  输出应该像下面这样:

```
-----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 6

基础概念

6.1 数据流

数据流（Data Flow）定义了数据在两个独立模块之间传递的方式，如数据加载器（dataloader）模块与模型（model）模块，如下图所示。

在 MMSelfSup 中，我们主要关注两类数据流，一是数据加载器（dataloader）与模型（model）之间，二是模型与可视化工具（visualizer）之间。而对于模型与 metric 之间数据流的介绍，大家可以参考 OpenMMLab 其他代码库中的文档，如 MMClassification. 此外，对于 model 与 visualizer 模块之间的数据流，感兴趣的话可以参考：*visualization*.

6.1.1 数据加载器与模型之间的数据流

数据加载器（dataloader）和模型（model）之间的数据流一般可以分为如下三个步骤:

i) 使用 `PackSelfSupInputs` 将转换完成的数据打包成为一个字典;

ii) 使用 `collate_fn` 将各个张量集成为一个批处理张量;

iii) 数据预处理器把以上所有数据迁移到 GPUS 等目标设备，并在数据加载器中将之前打包的字典解压为一个元组，该元祖包含输入图像与对应的元信息（`SelfSupDataSample`）。

数据集的数据处理

在 MMSelfSup 中，数据在投入到模型中前，会先进行一系列转换，称为 `pipeline`，如常用的 `RandomResizedCrop` 和 `ColorJitter` 转换。在 `pipeline` 中完成若干次转换后，最后一步转换是 `PackSelfSupInputs`，`PackSelfSupInputs` 会将转换好的数据打包到一个字典中，此字典包含两部分，即 `inputs` 和 `data_samples`.

```
# 在这部分，我们省略了一些不太重要的代码

class PackSelfSupInputs(BaseTransform):

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

        packed_results = dict()
        if self.key in results:
            ...
            packed_results['inputs'] = img

        ...
        packed_results['data_samples'] = data_sample

        return packed_results
```

提示：`inputs` 包含了一个图像列表，例如一个应用在对比学习中的多视图列表。即使输入是单个视图，`PackSelfSupInputs` 仍然会把信息输出到一个列表中。

数据加载器的数据处理

以数据集中的获取字典列表作为输入，数据加载器（dataloader）中的 `collect_fn` 会提取每个字典的 `inputs` 并将其整合成一个批处理张量；此外，每个字典中的 `data_sample` 也会被整合为一个列表，从而输出一个与先前字典有相同键的字典；最终数据加载器会通过 `collect_fn` 输出这个字典。

数据预处理器的数据处理

数据预处理是数据输入模型之前，处理数据过程的最后一步。数据预处理过程会对图像进行归一处理，如把 BGR 模式转换为 RGB 模式，并将所有数据迁移至 GPU 等目标设备中。上述各步骤完成后，最终会得到一个元组，该元组包含一个批处理图像的列表，和一个数据样本的列表。

```
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
            ]

        # 转换为 float 格式
        # 以保障效率
        batch_inputs = [input_.float() for input_ in batch_inputs]

        # 该步骤为归一化。 这与 :class:`mmengine.ImgDataPreprocessor` 有所不同。
        # 由于某些算法（如 SimCLR ）的图像有多个视图，所以输入中的每项都是一个列表，
        # 其中包含一张图像的多个视图。
        if self._enable_normalize:
            batch_inputs = [(_input - self.mean) / self.std
                            for _input in batch_inputs]

        return batch_inputs, batch_data_samples
```

6.2 数据结构

像 OpenMMLab 中其他仓库一样，MMSelfSup 也定义了一个数据结构，名为 `SelfSupDataSample`，这个数据结构用于接收和传递整个训练和测试过程中的数据。`SelfSupDataSample` 继承 MMEngine 中使用的 `BaseDataElement`。如果需要深入了解 `BaseDataElement`，我们建议参考 BaseDataElement。在这些教程中，我们主要讨论*SelfSupDataSample* 中一些定制化的属性。

6.2.1 SelfSupDataSample 中的定制化的属性

在 MMSelfSup 中，`SelfSupDataSample` 将模型需要的所有信息（除了图片）打包，比如 mask image modeling(MIM) 中请求的 `mask` 和前置任务中的 `pseudo_label` 。除了提供信息，它还能接受模型产生的信息，比如预测得分。为实现上述功能，`SelfSupDataSample` 定义以下五个属性：

- gt_label（标签数据），包含图片的真实标签。
- sample_idx（实例数据），包含一开始被数据集初始化的数据列表中的最近的图片的序号。
- mask（数据基类），包含 MIM 中的面具，比如 SimMIM 和 CAE。
- pred_label（标签数据），包含模型预测的标签。
- pseudo_label（数据基类），包含前置任务中用到的假的标签，比如 Relation Location 中的 location。

为了帮助使用者理解 SelfSupDataSample 中的基本思想，我们给出一个关于如何创建 `SelfSupDataSample` 实例并设置这些属性的简单例子。

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

selfsup_data_sample = SelfSupDataSample()
# 在 selfsup_data_sample 里加入真实标签数据
# 真实标签数据的类型应与 LabelData 的类型一致
selfsup_data_sample.gt_label = LabelData(value=torch.tensor([1]))

# 如果真实标签数据类型和 LabelData 不一致会报错
selfsup_data_sample.gt_label = torch.tensor([1])
# 报错： AssertionError: tensor([1]) should be a <class 'mmengine.data.label_data.
↪LabelData'> but got <class 'torch.Tensor'>

# 给 selfsup_data_sample 加入样例数据
```

(下页继续)

(续上页)

```
# 同样的，样例数据里的值的类型应与 InstanceData 保持一致
selfsup_data_sample.sample_idx = InstanceData(value=torch.tensor([1]))

# 给 selfsup_data_sample 加面具
selfsup_data_sample.mask = BaseDataElement(value=torch.ones((3, 3)))

# 给 selfsup_data_sample 加假标签
selfsup_data_sample.pseudo_label = InstanceData(location=torch.tensor([1, 2, 3]))


# 创建这些属性后，您可轻而易举得取这些属性里的值
print(selfsup_data_sample.gt_label.value)
# 输出 tensor([1])
print(selfsup_data_sample.mask.value.shape)
# 输出 torch.Size([3, 3])
```

6.2.2 用 MMSelfSup 把数据打包给 SelfSupDataSample

在把数据喂给模型之前，MMSelfSup 按照数据流程把数据打包进 `SelfSupDataSample` 。如果您不熟悉数据流程，可以参考 data transform。我们用一个叫*PackSelfSupInputs*的数据变换来打包数据。

```
class PackSelfSupInputs(BaseTransform):
    """ 把数据打包并让格式能与函数输入匹配

    需要的值:

    - img

    添加的值:

    - data_sample
    - inputs

    参数:
        key (str): 输入模型的图片的值，默认为 img 。
        algorithm_keys (List[str]): 和算法相关的组成部分的值，比如 mask 。默认为 [] 。
        pseudo_label_keys (List[str]): 假标签对应的属性。默认为 [] 。
        meta_keys (List[str]): 图片的 meta 信息的值。默认为 [] 。

    """

    def __init__(self,
```

(下页继续)

(续上页)

```
                 key: Optional[str] = 'img',
                 algorithm_keys: Optional[List[str]] = [],
                 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]:
        """ 打包数据的方法。

        参数:
            results (Dict): 数据变换返回的字典。

        返回:
            Dict:

            - 'inputs' (List[torch.Tensor]): 模型前面的数据。
            - 'data_sample' (SelfSupDataSample): 前面数据的注释信息。
        """
        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
```

(下页继续)

(续上页)

```
        # gt_label, sample_idx, mask, pred_label 在此设置
        for key in self.algorithm_keys:
            self.set_algorithm_keys(data_sample, key, results)

        # 除 gt_label, sample_idx, mask, pred_label 外的值会被设为假标签的属性
        for key in self.pseudo_label_keys:
            # convert data to torch.Tensor
            value = to_tensor(results[key])
            setattr(data_sample.pseudo_label, key, value)

        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:
        """ 设置 SelfSupDataSample 中算法的值."""
        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')
```

在 SelfSupDataSample 中 `algorithm_keys` 是除了 `pseudo_label` 的数据属性, `pseudo_label_keys` 是 SelfSupDataSample 中假标签对应的分支属性。感谢读完整个教程。有问题的话可以在 GitHub 上提 issue，我们会尽快联系您。

6.3 模型

我们可以把模型看作算法的特征提取器或者损失生成器。在 MMSelfSup 中，模型主要包括以下几个部分:

- 算法，包括模型的全部模块和构造算法时需要用到的子模块。
- 主干，里面是每个算法的支柱，比如 MAE 中的 VIT 和 SimMIM 中的 Swin Transformer。
- 颈部，指一些特殊的模块，比如解码器，它直接增加脊柱部分的输出结果。
- 头部，指一些特殊的模块，比如多层感知器的层，它增加脊柱部分或者颈部部分的输出结果。
- 记忆，也就是一些算法中的存储体或者队列，比如 MoCo v1/v2。
- 损失，用于算输出的预测值和目标之间的损失。
- 目标生成器，为自监督学习生成优化目标，例如 HOG，其它模块抽取的特征（DALL-E，CLIP）等.

