mmcv Documentation

Release 1.6.2

MMCV Contributors

Nov 17, 2022

GET STARTED

You can switch between Chinese and English documents in the lower-left corner of the layout.

CHAPTER

ONE

INTRODUCTION

MMCV is a foundational library for computer vision research and supports many research projects as below:

- MIM: MIM installs OpenMMLab packages.
- MMClassification: OpenMMLab image classification toolbox and benchmark.
- MMDetection: OpenMMLab detection toolbox and benchmark.
- MMDetection3D: OpenMMLab's next-generation platform for general 3D object detection.
- MMRotate: OpenMMLab rotated object detection toolbox and benchmark.
- MMSegmentation: OpenMMLab semantic segmentation toolbox and benchmark.
- MMOCR: OpenMMLab text detection, recognition, and understanding toolbox.
- MMPose: OpenMMLab pose estimation toolbox and benchmark.
- MMHuman3D: OpenMMLab 3D human parametric model toolbox and benchmark.
- MMSelfSup: OpenMMLab self-supervised learning toolbox and benchmark.
- MMRazor: OpenMMLab model compression toolbox and benchmark.
- MMFewShot: OpenMMLab fewshot learning toolbox and benchmark.
- MMAction2: OpenMMLab's next-generation action understanding toolbox and benchmark.
- MMTracking: OpenMMLab video perception toolbox and benchmark.
- MMFlow: OpenMMLab optical flow toolbox and benchmark.
- MMEditing: OpenMMLab image and video editing toolbox.
- MMGeneration: OpenMMLab image and video generative models toolbox.
- MMDeploy: OpenMMLab model deployment framework.

It provides the following functionalities.

- Universal IO APIs
- Image/Video processing
- Image and annotation visualization
- Useful utilities (progress bar, timer, ...)
- PyTorch runner with hooking mechanism
- Various CNN architectures
- High-quality implementation of common CUDA ops

It supports the following systems.

- Linux
- Windows
- macOS

Note: MMCV requires Python 3.6+.

CHAPTER

TWO

INSTALLATION

There are two versions of MMCV:

- **mmcv-full**: comprehensive, with full features and various CUDA ops out of box. It takes longer time to build.
- **mmcv**: lite, without CUDA ops but all other features, similar to mmcv<1.0.0. It is useful when you do not need those CUDA ops.

Warning: Do not install both versions in the same environment, otherwise you may encounter errors like `ModuleNotFound`. You need to uninstall one before installing the other. `Installing the full version is highly recommended if CUDA is avaliable`.

a. Install the full version.

Before installing mmcv-full, make sure that PyTorch has been successfully installed following the official guide.

We provide pre-built mmcv packages (recommended) with different PyTorch and CUDA versions to simplify the building for **Linux and Windows systems**. In addition, you can run check_installation.py to check the installation of mmcv-full after running the installation commands.

i. Install the latest version.

The rule for installing the latest `mmcv-full` is as follows:

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/{cu_version}/{torch_
↪version}/index.html
```

Please replace `{cu_version}` and `{torch_version}` in the url to your desired one. For example, to install the latest `mmcv-full` with `CUDA 11.1` and `PyTorch 1.9.0`, use the following command:

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu111/torch1.9.0/index.
↪html
```

For more details, please refer the the following tables and delete `=={mmcv_version}`.

ii. Install a specified version.

The rule for installing a specified `mmcv-full` is as follows:

```
pip install mmcv-full=={mmcv_version} -f https://download.openmmlab.com/mmcv/dist/{cu_
↪version}/{torch_version}/index.html
```

First of all, please refer to the Releases and replace `{mmcv_version}` a specified one. e.g. `1.3.9`. Then replace `{cu_version}` and `{torch_version}` in the url to your desired versions. For example, to install `mmcv-full==1.3.9` with `CUDA 11.1` and `PyTorch 1.9.0`, use the following command:

```
pip install mmcv-full==1.3.9 -f https://download.openmmlab.com/mmcv/dist/cu111/torch1.9.
↪0/index.html
```

Note: mmcv-full is only compiled on PyTorch 1.x.0 because the compatibility usually holds between 1.x.0 and 1.x.1. If your PyTorch version is 1.x.1, you can install mmcv-full compiled with PyTorch 1.x.0 and it usually works well. For example, if your PyTorch version is 1.8.1 and CUDA version is 11.1, you can use the following command to install mmcv-full.

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu111/torch1.8.0/index.
html
```

For more details, please refer the the following tables.

Note: The pre-built packages provided above do not include all versions of mmcv-full, you can click on the corresponding links to see the supported versions. For example, if you click cu102-torch1.8.0, you can see that `cu102-torch1.8.0` only provides 1.3.0 and above versions of mmcv-full. In addition, We no longer provide `mmcv-full` pre-built packages compiled with `PyTorch 1.3 & 1.4` since v1.3.17. You can find previous versions that compiled with PyTorch 1.3 & 1.4 here. The compatibility is still ensured in our CI, but we will discard the support of PyTorch 1.3 & 1.4 next year.

Note: mmcv-full does not provide pre-built packages for `cu102-torch1.11` and `cu92-torch*` on Windows.

Another way is to compile locally by running

```
pip install mmcv-full
```

Note that the local compiling may take up to 10 mins.

b. Install the lite version.

```
pip install mmcv
```

c. Install full version with custom operators for onnxruntime

- Check here for detailed instruction.

If you would like to build MMCV from source, please refer to the guide.

CHAPTER

THREE

BUILD MMCV FROM SOURCE

3.1 Build on Linux or macOS

After cloning the repo with

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

It is recommended to install `ninja` to speed up the compilation

```
pip install -r requirements/optional.txt
```

You can either

- install the lite version

  ```
  pip install -e .
  ```

- install the full version

  ```
  MMCV_WITH_OPS=1 pip install -e .
  ```

If you are on macOS, add the following environment variables before the installing command.

```
CC=clang CXX=clang++ CFLAGS='-stdlib=libc++'
```

e.g.,

```
CC=clang CXX=clang++ CFLAGS='-stdlib=libc++' MMCV_WITH_OPS=1 pip install -e .
```

Note: If you would like to use `opencv-python-headless` instead of `opencv-python`, e.g., in a minimum container environment or servers without GUI, you can first install it before installing MMCV to skip the installation of `opencv-python`.

3.2 Build on Windows

Building MMCV on Windows is a bit more complicated than that on Linux. The following instructions show how to get this accomplished.

3.2.1 Prerequisite

The following software is required for building MMCV on windows. Install them first.

- Git
  - During installation, tick **add git to Path**.
- Visual Studio Community 2019
  - A compiler for C++ and CUDA codes.
- Miniconda
  - Official distributions of Python should work too.
- CUDA 10.2
  - Not required for building CPU version.
  - Customize the installation if necessary. As a recommendation, skip the driver installation if a newer version is already installed.

Note: You should know how to set up environment variables, especially `Path`, on Windows. The following instruction relies heavily on this skill.

3.2.2 Setup Python Environment

1. Launch Anaconda prompt from Windows Start menu

   Do not use raw `cmd.exe` s instruction is based on PowerShell syntax.

2. Create a new conda environment

```
conda create --name mmcv python=3.7  # 3.6, 3.7, 3.8 should work too as tested
conda activate mmcv  # make sure to activate environment before any operation
```

3. Install PyTorch. Choose a version based on your need.

```
conda install pytorch torchvision cudatoolkit=10.2 -c pytorch
```

   We only tested PyTorch version >= 1.6.0.

4. Prepare MMCV source code

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

5. Install required Python packages

```
pip3 install -r requirements/runtime.txt
```

6. It is recommended to install `ninja` to speed up the compilation

```
pip install -r requirements/optional.txt
```

3.2.3 Build and install MMCV

MMCV can be built in three ways:

1. Lite version (without ops)

   In this way, no custom ops are compiled and mmcv is a pure python package.

2. Full version (CPU ops)

   Module `ops` will be compiled as a pytorch extension, but only x86 code will be compiled. The compiled ops can be executed on CPU only.

3. Full version (CUDA ops)

   Both x86 and CUDA codes of `ops` module will be compiled. The compiled version can be run on both CPU and CUDA-enabled GPU (if implemented).

Common steps

1. Set up MSVC compiler

   Set Environment variable, add `C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.27.29110\bin\Hostx86\x64` to `PATH`, so that `cl.exe` will be available in prompt, as shown below.

```
(base) PS C:\Users\xxx> cl
Microsoft (R) C/C++ Optimizing  Compiler Version 19.27.29111 for x64
Copyright (C) Microsoft Corporation.   All rights reserved.

usage: cl [ option... ] filename... [ / link linkoption... ]
```

   For compatibility, we use the x86-hosted and x64-targeted compiler. note `Hostx86\x64` in the path.

   You may want to change the system language to English because pytorch will parse text output from `cl.exe` to check its version. However only utf-8 is recognized. Navigate to Control Panel -> Region -> Administrative -> Language for Non-Unicode programs and change it to English.

Option 1: Build MMCV (lite version)

After finishing above common steps, launch Anaconda shell from Start menu and issue the following commands:

```
# activate environment
conda activate mmcv
# change directory
cd mmcv
# install
python setup.py develop
# check
pip list
```

Option 2: Build MMCV (full version with CPU)

1. Finish above common steps
2. Set up environment variables

```
$env:MMCV_WITH_OPS = 1
$env:MAX_JOBS = 8  # based on your available number of CPU cores and amount of␣
↪memory
```

3. Following build steps of the lite version

```
# activate environment
conda activate mmcv
# change directory
cd mmcv
# build
python setup.py build_ext # if success, cl will be launched to compile ops
# install
python setup.py develop
# check
pip list
```

Option 3: Build MMCV (full version with CUDA)

1. Finish above common steps
2. Make sure CUDA_PATH or CUDA_HOME is already set in envs via ls env:, desired output is shown as below:

```
(base) PS C:\Users\WRH> ls env:

Name                           Value
----                           -----
<... omit some lines ...>
CUDA_PATH                      C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\
↪v10.2
CUDA_PATH_V10_1                C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\
↪v10.1
CUDA_PATH_V10_2                C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\
↪v10.2
<... omit some lines ...>
```

   This should already be done by CUDA installer. If not, or you have multiple version of CUDA toolkit installed, set it with

```
$env:CUDA_HOME = "C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.2"
# OR
$env:CUDA_HOME = $env:CUDA_PATH_V10_2 # if CUDA_PATH_V10_2 is in envs:
```

3. Set CUDA target arch

```
# Suppose you are using GTX 1080, which is of capability 6.1
$env:TORCH_CUDA_ARCH_LIST="6.1"
```

(continues on next page)

(continued from previous page)

```
# OR build all supported arch, will be slow
$env:TORCH_CUDA_ARCH_LIST="3.5 3.7 5.0 5.2 6.0 6.1 7.0 7.5"
```

Note: Check your the compute capability of your GPU from here.

4. Launch compiling the same way as CPU

```
$env:MMCV_WITH_OPS = 1
$env:MAX_JOBS = 8  # based on available number of CPU cores and amount of memory
# activate environment
conda activate mmcv
# change directory
cd mmcv
# build
python setup.py build_ext # if success, cl will be launched to compile ops
# install
python setup.py develop
# check
pip list
```

Note: If you are compiling against PyTorch 1.6.0, you might meet some errors from PyTorch as described in this issue. Follow this pull request to modify the source code in your local PyTorch installation.

If you meet issues when running or compiling mmcv, we list some common issues in Frequently Asked Question.

CHAPTER

FOUR

[OPTIONAL] BUILD MMCV ON IPU MACHINE

Firstly, you need to apply for an IPU cloud machine, see here.

4.1 Option 1: Docker

1. Pull docker

```
docker pull graphcore/pytorch
```

2. Build MMCV under same python environment

4.2 Option 2: Install from SDK

1. Build MMCV
2. Use pip to install sdk according to IPU PyTorch document. Also, you need to apply for machine and sdk to Graphcore.

CHAPTER

FIVE

CONFIG

`Config` class is used for manipulating config and config files. It supports loading configs from multiple file formats including **python**, **json** and **yaml**. It provides dict-like apis to get and set values.

Here is an example of the config file `test.py`.

```
a = 1
b = dict(b1=[0, 1, 2], b2=None)
c = (1, 2)
d = 'string'
```

To load and use configs

```
>>> cfg = Config.fromfile('test.py')
>>> print(cfg)
>>> dict(a=1,
...      b=dict(b1=[0, 1, 2], b2=None),
...      c=(1, 2),
...      d='string')
```

For all format configs, some predefined variables are supported. It will convert the variable in `{{ var }}` with its real value.

Currently, it supports four predefined variables:

`{{ fileDirname }}` - the current opened file's dirname, e.g. /home/your-username/your-project/folder

`{{ fileBasename }}` - the current opened file's basename, e.g. file.ext

`{{ fileBasenameNoExtension }}` - the current opened file's basename with no file extension, e.g. file

`{{ fileExtname }}` - the current opened file's extension, e.g. .ext

These variable names are referred from VS Code.

Here is one examples of config with predefined variables.

`config_a.py`

```
a = 1
b = './work_dir/{{ fileBasenameNoExtension }}'
c = '{{ fileExtname }}'
```

```
>>> cfg = Config.fromfile('./config_a.py')
>>> print(cfg)
>>> dict(a=1,
```

(continues on next page)

(continued from previous page)

```
...       b='./work_dir/config_a',
...       c='.py')
```

For all format configs, inheritance is supported. To reuse fields in other config files, specify `_base_='./config_a.py'` or a list of configs `_base_=['./config_a.py', './config_b.py']`. Here are 4 examples of config inheritance.

config_a.py

```
a = 1
b = dict(b1=[0, 1, 2], b2=None)
```

5.1 Inherit from base config without overlapped keys

config_b.py

```
_base_ = './config_a.py'
c = (1, 2)
d = 'string'
```

```
>>> cfg = Config.fromfile('./config_b.py')
>>> print(cfg)
>>> dict(a=1,
...       b=dict(b1=[0, 1, 2], b2=None),
...       c=(1, 2),
...       d='string')
```

New fields in `config_b.py` are combined with old fields in `config_a.py`

5.2 Inherit from base config with overlapped keys

config_c.py

```
_base_ = './config_a.py'
b = dict(b2=1)
c = (1, 2)
```

```
>>> cfg = Config.fromfile('./config_c.py')
>>> print(cfg)
>>> dict(a=1,
...       b=dict(b1=[0, 1, 2], b2=1),
...       c=(1, 2))
```

`b.b2=None` in `config_a` is replaced with `b.b2=1` in `config_c.py`.

5.3 Inherit from base config with ignored fields

config_d.py

```
_base_ = './config_a.py'
b = dict(_delete_=True, b2=None, b3=0.1)
c = (1, 2)
```

```
>>> cfg = Config.fromfile('./config_d.py')
>>> print(cfg)
>>> dict(a=1,
...      b=dict(b2=None, b3=0.1),
...      c=(1, 2))
```

You may also set `_delete_=True` to ignore some fields in base configs. All old keys `b1, b2, b3` in `b` are replaced with new keys `b2, b3`.

5.4 Inherit from multiple base configs (the base configs should not contain the same keys)

config_e.py

```
c = (1, 2)
d = 'string'
```

config_f.py

```
_base_ = ['./config_a.py', './config_e.py']
```

```
>>> cfg = Config.fromfile('./config_f.py')
>>> print(cfg)
>>> dict(a=1,
...      b=dict(b1=[0, 1, 2], b2=None),
...      c=(1, 2),
...      d='string')
```

5.5 Reference variables from base

You can reference variables defined in base using the following grammar.

base.py

```
item1 = 'a'
item2 = dict(item3 = 'b')
```

config_g.py

```
_base_ = ['./base.py']
item = dict(a = {{ _base_.item1 }}, b = {{ _base_.item2.item3 }})
```

```
>>> cfg = Config.fromfile('./config_g.py')
>>> print(cfg.pretty_text)
item1 = 'a'
item2 = dict(item3='b')
item = dict(a='a', b='b')
```

5.6 Add deprecation information in configs

Deprecation information can be added in a config file, which will trigger a `UserWarning` when this config file is loaded.

deprecated_cfg.py

```
_base_ = 'expected_cfg.py'

_deprecation_ = dict(
    expected = 'expected_cfg.py',  # optional to show expected config path in the␣
↪warning information
    reference = 'url to related PR'  # optional to show reference link in the warning␣
↪information
)
```

```
>>> cfg = Config.fromfile('./deprecated_cfg.py')

UserWarning: The config file deprecated_cfg.py will be deprecated in the future. Please␣
↪use expected_cfg.py instead. More information can be found at https://github.com/open-
↪mmlab/mmcv/pull/1275
```

CHAPTER

SIX

REGISTRY

MMCV implements registry to manage different modules that share similar functionalities, e.g., backbones, head, and necks, in detectors. Most projects in OpenMMLab use registry to manage modules of datasets and models, such as MMDetection, MMDetection3D, MMClassification, MMEditing, etc.

Note: In v1.5.1 and later, the Registry supports registering functions and calling them.

6.1 What is registry

In MMCV, registry can be regarded as a mapping that maps a class or function to a string. These classes or functions contained by a single registry usually have similar APIs but implement different algorithms or support different datasets. With the registry, users can find the class or function through its corresponding string, and instantiate the corresponding module or call the function to obtain the result according to needs. One typical example is the config systems in most OpenMMLab projects, which use the registry to create hooks, runners, models, and datasets, through configs. The API reference could be found here.

To manage your modules in the codebase by `Registry`, there are three steps as below.

1. Create a build method (optional, in most cases you can just use the default one).
2. Create a registry.
3. Use this registry to manage the modules.

`build_func` argument of `Registry` is to customize how to instantiate the class instance or how to call the function to obtain the result, the default one is `build_from_cfg` implemented here.

6.2 A Simple Example

Here we show a simple example of using registry to manage modules in a package. You can find more practical examples in OpenMMLab projects.

Assuming we want to implement a series of Dataset Converter for converting different formats of data to the expected data format. We create a directory as a package named `converters`. In the package, we first create a file to implement builders, named `converters/builder.py`, as below

```
from mmcv.utils import Registry
# create a registry for converters
CONVERTERS = Registry('converters')
```

Then we can implement different converters that is class or function in the package. For example, implement `Converter1` in `converters/converter1.py`, and `converter2` in `converters/converter2.py`.

```
from .builder import CONVERTERS

# use the registry to manage the module
@CONVERTERS.register_module()
class Converter1(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

```
# converter2.py
from .builder import CONVERTERS
from .converter1 import Converter1

#
@CONVERTERS.register_module()
def converter2(a, b)
    return Converter1(a, b)
```

The key step to use registry for managing the modules is to register the implemented module into the registry `CONVERTERS` through `@CONVERTERS.register_module()` when you are creating the module. By this way, a mapping between a string and the class (function) is built and maintained by `CONVERTERS` as below

```
'Converter1' -> <class 'Converter1'>
'converter2' -> <function 'converter2'>
```

Note: The registry mechanism will be triggered only when the file where the module is located is imported. So you need to import that file somewhere. More details can be found at https://github.com/open-mmlab/mmdetection/issues/5974.

If the module is successfully registered, you can use this converter through configs as

```
converter1_cfg = dict(type='Converter1', a=a_value, b=b_value)
converter2_cfg = dict(type='converter2', a=a_value, b=b_value)
converter1 = CONVERTERS.build(converter1_cfg)
# returns the calling result
result = CONVERTERS.build(converter2_cfg)
```

6.3 Customize Build Function

Suppose we would like to customize how `converters` are built, we could implement a customized `build_func` and pass it into the registry.

```
from mmcv.utils import Registry

# create a build function
def build_converter(cfg, registry, *args, **kwargs):
```

(continues on next page)

(continued from previous page)

```
    cfg_ = cfg.copy()
    converter_type = cfg_.pop('type')
    if converter_type not in registry:
        raise KeyError(f'Unrecognized converter type {converter_type}')
    else:
        converter_cls = registry.get(converter_type)

    converter = converter_cls(*args, **kwargs, **cfg_)
    return converter

# create a registry for converters and pass ``build_converter`` function
CONVERTERS = Registry('converter', build_func=build_converter)
```

Note: In this example, we demonstrate how to use the `build_func` argument to customize the way to build a class instance. The functionality is similar to the default `build_from_cfg`. In most cases, default one would be sufficient. `build_model_from_cfg` is also implemented to build PyTorch module in `nn.Sequential`, you may directly use them instead of implementing by yourself.

6.4 Hierarchy Registry

You could also build modules from more than one OpenMMLab frameworks, e.g. you could use all backbones in MMClassification for object detectors in MMDetection, you may also combine an object detection model in MMDetection and semantic segmentation model in MMSegmentation.

All `MODELS` registries of downstream codebases are children registries of MMCV's `MODELS` registry. Basically, there are two ways to build a module from child or sibling registries.

1. Build from children registries.

   For example:

   In MMDetection we define:

   ```
   from mmcv.utils import Registry
   from mmcv.cnn import MODELS as MMCV_MODELS
   MODELS = Registry('model', parent=MMCV_MODELS)

   @MODELS.register_module()
   class NetA(nn.Module):
       def forward(self, x):
           return x
   ```

   In MMClassification we define:

   ```
   from mmcv.utils import Registry
   from mmcv.cnn import MODELS as MMCV_MODELS
   MODELS = Registry('model', parent=MMCV_MODELS)

   @MODELS.register_module()
   class NetB(nn.Module):
   ```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        return x + 1
```

We could build two net in either MMDetection or MMClassification by:

```
from mmdet.models import MODELS
net_a = MODELS.build(cfg=dict(type='NetA'))
net_b = MODELS.build(cfg=dict(type='mmcls.NetB'))
```

or

```
from mmcls.models import MODELS
net_a = MODELS.build(cfg=dict(type='mmdet.NetA'))
net_b = MODELS.build(cfg=dict(type='NetB'))
```

2. Build from parent registry.

   The shared `MODELS` registry in MMCV is the parent registry for all downstream codebases (root registry):

```
from mmcv.cnn import MODELS as MMCV_MODELS
net_a = MMCV_MODELS.build(cfg=dict(type='mmdet.NetA'))
net_b = MMCV_MODELS.build(cfg=dict(type='mmcls.NetB'))
```

CHAPTER

SEVEN

RUNNER

The runner class is designed to manage the training. It eases the training process with less code demanded from users while staying flexible and configurable. The main features are as listed:

- Support `EpochBasedRunner` and `IterBasedRunner` for different scenarios. Implementing customized runners is also allowed to meet customized needs.
- Support customized workflow to allow switching between different modes while training. Currently, supported modes are train and val.
- Enable extensibility through various hooks, including hooks defined in MMCV and customized ones.

7.1 EpochBasedRunner

As its name indicates, workflow in `EpochBasedRunner` should be set based on epochs. For example, [('train', 2), ('val', 1)] means running 2 epochs for training and 1 epoch for validation, iteratively. And each epoch may contain multiple iterations. Currently, MMDetection uses `EpochBasedRunner` by default.

Let's take a look at its core logic:

```
# the condition to stop training
while curr_epoch < max_epochs:
    # traverse the workflow.
    # e.g. workflow = [('train', 2), ('val', 1)]
    for i, flow in enumerate(workflow):
        # mode(e.g. train) determines which function to run
        mode, epochs = flow
        # epoch_runner will be either self.train() or self.val()
        epoch_runner = getattr(self, mode)
        # execute the corresponding function
        for _ in range(epochs):
            epoch_runner(data_loaders[i], **kwargs)
```

Currently, we support 2 modes: train and val. Let's take a train function for example and have a look at its core logic:

```
# Currently, epoch_runner could be either train or val
def train(self, data_loader, **kwargs):
    # traverse the dataset and get batch data for 1 epoch
    for i, data_batch in enumerate(data_loader):
        # it will execute all before_train_iter function in the hooks registered. You␣
↪may want to watch out for the order.
        self.call_hook('before_train_iter')
```

(continues on next page)

(continued from previous page)

```
        # set train_mode as False in val function
        self.run_iter(data_batch, train_mode=True, **kwargs)
        self.call_hook('after_train_iter')
    self.call_hook('after_train_epoch')
```

7.2 IterBasedRunner

Different from `EpochBasedRunner`, workflow in `IterBasedRunner` should be set based on iterations. For example, [('train', 2), ('val', 1)] means running 2 iters for training and 1 iter for validation, iteratively. Currently, MMSegmentation uses `IterBasedRunner` by default.

Let's take a look at its core logic:

```
# Although we set workflow by iters here, we might also need info on the epochs in some␣
↪using cases. That can be provided by IterLoader.
iter_loaders = [IterLoader(x) for x in data_loaders]
# the condition to stop training
while curr_iter < max_iters:
    # traverse the workflow.
    # e.g. workflow = [('train', 2), ('val', 1)]
    for i, flow in enumerate(workflow):
        # mode(e.g. train) determines which function to run
        mode, iters = flow
        # iter_runner will be either self.train() or self.val()
        iter_runner = getattr(self, mode)
        # execute the corresponding function
        for _ in range(iters):
            iter_runner(iter_loaders[i], **kwargs)
```

Currently, we support 2 modes: train and val. Let's take a val function for example and have a look at its core logic:

```
# Currently, iter_runner could be either train or val
def val(self, data_loader, **kwargs):
    # get batch data for 1 iter
    data_batch = next(data_loader)
    # it will execute all before_val_iter function in the hooks registered. You may want␣
↪to watch out for the order.
    self.call_hook('before_val_iter')
    outputs = self.model.val_step(data_batch, self.optimizer, **kwargs)
    self.outputs = outputs
    self.call_hook('after_val_iter')
```

Other than the basic functionalities explained above, `EpochBasedRunner` and `IterBasedRunner` provide methods such as `resume`, `save_checkpoint` and `register_hook`. In case you are not familiar with the term Hook mentioned earlier, we will also provide a tutorial about it.(coming soon...) Essentially, a hook is functionality to alter or augment the code behaviors through predefined api. It allows users to have their own code called under certain circumstances. It makes code extensible in a non-intrusive manner.

7.3 A Simple Example

We will walk you through the usage of runner with a classification task. The following code only contains essential steps for demonstration purposes. The following steps are necessary for any training tasks.

(1) Initialize dataloader, model, optimizer, etc.

```
# initialize model
model=...
# initialize optimizer, typically, we set: cfg.optimizer = dict(type='SGD', lr=0.1,␣
↪momentum=0.9, weight_decay=0.0001)
optimizer = build_optimizer(model, cfg.optimizer)
# initialize the dataloader corresponding to the workflow(train/val)
data_loaders = [
        build_dataloader(
            ds,
            cfg.data.samples_per_gpu,
            cfg.data.workers_per_gpu,
            ...) for ds in dataset
    ]
```

(2) Initialize runner

```
runner = build_runner(
    # cfg.runner is typically set as:
    # runner = dict(type='EpochBasedRunner', max_epochs=200)
    cfg.runner,
    default_args=dict(
        model=model,
        batch_processor=None,
        optimizer=optimizer,
        logger=logger))
```

(3) Register training hooks and customized hooks.

```
# register default hooks necessary for training
runner.register_training_hooks(
    # configs of learning rate, it is typically set as:
    # lr_config = dict(policy='step', step=[100, 150])
    cfg.lr_config,
    # configuration of optimizer, e.g. grad_clip
    optimizer_config,
    # configuration of saving checkpoints, it is typically set as:
    # checkpoint_config = dict(interval=1), saving checkpoints every epochs
    cfg.checkpoint_config,
    # configuration of logs
    cfg.log_config,
    ...)

# register customized hooks
# say we want to enable ema, then we could set custom_hooks=[dict(type='EMAHook')]
if cfg.get('custom_hooks', None):
    custom_hooks = cfg.custom_hooks
    for hook_cfg in cfg.custom_hooks:
```

(continues on next page)

(continued from previous page)

```
        hook_cfg = hook_cfg.copy()
        priority = hook_cfg.pop('priority', 'NORMAL')
        hook = build_from_cfg(hook_cfg, HOOKS)
        runner.register_hook(hook, priority=priority)
```

Then, we can use `resume` or `load_checkpoint` to load existing weights.

(4) Start training

```
# workflow is typically set as: workflow = [('train', 1)]
# here the training begins.
runner.run(data_loaders, cfg.workflow)
```

Let's take `EpochBasedRunner` for example and go a little bit into details about setting workflow:

- Say we only want to put train in the workflow, then we can set: workflow = [('train', 1)]. The runner will only execute train iteratively in this case.
- Say we want to put both train and val in the workflow, then we can set: workflow = [('train', 3), ('val',1)]. The runner will first execute train for 3 epochs and then switch to val mode and execute val for 1 epoch. The workflow will be repeated until the current epoch hit the max_epochs.
- Workflow is highly flexible. Therefore, you can set workflow = [('val', 1), ('train',1)] if you would like the runner to validate first and train after.

The code we demonstrated above is already in `train.py` in MM repositories. Simply modify the corresponding keys in the configuration files and the script will execute the expected workflow automatically.

CHAPTER

EIGHT

FILE IO

This module provides two universal API to load and dump files of different formats.

Note: Since v1.3.16, the IO modules support loading (dumping) data from (to) different backends, respectively. More details are in PR #1330.

8.1 Load and dump data

mmcv provides a universal api for loading and dumping data, currently supported formats are json, yaml and pickle.

8.1.1 Load from disk or dump to disk

```
import mmcv

# load data from a file
data = mmcv.load('test.json')
data = mmcv.load('test.yaml')
data = mmcv.load('test.pkl')
# load data from a file-like object
with open('test.json', 'r') as f:
    data = mmcv.load(f, file_format='json')

# dump data to a string
json_str = mmcv.dump(data, file_format='json')

# dump data to a file with a filename (infer format from file extension)
mmcv.dump(data, 'out.pkl')

# dump data to a file with a file-like object
with open('test.yaml', 'w') as f:
    data = mmcv.dump(data, f, file_format='yaml')
```

8.1.2 Load from other backends or dump to other backends

```
import mmcv

# load data from a file
data = mmcv.load('s3://bucket-name/test.json')
data = mmcv.load('s3://bucket-name/test.yaml')
data = mmcv.load('s3://bucket-name/test.pkl')

# dump data to a file with a filename (infer format from file extension)
mmcv.dump(data, 's3://bucket-name/out.pkl')
```

It is also very convenient to extend the api to support more file formats. All you need to do is to write a file handler inherited from `BaseFileHandler` and register it with one or several file formats.

You need to implement at least 3 methods.

```
import mmcv

# To register multiple file formats, a list can be used as the argument.
# @mmcv.register_handler(['txt', 'log'])
@mmcv.register_handler('txt')
class TxtHandler1(mmcv.BaseFileHandler):

    def load_from_fileobj(self, file):
        return file.read()

    def dump_to_fileobj(self, obj, file):
        file.write(str(obj))

    def dump_to_str(self, obj, **kwargs):
        return str(obj)
```

Here is an example of `PickleHandler`.

```
import pickle

class PickleHandler(mmcv.BaseFileHandler):

    def load_from_fileobj(self, file, **kwargs):
        return pickle.load(file, **kwargs)

    def load_from_path(self, filepath, **kwargs):
        return super(PickleHandler, self).load_from_path(
            filepath, mode='rb', **kwargs)

    def dump_to_str(self, obj, **kwargs):
        kwargs.setdefault('protocol', 2)
        return pickle.dumps(obj, **kwargs)

    def dump_to_fileobj(self, obj, file, **kwargs):
        kwargs.setdefault('protocol', 2)
        pickle.dump(obj, file, **kwargs)
```

(continues on next page)

(continued from previous page)

```
    def dump_to_path(self, obj, filepath, **kwargs):
        super(PickleHandler, self).dump_to_path(
            obj, filepath, mode='wb', **kwargs)
```

8.2 Load a text file as a list or dict

For example `a.txt` is a text file with 5 lines.

```
a
b
c
d
e
```

8.2.1 Load from disk

Use `list_from_file` to load the list from a.txt.

```
>>> mmcv.list_from_file('a.txt')
['a', 'b', 'c', 'd', 'e']
>>> mmcv.list_from_file('a.txt', offset=2)
['c', 'd', 'e']
>>> mmcv.list_from_file('a.txt', max_num=2)
['a', 'b']
>>> mmcv.list_from_file('a.txt', prefix='/mnt/')
['/mnt/a', '/mnt/b', '/mnt/c', '/mnt/d', '/mnt/e']
```

For example `b.txt` is a text file with 3 lines.

```
1 cat
2 dog cow
3 panda
```

Then use `dict_from_file` to load the dict from `b.txt`.

```
>>> mmcv.dict_from_file('b.txt')
{'1': 'cat', '2': ['dog', 'cow'], '3': 'panda'}
>>> mmcv.dict_from_file('b.txt', key_type=int)
{1: 'cat', 2: ['dog', 'cow'], 3: 'panda'}
```

8.2.2 Load from other backends

Use `list_from_file` to load the list from `s3://bucket-name/a.txt`.

```
>>> mmcv.list_from_file('s3://bucket-name/a.txt')
['a', 'b', 'c', 'd', 'e']
>>> mmcv.list_from_file('s3://bucket-name/a.txt', offset=2)
['c', 'd', 'e']
>>> mmcv.list_from_file('s3://bucket-name/a.txt', max_num=2)
['a', 'b']
>>> mmcv.list_from_file('s3://bucket-name/a.txt', prefix='/mnt/')
['/mnt/a', '/mnt/b', '/mnt/c', '/mnt/d', '/mnt/e']
```

Use `dict_from_file` to load the dict from `s3://bucket-name/b.txt`.

```
>>> mmcv.dict_from_file('s3://bucket-name/b.txt')
{'1': 'cat', '2': ['dog', 'cow'], '3': 'panda'}
>>> mmcv.dict_from_file('s3://bucket-name/b.txt', key_type=int)
{1: 'cat', 2: ['dog', 'cow'], 3: 'panda'}
```

8.3 Load and dump checkpoints

8.3.1 Load checkpoints from disk or save to disk

We can read the checkpoints from disk or save to disk in the following way.

```
import torch

filepath1 = '/path/of/your/checkpoint1.pth'
filepath2 = '/path/of/your/checkpoint2.pth'
# read from filepath1
checkpoint = torch.load(filepath1)
# save to filepath2
torch.save(checkpoint, filepath2)
```

MMCV provides many backends. `HardDiskBackend` is one of them and we can use it to read or save checkpoints.

```
import io
from mmcv.fileio.file_client import HardDiskBackend

disk_backend = HardDiskBackend()
with io.BytesIO(disk_backend.get(filepath1)) as buffer:
    checkpoint = torch.load(buffer)
with io.BytesIO() as buffer:
    torch.save(checkpoint, buffer)
    disk_backend.put(buffer.getvalue(), filepath2)
```

If we want to implement an interface which automatically select the corresponding backend based on the file path, we can use the `FileClient`. For example, we want to implement two methods for reading checkpoints as well as saving checkpoints, which need to support different types of file paths, either disk paths, network paths or other paths.

```
from mmcv.fileio.file_client import FileClient

def load_checkpoint(path):
    file_client = FileClient.infer(uri=path)
    with io.BytesIO(file_client.get(path)) as buffer:
        checkpoint = torch.load(buffer)
    return checkpoint

def save_checkpoint(checkpoint, path):
    with io.BytesIO() as buffer:
        torch.save(checkpoint, buffer)
        file_client.put(buffer.getvalue(), path)

file_client = FileClient.infer_client(uri=filepath1)
checkpoint = load_checkpoint(filepath1)
save_checkpoint(checkpoint, filepath2)
```

8.3.2 Load checkpoints from the Internet

Note: Currently, it only supports reading checkpoints from the Internet, and does not support saving checkpoints to the Internet.

```
import io
import torch
from mmcv.fileio.file_client import HTTPBackend, FileClient

filepath = 'http://path/of/your/checkpoint.pth'
checkpoint = torch.utils.model_zoo.load_url(filepath)

http_backend = HTTPBackend()
with io.BytesIO(http_backend.get(filepath)) as buffer:
    checkpoint = torch.load(buffer)

file_client = FileClient.infer_client(uri=filepath)
with io.BytesIO(file_client.get(filepath)) as buffer:
    checkpoint = torch.load(buffer)
```

CHAPTER

NINE

DATA PROCESS

9.1 Image

This module provides some image processing methods, which requires opencv to be installed.

9.1.1 Read/Write/Show

To read or write images files, use `imread` or `imwrite`.

```
import mmcv

img = mmcv.imread('test.jpg')
img = mmcv.imread('test.jpg', flag='grayscale')
img_ = mmcv.imread(img)  # nothing will happen, img_ = img
mmcv.imwrite(img, 'out.jpg')
```

To read images from bytes

```
with open('test.jpg', 'rb') as f:
    data = f.read()
img = mmcv.imfrombytes(data)
```

To show an image file or a loaded image

```
mmcv.imshow('tests/data/color.jpg')
# this is equivalent to

for i in range(10):
    img = np.random.randint(256, size=(100, 100, 3), dtype=np.uint8)
    mmcv.imshow(img, win_name='test image', wait_time=200)
```

9.1.2 Color space conversion

Supported conversion methods:

- bgr2gray
- gray2bgr
- bgr2rgb
- rgb2bgr
- bgr2hsv
- hsv2bgr

```
img = mmcv.imread('tests/data/color.jpg')
img1 = mmcv.bgr2rgb(img)
img2 = mmcv.rgb2gray(img1)
img3 = mmcv.bgr2hsv(img)
```

9.1.3 Resize

There are three resize methods. All `imresize_*` methods have an argument `return_scale`, if this argument is `False`, then the return value is merely the resized image, otherwise is a tuple `(resized_img, scale)`.

```
# resize to a given size
mmcv.imresize(img, (1000, 600), return_scale=True)

# resize to the same size of another image
mmcv.imresize_like(img, dst_img, return_scale=False)

# resize by a ratio
mmcv.imrescale(img, 0.5)

# resize so that the max edge no longer than 1000, short edge no longer than 800
# without changing the aspect ratio
mmcv.imrescale(img, (1000, 800))
```

9.1.4 Rotate

To rotate an image by some angle, use `imrotate`. The center can be specified, which is the center of original image by default. There are two modes of rotating, one is to keep the image size unchanged so that some parts of the image will be cropped after rotating, the other is to extend the image size to fit the rotated image.

```
img = mmcv.imread('tests/data/color.jpg')

# rotate the image clockwise by 30 degrees.
img_ = mmcv.imrotate(img, 30)

# rotate the image counterclockwise by 90 degrees.
img_ = mmcv.imrotate(img, -90)

# rotate the image clockwise by 30 degrees, and rescale it by 1.5x at the same time.
```

(continues on next page)

(continued from previous page)

```
img_ = mmcv.imrotate(img, 30, scale=1.5)

# rotate the image clockwise by 30 degrees, with (100, 100) as the center.
img_ = mmcv.imrotate(img, 30, center=(100, 100))

# rotate the image clockwise by 30 degrees, and extend the image size.
img_ = mmcv.imrotate(img, 30, auto_bound=True)
```

9.1.5 Flip

To flip an image, use `imflip`.

```
img = mmcv.imread('tests/data/color.jpg')

# flip the image horizontally
mmcv.imflip(img)

# flip the image vertically
mmcv.imflip(img, direction='vertical')
```

9.1.6 Crop

`imcrop` can crop the image with one or some regions, represented as (x1, y1, x2, y2).

```
import mmcv
import numpy as np

img = mmcv.imread('tests/data/color.jpg')

# crop the region (10, 10, 100, 120)
bboxes = np.array([10, 10, 100, 120])
patch = mmcv.imcrop(img, bboxes)

# crop two regions (10, 10, 100, 120) and (0, 0, 50, 50)
bboxes = np.array([[10, 10, 100, 120], [0, 0, 50, 50]])
patches = mmcv.imcrop(img, bboxes)

# crop two regions, and rescale the patches by 1.2x
patches = mmcv.imcrop(img, bboxes, scale=1.2)
```

9.1.7 Padding

There are two methods `impad` and `impad_to_multiple` to pad an image to the specific size with given values.

```
img = mmcv.imread('tests/data/color.jpg')

# pad the image to (1000, 1200) with all zeros
img_ = mmcv.impad(img, shape=(1000, 1200), pad_val=0)

# pad the image to (1000, 1200) with different values for three channels.
img_ = mmcv.impad(img, shape=(1000, 1200), pad_val=(100, 50, 200))

# pad the image on left, right, top, bottom borders with all zeros
img_ = mmcv.impad(img, padding=(10, 20, 30, 40), pad_val=0)

# pad the image on left, right, top, bottom borders with different values
# for three channels.
img_ = mmcv.impad(img, padding=(10, 20, 30, 40), pad_val=(100, 50, 200))

# pad an image so that each edge is a multiple of some value.
img_ = mmcv.impad_to_multiple(img, 32)
```

9.2 Video

This module provides the following functionalities.

- A `VideoReader` class with friendly apis to read and convert videos.
- Some methods for editing (cut, concat, resize) videos.
- Optical flow read/write/warp.

9.2.1 VideoReader

The `VideoReader` class provides sequence like apis to access video frames. It will internally cache the frames which have been visited.

```
video = mmcv.VideoReader('test.mp4')

# obtain basic information
print(len(video))
print(video.width, video.height, video.resolution, video.fps)

# iterate over all frames
for frame in video:
    print(frame.shape)

# read the next frame
img = video.read()

# read a frame by index
img = video[100]
```

(continues on next page)

(continued from previous page)

```
# read some frames
img = video[5:10]
```

To convert a video to images or generate a video from a image directory.

```
# split a video into frames and save to a folder
video = mmcv.VideoReader('test.mp4')
video.cvt2frames('out_dir')

# generate video from frames
mmcv.frames2video('out_dir', 'test.avi')
```

9.2.2 Editing utils

There are also some methods for editing videos, which wraps the commands of ffmpeg.

```
# cut a video clip
mmcv.cut_video('test.mp4', 'clip1.mp4', start=3, end=10, vcodec='h264')

# join a list of video clips
mmcv.concat_video(['clip1.mp4', 'clip2.mp4'], 'joined.mp4', log_level='quiet')

# resize a video with the specified size
mmcv.resize_video('test.mp4', 'resized1.mp4', (360, 240))

# resize a video with a scaling ratio of 2
mmcv.resize_video('test.mp4', 'resized2.mp4', ratio=2)
```

9.2.3 Optical flow

mmcv provides the following methods to operate on optical flows.

- IO
- Visualization
- Flow warping

We provide two options to dump optical flow files: uncompressed and compressed. The uncompressed way just dumps the floating numbers to a binary file. It is lossless but the dumped file has a larger size. The compressed way quantizes the optical flow to 0-255 and dumps it as a jpeg image. The flow of x-dim and y-dim will be concatenated into a single image.

1. IO

```
flow = np.random.rand(800, 600, 2).astype(np.float32)
# dump the flow to a flo file (~3.7M)
mmcv.flowwrite(flow, 'uncompressed.flo')
# dump the flow to a jpeg file (~230K)
# the shape of the dumped image is (800, 1200)
mmcv.flowwrite(flow, 'compressed.jpg', quantize=True, concat_axis=1)
```

(continues on next page)

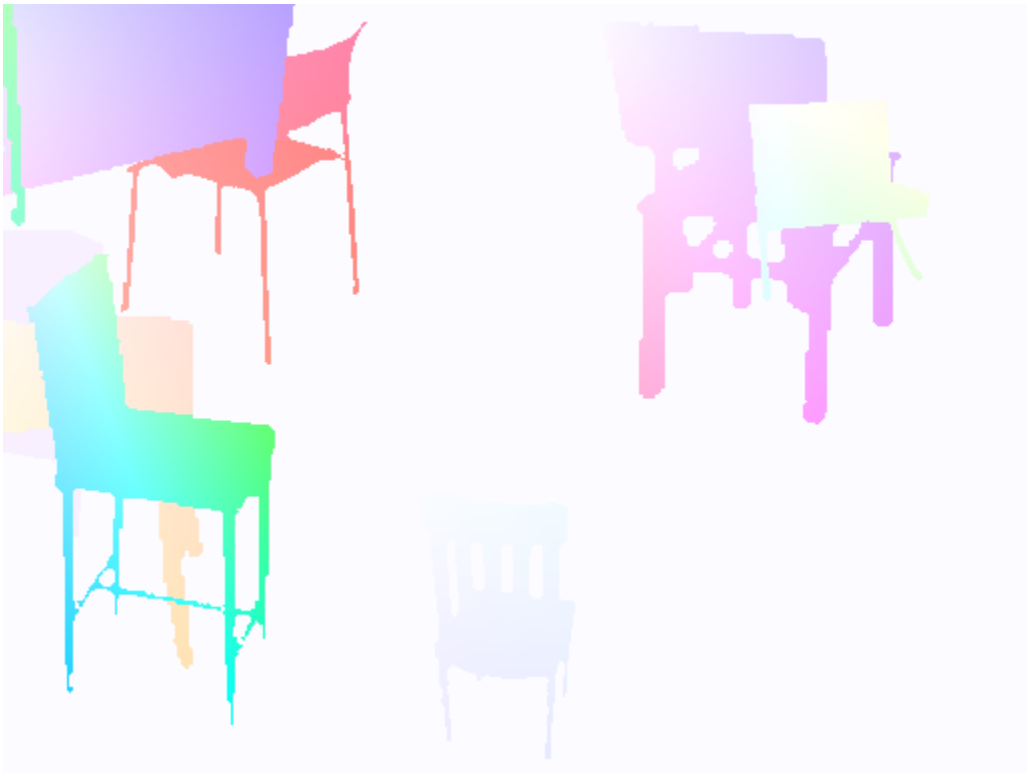
(continued from previous page)

```
# read the flow file, the shape of loaded flow is (800, 600, 2) for both ways
flow = mmcv.flowread('uncompressed.flo')
flow = mmcv.flowread('compressed.jpg', quantize=True, concat_axis=1)
```

2. Visualization

It is possible to visualize optical flows with `mmcv.flowshow()`.

```
mmcv.flowshow(flow)
```



3. Flow warpping

```
img1 = mmcv.imread('img1.jpg')
flow = mmcv.flowread('flow.flo')
warpped_img2 = mmcv.flow_warp(img1, flow)
```

img1 (left) and img2 (right)

optical flow (img2 -> img1)

warpped image and difference with ground truth

CHAPTER

TEN

VISUALIZATION

mmcv can show images and annotations (currently supported types include bounding boxes).

```
# show an image file
mmcv.imshow('a.jpg')

# show a loaded image
img = np.random.rand(100, 100, 3)
mmcv.imshow(img)

# show image with bounding boxes
img = np.random.rand(100, 100, 3)
bboxes = np.array([[0, 0, 50, 50], [20, 20, 60, 60]])
mmcv.imshow_bboxes(img, bboxes)
```

mmcv can also visualize special images such as optical flows.

```
flow = mmcv.flowread('test.flo')
mmcv.flowshow(flow)
```

CHAPTER

ELEVEN

CNN

We provide some building bricks for CNNs, including layer building, module bundles and weight initialization.

11.1 Layer building

We may need to try different layers of the same type when running experiments, but do not want to modify the code from time to time. Here we provide some layer building methods to construct layers from a dict, which can be written in configs or specified via command line arguments.

11.1.1 Usage

A simplest example is

```
cfg = dict(type='Conv3d')
layer = build_conv_layer(cfg, in_channels=3, out_channels=8, kernel_size=3)
```

- `build_conv_layer`: Supported types are Conv1d, Conv2d, Conv3d, Conv (alias for Conv2d).
- `build_norm_layer`: Supported types are BN1d, BN2d, BN3d, BN (alias for BN2d), SyncBN, GN, LN, IN1d, IN2d, IN3d, IN (alias for IN2d).
- `build_activation_layer`: Supported types are ReLU, LeakyReLU, PReLU, RReLU, ReLU6, ELU, Sigmoid, Tanh, GELU.
- `build_upsample_layer`: Supported types are nearest, bilinear, deconv, pixel_shuffle.
- `build_padding_layer`: Supported types are zero, reflect, replicate.

11.1.2 Extension

We also allow extending the building methods with custom layers and operators.

1. Write and register your own module.

```
from mmcv.cnn import UPSAMPLE_LAYERS

@UPSAMPLE_LAYERS.register_module()
class MyUpsample:

    def __init__(self, scale_factor):
        pass
```

(continues on next page)

(continued from previous page)

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

2. Import `MyUpsample` somewhere (e.g., in `__init__.py`) and then use it.

```
cfg = dict(type='MyUpsample', scale_factor=2)
layer = build_upsample_layer(cfg)
```

11.2 Module bundles

We also provide common module bundles to facilitate the network construction. `ConvModule` is a bundle of convolution, normalization and activation layers, please refer to the api for details.

```
# conv + bn + relu
conv = ConvModule(3, 8, 2, norm_cfg=dict(type='BN'))
# conv + gn + relu
conv = ConvModule(3, 8, 2, norm_cfg=dict(type='GN', num_groups=2))
# conv + relu
conv = ConvModule(3, 8, 2)
# conv
conv = ConvModule(3, 8, 2, act_cfg=None)
# conv + leaky relu
conv = ConvModule(3, 8, 3, padding=1, act_cfg=dict(type='LeakyReLU'))
# bn + conv + relu
conv = ConvModule(
    3, 8, 2, norm_cfg=dict(type='BN'), order=('norm', 'conv', 'act'))
```

11.3 Weight initialization

Implementation details are available at mmcv/cnn/utils/weight_init.py

During training, a proper initialization strategy is beneficial to speed up the training or obtain a higher performance. In MMCV, we provide some commonly used methods for initializing modules like `nn.Conv2d`. Of course, we also provide high-level APIs for initializing models containing one or more modules.

11.3.1 Initialization functions

Initialize a `nn.Module` such as `nn.Conv2d`, `nn.Linear` in a functional way.

We provide the following initialization methods.

- constant_init

  Initialize module parameters with constant values.

```
>>> import torch.nn as nn
>>> from mmcv.cnn import constant_init
>>> conv1 = nn.Conv2d(3, 3, 1)
```

(continues on next page)

(continued from previous page)

```
>>> # constant_init(module, val, bias=0)
>>> constant_init(conv1, 1, 0)
>>> conv1.weight
```

- xavier_init

  Initialize module parameters with values according to the method described in Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010)

```
>>> import torch.nn as nn
>>> from mmcv.cnn import xavier_init
>>> conv1 = nn.Conv2d(3, 3, 1)
>>> # xavier_init(module, gain=1, bias=0, distribution='normal')
>>> xavier_init(conv1, distribution='normal')
```

- normal_init

  Initialize module parameters with the values drawn from a normal distribution.

```
>>> import torch.nn as nn
>>> from mmcv.cnn import normal_init
>>> conv1 = nn.Conv2d(3, 3, 1)
>>> # normal_init(module, mean=0, std=1, bias=0)
>>> normal_init(conv1, std=0.01, bias=0)
```

- uniform_init

  Initialize module parameters with values drawn from a uniform distribution.

```
>>> import torch.nn as nn
>>> from mmcv.cnn import uniform_init
>>> conv1 = nn.Conv2d(3, 3, 1)
>>> # uniform_init(module, a=0, b=1, bias=0)
>>> uniform_init(conv1, a=0, b=1)
```

- kaiming_init

  Initialize module parameters with the values according to the method described in Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015)

```
>>> import torch.nn as nn
>>> from mmcv.cnn import kaiming_init
>>> conv1 = nn.Conv2d(3, 3, 1)
>>> # kaiming_init(module, a=0, mode='fan_out', nonlinearity='relu', bias=0,␣
↪distribution='normal')
>>> kaiming_init(conv1)
```

- caffe2_xavier_init

  The xavier initialization is implemented in caffe2, which corresponds to `kaiming_uniform_` in PyTorch.

```
>>> import torch.nn as nn
>>> from mmcv.cnn import caffe2_xavier_init
>>> conv1 = nn.Conv2d(3, 3, 1)
>>> # caffe2_xavier_init(module, bias=0)
>>> caffe2_xavier_init(conv1)
```

- bias_init_with_prob

  Initialize conv/fc bias value according to a given probability, as proposed in Focal Loss for Dense Object Detection.

```
>>> from mmcv.cnn import bias_init_with_prob
>>> # bias_init_with_prob is proposed in Focal Loss
>>> bias = bias_init_with_prob(0.01)
>>> bias
-4.59511985013459
```

11.3.2 Initializers and configs

On the basis of the initialization methods, we define the corresponding initialization classes and register them to `INITIALIZERS`, so we can use the configuration to initialize the model.

We provide the following initialization classes.

- ConstantInit
- XavierInit
- NormalInit
- UniformInit
- KaimingInit
- Caffe2XavierInit
- PretrainedInit

Let us introduce the usage of `initialize` in detail.

1. Initialize model by `layer` key

   If we only define `layer`, it just initialize the layer in `layer` key.

   NOTE: Value of `layer` key is the class name with attributes weights and bias of Pytorch, so `MultiheadAttention layer` is not supported.

- Define `layer` key for initializing module with same configuration.

```
import torch.nn as nn
from mmcv.cnn import initialize

class FooNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.feat = nn.Conv1d(3, 1, 3)
        self.reg = nn.Conv2d(3, 3, 3)
        self.cls = nn.Linear(1, 2)

model = FooNet()
init_cfg = dict(type='Constant', layer=['Conv1d', 'Conv2d', 'Linear'], val=1)
# initialize whole module with same configuration
initialize(model, init_cfg)
# model.feat.weight
# Parameter containing:
```

(continues on next page)

(continued from previous page)

```
# tensor([[[1., 1., 1.],
#          [1., 1., 1.],
#          [1., 1., 1.]]], requires_grad=True)
```

- Define `layer` key for initializing layer with different configurations.

```
import torch.nn as nn
from mmcv.cnn.utils import initialize

class FooNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.feat = nn.Conv1d(3, 1, 3)
        self.reg = nn.Conv2d(3, 3, 3)
        self.cls = nn.Linear(1,2)

model = FooNet()
init_cfg = [dict(type='Constant', layer='Conv1d', val=1),
            dict(type='Constant', layer='Conv2d', val=2),
            dict(type='Constant', layer='Linear', val=3)]
# nn.Conv1d will be initialized with dict(type='Constant', val=1)
# nn.Conv2d will be initialized with dict(type='Constant', val=2)
# nn.Linear will be initialized with dict(type='Constant', val=3)
initialize(model, init_cfg)
# model.reg.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]],
#          ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]]], requires_grad=True)
```

2. Initialize model by `override` key

- When initializing some specific part with its attribute name, we can use `override` key, and the value in `override` will ignore the value in init_cfg.

```
import torch.nn as nn
from mmcv.cnn import initialize

class FooNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.feat = nn.Conv1d(3, 1, 3)
        self.reg = nn.Conv2d(3, 3, 3)
        self.cls = nn.Sequential(nn.Conv1d(3, 1, 3), nn.Linear(1,2))

# if we would like to initialize model's weights as 1 and bias as 2
# but weight in `reg` as 3 and bias 4, we can use override key
model = FooNet()
init_cfg = dict(type='Constant', layer=['Conv1d','Conv2d'], val=1, bias=2,
```

(continues on next page)

(continued from previous page)

```
                override=dict(type='Constant', name='reg', val=3, bias=4))
# self.feat and self.cls will be initialized with dict(type='Constant', val=1, bias=2)
# The module called 'reg' will be initialized with dict(type='Constant', val=3, bias=4)
initialize(model, init_cfg)
# model.reg.weight
# Parameter containing:
# tensor([[[[3., 3., 3.],
#           [3., 3., 3.],
#           [3., 3., 3.]],
#           ...,
#           [[3., 3., 3.],
#            [3., 3., 3.],
#            [3., 3., 3.]]]], requires_grad=True)
```

- If `layer` is None in init_cfg, only sub-module with the name in override will be initialized, and type and other args in override can be omitted.

```
model = FooNet()
init_cfg = dict(type='Constant', val=1, bias=2, override=dict(name='reg'))
# self.feat and self.cls will be initialized by Pytorch
# The module called 'reg' will be initialized with dict(type='Constant', val=1, bias=2)
initialize(model, init_cfg)
# model.reg.weight
# Parameter containing:
# tensor([[[[1., 1., 1.],
#           [1., 1., 1.],
#           [1., 1., 1.]],
#           ...,
#           [[1., 1., 1.],
#            [1., 1., 1.],
#            [1., 1., 1.]]]], requires_grad=True)
```

- If we don't define `layer` key or `override` key, it will not initialize anything.
- Invalid usage

```
# It is invalid that override don't have name key
init_cfg = dict(type='Constant', layer=['Conv1d','Conv2d'],
                val=1, bias=2,
                override=dict(type='Constant', val=3, bias=4))

# It is also invalid that override has name and other args except type
init_cfg = dict(type='Constant', layer=['Conv1d','Conv2d'],
                val=1, bias=2,
                override=dict(name='reg', val=3, bias=4))
```

3. Initialize model with the pretrained model

```
import torch.nn as nn
import torchvision.models as models
from mmcv.cnn import initialize

# initialize model with pretrained model
```

(continues on next page)

(continued from previous page)

```
model = models.resnet50()
# model.conv1.weight
# Parameter containing:
# tensor([[[[-6.7435e-03, -2.3531e-02, -9.0143e-03,  ..., -2.1245e-03,
#            -1.8077e-03,  3.0338e-03],
#           [-1.2603e-02, -2.7831e-02,  2.3187e-02,  ..., -1.5793e-02,
#             1.1655e-02,  4.5889e-03],
#           [-3.7916e-02,  1.2014e-02,  1.3815e-02,  ..., -4.2651e-03,
#             1.7314e-02, -9.9998e-03],
#           ...,

init_cfg = dict(type='Pretrained',
                checkpoint='torchvision://resnet50')
initialize(model, init_cfg)
# model.conv1.weight
# Parameter containing:
# tensor([[[[ 1.3335e-02,  1.4664e-02, -1.5351e-02,  ..., -4.0896e-02,
#            -4.3034e-02, -7.0755e-02],
#           [ 4.1205e-03,  5.8477e-03,  1.4948e-02,  ...,  2.2060e-03,
#            -2.0912e-02, -3.8517e-02],
#           [ 2.2331e-02,  2.3595e-02,  1.6120e-02,  ...,  1.0281e-01,
#             6.2641e-02,  5.1977e-02],
#           ...,

# initialize weights of a sub-module with the specific part of a pretrained model␣
↪by using 'prefix'
model = models.resnet50()
url = 'http://download.openmmlab.com/mmdetection/v2.0/retinanet/'\
      'retinanet_r50_fpn_1x_coco/'\
      'retinanet_r50_fpn_1x_coco_20200130-c2398f9e.pth'
init_cfg = dict(type='Pretrained',
                checkpoint=url, prefix='backbone.')
initialize(model, init_cfg)
```

4. Initialize model inherited from BaseModule, Sequential, ModuleList, ModuleDict

   `BaseModule` is inherited from `torch.nn.Module`, and the only different between them is that `BaseModule` implements `init_weights()`.

   `Sequential` is inherited from `BaseModule` and `torch.nn.Sequential`.

   `ModuleList` is inherited from `BaseModule` and `torch.nn.ModuleList`.

   `ModuleDict` is inherited from `BaseModule` and `torch.nn.ModuleDict`.

```
import torch.nn as nn
from mmcv.runner import BaseModule, Sequential, ModuleList, ModuleDict

class FooConv1d(BaseModule):

    def __init__(self, init_cfg=None):
        super().__init__(init_cfg)
        self.conv1d = nn.Conv1d(4, 1, 4)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        return self.conv1d(x)

class FooConv2d(BaseModule):

    def __init__(self, init_cfg=None):
        super().__init__(init_cfg)
        self.conv2d = nn.Conv2d(3, 1, 3)

    def forward(self, x):
        return self.conv2d(x)

# BaseModule
init_cfg = dict(type='Constant', layer='Conv1d', val=0., bias=1.)
model = FooConv1d(init_cfg)
model.init_weights()
# model.conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#         [0., 0., 0., 0.],
#         [0., 0., 0., 0.],
#         [0., 0., 0., 0.]]], requires_grad=True)

# Sequential
init_cfg1 = dict(type='Constant', layer='Conv1d', val=0., bias=1.)
init_cfg2 = dict(type='Constant', layer='Conv2d', val=2., bias=3.)
model1 = FooConv1d(init_cfg1)
model2 = FooConv2d(init_cfg2)
seq_model = Sequential(model1, model2)
seq_model.init_weights()
# seq_model[0].conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.]]], requires_grad=True)
# seq_model[1].conv2d.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]],
#          ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]]], requires_grad=True)

# inner init_cfg has higher priority
model1 = FooConv1d(init_cfg1)
model2 = FooConv2d(init_cfg2)
init_cfg = dict(type='Constant', layer=['Conv1d', 'Conv2d'], val=4., bias=5.)
seq_model = Sequential(model1, model2, init_cfg=init_cfg)
seq_model.init_weights()
```

(continues on next page)

(continued from previous page)

```
# seq_model[0].conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.]]], requires_grad=True)
# seq_model[1].conv2d.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]],
#          ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]]], requires_grad=True)

# ModuleList
model1 = FooConv1d(init_cfg1)
model2 = FooConv2d(init_cfg2)
modellist = ModuleList([model1, model2])
modellist.init_weights()
# modellist[0].conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.]]], requires_grad=True)
# modellist[1].conv2d.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]],
#          ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]]], requires_grad=True)

# inner init_cfg has higher priority
model1 = FooConv1d(init_cfg1)
model2 = FooConv2d(init_cfg2)
init_cfg = dict(type='Constant', layer=['Conv1d', 'Conv2d'], val=4., bias=5.)
modellist = ModuleList([model1, model2], init_cfg=init_cfg)
modellist.init_weights()
# modellist[0].conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.],
#          [0., 0., 0., 0.]]], requires_grad=True)
# modellist[1].conv2d.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
```

(continues on next page)

(continued from previous page)

```
#           [2., 2., 2.],
#           [2., 2., 2.]],
#         ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]], requires_grad=True)

# ModuleDict
model1 = FooConv1d(init_cfg1)
model2 = FooConv2d(init_cfg2)
modeldict = ModuleDict(dict(model1=model1, model2=model2))
modeldict.init_weights()
# modeldict['model1'].conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#         [0., 0., 0., 0.],
#         [0., 0., 0., 0.],
#         [0., 0., 0., 0.]]], requires_grad=True)
# modeldict['model2'].conv2d.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]],
#         ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]], requires_grad=True)

# inner init_cfg has higher priority
model1 = FooConv1d(init_cfg1)
model2 = FooConv2d(init_cfg2)
init_cfg = dict(type='Constant', layer=['Conv1d', 'Conv2d'], val=4., bias=5.)
modeldict = ModuleDict(dict(model1=model1, model2=model2), init_cfg=init_cfg)
modeldict.init_weights()
# modeldict['model1'].conv1d.weight
# Parameter containing:
# tensor([[[0., 0., 0., 0.],
#         [0., 0., 0., 0.],
#         [0., 0., 0., 0.],
#         [0., 0., 0., 0.]]], requires_grad=True)
# modeldict['model2'].conv2d.weight
# Parameter containing:
# tensor([[[[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]],
#         ...,
#          [[2., 2., 2.],
#           [2., 2., 2.],
#           [2., 2., 2.]]], requires_grad=True)
```

11.4 Model Zoo

Besides torchvision pre-trained models, we also provide pre-trained models of following CNN:

- VGG Caffe
- ResNet Caffe
- ResNeXt
- ResNet with Group Normalization
- ResNet with Group Normalization and Weight Standardization
- HRNetV2
- Res2Net
- RegNet

11.4.1 Model URLs in JSON

The model zoo links in MMCV are managed by JSON files. The json file consists of key-value pair of model name and its url or path. An example json file could be like:

```
{
    "model_a": "https://example.com/models/model_a_9e5bac.pth",
    "model_b": "pretrain/model_b_ab3ef2c.pth"
}
```

The default links of the pre-trained models hosted on OpenMMLab AWS could be found here.

You may override default links by putting `open-mmlab.json` under `MMCV_HOME`. If `MMCV_HOME` is not find in the environment, `~/.cache/mmcv` will be used by default. You may `export MMCV_HOME=/your/path` to use your own path.

The external json files will be merged into default one. If the same key presents in both external json and default json, the external one will be used.

11.4.2 Load Checkpoint

The following types are supported for `filename` argument of `mmcv.load_checkpoint()`.

- filepath: The filepath of the checkpoint.
- `http://xxx` and `https://xxx`: The link to download the checkpoint. The `SHA256` postfix should be contained in the filename.
- `torchvision://xxx`: The model links in `torchvision.models`.Please refer to torchvision for details.
- `open-mmlab://xxx`: The model links or filepath provided in default and additional json files.

CHAPTER

TWELVE

OPS

We implement common ops used in detection, segmentation, etc.

CHAPTER

THIRTEEN

UTILS

13.1 ProgressBar

If you want to apply a method to a list of items and track the progress, `track_progress` is a good choice. It will display a progress bar to tell the progress and ETA.

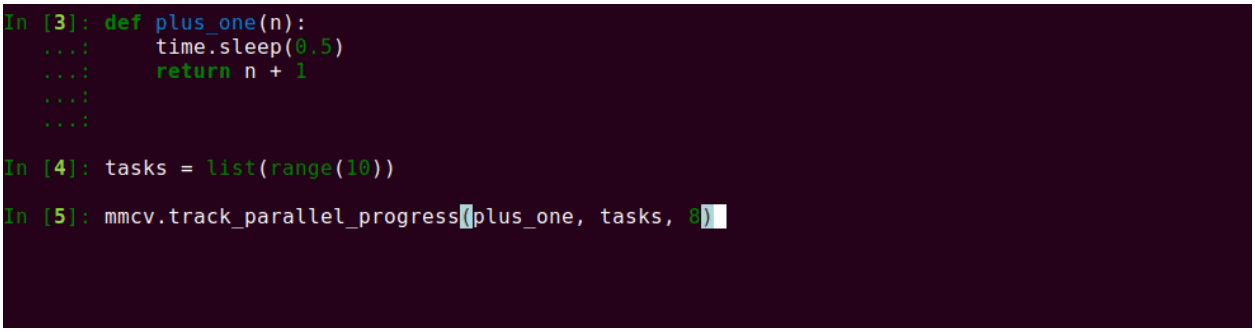
```
import mmcv

def func(item):
    # do something
    pass

tasks = [item_1, item_2, ..., item_n]

mmcv.track_progress(func, tasks)
```

The output is like the following.

```
Python 3.6.6 (default, Sep 12 2018, 18:26:19)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import time

In [2]: import mmcv

In [3]: def plus_one(n):
   ...:     time.sleep(0.5)
   ...:     return n + 1
   ...:
   ...:

In [4]: tasks = list(range(10))

In [5]: mmcv.track_progress(plus_one, tasks)
```

There is another method `track_parallel_progress`, which wraps multiprocessing and progress visualization.

```
mmcv.track_parallel_progress(func, tasks, 8)  # 8 workers
```



If you want to iterate or enumerate a list of items and track the progress, `track_iter_progress` is a good choice. It will display a progress bar to tell the progress and ETA.

```
import mmcv

tasks = [item_1, item_2, ..., item_n]

for task in mmcv.track_iter_progress(tasks):
    # do something like print
    print(task)

for i, task in enumerate(mmcv.track_iter_progress(tasks)):
    # do something like print
    print(i)
    print(task)
```

13.2 Timer

It is convenient to compute the runtime of a code block with `Timer`.

```
import time

with mmcv.Timer():
    # simulate some code block
    time.sleep(1)
```

or try with `since_start()` and `since_last_check()`. This former can return the runtime since the timer starts and the latter will return the time since the last time checked.

```
timer = mmcv.Timer()
# code block 1 here
print(timer.since_start())
# code block 2 here
print(timer.since_last_check())
print(timer.since_start())
```

CHAPTER

FOURTEEN

MMCV OPERATORS

To make custom operators in MMCV more standard, precise definitions of each operator are listed in this document.

14.1 MMCVBorderAlign

14.1.1 Description

Applies `border_align` over the input feature based on predicted bboxes.

For each border line (e.g. top, left, bottom or right) of each box, border_align does the following:

- uniformly samples `pool_size`+1 positions on this line, involving the start and end points.
- the corresponding features on these points are computed by bilinear interpolation.
- max pooling over all the `pool_size`+1 positions are used for computing pooled feature.

Read BorderDet: Border Feature for Dense Object Detection for more detailed information.

14.1.2 Parameters

14.1.3 Inputs

14.1.4 Outputs

14.1.5 Type Constraints

- T:tensor(float32)

14.2 MMCVCARAFE

14.2.1 Description

CARAFE operator performs feature upsampling.

Read CARAFE: Content-Aware ReAssembly of FEatures for more detailed information.

14.2.2 Parameters

14.2.3 Inputs

14.2.4 Outputs

14.2.5 Type Constraints

- T:tensor(float32)

14.3 MMCVCAWeight

14.3.1 Description

Operator for Criss-Cross Attention Read CCNet: Criss-Cross Attention for SemanticSegmentation for more detailed information.

14.3.2 Parameters

None

14.3.3 Inputs

14.3.4 Outputs

14.3.5 Type Constraints

- T:tensor(float32)

14.4 MMCVCAMap

14.4.1 Description

Operator for Criss-Cross Attention Read CCNet: Criss-Cross Attention for SemanticSegmentation for more detailed information.

14.4.2 Parameters

None

14.4.3 Inputs

14.4.4 Outputs

14.4.5 Type Constraints

- T:tensor(float32)

14.5 MMCVCornerPool

14.5.1 Description

Perform CornerPool on `input` features. Read CornerNet – Detecting Objects as Paired Keypoints for more details.

14.5.2 Parameters

14.5.3 Inputs

14.5.4 Outputs

14.5.5 Type Constraints

- T:tensor(float32)

14.6 MMCVDeformConv2d

14.6.1 Description

Applies a deformable 2D convolution over an input signal composed of several input planes.

Read Deformable Convolutional Networks for detail.

14.6.2 Parameters

14.6.3 Inputs

14.6.4 Outputs

14.6.5 Type Constraints

- T:tensor(float32, Linear)

14.7 MMCVModulatedDeformConv2d

14.7.1 Description

Perform Modulated Deformable Convolution on input feature, read Deformable ConvNets v2: More Deformable, Better Results for detail.

14.7.2 Parameters

14.7.3 Inputs

14.7.4 Outputs

14.7.5 Type Constraints

- T:tensor(float32, Linear)

14.8 MMCVDeformRoIPool

14.8.1 Description

Deformable roi pooling layer

14.8.2 Parameters

14.8.3 Inputs

14.8.4 Outputs

14.8.5 Type Constraints

- T:tensor(float32)

14.9 MMCVMaskedConv2d

14.9.1 Description

Performs a masked 2D convolution from PixelRNN Read Pixel Recurrent Neural Networks for more detailed information.

14.9.2 Parameters

14.9.3 Inputs

14.9.4 Outputs

14.9.5 Type Constraints

- T:tensor(float32)

14.10 MMCVPSAMask

14.10.1 Description

An operator from PSANet.

Read PSANet: Point-wise Spatial Attention Network for Scene Parsing for more detailed information.

14.10.2 Parameters

14.10.3 Inputs

14.10.4 Outputs

14.10.5 Type Constraints

- T:tensor(float32)

14.11 NonMaxSuppression

14.11.1 Description

Filter out boxes has high IoU overlap with previously selected boxes or low score. Output the indices of valid boxes.

Note this definition is slightly different with onnx: NonMaxSuppression

14.11.2 Parameters

14.11.3 Inputs

14.11.4 Outputs

14.11.5 Type Constraints

- T:tensor(float32, Linear)

14.12 MMCVRoIAlign

14.12.1 Description

Perform RoIAlign on output feature, used in bbox_head of most two-stage detectors.

14.12.2 Parameters

14.12.3 Inputs

14.12.4 Outputs

14.12.5 Type Constraints

- T:tensor(float32)

14.13 MMCVRoIAlignRotated

14.13.1 Description

Perform RoI align pooling for rotated proposals

14.13.2 Parameters

14.13.3 Inputs

14.13.4 Outputs

14.13.5 Type Constraints

- T:tensor(float32)

14.14 grid_sampler*

14.14.1 Description

Perform sample from `input` with pixel locations from `grid`.

Check torch.nn.functional.grid_sample for more information.

14.14.2 Parameters

14.14.3 Inputs

14.14.4 Outputs

14.14.5 Type Constraints

- T:tensor(float32, Linear)

14.15 cummax*

14.15.1 Description

Returns a tuple (`values`, `indices`) where `values` is the cumulative maximum elements of `input` in the dimension `dim`. And `indices` is the index location of each maximum value found in the dimension `dim`. Read torch.cummax for more details.

14.15.2 Parameters

14.15.3 Inputs

14.15.4 Outputs

14.15.5 Type Constraints

- T:tensor(float32)

14.16 cummin*

14.16.1 Description

Returns a tuple (`values`, `indices`) where `values` is the cumulative minimum elements of `input` in the dimension `dim`. And `indices` is the index location of each minimum value found in the dimension `dim`. Read torch.cummin for more details.

14.16.2 Parameters

14.16.3 Inputs

14.16.4 Outputs

14.16.5 Type Constraints

- T:tensor(float32)

14.17 Reminders

- Operators endwith * are defined in Torch and are included here for the conversion to ONNX.

CHAPTER

FIFTEEN

INTRODUCTION OF MMCV.ONNX MODULE

15.1 DeprecationWarning

ONNX support will be deprecated in the future. Welcome to use the unified model deployment toolbox MMDeploy: https://github.com/open-mmlab/mmdeploy

15.2 register_extra_symbolics

Some extra symbolic functions need to be registered before exporting PyTorch model to ONNX.

15.2.1 Example

```
import mmcv
from mmcv.onnx import register_extra_symbolics

opset_version = 11
register_extra_symbolics(opset_version)
```

15.2.2 Reminder

- *Please note that this feature is experimental and may change in the future.*

15.2.3 FAQs

- None

CHAPTER

SIXTEEN

ONNX RUNTIME CUSTOM OPS

16.1 SoftNMS

16.1.1 Description

Perform soft NMS on `boxes` with `scores`. Read Soft-NMS – Improving Object Detection With One Line of Code for detail.

16.1.2 Parameters

16.1.3 Inputs

16.1.4 Outputs

16.1.5 Type Constraints

- T:tensor(float32)

16.2 RoIAlign

16.2.1 Description

Perform RoIAlign on output feature, used in bbox_head of most two-stage detectors.

16.2.2 Parameters

16.2.3 Inputs

16.2.4 Outputs

16.2.5 Type Constraints

- T:tensor(float32)

16.3 NMS

16.3.1 Description

Filter out boxes has high IoU overlap with previously selected boxes.

16.3.2 Parameters

16.3.3 Inputs

16.3.4 Outputs

16.3.5 Type Constraints

- T:tensor(float32)

16.4 grid_sampler

16.4.1 Description

Perform sample from `input` with pixel locations from `grid`.

16.4.2 Parameters

16.4.3 Inputs

16.4.4 Outputs

16.4.5 Type Constraints

- T:tensor(float32, Linear)

16.5 CornerPool

16.5.1 Description

Perform CornerPool on `input` features. Read CornerNet – Detecting Objects as Paired Keypoints for more details.

16.5.2 Parameters

16.5.3 Inputs

16.5.4 Outputs

16.5.5 Type Constraints

- T:tensor(float32)

16.6 cummax

16.6.1 Description

Returns a tuple (`values`, `indices`) where `values` is the cumulative maximum elements of `input` in the dimension `dim`. And `indices` is the index location of each maximum value found in the dimension `dim`. Read torch.cummax for more details.

16.6.2 Parameters

16.6.3 Inputs

16.6.4 Outputs

16.6.5 Type Constraints

- T:tensor(float32)

16.7 cummin

16.7.1 Description

Returns a tuple (`values`, `indices`) where `values` is the cumulative minimum elements of `input` in the dimension `dim`. And `indices` is the index location of each minimum value found in the dimension `dim`. Read torch.cummin for more details.

16.7.2 Parameters

16.7.3 Inputs

16.7.4 Outputs

16.7.5 Type Constraints

- T:tensor(float32)

16.8 MMCVModulatedDeformConv2d

16.8.1 Description

Perform Modulated Deformable Convolution on input feature, read Deformable ConvNets v2: More Deformable, Better Results for detail.

16.8.2 Parameters

16.8.3 Inputs

16.8.4 Outputs

16.8.5 Type Constraints

- T:tensor(float32, Linear)

16.9 MMCVDeformConv2d

16.9.1 Description

Perform Deformable Convolution on input feature, read Deformable Convolutional Network for detail.

16.9.2 Parameters

16.9.3 Inputs

16.9.4 Outputs

16.9.5 Type Constraints

- T:tensor(float32, Linear)

CHAPTER

SEVENTEEN

ONNX RUNTIME DEPLOYMENT

17.1 DeprecationWarning

ONNX support will be deprecated in the future. Welcome to use the unified model deployment toolbox MMDeploy: https://github.com/open-mmlab/mmdeploy

17.2 Introduction of ONNX Runtime

ONNX Runtime is a cross-platform inferencing and training accelerator compatible with many popular ML/DNN frameworks. Check its github for more information.

17.3 Introduction of ONNX

ONNX stands for **Open Neural Network Exchange**, which acts as *Intermediate Representation(IR)* for ML/DNN models from many frameworks. Check its github for more information.

17.4 Why include custom operators for ONNX Runtime in MMCV

- To verify the correctness of exported ONNX models in ONNX Runtime.
- To ease the deployment of ONNX models with custom operators from `mmcv.ops` in ONNX Runtime.

17.5 List of operators for ONNX Runtime supported in MMCV

17.6 How to build custom operators for ONNX Runtime

*Please be noted that only **onnxruntime>=1.8.1** of CPU version on Linux platform is tested by now.*

17.6.1 Prerequisite

- Clone repository

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

- Download onnxruntime-linux from ONNX Runtime releases, extract it, expose ONNXRUNTIME_DIR and finally add the lib path to LD_LIBRARY_PATH as below:

```
wget https://github.com/microsoft/onnxruntime/releases/download/v1.8.1/onnxruntime-linux-
↪x64-1.8.1.tgz

tar -zxvf onnxruntime-linux-x64-1.8.1.tgz
cd onnxruntime-linux-x64-1.8.1
export ONNXRUNTIME_DIR=$(pwd)
export LD_LIBRARY_PATH=$ONNXRUNTIME_DIR/lib:$LD_LIBRARY_PATH
```

17.6.2 Build on Linux

```
cd mmcv ## to MMCV root directory
MMCV_WITH_OPS=1 MMCV_WITH_ORT=1 python setup.py develop
```

17.7 How to do inference using exported ONNX models with custom operators in ONNX Runtime in python

Install ONNX Runtime with pip

```
pip install onnxruntime==1.8.1
```

Inference Demo

```
import os

import numpy as np
import onnxruntime as ort

from mmcv.ops import get_onnxruntime_op_path

ort_custom_op_path = get_onnxruntime_op_path()
assert os.path.exists(ort_custom_op_path)
session_options = ort.SessionOptions()
session_options.register_custom_ops_library(ort_custom_op_path)
## exported ONNX model with custom operators
onnx_file = 'sample.onnx'
input_data = np.random.randn(1, 3, 224, 224).astype(np.float32)
sess = ort.InferenceSession(onnx_file, session_options)
onnx_results = sess.run(None, {'input' : input_data})
```

17.8 How to add a new custom operator for ONNX Runtime in MMCV

17.8.1 Reminder

- *Please note that this feature is experimental and may change in the future. Strongly suggest users always try with the latest master branch.*
- The custom operator is not included in supported operator list in ONNX Runtime.
- The custom operator should be able to be exported to ONNX.

17.8.2 Main procedures

Take custom operator `soft_nms` for example.

1. Add header `soft_nms.h` to ONNX Runtime include directory `mmcv/ops/csrc/onnxruntime/`
2. Add source `soft_nms.cpp` to ONNX Runtime source directory `mmcv/ops/csrc/onnxruntime/cpu/`
3. Register `soft_nms` operator in onnxruntime_register.cpp

```
#include "soft_nms.h"

SoftNmsOp c_SoftNmsOp;

if (auto status = ortApi->CustomOpDomain_Add(domain, &c_SoftNmsOp)) {
return status;
}
```

4. Add unit test into `tests/test_ops/test_onnx.py` Check here for examples.

Finally, welcome to send us PR of adding custom operators for ONNX Runtime in MMCV. :nerd_face:

17.9 Known Issues

- "RuntimeError: tuple appears in op that does not forward tuples, unsupported kind: `prim::PythonOp`."
    1. Note generally `cummax` or `cummin` is exportable to ONNX as long as the torch version >= 1.5.0, since `torch.cummax` is only supported with torch >= 1.5.0. But when `cummax` or `cummin` serves as an intermediate component whose outputs is used as inputs for another modules, it's expected that torch version must be >= 1.7.0. Otherwise the above error might arise, when running exported ONNX model with onnxruntime.
    2. Solution: update the torch version to 1.7.0 or higher.

17.10 References

- How to export Pytorch model with custom op to ONNX and run it in ONNX Runtime
- How to add a custom operator/kernel in ONNX Runtime

CHAPTER

EIGHTEEN

TENSORRT CUSTOM OPS

18.1 MMCVRoIAlign

18.1.1 Description

Perform RoIAlign on output feature, used in bbox_head of most two stage detectors.

18.1.2 Parameters

18.1.3 Inputs

18.1.4 Outputs

18.1.5 Type Constraints

- T:tensor(float32, Linear)

18.2 ScatterND

18.2.1 Description

ScatterND takes three inputs `data` tensor of rank r >= 1, `indices` tensor of rank q >= 1, and `updates` tensor of rank q + r - indices.shape[-1] - 1. The output of the operation is produced by creating a copy of the input `data`, and then updating its value to values specified by updates at specific index positions specified by `indices`. Its output shape is the same as the shape of `data`. Note that `indices` should not have duplicate entries. That is, two or more updates for the same index-location is not supported.

The `output` is calculated via the following equation:

```
output = np.copy(data)
update_indices = indices.shape[:-1]
for idx in np.ndindex(update_indices):
    output[indices[idx]] = updates[idx]
```

18.2.2 Parameters

None

18.2.3 Inputs

18.2.4 Outputs

18.2.5 Type Constraints

- T:tensor(float32, Linear), tensor(int32, Linear)

18.3 NonMaxSuppression

18.3.1 Description

Filter out boxes has high IoU overlap with previously selected boxes or low score. Output the indices of valid boxes. Indices of invalid boxes will be filled with -1.

18.3.2 Parameters

18.3.3 Inputs

18.3.4 Outputs

18.3.5 Type Constraints

- T:tensor(float32, Linear)

18.4 MMCVDeformConv2d

18.4.1 Description

Perform Deformable Convolution on input feature, read Deformable Convolutional Network for detail.

18.4.2 Parameters

18.4.3 Inputs

18.4.4 Outputs

18.4.5 Type Constraints

- T:tensor(float32, Linear)

18.5 grid_sampler

18.5.1 Description

Perform sample from `input` with pixel locations from `grid`.

18.5.2 Parameters

18.5.3 Inputs

18.5.4 Outputs

18.5.5 Type Constraints

- T:tensor(float32, Linear)

18.6 cummax

18.6.1 Description

Returns a namedtuple (`values`, `indices`) where `values` is the cumulative maximum of elements of `input` in the dimension `dim`. And `indices` is the index location of each maximum value found in the dimension `dim`.

18.6.2 Parameters

18.6.3 Inputs

18.6.4 Outputs

18.6.5 Type Constraints

- T:tensor(float32, Linear)

18.7 cummin

18.7.1 Description

Returns a namedtuple (`values`, `indices`) where `values` is the cumulative minimum of elements of `input` in the dimension `dim`. And `indices` is the index location of each minimum value found in the dimension `dim`.

18.7.2 Parameters

18.7.3 Inputs

18.7.4 Outputs

18.7.5 Type Constraints

- T:tensor(float32, Linear)

18.8 MMCVInstanceNormalization

18.8.1 Description

Carries out instance normalization as described in the paper https://arxiv.org/abs/1607.08022.

y = scale * (x - mean) / sqrt(variance + epsilon) + B, where mean and variance are computed per instance per channel.

18.8.2 Parameters

18.8.3 Inputs

18.8.4 Outputs

18.8.5 Type Constraints

- T:tensor(float32, Linear)

18.9 MMCVModulatedDeformConv2d

18.9.1 Description

Perform Modulated Deformable Convolution on input feature, read Deformable ConvNets v2: More Deformable, Better Results for detail.

18.9.2 Parameters

18.9.3 Inputs

18.9.4 Outputs

18.9.5 Type Constraints

- T:tensor(float32, Linear)

CHAPTER

NINETEEN

TENSORRT DEPLOYMENT

19.1 DeprecationWarning

TensorRT support will be deprecated in the future. Welcome to use the unified model deployment toolbox MMDeploy: https://github.com/open-mmlab/mmdeploy

19.2 Introduction

NVIDIA TensorRT is a software development kit(SDK) for high-performance inference of deep learning models. It includes a deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications. Please check its developer's website for more information. To ease the deployment of trained models with custom operators from `mmcv.ops` using TensorRT, a series of TensorRT plugins are included in MMCV.

19.3 List of TensorRT plugins supported in MMCV

Notes

- All plugins listed above are developed on TensorRT-7.2.1.6.Ubuntu-16.04.x86_64-gnu.cuda-10.2.cudnn8.0

19.4 How to build TensorRT plugins in MMCV

19.4.1 Prerequisite

- Clone repository

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

- Install TensorRT

Download the corresponding TensorRT build from NVIDIA Developer Zone.

For example, for Ubuntu 16.04 on x86-64 with cuda-10.2, the downloaded file is `TensorRT-7.2.1.6.Ubuntu-16.04.x86_64-gnu.cuda-10.2.cudnn8.0.tar.gz`.

Then, install as below:

```
cd ~/Downloads
tar -xvzf TensorRT-7.2.1.6.Ubuntu-16.04.x86_64-gnu.cuda-10.2.cudnn8.0.tar.gz
export TENSORRT_DIR=`pwd`/TensorRT-7.2.1.6
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TENSORRT_DIR/lib
```

Install python packages: tensorrt, graphsurgeon, onnx-graphsurgeon

```
pip install $TENSORRT_DIR/python/tensorrt-7.2.1.6-cp37-none-linux_x86_64.whl
pip install $TENSORRT_DIR/onnx_graphsurgeon/onnx_graphsurgeon-0.2.6-py2.py3-none-any.whl
pip install $TENSORRT_DIR/graphsurgeon/graphsurgeon-0.4.5-py2.py3-none-any.whl
```

For more detailed information of installing TensorRT using tar, please refer to Nvidia' website.

- Install cuDNN

Install cuDNN 8 following Nvidia' website.

19.4.2 Build on Linux

```
cd mmcv ## to MMCV root directory
MMCV_WITH_OPS=1 MMCV_WITH_TRT=1 pip install -e .
```

19.5 Create TensorRT engine and run inference in python

Here is an example.

```
import torch
import onnx

from mmcv.tensorrt import (TRTWrapper, onnx2trt, save_trt_engine,
                                    is_tensorrt_plugin_loaded)

assert is_tensorrt_plugin_loaded(), 'Requires to complie TensorRT plugins in mmcv'

onnx_file = 'sample.onnx'
trt_file = 'sample.trt'
onnx_model = onnx.load(onnx_file)

## Model input
inputs = torch.rand(1, 3, 224, 224).cuda()
## Model input shape info
opt_shape_dict = {
    'input': [list(inputs.shape),
              list(inputs.shape),
              list(inputs.shape)]
}

## Create TensorRT engine
max_workspace_size = 1 << 30
trt_engine = onnx2trt(
    onnx_model,
    opt_shape_dict,
    max_workspace_size=max_workspace_size)

## Save TensorRT engine
save_trt_engine(trt_engine, trt_file)

## Run inference with TensorRT
trt_model = TRTWrapper(trt_file, ['input'], ['output'])

with torch.no_grad():
    trt_outputs = trt_model({'input': inputs})
    output = trt_outputs['output']
```

19.6 How to add a TensorRT plugin for custom op in MMCV

19.6.1 Main procedures

Below are the main steps:

1. Add c++ header file
2. Add c++ source file
3. Add cuda kernel file
4. Register plugin in `trt_plugin.cpp`
5. Add unit test in `tests/test_ops/test_tensorrt.py`

Take RoIAlign plugin `roi_align` for example.

1. Add header `trt_roi_align.hpp` to TensorRT include directory `mmcv/ops/csrc/tensorrt/`
2. Add source `trt_roi_align.cpp` to TensorRT source directory `mmcv/ops/csrc/tensorrt/plugins/`
3. Add cuda kernel `trt_roi_align_kernel.cu` to TensorRT source directory `mmcv/ops/csrc/tensorrt/plugins/`
4. Register `roi_align` plugin in trt_plugin.cpp

```
#include "trt_plugin.hpp"

#include "trt_roi_align.hpp"

REGISTER_TENSORRT_PLUGIN(RoIAlignPluginDynamicCreator);

extern "C" {
bool initLibMMCVInferPlugins() { return true; }
}  // extern "C"
```

5. Add unit test into `tests/test_ops/test_tensorrt.py` Check here for examples.

19.6.2 Reminders

- *Please note that this feature is experimental and may change in the future. Strongly suggest users always try with the latest master branch.*
- Some of the custom ops in `mmcv` have their cuda implementations, which could be referred.

19.7 Known Issues

- None

19.8 References

- Developer guide of Nvidia TensorRT
- TensorRT Open Source Software
- onnx-tensorrt
- TensorRT python API
- TensorRT c++ plugin API

CHAPTER

TWENTY

ENGLISH

CHAPTER

TWENTYONE

CHAPTER

TWENTYTWO

V1.3.18

Some ops have different implementations on different devices. Lots of macros and type checks are scattered in several files, which makes the code hard to maintain. For example:

```
  if (input.device().is_cuda()) {
#ifdef MMCV_WITH_CUDA
    CHECK_CUDA_INPUT(input);
    CHECK_CUDA_INPUT(rois);
    CHECK_CUDA_INPUT(output);
    CHECK_CUDA_INPUT(argmax_y);
    CHECK_CUDA_INPUT(argmax_x);

    roi_align_forward_cuda(input, rois, output, argmax_y, argmax_x,
                           aligned_height, aligned_width, spatial_scale,
                           sampling_ratio, pool_mode, aligned);
#else
    AT_ERROR("RoIAlign is not compiled with GPU support");
#endif
  } else {
    CHECK_CPU_INPUT(input);
    CHECK_CPU_INPUT(rois);
    CHECK_CPU_INPUT(output);
    CHECK_CPU_INPUT(argmax_y);
    CHECK_CPU_INPUT(argmax_x);
    roi_align_forward_cpu(input, rois, output, argmax_y, argmax_x,
                          aligned_height, aligned_width, spatial_scale,
                          sampling_ratio, pool_mode, aligned);
  }
```

Registry and dispatcher are added to manage these implementations.