6.3.1 MMSelfSup 模型概述

首先，我们纵览 MMSelfSup 中已有的模型。我们根据上述的分类来展示这些模型。

6.3.2 用子模块来构造算法

正如上表所述，每个算法都是主干，颈部，头部，损失和记忆的结合体。您可以从这些模块中任意选出若干部分来构建你自己的算法。如果需要定制化的模块，您可参考*add_modules* 中的内容。MMSelfSup 提供一个基础模型，名为 `BaseModel`，所以的算法都应该继承这个基础模型，而且所有子模块（除了记忆部分）在基础模型中进行初始化。记忆部分在对应算法的 `__init__` 中被构造。损失部分在头部部分初始化时被构造。

```
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):
```

(下页继续)

(续上页)

```
        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)
```

正如上面代码所示，构造主干部分时需要配置，但是对颈部和头部而言这可有可无。除了构造算法之外，您还需要重写基础模型中的一些抽象函数才能得到正确结果，我们将在下一部分讨论这件事。

6.3.3 基础模型中的抽象函数

`forward` 函数是结果的入口。然而，它和大多数 Pytorch 代码中只有一种模式的 `forward` 函数不同。MMSelfSup 把所有的逻辑都混杂在 `forward` 中，从而限制了该方法的可拓展性。正如下面代码所示，MMSelfSup 中的 `forward` 函数根据不同模式进行前向处理，目前共有三种模式：张量，损失和预测。

```
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}".')
```

- 张量，如果模式为 `tensor`，`forward` 函数就返回从图片提取到的特征。您应该重写其中的 `extract_feat` 部分才能让定制化的提取过程有效。
- 损失，如果模式为 `loss`，`forward` 函数就返回预测值与目标之间的损失。同样的，您应该重写其中的 `loss` 部分才能让定制化的提取过程有效。
- 预测，如果模式为 `predict`，`forward` 函数就返回预测结果，比如用您的算法预测得到的标签。如果需要，`predict` 函数也需要重写。

本文中我们学习了 MMSelfSup 中的模型的基本组成部分，如果您想深入研究，可以参考每个算法的 API 文件。

6.4 数据集

`mmselfsup` 算法库中的 `datasets` 文件夹包罗了各种与数据加载相关的模块文件。此文件夹主要分为如下三个部分：

- 自定义数据集，用于图像读取与加载
- 自定义数据集采样器，用于图像加载之前进行索引的读取
- 数据变换工具，用于在数据输入模型之前进行数据增强，如 `RandomResizedCrop`

在本教程中，我们将对三个部分依次进行较为详尽的解释。

6.4.1 数据集

OpenMMLab 开源算法体系为用户提供了海量开箱即用的数据集，这些数据集都与 BaseDataset 一脉相承，并在 MMEngine 中得以实现。如想进一步了解 `BaseDataset` 中的各项功能，感兴趣的用户可以参考 MMEngine 中的文档。对于 `MMSelfSup`，`ImageNet`、`ADE20KDataset` 和 `CocoDataset` 是三个较为常用的数据集。在起步之前，用户需要对文件夹进行一些前置的重构工作，具体指南如下所述。

重构个人数据集

万事俱备，只欠东风。使用准备好的这些数据集，用户需要将数据集重构为如下格式。

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

更为详尽的注释文件以及各子文件夹的结构，可以参考 OpenMMLab 的其他代码库，如 MMClassfication, MMSegmentation 和 MMDetection.

在个人 Config 中调用 OpenMMLab 其他代码库的数据集

```
# 调用 MMClassification 中的 ImageNet 数据集
# 在数据加载器中调用 ImageNet
# 为简单起见，我们只提供与从 MMClassification 导入 ImageNet 的相关 config
# 而不是数据加载器的全量的 config
# ``mmcls`` 前缀传达给 ``Registry`` 需要在 MMClassification 中搜索 ``ImageNet``
train_dataloader=dict(dataset=dict(type='mmcls.ImageNet', ...), ...)
```

```
# 调用 MMSegmentation 中的 ADE20KDataset 数据集
# 在数据加载器中使用 ADE20KDataset
```

(下页继续)

(续上页)

```
# 为简单起见，我们只提供与从 MMSegmentation 导入 ADE20KDataset 的相关 config
# 而不是数据加载器的全量的 config
# ``mmseg`` 前缀传达给 ``Registry`` 需要在 MMSegmentation 中搜索 ``ADE20KDataset``
train_dataloader=dict(dataset=dict(type='mmseg.ADE20KDataset', ...), ...)
```

```
# 在数据加载器中调用 CocoDataset
# 为简单起见，我们只提供与从 MMDetection 导入 CocoDataset 的相关 config
# 而不是数据加载器的全量的 config
# ``mmdet`` 前缀传达给 ``Registry`` 需要在 MMDetection 中搜索 ``CocoDataset``
train_dataloader=dict(dataset=dict(type='mmdet.CocoDataset', ...), ...)
```

```
# 在 MMSelfSup 中调用数据集，如 ``DeepClusterImageNet``
train_dataloader=dict(dataset=dict(type='DeepClusterImageNet', ...), ...)
```

通过上文，我们介绍了调用数据集的两个关键的步骤，希望用户可以掌握如何在 MMSelfSup 中使用数据集的相关基本概念。如果用户有创建自定义数据集的意愿，可参考文档*add_datasets*。

6.4.2 采样器

在 pytorch 中，`Sampler` 用于在加载之前对数据的索引进行采样。`MMEngine` 中已经实现和开源了 `DefaultSampler` 和 `InfiniteSampler`。大多数情况下，我们可以直接调用，无需手动去实现自定义采样器。然而 `DeepClusterSampler` 是一个值得一提的特例，因为其中纳入了进行唯一索引采样的逻辑，因此，如果用户想对此采样器的相关信息一览无遗，则可进一步参考我们的 API 文档。如果你有自行实现自定义采样器的更进一步的想法，同样可以参考 `DeepClusterSampler` 在 `samplers` 文件夹进行实现。

6.4.3 数据变换

简而言之，`transform` 是指 `MM-repos` 中的数据变换模块，我们将一系列的 transform 组合成了一个列表，即 `pipeline`。`MMCV` 中已经完善了一些涵盖大多数场景的变换，此外，每个 `MM-repo` 也都遵循 MMCV 中的 用户指南 定义了自己的变换。实操而言，每个自定义的数据集需要：i) 继承 BaseTransform，ii) 覆盖 `transform` 函数并在其中实现自行设计的关键逻辑。在 MMSelfSup 中，我们已经实现了如下这些变换：

对于感兴趣的社区用户，可以参考 API 文档以更为全面了解这些转换。目前为止，我们已经初步介绍了关于转换的基本概念，若想进一步了解如何在个人的 config 中使用它们或实现自定义转换，可以参考文档：*transforms* 和*add_transforms*.

6.5 数据变化

6.5.1 数据变换概述

在*add_transforms* 中我们介绍了如何构建 `Pipeline` 。`Pipeline` 里有一系列的数据变换。MMSelfSup 中数据变换主要分为三类：

1. 处理数据用到的数据变换。processing.py 中定义了独特的数据变换，比如 `RandomCrop`，`RandomResizedCrop` 和 `RandomGaussianBlur`。我们也可以用其它仓库的数据变换，比如 MMCV 中的 `LoadImageFromFile`。
2. 不同视角看同一照片的数据变换打包器。这个定义在 wrappers.py。
3. 将数据变换使得数据能输入算法中。这个定义在 formatting.py。

总的来说，我们用的是如下的这些数据变换。我们将详细讨论最后两种数据变换。

6.5.2 MultiView 简介

我们为一些算法定义了名为*`MultiView`* 的多角度照片输入的封装器，比如 MoCo 系列，SimCLR，SwAV 等。在配置文件中，我们能这样定义：

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

这意味着数据管道里面有两个角度。

我们也可以这样定义有不同角度的数据管道：

```
pipeline = [
     dict(type='MultiView',
          num_views=[2, 6],
          transforms=[
```

(下页继续)

(续上页)

```
            [
              dict(type='Resize', scale=224)],
            [
              dict(type='Resize', scale=224),
              dict(type='RandomSolarize')],
          ])
]
```

这意味着有两个数据管道，他们分别有两个角度和六个角度。在 imagenet_mocov1.py 和 imagenet_mocov2.py 和 imagenet_swav_mcrop-2-6.py 中有更多例子。

6.5.3 PackSelfSupInputs 简介

我们定义了一个名为*PackSelfSupInputs* 的类来将数据转换为能输入算法中的格式。这种转换通常在数据管道的最后，就像下面这样：

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

6.6 模型评测

6.6.1 MMEngine 中的评测

在模型验证和测试过程中，经常需要进行定量评估. 在 MMEngine 中已经实现了 `Metric` 和 `Evaluator` 来执行此功能. 详情请见MMEngine Doc.

模型评估分为在线评测和离线评测.

在线评测

在线评测用于 `ValLoop` 和 `TestLoop` 中.

以 `ValLoop` 为例:

```
...
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')
        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)
```

离线评测

离线评测使用保存在文件中的预测结果. 在这种情况下, 由于没有 Runner, 我们需要构建 Evaluator 并调用 offline_evaluate() 函数.

一个例子:

```
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 MMSelfSup 中的评测

在预训练期间，因为不包含验证和测试，所以不需要使用模型评测.

在基准测试期间, 预训练模型需要其他的下游任务来评测其性能, 例如 classification、detection、segmentation 等. 建议使用其他的 OpenMMLab 仓库运行下游任务, 例如 MMClassification 或 MMDetection, 它们已经实现了自己评估功能.

但是 MMSelfSup 也实现了某些自定义的评测功能去支持下游任务, 如下所示:

- knn_classifier()

用于计算 knn 分类器预测的准确性, 并且用于 KNN evaluation.

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

- ResLayerExtraNorm

为原始的 ResLayer 增加了额外的规范, 并在 mmdetection 基准配置中使用.