```
void ROIAlignForwardCUDAKernelLauncher(Tensor input, Tensor rois, Tensor output,
                                       Tensor argmax_y, Tensor argmax_x,
                                       int aligned_height, int aligned_width,
                                       float spatial_scale, int sampling_ratio,
                                       int pool_mode, bool aligned);

void roi_align_forward_cuda(Tensor input, Tensor rois, Tensor output,
                            Tensor argmax_y, Tensor argmax_x,
                            int aligned_height, int aligned_width,
                            float spatial_scale, int sampling_ratio,
                            int pool_mode, bool aligned) {
```

(continues on next page)

(continued from previous page)

```
  ROIAlignForwardCUDAKernelLauncher(
      input, rois, output, argmax_y, argmax_x, aligned_height, aligned_width,
      spatial_scale, sampling_ratio, pool_mode, aligned);
}

// register cuda implementation
void roi_align_forward_impl(Tensor input, Tensor rois, Tensor output,
                            Tensor argmax_y, Tensor argmax_x,
                            int aligned_height, int aligned_width,
                            float spatial_scale, int sampling_ratio,
                            int pool_mode, bool aligned);
REGISTER_DEVICE_IMPL(roi_align_forward_impl, CUDA, roi_align_forward_cuda);

// roi_align.cpp
// use the dispatcher to invoke different implementation depending on device type of␣
↪input tensors.
void roi_align_forward_impl(Tensor input, Tensor rois, Tensor output,
                            Tensor argmax_y, Tensor argmax_x,
                            int aligned_height, int aligned_width,
                            float spatial_scale, int sampling_ratio,
                            int pool_mode, bool aligned) {
  DISPATCH_DEVICE_IMPL(roi_align_forward_impl, input, rois, output, argmax_y,
                       argmax_x, aligned_height, aligned_width, spatial_scale,
                       sampling_ratio, pool_mode, aligned);
}
```

CHAPTER

TWENTYTHREE

V1.3.11

In order to flexibly support more backends and hardwares like `NVIDIA GPUs` and `AMD GPUs`, the directory of `mmcv/ops/csrc` is refactored. Note that this refactoring will not affect the usage in API. For related information, please refer to PR1206.

The original directory was organized as follows.

```
.
├── common_cuda_helper.hpp
├── ops_cuda_kernel.cuh
├── pytorch_cpp_helper.hpp
├── pytorch_cuda_helper.hpp
├── parrots_cpp_helper.hpp
├── parrots_cuda_helper.hpp
├── parrots_cudawarpfunction.cuh
├── onnxruntime
│   ├── onnxruntime_register.h
│   ├── onnxruntime_session_options_config_keys.h
│   ├── ort_mmcv_utils.h
│   ├── ...
│   ├── onnx_ops.h
│   └── cpu
│       ├── onnxruntime_register.cpp
│       ├── ...
│       └── onnx_ops_impl.cpp
├── parrots
│   ├── ...
│   ├── ops.cpp
│   ├── ops_cuda.cu
│   ├── ops_parrots.cpp
│   └── ops_pytorch.h
├── pytorch
│   ├── ...
│   ├── ops.cpp
│   ├── ops_cuda.cu
│   ├── pybind.cpp
└── tensorrt
    ├── trt_cuda_helper.cuh
    ├── trt_plugin_helper.hpp
    ├── trt_plugin.hpp
    ├── trt_serialize.hpp
    ├── ...
```

(continues on next page)

(continued from previous page)

```
    ├── trt_ops.hpp
    └── plugins
        ├── trt_cuda_helper.cu
        ├── trt_plugin.cpp
        ├── ...
        ├── trt_ops.cpp
        └── trt_ops_kernel.cu
```

After refactored, it is organized as follows.

```
.
├── common
│   ├── box_iou_rotated_utils.hpp
│   ├── parrots_cpp_helper.hpp
│   ├── parrots_cuda_helper.hpp
│   ├── pytorch_cpp_helper.hpp
│   ├── pytorch_cuda_helper.hpp
│   └── cuda
│       ├── common_cuda_helper.hpp
│       ├── parrots_cudawarpfunction.cuh
│       ├── ...
│       └── ops_cuda_kernel.cuh
├── onnxruntime
│   ├── onnxruntime_register.h
│   ├── onnxruntime_session_options_config_keys.h
│   ├── ort_mmcv_utils.h
│   ├── ...
│   ├── onnx_ops.h
│   └── cpu
│       ├── onnxruntime_register.cpp
│       ├── ...
│       └── onnx_ops_impl.cpp
├── parrots
│   ├── ...
│   ├── ops.cpp
│   ├── ops_parrots.cpp
│   └── ops_pytorch.h
├── pytorch
│   ├── info.cpp
│   ├── pybind.cpp
│   ├── ...
│   ├── ops.cpp
│   └── cuda
│       ├── ...
│       └── ops_cuda.cu
└── tensorrt
    ├── trt_cuda_helper.cuh
    ├── trt_plugin_helper.hpp
    ├── trt_plugin.hpp
    ├── trt_serialize.hpp
    ├── ...
    ├── trt_ops.hpp
```

(continues on next page)

(continued from previous page)

```
    └── plugins
        ├── trt_cuda_helper.cu
        ├── trt_plugin.cpp
        ├── ...
        ├── trt_ops.cpp
        └── trt_ops_kernel.cu
```

CHAPTER

TWENTYFOUR

FREQUENTLY ASKED QUESTIONS

We list some common troubles faced by many users and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them.

24.1 Installation

- KeyError: "xxx: 'yyy is not in the zzz registry'"

  The registry mechanism will be triggered only when the file of the module is imported. So you need to import that file somewhere. More details can be found at KeyError: "MaskRCNN: 'RefineRoIHead is not in the models registry'".

- "No module named 'mmcv.ops'"; "No module named 'mmcv._ext'"

  1. Uninstall existing mmcv in the environment using `pip uninstall mmcv`
  2. Install mmcv-full following the installation instruction or Build MMCV from source

- "invalid device function" or "no kernel image is available for execution"

  1. Check the CUDA compute capability of you GPU
  2. Run `python mmdet/utils/collect_env.py` to check whether PyTorch, torchvision, and MMCV are built for the correct GPU architecture. You may need to set `TORCH_CUDA_ARCH_LIST` to reinstall MMCV. The compatibility issue could happen when using old GPUS, e.g., Tesla K80 (3.7) on colab.
  3. Check whether the running environment is the same as that when mmcv/mmdet is compiled. For example, you may compile mmcv using CUDA 10.0 bug run it on CUDA9.0 environments

- "undefined symbol" or "cannot open xxx.so"

  1. If those symbols are CUDA/C++ symbols (e.g., libcudart.so or GLIBCXX), check whether the CUDA/GCC runtimes are the same as those used for compiling mmcv
  2. If those symbols are Pytorch symbols (e.g., symbols containing caffe, aten, and TH), check whether the Pytorch version is the same as that used for compiling mmcv
  3. Run `python mmdet/utils/collect_env.py` to check whether PyTorch, torchvision, and MMCV are built by and running on the same environment

- "RuntimeError: CUDA error: invalid configuration argument"

  This error may be caused by the poor performance of GPU. Try to decrease the value of THREADS_PER_BLOCK and recompile mmcv.

- "RuntimeError: nms is not compiled with GPU support"

This error is because your CUDA environment is not installed correctly. You may try to re-install your CUDA environment and then delete the build/ folder before re-compile mmcv.

- "Segmentation fault"

  1. Check your GCC version and use GCC >= 5.4. This usually caused by the incompatibility between PyTorch and the environment (e.g., GCC < 4.9 for PyTorch). We also recommend the users to avoid using GCC 5.5 because many feedbacks report that GCC 5.5 will cause "segmentation fault" and simply changing it to GCC 5.4 could solve the problem

  2. Check whether PyTorch is correctly installed and could use CUDA op, e.g. type the following command in your terminal and see whether they could correctly output results

     ```
     python -c 'import torch; print(torch.cuda.is_available())'
     ```

  3. If PyTorch is correctly installed, check whether MMCV is correctly installed. If MMCV is correctly installed, then there will be no issue of the command

     ```
     python -c 'import mmcv; import mmcv.ops'
     ```

  4. If MMCV and PyTorch are correctly installed, you can use `ipdb` to set breakpoints or directly add `print` to debug and see which part leads the `segmentation fault`

- "libtorch_cuda_cu.so: cannot open shared object file"

  `mmcv-full` depends on the share object but it can not be found. We can check whether the object exists in `~/miniconda3/envs/{environment-name}/lib/python3.7/site-packages/torch/lib` or try to re-install the PyTorch.

- "fatal error C1189: #error: – unsupported Microsoft Visual Studio version!"

  If you are building mmcv-full on Windows and the version of CUDA is 9.2, you will probably encounter the error `"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.2\include\crt/host_config.h(133): fatal error C1189: #error: -- unsupported Microsoft Visual Studio version! Only the versions 2012, 2013, 2015 and 2017 are supported!"`, in which case you can use a lower version of Microsoft Visual Studio like vs2017.

- "error: member "torch::jit::detail::ModulePolicy::all_slots" may not be initialized"

  If your version of PyTorch is 1.5.0 and you are building mmcv-full on Windows, you will probably encounter the error `- torch/csrc/jit/api/module.h(474): error: member "torch::jit::detail::ModulePolicy::all_slots" may not be initialized`. The way to solve the error is to replace all the `static constexpr bool all_slots = false;` with `static bool all_slots = false;` at this file `https://github.com/pytorch/pytorch/blob/v1.5.0/torch/csrc/jit/api/module.h`. More details can be found at member "torch::jit::detail::AttributePolicy::all_slots" may not be initialized.

- "error: a member with an in-class initializer must be const"

  If your version of PyTorch is 1.6.0 and you are building mmcv-full on Windows, you will probably encounter the error `"- torch/include\torch/csrc/jit/api/module.h(483): error: a member with an in-class initializer must be const"`. The way to solve the error is to replace all the `CONSTEXPR_EXCEPT_WIN_CUDA` with `const` at `torch/include\torch/csrc/jit/api/module.h`. More details can be found at Ninja: build stopped: subcommand failed.

- "error: member "torch::jit::ProfileOptionalOp::Kind" may not be initialized"

  If your version of PyTorch is 1.7.0 and you are building mmcv-full on Windows, you will probably encounter the error `torch/include\torch/csrc/jit/ir/ir.h(1347): error: member "torch::jit::ProfileOptionalOp::Kind" may not be initialized`. The way to solve the error needs to modify several local files of PyTorch:

- delete `static constexpr Symbol Kind = ::c10::prim::profile;` and `tatic constexpr Symbol Kind = ::c10::prim::profile_optional;` at `torch/include\torch/csrc/jit/ir/ir.h`
- replace `explicit operator type&() { return *(this->value); }` with `explicit operator type&() { return *((type*)this->value); }` at `torch\include\pybind11\cast.h`
- replace all the `CONSTEXPR_EXCEPT_WIN_CUDA` with `const` at `torch/include\torch/csrc/jit/api/module.h`

More details can be found at Ensure default extra_compile_args.

- Compatibility issue between MMCV and MMDetection; "ConvWS is already registered in conv layer"

  Please install the correct version of MMCV for the version of your MMDetection following the installation instruction.

24.2 Usage

- "RuntimeError: Expected to have finished reduction in the prior iteration before starting a new one"

  1. This error indicates that your module has parameters that were not used in producing loss. This phenomenon may be caused by running different branches in your code in DDP mode. More datails at Expected to have finished reduction in the prior iteration before starting a new one.
  2. You can set `find_unused_parameters = True` in the config to solve the above problems or find those unused parameters manually

- "RuntimeError: Trying to backward through the graph a second time"

  `GradientCumulativeOptimizerHook` and `OptimizerHook` are both set which causes the `loss.backward()` to be called twice so `RuntimeError` was raised. We can only use one of these. More datails at Trying to backward through the graph a second time.

CHAPTER

TWENTYFIVE

PULL REQUEST (PR)

25.1 What is PR

PR is the abbreviation of `Pull Request`. Here's the definition of PR in the official document of Github.

```
Pull requests let you tell others about changes you have pushed to a branch in a␣
↪repository on GitHub. Once a pull request is opened, you can discuss and review the␣
↪potential changes with collaborators and add follow-up commits before your changes are␣
↪merged into the base branch.
```

25.2 Basic Workflow

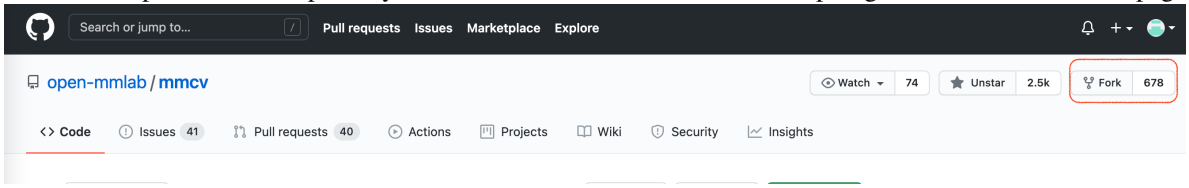
1. Get the most recent codebase
2. Checkout a new branch from the master branch
3. Commit your changes
4. Push your changes and create a PR
5. Discuss and review your code
6. Merge your branch to the master branch

25.3 Procedures in detail

25.3.1 1. Get the most recent codebase

- When you work on your first PR

  Fork the OpenMMLab repository: click the **fork** button at the top right corner of Github page



  Clone forked repository to local

```
git clone git@github.com:XXX/mmcv.git
```

Add source repository to upstream

```
git remote add upstream git@github.com:open-mmlab/mmcv
```

- After your first PR

  Checkout master branch of the local repository and pull the latest master branch of the source repository

```
git checkout master
git pull upstream master
```

25.3.2 2. Checkout a new branch from the master branch

```
git checkout -b branchname
```

Tip: To make commit history clear, we strongly recommend you checkout the master branch before create a new branch.

25.3.3 3. Commit your changes

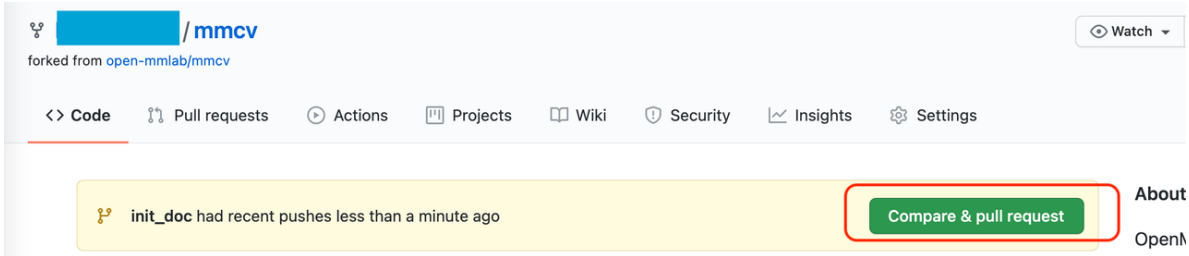
```
# coding
git add [files]
git commit -m 'messages'
```

25.3.4 4. Push your changes to the forked repository and create a PR

- Push the branch to your forked remote repository

```
git push origin branchname
```

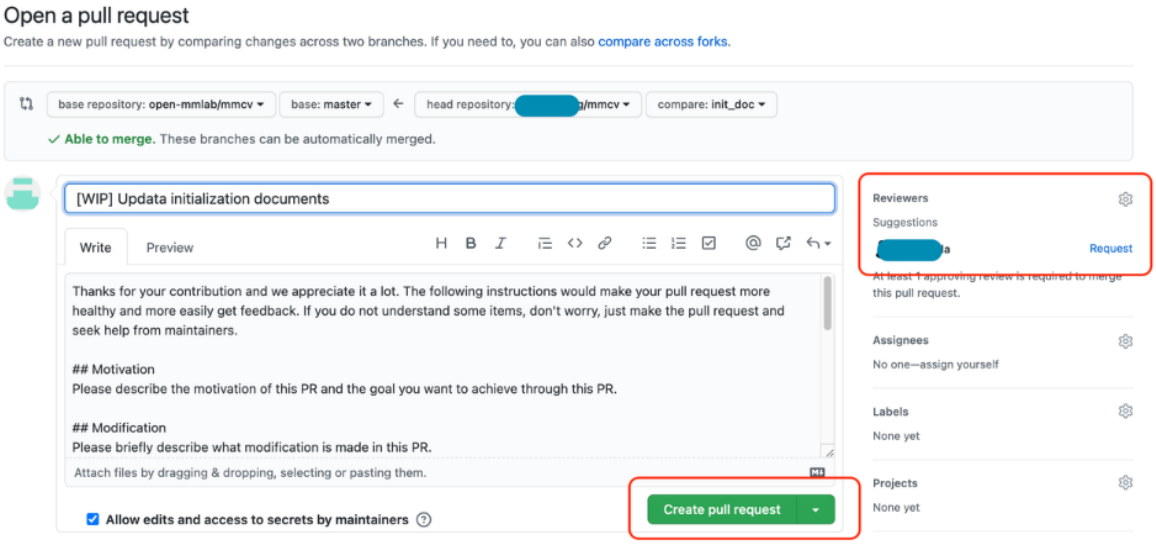
- Create a PR



- Revise PR message template to describe your motivation and modifications made in this PR. You can also link the related issue to the PR manually in the PR message (For more information, checkout the official guidance).

25.3.5 5. Discuss and review your code

- After creating a pull request, you can ask a specific person to review the changes you've proposed



- Modify your codes according to reviewers' suggestions and then push your changes

25.3.6 6. Merge your branch to the master branch and delete the branch

```
git branch -d branchname # delete local branch
git push origin --delete branchname # delete remote branch
```

25.4 PR Specs

1. Use pre-commit hook to avoid issues of code style
2. One short-time branch should be matched with only one PR
3. Accomplish a detailed change in one PR. Avoid large PR
   - Bad: Support Faster R-CNN
   - Acceptable: Add a box head to Faster R-CNN
   - Good: Add a parameter to box head to support custom conv-layer number
4. Provide clear and significant commit message
5. Provide clear and meaningful PR description
   - Task name should be clarified in title. The general format is: [Prefix] Short description of the PR (Suffix)
   - Prefix: add new feature [Feature], fix bug [Fix], related to documents [Docs], in developing [WIP] (which will not be reviewed temporarily)
   - Introduce main changes, results and influences on other modules in short description
   - Associate related issues and pull requests with a milestone

CHAPTER

TWENTYSIX

FILEIO

class mmcv.fileio.**BaseStorageBackend**

Abstract class of storage backends.

All backends need to implement two apis: `get()` and `get_text()`. `get()` reads the file as a byte stream and `get_text()` reads the file as texts.

class mmcv.fileio.**FileClient**(*backend=None*, *prefix=None*, ***kwargs*)

A general file client to access files in different backends.

The client loads a file or text in a specified backend from its path and returns it as a binary or text file. There are two ways to choose a backend, the name of backend and the prefix of path. Although both of them can be used to choose a storage backend, `backend` has a higher priority that is if they are all set, the storage backend will be chosen by the backend argument. If they are all *None*, the disk backend will be chosen. Note that It can also register other backend accessor with a given name, prefixes, and backend class. In addition, We use the singleton pattern to avoid repeated object creation. If the arguments are the same, the same object will be returned.

Parameters

- **backend** (*str, optional*) – The storage backend type. Options are "disk", "ceph", "memcached", "lmdb", "http" and "petrel". Default: None.
- **prefix** (*str, optional*) – The prefix of the registered storage backend. Options are "s3", "http", "https". Default: None.

Examples

```
>>> # only set backend
>>> file_client = FileClient(backend='petrel')
>>> # only set prefix
>>> file_client = FileClient(prefix='s3')
>>> # set both backend and prefix but use backend to choose client
>>> file_client = FileClient(backend='petrel', prefix='s3')
>>> # if the arguments are the same, the same object is returned
>>> file_client1 = FileClient(backend='petrel')
>>> file_client1 is file_client
True
```

client

The backend object.

Type *BaseStorageBackend*

exists(*filepath: Union[str, pathlib.Path]*) → bool

Check whether a file path exists.

Parameters **filepath** (*str or Path*) – Path to be checked whether exists.

Returns Return `True` if `filepath` exists, `False` otherwise.

Return type bool

get(*filepath: Union[str, pathlib.Path]*) → Union[bytes, memoryview]
Read data from a given `filepath` with 'rb' mode.

Note: There are two types of return values for `get`, one is `bytes` and the other is `memoryview`. The advantage of using memoryview is that you can avoid copying, and if you want to convert it to `bytes`, you can use `.tobytes()`.

Parameters **filepath** (*str or Path*) – Path to read data.

Returns Expected bytes object or a memory view of the bytes object.

Return type bytes | memoryview

get_local_path(*filepath: Union[str, pathlib.Path]*) → Generator[Union[str, pathlib.Path], None, None]
Download data from `filepath` and write the data to local path.

`get_local_path` is decorated by `contxtlib.contextmanager()`. It can be called with `with` statement, and when exists from the `with` statement, the temporary path will be released.

Note: If the `filepath` is a local path, just return itself.

Warning: `get_local_path` is an experimental interface that may change in the future.

Parameters **filepath** (*str or Path*) – Path to be read data.

Examples

```
>>> file_client = FileClient(prefix='s3')
>>> with file_client.get_local_path('s3://bucket/abc.jpg') as path:
...     # do something here
```

Yields *Iterable[str]* – Only yield one path.

get_text(*filepath: Union[str, pathlib.Path]*, *encoding='utf-8'*) → str
Read data from a given `filepath` with 'r' mode.

Parameters

- **filepath** (*str or Path*) – Path to read data.
- **encoding** (*str*) – The encoding format used to open the `filepath`. Default: 'utf-8'.

Returns Expected text reading from `filepath`.

Return type str

classmethod infer_client(*file_client_args: Optional[dict] = None*, *uri: Optional[Union[str, pathlib.Path]] = None*) → *mmcv.fileio.file_client.FileClient*

Infer a suitable file client based on the URI and arguments.

Parameters

- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. Default: None.
- **uri** (*str | Path, optional*) – Uri to be parsed that contains the file prefix. Default: None.

Examples

```
>>> uri = 's3://path/of/your/file'
>>> file_client = FileClient.infer_client(uri=uri)
>>> file_client_args = {'backend': 'petrel'}
>>> file_client = FileClient.infer_client(file_client_args)
```

Returns Instantiated FileClient object.

Return type *FileClient*

isdir(*filepath: Union[str, pathlib.Path]*) → bool

Check whether a file path is a directory.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a directory.

Returns Return `True` if `filepath` points to a directory, `False` otherwise.

Return type bool

isfile(*filepath: Union[str, pathlib.Path]*) → bool

Check whether a file path is a file.

Parameters **filepath** (*str or Path*) – Path to be checked whether it is a file.

Returns Return `True` if `filepath` points to a file, `False` otherwise.

Return type bool

join_path(*filepath: Union[str, pathlib.Path]*, **filepaths: Union[str, pathlib.Path]*) → str

Concatenate all file paths.

Join one or more filepath components intelligently. The return value is the concatenation of filepath and any members of *filepaths.

Parameters **filepath** (*str or Path*) – Path to be concatenated.

Returns The result of concatenation.

Return type str

list_dir_or_file(*dir_path: Union[str, pathlib.Path]*, *list_dir: bool = True*, *list_file: bool = True*, *suffix: Optional[Union[str, Tuple[str]]] = None*, *recursive: bool = False*) → Iterator[str]

Scan a directory to find the interested directories or files in arbitrary order.

Note: *list_dir_or_file()* returns the path relative to `dir_path`.

Parameters

- **dir_path** (*str | Path*) – Path of the directory.
- **list_dir** (*bool*) – List the directories. Default: True.
- **list_file** (*bool*) – List the path of files. Default: True.
- **suffix** (*str or tuple[str], optional*) – File suffix that we are interested in. Default: None.
- **recursive** (*bool*) – If set to True, recursively scan the directory. Default: False.

Yields *Iterable[str]* – A relative path to `dir_path`.

static parse_uri_prefix(*uri: Union[str, pathlib.Path]*) → Optional[str]
Parse the prefix of a uri.

Parameters **uri** (*str | Path*) – Uri to be parsed that contains the file prefix.

Examples

```
>>> FileClient.parse_uri_prefix('s3://path/of/your/file')
's3'
```

Returns Return the prefix of uri if the uri contains '://' else `None`.

Return type str | None

put(*obj: bytes*, *filepath: Union[str, pathlib.Path]*) → None
Write data to a given `filepath` with 'wb' mode.

Note: `put` should create a directory if the directory of `filepath` does not exist.

Parameters

- **obj** (*bytes*) – Data to be written.
- **filepath** (*str or Path*) – Path to write data.

put_text(*obj: str*, *filepath: Union[str, pathlib.Path]*) → None
Write data to a given `filepath` with 'w' mode.

Note: `put_text` should create a directory if the directory of `filepath` does not exist.

Parameters

- **obj** (*str*) – Data to be written.
- **filepath** (*str or Path*) – Path to write data.
- **encoding** (*str, optional*) – The encoding format used to open the *filepath*. Default: 'utf-8'.

classmethod register_backend(*name*, *backend=None*, *force=False*, *prefixes=None*)

Register a backend to FileClient.

This method can be used as a normal class method or a decorator.

```
class NewBackend(BaseStorageBackend):

    def get(self, filepath):
        return filepath

    def get_text(self, filepath):
        return filepath

FileClient.register_backend('new', NewBackend)
```

or

```
@FileClient.register_backend('new')
class NewBackend(BaseStorageBackend):

    def get(self, filepath):
        return filepath

    def get_text(self, filepath):
        return filepath
```

Parameters

- **name** (*str*) – The name of the registered backend.
- **backend** (*class, optional*) – The backend class to be registered, which must be a subclass of *BaseStorageBackend*. When this method is used as a decorator, backend is None. Defaults to None.
- **force** (*bool, optional*) – Whether to override the backend if the name has already been registered. Defaults to False.
- **prefixes** (*str or list[str] or tuple[str], optional*) – The prefixes of the registered storage backend. Default: None. *New in version 1.3.15.*

remove(*filepath: Union[str, pathlib.Path]*) → None

Remove a file.

Parameters **filepath** (*str, Path*) – Path to be removed.

mmcv.fileio.**dict_from_file**(*filename: Union[str, pathlib.Path], key_type: type = <class 'str'>, encoding: str = 'utf-8', file_client_args: Optional[Dict] = None*) → Dict

Load a text file and parse the content as a dict.

Each line of the text file will be two or more columns split by whitespaces or tabs. The first column will be parsed as dict keys, and the following columns will be parsed as dict values.

Note: In v1.3.16 and later, `dict_from_file` supports loading a text file which can be storaged in different backends and parsing the content as a dict.

Parameters

- **filename** (*str*) – Filename.
- **key_type** (*type*) – Type of the dict keys. str is user by default and type conversion will be performed if specified.
- **encoding** (*str*) – Encoding used to open the file. Default utf-8.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.

Examples

```
>>> dict_from_file('/path/of/your/file')  # disk
{'key1': 'value1', 'key2': 'value2'}
>>> dict_from_file('s3://path/of/your/file')  # ceph or petrel
{'key1': 'value1', 'key2': 'value2'}
```

Returns The parsed contents.

Return type dict

mmcv.fileio.**dump**(*obj: Any*, *file: Optional[Union[str, pathlib.Path, TextIO, _io.StringIO, _io.BytesIO]] = None*, *file_format: Optional[str] = None*, *file_client_args: Optional[Dict] = None*, ***kwargs*)

Dump data to json/yaml/pickle strings or files.

This method provides a unified api for dumping data as strings or to files, and also supports custom arguments for each file format.

Note: In v1.3.16 and later, dump supports dumping data as strings or to files which is saved to different backends.

Parameters

- **obj** (*any*) – The python object to be dumped.
- **file** (str or Path or file-like object, optional) – If not specified, then the object is dumped to a str, otherwise to a file specified by the filename or file-like object.
- **file_format** (*str, optional*) – Same as *load()*.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.

Examples

```
>>> dump('hello world', '/path/of/your/file')  # disk
>>> dump('hello world', 's3://path/of/your/file')  # ceph or petrel
```

Returns True for success, False otherwise.

Return type bool

mmcv.fileio.**list_from_file**(*filename: Union[str, pathlib.Path]*, *prefix: str = ''*, *offset: int = 0*, *max_num: int = 0*, *encoding: str = 'utf-8'*, *file_client_args: Optional[Dict] = None*) → List

Load a text file and parse the content as a list of strings.

Note: In v1.3.16 and later, `list_from_file` supports loading a text file which can be storaged in different backends and parsing the content as a list for strings.

Parameters

- **filename** (*str*) – Filename.
- **prefix** (*str*) – The prefix to be inserted to the beginning of each item.
- **offset** (*int*) – The offset of lines.
- **max_num** (*int*) – The maximum number of lines to be read, zeros and negatives mean no limitation.
- **encoding** (*str*) – Encoding used to open the file. Default utf-8.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.

Examples

```
>>> list_from_file('/path/of/your/file')  # disk
['hello', 'world']
>>> list_from_file('s3://path/of/your/file')  # ceph or petrel
['hello', 'world']
```

Returns A list of strings.

Return type list[str]

mmcv.fileio.**load**(*file: Union[str, pathlib.Path, TextIO, _io.StringIO, _io.BytesIO]*, *file_format: Optional[str] = None*, *file_client_args: Optional[Dict] = None*, ***kwargs*)

Load data from json/yaml/pickle files.

This method provides a unified api for loading data from serialized files.

Note: In v1.3.16 and later, `load` supports loading data from serialized files those can be storaged in different backends.

Parameters

- **file** (str or `Path` or file-like object) – Filename or a file-like object.
- **file_format** (*str, optional*) – If not specified, the file format will be inferred from the file extension, otherwise use the specified one. Currently supported formats include "json", "yaml/yml" and "pickle/pkl".
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.

Examples

```
>>> load('/path/of/your/file')  # file is storaged in disk
>>> load('https://path/of/your/file')  # file is storaged in Internet
>>> load('s3://path/of/your/file')  # file is storaged in petrel
```

Returns The content from the file.

CHAPTER

TWENTYSEVEN

IMAGE

mmcv.image.**adjust_brightness**(*img*, *factor=1.0*, *backend=None*)
Adjust image brightness.

This function controls the brightness of an image. An enhancement factor of 0.0 gives a black image. A factor of 1.0 gives the original image. This function blends the source image and the degenerated black image:

$$output = img * factor + degenerated * (1 - factor)$$

Parameters

- **img** (*ndarray*) – Image to be brightened.
- **factor** (*float*) – A value controls the enhancement. Factor 1.0 returns the original image, lower factors mean less color (brightness, contrast, etc), and higher values more. Default 1.
- **backend** (*str | None*) – The image processing backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global `imread_backend` specified by `mmcv.use_backend()` will be used. Defaults to None.

Returns The brightened image.

Return type ndarray

mmcv.image.**adjust_color**(*img*, *alpha=1*, *beta=None*, *gamma=0*, *backend=None*)
It blends the source image and its gray image:

$$output = img * alpha + gray\_img * beta + gamma$$

Parameters

- **img** (*ndarray*) – The input source image.
- **alpha** (*int | float*) – Weight for the source image. Default 1.
- **beta** (*int | float*) – Weight for the converted gray image. If None, it's assigned the value (1 - *alpha*).
- **gamma** (*int | float*) – Scalar added to each sum. Same as `cv2.addWeighted()`. Default 0.
- **backend** (*str | None*) – The image processing backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global `imread_backend` specified by `mmcv.use_backend()` will be used. Defaults to None.

Returns Colored image which has the same size and dtype as input.

Return type ndarray

mmcv.image.**adjust_contrast**(*img*, *factor=1.0*, *backend=None*)
Adjust image contrast.

This function controls the contrast of an image. An enhancement factor of 0.0 gives a solid grey image. A factor of 1.0 gives the original image. It blends the source image and the degenerated mean image:

$$output = img * factor + degenerated * (1 - factor)$$

Parameters

- **img** (*ndarray*) – Image to be contrasted. BGR order.
- **factor** (*float*) – Same as mmcv.adjust_brightness().
- **backend** (*str | None*) – The image processing backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global imread_backend specified by mmcv.use_backend() will be used. Defaults to None.

Returns The contrasted image.

Return type ndarray

mmcv.image.**adjust_hue**(*img: numpy.ndarray*, *hue_factor: float*, *backend: Optional[str] = None*) → numpy.ndarray
Adjust hue of an image.

The image hue is adjusted by converting the image to HSV and cyclically shifting the intensities in the hue channel (H). The image is then converted back to original image mode.

hue_factor is the amount of shift in H channel and must be in the interval *[-0.5, 0.5]*.

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

Parameters

- **img** (*ndarray*) – Image to be adjusted.
- **hue_factor** (*float*) – How much to shift the hue channel. Should be in [-0.5, 0.5]. 0.5 and -0.5 give complete reversal of hue channel in HSV space in positive and negative direction respectively. 0 means no shift. Therefore, both -0.5 and 0.5 will give an image with complementary colors while 0 gives the original image.
- **backend** (*str | None*) – The image processing backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global imread_backend specified by mmcv.use_backend() will be used. Defaults to None.

Returns Hue adjusted image.

Return type ndarray

mmcv.image.**adjust_lighting**(*img*, *eigval*, *eigvec*, *alphastd=0.1*, *to_rgb=True*)
AlexNet-style PCA jitter.

This data augmentation is proposed in ImageNet Classification with Deep Convolutional Neural Networks.

Parameters

- **img** (*ndarray*) – Image to be adjusted lighting. BGR order.
- **eigval** (*ndarray*) – the eigenvalue of the convariance matrix of pixel values, respectively.
- **eigvec** (*ndarray*) – the eigenvector of the convariance matrix of pixel values, respectively.
- **alphastd** (*float*) – The standard deviation for distribution of alpha. Defaults to 0.1
- **to_rgb** (*bool*) – Whether to convert img to rgb.

Returns The adjusted image.

Return type ndarray

mmcv.image.**adjust_sharpness**(*img*, *factor=1.0*, *kernel=None*)
Adjust image sharpness.

This function controls the sharpness of an image. An enhancement factor of 0.0 gives a blurred image. A factor of 1.0 gives the original image. And a factor of 2.0 gives a sharpened image. It blends the source image and the degenerated mean image:

$$output = img * factor + degenerated * (1 - factor)$$

Parameters

- **img** (*ndarray*) – Image to be sharpened. BGR order.
- **factor** (*float*) – Same as `mmcv.adjust_brightness()`.
- **kernel** (*np.ndarray, optional*) – Filter kernel to be applied on the img to obtain the degenerated img. Defaults to None.

Note: No value sanity check is enforced on the kernel set by users. So with an inappropriate kernel, the `adjust_sharpness` may fail to perform the function its name indicates but end up performing whatever transform determined by the kernel.

Returns The sharpened image.

Return type ndarray

mmcv.image.**auto_contrast**(*img*, *cutoff=0*)
Auto adjust image contrast.

This function maximize (normalize) image contrast by first removing cutoff percent of the lightest and darkest pixels from the histogram and remapping the image so that the darkest pixel becomes black (0), and the lightest becomes white (255).

Parameters

- **img** (*ndarray*) – Image to be contrasted. BGR order.
- **cutoff** (*int | float | tuple*) – The cutoff percent of the lightest and darkest pixels to be removed. If given as tuple, it shall be (low, high). Otherwise, the single value will be used for both. Defaults to 0.

Returns The contrasted image.

Return type ndarray

mmcv.image.**bgr2gray**(*img: numpy.ndarray*, *keepdim: bool = False*) → numpy.ndarray
Convert a BGR image to grayscale image.

Parameters

- **img** (*ndarray*) – The input image.
- **keepdim** (*bool*) – If False (by default), then return the grayscale image with 2 dims, otherwise 3 dims.

Returns The converted grayscale image.

Return type ndarray

mmcv.image.**bgr2hls**(*img: numpy.ndarray*) → numpy.ndarray

Convert a BGR image to HLS image.

Parameters **img** (*ndarray or str*) – The input image.

Returns The converted HLS image.

Return type ndarray

mmcv.image.**bgr2hsv**(*img: numpy.ndarray*) → numpy.ndarray

Convert a BGR image to HSV image.

Parameters **img** (*ndarray or str*) – The input image.

Returns The converted HSV image.

Return type ndarray

mmcv.image.**bgr2rgb**(*img: numpy.ndarray*) → numpy.ndarray

Convert a BGR image to RGB image.

Parameters **img** (*ndarray or str*) – The input image.

Returns The converted RGB image.

Return type ndarray

mmcv.image.**bgr2ycbcr**(*img: numpy.ndarray*, *y_only: bool = False*) → numpy.ndarray
Convert a BGR image to YCbCr image.

The bgr version of rgb2ycbcr. It implements the ITU-R BT.601 conversion for standard-definition television. See more details in https://en.wikipedia.org/wiki/YCbCr#ITU-R_BT.601_conversion.

It differs from a similar function in cv2.cvtColor: *BGR <-> YCrCb*. In OpenCV, it implements a JPEG conversion. See more details in https://en.wikipedia.org/wiki/YCbCr#JPEG_conversion.

Parameters

- **img** (*ndarray*) – The input image. It accepts: 1. np.uint8 type with range [0, 255]; 2. np.float32 type with range [0, 1].
- **y_only** (*bool*) – Whether to only return Y channel. Default: False.

Returns The converted YCbCr image. The output image has the same type and range as input image.

Return type ndarray

mmcv.image.**clahe**(*img*, *clip_limit=40.0*, *tile_grid_size=(8, 8)*)
Use CLAHE method to process the image.

See *ZUIDERVELD,K. Contrast Limited Adaptive Histogram Equalization[J]. Graphics Gems, 1994:474-485.* for more information.

Parameters

- **img** (*ndarray*) – Image to be processed.
- **clip_limit** (*float*) – Threshold for contrast limiting. Default: 40.0.

- **tile_grid_size** (*tuple[int]*) – Size of grid for histogram equalization. Input image will be divided into equally sized rectangular tiles. It defines the number of tiles in row and column. Default: (8, 8).

Returns The processed image.

Return type ndarray

mmcv.image.**cutout**(*img: numpy.ndarray*, *shape: Union[int, Tuple[int, int]]*, *pad_val: Union[int, float, tuple] = 0*) → numpy.ndarray

Randomly cut out a rectangle from the original img.

Parameters

- **img** (*ndarray*) – Image to be cutout.
- **shape** (*int | tuple[int]*) – Expected cutout shape (h, w). If given as a int, the value will be used for both h and w.
- **pad_val** (*int | float | tuple[int | float]*) – Values to be filled in the cut area. Defaults to 0.

Returns The cutout image.

Return type ndarray

mmcv.image.**gray2bgr**(*img: numpy.ndarray*) → numpy.ndarray

Convert a grayscale image to BGR image.

Parameters **img** (*ndarray*) – The input image.

Returns The converted BGR image.

Return type ndarray

mmcv.image.**gray2rgb**(*img: numpy.ndarray*) → numpy.ndarray

Convert a grayscale image to RGB image.

Parameters **img** (*ndarray*) – The input image.

Returns The converted RGB image.

Return type ndarray

mmcv.image.**hls2bgr**(*img: numpy.ndarray*) → numpy.ndarray

Convert a HLS image to BGR image.

Parameters **img** (*ndarray or str*) – The input image.

Returns The converted BGR image.

Return type ndarray

mmcv.image.**hsv2bgr**(*img: numpy.ndarray*) → numpy.ndarray

Convert a HSV image to BGR image.

Parameters **img** (*ndarray or str*) – The input image.

Returns The converted BGR image.

Return type ndarray

mmcv.image.**imconvert**(*img: numpy.ndarray*, *src: str*, *dst: str*) → numpy.ndarray
Convert an image from the src colorspace to dst colorspace.

Parameters

- **img** (*ndarray*) – The input image.
- **src** (*str*) – The source colorspace, e.g., 'rgb', 'hsv'.
- **dst** (*str*) – The destination colorspace, e.g., 'rgb', 'hsv'.

Returns The converted image.

Return type ndarray

mmcv.image.**imcrop**(*img: numpy.ndarray*, *bboxes: numpy.ndarray*, *scale: float = 1.0*, *pad_fill: Optional[Union[float, list]] = None*) → Union[numpy.ndarray, List[numpy.ndarray]]
Crop image patches.

3 steps: scale the bboxes -> clip bboxes -> crop and pad.

Parameters

- **img** (*ndarray*) – Image to be cropped.
- **bboxes** (*ndarray*) – Shape (k, 4) or (4,), location of cropped bboxes.
- **scale** (*float, optional*) – Scale ratio of bboxes, the default value 1.0 means no padding.
- **pad_fill** (*Number | list[Number]*) – Value to be filled for padding. Default: None, which means no padding.

Returns The cropped image patches.

Return type list[ndarray] | ndarray

mmcv.image.**imequalize**(*img*)
Equalize the image histogram.

This function applies a non-linear mapping to the input image, in order to create a uniform distribution of grayscale values in the output image.

Parameters **img** (*ndarray*) – Image to be equalized.

Returns The equalized image.

Return type ndarray

mmcv.image.**imflip**(*img: numpy.ndarray*, *direction: str = 'horizontal'*) → numpy.ndarray
Flip an image horizontally or vertically.

Parameters

- **img** (*ndarray*) – Image to be flipped.
- **direction** (*str*) – The flip direction, either "horizontal" or "vertical" or "diagonal".

Returns The flipped image.

Return type ndarray

mmcv.image.**imflip_**(*img: numpy.ndarray*, *direction: str = 'horizontal'*) → numpy.ndarray
Inplace flip an image horizontally or vertically.

Parameters

- **img** (*ndarray*) – Image to be flipped.
- **direction** (*str*) – The flip direction, either "horizontal" or "vertical" or "diagonal".

Returns The flipped image (inplace).

Return type ndarray

mmcv.image.**imfrombytes**(*content: bytes*, *flag: str = 'color'*, *channel_order: str = 'bgr'*, *backend: Optional[str] = None*) → numpy.ndarray

Read an image from bytes.

Parameters

- **content** (*bytes*) – Image bytes got from files or other streams.
- **flag** (*str*) – Same as *imread()*.
- **channel_order** (*str*) – The channel order of the output, candidates are 'bgr' and 'rgb'. Default to 'bgr'.
- **backend** (*str | None*) – The image decoding backend type. Options are *cv2*, *pillow*, *turbojpeg*, *tifffile*, *None*. If backend is None, the global imread_backend specified by `mmcv.use_backend()` will be used. Default: None.

Returns Loaded image array.

Return type ndarray

Examples