```
model = dict(
    backbone=...,
    roi_head=dict(
        shared_head=dict(
            type='ResLayerExtraNorm',
            norm_cfg=norm_cfg,
```

(下页继续)

(续上页)

```
            norm_eval=False,
            style='pytorch')))
```

6.6.3 自定义评测

支持 Metric 和 Evaluator 的自定义评测, 详情请见 MMEngine Doc

6.7 训练引擎

6.7.1 钩子 (Hook)

介绍

钩子机制在 OpenMMLab 开源算法库中被广泛使用, 嵌入在 `Runner` 中, 可以轻松管理训练过程的整个生命周期. 您可以通过相关文章了解有关钩子的更多信息.

钩子只有在 `Runner` 注册后才能工作. 目前钩子主要分为两类:

- 默认钩子 (default hooks)

这些钩子由 `Runner` 默认注册, 用以实现一些基本功能, 并且具有默认的优先级, 无需用户自行修改.

- 自定义钩子 (custom hooks)

自定义钩子通过 `custom_hooks` 注册. 通常来讲, 这些钩子主要用于功能增强, 并且需要在配置文档中指定优先级. 若没有指定钩子的优先级, 则默认情况下会设置为 `NORMAL`.

优先级列表:

优先级决定了钩子的执行顺序. 在训练之前，日志会打印出每个阶段的钩子的执行顺序，方便调试.

默认钩子

以下常见的钩子由 MMEngine 中的`register_default_hooks` 实现并在 default 中进行了注册:

MMEngine 中实现的常用钩子

在 MMEngine 中已经实现了一些钩子，它们是:

MMSelfsup 中实现的钩子

在 MMSelfsup 中已经实现了一些钩子，它们是:

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

例如:

以*DenseCLHook* 为例, 这个钩子包含了 DenseCL 中的 `loss_lambda` 预热.

`loss_lambda` 是单一和稠密对比损失的损失权重. 默认值为 0.5.

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

DenseCLHook 实现如下:

```
...
@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.
```

若钩子已在 MMEngine 或 MMSelfsup 中实现，则可以直接修改配置来使用这个钩子，如下所示:

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

例如使用 DenseCLHook, start_iters 是 500:

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

6.7.2 优化器

下面将通过 3 个不同的部分来介绍优化器章节: 优化器、优化器封装和构造器

优化器

定制 PyTorch 支持的优化器

我们已经支持了 PyTorch 实现的所有优化器，可参阅 `mmengine/optim/optimizer/builder.py`. 若要使用或修改, 请更改配置文件的 `optimizer` 字段.

例如, 若要使用 SGD, 则可进行如下修改.

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

要修改模型的学习率，只需优化器配置中修改 `lr` . 同时也可以根据 PyTorch 的 API doc 直接设置其他参数.

例如, 如果您期望使用如 PyTorch 中 `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)` 的 Adam 设置, 配置应该看起来像:

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

参数配置

有些模型在优化时可能有一些特定的参数设置, 例如不需要将权重衰减应用到 `BatchNorm` 层和每一层的 `bias` 中. 为了精确地配置它们, 我们可以使用优化器中的 `paramwise_cfg`.

例如, 在 MAE 中, 我们不想对 `ln`, `bias`, `pos_embed`, `mask_token` 和 `cls_token` 等参数应用权重衰减, 我们可以使用以下配置文件:

```
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.)
        }))
```

MMSelfsup 中实现的优化器

- *LARS*

除了 PyTorch 实现的优化器之外, 我们还在 `mmselfsup/engine/optimizers/lars.py` 中实现了一个定制的*LARS*, 为 SGD 实现了分层自适应学习率缩放.

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

优化器封装

除了 PyTorch 优化器的基本功能外, 我们还提供了一些增强功能, 例如梯度裁剪, 梯度累加, 自动混合精度训练等. 更多细节请参考 MMEngine.

梯度裁剪

目前我们在 `optim_wrapper` 中支持 `clip_grad` 选项，您可以参考 OptimWrapper 和 PyTorch Documentation 文档了解更多的参数. 下面是一个例子:

```
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.
```

如果 `clip_grad` 不是 `None`, 它将是 `torch.nn.utils.clip_grad.clip_grad_norm_()` 的参数.

梯度累加

当没有足够的计算资源时, 批量大小只能设置为一个小批量, 这可能会降低模型的性能. 梯度累加可以用来解决这个问题.

下面是一个例子:

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

这个例子表示在训练期间, 每 4 个 iter 进行一次反向传播. 并且上述内容相当于:

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

自动混合精度 (AMP) 训练

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

AmpOptimWrapper 中 loss_scale 的默认设置是 dynamic.

构造器

构造器旨在建立优化器、优化器封装以及定制不同层的超参数. 配置文件中 optim_wrapper 的 paramwise_cfg 函数控制这种定制.

MMSelfsup 中实现的构造器

- *LearningRateDecayOptimWrapperConstructor*

LearningRateDecayOptimWrapperConstructor 为主干网络的不同层设置不同的学习率. 注意: 目前, 这个优化器构造器是为 ViT, Swin, MixMIM 构建的.

一个例子:

```
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')
```

注意: 在 LearningRateDecayOptimWrapperConstructor 中, paramwise_cfg 只支持 weight_decay 的自定义.

6.8 约定

如果您想将 MMSelfSup 修改为您自己的项目，请检查以下约定。

6.8.1 损失

当算法实现时，函数 `loss` 返回的损失应该是 `dict` 类型。

举个 `MAE` 的例子：

```
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]:
        ...

    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)
```

(下页继续)

(续上页)

```
        return losses
```

在 MAE 模型正向传播期间, 这个 MAE.loss() 函数将被调用用于计算损失并返回这个损失值。

默认情况下, 只有 dict 中的键包含的 loss 值时, 才会进行反向传播, 如果你的算法需要多个损失值, 你可以用多个键打包损失字典。

```
class YourAlgorithm(BaseModel):

    def loss():
        ...

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

CHAPTER 7

组件模块自定义

7.1 添加模块

在本教程中，我们将要介绍创建用户自定义模块的基本步骤。在学习创建自定义模块之前，建议先了解一下文件*models.md*中模型的基本的概念。您可以自定义*models.md*文件中涉及的所有模型组件，例如**主干网络 (backbone)**, **neck**, **head** 和**损失 (loss)**。

7.1.1 添加新的主干网络

假如要创建新的主干网络 `NewBackbone`。

1. 新建一个文件 `mmselfsup/models/backbones/new_backbone.py` 并在其中实现新的主干网络 `NewBackbone`。

```
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

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

2. 导入新添加的主干网络到 mmselfsup/models/backbones/__init__.py。

```
...
from .new_backbone import NewBackbone

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

3. 在配置文件中使用自定义的主干网络。

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

7.1.2 添加新的 neck

您可以通过继承 mmengine 中的 `BaseModule` 来创建新的 neck，并且重定义其中的 `forward` 函数。我们在 mmengine 中有统一的权重初始化接口，您可以使用 `init_cfg` 来指定初始化函数和参数，或者可以重定义函数 `init_weights` 如果您希望使用自定义初始化方式。

所有的已有的 neck 都在 `mmselfsup/models/necks` 中。假如您要创建新的 neck `NewNeck`。

1. 新建一个文件 `mmselfsup/models/necks/new_neck.py` 并在其中实现 `NewNeck`。

```
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
```

您需要在 `forward` 函数中实现一些针对主干网络输出的操作，并将结果给 head。

2. 导入新定义的 neck 模块到 `mmselfsup/models/necks/__init__.py`。

```
...
from .new_neck import NewNeck

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

3. 在配置文件中使用自定义的 neck 模块。

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

(下页继续)

(续上页)

```
)
```

7.1.3 添加新的 head

您可以通过继承 mmengine 中的 `BaseModule` 来创建新的 head，并且重定义其中的 `forward` 函数。

所有已有的 head 都在 `mmselfsup/models/heads` 文件中。假如您想创建新的 head `NewHead`。

1. 创建文件 `mmselfsup/models/heads/new_head.py` 并在其中实现 `NewHead`。

```python
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
```

您需要在 `forward` 函数中实现一些针对主干网络或 neck 输出的操作，并计算损失。请注意，损失模块应构建在 head 模块中以进行损失计算。

2. 在 `mmselfsup/models/heads/__init__.py` 中导入新创建的 head 模块。

```python
...
from .new_head import NewHead

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

3. 在配置文件中使用自定义的 head。

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

7.1.4 添加新的损失

添加新的损失函数时，主要需要在 loss 模块中实现 `forward` 函数。同时您需要将 loss 模块也注册 (register) 为 `MODELS`。

所有已有的损失函数都在 `mmselfsup/models/losses` 中。假如您想创建新的损失 `NewLoss`。

1. 创建文件 `mmselfsup/models/losses/new_loss.py` 并在其中实现 `NewLoss`。

```
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. 在 `mmselfsup/models/losses/__init__.py` 中导入新定义的 loss 模块。

```
...
from .new_loss import NewLoss

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

3. 在配置文件中使用自定义的 loss 模块。

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

7.1.5 组合起来

在创建好上述的各个模块组件之后，我们需要创建一个新的算法 NewAlgorithm 来将各个组件按照逻辑顺序组合起来。NewAlgorithm 使用原始图像作为输入并输出损失函数值给优化器 (optimizer)。

1. 创建文件 mmselfsup/models/algorithms/new_algorithm.py 并在其中实现 NewAlgorithm。

```
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. 在 mmselfsup/models/algorithms/__init__.py 中导入新创建的算法模块 NewAlgorithm。