```
>>> img_path = '/path/to/img.jpg'
>>> with open(img_path, 'rb') as f:
>>>     img_buff = f.read()
>>> img = mmcv.imfrombytes(img_buff)
>>> img = mmcv.imfrombytes(img_buff, flag='color', channel_order='rgb')
>>> img = mmcv.imfrombytes(img_buff, backend='pillow')
>>> img = mmcv.imfrombytes(img_buff, backend='cv2')
```

mmcv.image.**iminvert**(*img*)

Invert (negate) an image.

Parameters **img** (*ndarray*) – Image to be inverted.

Returns The inverted image.

Return type ndarray

mmcv.image.**imnormalize**(*img*, *mean*, *std*, *to_rgb=True*)

Normalize an image with mean and std.

Parameters

- **img** (*ndarray*) – Image to be normalized.
- **mean** (*ndarray*) – The mean to be used for normalize.
- **std** (*ndarray*) – The std to be used for normalize.
- **to_rgb** (*bool*) – Whether to convert to rgb.

Returns The normalized image.

Return type ndarray

mmcv.image.**imnormalize_**(*img*, *mean*, *std*, *to_rgb=True*)

Inplace normalize an image with mean and std.

Parameters

- **img** (*ndarray*) – Image to be normalized.
- **mean** (*ndarray*) – The mean to be used for normalize.
- **std** (*ndarray*) – The std to be used for normalize.
- **to_rgb** (*bool*) – Whether to convert to rgb.

Returns The normalized image.

Return type ndarray

mmcv.image.**impad**(*img: numpy.ndarray*, ***, *shape: Optional[Tuple[int, int]] = None*, *padding: Optional[Union[int, tuple]] = None*, *pad_val: Union[float, List] = 0*, *padding_mode: str = 'constant'*) → numpy.ndarray

Pad the given image to a certain shape or pad on all sides with specified padding mode and padding value.

Parameters

- **img** (*ndarray*) – Image to be padded.
- **shape** (*tuple[int]*) – Expected padding shape (h, w). Default: None.
- **padding** (*int or tuple[int]*) – Padding on each border. If a single int is provided this is used to pad all borders. If tuple of length 2 is provided this is the padding on left/right and top/bottom respectively. If a tuple of length 4 is provided this is the padding for the left, top, right and bottom borders respectively. Default: None. Note that *shape* and *padding* can not be both set.
- **pad_val** (*Number | Sequence[Number]*) – Values to be filled in padding areas when padding_mode is 'constant'. Default: 0.
- **padding_mode** (*str*) – Type of padding. Should be: constant, edge, reflect or symmetric. Default: constant. - 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]

Returns The padded image.

Return type ndarray

mmcv.image.**impad_to_multiple**(*img: numpy.ndarray*, *divisor: int*, *pad_val: Union[float, List] = 0*) → numpy.ndarray

Pad an image to ensure each edge to be multiple to some number.

Parameters

- **img** (*ndarray*) – Image to be padded.
- **divisor** (*int*) – Padded image edges will be multiple to divisor.
- **pad_val** (*Number | Sequence[Number]*) – Same as *impad()*.

Returns The padded image.

Return type ndarray

mmcv.image.**imread**(*img_or_path: Union[numpy.ndarray, str, pathlib.Path]*, *flag: str = 'color'*, *channel_order: str = 'bgr'*, *backend: Optional[str] = None*, *file_client_args: Optional[dict] = None*) → numpy.ndarray

Read an image.

Note: In v1.4.1 and later, add *file_client_args* parameters.

Parameters

- **img_or_path** (*ndarray or str or Path*) – Either a numpy array or str or pathlib.Path. If it is a numpy array (loaded image), then it will be returned as is.
- **flag** (*str*) – Flags specifying the color type of a loaded image, candidates are *color*, *grayscale*, *unchanged*, *color_ignore_orientation* and *grayscale_ignore_orientation*. By default, *cv2* and *pillow* backend would rotate the image according to its EXIF info unless called with *unchanged* or **_ignore_orientation* flags. *turbojpeg* and *tifffile* backend always ignore image's EXIF info regardless of the flag. The *turbojpeg* backend only supports *color* and *grayscale*.
- **channel_order** (*str*) – Order of channel, candidates are *bgr* and *rgb*.
- **backend** (*str | None*) – The image decoding backend type. Options are *cv2*, *pillow*, *turbojpeg*, *tifffile*, *None*. If backend is None, the global imread_backend specified by `mmcv.use_backend()` will be used. Default: None.
- **file_client_args** (*dict | None*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.

Returns Loaded image array.

Return type ndarray

Examples

```
>>> import mmcv
>>> img_path = '/path/to/img.jpg'
>>> img = mmcv.imread(img_path)
>>> img = mmcv.imread(img_path, flag='color', channel_order='rgb',
...     backend='cv2')
>>> img = mmcv.imread(img_path, flag='color', channel_order='bgr',
...     backend='pillow')
>>> s3_img_path = 's3://bucket/img.jpg'
>>> # infer the file backend by the prefix s3
>>> img = mmcv.imread(s3_img_path)
>>> # manually set the file backend petrel
>>> img = mmcv.imread(s3_img_path, file_client_args={
...     'backend': 'petrel'})
>>> http_img_path = 'http://path/to/img.jpg'
>>> img = mmcv.imread(http_img_path)
>>> img = mmcv.imread(http_img_path, file_client_args={
...     'backend': 'http'})
```

mmcv.image.**imrescale**(*img: numpy.ndarray*, *scale: Union[float, Tuple[int, int]]*, *return_scale: bool = False*, *interpolation: str = 'bilinear'*, *backend: Optional[str] = None*) → Union[numpy.ndarray, Tuple[numpy.ndarray, float]]

Resize image while keeping the aspect ratio.

Parameters

- **img** (*ndarray*) – The input image.
- **scale** (*float | tuple[int]*) – The scaling factor or maximum size. If it is a float number, then the image will be rescaled by this factor, else if it is a tuple of 2 integers, then the image will be rescaled as large as possible within the scale.
- **return_scale** (*bool*) – Whether to return the scaling factor besides the rescaled image.
- **interpolation** (*str*) – Same as `resize()`.
- **backend** (*str | None*) – Same as `resize()`.

Returns The rescaled image.

Return type ndarray

mmcv.image.**imresize**(*img: numpy.ndarray*, *size: Tuple[int, int]*, *return_scale: bool = False*, *interpolation: str = 'bilinear'*, *out: Optional[numpy.ndarray] = None*, *backend: Optional[str] = None*) → Union[Tuple[numpy.ndarray, float, float], numpy.ndarray]

Resize image to a given size.

Parameters

- **img** (*ndarray*) – The input image.
- **size** (*tuple[int]*) – Target size (w, h).
- **return_scale** (*bool*) – Whether to return *w_scale* and *h_scale*.
- **interpolation** (*str*) – Interpolation method, accepted values are "nearest", "bilinear", "bicubic", "area", "lanczos" for 'cv2' backend, "nearest", "bilinear" for 'pillow' backend.
- **out** (*ndarray*) – The output destination.
- **backend** (*str | None*) – The image resize backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global imread_backend specified by `mmcv.use_backend()` will be used. Default: None.

Returns (*resized_img*, *w_scale*, *h_scale*) or *resized_img*.

Return type tuple | ndarray

mmcv.image.**imresize_like**(*img: numpy.ndarray*, *dst_img: numpy.ndarray*, *return_scale: bool = False*, *interpolation: str = 'bilinear'*, *backend: Optional[str] = None*) → Union[Tuple[numpy.ndarray, float, float], numpy.ndarray]

Resize image to the same size of a given image.

Parameters

- **img** (*ndarray*) – The input image.
- **dst_img** (*ndarray*) – The target image.
- **return_scale** (*bool*) – Whether to return *w_scale* and *h_scale*.
- **interpolation** (*str*) – Same as `resize()`.
- **backend** (*str | None*) – Same as `resize()`.

Returns (*resized_img*, *w_scale*, *h_scale*) or *resized_img*.

Return type tuple or ndarray

mmcv.image.**imresize_to_multiple**(*img: numpy.ndarray*, *divisor: Union[int, Tuple[int, int]]*, *size: Optional[Union[int, Tuple[int, int]]] = None*, *scale_factor: Optional[Union[float, Tuple[float, float]]] = None*, *keep_ratio: bool = False*, *return_scale: bool = False*, *interpolation: str = 'bilinear'*, *out: Optional[numpy.ndarray] = None*, *backend: Optional[str] = None*) → Union[Tuple[numpy.ndarray, float, float], numpy.ndarray]

Resize image according to a given size or scale factor and then rounds up the the resized or rescaled image size to the nearest value that can be divided by the divisor.

Parameters

- **img** (*ndarray*) – The input image.
- **divisor** (*int | tuple*) – Resized image size will be a multiple of divisor. If divisor is a tuple, divisor should be (w_divisor, h_divisor).
- **size** (*None | int | tuple[int]*) – Target size (w, h). Default: None.
- **scale_factor** (*None | float | tuple[float]*) – Multiplier for spatial size. Should match input size if it is a tuple and the 2D style is (w_scale_factor, h_scale_factor). Default: None.
- **keep_ratio** (*bool*) – Whether to keep the aspect ratio when resizing the image. Default: False.
- **return_scale** (*bool*) – Whether to return *w_scale* and *h_scale*.
- **interpolation** (*str*) – Interpolation method, accepted values are "nearest", "bilinear", "bicubic", "area", "lanczos" for 'cv2' backend, "nearest", "bilinear" for 'pillow' backend.
- **out** (*ndarray*) – The output destination.
- **backend** (*str | None*) – The image resize backend type. Options are *cv2*, *pillow*, *None*. If backend is None, the global imread_backend specified by `mmcv.use_backend()` will be used. Default: None.

Returns (*resized_img*, *w_scale*, *h_scale*) or *resized_img*.

Return type tuple | ndarray

mmcv.image.**imrotate**(*img: numpy.ndarray*, *angle: float*, *center: Optional[Tuple[float, float]] = None*, *scale: float = 1.0*, *border_value: int = 0*, *interpolation: str = 'bilinear'*, *auto_bound: bool = False*, *border_mode: str = 'constant'*) → numpy.ndarray

Rotate an image.

Parameters

- **img** (*np.ndarray*) – Image to be rotated.
- **angle** (*float*) – Rotation angle in degrees, positive values mean clockwise rotation.
- **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.
- **scale** (*float*) – Isotropic scale factor.
- **border_value** (*int*) – Border value used in case of a constant border. Defaults to 0.
- **interpolation** (*str*) – Same as `resize()`.
- **auto_bound** (*bool*) – Whether to adjust the image size to cover the whole rotated image.
- **border_mode** (*str*) – Pixel extrapolation method. Defaults to 'constant'.

Returns The rotated image.

Return type np.ndarray

mmcv.image.**imshear**(*img: numpy.ndarray*, *magnitude: Union[int, float]*, *direction: str = 'horizontal'*, *border_value: Union[int, Tuple[int, int]] = 0*, *interpolation: str = 'bilinear'*) → numpy.ndarray

Shear an image.

Parameters

- **img** (*ndarray*) – Image to be sheared with format (h, w) or (h, w, c).
- **magnitude** (*int | float*) – The magnitude used for shear.
- **direction** (*str*) – The flip direction, either "horizontal" or "vertical".
- **border_value** (*int | tuple[int]*) – Value used in case of a constant border.
- **interpolation** (*str*) – Same as `resize()`.

Returns The sheared image.

Return type ndarray

mmcv.image.**imtranslate**(*img: numpy.ndarray*, *offset: Union[int, float]*, *direction: str = 'horizontal'*, *border_value: Union[int, tuple] = 0*, *interpolation: str = 'bilinear'*) → numpy.ndarray

Translate an image.

Parameters

- **img** (*ndarray*) – Image to be translated with format (h, w) or (h, w, c).
- **offset** (*int | float*) – The offset used for translate.
- **direction** (*str*) – The translate direction, either "horizontal" or "vertical".
- **border_value** (*int | tuple[int]*) – Value used in case of a constant border.
- **interpolation** (*str*) – Same as `resize()`.

Returns The translated image.

Return type ndarray

mmcv.image.**imwrite**(*img: numpy.ndarray*, *file_path: str*, *params: Optional[list] = None*, *auto_mkdir: Optional[bool] = None*, *file_client_args: Optional[dict] = None*) → bool

Write image to file.

Note: In v1.4.1 and later, add *file_client_args* parameters.

Warning: The parameter *auto_mkdir* will be deprecated in the future and every file clients will make directory automatically.

Parameters

- **img** (*ndarray*) – Image array to be written.
- **file_path** (*str*) – Image file path.
- **params** (*None or list*) – Same as opencv *imwrite()* interface.

- **auto_mkdir** (*bool*) – If the parent folder of *file_path* does not exist, whether to create it automatically. It will be deprecated.
- **file_client_args** (*dict | None*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.

Returns Successful or not.

Return type bool

Examples

```
>>> # write to hard disk client
>>> ret = mmcv.imwrite(img, '/path/to/img.jpg')
>>> # infer the file backend by the prefix s3
>>> ret = mmcv.imwrite(img, 's3://bucket/img.jpg')
>>> # manually set the file backend petrel
>>> ret = mmcv.imwrite(img, 's3://bucket/img.jpg', file_client_args={
...     'backend': 'petrel'})
```

mmcv.image.**lut_transform**(*img*, *lut_table*)
Transform array by look-up table.

The function lut_transform fills the output array with values from the look-up table. Indices of the entries are taken from the input array.

Parameters

- **img** (*ndarray*) – Image to be transformed.
- **lut_table** (*ndarray*) – look-up table of 256 elements; in case of multi-channel input array, the table should either have a single channel (in this case the same table is used for all channels) or the same number of channels as in the input array.

Returns The transformed image.

Return type ndarray

mmcv.image.**posterize**(*img*, *bits*)
Posterize an image (reduce the number of bits for each color channel)

Parameters

- **img** (*ndarray*) – Image to be posterized.
- **bits** (*int*) – Number of bits (1 to 8) to use for posterizing.

Returns The posterized image.

Return type ndarray

mmcv.image.**rescale_size**(*old_size: tuple*, *scale: Union[float, int, tuple]*, *return_scale: bool = False*) → tuple
Calculate the new size to be rescaled to.

Parameters

- **old_size** (*tuple[int]*) – The old size (w, h) of image.
- **scale** (*float | tuple[int]*) – The scaling factor or maximum size. If it is a float number, then the image will be rescaled by this factor, else if it is a tuple of 2 integers, then the image will be rescaled as large as possible within the scale.

- **return_scale** (*bool*) – Whether to return the scaling factor besides the rescaled image size.

Returns The new rescaled image size.

Return type tuple[int]

mmcv.image.**rgb2bgr**(*img: numpy.ndarray*) → numpy.ndarray

Convert a RGB image to BGR image.

Parameters **img** (*ndarray or str*) – The input image.

Returns The converted BGR image.

Return type ndarray

mmcv.image.**rgb2gray**(*img: numpy.ndarray*, *keepdim: bool = False*) → numpy.ndarray
Convert a RGB image to grayscale image.

Parameters

- **img** (*ndarray*) – The input image.
- **keepdim** (*bool*) – If False (by default), then return the grayscale image with 2 dims, otherwise 3 dims.

Returns The converted grayscale image.

Return type ndarray

mmcv.image.**rgb2ycbcr**(*img: numpy.ndarray*, *y_only: bool = False*) → numpy.ndarray
Convert a RGB image to YCbCr image.

This function produces the same results as Matlab's *rgb2ycbcr* function. It implements the ITU-R BT.601 conversion for standard-definition television. See more details in https://en.wikipedia.org/wiki/YCbCr#ITU-R_BT.601_conversion.

It differs from a similar function in cv2.cvtColor: *RGB <-> YCrCb*. In OpenCV, it implements a JPEG conversion. See more details in https://en.wikipedia.org/wiki/YCbCr#JPEG_conversion.

Parameters

- **img** (*ndarray*) – The input image. It accepts: 1. np.uint8 type with range [0, 255]; 2. np.float32 type with range [0, 1].
- **y_only** (*bool*) – Whether to only return Y channel. Default: False.

Returns The converted YCbCr image. The output image has the same type and range as input image.

Return type ndarray

mmcv.image.**solarize**(*img*, *thr=128*)
Solarize an image (invert all pixel values above a threshold)

Parameters

- **img** (*ndarray*) – Image to be solarized.
- **thr** (*int*) – Threshold for solarizing (0 - 255).

Returns The solarized image.

Return type ndarray

mmcv.image.**tensor2imgs**(*tensor*, *mean: Optional[tuple] = None*, *std: Optional[tuple] = None*, *to_rgb: bool = True*) → list

Convert tensor to 3-channel images or 1-channel gray images.

Parameters

- **tensor** (*torch.Tensor*) – Tensor that contains multiple images, shape (N, C, H, W). C can be either 3 or 1.
- **mean** (*tuple[float], optional*) – Mean of images. If None, (0, 0, 0) will be used for tensor with 3-channel, while (0,) for tensor with 1-channel. Defaults to None.
- **std** (*tuple[float], optional*) – Standard deviation of images. If None, (1, 1, 1) will be used for tensor with 3-channel, while (1,) for tensor with 1-channel. Defaults to None.
- **to_rgb** (*bool, optional*) – Whether the tensor was converted to RGB format in the first place. If so, convert it back to BGR. For the tensor with 1 channel, it must be False. Defaults to True.

Returns A list that contains multiple images.

Return type list[np.ndarray]

mmcv.image.**use_backend**(*backend: str*) → None

Select a backend for image decoding.

Parameters

- **backend** (*str*) – The image decoding backend type. Options are *cv2*,
- **pillow** – //github.com/lilohuang/PyTurboJPEG)
- **(see https** (*turbojpeg*) – //github.com/lilohuang/PyTurboJPEG)
- **tifffile. turbojpeg is faster but it only supports .jpeg** (*and*) –
- **format.** (*file*) –

mmcv.image.**ycbcr2bgr**(*img: numpy.ndarray*) → numpy.ndarray

Convert a YCbCr image to BGR image.

The bgr version of ycbcr2rgb. It implements the ITU-R BT.601 conversion for standard-definition television. See more details in https://en.wikipedia.org/wiki/YCbCr#ITU-R_BT.601_conversion.

It differs from a similar function in cv2.cvtColor: *YCrCb <-> BGR*. In OpenCV, it implements a JPEG conversion. See more details in https://en.wikipedia.org/wiki/YCbCr#JPEG_conversion.

Parameters **img** (*ndarray*) – The input image. It accepts: 1. np.uint8 type with range [0, 255]; 2. np.float32 type with range [0, 1].

Returns The converted BGR image. The output image has the same type and range as input image.

Return type ndarray

mmcv.image.**ycbcr2rgb**(*img: numpy.ndarray*) → numpy.ndarray

Convert a YCbCr image to RGB image.

This function produces the same results as Matlab's ycbcr2rgb function. It implements the ITU-R BT.601 conversion for standard-definition television. See more details in https://en.wikipedia.org/wiki/YCbCr#ITU-R_BT.601_conversion.

It differs from a similar function in cv2.cvtColor: *YCrCb <-> RGB*. In OpenCV, it implements a JPEG conversion. See more details in https://en.wikipedia.org/wiki/YCbCr#JPEG_conversion.

Parameters **img** (*ndarray*) – The input image. It accepts: 1. np.uint8 type with range [0, 255]; 2. np.float32 type with range [0, 1].

Returns The converted RGB image. The output image has the same type and range as input image.

Return type ndarray

CHAPTER

TWENTYEIGHT

VIDEO

class mmcv.video.**VideoReader**(*filename*, *cache_capacity=10*)

Video class with similar usage to a list object.

This video warpper class provides convenient apis to access frames. There exists an issue of OpenCV's VideoCapture class that jumping to a certain frame may be inaccurate. It is fixed in this class by checking the position after jumping each time. Cache is used when decoding videos. So if the same frame is visited for the second time, there is no need to decode again if it is stored in the cache.

Examples

```
>>> import mmcv
>>> v = mmcv.VideoReader('sample.mp4')
>>> len(v)  # get the total frame number with `len()`
120
>>> for img in v:  # v is iterable
>>>     mmcv.imshow(img)
>>> v[5]  # get the 6th frame
```

current_frame()

Get the current frame (frame that is just visited).

Returns If the video is fresh, return None, otherwise return the frame.

Return type ndarray or None

cvt2frames(*frame_dir*, *file_start=0*, *filename_tmpl='{:06d}.jpg'*, *start=0*, *max_num=0*, *show_progress=True*)

Convert a video to frame images.

Parameters

- **frame_dir** (*str*) – Output directory to store all the frame images.
- **file_start** (*int*) – Filenames will start from the specified number.
- **filename_tmpl** (*str*) – Filename template with the index as the placeholder.
- **start** (*int*) – The starting frame index.
- **max_num** (*int*) – Maximum number of frames to be written.
- **show_progress** (*bool*) – Whether to show a progress bar.

property fourcc

"Four character code" of the video.

Type str

property fps
FPS of the video.

Type float

property frame_cnt
Total frames of the video.

Type int

get_frame(*frame_id*)
Get frame by index.

Parameters **frame_id** (*int*) – Index of the expected frame, 0-based.

Returns Return the frame if successful, otherwise None.

Return type ndarray or None

property height
Height of video frames.

Type int

property opened
Indicate whether the video is opened.

Type bool

property position
Current cursor position, indicating frame decoded.

Type int

read()
Read the next frame.

If the next frame have been decoded before and in the cache, then return it directly, otherwise decode, cache and return it.

Returns Return the frame if successful, otherwise None.

Return type ndarray or None

property resolution
Video resolution (width, height).

Type tuple

property vcap
The raw VideoCapture object.

Type cv2.VideoCapture

property width
Width of video frames.

Type int

mmcv.video.**concat_video**(*video_list: List*, *out_file: str*, *vcodec: Optional[str] = None*, *acodec: Optional[str] = None*, *log_level: str = 'info'*, *print_cmd: bool = False*) → None
Concatenate multiple videos into a single one.

Parameters

- **video_list** (*list*) – A list of video filenames
- **out_file** (*str*) – Output video filename
- **vcodec** (*None or str*) – Output video codec, None for unchanged
- **acodec** (*None or str*) – Output audio codec, None for unchanged
- **log_level** (*str*) – Logging level of ffmpeg.
- **print_cmd** (*bool*) – Whether to print the final ffmpeg command.

mmcv.video.**convert_video**(*in_file: str*, *out_file: str*, *print_cmd: bool = False*, *pre_options: str = ''*, ***kwargs*) → None

Convert a video with ffmpeg.

This provides a general api to ffmpeg, the executed command is:

```
`ffmpeg -y <pre_options> -i <in_file> <options> <out_file>`
```

Options(kwargs) are mapped to ffmpeg commands with the following rules:

- key=val: "-key val"
- key=True: "-key"
- key=False: ""

Parameters

- **in_file** (*str*) – Input video filename.
- **out_file** (*str*) – Output video filename.
- **pre_options** (*str*) – Options appears before "-i <in_file>".
- **print_cmd** (*bool*) – Whether to print the final ffmpeg command.

mmcv.video.**cut_video**(*in_file: str*, *out_file: str*, *start: Optional[float] = None*, *end: Optional[float] = None*, *vcodec: Optional[str] = None*, *acodec: Optional[str] = None*, *log_level: str = 'info'*, *print_cmd: bool = False*) → None

Cut a clip from a video.

Parameters

- **in_file** (*str*) – Input video filename.
- **out_file** (*str*) – Output video filename.
- **start** (*None or float*) – Start time (in seconds).
- **end** (*None or float*) – End time (in seconds).
- **vcodec** (*None or str*) – Output video codec, None for unchanged.
- **acodec** (*None or str*) – Output audio codec, None for unchanged.
- **log_level** (*str*) – Logging level of ffmpeg.
- **print_cmd** (*bool*) – Whether to print the final ffmpeg command.

mmcv.video.**dequantize_flow**(*dx: numpy.ndarray*, *dy: numpy.ndarray*, *max_val: float = 0.02*, *denorm: bool = True*) → numpy.ndarray

Recover from quantized flow.

Parameters

- **dx** (*ndarray*) – Quantized dx.
- **dy** (*ndarray*) – Quantized dy.
- **max_val** (*float*) – Maximum value used when quantizing.
- **denorm** (*bool*) – Whether to multiply flow values with width/height.

Returns Dequantized flow.

Return type ndarray

mmcv.video.**flow_from_bytes**(*content: bytes*) → numpy.ndarray
Read dense optical flow from bytes.

Note: This load optical flow function works for FlyingChairs, FlyingThings3D, Sintel, FlyingChairsOcc datasets, but cannot load the data from ChairsSDHom.

Parameters **content** (*bytes*) – Optical flow bytes got from files or other streams.

Returns Loaded optical flow with the shape (H, W, 2).

Return type ndarray

mmcv.video.**flow_warp**(*img: numpy.ndarray*, *flow: numpy.ndarray*, *filling_value: int = 0*, *interpolate_mode: str = 'nearest'*) → numpy.ndarray
Use flow to warp img.

Parameters

- **img** (*ndarray*) – Image to be warped.
- **flow** (*ndarray*) – Optical Flow.
- **filling_value** (*int*) – The missing pixels will be set with filling_value.
- **interpolate_mode** (*str*) – bilinear -> Bilinear Interpolation; nearest -> Nearest Neighbor.

Returns Warped image with the same shape of img

Return type ndarray

mmcv.video.**flowread**(*flow_or_path: Union[numpy.ndarray, str]*, *quantize: bool = False*, *concat_axis: int = 0*, **args*, ***kwargs*) → numpy.ndarray
Read an optical flow map.

Parameters

- **flow_or_path** (*ndarray or str*) – A flow map or filepath.
- **quantize** (*bool*) – whether to read quantized pair, if set to True, remaining args will be passed to *dequantize_flow()*.
- **concat_axis** (*int*) – The axis that dx and dy are concatenated, can be either 0 or 1. Ignored if quantize is False.

Returns Optical flow represented as a (h, w, 2) numpy array

Return type ndarray

mmcv.video.**flowwrite**(*flow: numpy.ndarray*, *filename: str*, *quantize: bool = False*, *concat_axis: int = 0*, **args*, ***kwargs*) → None
Write optical flow to file.

If the flow is not quantized, it will be saved as a .flo file losslessly, otherwise a jpeg image which is lossy but of much smaller size. (dx and dy will be concatenated horizontally into a single image if quantize is True.)

Parameters

- **flow** (*ndarray*) – (h, w, 2) array of optical flow.
- **filename** (*str*) – Output filepath.
- **quantize** (*bool*) – Whether to quantize the flow and save it to 2 jpeg images. If set to True, remaining args will be passed to *quantize_flow()*.
- **concat_axis** (*int*) – The axis that dx and dy are concatenated, can be either 0 or 1. Ignored if quantize is False.

mmcv.video.**frames2video**(*frame_dir: str*, *video_file: str*, *fps: float = 30*, *fourcc: str = 'XVID'*, *filename_tmpl: str = '{:06d}.jpg'*, *start: int = 0*, *end: int = 0*, *show_progress: bool = True*) → None

Read the frame images from a directory and join them as a video.

Parameters

- **frame_dir** (*str*) – The directory containing video frames.
- **video_file** (*str*) – Output filename.
- **fps** (*float*) – FPS of the output video.
- **fourcc** (*str*) – Fourcc of the output video, this should be compatible with the output file type.
- **filename_tmpl** (*str*) – Filename template with the index as the variable.
- **start** (*int*) – Starting frame index.
- **end** (*int*) – Ending frame index.
- **show_progress** (*bool*) – Whether to show a progress bar.

mmcv.video.**quantize_flow**(*flow: numpy.ndarray*, *max_val: float = 0.02*, *norm: bool = True*) → tuple

Quantize flow to [0, 255].

After this step, the size of flow will be much smaller, and can be dumped as jpeg images.

Parameters

- **flow** (*ndarray*) – (h, w, 2) array of optical flow.
- **max_val** (*float*) – Maximum value of flow, values beyond [-max_val, max_val] will be truncated.
- **norm** (*bool*) – Whether to divide flow values by image width/height.

Returns Quantized dx and dy.

Return type tuple[ndarray]

mmcv.video.**resize_video**(*in_file: str*, *out_file: str*, *size: Optional[tuple] = None*, *ratio: Optional[Union[tuple, float]] = None*, *keep_ar: bool = False*, *log_level: str = 'info'*, *print_cmd: bool = False*) → None

Resize a video.

Parameters

- **in_file** (*str*) – Input video filename.
- **out_file** (*str*) – Output video filename.
- **size** (*tuple*) – Expected size (w, h), eg, (320, 240) or (320, -1).

- **ratio** (*tuple or float*) – Expected resize ratio, (2, 0.5) means (w*2, h*0.5).
- **keep_ar** (*bool*) – Whether to keep original aspect ratio.
- **log_level** (*str*) – Logging level of ffmpeg.
- **print_cmd** (*bool*) – Whether to print the final ffmpeg command.

mmcv.video.**sparse_flow_from_bytes**(*content: bytes*) → Tuple[numpy.ndarray, numpy.ndarray]

Read the optical flow in KITTI datasets from bytes.

This function is modified from RAFT load the KITTI datasets.

Parameters **content** (*bytes*) – Optical flow bytes got from files or other streams.

Returns Loaded optical flow with the shape (H, W, 2) and flow valid mask with the shape (H, W).

Return type Tuple(ndarray, ndarray)

CHAPTER

TWENTYNINE

ARRAYMISC

mmcv.arraymisc.**dequantize**(*arr: numpy.ndarray, min_val: Union[int, float], max_val: Union[int, float], levels: int, dtype=<class 'numpy.float64'>*) → tuple

Dequantize an array.

Parameters

- **arr** (*ndarray*) – Input array.
- **min_val** (*int or float*) – Minimum value to be clipped.
- **max_val** (*int or float*) – Maximum value to be clipped.
- **levels** (*int*) – Quantization levels.
- **dtype** (*np.type*) – The type of the dequantized array.

Returns Dequantized array.

Return type tuple

mmcv.arraymisc.**quantize**(*arr: numpy.ndarray, min_val: Union[int, float], max_val: Union[int, float], levels: int, dtype=<class 'numpy.int64'>*) → tuple

Quantize an array of (-inf, inf) to [0, levels-1].

Parameters

- **arr** (*ndarray*) – Input array.
- **min_val** (*int or float*) – Minimum value to be clipped.
- **max_val** (*int or float*) – Maximum value to be clipped.
- **levels** (*int*) – Quantization levels.
- **dtype** (*np.type*) – The type of the quantized array.

Returns Quantized array.

Return type tuple

CHAPTER

THIRTY

VISUALIZATION

class mmcv.visualization.**Color**(*value*)
An enum that defines common colors.

Contains red, green, blue, cyan, yellow, magenta, white and black.

mmcv.visualization.**color_val**(*color: Union[*mmcv.visualization.color.Color*, str, tuple, int, numpy.ndarray]*) → tuple
Convert various input to color tuples.

Parameters **color** (*Color*/str/tuple/int/ndarray) – Color inputs

Returns A tuple of 3 integers indicating BGR channels.

Return type tuple[int]

mmcv.visualization.**flow2rgb**(*flow: numpy.ndarray*, *color_wheel: Optional[numpy.ndarray] = None*, *unknown_thr: float = 1000000.0*) → numpy.ndarray
Convert flow map to RGB image.

Parameters

- **flow** (*ndarray*) – Array of optical flow.
- **color_wheel** (*ndarray or None*) – Color wheel used to map flow field to RGB colorspace. Default color wheel will be used if not specified.
- **unknown_thr** (*float*) – Values above this threshold will be marked as unknown and thus ignored.

Returns RGB image that can be visualized.

Return type ndarray

mmcv.visualization.**flowshow**(*flow: Union[numpy.ndarray, str]*, *win_name: str = ''*, *wait_time: int = 0*) → None
Show optical flow.

Parameters

- **flow** (*ndarray or str*) – The optical flow to be displayed.
- **win_name** (*str*) – The window name.
- **wait_time** (*int*) – Value of waitKey param.

mmcv.visualization.**imshow**(*img: Union[str, numpy.ndarray]*, *win_name: str = ''*, *wait_time: int = 0*)
Show an image.

Parameters

- **img** (*str or ndarray*) – The image to be displayed.

- **win_name** (*str*) – The window name.
- **wait_time** (*int*) – Value of waitKey param.

mmcv.visualization.**imshow_bboxes**(*img: Union[str, numpy.ndarray]*, *bboxes: Union[list, numpy.ndarray]*, *colors: Union[*mmcv.visualization.color.Color*, str, tuple, int, numpy.ndarray] = 'green'*, *top_k: int = - 1*, *thickness: int = 1*, *show: bool = True*, *win_name: str = ''*, *wait_time: int = 0*, *out_file: Optional[str] = None*)

Draw bboxes on an image.

Parameters

- **img** (*str or ndarray*) – The image to be displayed.
- **bboxes** (*list or ndarray*) – A list of ndarray of shape (k, 4).
- **colors** (Color *or str or tuple or int or ndarray*) – A list of colors.
- **top_k** (*int*) – Plot the first k bboxes only if set positive.
- **thickness** (*int*) – Thickness of lines.
- **show** (*bool*) – Whether to show the image.
- **win_name** (*str*) – The window name.
- **wait_time** (*int*) – Value of waitKey param.
- **out_file** (*str, optional*) – The filename to write the image.

Returns The image with bboxes drawn on it.

Return type ndarray

mmcv.visualization.**imshow_det_bboxes**(*img: Union[str, numpy.ndarray]*, *bboxes: numpy.ndarray*, *labels: numpy.ndarray*, *class_names: Optional[List[str]] = None*, *score_thr: float = 0*, *bbox_color: Union[*mmcv.visualization.color.Color*, str, tuple, int, numpy.ndarray] = 'green'*, *text_color: Union[*mmcv.visualization.color.Color*, str, tuple, int, numpy.ndarray] = 'green'*, *thickness: int = 1*, *font_scale: float = 0.5*, *show: bool = True*, *win_name: str = ''*, *wait_time: int = 0*, *out_file: Optional[str] = None*)

Draw bboxes and class labels (with scores) on an image.

Parameters

- **img** (*str or ndarray*) – The image to be displayed.
- **bboxes** (*ndarray*) – Bounding boxes (with scores), shaped (n, 4) or (n, 5).
- **labels** (*ndarray*) – Labels of bboxes.
- **class_names** (*list[str]*) – Names of each classes.
- **score_thr** (*float*) – Minimum score of bboxes to be shown.
- **bbox_color** (Color *or str or tuple or int or ndarray*) – Color of bbox lines.
- **text_color** (Color *or str or tuple or int or ndarray*) – Color of texts.
- **thickness** (*int*) – Thickness of lines.
- **font_scale** (*float*) – Font scales of texts.
- **show** (*bool*) – Whether to show the image.

- **win_name** (*str*) – The window name.
- **wait_time** (*int*) – Value of waitKey param.
- **out_file** (*str or None*) – The filename to write the image.

Returns The image with bboxes drawn on it.

Return type ndarray

mmcv.visualization.**make_color_wheel**(*bins: Optional[Union[list, tuple]] = None*) → numpy.ndarray
Build a color wheel.

Parameters **bins** (*list or tuple, optional*) – Specify the number of bins for each color range, corresponding to six ranges: red -> yellow, yellow -> green, green -> cyan, cyan -> blue, blue -> magenta, magenta -> red. [15, 6, 4, 11, 13, 6] is used for default (see Middlebury).

Returns Color wheel of shape (total_bins, 3).

Return type ndarray

CHAPTER

THIRTYONE

UTILS

class mmcv.utils.**BuildExtension**(**args*, ***kwargs*)

A custom `setuptools` build extension .

This `setuptools.build_ext` subclass takes care of passing the minimum required compiler flags (e.g. `-std=c++14`) as well as mixed C++/CUDA compilation (and support for CUDA files in general).

When using *BuildExtension*, it is allowed to supply a dictionary for `extra_compile_args` (rather than the usual list) that maps from languages (`cxx` or `nvcc`) to a list of additional compiler flags to supply to the compiler. This makes it possible to supply different flags to the C++ and CUDA compiler during mixed compilation.

`use_ninja` (bool): If `use_ninja` is `True` (default), then we attempt to build using the Ninja backend. Ninja greatly speeds up compilation compared to the standard `setuptools.build_ext`. Fallbacks to the standard distutils backend if Ninja is not available.

Note: By default, the Ninja backend uses #CPUS + 2 workers to build the extension. This may use up too many resources on some systems. One can control the number of workers by setting the *MAX_JOBS* environment variable to a non-negative number.

---

finalize_options() → None

Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if 'foo' depends on 'bar', then it is safe to set 'foo' from 'bar' as long as 'foo' still has the same value it was assigned in 'initialize_options()'.

This method must be implemented by all command classes.

get_ext_filename(*ext_name*)

Convert the name of an extension (eg. "foo.bar") into the name of the file from which it will be loaded (eg. "foo/bar.so", or "foobar.pyd").

classmethod with_options(***options*)

Returns a subclass with alternative constructor that extends any original keyword arguments to the original constructor with the given options.

mmcv.utils.**CUDAExtension**(*name*, *sources*, **args*, ***kwargs*)

Creates a `setuptools.Extension` for CUDA/C++.

Convenience method that creates a `setuptools.Extension` with the bare minimum (but often sufficient) arguments to build a CUDA/C++ extension. This includes the CUDA include path, library path and runtime library.

All arguments are forwarded to the `setuptools.Extension` constructor.

Example

```
>>> # xdoctest: +SKIP
>>> from setuptools import setup
>>> from torch.utils.cpp_extension import BuildExtension, CUDAExtension
>>> setup(
...     name='cuda_extension',
...     ext_modules=[
...         CUDAExtension(
...                 name='cuda_extension',
...                 sources=['extension.cpp', 'extension_kernel.cu'],
...                 extra_compile_args={'cxx': ['-g'],
...                                     'nvcc': ['-O2']})
...     ],
...     cmdclass={
...         'build_ext': BuildExtension
...     })
```

Compute capabilities:

By default the extension will be compiled to run on all archs of the cards visible during the building process of the extension, plus PTX. If down the road a new card is installed the extension may need to be recompiled. If a visible card has a compute capability (CC) that's newer than the newest version for which your nvcc can build fully-compiled binaries, Pytorch will make nvcc fall back to building kernels with the newest version of PTX your nvcc does support (see below for details on PTX).

You can override the default behavior using *TORCH_CUDA_ARCH_LIST* to explicitly specify which CCs you want the extension to support:

TORCH_CUDA_ARCH_LIST="6.1 8.6" python build_my_extension.py TORCH_CUDA_ARCH_LIST="5.2 6.0 6.1 7.0 7.5 8.0 8.6+PTX" python build_my_extension.py

The +PTX option causes extension kernel binaries to include PTX instructions for the specified CC. PTX is an intermediate representation that allows kernels to runtime-compile for any CC >= the specified CC (for example, 8.6+PTX generates PTX that can runtime-compile for any GPU with CC >= 8.6). This improves your binary's forward compatibility. However, relying on older PTX to provide forward compat by runtime-compiling for newer CCs can modestly reduce performance on those newer CCs. If you know exact CC(s) of the GPUs you want to target, you're always better off specifying them individually. For example, if you want your extension to run on 8.0 and 8.6, "8.0+PTX" would work functionally because it includes PTX that can runtime-compile for 8.6, but "8.0 8.6" would be better.

Note that while it's possible to include all supported archs, the more archs get included the slower the building process will be, as it will build a separate kernel image for each arch.

Note that CUDA-11.5 nvcc will hit internal compiler error while parsing torch/extension.h on Windows. To workaround the issue, move python binding logic to pure C++ file.

Example use:

```
>>> # xdoctest: +SKIP
>>> #include <ATen/ATen.h>
>>> at::Tensor SigmoidAlphaBlendForwardCuda(....)
```

Instead of:

```
>>> # xdoctest: +SKIP
>>> #include <torch/extension.h>
>>> torch::Tensor SigmoidAlphaBlendForwardCuda(...)
```

Currently open issue for nvcc bug: https://github.com/pytorch/pytorch/issues/69460 Complete workaround code example: https://github.com/facebookresearch/pytorch3d/commit/cb170ac024a949f1f9614ffe6af1c38d972f7d48

Relocatable device code linking:

If you want to reference device symbols across compilation units (across object files), the object files need to be built with *relocatable device code* (-rdc=true or -dc). An exception to this rule is "dynamic parallelism" (nested kernel launches) which is not used a lot anymore. *Relocatable device code* is less optimized so it needs to be used only on object files that need it. Using *-dlto* (Device Link Time Optimization) at the device code compilation step and *dlink* step help reduce the protentional perf degradation of *-rdc*. Note that it needs to be used at both steps to be useful.

If you have *rdc* objects you need to have an extra *-dlink* (device linking) step before the CPU symbol linking step. There is also a case where *-dlink* is used without *-rdc*: when an extension is linked against a static lib containing rdc-compiled objects like the [NVSHMEM library](https://developer.nvidia.com/nvshmem).

Note: Ninja is required to build a CUDA Extension with RDC linking.

Example

```
>>> # xdoctest: +SKIP
>>> CUDAExtension(
...         name='cuda_extension',
...         sources=['extension.cpp', 'extension_kernel.cu'],
...         dlink=True,
...         dlink_libraries=["dlink_lib"],
...         extra_compile_args={'cxx': ['-g'],
...                             'nvcc': ['-O2', '-rdc=true']})
```

class mmcv.utils.**Config**(*cfg_dict=None*, *cfg_text=None*, *filename=None*)

A facility for config and config files.

It supports common file formats as configs: python/json/yaml. The interface is the same as a dict object and also allows access config values as attributes.

Example

```
>>> cfg = Config(dict(a=1, b=dict(b1=[0, 1])))
>>> cfg.a
1
>>> cfg.b
{'b1': [0, 1]}
>>> cfg.b.b1
[0, 1]
>>> cfg = Config.fromfile('tests/data/config/a.py')
>>> cfg.filename
"/home/kchen/projects/mmcv/tests/data/config/a.py"
>>> cfg.item4
```

(continues on next page)

(continued from previous page)

```
'test'
>>> cfg
"Config [path: /home/kchen/projects/mmcv/tests/data/config/a.py]: "
"{'item1': [1, 2], 'item2': {'a': 0}, 'item3': True, 'item4': 'test'}"
```

static auto_argparser(*description=None*)
Generate argparser from config file automatically (experimental)

dump(*file=None*)
Dumps config into a file or returns a string representation of the config.

If a file argument is given, saves the config to that file using the format defined by the file argument extension.

Otherwise, returns a string representing the config. The formatting of this returned string is defined by the extension of *self.filename*. If *self.filename* is not defined, returns a string representation of a

dict (lowercased and using ‘ for strings).

Examples

```
>>> cfg_dict = dict(item1=[1, 2], item2=dict(a=0),
...     item3=True, item4='test')
>>> cfg = Config(cfg_dict=cfg_dict)
>>> dump_file = "a.py"
>>> cfg.dump(dump_file)
```

Parameters **file** (*str, optional*) – Path of the output file where the config will be dumped. Defaults to None.

static fromstring(*cfg_str*, *file_format*)
Generate config from config str.

Parameters

- **cfg_str** (*str*) – Config str.
- **file_format** (*str*) – Config file format corresponding to the config str. Only py/yml/yaml/json type are supported now!

Returns Config obj.

Return type *Config*

merge_from_dict(*options*, *allow_list_keys=True*)
Merge list into cfg_dict.

Merge the dict parsed by MultipleKVAction into this cfg.

Examples

```
>>> options = {'model.backbone.depth': 50,
...            'model.backbone.with_cp':True}
>>> cfg = Config(dict(model=dict(backbone=dict(type='ResNet'))))
>>> cfg.merge_from_dict(options)
>>> cfg_dict = super(Config, self).__getattribute__('_cfg_dict')
>>> assert cfg_dict == dict(
...     model=dict(backbone=dict(depth=50, with_cp=True)))
```

```
>>> # Merge list element
>>> cfg = Config(dict(pipeline=[
...     dict(type='LoadImage'), dict(type='LoadAnnotations')]))
>>> options = dict(pipeline={'0': dict(type='SelfLoadImage')})
>>> cfg.merge_from_dict(options, allow_list_keys=True)
>>> cfg_dict = super(Config, self).__getattribute__('_cfg_dict')
>>> assert cfg_dict == dict(pipeline=[
...     dict(type='SelfLoadImage'), dict(type='LoadAnnotations')])
```

Parameters

- **options** (*dict*) – dict of configs to merge from.
- **allow_list_keys** (*bool*) – If True, int string keys (e.g. '0', '1') are allowed in `options` and will replace the element of the corresponding index in the config if the config is a list. Default: True.

class mmcv.utils.**ConfigDict**(**args*, ***kwargs*)

mmcv.utils.**CppExtension**(*name*, *sources*, **args*, ***kwargs*)

Creates a `setuptools.Extension` for C++.

Convenience method that creates a `setuptools.Extension` with the bare minimum (but often sufficient) arguments to build a C++ extension.

All arguments are forwarded to the `setuptools.Extension` constructor.

Example

```
>>> # xdoctest: +SKIP
>>> from setuptools import setup
>>> from torch.utils.cpp_extension import BuildExtension, CppExtension
>>> setup(
...     name='extension',
...     ext_modules=[
...         CppExtension(
...             name='extension',
...             sources=['extension.cpp'],
...             extra_compile_args=['-g']),
...     ],
...     cmdclass={
...         'build_ext': BuildExtension
...     })
```

class mmcv.utils.**DataLoader**(*dataset: torch.utils.data.dataset.Dataset[torch.utils.data.dataloader.T_co]*, *batch_size: Optional[int] = 1*, *shuffle: Optional[bool] = None*, *sampler: Optional[Union[torch.utils.data.sampler.Sampler, Iterable]] = None*, *batch_sampler: Optional[Union[torch.utils.data.sampler.Sampler[Sequence], Iterable[Sequence]]] = None*, *num_workers: int = 0*, *collate_fn: Optional[Callable[[List[torch.utils.data.dataloader.T]], Any]] = None*, *pin_memory: bool = False*, *drop_last: bool = False*, *timeout: float = 0*, *worker_init_fn: Optional[Callable[[int], None]] = None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor: int = 2*, *persistent_workers: bool = False*, *pin_memory_device: str = ''*)

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: `1`).
- **shuffle** (*bool, optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like `sampler`, but returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)
- **collate_fn** (*Callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (*bool, optional*) – If `True`, the data loader will copy Tensors into device/CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.
- **drop_last** (*bool, optional*) – set to `True` to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. (default: `False`)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: `0`)
- **worker_init_fn** (*Callable, optional*) – If not `None`, this will be called on each worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. (default: `None`)
- **generator** (*torch.Generator, optional*) – If not `None`, this RNG will be used by RandomSampler to generate random indexes and multiprocessing to generate *base_seed* for workers. (default: `None`)
- **prefetch_factor** (*int, optional, keyword-only arg*) – Number of batches loaded in advance by each worker. `2` means there will be a total of 2 * num_workers batches prefetched across all workers. (default: `2`)

- **persistent_workers** (*bool, optional*) – If `True`, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. (default: `False`)
- **pin_memory_device** (*str, optional*) – the data loader will copy Tensors into device pinned memory before returning them if pin_memory is set to true.

Warning: If the `spawn` start method is used, *worker_init_fn* cannot be an unpicklable object, e.g., a lambda function. See multiprocessing-best-practices on more details related to multiprocessing in PyTorch.

Warning: `len(dataloader)` heuristic is based on the length of the sampler used. When `dataset` is an `IterableDataset`, it instead returns an estimate based on `len(dataset) / batch_size`, with proper rounding depending on `drop_last`, regardless of multi-process loading configurations. This represents the best guess PyTorch can make because PyTorch trusts user `dataset` code in correctly handling multi-process loading to avoid duplicate data.

However, if sharding results in multiple workers having incomplete last batches, this estimate can still be inaccurate, because (1) an otherwise complete batch can be broken into multiple ones and (2) more than one batch worth of samples can be dropped when `drop_last` is set. Unfortunately, PyTorch can not detect such cases in general.

See **`Dataset Types`_** for more details on these two types of datasets and how `IterableDataset` interacts with **`Multi-process data loading`_**.

Warning: See reproducibility, and dataloader-workers-random-seed, and data-loading-randomness notes for random seed related questions.

class mmcv.utils.**DictAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

argparse action to split an argument into KEY=VALUE form on the first = and append to a dictionary. List options can be passed as comma separated values, i.e 'KEY=V1,V2,V3', or with explicit brackets, i.e. 'KEY=[V1,V2,V3]'. It also support nested brackets to build list/tuple values. e.g. 'KEY=[(V1,V2),(V3,V4)]'

mmcv.utils.**PoolDataLoader**

alias of *torch.utils.data.dataloader.DataLoader*

class mmcv.utils.**ProgressBar**(*task_num=0*, *bar_width=50*, *start=True*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

A progress bar which can print the progress.

class mmcv.utils.**Registry**(*name*, *build_func=None*, *parent=None*, *scope=None*)

A registry to map strings to classes or functions.

Registered object could be built from registry. Meanwhile, registered functions could be called from registry.

Example