```
...
from .new_algorithm import NewAlgorithm

__all__ = [
```

(下页继续)

(续上页)

```
    ...
    'NewAlgorithm',
    ...
]
```

3. 在配置文件中使用自定义的算法模块。

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

7.2 添加数据集

在本教程中，我们介绍了创建自定义数据集的基本步骤。在学习创建自定义数据集之前，建议学习文件*datasets.md* 中数据集的基本概念。

如果您的算法不需要任何自定义数据集类，您可以使用*datasets directory* 里的现成的数据集类。但使用这些现有的数据集类，您必须将您的数据集转换成现有数据集类要求的格式。

关于图像预训练，建议遵循 MMClassification 的格式。

7.2.1 步骤 1: 创建数据集

您可以实现一个新的数据集类，它继承自 MMClassification 的 `CustomDataset`，用于图像预训练。

假如您的数据集类为 `NewDataset`，您可以在 `mmselfsup/datasets` 下创建文件 `new_dataset.py` 并在其中实现自定义数据集类 `NewDataset`。

```
from typing import List, Optional, Union

from mmcls.datasets import CustomDataset

```

(下页继续)

(续上页)

```
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}
        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 步骤 2: 添加数据集到 __init__py

然后将 NewDataset 导入到 mmselfsup/dataset/__init__.py 中。如果没有导入，则 NewDataset 没有注册 (register) 成功。

```
...
from .new_dataset import NewDataset

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

7.2.3 步骤 3: 修改配置文件

使用 NewDataset 时，您可以参考下面修改配置文件:

```
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 添加数据变换

在本教程中，我们将介绍创建自定义转换的基本步骤。在学习创建自定义转换之前，建议先了解文件*transforms.md* 中转换的基本概念。

7.3.1 管道概述

在 Dataset 中，Pipeline 是中的一个重要组件，主要负责对图像应用一系列数据增强，例如: RandomResizedCrop, RandomFlip 等操作。

以下代码是 Pipeline 用于 SimCLR 训练的配置示例:

```
view_pipeline = [
    dict(type='RandomResizedCrop', size=224, backend='pillow'),
    dict(type='RandomFlip', prob=0.5),
    dict(
        type='RandomApply',
        transforms=[
            dict(
                type='ColorJitter',
```

(下页继续)

(续上页)

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

在这个 Pipeline 中, 每个数据增强接收一个 dict , 它们作为输入和输出时刻, 包含图像增强以及其他相关信息的 dict 。

7.3.2 在管道中创建新转换

以下是创建新转换的步骤。

步骤 1: 创建转换

在 processing.py 中编写一个新的转换类, 并在类中覆盖这个 transform 函数, 这个函数接收一个 dict 的对象, 并返回一个 dict 对象

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

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

注意: 对于这些转换的实现, 您可以应用 mmcv 中的函数。

步骤 2: 将新转换添加到 __init__py

然后, 将转换添加到 __init__.py 。

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

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

步骤 3: 修改配置文件

要使用新添加的 NewTransform, 你可以按以下的方式修改配置文件:

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

7.4 自定义运行

在本教程中，我们将介绍一些关于如何设置项目自定义运行的方法。

7.4.1 循环 (Loop)

Loop 表示训练、验证或测试的工作流，我们使用 train_cfg,val_cfg 和 test_cfg 来构建 Loop。

示例:

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

MMEngine 定义了几个基本的循环。如果定义的循环不满足需求，用户可以实现自定义的循环。

7.4.2 钩子 (Hook)

在学习创建自定义钩子之前，建议先学习文件*engine.md*中关于钩子的基本概念。

步骤 1：创建一个新的钩子

根据钩子的目的，您需要根据期望的钩子点实现相应的函数。

例如，如果您想根据训练迭代次数和另外两个超参数的值在每个训练迭代后修改超参数的值，您可以实现一个类似以下的钩子：

```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
```

步骤 2：导入新的钩子

然后我们需要确保 NewHook 已经被导入。假设 NewHook 在 mmselfsup/engine/hooks/new_hook.py 中，按照以下方式修改 mmselfsup/engine/hooks/__init__.py 文件：

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

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

步骤 3：修改配置文件

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

您还可以按照以下方式设置钩子的优先级：

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

默认情况下，在注册时，钩子的优先级被设置为 NORMAL。

7.4.3 优化器 (Optimizer)

在自定义优化器配置之前，建议先学习文件*engine.md*中有关优化器的基本概念。

以下是 SGD 优化器的示例：

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

我们支持 PyTorch 中的所有优化器。更多细节，请参见MMEngine 优化器文档。

优化器包装器

优化器包装器提供了单精度训练和不同硬件的自动混合精度训练的统一接口。以下是一个 optim_wrapper 配置的示例：

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

此外，如果您想要应用自动混合精度训练，可以修改上面的配置，例如：

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

AmpOptimWrapper 的 loss_scale 的默认设置为 dynamic。

构造器

构造器旨在构建优化器、优化器包装器并自定义不同层的超参数。配置文件中 optim_wrapper 的 paramwise_cfg 键控制此自定义。

有关示例和详细信息，请参见MMEngine 优化器文档。

此外，我们可以使用 custom_keys 为不同模块设置不同的超参数。

以下是 MAE 的 optim_wrapper 示例。以下配置将 pos_embed, mask_token, cls_token 模块和名称包含 ln 和 bias 的那些层的权重衰减的乘法乘数设置为 0。在训练过程中，这些模块的权重衰减将是 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.)
        }))
```

此外，对于某些特定设置，我们可以使用布尔类型的参数来控制优化过程或参数。例如，以下是 SimCLR 的示例配置：

```
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),
        }))
```

在 LARS 优化器中，我们有 lars_exclude 选项来决定指定的层是否应用 LARS 优化方法。

7.4.4 调度器 (Scheduler)

在自定义调度器配置之前，建议先学习 MMEngine 文档 中关于调度器的基本概念。

以下是一个调度器的示例：

```
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)
]
```

注意： 当您更改 `train_cfg` 中的 `max_epochs` 时，请确保同时修改 `param_scheduler` 中的参数。

CHAPTER 8

模型库数据汇总

- 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 9

模型库

本部分内容主要介绍 MMSelfSup 支持的模型和部分下游任务的评测结果。

9.1 下游任务评测

9.1.1 ImageNet

ImageNet 有多个版本，不过最常用的是 ILSVRC 2012。我们提供了基于各类算法的预训练模型的分类结果，包括线性评估和微调，同时有对应的模型和日志文件。

CHAPTER 10

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 11

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 12

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 13

BYOL

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

13.1 Abstract

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

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 14

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 15

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 16

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 17

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 18

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 19

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 20

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 21

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 22

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 23

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 24

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 25

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 26

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 27

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 28

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 29

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 30

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 31

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 32

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 33

迁移文档

33.1 迁移自 MMSelfSup 0.x 版本

警告： MMSelfSup 1.x 版本依赖于一些新的代码包，您应该根据安装教程 来创建新的环境，尽管你可能已经拥有了一个可以正常运行 MMSelfSup 0.x 的环境。请参考安装文档 对依赖库进行对应的安装。

我们将介绍一些 MMSelfSup 1.x 版本的变换，帮助用户更顺利的将项目从 MMSelfSup 0.x 版本迁移到 1.x 版本。

三个重要的依赖库已列出：

1. MMEngine: MMEngine 是所有 OpenMMLab 2.0 项目的基础库，一部分非计算机视觉强相关的模块从 MMCV 迁移到了 MMEngine。
2. MMCV: OpenMMLab 计算机视觉基础库。这不是新的依赖项，但是您需要将其升级到至少 `2.0.0rc1` 版本。
3. MMClassification: OpenMMLab 图像分类代码库。这不是新的依赖项，但是您需要将其升级到至少 `1.0.0rc0` 版本。

33.2 配置文件

本章节将介绍 `_base_` 文件夹中的配置文件的变化，主要包含以下三个部分：

- 数据集：`mmselfsup/configs/selfsup/_base_/datasets`
- 模型：`mmselfsup/configs/selfsup/_base_/models`
- 优化器及调度：`mmselfsup/configs/selfsup/_base_/schedules`

33.2.1 数据集

在 **MMSelfSup 0.x** 中，我们使用字段 `data` 来整合数据相关信息, 例如 `samples_per_gpu`，`train`，`val` 等。

在 **MMSelfSup 1.x** 中，我们分别使用字段 `train_dataloader`, `val_dataloader` 整理训练和验证的数据相关信息，并且 `data` 字段已经被 **移除**。

```
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,
```

(下页继续)

(续上页)

```
    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 = ...
```

另外，我们 **移除**了字段 `data_source`，以此来保证我们项目和其它 OpenMMLab 项目数据流的一致性。请查阅*Config* 获取更详细的信息。

`pipeline` 中的变化：

以 MAE 的 `pipeline` 作为例子，新的写法如下：

```
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 模型

在模型的配置文件中，和 MMSeflSup 0.x 版本相比，主要有两点不同。

1. 有一个新的字段 `data_preprocessor`，主要负责对数据进行预处理，例如归一化，通道转换等。例子如下：

```
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),
```

(下页继续)

(续上页)

```
    backbone=...,
    neck=...,
    head=...,
    init_cfg=...)
```

注意： data_preprocessor 可以被定义在模型字段之外，外部定义将拥有更高优先级，并且覆盖模型字段内部定义。

例如以下写法中，Runner 将会基于 mean=[123.675, 116.28, 103.53] 和 std=[58.395, 57.12, 57.375] 这套参数进行构建 data_preprocessor，而忽略 127.5 的参数。

```
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=...)
```