```
>>> MODELS = Registry('models')
>>> @MODELS.register_module()
>>> class ResNet:
>>>     pass
>>> resnet = MODELS.build(dict(type='ResNet'))
>>> @MODELS.register_module()
>>> def resnet50():
>>>     pass
>>> resnet = MODELS.build(dict(type='resnet50'))
```

Please refer to https://mmcv.readthedocs.io/en/latest/understand_mmcv/registry.html for advanced usage.

Parameters

- **name** (*str*) – Registry name.
- **build_func** (*func, optional*) – Build function to construct instance from Registry, func:*build_from_cfg* is used if neither `parent` or `build_func` is specified. If `parent` is specified and `build_func` is not given, `build_func` will be inherited from `parent`. Default: None.
- **parent** (`Registry`, *optional*) – Parent registry. The class registered in children registry could be built from parent. Default: None.
- **scope** (*str, optional*) – The scope of registry. It is the key to search for children registry. If not specified, scope will be the name of the package where class is defined, e.g. mmdet, mmcls, mmseg. Default: None.

get(*key*)

Get the registry record.

Parameters **key** (*str*) – The class name in string format.

Returns The corresponding class.

Return type class

static infer_scope()

Infer the scope of registry.

The name of the package where registry is defined will be returned.

Example

```
>>> # in mmdet/models/backbone/resnet.py
>>> MODELS = Registry('models')
>>> @MODELS.register_module()
>>> class ResNet:
>>>     pass
The scope of ``ResNet`` will be ``mmdet``.
```

Returns The inferred scope name.

Return type str

register_module(*name=None*, *force=False*, *module=None*)

Register a module.

A record will be added to *self._module_dict*, whose key is the class name or the specified name, and value is the class itself. It can be used as a decorator or a normal function.

Example

```
>>> backbones = Registry('backbone')
>>> @backbones.register_module()
>>> class ResNet:
>>>     pass
```

```
>>> backbones = Registry('backbone')
>>> @backbones.register_module(name='mnet')
>>> class MobileNet:
>>>     pass
```

```
>>> backbones = Registry('backbone')
>>> class ResNet:
>>>     pass
>>> backbones.register_module(ResNet)
```

Parameters

- **name** (*str | None*) – The module name to be registered. If not specified, the class name will be used.
- **force** (*bool, optional*) – Whether to override an existing class with the same name. Default: False.
- **module** (*type*) – Module class or function to be registered.

static split_scope_key(*key*)

Split scope and key.

The first scope will be split from key.

Examples

```
>>> Registry.split_scope_key('mmdet.ResNet')
'mmdet', 'ResNet'
>>> Registry.split_scope_key('ResNet')
None, 'ResNet'
```

Returns The former element is the first scope of the key, which can be None. The latter is the remaining key.

Return type tuple[str | None, str]

class mmcv.utils.**SyncBatchNorm**(*num_features: int*, *eps: float = 1e-05*, *momentum: float = 0.1*, *affine: bool = True*, *track_running_stats: bool = True*, *process_group: Optional[Any] = None*, *device=None*, *dtype=None*)

class mmcv.utils.**Timer**(*start=True*, *print_tmpl=None*)
A flexible Timer class.

Examples

```
>>> import time
>>> import mmcv
>>> with mmcv.Timer():
>>>     # simulate a code block that will run for 1s
>>>     time.sleep(1)
1.000
>>> with mmcv.Timer(print_tmpl='it takes {:.1f} seconds'):
>>>     # simulate a code block that will run for 1s
>>>     time.sleep(1)
it takes 1.0 seconds
>>> timer = mmcv.Timer()
>>> time.sleep(0.5)
>>> print(timer.since_start())
0.500
>>> time.sleep(0.5)
>>> print(timer.since_last_check())
0.500
>>> print(timer.since_start())
1.000
```

property is_running
indicate whether the timer is running

Type bool

since_last_check()
Time since the last checking.

Either *since_start()* or *since_last_check()* is a checking operation.

Returns Time in seconds.

Return type float

since_start()
Total time since the timer is started.

Returns Time in seconds.

Return type float

start()
Start the timer.

exception mmcv.utils.**TimerError**(*message*)

mmcv.utils.**assert_attrs_equal**(*obj: Any*, *expected_attrs: Dict[str, Any]*) → bool
Check if attribute of class object is correct.

Parameters

- **obj** (*object*) – Class object to be checked.
- **expected_attrs** (*Dict[str, Any]*) – Dict of the expected attrs.

Returns Whether the attribute of class object is correct.

Return type bool

mmcv.utils.**assert_dict_contains_subset**(*dict_obj: Dict[Any, Any]*, *expected_subset: Dict[Any, Any]*) → bool

Check if the dict_obj contains the expected_subset.

Parameters

- **dict_obj** (*Dict[Any, Any]*) – Dict object to be checked.
- **expected_subset** (*Dict[Any, Any]*) – Subset expected to be contained in dict_obj.

Returns Whether the dict_obj contains the expected_subset.

Return type bool

mmcv.utils.**assert_dict_has_keys**(*obj: Dict[str, Any]*, *expected_keys: List[str]*) → bool

Check if the obj has all the expected_keys.

Parameters

- **obj** (*Dict[str, Any]*) – Object to be checked.
- **expected_keys** (*List[str]*) – Keys expected to contained in the keys of the obj.

Returns Whether the obj has the expected keys.

Return type bool

mmcv.utils.**assert_is_norm_layer**(*module*) → bool

Check if the module is a norm layer.

Parameters **module** (*nn.Module*) – The module to be checked.

Returns Whether the module is a norm layer.

Return type bool

mmcv.utils.**assert_keys_equal**(*result_keys: List[str]*, *target_keys: List[str]*) → bool

Check if target_keys is equal to result_keys.

Parameters

- **result_keys** (*List[str]*) – Result keys to be checked.
- **target_keys** (*List[str]*) – Target keys to be checked.

Returns Whether target_keys is equal to result_keys.

Return type bool

mmcv.utils.**assert_params_all_zeros**(*module*) → bool

Check if the parameters of the module is all zeros.

Parameters **module** (*nn.Module*) – The module to be checked.

Returns Whether the parameters of the module is all zeros.

Return type bool

mmcv.utils.**build_from_cfg**(*cfg: Dict*, *registry:* mmcv.utils.registry.Registry, *default_args: Optional[Dict] = None*) → Any

Build a module from config dict when it is a class configuration, or call a function from config dict when it is a function configuration.

Example

```
>>> MODELS = Registry('models')
>>> @MODELS.register_module()
>>> class ResNet:
>>>     pass
>>> resnet = build_from_cfg(dict(type='Resnet'), MODELS)
>>> # Returns an instantiated object
>>> @MODELS.register_module()
>>> def resnet50():
>>>     pass
>>> resnet = build_from_cfg(dict(type='resnet50'), MODELS)
>>> # Return a result of the calling function
```

Parameters

- **cfg** (*dict*) – Config dict. It should at least contain the key "type".
- **registry** (*Registry*) – The registry to search the type from.
- **default_args** (*dict, optional*) – Default initialization arguments.

Returns The constructed object.

Return type object

mmcv.utils.**check_prerequisites**(*prerequisites*, *checker*, *msg_tmpl='Prerequisites "{}" are required in method "{}" but not found, please install them first.'*)

A decorator factory to check if prerequisites are satisfied.

Parameters

- **prerequisites** (*str of list[str]*) – Prerequisites to be checked.
- **checker** (*callable*) – The checker method that returns True if a prerequisite is meet, False otherwise.
- **msg_tmpl** (*str*) – The message template with two variables.

Returns A specific decorator.

Return type decorator

mmcv.utils.**check_python_script**(*cmd*)

Run the python cmd script with *__main__*. The difference between *os.system* is that, this function exectues code in the current process, so that it can be tracked by coverage tools. Currently it supports two forms:

- ./tests/data/scripts/hello.py zz
- python tests/data/scripts/hello.py zz

mmcv.utils.**check_time**(*timer_id*)

Add check points in a single line.

This method is suitable for running a task on a list of items. A timer will be registered when the method is called for the first time.

Examples

```
>>> import time
>>> import mmcv
>>> for i in range(1, 6):
>>>     # simulate a code block
>>>     time.sleep(i)
>>>     mmcv.check_time('task1')
2.000
3.000
4.000
5.000
```

Parameters **str** – Timer identifier.

mmcv.utils.**collect_env**()
Collect the information of the running environments.

Returns

The environment information. The following fields are contained.

- sys.platform: The variable of `sys.platform`.
- Python: Python version.
- CUDA available: Bool, indicating if CUDA is available.
- GPU devices: Device type of each GPU.
- CUDA_HOME (optional): The env var `CUDA_HOME`.
- NVCC (optional): NVCC version.
- GCC: GCC version, "n/a" if GCC is not installed.
- MSVC: Microsoft Virtual C++ Compiler version, Windows only.
- PyTorch: PyTorch version.
- PyTorch compiling details: The output of `torch.__config__.show()`.
- TorchVision (optional): TorchVision version.
- OpenCV: OpenCV version.
- MMCV: MMCV version.
- MMCV Compiler: The GCC version for compiling MMCV ops.
- MMCV CUDA Compiler: The CUDA version for compiling MMCV ops.

Return type dict

mmcv.utils.**concat_list**(*in_list*)
Concatenate a list of list into a single list.

Parameters **in_list** (*list*) – The list of list to be merged.

Returns The concatenated flat list.

Return type list

mmcv.utils.**deprecated_api_warning**(*name_dict*, *cls_name=None*)
A decorator to check if some arguments are deprecate and try to replace deprecate src_arg_name to dst_arg_name.

Parameters **name_dict** (*dict*) – key (str): Deprecate argument names. val (str): Expected argument names.

Returns New function.

Return type func

mmcv.utils.**digit_version**(*version_str: str*, *length: int = 4*)
Convert a version string into a tuple of integers.

This method is usually used for comparing two versions. For pre-release versions: alpha < beta < rc.

Parameters

- **version_str** (*str*) – The version string.
- **length** (*int*) – The maximum number of version levels. Default: 4.

Returns The version info in digits (integers).

Return type tuple[int]

mmcv.utils.**get_git_hash**(*fallback='unknown'*, *digits=None*)
Get the git hash of the current repo.

Parameters

- **fallback** (*str, optional*) – The fallback string when git hash is unavailable. Defaults to 'unknown'.
- **digits** (*int, optional*) – kept digits of the hash. Defaults to None, meaning all digits are kept.

Returns Git commit hash.

Return type str

mmcv.utils.**get_logger**(*name*, *log_file=None*, *log_level=20*, *file_mode='w'*)
Initialize and get a logger by name.

If the logger has not been initialized, this method will initialize the logger by adding one or two handlers, otherwise the initialized logger will be directly returned. During initialization, a StreamHandler will always be added. If *log_file* is specified and the process rank is 0, a FileHandler will also be added.

Parameters

- **name** (*str*) – Logger name.
- **log_file** (*str | None*) – The log filename. If specified, a FileHandler will be added to the logger.
- **log_level** (*int*) – The logger level. Note that only the process of rank 0 is affected, and other processes will set the level to "Error" thus be silent most of the time.
- **file_mode** (*str*) – The file mode used in opening log file. Defaults to 'w'.

Returns The expected logger.

Return type logging.Logger

mmcv.utils.**has_method**(*obj: object*, *method: str*) → bool
Check whether the object has a method.

Parameters

- **method** (*str*) – The method name to check.
- **obj** (*object*) – The object to check.

Returns True if the object has the method else False.

Return type bool

mmcv.utils.**import_modules_from_strings**(*imports*, *allow_failed_imports=False*)
Import modules from the given list of strings.

Parameters

- **imports** (*list | str | None*) – The given module names to be imported.
- **allow_failed_imports** (*bool*) – If True, the failed imports will return None. Otherwise, an ImportError is raise. Default: False.

Returns The imported modules.

Return type list[module] | module | None

Examples

```
>>> osp, sys = import_modules_from_strings(
...     ['os.path', 'sys'])
>>> import os.path as osp_
>>> import sys as sys_
>>> assert osp == osp_
>>> assert sys == sys_
```

mmcv.utils.**is_list_of**(*seq*, *expected_type*)
Check whether it is a list of some type.

A partial method of *is_seq_of()*.

mmcv.utils.**is_method_overridden**(*method*, *base_class*, *derived_class*)
Check if a method of base class is overridden in derived class.

Parameters

- **method** (*str*) – the method name to check.
- **base_class** (*type*) – the class of the base class.
- **derived_class** (*type | Any*) – the class or instance of the derived class.

mmcv.utils.**is_seq_of**(*seq*, *expected_type*, *seq_type=None*)
Check whether it is a sequence of some type.

Parameters

- **seq** (*Sequence*) – The sequence to be checked.
- **expected_type** (*type*) – Expected type of sequence items.
- **seq_type** (*type, optional*) – Expected sequence type.

Returns Whether the sequence is valid.

Return type bool

mmcv.utils.**is_str**(*x*)
Whether the input is an string instance.

Note: This method is deprecated since python 2 is no longer supported.

mmcv.utils.**is_tuple_of**(*seq*, *expected_type*)
Check whether it is a tuple of some type.

A partial method of *is_seq_of()*.

mmcv.utils.**iter_cast**(*inputs*, *dst_type*, *return_type=None*)
Cast elements of an iterable object into some type.

Parameters

- **inputs** (*Iterable*) – The input object.
- **dst_type** (*type*) – Destination type.
- **return_type** (*type, optional*) – If specified, the output object will be converted to this type, otherwise an iterator.

Returns The converted object.

Return type iterator or specified type

mmcv.utils.**list_cast**(*inputs*, *dst_type*)
Cast elements of an iterable object into a list of some type.

A partial method of *iter_cast()*.

mmcv.utils.**load_url**(*url: str*, *model_dir: Optional[str] = None*, *map_location: Optional[Union[Callable[[torch.Tensor, str], torch.Tensor], torch.device, str, Dict[str, str]]] = None*, *progress: bool = True*, *check_hash: bool = False*, *file_name: Optional[str] = None*) → Dict[str, Any]
Loads the Torch serialized object at the given URL.

If downloaded file is a zip file, it will be automatically decompressed.

If the object is already present in *model_dir*, it's deserialized and returned. The default value of `model_dir` is `<hub_dir>/checkpoints` where `hub_dir` is the directory returned by `get_dir()`.

Parameters

- **url** (*str*) – URL of the object to download
- **model_dir** (*str, optional*) – directory in which to save the object
- **map_location** (*optional*) – a function or a dict specifying how to remap storage locations (see torch.load)
- **progress** (*bool, optional*) – whether or not to display a progress bar to stderr. Default: True
- **check_hash** (*bool, optional*) – If True, the filename part of the URL should follow the naming convention `filename-<sha256>.ext` where `<sha256>` is the first eight or more digits of the SHA256 hash of the contents of the file. The hash is used to ensure unique names and to verify the contents of the file. Default: False
- **file_name** (*str, optional*) – name for the downloaded file. Filename from `url` will be used if not set.

Example

```
>>> state_dict = torch.hub.load_state_dict_from_url('https://s3.amazonaws.com/
↪pytorch/models/resnet18-5c106cde.pth')
```

mmcv.utils.**print_log**(*msg*, *logger=None*, *level=20*)
Print a log message.

Parameters

- **msg** (*str*) – The message to be logged.
- **logger** (*logging.Logger | str | None*) – The logger to be used. Some special loggers are:
  - "silent": no message will be printed.
  - other str: the logger obtained with *get_root_logger(logger)*.
  - None: The *print()* method will be used to print log messages.
- **level** (*int*) – Logging level. Only available when *logger* is a Logger object or "root".

mmcv.utils.**requires_executable**(*prerequisites*)
A decorator to check if some executable files are installed.

Example

```
>>> @requires_executable('ffmpeg')
>>> func(arg1, args):
>>>     print(1)
1
```

mmcv.utils.**requires_package**(*prerequisites*)
A decorator to check if some python packages are installed.

Example

```
>>> @requires_package('numpy')
>>> func(arg1, args):
>>>     return numpy.zeros(1)
array([0.])
>>> @requires_package(['numpy', 'non_package'])
>>> func(arg1, args):
>>>     return numpy.zeros(1)
ImportError
```

mmcv.utils.**scandir**(*dir_path*, *suffix=None*, *recursive=False*, *case_sensitive=True*)
Scan a directory to find the interested files.

Parameters

- **dir_path** (str | Path) – Path of the directory.
- **suffix** (*str | tuple(str), optional*) – File suffix that we are interested in. Default: None.

- **recursive** (*bool, optional*) – If set to True, recursively scan the directory. Default: False.
- **case_sensitive** (*bool, optional*) – If set to False, ignore the case of suffix. Default: True.

Returns A generator for all the interested files with relative paths.

mmcv.utils.**slice_list**(*in_list*, *lens*)

Slice a list into several sub lists by a list of given length.

Parameters

- **in_list** (*list*) – The list to be sliced.
- **lens** (*int or list*) – The expected length of each out list.

Returns A list of sliced list.

Return type list

mmcv.utils.**torch_meshgrid**(**tensors*)

A wrapper of torch.meshgrid to compat different PyTorch versions.

Since PyTorch 1.10.0a0, torch.meshgrid supports the arguments `indexing`. So we implement a wrapper here to avoid warning when using high-version PyTorch and avoid compatibility issues when using previous versions of PyTorch.

Parameters **tensors** (*List[Tensor]*) – List of scalars or 1 dimensional tensors.

Returns Sequence of meshgrid tensors.

Return type Sequence[Tensor]

mmcv.utils.**track_iter_progress**(*tasks*, *bar_width=50*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Track the progress of tasks iteration or enumeration with a progress bar.

Tasks are yielded with a simple for-loop.

Parameters

- **tasks** (*list or tuple[Iterable, int]*) – A list of tasks or (tasks, total num).
- **bar_width** (*int*) – Width of progress bar.

Yields *list* – The task results.

mmcv.utils.**track_parallel_progress**(*func*, *tasks*, *nproc*, *initializer=None*, *initargs=None*, *bar_width=50*, *chunksize=1*, *skip_first=False*, *keep_order=True*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Track the progress of parallel task execution with a progress bar.

The built-in `multiprocessing` module is used for process pools and tasks are done with `Pool.map()` or `Pool.imap_unordered()`.

Parameters

- **func** (*callable*) – The function to be applied to each task.
- **tasks** (*list or tuple[Iterable, int]*) – A list of tasks or (tasks, total num).
- **nproc** (*int*) – Process (worker) number.
- **initializer** (*None or callable*) – Refer to `multiprocessing.Pool` for details.

- **initargs** (*None or tuple*) – Refer to `multiprocessing.Pool` for details.
- **chunksize** (*int*) – Refer to `multiprocessing.Pool` for details.
- **bar_width** (*int*) – Width of progress bar.
- **skip_first** (*bool*) – Whether to skip the first sample for each worker when estimating fps, since the initialization step may takes longer.
- **keep_order** (*bool*) – If True, `Pool.imap()` is used, otherwise `Pool.imap_unordered()` is used.

Returns The task results.

Return type list

mmcv.utils.**track_progress**(*func*, *tasks*, *bar_width=50*, *file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, ***kwargs*)

Track the progress of tasks execution with a progress bar.

Tasks are done with a simple for-loop.

Parameters

- **func** (*callable*) – The function to be applied to each task.
- **tasks** (*list or tuple[Iterable, int]*) – A list of tasks or (tasks, total num).
- **bar_width** (*int*) – Width of progress bar.

Returns The task results.

Return type list

mmcv.utils.**tuple_cast**(*inputs*, *dst_type*)

Cast elements of an iterable object into a tuple of some type.

A partial method of *iter_cast()*.

mmcv.utils.**worker_init_fn**(*worker_id: int*, *num_workers: int*, *rank: int*, *seed: int*)

Function to initialize each worker.

The seed of each worker equals to `num_worker * rank + worker_id + user_seed`.

Parameters

- **worker_id** (*int*) – Id for each worker.
- **num_workers** (*int*) – Number of workers.
- **rank** (*int*) – Rank in distributed training.
- **seed** (*int*) – Random seed.

CHAPTER

THIRTYTWO

CNN

class mmcv.cnn.**AlexNet**(*num_classes: int = - 1*)
AlexNet backbone.

Parameters **num_classes** (*int*) – number of classes for classification.

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**Caffe2XavierInit**(***kwargs*)

class mmcv.cnn.**ConstantInit**(*val: Union[int, float]*, ***kwargs*)
Initialize module parameters with constant values.

Parameters

- **val** (*int | float*) – the value to fill the weights in the module with
- **bias** (*int | float*) – the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) – the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) – the layer will be initialized. Defaults to None.

class mmcv.cnn.**ContextBlock**(*in_channels: int*, *ratio: float*, *pooling_type: str = 'att'*, *fusion_types: tuple = ('channel_add')*)
ContextBlock module in GCNet.

See 'GCNet: Non-local Networks Meet Squeeze-Excitation Networks and Beyond' (https://arxiv.org/abs/1904.11492) for details.

Parameters

- **in_channels** (*int*) – Channels of the input feature map.
- **ratio** (*float*) – Ratio of channels of transform bottleneck
- **pooling_type** (*str*) – Pooling method for context modeling. Options are 'att' and 'avg', stand for attention pooling and average pooling respectively. Default: 'att'.
- **fusion_types** (*Sequence[str]*) – Fusion method for feature fusion, Options are 'channels_add', 'channel_mul', stand for channelwise addition and multiplication respectively. Default: ('channel_add',)

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**Conv2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[str, int, Tuple[int, int]] = 0*, *dilation: Union[int, Tuple[int, int]] = 1*, *groups: int = 1*, *bias: bool = True*, *padding_mode: str = 'zeros'*, *device=None*, *dtype=None*)

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**Conv3d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int, int]]*, *stride: Union[int, Tuple[int, int, int]] = 1*, *padding: Union[str, int, Tuple[int, int, int]] = 0*, *dilation: Union[int, Tuple[int, int, int]] = 1*, *groups: int = 1*, *bias: bool = True*, *padding_mode: str = 'zeros'*, *device=None*, *dtype=None*)

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**ConvAWS2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[int, Tuple[int, int]] = 0*, *dilation: Union[int, Tuple[int, int]] = 1*, *groups: int = 1*, *bias: bool = True*)
AWS (Adaptive Weight Standardization)

This is a variant of Weight Standardization (https://arxiv.org/pdf/1903.10520.pdf) It is used in DetectoRS to avoid NaN (https://arxiv.org/pdf/2006.02334.pdf)

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the conv kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1

- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If set True, adds a learnable bias to the output. Default: True

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**ConvModule**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[int, Tuple[int, int]] = 0*, *dilation: Union[int, Tuple[int, int]] = 1*, *groups: int = 1*, *bias: Union[bool, str] = 'auto'*, *conv_cfg: Optional[Dict] = None*, *norm_cfg: Optional[Dict] = None*, *act_cfg: Optional[Dict] = {'type': 'ReLU'}*, *inplace: bool = True*, *with_spectral_norm: bool = False*, *padding_mode: str = 'zeros'*, *order: tuple = ('conv', 'norm', 'act')*)

A conv block that bundles conv/norm/activation layers.

This block simplifies the usage of convolution layers, which are commonly used with a norm layer (e.g., BatchNorm) and activation layer (e.g., ReLU). It is based upon three build methods: *build_conv_layer()*, *build_norm_layer()* and *build_activation_layer()*.

Besides, we add some additional features in this module. 1. Automatically set *bias* of the conv layer. 2. Spectral norm is supported. 3. More padding modes are supported. Before PyTorch 1.5, nn.Conv2d only supports zero and circular padding, and we add "reflect" padding mode.

Parameters

- **in_channels** (*int*) – Number of channels in the input feature map. Same as that in `nn._ConvNd`.
- **out_channels** (*int*) – Number of channels produced by the convolution. Same as that in `nn._ConvNd`.
- **kernel_size** (*int | tuple[int]*) – Size of the convolving kernel. Same as that in `nn._ConvNd`.
- **stride** (*int | tuple[int]*) – Stride of the convolution. Same as that in `nn._ConvNd`.
- **padding** (*int | tuple[int]*) – Zero-padding added to both sides of the input. Same as that in `nn._ConvNd`.
- **dilation** (*int | tuple[int]*) – Spacing between kernel elements. Same as that in `nn._ConvNd`.
- **groups** (*int*) – Number of blocked connections from input channels to output channels. Same as that in `nn._ConvNd`.
- **bias** (*bool | str*) – If specified as *auto*, it will be decided by the norm_cfg. Bias will be set as True if *norm_cfg* is None, otherwise False. Default: "auto".

- **conv_cfg** (*dict*) – Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: None.
- **act_cfg** (*dict*) – Config dict for activation layer. Default: dict(type='ReLU').
- **inplace** (*bool*) – Whether to use inplace mode for activation. Default: True.
- **with_spectral_norm** (*bool*) – Whether use spectral norm in conv module. Default: False.
- **padding_mode** (*str*) – If the *padding_mode* has not been supported by current *Conv2d* in PyTorch, we will use our own padding layer instead. Currently, we support ['zeros', 'circular'] with official implementation and ['reflect'] with our own implementation. Default: 'zeros'.
- **order** (*tuple[str]*) – The order of conv/norm/activation layers. It is a sequence of "conv", "norm" and "act". Common examples are ("conv", "norm", "act") and ("act", "conv", "norm"). Default: ('conv', 'norm', 'act').

forward(*x: torch.Tensor*, *activate: bool = True*, *norm: bool = True*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**ConvTranspose2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[int, Tuple[int, int]] = 0*, *output_padding: Union[int, Tuple[int, int]] = 0*, *groups: int = 1*, *bias: bool = True*, *dilation: Union[int, Tuple[int, int]] = 1*, *padding_mode: str = 'zeros'*, *device=None*, *dtype=None*)

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**ConvTranspose3d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int, int]]*, *stride: Union[int, Tuple[int, int, int]] = 1*, *padding: Union[int, Tuple[int, int, int]] = 0*, *output_padding: Union[int, Tuple[int, int, int]] = 0*, *groups: int = 1*, *bias: bool = True*, *dilation: Union[int, Tuple[int, int, int]] = 1*, *padding_mode: str = 'zeros'*, *device=None*, *dtype=None*)

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.cnn.**ConvWS2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[int, Tuple[int, int]] = 0*, *dilation: Union[int, Tuple[int, int]] = 1*, *groups: int = 1*, *bias: bool = True*, *eps: float = 1e-05*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.cnn.**DepthwiseSeparableConvModule**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, int]]*, *stride: Union[int, Tuple[int, int]] = 1*, *padding: Union[int, Tuple[int, int]] = 0*, *dilation: Union[int, Tuple[int, int]] = 1*, *norm_cfg: Optional[Dict] = None*, *act_cfg: Dict = {'type': 'ReLU'}*, *dw_norm_cfg: Union[Dict, str] = 'default'*, *dw_act_cfg: Union[Dict, str] = 'default'*, *pw_norm_cfg: Union[Dict, str] = 'default'*, *pw_act_cfg: Union[Dict, str] = 'default'*, ***kwargs*)

Depthwise separable convolution module.

See https://arxiv.org/pdf/1704.04861.pdf for details.

This module can replace a ConvModule with the conv block replaced by two conv block: depthwise conv block and pointwise conv block. The depthwise conv block contains depthwise-conv/norm/activation layers. The pointwise conv block contains pointwise-conv/norm/activation layers. It should be noted that there will be norm/activation layer in the depthwise conv block if *norm_cfg* and *act_cfg* are specified.

Parameters

- **in_channels** (*int*) – Number of channels in the input feature map. Same as that in `nn._ConvNd`.
- **out_channels** (*int*) – Number of channels produced by the convolution. Same as that in `nn._ConvNd`.
- **kernel_size** (*int | tuple[int]*) – Size of the convolving kernel. Same as that in `nn._ConvNd`.
- **stride** (*int | tuple[int]*) – Stride of the convolution. Same as that in `nn._ConvNd`. Default: 1.
- **padding** (*int | tuple[int]*) – Zero-padding added to both sides of the input. Same as that in `nn._ConvNd`. Default: 0.
- **dilation** (*int | tuple[int]*) – Spacing between kernel elements. Same as that in `nn._ConvNd`. Default: 1.
- **norm_cfg** (*dict*) – Default norm config for both depthwise ConvModule and pointwise ConvModule. Default: None.

- **act_cfg** (*dict*) – Default activation config for both depthwise ConvModule and pointwise ConvModule. Default: dict(type='ReLU').
- **dw_norm_cfg** (*dict*) – Norm config of depthwise ConvModule. If it is 'default', it will be the same as *norm_cfg*. Default: 'default'.
- **dw_act_cfg** (*dict*) – Activation config of depthwise ConvModule. If it is 'default', it will be the same as *act_cfg*. Default: 'default'.
- **pw_norm_cfg** (*dict*) – Norm config of pointwise ConvModule. If it is 'default', it will be the same as *norm_cfg*. Default: 'default'.
- **pw_act_cfg** (*dict*) – Activation config of pointwise ConvModule. If it is 'default', it will be the same as *act_cfg*. Default: 'default'.
- **kwargs** (*optional*) – Other shared arguments for depthwise and pointwise ConvModule. See ConvModule for ref.

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**GeneralizedAttention**(*in_channels: int*, *spatial_range: int = - 1*, *num_heads: int = 9*, *position_embedding_dim: int = - 1*, *position_magnitude: int = 1*, *kv_stride: int = 2*, *q_stride: int = 1*, *attention_type: str = '1111'*)

GeneralizedAttention module.

See 'An Empirical Study of Spatial Attention Mechanisms in Deep Networks' (https://arxiv.org/abs/1711.07971) for details.

Parameters

- **in_channels** (*int*) – Channels of the input feature map.
- **spatial_range** (*int*) – The spatial range. -1 indicates no spatial range constraint. Default: -1.
- **num_heads** (*int*) – The head number of empirical_attention module. Default: 9.
- **position_embedding_dim** (*int*) – The position embedding dimension. Default: -1.
- **position_magnitude** (*int*) – A multiplier acting on coord difference. Default: 1.
- **kv_stride** (*int*) – The feature stride acting on key/value feature map. Default: 2.
- **q_stride** (*int*) – The feature stride acting on query feature map. Default: 1.
- **attention_type** (*str*) – A binary indicator string for indicating which items in generalized empirical_attention module are used. Default: '1111'.
  - '1000' indicates 'query and key content' (appr - appr) item,
  - '0100' indicates 'query content and relative position' (appr - position) item,
  - '0010' indicates 'key content only' (bias - appr) item,
  - '0001' indicates 'relative position only' (bias - position) item.

forward(*x_input: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**HSigmoid**(*bias: float = 3.0*, *divisor: float = 6.0*, *min_value: float = 0.0*, *max_value: float = 1.0*)
Hard Sigmoid Module. Apply the hard sigmoid function: Hsigmoid(x) = min(max((x + bias) / divisor, min_value), max_value) Default: Hsigmoid(x) = min(max((x + 3) / 6, 0), 1)

Note: In MMCV v1.4.4, we modified the default value of args to align with PyTorch official.

Parameters

- **bias** (*float*) – Bias of the input feature map. Default: 3.0.
- **divisor** (*float*) – Divisor of the input feature map. Default: 6.0.
- **min_value** (*float*) – Lower bound value. Default: 0.0.
- **max_value** (*float*) – Upper bound value. Default: 1.0.

Returns The output tensor.

Return type Tensor

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**HSwish**(*inplace: bool = False*)
Hard Swish Module.

This module applies the hard swish function:

$$Hswish(x) = x * ReLU6(x + 3)/6$$

Parameters **inplace** (*bool*) – can optionally do the operation in-place. Default: False.

Returns The output tensor.

Return type Tensor

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.cnn.**KaimingInit**(*a: float = 0*, *mode: str = 'fan_out'*, *nonlinearity: str = 'relu'*, *distribution: str = 'normal'*, ***kwargs*)

Initialize module parameters with the values according to the method described in Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification - He, K. et al. (2015).

Parameters

- **a** (*int | float*) – the negative slope of the rectifier used after this layer (only used with `'leaky_relu'`). Defaults to 0.
- **mode** (*str*) – either `'fan_in'` or `'fan_out'`. Choosing `'fan_in'` preserves the magnitude of the variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitudes in the backwards pass. Defaults to `'fan_out'`.
- **nonlinearity** (*str*) – the non-linear function (*nn.functional* name), recommended to use only with `'relu'` or `'leaky_relu'` . Defaults to 'relu'.
- **bias** (*int | float*) – the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) – the probability for bias initialization. Defaults to None.
- **distribution** (*str*) – distribution either be `'normal'` or `'uniform'`. Defaults to `'normal'`.
- **layer** (*str | list[str], optional*) – the layer will be initialized. Defaults to None.

class mmcv.cnn.**Linear**(*in_features: int*, *out_features: int*, *bias: bool = True*, *device=None*, *dtype=None*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.cnn.**MaxPool2d**(*kernel_size: Union[int, Tuple[int, ...]]*, *stride: Optional[Union[int, Tuple[int, ...]]] = None*, *padding: Union[int, Tuple[int, ...]] = 0*, *dilation: Union[int, Tuple[int, ...]] = 1*, *return_indices: bool = False*, *ceil_mode: bool = False*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.cnn.**MaxPool3d**(*kernel_size: Union[int, Tuple[int, ...]]*, *stride: Optional[Union[int, Tuple[int, ...]]] = None*, *padding: Union[int, Tuple[int, ...]] = 0*, *dilation: Union[int, Tuple[int, ...]] = 1*, *return_indices: bool = False*, *ceil_mode: bool = False*)

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**NonLocal1d**(*in_channels: int*, *sub_sample: bool = False*, *conv_cfg: Dict = {'type': 'Conv1d'}*, ***kwargs*)

1D Non-local module.

Parameters

- **in_channels** (*int*) – Same as *NonLocalND*.
- **sub_sample** (*bool*) – Whether to apply max pooling after pairwise function (Note that the *sub_sample* is applied on spatial only). Default: False.
- **conv_cfg** (*None | dict*) – Same as *NonLocalND*. Default: dict(type='Conv1d').

class mmcv.cnn.**NonLocal2d**(*in_channels: int*, *sub_sample: bool = False*, *conv_cfg: Dict = {'type': 'Conv2d'}*, ***kwargs*)

2D Non-local module.

Parameters

- **in_channels** (*int*) – Same as *NonLocalND*.
- **sub_sample** (*bool*) – Whether to apply max pooling after pairwise function (Note that the *sub_sample* is applied on spatial only). Default: False.
- **conv_cfg** (*None | dict*) – Same as *NonLocalND*. Default: dict(type='Conv2d').

class mmcv.cnn.**NonLocal3d**(*in_channels: int*, *sub_sample: bool = False*, *conv_cfg: Dict = {'type': 'Conv3d'}*, ***kwargs*)

3D Non-local module.

Parameters

- **in_channels** (*int*) – Same as *NonLocalND*.
- **sub_sample** (*bool*) – Whether to apply max pooling after pairwise function (Note that the *sub_sample* is applied on spatial only). Default: False.
- **conv_cfg** (*None | dict*) – Same as *NonLocalND*. Default: dict(type='Conv3d').

class mmcv.cnn.**NormalInit**(*mean: float = 0*, *std: float = 1*, ***kwargs*)

Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$.

Parameters

- **mean** (*int | float*) – the mean of the normal distribution. Defaults to 0.
- **std** (*int | float*) – the standard deviation of the normal distribution. Defaults to 1.
- **bias** (*int | float*) – the value to fill the bias. Defaults to 0.

- **bias_prob** (*float, optional*) – the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) – the layer will be initialized. Defaults to None.

class mmcv.cnn.**PretrainedInit**(*checkpoint: str*, *prefix: Optional[str] = None*, *map_location: Optional[str] = None*)

Initialize module by loading a pretrained model.

Parameters

- **checkpoint** (*str*) – the checkpoint file of the pretrained model should be load.
- **prefix** (*str, optional*) – the prefix of a sub-module in the pretrained model. it is for loading a part of the pretrained model to initialize. For example, if we would like to only load the backbone of a detector model, we can set `prefix='backbone.'`. Defaults to None.
- **map_location** (*str*) – map tensors into proper locations.

class mmcv.cnn.**ResNet**(*depth: int*, *num_stages: int = 4*, *strides: Sequence[int] = (1, 2, 2, 2)*, *dilations: Sequence[int] = (1, 1, 1, 1)*, *out_indices: Sequence[int] = (0, 1, 2, 3)*, *style: str = 'pytorch'*, *frozen_stages: int = - 1*, *bn_eval: bool = True*, *bn_frozen: bool = False*, *with_cp: bool = False*)

ResNet backbone.

Parameters

- **depth** (*int*) – Depth of resnet, from {18, 34, 50, 101, 152}.
- **num_stages** (*int*) – Resnet stages, normally 4.
- **strides** (*Sequence[int]*) – Strides of the first block of each stage.
- **dilations** (*Sequence[int]*) – Dilation of each stage.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **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.
- **frozen_stages** (*int*) – Stages to be frozen (all param fixed). -1 means not freezing any parameters.
- **bn_eval** (*bool*) – Whether to set BN layers as eval mode, namely, freeze running stats (mean and var).
- **bn_frozen** (*bool*) – Whether to freeze weight and bias of BN layers.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed.

forward(*x: torch.Tensor*) → Union[torch.Tensor, Tuple[torch.Tensor]]

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*mode: bool = True*) → None

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

class mmcv.cnn.**Scale**(*scale: float = 1.0*)

A learnable scale parameter.

This layer scales the input by a learnable factor. It multiplies a learnable scale parameter of shape (1,) with input of any shape.

Parameters **scale** (*float*) – Initial value of scale factor. Default: 1.0

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**Swish**

Swish Module.

This module applies the swish function:

$$Swish(x) = x * Sigmoid(x)$$

Returns The output tensor.

Return type Tensor

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.cnn.**TruncNormalInit**(*mean: float = 0*, *std: float = 1*, *a: float = - 2*, *b: float = 2*, ***kwargs*)

Initialize module parameters with the values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$ with values outside $[a, b]$.

Parameters

- **mean** (*float*) – the mean of the normal distribution. Defaults to 0.
- **std** (*float*) – the standard deviation of the normal distribution. Defaults to 1.
- **a** (*float*) – The minimum cutoff value.
- **b** (*float*) – The maximum cutoff value.

- **bias** (*float*) – the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) – the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) – the layer will be initialized. Defaults to None.

class mmcv.cnn.**UniformInit**(*a: float = 0.0*, *b: float = 1.0*, ***kwargs*)

Initialize module parameters with values drawn from the uniform distribution $\mathcal{U}(a, b)$.

Parameters

- **a** (*int | float*) – the lower bound of the uniform distribution. Defaults to 0.
- **b** (*int | float*) – the upper bound of the uniform distribution. Defaults to 1.
- **bias** (*int | float*) – the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) – the probability for bias initialization. Defaults to None.
- **layer** (*str | list[str], optional*) – the layer will be initialized. Defaults to None.

class mmcv.cnn.**VGG**(*depth: int*, *with_bn: bool = False*, *num_classes: int = - 1*, *num_stages: int = 5*, *dilations: Sequence[int] = (1, 1, 1, 1, 1)*, *out_indices: Sequence[int] = (0, 1, 2, 3, 4)*, *frozen_stages: int = - 1*, *bn_eval: bool = True*, *bn_frozen: bool = False*, *ceil_mode: bool = False*, *with_last_pool: bool = True*)

VGG backbone.

Parameters

- **depth** (*int*) – Depth of vgg, from {11, 13, 16, 19}.
- **with_bn** (*bool*) – Use BatchNorm or not.
- **num_classes** (*int*) – number of classes for classification.
- **num_stages** (*int*) – VGG stages, normally 5.
- **dilations** (*Sequence[int]*) – Dilation of each stage.
- **out_indices** (*Sequence[int]*) – Output from which stages.
- **frozen_stages** (*int*) – Stages to be frozen (all param fixed). -1 means not freezing any parameters.
- **bn_eval** (*bool*) – Whether to set BN layers as eval mode, namely, freeze running stats (mean and var).
- **bn_frozen** (*bool*) – Whether to freeze weight and bias of BN layers.

forward(*x: torch.Tensor*) → Union[torch.Tensor, Tuple[torch.Tensor, ...]]

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

train(*mode: bool = True*) → None

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. `Dropout`, `BatchNorm`, etc.

Parameters **mode** (*bool*) – whether to set training mode (`True`) or evaluation mode (`False`). Default: `True`.

Returns self

Return type Module

class mmcv.cnn.**XavierInit**(*gain: float = 1*, *distribution: str = 'normal'*, ***kwargs*)
Initialize module parameters with values according to the method described in Understanding the difficulty of training deep feedforward neural networks - Glorot, X. & Bengio, Y. (2010).

Parameters

- **gain** (*int | float*) – an optional scaling factor. Defaults to 1.
- **bias** (*int | float*) – the value to fill the bias. Defaults to 0.
- **bias_prob** (*float, optional*) – the probability for bias initialization. Defaults to None.
- **distribution** (*str*) – distribution either be `'normal'` or `'uniform'`. Defaults to `'normal'`.
- **layer** (*str | list[str], optional*) – the layer will be initialized. Defaults to None.

mmcv.cnn.**bias_init_with_prob**(*prior_prob: float*) → float
initialize conv/fc bias value according to a given probability value.

mmcv.cnn.**build_activation_layer**(*cfg: Dict*) → torch.nn.modules.module.Module
Build activation layer.

Parameters **cfg** (*dict*) – The activation layer config, which should contain:

- type (str): Layer type.
- layer args: Args needed to instantiate an activation layer.

Returns Created activation layer.

Return type nn.Module

mmcv.cnn.**build_conv_layer**(*cfg: Optional[Dict]*, **args*, ***kwargs*) → torch.nn.modules.module.Module
Build convolution layer.

Parameters

- **cfg** (*None or dict*) – The conv layer config, which should contain: - type (str): Layer type. - layer args: Args needed to instantiate an conv layer.
- **args** (*argument list*) – Arguments passed to the *__init__* method of the corresponding conv layer.
- **kwargs** (*keyword arguments*) – Keyword arguments passed to the *__init__* method of the corresponding conv layer.

Returns Created conv layer.

Return type nn.Module

mmcv.cnn.**build_model_from_cfg**(*cfg*, *registry*, *default_args=None*)
Build a PyTorch model from config dict(s). Different from `build_from_cfg`, if cfg is a list, a `nn.Sequential` will be built.

Parameters

- **cfg** (*dict, list[dict]*) – The config of modules, is is either a config dict or a list of config dicts. If cfg is a list, a the built modules will be wrapped with `nn.Sequential`.

- **registry** (Registry) – A registry the module belongs to.
- **default_args** (*dict, optional*) – Default arguments to build the module. Defaults to None.

Returns A built nn module.

Return type nn.Module

mmcv.cnn.**build_norm_layer**(*cfg: Dict*, *num_features: int*, *postfix: Union[int, str] = ''*) → Tuple[str, torch.nn.modules.module.Module]

Build normalization layer.

Parameters

- **cfg** (*dict*) – The norm layer config, which should contain:
  - type (str): Layer type.
  - layer args: Args needed to instantiate a norm layer.
  - requires_grad (bool, optional): Whether stop gradient updates.
- **num_features** (*int*) – Number of input channels.
- **postfix** (*int | str*) – The postfix to be appended into norm abbreviation to create named layer.

Returns The first element is the layer name consisting of abbreviation and postfix, e.g., bn1, gn. The second element is the created norm layer.

Return type tuple[str, nn.Module]

mmcv.cnn.**build_padding_layer**(*cfg: Dict*, **args*, ***kwargs*) → torch.nn.modules.module.Module

Build padding layer.

Parameters **cfg** (*dict*) – The padding layer config, which should contain: - type (str): Layer type. - layer args: Args needed to instantiate a padding layer.

Returns Created padding layer.

Return type nn.Module

mmcv.cnn.**build_plugin_layer**(*cfg: Dict*, *postfix: Union[int, str] = ''*, ***kwargs*) → Tuple[str, torch.nn.modules.module.Module]

Build plugin layer.

Parameters

- **cfg** (*dict*) – cfg should contain:
  - type (str): identify plugin layer type.
  - layer args: args needed to instantiate a plugin layer.
- **postfix** (*int, str*) – appended into norm abbreviation to create named layer. Default: ‘’.

Returns The first one is the concatenation of abbreviation and postfix. The second is the created plugin layer.

Return type tuple[str, nn.Module]

mmcv.cnn.**build_upsample_layer**(*cfg: Dict*, **args*, ***kwargs*) → torch.nn.modules.module.Module

Build upsample layer.

Parameters

- **cfg** (*dict*) – The upsample layer config, which should contain:

- type (str): Layer type.
- scale_factor (int): Upsample ratio, which is not applicable to deconv.
- layer args: Args needed to instantiate a upsample layer.

- **args** (*argument list*) – Arguments passed to the `__init__` method of the corresponding conv layer.
- **kwargs** (*keyword arguments*) – Keyword arguments passed to the `__init__` method of the corresponding conv layer.

Returns Created upsample layer.

Return type nn.Module

mmcv.cnn.**fuse_conv_bn**(*module: torch.nn.modules.module.Module*) → torch.nn.modules.module.Module
Recursively fuse conv and bn in a module.

During inference, the functionary of batch norm layers is turned off but only the mean and var alone channels are used, which exposes the chance to fuse it with the preceding conv layers to save computations and simplify network structures.

Parameters **module** (*nn.Module*) – Module to be fused.

Returns Fused module.

Return type nn.Module

mmcv.cnn.**get_model_complexity_info**(*model: torch.nn.modules.module.Module*, *input_shape: tuple*, *print_per_layer_stat: bool = True*, *as_strings: bool = True*, *input_constructor: Optional[Callable] = None*, *flush: bool = False*, *ost: TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*) → tuple
Get complexity information of a model.

This method can calculate FLOPs and parameter counts of a model with corresponding input shape. It can also print complexity information for each layer in a model.

Supported layers are listed as below:

- Convolutions: `nn.Conv1d`, `nn.Conv2d`, `nn.Conv3d`.
- Activations: `nn.ReLU`, `nn.PReLU`, `nn.ELU`, `nn.LeakyReLU`, `nn.ReLU6`.
- Poolings: `nn.MaxPool1d`, `nn.MaxPool2d`, `nn.MaxPool3d`, `nn.AvgPool1d`, `nn.AvgPool2d`, `nn.AvgPool3d`, `nn.AdaptiveMaxPool1d`, `nn.AdaptiveMaxPool2d`, `nn.AdaptiveMaxPool3d`, `nn.AdaptiveAvgPool1d`, `nn.AdaptiveAvgPool2d`, `nn.AdaptiveAvgPool3d`.
- BatchNorms: `nn.BatchNorm1d`, `nn.BatchNorm2d`, `nn.BatchNorm3d`, `nn.GroupNorm`, `nn.InstanceNorm1d`, `InstanceNorm2d`, `InstanceNorm3d`, `nn.LayerNorm`.
- Linear: `nn.Linear`.
- Deconvolution: `nn.ConvTranspose2d`.
- Upsample: `nn.Upsample`.

Parameters

- **model** (*nn.Module*) – The model for complexity calculation.
- **input_shape** (*tuple*) – Input shape used for calculation.
- **print_per_layer_stat** (*bool*) – Whether to print complexity information for each layer in a model. Default: True.

- **as_strings** (*bool*) – Output FLOPs and params counts in a string form. Default: True.
- **input_constructor** (*None | callable*) – If specified, it takes a callable method that generates input. otherwise, it will generate a random tensor with input shape to calculate FLOPs. Default: None.
- **flush** (*bool*) – same as that in `print()`. Default: False.
- **ost** (*stream*) – same as `file` param in `print()`. Default: sys.stdout.

Returns If `as_strings` is set to True, it will return FLOPs and parameter counts in a string format. otherwise, it will return those in a float number format.

Return type tuple[float | str]

mmcv.cnn.**initialize**(*module: torch.nn.modules.module.Module*, *init_cfg: Union[Dict, List[dict]]*) → None

Initialize a module.

Parameters

- **module** (`torch.nn.Module`) – the module will be initialized.
- **init_cfg** (*dict | list[dict]*) – initialization configuration dict to define initializer. OpenMMLab has implemented 6 initializers including `Constant`, `Xavier`, `Normal`, `Uniform`, `Kaiming`, and `Pretrained`.

Example

```
>>> module = nn.Linear(2, 3, bias=True)
>>> init_cfg = dict(type='Constant', layer='Linear', val =1 , bias =2)
>>> initialize(module, init_cfg)
```

```
>>> module = nn.Sequential(nn.Conv1d(3, 1, 3), nn.Linear(1,2))
>>> # define key ``'layer'`` for initializing layer with different
>>> # configuration
>>> init_cfg = [dict(type='Constant', layer='Conv1d', val=1),
        dict(type='Constant', layer='Linear', val=2)]
>>> initialize(module, init_cfg)
```

```
>>> # define key``'override'`` to initialize some specific part in
>>> # module
>>> class FooNet(nn.Module):
>>>     def __init__(self):
>>>         super().__init__()
>>>         self.feat = nn.Conv2d(3, 16, 3)
>>>         self.reg = nn.Conv2d(16, 10, 3)
>>>         self.cls = nn.Conv2d(16, 5, 3)
>>> model = FooNet()
>>> init_cfg = dict(type='Constant', val=1, bias=2, layer='Conv2d',
>>>     override=dict(type='Constant', name='reg', val=3, bias=4))
>>> initialize(model, init_cfg)
```

```
>>> model = ResNet(depth=50)
>>> # Initialize weights with the pretrained model.
>>> init_cfg = dict(type='Pretrained',
```

(continues on next page)

(continued from previous page)

```
        checkpoint='torchvision://resnet50')
>>> initialize(model, init_cfg)
```

```
>>> # Initialize weights of a sub-module with the specific part of
>>> # a pretrained model by using "prefix".
>>> url = 'http://download.openmmlab.com/mmdetection/v2.0/retinanet/'\
>>>     'retinanet_r50_fpn_1x_coco/'\
>>>     'retinanet_r50_fpn_1x_coco_20200130-c2398f9e.pth'
>>> init_cfg = dict(type='Pretrained',
        checkpoint=url, prefix='backbone.')
```

mmcv.cnn.**is_norm**(*layer: torch.nn.modules.module.Module*, *exclude: Optional[Union[type, tuple]] = None*) → bool

Check if a layer is a normalization layer.

Parameters

- **layer** (*nn.Module*) – The layer to be checked.
- **exclude** (*type | tuple[type]*) – Types to be excluded.

Returns Whether the layer is a norm layer.

Return type bool

CHAPTER

THIRTYTHREE

RUNNER

class mmcv.runner.**BaseModule**(*init_cfg: Optional[dict] = None*)

Base module for all modules in openmmlab.

BaseModule is a wrapper of torch.nn.Module with additional functionality of parameter initialization. Compared with torch.nn.Module, BaseModule mainly adds three attributes.

- init_cfg: the config to control the initialization.
- init_weights: The function of parameter initialization and recording initialization information.
- _params_init_info: Used to track the parameter initialization information. This attribute only exists during executing the init_weights.

Parameters **init_cfg** (*dict, optional*) – Initialization config dict.

init_weights() → None

Initialize the weights.

class mmcv.runner.**BaseRunner**(*model: torch.nn.modules.module.Module*, *batch_processor: Optional[Callable] = None*, *optimizer: Optional[Union[Dict, torch.optim.optimizer.Optimizer]] = None*, *work_dir: Optional[str] = None*, *logger: Optional[logging.Logger] = None*, *meta: Optional[Dict] = None*, *max_iters: Optional[int] = None*, *max_epochs: Optional[int] = None*)

The base class of Runner, a training helper for PyTorch.

All subclasses should implement the following APIs:

- run()
- train()
- val()
- save_checkpoint()

Parameters

- **model** (torch.nn.Module) – The model to be run.
- **batch_processor** (*callable*) – A callable method that process a data batch. The interface of this method should be *batch_processor(model, data, train_mode) -> dict*
- **optimizer** (dict or torch.optim.Optimizer) – It can be either an optimizer (in most cases) or a dict of optimizers (in models that requires more than one optimizer, e.g., GAN).
- **work_dir** (*str, optional*) – The working directory to save checkpoints and logs. Defaults to None.

- **logger** (logging.Logger) – Logger used during training. Defaults to None. (The default value is just for backward compatibility)
- **meta** (*dict | None*) – A dict records some import information such as environment info and seed, which will be logged in logger hook. Defaults to None.
- **max_epochs** (*int, optional*) – Total training epochs.
- **max_iters** (*int, optional*) – Total training iterations.

call_hook(*fn_name: str*) → None
Call all hooks.

Parameters **fn_name** (*str*) – The function name in each hook to be called, such as "before_train_epoch".

current_lr() → Union[List[float], Dict[str, List[float]]]
Get current learning rates.

Returns Current learning rates of all param groups. If the runner has a dict of optimizers, this method will return a dict.

Return type list[float] | dict[str, list[float]]

current_momentum() → Union[List[float], Dict[str, List[float]]]
Get current momentums.

Returns Current momentums of all param groups. If the runner has a dict of optimizers, this method will return a dict.

Return type list[float] | dict[str, list[float]]

property epoch: int
Current epoch.

Type int

property hooks: List[mmcv.runner.hooks.hook.Hook]
A list of registered hooks.

Type list[Hook]

property inner_iter: int
Iteration in an epoch.

Type int

property iter: int
Current iteration.

Type int

property max_epochs
Maximum training epochs.

Type int

property max_iters
Maximum training iterations.

Type int

property model_name: str
Name of the model, usually the module class name.

Type str

property rank: int
Rank of current process. (distributed training)

Type int

register_hook(*hook: mmcv.runner.hooks.hook.Hook*, *priority: Union[int, str, mmcv.runner.priority.Priority] = 'NORMAL'*) → None
Register a hook into the hook list.

The hook will be inserted into a priority queue, with the specified priority (See *Priority* for details of priorities). For hooks with the same priority, they will be triggered in the same order as they are registered.

Parameters

- **hook** (Hook) – The hook to be registered.
- **priority** (int or str or *Priority*) – Hook priority. Lower value means higher priority.

register_hook_from_cfg(*hook_cfg: Dict*) → None
Register a hook from its cfg.

Parameters **hook_cfg** (*dict*) – Hook config. It should have at least keys 'type' and 'priority' indicating its type and priority.

Note: The specific hook class to register should not use 'type' and 'priority' arguments during initialization.

---

register_training_hooks(*lr_config: Optional[Union[Dict, mmcv.runner.hooks.hook.Hook]]*, *optimizer_config: Optional[Union[Dict, mmcv.runner.hooks.hook.Hook]] = None*, *checkpoint_config: Optional[Union[Dict, mmcv.runner.hooks.hook.Hook]] = None*, *log_config: Optional[Dict] = None*, *momentum_config: Optional[Union[Dict, mmcv.runner.hooks.hook.Hook]] = None*, *timer_config: Union[Dict, mmcv.runner.hooks.hook.Hook] = {'type': 'IterTimerHook'}*, *custom_hooks_config: Optional[Union[List, Dict, mmcv.runner.hooks.hook.Hook]] = None*) → None
Register default and custom hooks for training.

Default and custom hooks include:

Hooks	Priority
LrUpdaterHook	VERY_HIGH (10)
MomentumUpdaterHook	HIGH (30)
OptimizerStepperHook	ABOVE_NORMAL (40)
CheckpointSaverHook	NORMAL (50)
IterTimerHook	LOW (70)
LoggerHook(s)	VERY_LOW (90)
CustomHook(s)	defaults to NORMAL (50)

If custom hooks have same priority with default hooks, custom hooks will be triggered after default hooks.

property world_size: int
Number of processes participating in the job. (distributed training)

Type int

class mmcv.runner.**CheckpointHook**(*interval: int = - 1*, *by_epoch: bool = True*, *save_optimizer: bool = True*, *out_dir: Optional[str] = None*, *max_keep_ckpts: int = - 1*, *save_last: bool = True*, *sync_buffer: bool = False*, *file_client_args: Optional[dict] = None*, ***kwargs*)

Save checkpoints periodically.

Parameters

- **interval** (*int*) – The saving period. If `by_epoch=True`, interval indicates epochs, otherwise it indicates iterations. Default: -1, which means "never".
- **by_epoch** (*bool*) – Saving checkpoints by epoch or by iteration. Default: True.
- **save_optimizer** (*bool*) – Whether to save optimizer state_dict in the checkpoint. It is usually used for resuming experiments. Default: True.
- **out_dir** (*str, optional*) – The root directory to save checkpoints. If not specified, `runner.work_dir` will be used by default. If specified, the `out_dir` will be the concatenation of `out_dir` and the last level directory of `runner.work_dir`. *Changed in version 1.3.16.*
- **max_keep_ckpts** (*int, optional*) – The maximum checkpoints to keep. In some cases we want only the latest few checkpoints and would like to delete old ones to save the disk space. Default: -1, which means unlimited.
- **save_last** (*bool, optional*) – Whether to force the last checkpoint to be saved regardless of interval. Default: True.
- **sync_buffer** (*bool, optional*) – Whether to synchronize buffers in different gpus. Default: False.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None. *New in version 1.3.16.*

Warning: Before v1.3.16, the `out_dir` argument indicates the path where the checkpoint is stored. However, since v1.3.16, `out_dir` indicates the root directory and the final path to save checkpoint is the concatenation of `out_dir` and the last level directory of `runner.work_dir`. Suppose the value of `out_dir` is "/path/of/A" and the value of `runner.work_dir` is "/path/of/B", then the final path will be "/path/of/A/B".

class mmcv.runner.**CheckpointLoader**

A general checkpoint loader to manage all schemes.

classmethod **load_checkpoint**(*filename: str*, *map_location: Optional[Union[str, Callable]] = None*, *logger: Optional[logging.Logger] = None*) → Union[dict, collections.OrderedDict]

load checkpoint through URL scheme path.

Parameters

- **filename** (*str*) – checkpoint file name with given prefix
- **map_location** (*str, optional*) – Same as `torch.load()`. Default: None
- **logger** (`logging.Logger`, optional) – The logger for message. Default: None

Returns The loaded checkpoint.

Return type dict or OrderedDict

classmethod **register_scheme**(*prefixes: Union[str, List[str], Tuple[str, ...]]*, *loader: Optional[Callable] = None*, *force: bool = False*) → Callable

Register a loader to CheckpointLoader.

This method can be used as a normal class method or a decorator.

Parameters

- **prefixes** (*str or Sequence[str]*) –
- **prefix of the registered loader.** (*The*) –
- **loader** (*function, optional*) – The loader function to be registered. When this method is used as a decorator, loader is None. Defaults to None.
- **force** (*bool, optional*) – Whether to override the loader if the prefix has already been registered. Defaults to False.

class mmcv.runner.**ClearMLLoggerHook**(*init_kwargs: Optional[Dict] = None*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch: bool = True*)

Class to log metrics with clearml.

It requires clearml to be installed.

Parameters

- **init_kwargs** (*dict*) – A dict contains the *clearml.Task.init* initialization keys. See taskinit for more details.
- **interval** (*int*) – Logging interval (every k iterations). Default 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.

class mmcv.runner.**CosineAnnealingLrUpdaterHook**(*min_lr: Optional[float] = None*, *min_lr_ratio: Optional[float] = None*, ***kwargs*)

CosineAnnealing LR scheduler.

Parameters

- **min_lr** (*float, optional*) – The minimum lr. Default: None.
- **min_lr_ratio** (*float, optional*) – The ratio of minimum lr to the base lr. Either *min_lr* or *min_lr_ratio* should be specified. Default: None.

class mmcv.runner.**CosineAnnealingMomentumUpdaterHook**(*min_momentum: Optional[float] = None*, *min_momentum_ratio: Optional[float] = None*, ***kwargs*)

Cosine annealing LR Momentum decays the Momentum of each parameter group linearly.

Parameters

- **min_momentum** (*float, optional*) – The minimum momentum. Default: None.
- **min_momentum_ratio** (*float, optional*) – The ratio of minimum momentum to the base momentum. Either *min_momentum* or *min_momentum_ratio* should be specified. Default: None.

class mmcv.runner.**CosineRestartLrUpdaterHook**(*periods: List[int]*, *restart_weights: List[float] = [1]*, *min_lr: Optional[float] = None*, *min_lr_ratio: Optional[float] = None*, ***kwargs*)

Cosine annealing with restarts learning rate scheme.

Parameters

- **periods** (*list[int]*) – Periods for each cosine anneling cycle.
- **restart_weights** (*list[float]*) – Restart weights at each restart iteration. Defaults to [1].
- **min_lr** (*float, optional*) – The minimum lr. Default: None.
- **min_lr_ratio** (*float, optional*) – The ratio of minimum lr to the base lr. Either *min_lr* or *min_lr_ratio* should be specified. Default: None.

class mmcv.runner.**CyclicLrUpdaterHook**(*by_epoch: bool = False*, *target_ratio: Union[float, tuple] = (10, 0.0001)*, *cyclic_times: int = 1*, *step_ratio_up: float = 0.4*, *anneal_strategy: str = 'cos'*, *gamma: float = 1*, ***kwargs*)

Cyclic LR Scheduler.

Implement the cyclical learning rate policy (CLR) described in https://arxiv.org/pdf/1506.01186.pdf

Different from the original paper, we use cosine annealing rather than triangular policy inside a cycle. This improves the performance in the 3D detection area.

Parameters

- **by_epoch** (*bool, optional*) – Whether to update LR by epoch.
- **target_ratio** (*tuple[float], optional*) – Relative ratio of the highest LR and the lowest LR to the initial LR.
- **cyclic_times** (*int, optional*) – Number of cycles during training
- **step_ratio_up** (*float, optional*) – The ratio of the increasing process of LR in the total cycle.
- **anneal_strategy** (*str, optional*) – {'cos', 'linear'} Specifies the annealing strategy: 'cos' for cosine annealing, 'linear' for linear annealing. Default: 'cos'.
- **gamma** (*float, optional*) – Cycle decay ratio. Default: 1. It takes values in the range (0, 1]. The difference between the maximum learning rate and the minimum learning rate decreases periodically when it is less than 1. *New in version 1.4.4.*

class mmcv.runner.**CyclicMomentumUpdaterHook**(*by_epoch: bool = False*, *target_ratio: Tuple[float, float] = (0.8947368421052632, 1.0)*, *cyclic_times: int = 1*, *step_ratio_up: float = 0.4*, *anneal_strategy: str = 'cos'*, *gamma: float = 1.0*, ***kwargs*)

Cyclic momentum Scheduler.

Implement the cyclical momentum scheduler policy described in https://arxiv.org/pdf/1708.07120.pdf

This momentum scheduler usually used together with the CyclicLRUpdater to improve the performance in the 3D detection area.

Parameters

- **target_ratio** (*tuple[float]*) – Relative ratio of the lowest momentum and the highest momentum to the initial momentum.
- **cyclic_times** (*int*) – Number of cycles during training

- **step_ratio_up** (*float*) – The ratio of the increasing process of momentum in the total cycle.
- **by_epoch** (*bool*) – Whether to update momentum by epoch.
- **anneal_strategy** (*str, optional*) – {'cos', 'linear'} Specifies the annealing strategy: 'cos' for cosine annealing, 'linear' for linear annealing. Default: 'cos'.
- **gamma** (*float, optional*) – Cycle decay ratio. Default: 1. It takes values in the range (0, 1]. The difference between the maximum learning rate and the minimum learning rate decreases periodically when it is less than 1. *New in version 1.4.4.*

class mmcv.runner.**DefaultOptimizerConstructor**(*optimizer_cfg: Dict*, *paramwise_cfg: Optional[Dict] = None*)

Default constructor for optimizers.

By default each parameter share the same optimizer settings, and we provide an argument `paramwise_cfg` to specify parameter-wise settings. It is a dict and may contain the following fields:

- `custom_keys` (dict): Specified parameters-wise settings by keys. If one of the keys in `custom_keys` is a substring of the name of one parameter, then the setting of the parameter will be specified by `custom_keys[key]` and other setting like `bias_lr_mult` etc. will be ignored. It should be noted that the aforementioned `key` is the longest key that is a substring of the name of the parameter. If there are multiple matched keys with the same length, then the key with lower alphabet order will be chosen. `custom_keys[key]` should be a dict and may contain fields `lr_mult` and `decay_mult`. See Example 2 below.
- `bias_lr_mult` (float): It will be multiplied to the learning rate for all bias parameters (except for those in normalization layers and offset layers of DCN).
- `bias_decay_mult` (float): It will be multiplied to the weight decay for all bias parameters (except for those in normalization layers, depthwise conv layers, offset layers of DCN).
- `norm_decay_mult` (float): It will be multiplied to the weight decay for all weight and bias parameters of normalization layers.
- `dwconv_decay_mult` (float): It will be multiplied to the weight decay for all weight and bias parameters of depthwise conv layers.
- `dcn_offset_lr_mult` (float): It will be multiplied to the learning rate for parameters of offset layer in the deformable convs of a model.
- `bypass_duplicate` (bool): If true, the duplicate parameters would not be added into optimizer. Default: False.

Note: 1. If the option `dcn_offset_lr_mult` is used, the constructor will override the effect of `bias_lr_mult` in the bias of offset layer. So be careful when using both `bias_lr_mult` and `dcn_offset_lr_mult`. If you wish to apply both of them to the offset layer in deformable convs, set `dcn_offset_lr_mult` to the original `dcn_offset_lr_mult` * `bias_lr_mult`.

2. If the option `dcn_offset_lr_mult` is used, the constructor will apply it to all the DCN layers in the model. So be careful when the model contains multiple DCN layers in places other than backbone.

Parameters

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

Optional fields are

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

- **paramwise_cfg** (*dict, optional*) – Parameter-wise options.

Example 1:

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

Example 2:

```
>>> # assume model have attribute model.backbone and model.cls_head
>>> optimizer_cfg = dict(type='SGD', lr=0.01, weight_decay=0.95)
>>> paramwise_cfg = dict(custom_keys={
        'backbone': dict(lr_mult=0.1, decay_mult=0.9)})
>>> optim_builder = DefaultOptimizerConstructor(
>>>     optimizer_cfg, paramwise_cfg)
>>> optimizer = optim_builder(model)
>>> # Then the `lr` and `weight_decay` for model.backbone is
>>> # (0.01 * 0.1, 0.95 * 0.9). `lr` and `weight_decay` for
>>> # model.cls_head is (0.01, 0.95).
```

add_params(*params: List[Dict]*, *module: torch.nn.modules.module.Module*, *prefix: str = ''*, *is_dcn_module: Optional[Union[int, float]] = None*) → None

Add all parameters of module to the params list.

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

Parameters

- **params** (*list[dict]*) – A list of param groups, it will be modified in place.
- **module** (*nn.Module*) – The module to be added.
- **prefix** (*str*) – The prefix of the module
- **is_dcn_module** (*int|float|None*) – If the current module is a submodule of DCN, *is_dcn_module* will be passed to control conv_offset layer's learning rate. Defaults to None.

class mmcv.runner.**DefaultRunnerConstructor**(*runner_cfg: dict*, *default_args: Optional[dict] = None*)

Default constructor for runners.

Custom existing *Runner* like *EpocBasedRunner* though *RunnerConstructor*. For example, We can inject some new properties and functions for *Runner*.

Example

```
>>> from mmcv.runner import RUNNER_BUILDERS, build_runner
>>> # Define a new RunnerReconstructor
>>> @RUNNER_BUILDERS.register_module()
>>> class MyRunnerConstructor:
...     def __init__(self, runner_cfg, default_args=None):
...         if not isinstance(runner_cfg, dict):
...             raise TypeError('runner_cfg should be a dict',
...                             f'but got {type(runner_cfg)}')
...         self.runner_cfg = runner_cfg
...         self.default_args = default_args
...
...     def __call__(self):
...         runner = RUNNERS.build(self.runner_cfg,
...                                default_args=self.default_args)
...         # Add new properties for existing runner
...         runner.my_name = 'my_runner'
...         runner.my_function = lambda self: print(self.my_name)
...         ...
>>> # build your runner
>>> runner_cfg = dict(type='EpochBasedRunner', max_epochs=40,
...                   constructor='MyRunnerConstructor')
>>> runner = build_runner(runner_cfg)
```

class mmcv.runner.**DistEvalHook**(*dataloader: torch.utils.data.dataloader.DataLoader*, *start: Optional[int] = None*, *interval: int = 1*, *by_epoch: bool = True*, *save_best: Optional[str] = None*, *rule: Optional[str] = None*, *test_fn: Optional[Callable] = None*, *greater_keys: Optional[List[str]] = None*, *less_keys: Optional[List[str]] = None*, *broadcast_bn_buffer: bool = True*, *tmpdir: Optional[str] = None*, *gpu_collect: bool = False*, *out_dir: Optional[str] = None*, *file_client_args: Optional[dict] = None*, ***eval_kwargs*)

Distributed evaluation hook.

This hook will regularly perform evaluation in a given interval when performing in distributed environment.

Parameters

- **dataloader** (DataLoader) – A PyTorch dataloader, whose dataset has implemented `evaluate` function.
- **start** (*int | None, optional*) – Evaluation starting epoch. It enables evaluation before the training starts if `start` <= the resuming epoch. If None, whether to evaluate is merely decided by `interval`. Default: None.
- **interval** (*int*) – Evaluation interval. Default: 1.
- **by_epoch** (*bool*) – Determine perform evaluation by epoch or by iteration. If set to True, it will perform by epoch. Otherwise, by iteration. default: True.
- **save_best** (*str, optional*) – If a metric is specified, it would measure the best checkpoint during evaluation. The information about best checkpoint would be saved in `runner.meta['hook_msgs']` to keep best score value and best checkpoint path, which will be also loaded when resume checkpoint. Options are the evaluation metrics on the test dataset. e.g., `bbox_mAP`, `segm_mAP` for bbox detection and instance segmentation. `AR@100` for proposal recall. If `save_best` is `auto`, the first key of the returned `OrderedDict` result will be used. Default: None.

- **rule** (*str | None, optional*) – Comparison rule for best score. If set to None, it will infer a reasonable rule. Keys such as 'acc', 'top' .etc will be inferred by 'greater' rule. Keys contain 'loss' will be inferred by 'less' rule. Options are 'greater', 'less', None. Default: None.
- **test_fn** (*callable, optional*) – test a model with samples from a dataloader in a multi-gpu manner, and return the test results. If `None`, the default test function `mmcv.engine.multi_gpu_test` will be used. (default: `None`)
- **tmpdir** (*str | None*) – Temporary directory to save the results of all processes. Default: None.
- **gpu_collect** (*bool*) – Whether to use gpu or cpu to collect results. Default: False.
- **broadcast_bn_buffer** (*bool*) – Whether to broadcast the buffer(running_mean and running_var) of rank 0 to other rank before evaluation. Default: True.
- **out_dir** (*str, optional*) – The root directory to save checkpoints. If not specified, *runner.work_dir* will be used by default. If specified, the *out_dir* will be the concatenation of *out_dir* and the last level directory of *runner.work_dir*.
- **file_client_args** (*dict*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None.
- ****eval_kwargs** – Evaluation arguments fed into the evaluate function of the dataset.

class mmcv.runner.**DistSamplerSeedHook**

Data-loading sampler for distributed training.

When distributed training, it is only useful in conjunction with *EpochBasedRunner*, while *IterBasedRunner* achieves the same purpose with `IterLoader`.

class mmcv.runner.**DvcliveLoggerHook**(*model_file: Optional[str] = None*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch: bool = True*, ***kwargs*)

Class to log metrics with dvclive.

It requires dvclive to be installed.

Parameters

- **model_file** (*str*) – Default None. If not None, after each epoch the model will be saved to {model_file}.
- **interval** (*int*) – Logging interval (every k iterations). Default 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.
- **kwargs** – Arguments for instantiating Live.

class mmcv.runner.**EMAHook**(*momentum: float = 0.0002*, *interval: int = 1*, *warm_up: int = 100*, *resume_from: Optional[str] = None*)

Exponential Moving Average Hook.

Use Exponential Moving Average on all parameters of model in training process. All parameters have a ema backup, which update by the formula as below. EMAHook takes priority over EvalHook and CheckpointSaverHook.

$$Xema\_t + 1 = (1 - \text{momentum}) \times Xema\_t + \text{momentum} \times X_t$$

Parameters

- **momentum** (*float*) – The momentum used for updating ema parameter. Defaults to 0.0002.
- **interval** (*int*) – Update ema parameter every interval iteration. Defaults to 1.
- **warm_up** (*int*) – During first warm_up steps, we may use smaller momentum to update ema parameters more slowly. Defaults to 100.
- **resume_from** (*str, optional*) – The checkpoint path. Defaults to None.

after_train_epoch(*runner*)
We load parameter values from ema backup to model before the EvalHook.

after_train_iter(*runner*)
Update ema parameter every self.interval iterations.

before_run(*runner*)
To resume model with it's ema parameters more friendly.

Register ema parameter as `named_buffer` to model

before_train_epoch(*runner*)
We recover model's parameter from ema backup after last epoch's EvalHook.

class mmcv.runner.**EpochBasedRunner**(*model: torch.nn.modules.module.Module*, *batch_processor: Optional[Callable] = None*, *optimizer: Optional[Union[Dict, torch.optim.optimizer.Optimizer]] = None*, *work_dir: Optional[str] = None*, *logger: Optional[logging.Logger] = None*, *meta: Optional[Dict] = None*, *max_iters: Optional[int] = None*, *max_epochs: Optional[int] = None*)

Epoch-based Runner.

This runner train models epoch by epoch.

run(*data_loaders: List[*torch.utils.data.dataloader.DataLoader*]*, *workflow: List[Tuple[str, int]]*, *max_epochs: Optional[int] = None*, ***kwargs*) → None
Start running.

Parameters

- **data_loaders** (list[DataLoader]) – Dataloaders for training and validation.
- **workflow** (*list[tuple]*) – A list of (phase, epochs) to specify the running order and epochs. E.g, [('train', 2), ('val', 1)] means running 2 epochs for training and 1 epoch for validation, iteratively.

save_checkpoint(*out_dir: str*, *filename_tmpl: str = 'epoch_{}.pth'*, *save_optimizer: bool = True*, *meta: Optional[Dict] = None*, *create_symlink: bool = True*) → None
Save the checkpoint.

Parameters

- **out_dir** (*str*) – The directory that checkpoints are saved.
- **filename_tmpl** (*str, optional*) – The checkpoint filename template, which contains a placeholder for the epoch number. Defaults to 'epoch_{}.pth'.
- **save_optimizer** (*bool, optional*) – Whether to save the optimizer to the checkpoint. Defaults to True.

- **meta** (*dict, optional*) – The meta information to be saved in the checkpoint. Defaults to None.
- **create_symlink** (*bool, optional*) – Whether to create a symlink "latest.pth" to point to the latest checkpoint. Defaults to True.

class mmcv.runner.**EvalHook**(*dataloader: torch.utils.data.dataloader.DataLoader, start: Optional[int] = None, interval: int = 1, by_epoch: bool = True, save_best: Optional[str] = None, rule: Optional[str] = None, test_fn: Optional[Callable] = None, greater_keys: Optional[List[str]] = None, less_keys: Optional[List[str]] = None, out_dir: Optional[str] = None, file_client_args: Optional[dict] = None, **eval_kwargs*)

Non-Distributed evaluation hook.

This hook will regularly perform evaluation in a given interval when performing in non-distributed environment.

Parameters

- **dataloader** (DataLoader) – A PyTorch dataloader, whose dataset has implemented `evaluate` function.
- **start** (*int | None, optional*) – Evaluation starting epoch. It enables evaluation before the training starts if `start` <= the resuming epoch. If None, whether to evaluate is merely decided by `interval`. Default: None.
- **interval** (*int*) – Evaluation interval. Default: 1.
- **by_epoch** (*bool*) – Determine perform evaluation by epoch or by iteration. If set to True, it will perform by epoch. Otherwise, by iteration. Default: True.
- **save_best** (*str, optional*) – If a metric is specified, it would measure the best checkpoint during evaluation. The information about best checkpoint would be saved in `runner.meta['hook_msgs']` to keep best score value and best checkpoint path, which will be also loaded when resume checkpoint. Options are the evaluation metrics on the test dataset. e.g., `bbox_mAP`, `segm_mAP` for bbox detection and instance segmentation. `AR@100` for proposal recall. If `save_best` is `auto`, the first key of the returned `OrderedDict` result will be used. Default: None.
- **rule** (*str | None, optional*) – Comparison rule for best score. If set to None, it will infer a reasonable rule. Keys such as 'acc', 'top' .etc will be inferred by 'greater' rule. Keys contain 'loss' will be inferred by 'less' rule. Options are 'greater', 'less', None. Default: None.
- **test_fn** (*callable, optional*) – test a model with samples from a dataloader, and return the test results. If `None`, the default test function `mmcv.engine.single_gpu_test` will be used. (default: `None`)
- **greater_keys** (*List[str] | None, optional*) – Metric keys that will be inferred by 'greater' comparison rule. If `None`, _default_greater_keys will be used. (default: `None`)
- **less_keys** (*List[str] | None, optional*) – Metric keys that will be inferred by 'less' comparison rule. If `None`, _default_less_keys will be used. (default: `None`)
- **out_dir** (*str, optional*) – The root directory to save checkpoints. If not specified, *runner.work_dir* will be used by default. If specified, the *out_dir* will be the concatenation of *out_dir* and the last level directory of *runner.work_dir*. *New in version 1.3.16.*
- **file_client_args** (*dict*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None. *New in version 1.3.16.*
- ****eval_kwargs** – Evaluation arguments fed into the evaluate function of the dataset.

Note: If new arguments are added for EvalHook, tools/test.py, tools/eval_metric.py may be affected.

after_train_epoch(*runner*)
Called after every training epoch to evaluate the results.

after_train_iter(*runner*)
Called after every training iter to evaluate the results.

before_train_epoch(*runner*)
Evaluate the model only at the start of training by epoch.

before_train_iter(*runner*)
Evaluate the model only at the start of training by iteration.

evaluate(*runner*, *results*)
Evaluate the results.

Parameters

- **runner** (mmcv.Runner) – The underlined training runner.
- **results** (*list*) – Output results.

class mmcv.runner.**ExpLrUpdaterHook**(*gamma: float*, ***kwargs*)

class mmcv.runner.**FixedLrUpdaterHook**(***kwargs*)

class mmcv.runner.**FlatCosineAnnealingLrUpdaterHook**(*start_percent: float = 0.75*, *min_lr: Optional[float] = None*, *min_lr_ratio: Optional[float] = None*, ***kwargs*)

Flat + Cosine lr schedule.

Modified from https://github.com/fastai/fastai/blob/master/fastai/callback/schedule.py#L128 # noqa: E501

Parameters

- **start_percent** (*float*) – When to start annealing the learning rate after the percentage of the total training steps. The value should be in range [0, 1). Default: 0.75
- **min_lr** (*float, optional*) – The minimum lr. Default: None.
- **min_lr_ratio** (*float, optional*) – The ratio of minimum lr to the base lr. Either *min_lr* or *min_lr_ratio* should be specified. Default: None.

class mmcv.runner.**Fp16OptimizerHook**(*grad_clip: Optional[dict] = None*, *coalesce: bool = True*, *bucket_size_mb: int = - 1*, *loss_scale: Union[float, str, dict] = 512.0*, *distributed: bool = True*)

FP16 optimizer hook (using PyTorch's implementation).

If you are using PyTorch >= 1.6, torch.cuda.amp is used as the backend, to take care of the optimization procedure.

Parameters **loss_scale** (*float | str | dict*) – Scale factor configuration. If loss_scale is a float, static loss scaling will be used with the specified scale. If loss_scale is a string, it must be 'dynamic', then dynamic loss scaling will be used. It can also be a dict containing arguments of GradScalar. Defaults to 512. For Pytorch >= 1.6, mmcv uses official implementation of GradScaler. If you use a dict version of loss_scale to create GradScaler, please refer to: https://pytorch.org/docs/stable/amp.html#torch.cuda.amp.GradScaler for the parameters.

Examples

```
>>> loss_scale = dict(
...     init_scale=65536.0,
...     growth_factor=2.0,
...     backoff_factor=0.5,
...     growth_interval=2000
... )
>>> optimizer_hook = Fp16OptimizerHook(loss_scale=loss_scale)
```

after_train_iter(*runner*) → None
Backward optimization steps for Mixed Precision Training. For dynamic loss scaling, please refer to https://pytorch.org/docs/stable/amp.html#torch.cuda.amp.GradScaler.

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

before_run(*runner*) → None
Preparing steps before Mixed Precision Training.

copy_grads_to_fp32(*fp16_net: torch.nn.modules.module.Module*, *fp32_weights: torch.Tensor*) → None
Copy gradients from fp16 model to fp32 weight copy.

copy_params_to_fp16(*fp16_net: torch.nn.modules.module.Module*, *fp32_weights: torch.Tensor*) → None
Copy updated params from fp32 weight copy to fp16 model.

class mmcv.runner.**GradientCumulativeFp16OptimizerHook**(**args*, ***kwargs*)
Fp16 optimizer Hook (using PyTorch's implementation) implements multi-iters gradient cumulating.

If you are using PyTorch >= 1.6, torch.cuda.amp is used as the backend, to take care of the optimization procedure.

after_train_iter(*runner*) → None
Backward optimization steps for Mixed Precision Training. For dynamic loss scaling, please refer to https://pytorch.org/docs/stable/amp.html#torch.cuda.amp.GradScaler.

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

class mmcv.runner.**GradientCumulativeOptimizerHook**(*cumulative_iters: int = 1*, ***kwargs*)
Optimizer Hook implements multi-iters gradient cumulating.

Parameters **cumulative_iters** (*int, optional*) – Num of gradient cumulative iters. The optimizer will step every *cumulative_iters* iters. Defaults to 1.

Examples