相关 MMEngine 代码链接：Runner 会获取 cfg.data_preprocessor，并且 合并 进 cfg.model。

2. 在新版本的 head 字段中，我们新增加了 loss，主要负责损失函数的构建。例子如下：

```
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 优化器及调度

1. **optimizer** 和 **optimizer_config** 的变化：

- 现在我们使用 optim_wrapper 字段来说明所有优化过程相关的设置，而 optimizer 是 optim_wrapper 的一个子字段。
- paramwise_cfg 也是 optim_wrapper 的一个子字段，而不再是 optimizer 的子字段。
- optimizer_config 以经被**移除**，所有优化相关配置定义在 optim_wrapper 中。
- grad_clip 重命名为 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. **lr_config** 的变化：

- lr_config 已经被**移除**，并且我们使用新的字段 param_scheduler 来代替它。
- warmup 相关字段也被**移除**，因为我们使用学习率调度器组合来完成这项功能。新的调度器组合功能非常灵活，您可以用它设计各种不同的学习率或者动量变化曲线。参考 教程 获取更多信息。

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

3. **runner** 的变化:

在原来的 runner 字段中的配置已经被移到 train_cfg，val_cfg 和 test_cfg 当中，主要控制训练、验证、测试等循环流程。

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

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

33.2.4 运行相关设置

1. **checkpoint_config** 和 **log_config** 的变化:

checkpoint_config 相关配置被移动到了 default_hooks.checkpoint ， 而 log_config 被移动到了 default_hooks.logger。

并且，我们将一些钩子相关的设置均移进 default_hooks 字段进行统一管理。

```
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.
```

(下页继续)

(续上页)

```
    visualization=dict(type='VisualizationHook', enable=False),
)
```

另外，我们将原有的 logger 拆分为 logger 和 visualizer。logger 主要负责信息记录，而 visualizer 则是控制在不同后端来展示记录的信息，例如终端，TensorBoar 和 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. **load_from** 和 **resume_from** 的变化：

- resume_from 已经被**移除**，我们使用 resume 和 load_from 代替它：
  - 如果 resume=True 并且 load_from 不是 None，将读取 load_from 字段的模型文件继续训练。
  - 如果 resume=True 并且 load_from 是 None，将在工作目录中尝试读取最近的模型文件继续训练。
  - 如果 resume=False 并且 load_from 不是 None，则只读取模型文件，不会继续训练。
  - 如果 resume=False 并且 load_from 是 None，不会读取模型文件，也不会继续训练，即随机初始化重新训练。

3. **dist_params** 的变化：

dist_params 字段现在是 env_cfg 的一部分，另外现在有一些新的配置在 env_cfg 当中。

```
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
```

(下页继续)

(续上页)

```
    dist_cfg=dict(backend='nccl'),
)
```

4. **workflow** 的变化：workflow 相关字段已经被 **移除**.

5. 新字段 **visualizer**：

可视化器是 OpenMMLab 2.0 架构新设计的一部分。在 runner 中，我们使用可视化器的实例来处理结果和日志的可视化，并且将对应数据储存到不同的后端。请查阅 MMEngine 可视化文档 获取更多信息。

```
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')
    ]
)
```

6. 新字段 **default_scope**：起始点来搜索所有注册的模块。MMSelfSup 的 default_scope 即是 mmselfsup。请查阅 注册机制文档 获取更多信息

33.3 代码包

下列表格记录了代码模块、文件夹的主要改变。

CHAPTER 34

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.

参数

- **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*.

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

返回 None

prepare_data(*idx: int*) → Any

Get data processed by `self.pipeline`.

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

返回 Depends on `self.pipeline`.

返回类型 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
    ...
```

参数

- **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.

返回 A list of data information.

返回类型 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`.

参数

- **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

参数

- **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.

返回 The shape of mask.

返回类型 Tuple[int, int]

transform(*results: dict*) → dict

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

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

返回 Result dict with added key `mask`.

返回类型 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

参数

- **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.

参数

- **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.

返回

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

返回类型 tuple

transform(*results: dict*) → dict

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

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

返回 Results after applying this transformation.

返回类型 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.

参数

- **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.

实际案例

```
>>> # 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.

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

返回 Transformed results.

返回类型 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

参数

- **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.

参数

- **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.

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

返回

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

返回类型 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

参数

- **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.

参数

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

返回

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

返回类型 tuple

transform(*results: dict*) → dict

Randomly crop the image.

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

返回 Result dict with the transformed image.

返回类型 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

参数

- **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.

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

返回 Results after applying this transformation.

返回类型 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.

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

返回 Results after applying this transformation.

返回类型 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

参数

- **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.

参数

- **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.

返回

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

返回类型 tuple

transform(*results: dict*) → dict

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

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

返回 Result dict with the transformed image.

返回类型 dict

class mmselfsup.datasets.transforms.**RandomResizedCropAndInterpolationWithTwoPic**(*size: Union[tuple, int], second_size=None, scale=(0.08, 1.0), ratio=(0.75, 1.3333333333 interpolation='bilinear', second_interpolatio*

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.

参数

- **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.

参数

- **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

返回

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

返回类型 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.

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

返回 Results after applying this transformation.

返回类型 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

参数

- **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.

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

返回

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

返回类型 float

transform (*results: dict*) → dict

Randomly rotate the image.

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

返回 Result dict with the transformed image.

返回类型 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

参数

- **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.

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

返回 Results after applying this transformation.

返回类型 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.

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

返回 Results after applying this transformation.

返回类型 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.

参数

- **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.

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

返回 Result dict with added key `mask`.

返回类型 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`.

参数

- **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.

参数

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

返回 None

CHAPTER 35

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.

参数

- **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.

参数

- **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.

参数 **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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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:.

参数

- **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.

示例

```
>>> 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.

参数 **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.

参数

- **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 36

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.

参数

- **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.

返回 The top1 and top5 accuracy.

返回类型 Tuple[float, float]

CHAPTER 37

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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

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

返回 The output of specified stage. The output depends on detailed implementation.

返回类型 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.

参数

- **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.

返回

- 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.

返回类型 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.

参数

- **inputs** (*torch.Tensor*) –The input tensor with shape (N, C, ···) in general.
- **data_samples** (*List[SelfSupDataSample]*) –The annotation data of every samples.

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

参数

- **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.

参数

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

返回 A dictionary of loss components.

返回类型 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*.

参数

- **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.

参数

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 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*.

参数

- **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.

参数

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 *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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数 **inputs** (*List[torch.Tensor]*) –The input images.

返回 Neck outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 *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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

返回 Neck outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 *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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

参数

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

参数

- **inputs** (*List[torch.Tensor]*) –The input images.
- **data_samples** (*List[SelfSupDataSample]*) –All

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

- **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.

参数

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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*.

参数

- **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.

参数

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

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 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.

参数 **inputs** (*List[torch.Tensor]*) –The input images.

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 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.

参数 **inputs** (*List[torch.Tensor]*) –The input images.

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 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.

参数 **inputs** (*List[torch.Tensor]*) –The input images.

返回 Backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The reconstructed images.

返回类型 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.

参数

- **inputs** (*List[torch.Tensor]*) –The input images.

- **data_samples** (*List[SelfSupDataSample]*) –All elements required during the forward function.

返回 A dictionary of loss components.

返回类型 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.

参数

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

返回 The prediction from model.

返回类型 *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.

参数 **inputs** (*List[torch.Tensor]*) –The input images.

返回 Backbone outputs.

返回类型 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.

参数

- **inputs** (*List[torch.Tensor]*) –The input images.

- **data_samples** (*List[SelfSupDataSample]*) –All elements required during the forward function.

返回 A dictionary of loss components.

返回类型 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.

参数 **inputs** (*List[torch.Tensor]*) –The input images.

返回 backbone outputs.

返回类型 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.

参数

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

返回 A dictionary of loss components.

返回类型 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

参数

- **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.

参数

- **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]).

返回 Hidden features.

返回类型 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

参数

- **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.

参数

- **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.

返回 hidden features.

返回类型 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.

参数

- **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.

参数 **x** (*torch.Tensor*) –Input images, which is of shape B x C x H x W.

返回

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.

返回类型 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.

参数

- **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.

返回

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.

返回类型 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.

参数

- **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.

返回 masked image, mask, the ids to restore original image, ids of the shuffled patches, ids of the kept patches, ids of the removed patches.

返回类型 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.

参数

- **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.

返回

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.

返回类型 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.

参数

- **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.

参数

- **x** (*torch.Tensor*) –Input images.
- **mask** (*torch.Tensor*) –Input masks.

返回 Features with cls_tokens.

返回类型 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>‘_

参数

- **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.

参数 **x** (*torch.Tensor*) –Input images, which is of shape B x C x H x W.

返回

- 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.

返回类型 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.

参数

- **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.

返回

- 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.

返回类型 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.

参数

- **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.

参数 **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*.

参数

- **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.

示例

```
>>> 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.

参数

- **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*) –

示例

```
>>> 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.

参数

- **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.

参数

- **x** (*torch.Tensor*) –Input images.
- **mask** (*torch.Tensor*) –Masks used to construct masked images.

返回 A tuple containing features from multi-stages.

返回类型 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.

参数

- **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.

参数

- **x** (*Tuple[torch.Tensor]*) –Features of tokens.
- **rel_pos_bias** (*torch.Tensor*) –Shared relative position bias table.

返回

- 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.

返回类型 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.

参数

- **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.

参数

- **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.

返回

Final prediction and latent prediction.