```
>>> # Use cumulative_iters to simulate a large batch size
>>> # It is helpful when the hardware cannot handle a large batch size.
>>> loader = DataLoader(data, batch_size=64)
>>> optim_hook = GradientCumulativeOptimizerHook(cumulative_iters=4)
>>> # almost equals to
>>> loader = DataLoader(data, batch_size=256)
>>> optim_hook = OptimizerHook()
```

class mmcv.runner.**InvLrUpdaterHook**(*gamma: float*, *power: float = 1.0*, ***kwargs*)

class mmcv.runner.**IterBasedRunner**(*model: torch.nn.modules.module.Module*, *batch_processor: Optional[Callable] = None*, *optimizer: Optional[Union[Dict, torch.optim.optimizer.Optimizer]] = None*, *work_dir: Optional[str] = None*, *logger: Optional[logging.Logger] = None*, *meta: Optional[Dict] = None*, *max_iters: Optional[int] = None*, *max_epochs: Optional[int] = None*)

Iteration-based Runner.

This runner train models iteration by iteration.

register_training_hooks(*lr_config*, *optimizer_config=None*, *checkpoint_config=None*, *log_config=None*, *momentum_config=None*, *custom_hooks_config=None*)

Register default hooks for iter-based training.

Checkpoint hook, optimizer stepper hook and logger hooks will be set to *by_epoch=False* by default.

Default hooks include:

Hooks	Priority
LrUpdaterHook	VERY_HIGH (10)
MomentumUpdaterHook	HIGH (30)
OptimizerStepperHook	ABOVE_NORMAL (40)
CheckpointSaverHook	NORMAL (50)
IterTimerHook	LOW (70)
LoggerHook(s)	VERY_LOW (90)
CustomHook(s)	defaults to NORMAL (50)

If custom hooks have same priority with default hooks, custom hooks will be triggered after default hooks.

resume(*checkpoint: str*, *resume_optimizer: bool = True*, *map_location: Union[str, Callable] = 'default'*) → None

Resume model from checkpoint.

Parameters

- **checkpoint** (*str*) – Checkpoint to resume from.
- **resume_optimizer** (*bool, optional*) – Whether resume the optimizer(s) if the checkpoint file includes optimizer(s). Default to True.
- **map_location** (*str, optional*) – Same as `torch.load()`. Default to 'default'.

run(*data_loaders: List[*torch.utils.data.dataloader.DataLoader*]*, *workflow: List[Tuple[str, int]]*, *max_iters: Optional[int] = None*, ***kwargs*) → None

Start running.

Parameters

- **data_loaders** (list[DataLoader]) – Dataloaders for training and validation.
- **workflow** (*list[tuple]*) – A list of (phase, iters) to specify the running order and iterations. E.g, [('train', 10000), ('val', 1000)] means running 10000 iterations for training and 1000 iterations for validation, iteratively.

save_checkpoint(*out_dir: str*, *filename_tmpl: str = 'iter_{}.pth'*, *meta: Optional[Dict] = None*, *save_optimizer: bool = True*, *create_symlink: bool = True*) → None

Save checkpoint to file.

Parameters

- **out_dir** (*str*) – Directory to save checkpoint files.
- **filename_tmpl** (*str, optional*) – Checkpoint file template. Defaults to 'iter_{}.pth'.
- **meta** (*dict, optional*) – Metadata to be saved in checkpoint. Defaults to None.
- **save_optimizer** (*bool, optional*) – Whether save optimizer. Defaults to True.
- **create_symlink** (*bool, optional*) – Whether create symlink to the latest checkpoint file. Defaults to True.

class mmcv.runner.**LinearAnnealingLrUpdaterHook**(*min_lr: Optional[float] = None*, *min_lr_ratio: Optional[float] = None*, ***kwargs*)

Linear annealing LR Scheduler decays the learning rate of each parameter group linearly.

Parameters

- **min_lr** (*float, optional*) – The minimum lr. Default: None.
- **min_lr_ratio** (*float, optional*) – The ratio of minimum lr to the base lr. Either *min_lr* or *min_lr_ratio* should be specified. Default: None.

class mmcv.runner.**LinearAnnealingMomentumUpdaterHook**(*min_momentum: Optional[float] = None*, *min_momentum_ratio: Optional[float] = None*, ***kwargs*)

Linear annealing LR Momentum decays the Momentum of each parameter group linearly.

Parameters

- **min_momentum** (*float, optional*) – The minimum momentum. Default: None.
- **min_momentum_ratio** (*float, optional*) – The ratio of minimum momentum to the base momentum. Either *min_momentum* or *min_momentum_ratio* should be specified. Default: None.

class mmcv.runner.**LoggerHook**(*interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch: bool = True*)

Base class for logger hooks.

Parameters

- **interval** (*int*) – Logging interval (every k iterations). Default 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default True.

get_iter(*runner*, *inner_iter: bool = False*) → int

Get the current training iteration step.

static is_scalar(*val*, *include_np: bool = True*, *include_torch: bool = True*) → bool
Tell the input variable is a scalar or not.

Parameters

- **val** – Input variable.
- **include_np** (*bool*) – Whether include 0-d np.ndarray as a scalar.
- **include_torch** (*bool*) – Whether include 0-d torch.Tensor as a scalar.

Returns True or False.

Return type bool

class mmcv.runner.**LossScaler**(*init_scale: float = 4294967296*, *mode: str = 'dynamic'*, *scale_factor: float = 2.0*, *scale_window: int = 1000*)
Class that manages loss scaling in mixed precision training which supports both dynamic or static mode.

The implementation refers to https://github.com/NVIDIA/apex/blob/master/apex/fp16_utils/loss_scaler.py. Indirectly, by supplying `mode='dynamic'` for dynamic loss scaling. It's important to understand how *LossScaler* operates. Loss scaling is designed to combat the problem of underflowing gradients encountered at long times when training fp16 networks. Dynamic loss scaling begins by attempting a very high loss scale. Ironically, this may result in OVERflowing gradients. If overflowing gradients are encountered, `FP16_Optimizer` then skips the update step for this particular iteration/minibatch, and *LossScaler* adjusts the loss scale to a lower value. If a certain number of iterations occur without overflowing gradients detected,:class:*LossScaler* increases the loss scale once more. In this way *LossScaler* attempts to "ride the edge" of always using the highest loss scale possible without incurring overflow.

Parameters

- **init_scale** (*float*) – Initial loss scale value, default: 2**32.
- **scale_factor** (*float*) – Factor used when adjusting the loss scale. Default: 2.
- **mode** (*str*) – Loss scaling mode. 'dynamic' or 'static'
- **scale_window** (*int*) – Number of consecutive iterations without an overflow to wait before increasing the loss scale. Default: 1000.

has_overflow(*params: List[torch.nn.parameter.Parameter]*) → bool
Check if params contain overflow.

load_state_dict(*state_dict: dict*) → None
Loads the loss_scaler state dict.

Parameters **state_dict** (*dict*) – scaler state.

state_dict() → dict
Returns the state of the scaler as a `dict`.

update_scale(*overflow: bool*) → None
update the current loss scale value when overflow happens.

class mmcv.runner.**LrUpdaterHook**(*by_epoch: bool = True*, *warmup: Optional[str] = None*, *warmup_iters: int = 0*, *warmup_ratio: float = 0.1*, *warmup_by_epoch: bool = False*)
LR Scheduler in MMCV.

Parameters

- **by_epoch** (*bool*) – LR changes epoch by epoch
- **warmup** (*string*) – Type of warmup used. It can be None(use no warmup), 'constant', 'linear' or 'exp'

- **warmup_iters** (*int*) – The number of iterations or epochs that warmup lasts
- **warmup_ratio** (*float*) – LR used at the beginning of warmup equals to warmup_ratio * initial_lr
- **warmup_by_epoch** (*bool*) – When warmup_by_epoch == True, warmup_iters means the number of epochs that warmup lasts, otherwise means the number of iteration that warmup lasts

class mmcv.runner.**MlflowLoggerHook**(*exp_name: Optional[str] = None*, *tags: Optional[Dict] = None*, *params: Optional[Dict] = None*, *log_model: bool = True*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch: bool = True*)

Class to log metrics and (optionally) a trained model to MLflow.

It requires MLflow to be installed.

Parameters

- **exp_name** (*str, optional*) – Name of the experiment to be used. Default None. If not None, set the active experiment. If experiment does not exist, an experiment with provided name will be created.
- **tags** (*Dict[str], optional*) – Tags for the current run. Default None. If not None, set tags for the current run.
- **params** (*Dict[str], optional*) – Params for the current run. Default None. If not None, set params for the current run.
- **log_model** (*bool, optional*) – Whether to log an MLflow artifact. Default True. If True, log runner.model as an MLflow artifact for the current run.
- **interval** (*int*) – Logging interval (every k iterations). Default: 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.

class mmcv.runner.**ModuleDict**(*modules: Optional[dict] = None*, *init_cfg: Optional[dict] = None*)

ModuleDict in openmmlab.

Parameters

- **modules** (*dict, optional*) – a mapping (dictionary) of (string: module) or an iterable of key-value pairs of type (string, module).
- **init_cfg** (*dict, optional*) – Initialization config dict.

class mmcv.runner.**ModuleList**(*modules: Optional[Iterable] = None*, *init_cfg: Optional[dict] = None*)

ModuleList in openmmlab.

Parameters

- **modules** (*iterable, optional*) – an iterable of modules to add.
- **init_cfg** (*dict, optional*) – Initialization config dict.

class mmcv.runner.**NeptuneLoggerHook**(*init_kwargs: Optional[Dict] = None*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = True*, *with_step: bool = True*, *by_epoch: bool = True*)

Class to log metrics to NeptuneAI.

It requires Neptune to be installed.

Parameters

- **init_kwargs** (*dict*) – a dict contains the initialization keys as below:
  - project (str): Name of a project in a form of namespace/project_name. If None, the value of NEPTUNE_PROJECT environment variable will be taken.
  - api_token (str): User's API token. If None, the value of NEPTUNE_API_TOKEN environment variable will be taken. Note: It is strongly recommended to use NEPTUNE_API_TOKEN environment variable rather than placing your API token in plain text in your source code.
  - name (str, optional, default is 'Untitled'): Editable name of the run. Name is displayed in the run's Details and in Runs table as a column.

  Check https://docs.neptune.ai/api-reference/neptune#init for more init arguments.
- **interval** (*int*) – Logging interval (every k iterations). Default: 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than `interval`. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: True.
- **with_step** (*bool*) – If True, the step will be logged from `self.get_iters`. Otherwise, step will not be logged. Default: True.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.

class mmcv.runner.**OneCycleLrUpdaterHook**(*max_lr: Union[float, List]*, *total_steps: Optional[int] = None*, *pct_start: float = 0.3*, *anneal_strategy: str = 'cos'*, *div_factor: float = 25*, *final_div_factor: float = 10000.0*, *three_phase: bool = False*, ***kwargs*)

One Cycle LR Scheduler.

The 1cycle learning rate policy changes the learning rate after every batch. The one cycle learning rate policy is described in https://arxiv.org/pdf/1708.07120.pdf

Parameters

- **max_lr** (*float or list*) – Upper learning rate boundaries in the cycle for each parameter group.
- **total_steps** (*int, optional*) – The total number of steps in the cycle. Note that if a value is not provided here, it will be the max_iter of runner. Default: None.
- **pct_start** (*float*) – The percentage of the cycle (in number of steps) spent increasing the learning rate. Default: 0.3
- **anneal_strategy** (*str*) – {'cos', 'linear'} Specifies the annealing strategy: 'cos' for cosine annealing, 'linear' for linear annealing. Default: 'cos'
- **div_factor** (*float*) – Determines the initial learning rate via initial_lr = max_lr/div_factor Default: 25
- **final_div_factor** (*float*) – Determines the minimum learning rate via min_lr = initial_lr/final_div_factor Default: 1e4
- **three_phase** (*bool*) – If three_phase is True, use a third phase of the schedule to annihilate the learning rate according to final_div_factor instead of modifying the second phase (the first two phases will be symmetrical about the step indicated by pct_start). Default: False

class mmcv.runner.**OneCycleMomentumUpdaterHook**(*base_momentum: Union[float, list, dict] = 0.85*, *max_momentum: Union[float, list, dict] = 0.95*, *pct_start: float = 0.3*, *anneal_strategy: str = 'cos'*, *three_phase: bool = False*, ***kwargs*)

OneCycle momentum Scheduler.

This momentum scheduler usually used together with the OneCycleLrUpdater to improve the performance.

Parameters

- **base_momentum** (*float or list*) – Lower momentum boundaries in the cycle for each parameter group. Note that momentum is cycled inversely to learning rate; at the peak of a cycle, momentum is 'base_momentum' and learning rate is 'max_lr'. Default: 0.85
- **max_momentum** (*float or list*) – Upper momentum boundaries in the cycle for each parameter group. Functionally, it defines the cycle amplitude (max_momentum - base_momentum). Note that momentum is cycled inversely to learning rate; at the start of a cycle, momentum is 'max_momentum' and learning rate is 'base_lr' Default: 0.95
- **pct_start** (*float*) – The percentage of the cycle (in number of steps) spent increasing the learning rate. Default: 0.3
- **anneal_strategy** (*str*) – {'cos', 'linear'} Specifies the annealing strategy: 'cos' for cosine annealing, 'linear' for linear annealing. Default: 'cos'
- **three_phase** (*bool*) – If three_phase is True, use a third phase of the schedule to annihilate the learning rate according to final_div_factor instead of modifying the second phase (the first two phases will be symmetrical about the step indicated by pct_start). Default: False

class mmcv.runner.**OptimizerHook**(*grad_clip: Optional[dict] = None*, *detect_anomalous_params: bool = False*)

A hook contains custom operations for the optimizer.

Parameters

- **grad_clip** (*dict, optional*) – A config dict to control the clip_grad. Default: None.
- **detect_anomalous_params** (*bool*) – This option is only used for debugging which will slow down the training speed. Detect anomalous parameters that are not included in the computational graph with *loss* as the root. There are two cases
  - Parameters were not used during forward pass.
  - Parameters were not used to produce loss.

  Default: False.

class mmcv.runner.**PaviLoggerHook**(*init_kwargs: Optional[Dict] = None*, *add_graph: Optional[bool] = None*, *img_key: Optional[str] = None*, *add_last_ckpt: bool = False*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch: bool = True*, *add_graph_kwargs: Optional[Dict] = None*, *add_ckpt_kwargs: Optional[Dict] = None*)

Class to visual model, log metrics (for internal use).

Parameters

- **init_kwargs** (*dict*) – A dict contains the initialization keys as below:
  - name (str, optional): Custom training name. Defaults to None, which means current work_dir.
  - project (str, optional): Project name. Defaults to "default".
  - model (str, optional): Training model name. Defaults to current model.

  - session_text (str, optional): Session string in YAML format. Defaults to current config.
  - training_id (int, optional): Training ID in PAVI, if you want to use an existing training. Defaults to None.
  - compare_id (int, optional): Compare ID in PAVI, if you want to add the task to an existing compare. Defaults to None.
  - overwrite_last_training (bool, optional): Whether to upload data to the training with the same name in the same project, rather than creating a new one. Defaults to False.
- **add_graph** (*bool, optional*) – **Deprecated**. Whether to visual model. Default: False.
- **img_key** (*str, optional*) – **Deprecated**. Image key. Defaults to None.
- **add_last_ckpt** (*bool*) – Whether to save checkpoint after run. Default: False.
- **interval** (*int*) – Logging interval (every k iterations). Default: True.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.
- **add_graph_kwargs** (*dict, optional*) – A dict contains the params for adding graph, the keys are as below: - active (bool): Whether to use `add_graph`. Default: False. - start (int): The epoch or iteration to start. Default: 0. - interval (int): Interval of `add_graph`. Default: 1. - img_key (str): Get image data from Dataset. Default: 'img'. - opset_version (int): `opset_version` of exporting onnx.

  Default: 11.

  - **dummy_forward_kwargs (dict, optional): Set default parameters to** model forward function except image. For example, you can set {'return_loss': False} for mmcls. Default: None.
- **add_ckpt_kwargs** (*dict, optional*) – A dict contains the params for adding checkpoint, the keys are as below: - active (bool): Whether to upload checkpoint. Default: False. - start (int): The epoch or iteration to start. Default: 0. - interval (int): Interval of upload checkpoint. Default: 1.

get_step(*runner*) → int
Get the total training step/epoch.

class mmcv.runner.**PolyLrUpdaterHook**(*power: float = 1.0*, *min_lr: float = 0.0*, ***kwargs*)

class mmcv.runner.**Priority**(*value*)
Hook priority levels.

Level	Value
HIGHEST	0
VERY_HIGH	10
HIGH	30
ABOVE_NORMAL	40
NORMAL	50
BELOW_NORMAL	60
LOW	70
VERY_LOW	90
LOWEST	100

class mmcv.runner.**Runner**(**args*, ***kwargs*)
Deprecated name of EpochBasedRunner.

class mmcv.runner.**SegmindLoggerHook**(*interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch=True*)
Class to log metrics to Segmind.

It requires Segmind to be installed.

Parameters

- **interval** (*int*) – Logging interval (every k iterations). Default: 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default True.

class mmcv.runner.**Sequential**(**args*, *init_cfg: Optional[dict] = None*)
Sequential module in openmmlab.

Parameters **init_cfg** (*dict, optional*) – Initialization config dict.

class mmcv.runner.**StepLrUpdaterHook**(*step: Union[int, List[int]]*, *gamma: float = 0.1*, *min_lr: Optional[float] = None*, ***kwargs*)
Step LR scheduler with min_lr clipping.

Parameters

- **step** (*int | list[int]*) – Step to decay the LR. If an int value is given, regard it as the decay interval. If a list is given, decay LR at these steps.
- **gamma** (*float*) – Decay LR ratio. Defaults to 0.1.
- **min_lr** (*float, optional*) – Minimum LR value to keep. If LR after decay is lower than *min_lr*, it will be clipped to this value. If None is given, we don't perform lr clipping. Default: None.

class mmcv.runner.**StepMomentumUpdaterHook**(*step: Union[int, List[int]]*, *gamma: float = 0.5*, *min_momentum: Optional[float] = None*, ***kwargs*)
Step momentum scheduler with min value clipping.

Parameters

- **step** (*int | list[int]*) – Step to decay the momentum. If an int value is given, regard it as the decay interval. If a list is given, decay momentum at these steps.
- **gamma** (*float, optional*) – Decay momentum ratio. Default: 0.5.
- **min_momentum** (*float, optional*) – Minimum momentum value to keep. If momentum after decay is lower than this value, it will be clipped accordingly. If None is given, we don't perform lr clipping. Default: None.

class mmcv.runner.**SyncBuffersHook**(*distributed: bool = True*)
Synchronize model buffers such as running_mean and running_var in BN at the end of each epoch.

Parameters **distributed** (*bool*) – Whether distributed training is used. It is effective only for distributed training. Defaults to True.

after_epoch(*runner*)
All-reduce model buffers at the end of each epoch.

class mmcv.runner.**TensorboardLoggerHook**(*log_dir: Optional[str] = None*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *by_epoch: bool = True*)

Class to log metrics to Tensorboard.

Parameters

- **log_dir** (*string*) – Save directory location. Default: None. If default values are used, directory location is `runner.work_dir`/tf_logs.
- **interval** (*int*) – Logging interval (every k iterations). Default: True.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: False.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.

class mmcv.runner.**TextLoggerHook**(*by_epoch: bool = True*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *interval_exp_name: int = 1000*, *out_dir: Optional[str] = None*, *out_suffix: Union[str, tuple] = ('.log.json', '.log', '.py')*, *keep_local: bool = True*, *file_client_args: Optional[Dict] = None*)

Logger hook in text.

In this logger hook, the information will be printed on terminal and saved in json file.

Parameters

- **by_epoch** (*bool, optional*) – Whether EpochBasedRunner is used. Default: True.
- **interval** (*int, optional*) – Logging interval (every k iterations). Default: 10.
- **ignore_last** (*bool, optional*) – Ignore the log of last iterations in each epoch if less than `interval`. Default: True.
- **reset_flag** (*bool, optional*) – Whether to clear the output buffer after logging. Default: False.
- **interval_exp_name** (*int, optional*) – Logging interval for experiment name. This feature is to help users conveniently get the experiment information from screen or log file. Default: 1000.
- **out_dir** (*str, optional*) – Logs are saved in `runner.work_dir` default. If `out_dir` is specified, logs will be copied to a new directory which is the concatenation of `out_dir` and the last level directory of `runner.work_dir`. Default: None. *New in version 1.3.16.*
- **out_suffix** (*str or tuple[str], optional*) – Those filenames ending with `out_suffix` will be copied to `out_dir`. Default: ('.log.json', '.log', '.py'). *New in version 1.3.16.*
- **keep_local** (*bool, optional*) – Whether to keep local log when `out_dir` is specified. If False, the local log will be removed. Default: True. *New in version 1.3.16.*
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None. *New in version 1.3.16.*

class mmcv.runner.**WandbLoggerHook**(*init_kwargs: Optional[Dict] = None*, *interval: int = 10*, *ignore_last: bool = True*, *reset_flag: bool = False*, *commit: bool = True*, *by_epoch: bool = True*, *with_step: bool = True*, *log_artifact: bool = True*, *out_suffix: Union[str, tuple] = ('.log.json', '.log', '.py')*)

Class to log metrics with wandb.

It requires wandb to be installed.

Parameters

- **init_kwargs** (*dict*) – A dict contains the initialization keys. Check https://docs.wandb.ai/ref/python/init for more init arguments.
- **interval** (*int*) – Logging interval (every k iterations). Default 10.
- **ignore_last** (*bool*) – Ignore the log of last iterations in each epoch if less than *interval*. Default: True.
- **reset_flag** (*bool*) – Whether to clear the output buffer after logging. Default: False.
- **commit** (*bool*) – Save the metrics dict to the wandb server and increment the step. If false `wandb.log` just updates the current metrics dict with the row argument and metrics won't be saved until `wandb.log` is called with `commit=True`. Default: True.
- **by_epoch** (*bool*) – Whether EpochBasedRunner is used. Default: True.
- **with_step** (*bool*) – If True, the step will be logged from `self.get_iters`. Otherwise, step will not be logged. Default: True.
- **log_artifact** (*bool*) – If True, artifacts in {work_dir} will be uploaded to wandb after training ends. Default: True *New in version 1.4.3.*
- **out_suffix** (*str or tuple[str], optional*) – Those filenames ending with `out_suffix` will be uploaded to wandb. Default: ('.log.json', '.log', '.py'). *New in version 1.4.3.*

mmcv.runner.**allreduce_grads**(*params: List[torch.nn.parameter.Parameter]*, *coalesce: bool = True*, *bucket_size_mb: int = - 1*) → None

Allreduce gradients.

Parameters

- **params** (*list[torch.nn.Parameter]*) – List of parameters of a model.
- **coalesce** (*bool, optional*) – Whether allreduce parameters as a whole. Defaults to True.
- **bucket_size_mb** (*int, optional*) – Size of bucket, the unit is MB. Defaults to -1.

mmcv.runner.**allreduce_params**(*params: List[torch.nn.parameter.Parameter]*, *coalesce: bool = True*, *bucket_size_mb: int = - 1*) → None

Allreduce parameters.

Parameters

- **params** (*list[torch.nn.Parameter]*) – List of parameters or buffers of a model.
- **coalesce** (*bool, optional*) – Whether allreduce parameters as a whole. Defaults to True.
- **bucket_size_mb** (*int, optional*) – Size of bucket, the unit is MB. Defaults to -1.

mmcv.runner.**auto_fp16**(*apply_to: Optional[Iterable] = None*, *out_fp32: bool = False*, *supported_types: tuple = (<class 'torch.nn.modules.module.Module'>,)*) → Callable

Decorator to enable fp16 training automatically.

This decorator is useful when you write custom modules and want to support mixed precision training. If inputs arguments are fp32 tensors, they will be converted to fp16 automatically. Arguments other than fp32 tensors are ignored. If you are using PyTorch >= 1.6, torch.cuda.amp is used as the backend, otherwise, original mmcv implementation will be adopted.

Parameters

- **apply_to** (*Iterable, optional*) – The argument names to be converted. *None* indicates all arguments.

- **out_fp32** (*bool*) – Whether to convert the output back to fp32.
- **supported_types** (*tuple*) – Classes can be decorated by `auto_fp16`. *New in version 1.5.0.*

Example

```
>>> import torch.nn as nn
>>> class MyModule1(nn.Module):
>>>
>>>     # Convert x and y to fp16
>>>     @auto_fp16()
>>>     def forward(self, x, y):
>>>         pass
```

```
>>> import torch.nn as nn
>>> class MyModule2(nn.Module):
>>>
>>>     # convert pred to fp16
>>>     @auto_fp16(apply_to=('pred', ))
>>>     def do_something(self, pred, others):
>>>         pass
```

mmcv.runner.**force_fp32**(*apply_to: Optional[Iterable] = None*, *out_fp16: bool = False*) → Callable

Decorator to convert input arguments to fp32 in force.

This decorator is useful when you write custom modules and want to support mixed precision training. If there are some inputs that must be processed in fp32 mode, then this decorator can handle it. If inputs arguments are fp16 tensors, they will be converted to fp32 automatically. Arguments other than fp16 tensors are ignored. If you are using PyTorch >= 1.6, torch.cuda.amp is used as the backend, otherwise, original mmcv implementation will be adopted.

Parameters

- **apply_to** (*Iterable, optional*) – The argument names to be converted. *None* indicates all arguments.
- **out_fp16** (*bool*) – Whether to convert the output back to fp16.

Example

```
>>> import torch.nn as nn
>>> class MyModule1(nn.Module):
>>>
>>>     # Convert x and y to fp32
>>>     @force_fp32()
>>>     def loss(self, x, y):
>>>         pass
```

```
>>> import torch.nn as nn
>>> class MyModule2(nn.Module):
>>>
>>>     # convert pred to fp32
```

(continues on next page)

(continued from previous page)

```
>>>     @force_fp32(apply_to=('pred', ))
>>>     def post_process(self, pred, others):
>>>         pass
```

mmcv.runner.**get_host_info**() → str
Get hostname and username.

Return empty string if exception raised, e.g. `getpass.getuser()` will lead to error in docker container

mmcv.runner.**get_priority**(*priority: Union[int, str,* mmcv.runner.priority.Priority*]*) → int
Get priority value.

Parameters **priority** (int or str or *Priority*) – Priority.

Returns The priority value.

Return type int

mmcv.runner.**load_checkpoint**(*model: torch.nn.modules.module.Module*, *filename: str*, *map_location: Optional[Union[str, Callable]] = None*, *strict: bool = False*, *logger: Optional[logging.Logger] = None*, *revise_keys: list = [('^module\\.', '')]*) → Union[dict, collections.OrderedDict]
Load checkpoint from a file or URI.

Parameters

- **model** (*Module*) – Module to load checkpoint.
- **filename** (*str*) – Accept local filepath, URL, `torchvision://xxx`, `open-mmlab://xxx`. Please refer to `docs/model_zoo.md` for details.
- **map_location** (*str*) – Same as `torch.load()`.
- **strict** (*bool*) – Whether to allow different params for the model and checkpoint.
- **logger** (`logging.Logger` or None) – The logger for error message.
- **revise_keys** (*list*) – A list of customized keywords to modify the state_dict in checkpoint. Each item is a (pattern, replacement) pair of the regular expression operations. Default: strip the prefix 'module.' by [(r'^module.', '')].

Returns The loaded checkpoint.

Return type dict or OrderedDict

mmcv.runner.**load_state_dict**(*module: torch.nn.modules.module.Module*, *state_dict: Union[dict, collections.OrderedDict]*, *strict: bool = False*, *logger: Optional[logging.Logger] = None*) → None
Load state_dict to a module.

This method is modified from `torch.nn.Module.load_state_dict()`. Default value for `strict` is set to `False` and the message for param mismatch will be shown even if strict is False.

Parameters

- **module** (*Module*) – Module that receives the state_dict.
- **state_dict** (*dict or OrderedDict*) – Weights.
- **strict** (*bool*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this module's `state_dict()` function. Default: `False`.
- **logger** (`logging.Logger`, optional) – Logger to log the error message. If not specified, print function will be used.

mmcv.runner.**obj_from_dict**(*info: dict*, *parent: Optional[module] = None*, *default_args: Optional[dict] = None*)

Initialize an object from dict.

The dict must contain the key "type", which indicates the object type, it can be either a string or type, such as "list" or `list`. Remaining fields are treated as the arguments for constructing the object.

Parameters

- **info** (*dict*) – Object types and arguments.
- **parent** (`module`) – Module which may containing expected object classes.
- **default_args** (*dict, optional*) – Default arguments for initializing the object.

Returns Object built from the dict.

Return type any type

mmcv.runner.**save_checkpoint**(*model: torch.nn.modules.module.Module*, *filename: str*, *optimizer: Optional[torch.optim.optimizer.Optimizer] = None*, *meta: Optional[dict] = None*, *file_client_args: Optional[dict] = None*) → None

Save checkpoint to file.

The checkpoint will have 3 fields: `meta`, `state_dict` and `optimizer`. By default `meta` will contain version and time info.

Parameters

- **model** (*Module*) – Module whose params are to be saved.
- **filename** (*str*) – Checkpoint filename.
- **optimizer** (`Optimizer`, optional) – Optimizer to be saved.
- **meta** (*dict, optional*) – Metadata to be saved in checkpoint.
- **file_client_args** (*dict, optional*) – Arguments to instantiate a FileClient. See *mmcv.fileio.FileClient* for details. Default: None. *New in version 1.3.16.*

mmcv.runner.**set_random_seed**(*seed: int*, *deterministic: bool = False*, *use_rank_shift: bool = False*) → None

Set random seed.

Parameters

- **seed** (*int*) – Seed to be used.
- **deterministic** (*bool*) – Whether to set the deterministic option for CUDNN backend, i.e., set *torch.backends.cudnn.deterministic* to True and *torch.backends.cudnn.benchmark* to False. Default: False.
- **rank_shift** (*bool*) – Whether to add rank number to the random seed to have different random seed in different threads. Default: False.

mmcv.runner.**weights_to_cpu**(*state_dict: collections.OrderedDict*) → collections.OrderedDict

Copy a model state_dict to cpu.

Parameters **state_dict** (*OrderedDict*) – Model weights on GPU.

Returns Model weights on GPU.

Return type OrderedDict

mmcv.runner.**wrap_fp16_model**(*model: torch.nn.modules.module.Module*) → None

Wrap the FP32 model to FP16.

If you are using PyTorch >= 1.6, torch.cuda.amp is used as the backend, otherwise, original mmcv implementation will be adopted.

For PyTorch >= 1.6, this function will 1. Set fp16 flag inside the model to True.

Otherwise: 1. Convert FP32 model to FP16. 2. Remain some necessary layers to be FP32, e.g., normalization layers. 3. Set *fp16_enabled* flag inside the model to True.

Parameters `model` (*nn.Module*) – Model in FP32.

CHAPTER

THIRTYFOUR

ENGINE

mmcv.engine.**collect_results_cpu**(*result_part: list*, *size: int*, *tmpdir: Optional[str] = None*) → Optional[list]
Collect results under cpu mode.

On cpu mode, this function will save the results on different gpus to `tmpdir` and collect them by the rank 0 worker.

Parameters

- **result_part** (*list*) – Result list containing result parts to be collected.
- **size** (*int*) – Size of the results, commonly equal to length of the results.
- **tmpdir** (*str | None*) – temporal directory for collected results to store. If set to None, it will create a random temporal directory for it.

Returns The collected results.

Return type list

mmcv.engine.**collect_results_gpu**(*result_part: list*, *size: int*) → Optional[list]
Collect results under gpu mode.

On gpu mode, this function will encode results to gpu tensors and use gpu communication for results collection.

Parameters

- **result_part** (*list*) – Result list containing result parts to be collected.
- **size** (*int*) – Size of the results, commonly equal to length of the results.

Returns The collected results.

Return type list

mmcv.engine.**multi_gpu_test**(*model: torch.nn.modules.module.Module*, *data_loader: torch.utils.data.dataloader.DataLoader*, *tmpdir: Optional[str] = None*, *gpu_collect: bool = False*) → Optional[list]
Test model with multiple gpus.

This method tests model with multiple gpus and collects the results under two different modes: gpu and cpu modes. By setting `gpu_collect=True`, it encodes results to gpu tensors and use gpu communication for results collection. On cpu mode it saves the results on different gpus to `tmpdir` and collects them by the rank 0 worker.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.
- **tmpdir** (*str*) – Path of directory to save the temporary results from different gpus under cpu mode.

- **gpu_collect** (*bool*) – Option to use either gpu or cpu to collect results.

Returns The prediction results.

Return type list

mmcv.engine.**single_gpu_test**(*model: torch.nn.modules.module.Module*, *data_loader: torch.utils.data.dataloader.DataLoader*) → list

Test model with a single gpu.

This method tests model with a single gpu and displays test progress bar.

Parameters

- **model** (*nn.Module*) – Model to be tested.
- **data_loader** (*nn.Dataloader*) – Pytorch data loader.

Returns The prediction results.

Return type list

CHAPTER

THIRTYFIVE

OPS

class mmcv.ops.**BorderAlign**(*pool_size: int*)

Border align pooling layer.

Applies border_align over the input feature based on predicted bboxes. The details were described in the paper BorderDet: Border Feature for Dense Object Detection.

For each border line (e.g. top, left, bottom or right) of each box, border_align does the following:

1. uniformly samples `pool_size` +1 positions on this line, involving the start and end points.
2. the corresponding features on these points are computed by bilinear interpolation.
3. max pooling over all the `pool_size` +1 positions are used for computing pooled feature.

Parameters **pool_size** (*int*) – number of positions sampled over the boxes' borders (e.g. top, bottom, left, right).

forward(*input: torch.Tensor*, *boxes: torch.Tensor*) → torch.Tensor

Parameters

- **input** – Features with shape [N,4C,H,W]. Channels ranged in [0,C), [C,2C), [2C,3C), [3C,4C) represent the top, left, bottom, right features respectively.
- **boxes** – Boxes with shape [N,H*W,4]. Coordinate format (x1,y1,x2,y2).

Returns Pooled features with shape [N,C,H*W,4]. The order is (top,left,bottom,right) for the last dimension.

Return type torch.Tensor

class mmcv.ops.**CARAFE**(*kernel_size: int*, *group_size: int*, *scale_factor: int*)

CARAFE: Content-Aware ReAssembly of FEatures

Please refer to CARAFE: Content-Aware ReAssembly of FEatures for more details.

Parameters

- **kernel_size** (*int*) – reassemble kernel size
- **group_size** (*int*) – reassemble group size
- **scale_factor** (*int*) – upsample ratio

Returns upsampled feature map

forward(*features: torch.Tensor*, *masks: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**CARAFENaive**(*kernel_size: int*, *group_size: int*, *scale_factor: int*)

forward(*features: torch.Tensor*, *masks: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**CARAFEPack**(*channels: int*, *scale_factor: int*, *up_kernel: int = 5*, *up_group: int = 1*, *encoder_kernel: int = 3*, *encoder_dilation: int = 1*, *compressed_channels: int = 64*)

A unified package of CARAFE upsampler that contains: 1) channel compressor 2) content encoder 3) CARAFE op.

Official implementation of ICCV 2019 paper CARAFE: Content-Aware ReAssembly of FEatures.

Parameters

- **channels** (*int*) – input feature channels
- **scale_factor** (*int*) – upsample ratio
- **up_kernel** (*int*) – kernel size of CARAFE op
- **up_group** (*int*) – group size of CARAFE op
- **encoder_kernel** (*int*) – kernel size of content encoder
- **encoder_dilation** (*int*) – dilation of content encoder
- **compressed_channels** (*int*) – output channels of channels compressor

Returns upsampled feature map

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

mmcv.ops.**Conv2d**
alias of `mmcv.ops.deprecated_wrappers.Conv2d_deprecated`

mmcv.ops.**ConvTranspose2d**
alias of `mmcv.ops.deprecated_wrappers.ConvTranspose2d_deprecated`

class mmcv.ops.**CornerPool**(*mode: str*)

Corner Pooling.

Corner Pooling is a new type of pooling layer that helps a convolutional network better localize corners of bounding boxes.

Please refer to CornerNet: Detecting Objects as Paired Keypoints for more details.

Code is modified from https://github.com/princeton-vl/CornerNet-Lite.

Parameters **mode** (*str*) – Pooling orientation for the pooling layer

- 'bottom': Bottom Pooling
- 'left': Left Pooling
- 'right': Right Pooling
- 'top': Top Pooling

Returns Feature map after pooling.

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**Correlation**(*kernel_size: int = 1*, *max_displacement: int = 1*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, *dilation_patch: int = 1*)

Correlation operator

This correlation operator works for optical flow correlation computation.

There are two batched tensors with shape (N, C, H, W), and the correlation output's shape is $(N, max\_displacement \times 2 + 1, max\_displacement * 2 + 1, H_{out}, W_{out})$

where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times padding - dilation \times (kernel\_size - 1) - 1}{stride} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times padding - dilation \times (kernel\_size - 1) - 1}{stride} + 1 \right\rfloor$$

the correlation item (N_i, dy, dx) is formed by taking the sliding window convolution between input1 and shifted input2,

$$Corr(N_i, dx, dy) = \sum_{c=0}^{C-1} input1(N_i, c) \star \mathcal{S}(input2(N_i, c), dy, dx)$$

where $\star$ is the valid 2d sliding window convolution operator, and $\mathcal{S}$ means shifting the input features (auto-complete zero marginal), and dx, dy are shifting distance, $dx, dy \in [-max\_displacement \times dilation\_patch, max\_displacement \times dilation\_patch]$.

Parameters

- **kernel_size** (*int*) – The size of sliding window i.e. local neighborhood representing the center points and involved in correlation computation. Defaults to 1.

- **max_displacement** (*int*) – The radius for computing correlation volume, but the actual working space can be dilated by dilation_patch. Defaults to 1.
- **stride** (*int*) – The stride of the sliding blocks in the input spatial dimensions. Defaults to 1.
- **padding** (*int*) – Zero padding added to all four sides of the input1. Defaults to 0.
- **dilation** (*int*) – The spacing of local neighborhood that will involved in correlation. Defaults to 1.
- **dilation_patch** (*int*) – The spacing between position need to compute correlation. Defaults to 1.

forward(*input1: torch.Tensor*, *input2: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**CrissCrossAttention**(*in_channels: int*)
Criss-Cross Attention Module.

Note: Before v1.3.13, we use a CUDA op. Since v1.3.13, we switch to a pure PyTorch and equivalent implementation. For more details, please refer to https://github.com/open-mmlab/mmcv/pull/1201.

Speed comparison for one forward pass

- Input size: [2,512,97,97]
- Device: 1 NVIDIA GeForce RTX 2080 Ti

	PyTorch version	CUDA version	Relative speed
with torch.no_grad()	0.00554402 s	0.0299619 s	5.4x
no with torch.no_grad()	0.00562803 s	0.0301349 s	5.4x

Parameters **in_channels** (*int*) – Channels of the input feature map.

forward(*x: torch.Tensor*) → torch.Tensor
forward function of Criss-Cross Attention.

Parameters **x** (*torch.Tensor*) – Input feature with the shape of (batch_size, in_channels, height, width).

Returns Output of the layer, with the shape of (batch_size, in_channels, height, width)

Return type torch.Tensor

class mmcv.ops.**DeformConv2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, ...]]*, *stride: Union[int, Tuple[int, ...]] = 1*, *padding: Union[int, Tuple[int, ...]] = 0*, *dilation: Union[int, Tuple[int, ...]] = 1*, *groups: int = 1*, *deform_groups: int = 1*, *bias: bool = False*, *im2col_step: int = 32*)
Deformable 2D convolution.

Applies a deformable 2D convolution over an input signal composed of several input planes. DeformConv2d was described in the paper Deformable Convolutional Networks

Note: The argument `im2col_step` was added in version 1.3.17, which means number of samples processed by the `im2col_cuda_kernel` per call. It enables users to define `batch_size` and `im2col_step` more flexibly and solved issue mmcv#1440.

Parameters

- **in_channels** (*int*) – Number of channels in the input image.
- **out_channels** (*int*) – Number of channels produced by the convolution.
- **kernel_size** (*int, tuple*) – Size of the convolving kernel.
- **stride** (*int, tuple*) – Stride of the convolution. Default: 1.
- **padding** (*int or tuple*) – Zero-padding added to both sides of the input. Default: 0.
- **dilation** (*int or tuple*) – Spacing between kernel elements. Default: 1.
- **groups** (*int*) – Number of blocked connections from input. channels to output channels. Default: 1.
- **deform_groups** (*int*) – Number of deformable group partitions.
- **bias** (*bool*) – If True, adds a learnable bias to the output. Default: False.
- **im2col_step** (*int*) – Number of samples processed by im2col_cuda_kernel per call. It will work when `batch_size > im2col_step`, but `batch_size` must be divisible by `im2col_step`. Default: 32. *New in version 1.3.17.*

forward(*x: torch.Tensor*, *offset: torch.Tensor*) → torch.Tensor

Deformable Convolutional forward function.

Parameters

- **x** (*Tensor*) – Input feature, shape (B, C_in, H_in, W_in)
- **offset** (*Tensor*) – Offset for deformable convolution, shape (B, deform_groups*kernel_size[0]*kernel_size[1]*2, H_out, W_out), H_out, W_out are equal to the output's.

  An offset is like *[y0, x0, y1, x1, y2, x2, …, y8, x8]*. The spatial arrangement is like:

```
(x0, y0) (x1, y1) (x2, y2)
(x3, y3) (x4, y4) (x5, y5)
(x6, y6) (x7, y7) (x8, y8)
```

Returns Output of the layer.

Return type Tensor

class mmcv.ops.**DeformConv2dPack**(**args*, ***kwargs*)

A Deformable Conv Encapsulation that acts as normal Conv layers.

The offset tensor is like *[y0, x0, y1, x1, y2, x2, …, y8, x8]*. The spatial arrangement is like:

```
(x0, y0) (x1, y1) (x2, y2)
(x3, y3) (x4, y4) (x5, y5)
(x6, y6) (x7, y7) (x8, y8)
```

Parameters

- **in_channels** (*int*) – Same as nn.Conv2d.
- **out_channels** (*int*) – Same as nn.Conv2d.
- **kernel_size** (*int or tuple[int]*) – Same as nn.Conv2d.
- **stride** (*int or tuple[int]*) – Same as nn.Conv2d.
- **padding** (*int or tuple[int]*) – Same as nn.Conv2d.
- **dilation** (*int or tuple[int]*) – Same as nn.Conv2d.
- **groups** (*int*) – Same as nn.Conv2d.
- **bias** (*bool or str*) – If specified as *auto*, it will be decided by the norm_cfg. Bias will be set as True if norm_cfg is None, otherwise False.

forward(*x: torch.Tensor*) → torch.Tensor
Deformable Convolutional forward function.

Parameters

- **x** (*Tensor*) – Input feature, shape (B, C_in, H_in, W_in)
- **offset** (*Tensor*) – Offset for deformable convolution, shape (B, deform_groups*kernel_size[0]*kernel_size[1]*2, H_out, W_out), H_out, W_out are equal to the output's.

  An offset is like *[y0, x0, y1, x1, y2, x2, ..., y8, x8]*. The spatial arrangement is like:

```
(x0, y0) (x1, y1) (x2, y2)
(x3, y3) (x4, y4) (x5, y5)
(x6, y6) (x7, y7) (x8, y8)
```

Returns Output of the layer.

Return type Tensor

class mmcv.ops.**DeformRoIPool**(*output_size: Tuple[int, ...]*, *spatial_scale: float = 1.0*, *sampling_ratio: int = 0*, *gamma: float = 0.1*)

forward(*input: torch.Tensor*, *rois: torch.Tensor*, *offset: Optional[torch.Tensor] = None*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**DeformRoIPoolPack**(*output_size: Tuple[int, ...]*, *output_channels: int*, *deform_fc_channels: int = 1024*, *spatial_scale: float = 1.0*, *sampling_ratio: int = 0*, *gamma: float = 0.1*)

forward(*input: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.ops.**DynamicScatter**(*voxel_size: List*, *point_cloud_range: List*, *average_points: bool*)

Scatters points into voxels, used in the voxel encoder with dynamic voxelization.

Note: The CPU and GPU implementation get the same output, but have numerical difference after summation and division (e.g., 5e-7).

Parameters

- **voxel_size** (*list*) – list [x, y, z] size of three dimension.
- **point_cloud_range** (*list*) – The coordinate range of points, [x_min, y_min, z_min, x_max, y_max, z_max].
- **average_points** (*bool*) – whether to use avg pooling to scatter points into voxel.

forward(*points: torch.Tensor*, *coors: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Scatters points/features into voxels.

Parameters

- **points** (*torch.Tensor*) – Points to be reduced into voxels.
- **coors** (*torch.Tensor*) – Corresponding voxel coordinates (specifically multi-dim voxel index) of each points.

Returns A tuple contains two elements. The first one is the voxel features with shape [M, C] which are respectively reduced from input features that share the same voxel coordinates. The second is voxel coordinates with shape [M, ndim].

Return type tuple[torch.Tensor]

forward_single(*points: torch.Tensor*, *coors: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Scatters points into voxels.

Parameters

- **points** (*torch.Tensor*) – Points to be reduced into voxels.
- **coors** (*torch.Tensor*) – Corresponding voxel coordinates (specifically multi-dim voxel index) of each points.

Returns A tuple contains two elements. The first one is the voxel features with shape [M, C] which are respectively reduced from input features that share the same voxel coordinates. The second is voxel coordinates with shape [M, ndim].

Return type tuple[torch.Tensor]

class mmcv.ops.**FusedBiasLeakyReLU**(*num_channels: int*, *negative_slope: float = 0.2*, *scale: float = 1.4142135623730951*)

Fused bias leaky ReLU.

This function is introduced in the StyleGAN2: Analyzing and Improving the Image Quality of StyleGAN

The bias term comes from the convolution operation. In addition, to keep the variance of the feature map or gradients unchanged, they also adopt a scale similarly with Kaiming initialization. However, since the $1+alpha^2$

is too small, we can just ignore it. Therefore, the final scale is just $\sqrt{2}$. Of course, you may change it with your own scale.

TODO: Implement the CPU version.

Parameters

- **num_channels** (*int*) – The channel number of the feature map.
- **negative_slope** (*float, optional*) – Same as nn.LeakyRelu. Defaults to 0.2.
- **scale** (*float, optional*) – A scalar to adjust the variance of the feature map. Defaults to 2**0.5.

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**GroupAll**(*use_xyz: bool = True*)

Group xyz with feature.

Parameters **use_xyz** (*bool*) – Whether to use xyz.

forward(*xyz: torch.Tensor*, *new_xyz: torch.Tensor*, *features: Optional[torch.Tensor] = None*) → torch.Tensor

Parameters

- **xyz** (*Tensor*) – (B, N, 3) xyz coordinates of the features.
- **new_xyz** (*Tensor*) – new xyz coordinates of the features.
- **features** (*Tensor*) – (B, C, N) features to group.

Returns (B, C + 3, 1, N) Grouped feature.

Return type Tensor

mmcv.ops.**Linear**

alias of mmcv.ops.deprecated_wrappers.Linear_deprecated

class mmcv.ops.**MaskedConv2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int, ...]]*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, *groups: int = 1*, *bias: bool = True*)

A MaskedConv2d which inherits the official Conv2d.

The masked forward doesn't implement the backward function and only supports the stride parameter to be 1 currently.

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while

the latter silently ignores them.

---

mmcv.ops.**MaxPool2d**
alias of mmcv.ops.deprecated_wrappers.MaxPool2d_deprecated

class mmcv.ops.**ModulatedDeformConv2d**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int]]*, *stride: int = 1*, *padding: int = 0*, *dilation: int = 1*, *groups: int = 1*, *deform_groups: int = 1*, *bias: Union[bool, str] = True*)

forward(*x: torch.Tensor*, *offset: torch.Tensor*, *mask: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**ModulatedDeformConv2dPack**(**args*, ***kwargs*)
A ModulatedDeformable Conv Encapsulation that acts as normal Conv layers.

Parameters

- **in_channels** (*int*) – Same as nn.Conv2d.
- **out_channels** (*int*) – Same as nn.Conv2d.
- **kernel_size** (*int or tuple[int]*) – Same as nn.Conv2d.
- **stride** (*int*) – Same as nn.Conv2d, while tuple is not supported.
- **padding** (*int*) – Same as nn.Conv2d, while tuple is not supported.
- **dilation** (*int*) – Same as nn.Conv2d, while tuple is not supported.
- **groups** (*int*) – Same as nn.Conv2d.
- **bias** (*bool or str*) – If specified as *auto*, it will be decided by the norm_cfg. Bias will be set as True if norm_cfg is None, otherwise False.

forward(*x: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**ModulatedDeformRoIPoolPack**(*output_size: Tuple[int, ...]*, *output_channels: int*, *deform_fc_channels: int = 1024*, *spatial_scale: float = 1.0*, *sampling_ratio: int = 0*, *gamma: float = 0.1*)

forward(*input: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**MultiScaleDeformableAttention**(*embed_dims: int = 256*, *num_heads: int = 8*, *num_levels: int = 4*, *num_points: int = 4*, *im2col_step: int = 64*, *dropout: float = 0.1*, *batch_first: bool = False*, *norm_cfg: Optional[dict] = None*, *init_cfg: Optional[*mmcv.utils.config.ConfigDict*] = None*)

An attention module used in Deformable-Detr.

Deformable DETR: Deformable Transformers for End-to-End Object Detection..

Parameters

- **embed_dims** (*int*) – The embedding dimension of Attention. Default: 256.
- **num_heads** (*int*) – Parallel attention heads. Default: 64.
- **num_levels** (*int*) – The number of feature map used in Attention. Default: 4.
- **num_points** (*int*) – The number of sampling points for each query in each head. Default: 4.
- **im2col_step** (*int*) – The step used in image_to_column. Default: 64.
- **dropout** (*float*) – A Dropout layer on *inp_identity*. Default: 0.1.
- **batch_first** (*bool*) – Key, Query and Value are shape of (batch, n, embed_dim) or (n, batch, embed_dim). Default to False.
- **norm_cfg** (*dict*) – Config dict for normalization layer. Default: None.
- **(obj** (*init_cfg*) – *mmcv.ConfigDict*): The Config for initialization. Default: None.

forward(*query: torch.Tensor*, *key: Optional[torch.Tensor] = None*, *value: Optional[torch.Tensor] = None*, *identity: Optional[torch.Tensor] = None*, *query_pos: Optional[torch.Tensor] = None*, *key_padding_mask: Optional[torch.Tensor] = None*, *reference_points: Optional[torch.Tensor] = None*, *spatial_shapes: Optional[torch.Tensor] = None*, *level_start_index: Optional[torch.Tensor] = None*, ***kwargs*) → torch.Tensor

Forward Function of MultiScaleDeformAttention.

Parameters

- **query** (*torch.Tensor*) – Query of Transformer with shape (num_query, bs, embed_dims).
- **key** (*torch.Tensor*) – The key tensor with shape *(num_key, bs, embed_dims)*.
- **value** (*torch.Tensor*) – The value tensor with shape *(num_key, bs, embed_dims)*.
- **identity** (*torch.Tensor*) – The tensor used for addition, with the same shape as *query*. Default None. If None, *query* will be used.
- **query_pos** (*torch.Tensor*) – The positional encoding for *query*. Default: None.
- **key_padding_mask** (*torch.Tensor*) – ByteTensor for *query*, with shape [bs, num_key].
- **reference_points** (*torch.Tensor*) – The normalized reference points with shape (bs, num_query, num_levels, 2), all elements is range in [0, 1], top-left (0,0), bottom-right (1,

1), including padding area. or (N, Length_{query}, num_levels, 4), add additional two dimensions is (w, h) to form reference boxes.

- **spatial_shapes** (*torch.Tensor*) – Spatial shape of features in different levels. With shape (num_levels, 2), last dimension represents (h, w).
- **level_start_index** (*torch.Tensor*) – The start index of each level. A tensor has shape `(num_levels, )` and can be represented as [0, h_0*w_0, h_0*w_0+h_1*w_1, ...].

Returns forwarded results with shape [num_query, bs, embed_dims].

Return type torch.Tensor

init_weights() → None
Default initialization for Parameters of Module.

class mmcv.ops.**PSAMask**(*psa_type: str*, *mask_size: Optional[tuple] = None*)

forward(*input: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**PointsSampler**(*num_point: List[int]*, *fps_mod_list: List[str] = ['D-FPS']*, *fps_sample_range_list: List[int] = [- 1]*)
Points sampling.

Parameters

- **num_point** (*list[int]*) – Number of sample points.
- **fps_mod_list** (*list[str], optional*) – Type of FPS method, valid mod ['F-FPS', 'D-FPS', 'FS'], Default: ['D-FPS']. F-FPS: using feature distances for FPS. D-FPS: using Euclidean distances of points for FPS. FS: using F-FPS and D-FPS simultaneously.
- **fps_sample_range_list** (*list[int], optional*) – Range of points to apply FPS. Default: [-1].

forward(*points_xyz: torch.Tensor*, *features: torch.Tensor*) → torch.Tensor

Parameters

- **points_xyz** (*torch.Tensor*) – (B, N, 3) xyz coordinates of the points.
- **features** (*torch.Tensor*) – (B, C, N) features of the points.

Returns (B, npoint, sample_num) Indices of sampled points.

Return type torch.Tensor

class mmcv.ops.**PrRoIPool**(*output_size: Union[int, tuple]*, *spatial_scale: float = 1.0*)
The operation of precision RoI pooling. The implementation of PrRoIPool is modified from https://github.com/vacancy/PreciseRoIPooling/

Precise RoI Pooling (PrRoIPool) is an integration-based (bilinear interpolation) average pooling method for RoI Pooling. It avoids any quantization and has a continuous gradient on bounding box coordinates. It is:

1. different from the original RoI Pooling proposed in Fast R-CNN. PrRoI Pooling uses average pooling instead of max pooling for each bin and has a continuous gradient on bounding box coordinates. That is, one can take the derivatives of some loss function w.r.t the coordinates of each RoI and optimize the RoI coordinates. 2. different from the RoI Align proposed in Mask R-CNN. PrRoI Pooling uses a full integration-based average pooling instead of sampling a constant number of points. This makes the gradient w.r.t. the coordinates continuous.

Parameters

- **output_size** (*Union[int, tuple]*) – h, w.
- **spatial_scale** (*float, optional*) – scale the input boxes by this number. Defaults to 1.0.

forward(*features: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor

Forward function.

Parameters

- **features** (*torch.Tensor*) – The feature map.
- **rois** (*torch.Tensor*) – The RoI bboxes in [tl_x, tl_y, br_x, br_y] format.

Returns The pooled results.

Return type torch.Tensor

class mmcv.ops.**QueryAndGroup**(*max_radius: float*, *sample_num: int*, *min_radius: float = 0.0*, *use_xyz: bool = True*, *return_grouped_xyz: bool = False*, *normalize_xyz: bool = False*, *uniform_sample: bool = False*, *return_unique_cnt: bool = False*, *return_grouped_idx: bool = False*)

Groups points with a ball query of radius.

Parameters

- **max_radius** (*float*) – The maximum radius of the balls. If None is given, we will use kNN sampling instead of ball query.
- **sample_num** (*int*) – Maximum number of features to gather in the ball.
- **min_radius** (*float, optional*) – The minimum radius of the balls. Default: 0.
- **use_xyz** (*bool, optional*) – Whether to use xyz. Default: True.
- **return_grouped_xyz** (*bool, optional*) – Whether to return grouped xyz. Default: False.
- **normalize_xyz** (*bool, optional*) – Whether to normalize xyz. Default: False.
- **uniform_sample** (*bool, optional*) – Whether to sample uniformly. Default: False
- **return_unique_cnt** (*bool, optional*) – Whether to return the count of unique samples. Default: False.
- **return_grouped_idx** (*bool, optional*) – Whether to return grouped idx. Default: False.

forward(*points_xyz: torch.Tensor*, *center_xyz: torch.Tensor*, *features: Optional[torch.Tensor] = None*) → Union[torch.Tensor, Tuple]

Parameters

- **points_xyz** (*torch.Tensor*) – (B, N, 3) xyz coordinates of the points.
- **center_xyz** (*torch.Tensor*) – (B, npoint, 3) coordinates of the centriods.

- **features** (*torch.Tensor*) – (B, C, N) The features of grouped points.

Returns (B, 3 + C, npoint, sample_num) Grouped concatenated coordinates and features of points.

Return type Tuple | torch.Tensor

class mmcv.ops.**RiRoIAlignRotated**(*out_size: tuple*, *spatial_scale: float*, *num_samples: int = 0*, *num_orientations: int = 8*, *clockwise: bool = False*)

Rotation-invariant RoI align pooling layer for rotated proposals.

It accepts a feature map of shape (N, C, H, W) and rois with shape (n, 6) with each roi decoded as (batch_index, center_x, center_y, w, h, angle). The angle is in radian.

The details are described in the paper ReDet: A Rotation-equivariant Detector for Aerial Object Detection.

Parameters

- **out_size** (*tuple*) – fixed dimensional RoI output with shape (h, w).
- **spatial_scale** (*float*) – scale the input boxes by this number
- **num_samples** (*int*) – number of inputs samples to take for each output sample. 0 to take samples densely for current models.
- **num_orientations** (*int*) – number of oriented channels.
- **clockwise** (*bool*) – If True, the angle in each proposal follows a clockwise fashion in image space, otherwise, the angle is counterclockwise. Default: False.

forward(*features: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**RoIAlign**(*output_size: tuple*, *spatial_scale: float = 1.0*, *sampling_ratio: int = 0*, *pool_mode: str = 'avg'*, *aligned: bool = True*, *use_torchvision: bool = False*)

RoI align pooling layer.

Parameters

- **output_size** (*tuple*) – h, w
- **spatial_scale** (*float*) – scale the input boxes by this number
- **sampling_ratio** (*int*) – number of inputs samples to take for each output sample. 0 to take samples densely for current models.
- **pool_mode** (*str, 'avg' or 'max'*) – pooling mode in each bin.
- **aligned** (*bool*) – if False, use the legacy implementation in MMDetection. If True, align the results more perfectly.
- **use_torchvision** (*bool*) – whether to use roi_align from torchvision.

Note: The implementation of RoIAlign when aligned=True is modified from https://github.com/facebookresearch/detectron2/

The meaning of aligned=True:

Given a continuous coordinate c, its two neighboring pixel indices (in our pixel model) are computed by floor(c - 0.5) and ceil(c - 0.5). For example, c=1.3 has pixel neighbors with discrete indices [0] and [1] (which are sampled from the underlying signal at continuous coordinates 0.5 and 1.5). But the original roi_align (aligned=False) does not subtract the 0.5 when computing neighboring pixel indices and therefore it uses pixels with a slightly incorrect alignment (relative to our pixel model) when performing bilinear interpolation.

With *aligned=True*, we first appropriately scale the ROI and then shift it by -0.5 prior to calling roi_align. This produces the correct neighbors;

The difference does not make a difference to the model's performance if ROIAlign is used together with conv layers.

---

forward(*input: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor

Parameters

- **input** – NCHW images
- **rois** – Bx5 boxes. First column is the index into N. The other 4 columns are xyxy.

class mmcv.ops.**RoIAlignRotated**(*output_size: Union[int, tuple]*, *spatial_scale: float*, *sampling_ratio: int = 0*, *aligned: bool = True*, *clockwise: bool = False*)

RoI align pooling layer for rotated proposals.

It accepts a feature map of shape (N, C, H, W) and rois with shape (n, 6) with each roi decoded as (batch_index, center_x, center_y, w, h, angle). The angle is in radian.

Parameters

- **output_size** (*tuple*) – h, w
- **spatial_scale** (*float*) – scale the input boxes by this number
- **sampling_ratio** (*int*) – number of inputs samples to take for each output sample. 0 to take samples densely for current models.
- **aligned** (*bool*) – if False, use the legacy implementation in MMDetection. If True, align the results more perfectly. Default: True.
- **clockwise** (*bool*) – If True, the angle in each proposal follows a clockwise fashion in image space, otherwise, the angle is counterclockwise. Default: False.

Note: The implementation of RoIAlign when aligned=True is modified from https://github.com/facebookresearch/detectron2/

The meaning of aligned=True:

Given a continuous coordinate c, its two neighboring pixel indices (in our pixel model) are computed by floor(c - 0.5) and ceil(c - 0.5). For example, c=1.3 has pixel neighbors with discrete indices [0] and [1] (which are sampled from the underlying signal at continuous coordinates 0.5 and 1.5). But the original roi_align (aligned=False) does not subtract the 0.5 when computing neighboring pixel indices and therefore it uses pixels with a slightly incorrect alignment (relative to our pixel model) when performing bilinear interpolation.

With *aligned=True*, we first appropriately scale the ROI and then shift it by -0.5 prior to calling roi_align. This produces the correct neighbors;

The difference does not make a difference to the model's performance if ROIAlign is used together with conv layers.

---

forward(*input: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**RoIAwarePool3d**(*out_size: Union[int, tuple]*, *max_pts_per_voxel: int = 128*, *mode: str = 'max'*)
Encode the geometry-specific features of each 3D proposal.

Please refer to PartA2 for more details.

Parameters

- **out_size** (*int or tuple*) – The size of output features. n or [n1, n2, n3].
- **max_pts_per_voxel** (*int, optional*) – The maximum number of points per voxel. Default: 128.
- **mode** (*str, optional*) – Pooling method of RoIAware, 'max' or 'avg'. Default: 'max'.

forward(*rois: torch.Tensor*, *pts: torch.Tensor*, *pts_feature: torch.Tensor*) → torch.Tensor

Parameters

- **rois** (*torch.Tensor*) – [N, 7], in LiDAR coordinate, (x, y, z) is the bottom center of rois.
- **pts** (*torch.Tensor*) – [npoints, 3], coordinates of input points.
- **pts_feature** (*torch.Tensor*) – [npoints, C], features of input points.

Returns Pooled features whose shape is [N, out_x, out_y, out_z, C].

Return type torch.Tensor

class mmcv.ops.**RoIPointPool3d**(*num_sampled_points: int = 512*)
Encode the geometry-specific features of each 3D proposal.

Please refer to Paper of PartA2 for more details.

Parameters **num_sampled_points** (*int, optional*) – Number of samples in each roi. Default: 512.

forward(*points: torch.Tensor*, *point_features: torch.Tensor*, *boxes3d: torch.Tensor*) → Tuple[torch.Tensor]

Parameters

- **points** (*torch.Tensor*) – Input points whose shape is (B, N, C).
- **point_features** (*torch.Tensor*) – Features of input points whose shape is (B, N, C).
- **boxes3d** (*B, M, 7), Input bounding boxes whose shape is (B, M, 7*) –

Returns A tuple contains two elements. The first one is the pooled features whose shape is (B, M, 512, 3 + C). The second is an empty flag whose shape is (B, M).

Return type tuple[torch.Tensor]

class mmcv.ops.**RoIPool**(*output_size: Union[int, tuple]*, *spatial_scale: float = 1.0*)

forward(*input: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**SAConv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *use_deform=False*)
SAC (Switchable Atrous Convolution)

This is an implementation of DetectoRS: Detecting Objects with Recursive Feature Pyramid and Switchable Atrous Convolution.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`
- **use_deform** – If `True`, replace convolution with deformable convolution. Default: `False`.

forward(*x*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**SigmoidFocalLoss**(*gamma: float*, *alpha: float*, *weight: Optional[torch.Tensor] = None*, *reduction: str = 'mean'*)

forward(*input: torch.Tensor*, *target: Union[torch.LongTensor, torch.cuda.LongTensor]*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.ops.**SimpleRoIAlign**(*output_size: Tuple[int]*, *spatial_scale: float*, *aligned: bool = True*)

forward(*features: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.ops.**SoftmaxFocalLoss**(*gamma: float*, *alpha: float*, *weight: Optional[torch.Tensor] = None*, *reduction: str = 'mean'*)

forward(*input: torch.Tensor*, *target: Union[torch.LongTensor, torch.cuda.LongTensor]*) → torch.Tensor
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class mmcv.ops.**SparseConv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *indice_key=None*)

class mmcv.ops.**SparseConv3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *indice_key=None*)

class mmcv.ops.**SparseConvTranspose2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *indice_key=None*)

class mmcv.ops.**SparseConvTranspose3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *indice_key=None*)

class mmcv.ops.**SparseInverseConv2d**(*in_channels*, *out_channels*, *kernel_size*, *indice_key=None*, *bias=True*)

class mmcv.ops.**SparseInverseConv3d**(*in_channels*, *out_channels*, *kernel_size*, *indice_key=None*, *bias=True*)

class mmcv.ops.**SparseMaxPool2d**(*kernel_size*, *stride=1*, *padding=0*, *dilation=1*)

class mmcv.ops.**SparseMaxPool3d**(*kernel_size*, *stride=1*, *padding=0*, *dilation=1*)

class mmcv.ops.**SparseModule**
place holder, All module subclass from this will take sptensor in SparseSequential.

class mmcv.ops.**SparseSequential**(**args*, ***kwargs*)
A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of modules can also be passed in.

To make it easier to understand, given is a small example:

```
.. rubric:: Example
```

```
>>> # using Sequential:
>>> from mmcv.ops import SparseSequential
>>> model = SparseSequential(
            SparseConv2d(1,20,5),
            nn.ReLU(),
            SparseConv2d(20,64,5),
            nn.ReLU()
            )
```

```
>>> # using Sequential with OrderedDict
>>> model = SparseSequential(OrderedDict([
              ('conv1', SparseConv2d(1,20,5)),
              ('relu1', nn.ReLU()),
              ('conv2', SparseConv2d(20,64,5)),
              ('relu2', nn.ReLU())
            ]))
```

```
>>> # using Sequential with kwargs(python 3.6+)
>>> model = SparseSequential(
              conv1=SparseConv2d(1,20,5),
              relu1=nn.ReLU(),
              conv2=SparseConv2d(20,64,5),
              relu2=nn.ReLU()
            )
```

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**SubMConv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *indice_key=None*)

class mmcv.ops.**SubMConv3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *indice_key=None*)

class mmcv.ops.**SyncBatchNorm**(*num_features: int*, *eps: float = 1e-05*, *momentum: float = 0.1*, *affine: bool = True*, *track_running_stats: bool = True*, *group: Optional[int] = None*, *stats_mode: str = 'default'*)

Synchronized Batch Normalization.

Parameters

- **num_features** (*int*) – number of features/chennels in input tensor
- **eps** (*float, optional*) – a value added to the denominator for numerical stability. Defaults to 1e-5.

- **momentum** (*float, optional*) – the value used for the running_mean and running_var computation. Defaults to 0.1.
- **affine** (*bool, optional*) – whether to use learnable affine parameters. Defaults to True.
- **track_running_stats** (*bool, optional*) – whether to track the running mean and variance during training. When set to False, this module does not track such statistics, and initializes statistics buffers `running_mean` and `running_var` as `None`. When these buffers are `None`, this module always uses batch statistics in both training and eval modes. Defaults to True.
- **group** (*int, optional*) – synchronization of stats happen within each process group individually. By default it is synchronization across the whole world. Defaults to None.
- **stats_mode** (*str, optional*) – The statistical mode. Available options includes `'default'` and `'N'`. Defaults to 'default'. When `stats_mode=='default'`, it computes the overall statistics using those from each worker with equal weight, i.e., the statistics are synchronized and simply divied by `group`. This mode will produce inaccurate statistics when empty tensors occur. When `stats_mode=='N'`, it compute the overall statistics using the total number of batches in each worker ignoring the number of group, i.e., the statistics are synchronized and then divied by the total batch `N`. This mode is beneficial when empty tensors occur during training, as it average the total mean by the real number of batch.

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmcv.ops.**TINShift**

Temporal Interlace Shift.

Temporal Interlace shift is a differentiable temporal-wise frame shifting which is proposed in "Temporal Interlacing Network"

Please refer to Temporal Interlacing Network for more details.

Code is modified from https://github.com/mit-han-lab/temporal-shift-module

forward(*input*, *shift*)

Perform temporal interlace shift.

Parameters

- **input** (*torch.Tensor*) – Feature map with shape [N, num_segments, C, H * W].
- **shift** (*torch.Tensor*) – Shift tensor with shape [N, num_segments].

Returns Feature map after temporal interlace shift.

class mmcv.ops.**Voxelization**(*voxel_size: List*, *point_cloud_range: List*, *max_num_points: int*, *max_voxels: Union[tuple, int] = 20000*, *deterministic: bool = True*)

Convert kitti points(N, >=3) to voxels.

Please refer to Point-Voxel CNN for Efficient 3D Deep Learning for more details.

Parameters

- **voxel_size** (*tuple or float*) – The size of voxel with the shape of [3].

- **point_cloud_range** (*tuple or float*) – The coordinate range of voxel with the shape of [6].
- **max_num_points** (*int*) – maximum points contained in a voxel. if max_points=-1, it means using dynamic_voxelize.
- **max_voxels** (*int, optional*) – maximum voxels this function create. for second, 20000 is a good choice. Users should shuffle points before call this function because max_voxels may drop points. Default: 20000.

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

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

mmcv.ops.**batched_nms**(*boxes: torch.Tensor*, *scores: torch.Tensor*, *idxs: torch.Tensor*, *nms_cfg: Optional[Dict]*, *class_agnostic: bool = False*) → Tuple[torch.Tensor, torch.Tensor]

Performs non-maximum suppression in a batched fashion.

Modified from torchvision/ops/boxes.py#L39. In order to perform NMS independently per class, we add an offset to all the boxes. The offset is dependent only on the class idx, and is large enough so that boxes from different classes do not overlap.

Note: In v1.4.1 and later, `batched_nms` supports skipping the NMS and returns sorted raw results when *nms_cfg* is None.

Parameters

- **boxes** (*torch.Tensor*) – boxes in shape (N, 4) or (N, 5).
- **scores** (*torch.Tensor*) – scores in shape (N,).
- **idxs** (*torch.Tensor*) – each index value correspond to a bbox cluster, and NMS will not be applied between elements of different idxs, shape (N,).
- **nms_cfg** (*dict | optional*) – Supports skipping the nms when *nms_cfg* is None, otherwise it should specify nms type and other parameters like *iou_thr*. Possible keys includes the following.
  - iou_threshold (float): IoU threshold used for NMS.
  - split_thr (float): threshold number of boxes. In some cases the number of boxes is large (e.g., 200k). To avoid OOM during training, the users could set *split_thr* to a small value. If the number of boxes is greater than the threshold, it will perform NMS on each group of boxes separately and sequentially. Defaults to 10000.
- **class_agnostic** (*bool*) – if true, nms is class agnostic, i.e. IoU thresholding happens over all boxes, regardless of the predicted class. Defaults to False.

Returns

kept dets and indice.

- boxes (Tensor): Bboxes with score after nms, has shape (num_bboxes, 5). last dimension 5 arrange as (x1, y1, x2, y2, score)

- keep (Tensor): The indices of remaining boxes in input boxes.

Return type tuple

mmcv.ops.**bbox_overlaps**(*bboxes1: torch.Tensor*, *bboxes2: torch.Tensor*, *mode: str = 'iou'*, *aligned: bool = False*, *offset: int = 0*) → torch.Tensor

Calculate overlap between two set of bboxes.

If `aligned` is `False`, then calculate the ious between each bbox of bboxes1 and bboxes2, otherwise the ious between each aligned pair of bboxes1 and bboxes2.

Parameters

- **bboxes1** (*torch.Tensor*) – shape (m, 4) in <x1, y1, x2, y2> format or empty.
- **bboxes2** (*torch.Tensor*) – shape (n, 4) in <x1, y1, x2, y2> format or empty. If aligned is `True`, then m and n must be equal.
- **mode** (*str*) – "iou" (intersection over union) or iof (intersection over foreground).

Returns Return the ious betweens boxes. If `aligned` is `False`, the shape of ious is (m, n) else (m, 1).

Return type torch.Tensor

Example

```
>>> bboxes1 = torch.FloatTensor([
>>>     [0, 0, 10, 10],
>>>     [10, 10, 20, 20],
>>>     [32, 32, 38, 42],
>>> ])
>>> bboxes2 = torch.FloatTensor([
>>>     [0, 0, 10, 20],
>>>     [0, 10, 10, 19],
>>>     [10, 10, 20, 20],
>>> ])
>>> bbox_overlaps(bboxes1, bboxes2)
tensor([[0.5000, 0.0000, 0.0000],
        [0.0000, 0.0000, 1.0000],
        [0.0000, 0.0000, 0.0000]])
```

Example

```
>>> empty = torch.FloatTensor([])
>>> nonempty = torch.FloatTensor([
>>>     [0, 0, 10, 9],
>>> ])
>>> assert tuple(bbox_overlaps(empty, nonempty).shape) == (0, 1)
>>> assert tuple(bbox_overlaps(nonempty, empty).shape) == (1, 0)
>>> assert tuple(bbox_overlaps(empty, empty).shape) == (0, 0)
```

mmcv.ops.**box_iou_rotated**(*bboxes1: torch.Tensor*, *bboxes2: torch.Tensor*, *mode: str = 'iou'*, *aligned: bool = False*, *clockwise: bool = True*) → torch.Tensor

Return intersection-over-union (Jaccard index) of boxes.

Both sets of boxes are expected to be in (x_center, y_center, width, height, angle) format.

If `aligned` is `False`, then calculate the ious between each bbox of bboxes1 and bboxes2, otherwise the ious between each aligned pair of bboxes1 and bboxes2.

Note: The operator assumes:

1) The positive direction along x axis is left -> right.
2) The positive direction along y axis is top -> down.
3) The w border is in parallel with x axis when angle = 0.

However, there are 2 opposite definitions of the positive angular direction, clockwise (CW) and counter-clockwise (CCW). MMCV supports both definitions and uses CW by default.

Please set `clockwise=False` if you are using the CCW definition.

The coordinate system when `clockwise` is `True` (default)

```
0------------------> x (0 rad)
|  A-------------B
|  |             |
|  |     box     h
|  |   angle=0   |
|  D------w------C
v
y (pi/2 rad)
```

In such coordination system the rotation matrix is

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

The coordinates of the corner point A can be calculated as:

$$\begin{aligned} P_A = \begin{pmatrix} x_A \\ y_A \end{pmatrix} &= \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix} + \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} -0.5w \\ -0.5h \end{pmatrix} \\ &= \begin{pmatrix} x_{center} - 0.5w\cos\alpha + 0.5h\sin\alpha \\ y_{center} - 0.5w\sin\alpha - 0.5h\cos\alpha \end{pmatrix} \end{aligned}$$

The coordinate system when `clockwise` is `False`

```
0------------------> x (0 rad)
|  A-------------B
|  |             |
|  |     box     h
|  |   angle=0   |
|  D------w------C
v
y (-pi/2 rad)
```

In such coordination system the rotation matrix is

$$\begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

The coordinates of the corner point A can be calculated as:

$$P_A = \begin{pmatrix} x_A \\ y_A \end{pmatrix} = \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix} + \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} -0.5w \\ -0.5h \end{pmatrix} = \begin{pmatrix} x_{center} - 0.5w\cos\alpha - 0.5h\sin\alpha \\ y_{center} + 0.5w\sin\alpha - 0.5h\cos\alpha \end{pmatrix}$$

Parameters

- **boxes1** (*torch.Tensor*) – rotated bboxes 1. It has shape (N, 5), indicating (x, y, w, h, theta) for each row. Note that theta is in radian.
- **boxes2** (*torch.Tensor*) – rotated bboxes 2. It has shape (M, 5), indicating (x, y, w, h, theta) for each row. Note that theta is in radian.
- **mode** (*str*) – "iou" (intersection over union) or iof (intersection over foreground).
- **clockwise** (*bool*) – flag indicating whether the positive angular orientation is clockwise. default True. *New in version 1.4.3.*

Returns Return the ious betweens boxes. If `aligned` is `False`, the shape of ious is (N, M) else (N,).

Return type torch.Tensor

mmcv.ops.**boxes_iou3d**(*boxes_a: torch.Tensor*, *boxes_b: torch.Tensor*) → torch.Tensor
Calculate boxes 3D IoU.

Parameters

- **boxes_a** (*torch.Tensor*) – Input boxes a with shape (M, 7).
- **boxes_b** (*torch.Tensor*) – Input boxes b with shape (N, 7).

Returns 3D IoU result with shape (M, N).

Return type torch.Tensor

mmcv.ops.**boxes_iou_bev**(*boxes_a: torch.Tensor*, *boxes_b: torch.Tensor*) → torch.Tensor
Calculate boxes IoU in the Bird's Eye View.

Parameters

- **boxes_a** (*torch.Tensor*) – Input boxes a with shape (M, 5) ([x1, y1, x2, y2, ry]).
- **boxes_b** (*torch.Tensor*) – Input boxes b with shape (N, 5) ([x1, y1, x2, y2, ry]).

Returns IoU result with shape (M, N).

Return type torch.Tensor

mmcv.ops.**boxes_overlap_bev**(*boxes_a: torch.Tensor*, *boxes_b: torch.Tensor*) → torch.Tensor
Calculate boxes BEV overlap.

Parameters

- **boxes_a** (*torch.Tensor*) – Input boxes a with shape (M, 7).
- **boxes_b** (*torch.Tensor*) – Input boxes b with shape (N, 7).

Returns BEV overlap result with shape (M, N).

Return type torch.Tensor

mmcv.ops.**contour_expand**(*kernel_mask: Union[numpy.array, torch.Tensor]*, *internal_kernel_label: Union[numpy.array, torch.Tensor]*, *min_kernel_area: int*, *kernel_num: int*) → list

Expand kernel contours so that foreground pixels are assigned into instances.

Parameters

- **kernel_mask** (*np.array or torch.Tensor*) – The instance kernel mask with size hxw.
- **internal_kernel_label** (*np.array or torch.Tensor*) – The instance internal kernel label with size hxw.
- **min_kernel_area** (*int*) – The minimum kernel area.
- **kernel_num** (*int*) – The instance kernel number.

Returns The instance index map with size hxw.

Return type list

mmcv.ops.**convex_giou**(*pointsets: torch.Tensor*, *polygons: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Return generalized intersection-over-union (Jaccard index) between point sets and polygons.

Parameters

- **pointsets** (*torch.Tensor*) – It has shape (N, 18), indicating (x1, y1, x2, y2, ..., x9, y9) for each row.
- **polygons** (*torch.Tensor*) – It has shape (N, 8), indicating (x1, y1, x2, y2, x3, y3, x4, y4) for each row.

Returns The first element is the gious between point sets and polygons with the shape (N,). The second element is the gradient of point sets with the shape (N, 18).

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

mmcv.ops.**convex_iou**(*pointsets: torch.Tensor*, *polygons: torch.Tensor*) → torch.Tensor

Return intersection-over-union (Jaccard index) between point sets and polygons.

Parameters

- **pointsets** (*torch.Tensor*) – It has shape (N, 18), indicating (x1, y1, x2, y2, ..., x9, y9) for each row.
- **polygons** (*torch.Tensor*) – It has shape (K, 8), indicating (x1, y1, x2, y2, x3, y3, x4, y4) for each row.

Returns Return the ious between point sets and polygons with the shape (N, K).

Return type torch.Tensor

mmcv.ops.**diff_iou_rotated_2d**(*box1: torch.Tensor*, *box2: torch.Tensor*) → torch.Tensor

Calculate differentiable iou of rotated 2d boxes.

Parameters

- **box1** (*Tensor*) – (B, N, 5) First box.
- **box2** (*Tensor*) – (B, N, 5) Second box.

Returns (B, N) IoU.

Return type Tensor

mmcv.ops.**diff_iou_rotated_3d**(*box3d1: torch.Tensor*, *box3d2: torch.Tensor*) → torch.Tensor

Calculate differentiable iou of rotated 3d boxes.

Parameters

- **box3d1** (*Tensor*) – (B, N, 3+3+1) First box (x,y,z,w,h,l,alpha).
- **box3d2** (*Tensor*) – (B, N, 3+3+1) Second box (x,y,z,w,h,l,alpha).

Returns (B, N) IoU.

Return type Tensor

mmcv.ops.**fused_bias_leakyrelu**(*input: torch.Tensor*, *bias: torch.nn.parameter.Parameter*, *negative_slope: float = 0.2*, *scale: float = 1.4142135623730951*) → torch.Tensor

Fused bias leaky ReLU function.

This function is introduced in the StyleGAN2: Analyzing and Improving the Image Quality of StyleGAN

The bias term comes from the convolution operation. In addition, to keep the variance of the feature map or gradients unchanged, they also adopt a scale similarly with Kaiming initialization. However, since the $1+alpha^2$ is too small, we can just ignore it. Therefore, the final scale is just $\sqrt{2}$. Of course, you may change it with your own scale.

Parameters

- **input** (*torch.Tensor*) – Input feature map.
- **bias** (*nn.Parameter*) – The bias from convolution operation.
- **negative_slope** (*float, optional*) – Same as nn.LeakyRelu. Defaults to 0.2.
- **scale** (*float, optional*) – A scalar to adjust the variance of the feature map. Defaults to 2**0.5.

Returns Feature map after non-linear activation.

Return type torch.Tensor

mmcv.ops.**min_area_polygons**(*pointsets: torch.Tensor*) → torch.Tensor

Find the smallest polygons that surrounds all points in the point sets.

Parameters **pointsets** (*Tensor*) – point sets with shape (N, 18).

Returns Return the smallest polygons with shape (N, 8).

Return type torch.Tensor

mmcv.ops.**nms**(*boxes: Union[torch.Tensor, numpy.ndarray]*, *scores: Union[torch.Tensor, numpy.ndarray]*, *iou_threshold: float*, *offset: int = 0*, *score_threshold: float = 0*, *max_num: int = - 1*) → Tuple[Union[torch.Tensor, numpy.ndarray], Union[torch.Tensor, numpy.ndarray]]

Dispatch to either CPU or GPU NMS implementations.

The input can be either torch tensor or numpy array. GPU NMS will be used if the input is gpu tensor, otherwise CPU NMS will be used. The returned type will always be the same as inputs.

Parameters

- **boxes** (*torch.Tensor or np.ndarray*) – boxes in shape (N, 4).
- **scores** (*torch.Tensor or np.ndarray*) – scores in shape (N,).
- **iou_threshold** (*float*) – IoU threshold for NMS.
- **offset** (*int, 0 or 1*) – boxes' width or height is (x2 - x1 + offset).
- **score_threshold** (*float*) – score threshold for NMS.
- **max_num** (*int*) – maximum number of boxes after NMS.

Returns kept dets (boxes and scores) and indice, which always have the same data type as the input.

Return type tuple

Example

```
>>> boxes = np.array([[49.1, 32.4, 51.0, 35.9],
>>>                   [49.3, 32.9, 51.0, 35.3],
>>>                   [49.2, 31.8, 51.0, 35.4],
>>>                   [35.1, 11.5, 39.1, 15.7],
>>>                   [35.6, 11.8, 39.3, 14.2],
>>>                   [35.3, 11.5, 39.9, 14.5],
>>>                   [35.2, 11.7, 39.7, 15.7]], dtype=np.float32)
>>> scores = np.array([0.9, 0.9, 0.5, 0.5, 0.5, 0.4, 0.3],                dtype=np.
↪float32)
>>> iou_threshold = 0.6
>>> dets, inds = nms(boxes, scores, iou_threshold)
>>> assert len(inds) == len(dets) == 3
```

mmcv.ops.**nms3d**(*boxes: torch.Tensor*, *scores: torch.Tensor*, *iou_threshold: float*) → torch.Tensor
3D NMS function GPU implementation (for BEV boxes).

Parameters

- **boxes** (*torch.Tensor*) – Input boxes with the shape of (N, 7) ([x, y, z, dx, dy, dz, heading]).
- **scores** (*torch.Tensor*) – Scores of boxes with the shape of (N).
- **iou_threshold** (*float*) – Overlap threshold of NMS.

Returns Indexes after NMS.

Return type torch.Tensor

mmcv.ops.**nms3d_normal**(*boxes: torch.Tensor*, *scores: torch.Tensor*, *iou_threshold: float*) → torch.Tensor
Normal 3D NMS function GPU implementation. The overlap of two boxes for IoU calculation is defined as the exact overlapping area of the two boxes WITH their yaw angle set to 0.

Parameters

- **boxes** (*torch.Tensor*) – Input boxes with shape (N, 7). ([x, y, z, dx, dy, dz, heading]).
- **scores** (*torch.Tensor*) – Scores of predicted boxes with shape (N).
- **iou_threshold** (*float*) – Overlap threshold of NMS.

Returns Remaining indices with scores in descending order.

Return type torch.Tensor

mmcv.ops.**nms_bev**(*boxes: torch.Tensor*, *scores: torch.Tensor*, *thresh: float*, *pre_max_size: Optional[int] = None*, *post_max_size: Optional[int] = None*) → torch.Tensor
NMS function GPU implementation (for BEV boxes).

The overlap of two boxes for IoU calculation is defined as the exact overlapping area of the two boxes. In this function, one can also set `pre_max_size` and `post_max_size`. :param boxes: Input boxes with the shape of (N, 5)

([x1, y1, x2, y2, ry]).

Parameters

- **scores** (*torch.Tensor*) – Scores of boxes with the shape of (N,).
- **thresh** (*float*) – Overlap threshold of NMS.
- **pre_max_size** (*int, optional*) – Max size of boxes before NMS. Default: None.

- **post_max_size** (*int, optional*) – Max size of boxes after NMS. Default: None.

Returns Indexes after NMS.

Return type torch.Tensor

mmcv.ops.**nms_match**(*dets: Union[torch.Tensor, numpy.ndarray]*, *iou_threshold: float*) → List[Union[torch.Tensor, numpy.ndarray]]

Matched dets into different groups by NMS.

NMS match is Similar to NMS but when a bbox is suppressed, nms match will record the indice of suppressed bbox and form a group with the indice of kept bbox. In each group, indice is sorted as score order.

Parameters

- **dets** (*torch.Tensor | np.ndarray*) – Det boxes with scores, shape (N, 5).
- **iou_threshold** (*float*) – IoU thresh for NMS.

Returns The outer list corresponds different matched group, the inner Tensor corresponds the indices for a group in score order.

Return type list[torch.Tensor | np.ndarray]

mmcv.ops.**nms_normal_bev**(*boxes: torch.Tensor*, *scores: torch.Tensor*, *thresh: float*) → torch.Tensor

Normal NMS function GPU implementation (for BEV boxes).

The overlap of two boxes for IoU calculation is defined as the exact overlapping area of the two boxes WITH their yaw angle set to 0. :param boxes: Input boxes with shape (N, 5)

([x1, y1, x2, y2, ry]).

Parameters

- **scores** (*torch.Tensor*) – Scores of predicted boxes with shape (N,).
- **thresh** (*float*) – Overlap threshold of NMS.

Returns Remaining indices with scores in descending order.

Return type torch.Tensor

mmcv.ops.**nms_rotated**(*dets: torch.Tensor*, *scores: torch.Tensor*, *iou_threshold: float*, *labels: Optional[torch.Tensor] = None*, *clockwise: bool = True*) → Tuple[torch.Tensor, torch.Tensor]

Performs non-maximum suppression (NMS) on the rotated boxes according to their intersection-over-union (IoU).

Rotated NMS iteratively removes lower scoring rotated boxes which have an IoU greater than iou_threshold with another (higher scoring) rotated box.

Parameters

- **dets** (*torch.Tensor*) – Rotated boxes in shape (N, 5). They are expected to be in (x_ctr, y_ctr, width, height, angle_radian) format.
- **scores** (*torch.Tensor*) – scores in shape (N,).
- **iou_threshold** (*float*) – IoU thresh for NMS.
- **labels** (*torch.Tensor, optional*) – boxes' label in shape (N,).
- **clockwise** (*bool*) – flag indicating whether the positive angular orientation is clockwise. default True. *New in version 1.4.3.*

Returns kept dets(boxes and scores) and indice, which is always the same data type as the input.

Return type tuple

mmcv.ops.**pixel_group**(*score: Union[numpy.ndarray, torch.Tensor]*, *mask: Union[numpy.ndarray, torch.Tensor]*, *embedding: Union[numpy.ndarray, torch.Tensor]*, *kernel_label: Union[numpy.ndarray, torch.Tensor]*, *kernel_contour: Union[numpy.ndarray, torch.Tensor]*, *kernel_region_num: int*, *distance_threshold: float*) → List[List[float]]

Group pixels into text instances, which is widely used text detection methods.

Parameters

- **score** (*np.array or torch.Tensor*) – The foreground score with size hxw.
- **mask** (*np.array or Tensor*) – The foreground mask with size hxw.
- **embedding** (*np.array or torch.Tensor*) – The embedding with size hxwxc to distinguish instances.
- **kernel_label** (*np.array or torch.Tensor*) – The instance kernel index with size hxw.
- **kernel_contour** (*np.array or torch.Tensor*) – The kernel contour with size hxw.
- **kernel_region_num** (*int*) – The instance kernel region number.
- **distance_threshold** (*float*) – The embedding distance threshold between kernel and pixel in one instance.

Returns The instance coordinates and attributes list. Each element consists of averaged confidence, pixel number, and coordinates (x_i, y_i for all pixels) in order.

Return type list[list[float]]

mmcv.ops.**point_sample**(*input: torch.Tensor*, *points: torch.Tensor*, *align_corners: bool = False*, ***kwargs*) → torch.Tensor

A wrapper around `grid_sample()` to support 3D point_coords tensors Unlike `torch.nn.functional.grid_sample()` it assumes point_coords to lie inside `[0, 1] x [0, 1]` square.

Parameters

- **input** (*torch.Tensor*) – Feature map, shape (N, C, H, W).
- **points** (*torch.Tensor*) – Image based absolute point coordinates (normalized), range [0, 1] x [0, 1], shape (N, P, 2) or (N, Hgrid, Wgrid, 2).
- **align_corners** (*bool, optional*) – Whether align_corners. Default: False

Returns Features of *point* on *input*, shape (N, C, P) or (N, C, Hgrid, Wgrid).

Return type torch.Tensor

mmcv.ops.**points_in_boxes_all**(*points: torch.Tensor*, *boxes: torch.Tensor*) → torch.Tensor

Find all boxes in which each point is (CUDA).

Parameters

- **points** (*torch.Tensor*) – [B, M, 3], [x, y, z] in LiDAR/DEPTH coordinate
- **boxes** (*torch.Tensor*) – [B, T, 7], num_valid_boxes <= T, [x, y, z, x_size, y_size, z_size, rz], (x, y, z) is the bottom center.

Returns Return the box indices of points with the shape of (B, M, T). Default background = 0.

Return type torch.Tensor

mmcv.ops.**points_in_boxes_cpu**(*points: torch.Tensor*, *boxes: torch.Tensor*) → torch.Tensor

Find all boxes in which each point is (CPU). The CPU version of *points_in_boxes_all()*.

Parameters

- **points** (*torch.Tensor*) – [B, M, 3], [x, y, z] in LiDAR/DEPTH coordinate
- **boxes** (*torch.Tensor*) – [B, T, 7], num_valid_boxes <= T, [x, y, z, x_size, y_size, z_size, rz], (x, y, z) is the bottom center.

Returns Return the box indices of points with the shape of (B, M, T). Default background = 0.

Return type torch.Tensor

mmcv.ops.**points_in_boxes_part**(*points: torch.Tensor*, *boxes: torch.Tensor*) → torch.Tensor

Find the box in which each point is (CUDA).

Parameters

- **points** (*torch.Tensor*) – [B, M, 3], [x, y, z] in LiDAR/DEPTH coordinate.
- **boxes** (*torch.Tensor*) – [B, T, 7], num_valid_boxes <= T, [x, y, z, x_size, y_size, z_size, rz] in LiDAR/DEPTH coordinate, (x, y, z) is the bottom center.

Returns Return the box indices of points with the shape of (B, M). Default background = -1.

Return type torch.Tensor

mmcv.ops.**points_in_polygons**(*points: torch.Tensor*, *polygons: torch.Tensor*) → torch.Tensor

Judging whether points are inside polygons, which is used in the ATSS assignment for the rotated boxes.

It should be noted that when the point is just at the polygon boundary, the judgment will be inaccurate, but the effect on assignment is limited.

Parameters

- **points** (*torch.Tensor*) – It has shape (B, 2), indicating (x, y). M means the number of predicted points.
- **polygons** (*torch.Tensor*) – It has shape (M, 8), indicating (x1, y1, x2, y2, x3, y3, x4, y4). M means the number of ground truth polygons.

Returns Return the result with the shape of (B, M), 1 indicates that the point is inside the polygon, 0 indicates that the point is outside the polygon.

Return type torch.Tensor

mmcv.ops.**rel_roi_point_to_rel_img_point**(*rois: torch.Tensor*, *rel_roi_points: torch.Tensor*, *img: Union[tuple, torch.Tensor]*, *spatial_scale: float = 1.0*) → torch.Tensor

Convert roi based relative point coordinates to image based absolute point coordinates.

Parameters

- **rois** (*torch.Tensor*) – RoIs or BBoxes, shape (N, 4) or (N, 5)
- **rel_roi_points** (*torch.Tensor*) – Point coordinates inside RoI, relative to RoI, location, range (0, 1), shape (N, P, 2)
- **img** (*tuple or torch.Tensor*) – (height, width) of image or feature map.
- **spatial_scale** (*float, optional*) – Scale points by this factor. Default: 1.

Returns Image based relative point coordinates for sampling, shape (N, P, 2).

Return type torch.Tensor

mmcv.ops.**scatter_nd**(*indices: torch.Tensor*, *updates: torch.Tensor*, *shape: torch.Tensor*) → torch.Tensor

pytorch edition of tensorflow scatter_nd.

this function don't contain except handle code. so use this carefully when indice repeats, don't support repeat add which is supported in tensorflow.

mmcv.ops.**soft_nms**(*boxes: Union[torch.Tensor, numpy.ndarray]*, *scores: Union[torch.Tensor, numpy.ndarray]*, *iou_threshold: float = 0.3*, *sigma: float = 0.5*, *min_score: float = 0.001*, *method: str = 'linear'*, *offset: int = 0*) → Tuple[Union[torch.Tensor, numpy.ndarray], Union[torch.Tensor, numpy.ndarray]]

Dispatch to only CPU Soft NMS implementations.

The input can be either a torch tensor or numpy array. The returned type will always be the same as inputs.

Parameters

- **boxes** (*torch.Tensor or np.ndarray*) – boxes in shape (N, 4).
- **scores** (*torch.Tensor or np.ndarray*) – scores in shape (N,).
- **iou_threshold** (*float*) – IoU threshold for NMS.
- **sigma** (*float*) – hyperparameter for gaussian method
- **min_score** (*float*) – score filter threshold
- **method** (*str*) – either 'linear' or 'gaussian'
- **offset** (*int, 0 or 1*) – boxes' width or height is (x2 - x1 + offset).

Returns kept dets (boxes and scores) and indice, which always have the same data type as the input.

Return type tuple

Example

```
>>> boxes = np.array([[4., 3., 5., 3.],
>>>                   [4., 3., 5., 4.],
>>>                   [3., 1., 3., 1.],
>>>                   [3., 1., 3., 1.],
>>>                   [3., 1., 3., 1.],
>>>                   [3., 1., 3., 1.]], dtype=np.float32)
>>> scores = np.array([0.9, 0.9, 0.5, 0.5, 0.4, 0.0], dtype=np.float32)
>>> iou_threshold = 0.6
>>> dets, inds = soft_nms(boxes, scores, iou_threshold, sigma=0.5)
>>> assert len(inds) == len(dets) == 5
```

mmcv.ops.**upfirdn2d**(*input: torch.Tensor*, *kernel: torch.Tensor*, *up: Union[int, tuple] = 1*, *down: Union[int, tuple] = 1*, *pad: tuple = (0, 0)*) → torch.Tensor

UpFRIDn for 2d features.

UpFIRDn is short for upsample, apply FIR filter and downsample. More details can be found in: https://www.mathworks.com/help/signal/ref/upfirdn.html

Parameters

- **input** (*torch.Tensor*) – Tensor with shape of (n, c, h, w).
- **kernel** (*torch.Tensor*) – Filter kernel.
- **up** (*int | tuple[int], optional*) – Upsampling factor. If given a number, we will use this factor for the both height and width side. Defaults to 1.
- **down** (*int | tuple[int], optional*) – Downsampling factor. If given a number, we will use this factor for the both height and width side. Defaults to 1.

- **pad** (*tuple[int], optional*) – Padding for tensors, (x_pad, y_pad) or (x_pad_0, x_pad_1, y_pad_0, y_pad_1). Defaults to (0, 0).

Returns Tensor after UpFIRDn.

Return type torch.Tensor

CHAPTER

THIRTYSIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y