返回类型 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.

参数

- **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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –feature map of backbone.

返回

The global feature vectors and dense feature vectors. - avgpooled_x: Global feature vectors. - x: Dense feature vectors. - avgpooled_x2: Dense feature vectors for queue.

返回类型 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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –The feature map of backbone.

返回 The output features.

返回类型 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

参数

- **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.

示例

```
>>> 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.

参数

- **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.

返回

The reconstructed feature vectors, which is of shape B x (num_patches) x C.

返回类型 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.

参数

- **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.

参数

- **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.

返回

The reconstructed features, which is of shape (N, L, C).

返回类型 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

参数

- **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.

参数

- **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.

返回

The reconstructed features, which is of shape (N, L, C).

返回类型 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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –The feature map of backbone.

返回 The output features.

返回类型 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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –The feature map of backbone.

返回 The output features.

返回类型 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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –The feature map of backbone.

返回 The output features.

返回类型 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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –The feature map of backbone.

返回 The output features.

返回类型 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.

参数

- **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.

参数

- **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.

参数 **x** (*List[torch.Tensor]*) –list of feature maps, len(x) according to len(num_crops).

返回 The projection vectors.

返回类型 List[torch.Tensor]

forward_projection(*x: torch.Tensor*) → torch.Tensor

Compute projection.

参数 **x** (*torch.Tensor*) –The feature vectors after pooling.

返回 The output features with projection or L2-norm.

返回类型 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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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.

返回

The tuple of loss.

- loss_main (torch.Tensor): Cross entropy loss.
- loss_align (torch.Tensor): MSE loss.

返回类型 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.

参数

- **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.

参数

- **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.

返回 The cross entropy loss.

返回类型 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.

参数 **x** (*List[Tensor] | Tuple[Tensor]*) –Feature maps of backbone, each tensor has shape (N, C, H, W).

返回 A list of class scores.

返回类型 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.

参数

- **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.

参数

- **pos** (*torch.Tensor*) –Nx1 positive similarity.
- **neg** (*torch.Tensor*) –Nxk negative similarity.

返回 The contrastive loss.

返回类型 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.

参数

- **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.

参数

- **input** (*torch.Tensor*) –NxC input features.
- **target** (*torch.Tensor*) –NxC target features.

返回 The cross correlation loss.

返回类型 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.

参数

- **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.

参数

- **input** (*torch.Tensor*) –NxC input features.
- **target** (*torch.Tensor*) –NxC target features.

返回 The latent predict loss.

返回类型 torch.Tensor

class mmselfsup.models.heads.**MAEPretrainHead**(*loss: dict*, *norm_pix: bool = False*, *patch_size: int = 16*)

Pre-training head for MAE.

参数

- **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`.

参数 **target** (*torch.Tensor*) –Image with the shape of B x 3 x H x W

返回 Tokenized images with the shape of B x L x C

返回类型 torch.Tensor

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Forward function of MAE head.

参数

- **pred** (*torch.Tensor*) –The reconstructed image.
- **target** (*torch.Tensor*) –The target image.

- **mask** (*torch.Tensor*) –The mask of the target image.

返回 The reconstruction loss.

返回类型 torch.Tensor

patchify(*imgs: torch.Tensor*) → torch.Tensor

Split images into non-overlapped patches.

参数 **imgs** (*torch.Tensor*) –A batch of images, of shape B x H x W x C.

返回 Patchified images. The shape is B x L x D.

返回类型 torch.Tensor

unpatchify(*x: torch.Tensor*) → torch.Tensor

Combine non-overlapped patches into images.

参数 **x** (*torch.Tensor*) –The shape is (N, L, patch_size**2 *3)

返回 The shape is (N, 3, H, W)

返回类型 imgs (torch.Tensor)

class mmselfsup.models.heads.**MILANPretrainHead**(*loss: dict*)

MILAN pretrain head.

参数 **loss** (*dict*) –Config of loss.

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: Optional[torch.Tensor] = None*) → torch.Tensor

Forward function.

参数

- **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.

返回 the reconstructed loss.

返回类型 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.

参数 **loss** (*dict*) –Config dict for module of loss functions.

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor

Forward head.

参数

- **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.

返回 The loss tensor.

返回类型 torch.Tensor

class mmselfsup.models.heads.**MixMIMPretrainHead**(*loss: dict*, *norm_pix: bool = False*, *patch_size: int = 16*)

MixMIM pretrain head.

参数

- **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.

参数

- **pred** (*torch.Tensor*) –The reconstructed image.
- **target** (*torch.Tensor*) –The target image.
- **mask** (*torch.Tensor*) –The mask of the target image.

返回 The reconstruction loss.

返回类型 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.

参数

- **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.

参数

- **base_out** (*torch.Tensor*) –NxC features from base_encoder.
- **momentum_out** (*torch.Tensor*) –NxC features from momentum_encoder.

返回 The loss tensor.

返回类型 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.

参数

- **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.

参数 **feats** (*Sequence[torch.Tensor]*) –Feature maps of backbone, each tensor has shape (N, C, H, W).

返回 A list of class scores.

返回类型 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.

参数

- **x** (*Sequence[torch.Tensor]*) –Feature maps of backbone, each tensor has shape (N, C, H, W).
- **gt_label** (*torch.Tensor*) –The ground truth label.

返回 Dict of loss and accuracy.

返回类型 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.

参数

- **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.

返回

The data samples containing annotation, prediction, etc.

返回类型 List[BaseDataElement]

class mmselfsup.models.heads.**SimMIMHead**(*patch_size: int*, *loss: dict*)

Pretrain Head for SimMIM.

参数

- **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.

参数

- **pred** (*torch.Tensor*) –The reconstructed image.
- **target** (*torch.Tensor*) –The target image.
- **mask** (*torch.Tensor*) –The mask of the target image.

返回 The reconstruction loss.

返回类型 torch.Tensor

class mmselfsup.models.heads.**SwAVHead**(*loss: dict*)

Head for SwAV.

参数 **loss** (*dict*) –Config dict for module of loss functions.

forward(*pred: torch.Tensor*) → torch.Tensor

Forward function of SwAV head.

参数 **pred** (*torch.Tensor*) –NxC input features.

返回 The SwAV loss.

返回类型 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.

参数

- **logits** (*torch.Tensor*) –The outputs from the decoder.
- **target** (*torch.Tensor*) –The targets generated by dalle.

返回 The main loss.

返回类型 Tuple[torch.Tensor, torch.Tensor]

class mmselfsup.models.losses.**CAELoss**(*lambd: float*)

Loss function for CAE.

Compute the align loss and the main loss.

参数 **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.

参数

- **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.

返回 The main loss and align loss.

返回类型 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.

参数

- **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.

参数

- **pred** (*torch.Tensor*) –The predicted features.
- **target** (*torch.Tensor*) –The target features.

返回 The cosine similarity loss.

返回类型 torch.Tensor

class mmselfsup.models.losses.**CrossCorrelationLoss** (*lambd: float = 0.0051*)

Cross correlation loss function.

Compute the on-diagnal and off-diagnal loss.

参数 **lambd** (*float*) –The weight for the off-diag loss.

forward (*cross_correlation_matrix: torch.Tensor*) → torch.Tensor

Forward function of cross correlation loss.

参数 **cross_correlation_matrix** (*torch.Tensor*) –The cross correlation matrix.

返回 cross correlation loss.

返回类型 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.

参数

- **pred** (*torch.Tensor*) –The reconstructed image.
- **target** (*torch.Tensor*) –The target image.
- **mask** (*torch.Tensor*) –The mask of the target image.

返回 The reconstruction loss.

返回类型 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.

参数

- **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.

参数

- **pred** (*torch.Tensor*) –The reconstructed image.
- **target** (*torch.Tensor*) –The target image.
- **mask** (*torch.Tensor*) –The mask of the target image.

返回 The reconstruction loss.

返回类型 torch.Tensor

class mmselfsup.models.losses.**SimMIMReconstructionLoss**(*encoder_in_channels: int*)

Loss function for MAE.

Compute the loss in masked region.

参数 **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.

参数

- **pred** (*torch.Tensor*) –The reconstructed image.
- **target** (*torch.Tensor*) –The target image.
- **mask** (*torch.Tensor*) –The mask of the target image.

返回 The reconstruction loss.

返回类型 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*.

参数

- **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.

参数 **x** (*torch.Tensor*) –NxC input features.

返回 The returned loss.

返回类型 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.

参数

- **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.

参数

- **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.

参数

- **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.

参数 **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.

参数 **x** (*torch.Tensor*) –The input image, which is of shape (N, 3, H, W).

返回 The features and attention from the last layer of CLIP, which are of shape (N, L, C) and (N, L, L), respectively.

返回类型 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.

注解: 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 slowfast/models/operators.py. Here is the link of HOG wikipedia.

参数

- **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.

参数 **x** (*torch.Tensor*) –Input images of shape (N, 3, H, W).

返回 Hog features.

返回类型 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

参数

- **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 normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **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`.

返回 Data in the same format as the model input.

返回类型 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.

参数

- **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.

参数

- **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.

参数

- **averaged_param** (*Tensor*) –The averaged parameters.
- **source_param** (*Tensor*) –The source parameters.
- **steps** (*int*) –The number of times the parameters have been updated.

返回 The averaged parameters.

返回类型 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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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.

参数

- **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*.

参数

- **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.

返回 Output features with shape N x L x C.

返回类型 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 normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **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`.

返回 Data in the same format as the model input.

返回类型 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 normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **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`.

返回 Data in the same format as the model input.

返回类型 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 normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **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`.

返回 Data in the same format as the model input.

返回类型 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.

参数

- **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.

参数

- **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 normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **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`.

返回

Data in the same format as the model input.

返回类型 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.

参数

- **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 normalization、padding and bgr2rgb conversion based on `BaseDataPreprocessor`.

参数

- **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`.

返回

Data in the same format as the model input.

返回类型 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.

参数

- **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.

返回 The position embedding vector.

返回类型 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.

参数

- **state_dict** (*dict*) –The pretrained state dict.
- **finetune** (*bool*) –Whether to fineturn the model.
- **average_targets** (*bool*) –Whether to average the target.

返回 The CLIP model.

返回类型 nn.Module

CHAPTER 38

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.

实际案例

```
>>> 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]
        ) 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])
```

(下页继续)

(续上页)

```
>>> 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 39

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.

参数

- **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.

实际案例

```
>>> 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,
...                        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.

参数

- **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 40

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/

参数 **probs** (*torch.Tensor*) –Sampling probabilities.

draw(*N: int*) → None

Draw N samples from multinomial.

参数 **N** (*int*) –Number of samples.

返回 Samples.

返回类型 torch.Tensor

mmselfsup.utils.**batch_shuffle_ddp**(*x: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Batch shuffle, for making use of BatchNorm.

参数 **x** (*torch.Tensor*) –Data in each GPU.

返回

Output of shuffle operation.

- x_gather[idx_this]: Shuffled data.
- idx_unshuffle: Index for restoring.

返回类型 Tuple[torch.Tensor, torch.Tensor]

mmselfsup.utils.**batch_unshuffle_ddp**(*x: torch.Tensor*, *idx_unshuffle: torch.Tensor*) → torch.Tensor

Undo batch shuffle.

参数

- **x** (*torch.Tensor*) –Data in each GPU.
- **idx_unshuffle** (*torch.Tensor*) –Index for restoring.

返回 Output of unshuffle operation.

返回类型 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.

参数 **tensor** (*torch.Tensor*) –Tensor to be broadcast from current process.

返回 The concatnated tensor.

返回类型 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.

参数

- **func** (*function*) –The function to process data.
- **data_loader** (*DataLoader*) –the torch DataLoader to yield data.
- **length** (*int*) –Expected length of output arrays.

返回 The collected outputs.

返回类型 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.

参数

- **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.

返回 Output of sinkhorn algorithm.

返回类型 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.

参数 **model** (*nn.Module*) –A model may be a model wrapper.

返回 The model without model wrapper.

返回类型 *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.

参数

- **func** (*function*) –The function to process data.
- **data_loader** (*DataLoader*) –the torch DataLoader to yield data.
- **length** (*int*) –Expected length of output arrays.

返回 The concatenated outputs.

返回类型 Dict[str, torch.Tensor]

mmselfsup.utils.**register_all_modules**(*init_default_scope: bool = True*) → None

Register all modules in mmselfsup into the registries.

参数 **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 41

更新日志

41.1 MMSelfSup

41.1.1 v1.0.0rc6 (10/02/2023)

`master` 仍然是 0.x 版本，我们将会 checkout 一个新的 `1.x` 用来发布 1.x 版本。未来我们会同时维护两个版本。

我们简要的列出主要的改变。请参考迁移文档来查看细节和迁移指引

亮点

- 在 `projects/maskfeat_video/` 支持了 `MaskFeat` 视频数据集的预训练
- 将部分文档翻译为中文

新特性

- 在 `projects/maskfeat_video/` 支持了 `MaskFeat` 视频数据集的预训练 (#678)

Bug 修复

- 修复 `shape bias` 分布式训练的问题 (#689)
- 更新 BEiT v2 的链接 (#676)
- 修复传参时未指定参数的问题 (#654)
- 更新 `default_runtime.py` 文件 (#681)
- 将 `metafile.yaml` 重命名为 `metafile.yml` (#680)
- 修复 `configs/selfsup/eva/metafile.yml` 中的问题 (#669)

改进

- 默认分支修改为 1.x (#686)
- 更新 pre-commit (#685)
- 暂停 python 3.6 的支持 (#657)

文档

- 翻译 add_transforms.md 和 conventions.md (#674)
- 翻译 classification.md, detection.md, segmentation.md (#665)
- 更新 KNN 脚本的链接 (#661)
- 翻译两篇文档 (#653)
- 翻译三篇文档 (#651)

41.1.2 v1.0.0rc5 (30/12/2022)

`master` 仍然是 0.x 版本，我们将会 checkout 一个新的 `1.x` 用来发布 1.x 版本。未来我们会同时维护两个版本。

我们简要的列出主要的改变。请参考迁移文档来查看细节和迁移指引

亮点

- 支持了 BEiT v2, MixMIM, EVA
- 支持了模型分析工具 ShapeBias
- 增加 FGIA ACCV 2022 第一名解决方案
- 重构 t-SNE

新特性

- 支持了 BEiT v2 (#627)
- 支持了 MixMIM (#626)
- 支持了 EVA (#632)
- 支持了 ShapeBias 评价方式 (#635)
- 增加模型转换脚本和指引 (#621)
- 增加 FGIA 比赛预训练方案 (#607)

Bug 修复

- 将 pseudo_collect 改为 default_collect (#616)
- 修复 SimMIM 链接问题 (#622)
- 修改 map_location 为 cpu (#623) 修复 import 问题 (#631)
- 修复配置文件字段问题 (#630)
- 修改 np.int 为 int (#636)
- 修复 knn 多卡 bug (#634)

改进

- 重构 projects/ 文件夹 (#620)
- 重构 t-SNE (#629)
- 基于 target_generator 重构 CAE (#645)
- 重构回归测试相关内容 (#637)

文档

- 更新 data_flow.md 文档 (#612)
- 更新 datasets.md 文档 (#633)

41.1.3 v1.0.0rc4 (07/12/2022)

`master` 仍然是 0.x 版本，我们将会 checkout 一个新的 `1.x` 用来发布 1.x 版本。未来我们会同时维护两个版本。

我们简要的列出主要的改变。请参考迁移文档来查看细节和迁移指引

Highlight

- 支持 `BEiT` 和 `MILAN`
- 支持 low-level 重建可视化

New Features

- 支持 `BEiT` (#425)
- 支持 `MILAN` (#600)
- 支持 low-level 重建可视化 (#570)

Bug Fixes

- 修复数据预处理潜在的注册问题 (#603)
- 修复代码依赖和字段错误 (#611)

Improvements

- 重构 file io (#582))
- 增加‘./projects’文件夹和示例 (#586))
- 更新 CI 和 UT (#601))

Docs

- 更新 readthedocs 和菜单栏 (#572)
- 增加 readthedocs 算法页面并修复部分渲染错误 (#599)

41.1.4 v1.0.0rc2 (12/10/2022)

`master` 仍然是 0.x 版本，我们将会 checkout 一个新的 `1.x` 用来发布 1.x 版本。未来我们会同时维护两个版本。

我们简要的列出主要的改变。请参考迁移文档来查看细节和迁移指引

亮点

- 全量支持 `MAE`, `SimMIM`, `MoCoV3`.

新特性

- 全量支持 `MAE` (#483)
- 全量支持 `SimMIM` (#487)
- 全量支持 of `MoCoV3` (#496)

修复 Bug

- 修复 classification configs (#488)
- 修复 MAE config 中名字问题 (#498)

改进

- 修改 colab 指引 (#470))
- 更新 readthedocs 要求 (#472)
- 更新 CI (#476)
- 优化 `mim_slurm_test.sh` 和 `mim_dist_test.sh` 为 benchmarks (#477)
- 更新 Metafile format 和 content (#478)

文档

- 添加 advanced_guides/engine.md (#454)
- 添加 advanced_guides/evaluation.md (#456)
- 添加 advanced_guides/transforms.md (#463)
- 添加 dataset docs (#437)
- 优化 contribution guide (#492)
- 更新 convention (#475)

41.1.5 v1.0.0rc1 (01/09/2022)

我们很高兴宣布发布 MMSelfSup v1.0.0rc1。We are excited to announce the release of MMSelfSup v1.0.0rc1. MMSelfSup v1.0.0rc1 是 MMSelfSup 1.x 的第一个版本，是 OpenMMLab 2.0 项目中的一部分。`master` 仍然是 0.x 版本，我们将会 checkout 一个新的 `1.x` 用来发布 1.x 版本。未来我们会同时维护两个版本。

我们简要的列出主要的改变。请参考迁移文档来查看细节和迁移指引

亮点

- 基于 MMEngine 和 MMCV.
- 发布重构.
  - Datasets
  - Models
  - Config
  - …
- 优化所有文档.

新特性

- 增加 `SelfSupDataSample` 来统一接口。
- 增加 `SelfSupVisualizer` 可视化功能。
- 增加 `SelfSupDataPreprocessor` 来进行模型的数据预处理。

改进

- 大部分方法都支持非分布式方法。
- 改变不同的数据增强的接口为 `dict`。
- 使用 MMCLassification 运行下游分类任务。

文档

- 优化所有文档和重新整理路径。
- 为不同组件增加新的概念。

41.2 MMSelfSup

41.2.1 v0.10.0 (30/09/2022)

亮点

- 支持 MaskFeat (#485)
- 更新 README 宣传 1.0.0rc 版本 (#474)

新特性

- 支持 MaskFeat (#485)

Bug 修复

- 修复 DenseCL 初始化的问题 (#411)
- 修复配置文件中归一化的错误 (#418)
- 修复读取图片的问题 (#386)

改进

- 更新 hook_cfg 获取方式 (#409)
- 支持输出配置文件 (#410)
- 支持保存 MAE 可视化结果 (#388)
- 删除废弃项的默认值 (#490)

文档

- 更新 MAE 配置文件链接 (#497)
- 更新 README 宣传 1.0.0rc 版本 (#474)
- 更新 get_started 文档 (#402)

41.2.2 v0.9.2 (28/07/2022)

新特性

- 支持 MAE 重建图像的可视化 (#376)

Bug 修复

- 修复 extract.py 文件中 cfg/args 路径问题，应用 cfg 中的路径进行处理 (#357)
- 修复 SimMIM 配置文件中掩码生成器类型名称的错误 (#360)

改进

- 更新 mdformat 设置 (#323)
- 添加 circle ci 配置 (#374)

文档

- 修复语言更换链接问题 (#327)
- 更新 tutorials/4_schedule.md 中的文档链接 (#354)

41.2.3 v0.9.1 (31/05/2022)

亮点

- 更新 **BYOL** 模型和结果 (#319)
- 改进部分文档

新特性

- 更新 **BYOL** 模型和结果 (#319)

Bug 修复

- 对于 CAE 和 MAE 设置 qkv 偏置参数 (#303)
- 修复 MAE 配置文件拼写错误 (#307)

改进

- 修改文件名 (#304)
- 应用 mdformat (#311)

文档

- 改正教程中的打字错误 (#308)
- 配置 Myst-parser (#309)
- 更新文档算法简介 (#310)
- 改进安装文档 (#317)
- 改进首页 README (#318)

41.2.4 v0.9.0 (29/04/2022)

亮点

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

新特性

- 支持 CAE (#284)
- 支持 Barlow twins (#207)
- 增加 SimMIM 192 预训练及 224 微调的结果 (#280)
- 增加 MAE fp16 预训练设置 (#271)

Bug 修复

- 修复参数问题 (#290)
- 在 MAE 配置中修改 imgs_per_gpu 为 samples_per_gpu (#278)
- 使用 prefetch dataloader 时避免 GPU 内存溢出 (#277)
- 修复在注册自定义钩子时键值错误的问题 (#273)

改进

- 更新 SimCLR 模型和结果 (#295)
- 单元测试减少内存使用 (#291)
- 去除 pytorch 1.5 测试 (#288)
- 重命名线性评估配置文件 (#281)
- 为 api 增加单元测试 (#276)

文档

- 在模型库增加 SimMIM 并修复链接 (#272)

41.2.5 v0.8.0 (31/03/2022)

亮点

- 支持 **SimMIM** (#239)
- 增加 **KNN** 基准测试，支持中间 checkpoint 和提取的 backbone 权重进行评估 (#243)
- 支持 ImageNet-21k 数据集 (#225)

新特性

- 支持 SimMIM (#239)
- 增加 KNN 基准测试，支持中间 checkpoint 和提取的 backbone 权重进行评估 (#243)
- 支持 ImageNet-21k 数据集 (#225)
- 支持自动继续 checkpoint 文件的训练 (#245)

Bug 修复

- 在分布式 sampler 中增加种子 (#250)
- 修复 dist_test_svm_epoch.sh 中参数位置问题 (#260)
- 修复 prepare_voc07_cls.sh 中 mkdir 潜在错误 (#261)

改进

- 更新命令行参数模式 (#253)

文档

- 修复 6_benchmarks.md 中命令文档 (#263)
- 翻译 6_benchmarks.md 到中文 (#262)

41.2.6 v0.7.0 (03/03/2022)

亮点

- 支持 MAE 算法 (#221)
- 增加 Places205 下游基准测试 (#210)
- 在 CI 工作流中添加 Windows 测试 (#215)

新特性

- 支持 MAE 算法 (#221)
- 增加 Places205 下游基准测试 (#210)

Bug 修复

- 修复部分配置文件中的错误 (#200)
- 修复图像读取通道问题并更新相关结果 (#210)
- 修复在使用 prefetch 时，部分 dataset 输出格式不匹配的问题 (#218)
- 修复 t-sne ‘no init_cfg’ 的错误 (#222)

改进

- 配置文件中弃用 `imgs_per_gpu`，改用 `samples_per_gpu` (#204)
- 更新 MMCV 的安装方式 (#208)
- 为算法 readme 和代码版权增加 pre-commit 钩子 (#213)
- 在 CI 工作流中添加 Windows 测试 (#215)

文档

- 将 0_config.md 翻译成中文 (#216)
- 更新主页 OpenMMLab 项目和介绍 (#219)

41.2.7 v0.6.0 (02/02/2022)

亮点

- 支持基于 vision transformer 的 MoCo v3 (#194)
- 加速训练和启动时间 (#181)
- 支持 cpu 训练 (#188)

新特性

- 支持基于 vision transformer 的 MoCo v3 (#194)
- 支持 cpu 训练 (#188)

Bug 修复

- 修复问题 (#159, #160) 中提到的相关 bugs (#161)
- 修复 `RandomAppliedTrans` 中缺失的 prob 赋值 (#173)
- 修复 k-means losses 显示的 bug (#182)
- 修复非分布式多 gpu 训练/测试中的 bug (#189)
- 修复加载 cifar 数据集时的 bug (#191)
- 修复 `dataset.evaluate` 的参数 bug (#192)

改进

- 取消之前在 CI 中未完成的运行 (#145)
- 增强 MIM 功能 (#152)
- 更改某些特定文件时跳过 CI (#154)
- 在构建 eval 优化器时添加 `drop_last` 选项 (#158)
- 弃用对“python setup.py test”的支持 (#174)
- 加速训练和启动时间 (#181)
- 升级 `isort` 到 5.10.1 (#184)

文档

- 重构文档目录结构 (#146)
- 修复 readthedocs (#148, #149, #153)
- 修复一些文档中的拼写错误和无效链接 (#155, #180, #195)
- 更新模型库里的训练日志和基准测试结果 (#157, #165, #195)
- 更新部分文档并翻译成中文 (#163, #164, #165, #166, #167, #168, #169, #172, #176, #178, #179)
- 更新算法 README 到新格式 (#177)

41.2.8 v0.5.0 (16/12/2021)

亮点

- 代码重构后发版。
- 添加 3 个新的自监督学习算法。
- 支持 MMDet 和 MMSeg 的基准测试。
- 添加全面的文档。

重构

- 合并冗余数据集文件。
- 适配新版 MMCV，去除旧版相关代码。
- 继承 MMCV BaseModule。
- 优化目录结构。

- 重命名所有配置文件。

新特性

- 添加 SwAV、SimSiam、DenseCL 算法。
- 添加 t-SNE 可视化工具。
- 支持 MMCV 版本 fp16。

基准

- 更多基准测试结果，包括分类、检测和分割。
- 支持下游任务中的一些新数据集。
- 使用 MIM 启动 MMDet 和 MMSeg 训练。

文档

- 重构 README、getting_started、install、model_zoo 文档。
- 添加数据准备文档。
- 添加全面的教程。

41.3 OpenSelfSup (历史)

41.3.1 v0.3.0 (14/10/2020)

亮点

- 支持混合精度训练。
- 改进 GaussianBlur 使训练速度加倍。
- 更多基准测试结果。

Bug 修复

- 修复 moco v2 中的 bugs，现在结果可复现。
- 修复 byol 中的 bugs。

新特性

- 混合精度训练。
- 改进 GaussianBlur 使 MoCo V2、SimCLR、BYOL 的训练速度加倍。
- 更多基准测试结果，包括 Places、VOC、COCO。

41.3.2 v0.2.0 (26/6/2020)

亮点

- 支持 BYOL。
- 支持半监督基准测试。

Bug 修复

- 修复 publish_model.py 中的哈希 id。

新特性

- 支持 BYOL。
- 在线性和半监督评估中将训练和测试脚本分开。
- 支持半监督基准测试：benchmarks/dist_train_semi.sh。
- 将基准测试相关的配置文件移动到 configs/benchmarks/。
- 提供基准测试结果和模型下载链接。
- 支持每隔几次迭代更新网络。
- 支持带有 Nesterov 的 LARS 优化器。
- 支持 SimCLR 和 BYOL 从 LARS 适应和权重衰减中排除特定参数的需求。

CHAPTER 42

FAQ

我们列出来一些用户常见的问题，并将他们的解决方案列出。您可以将一些您发现的常见的问题添加进列表中，来帮助其他用户解决问题。如果这里面的内容没有覆盖您的问题，请按照 模板 创建一个 issue，并确保您在模板中填写了所有要求的信息。

42.1 安装

MMCV, MMClassification, MMDetection and MMSegmentation 的版本兼容性如下所示。请安装正确的版本来避免安装问题。

Note:

- MMDetection 和 MMSegmentation 是可选的。
- 如果您仍然有版本错误，请创建一个 issue 并提供您的包的版本信息。

42.2 DeepCluster 在 A100 GPU

如果您想尝试 DeepCluster 在 A100 GPU 上，使用 pip 安装 `faiss` 将会引发错误，他在这里被提及过。

请使用 conda 安装：

```
conda install -c pytorch faiss-gpu cudatoolkit=11.3
```

同时您需要安装支持 CUDA11.3 的 PyTorch，同时 faiss-gpu==1.7.2 要求 python 3.6-3.8。

CHAPTER 43

English

CHAPTER 44

简体中文

CHAPTER 45

导引

- genindex
- modindex
- search

Python 模块索引

m

索引

A

B

C

D

E

F

M

N

O

P

R

S

T

U

V

W

模块